Quick answers to common problems

iPhone JavaScript Cookbook

Clear and practical recipes for building web applications using JavaScript and AJAX without having to learn Objective-C or Cocoa

Arturo Fernandez Montoro

[PACKT] PUBLISHING

iPhone JavaScript Cookbook

Clear and practical recipes for building web applications using JavaScript and AJAX without having to learn Objective-C or Cocoa

Arturo Fernandez Montoro

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

iPhone JavaScript Cookbook

First published: June 2011

Production Reference: 1170611

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849691-08-6

www.packtpub.com

Cover Image by Duraid Fatouhi (duraidfatouhi@yahoo.com)

Credits

Author

Arturo Fernandez Montoro

Reviewers

Taylor Jasko

Thomas Schreiber

Acquisition Editor

Steven Wilding

Development Editor

Alina Lewis

Technical Editors

Joyslita D'Souza

Merwine Machado

Aditi Suvarna

Copy Editors

Neha Shetty

Laxmi Subramanian

Project Coordinator

Zainab Bagasrawala

Indexer

Monica Ajmera Mehta

Proofreader

Lynda Sliwoski

Graphics

Geetanjali Sawant

Production Coordinator

Shantanu Zagade

Cover Work

Shantanu Zagade

About the Author

Arturo Fernandez Montoro is a web software engineer, developer, author, and technical writer specializing in Free and Open Source Software.

His professional experience includes technologies, such as Django, Rails, J2EE, PHP, XHTML, CSS, and JavaScript, and working as a software developer and architect, project manager, sysadmin, and consultant for different companies in Europe.

Since 2002, he often writes for different Linux and Open Source printed and online magazines, such as Todo Linux, Linux+, Linux Magazine, Free Software Magazine, and Rails Magazine.

Currently, Arturo works as a Python/Django developer, contributing to one of the most important and visited websites in Spain. He can be reached at `arturo@bsnux.com`.

Many thanks to my friends and colleagues Lui Palacios and Thomas Schreiber for contributing to this book with their advice and revisions.

My wife Alicia is a living proof of the power of love. Thank you for starting a family together.

This book would have never been possible without the help and work of the team at Packt Publishing. My sincere acknowledgements to Steven, Zainab, and Alina.

Special thanks to all people who contribute to Free and Open Source with their knowledge, effort, time, patience, and enthusiasm. We're changing the world.

Thanks to my parents Jose and Aurora for teaching me to be the person I am today.

My brother Ernesto is someone who never gives up. Thank you for making my life enjoyable.

In memoriam of my grandmother Aurora, who passed away during the writing of this book.

About the Reviewers

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

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

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

He can be reached at `taylor.jasko@gmail.com`.

Thomas Schreiber was born and raised in the United States and holds a Bachelor of Arts in Digital Media from Temple University's Tyler School of Art. He is a Python web developer using Django, Pinax, and many other exciting technologies. He is also a techno musician who performed at and organized hundreds of events in Philadelphia spanning the last nine years. Thomas currently lives in Madrid with his wife Yulka, dog Finnegan, and is currently working as a lead web developer for Unidad Editorial on a social media platform. His detailed profile can be found at `http://insatsu.us/collective/thomas-schreiber/`.

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

Undoubtedly, iPhone is one of the most exciting mobile devices in the world. Its iOS is used in other Apple devices, such as iPad and iPod Touch. With this book, you'll learn how to build and develop applications for these devices without applying Apple's burdensome and, at times, restrictive technologies. Just use your experience and knowledge combined with web frontend technologies, such as JavaScript, to build quality web applications. Nobody will ever come to know that you haven't used Objective-C and Cocoa.

iPhone JavaScript Cookbook offers a set of practical and clear recipes with a step-by-step approach for building your own iPhone applications by only applying web technologies such as JavaScript and AJAX. Web developers won't need to learn a new programming language for building iOS applications with a native look and feel.

What this book covers

Chapter 1, *Frameworks Make Life Easier*, is the "getting started" chapter of this book. It covers how to install and set up different frameworks, which will be used for the recipes of the book.

Chapter 2, *Building Interfaces*, introduces you to the world of iPhone applications. You'll learn how to build essential and advanced interfaces, such as buttons, lists, forms, and date pickers.

Chapter 3, *Events and Actions*, discovers how to deal with events and actions. Both allow us a better control of the interaction between the user and the device.

Chapter 4, *A Picture Speaks a Thousand Words*, takes advantage of the great screens of iPhone and iPad teaching you how to display a grid of images, how to apply different effects, and how to work with the built-in camera of the device.

Chapter 5, *Mastering Sound and Music*, explores the audio and video capabilities of iPhone. You'll learn how to play and record audio and how to create iPod playlists.

Chapter 6, *Exchanging Data: AJAX*, covers how to use this technology for exchanging data between the server and the client. Readers are walked through the process of sending HTTP requests and processing JSON responses.

Chapter 7, *Working with Data: Storage and SQL*, provides coverage of the process for storing and retrieving data using the SQL language. Also, you'll learn how to deal with different kinds of storage available in iPhone.

Chapter 8, *This is a Phone*, enlightens that we cannot forget that iPhone is a smartphone. This is the reason to get focused on learning how to create, select and display contacts, and how to call a number and send an SMS.

Chapter 9, *Location, Location, Location*, introduces to readers to geolocation, showing how to detect the current orientation and position, and how to use the API provided by Google Maps for displaying a map at a specific location.

Chapter 10, *Web 2.0 Integration*, helps readers learn how to integrate their iPhone applications with third-party popular services such as Facebook, YouTube, Twitter, and Flickr.

What you need for this book

If you're planning to build native applications, you'll need a computer with Mac OS X, iOS SDK, and Xcode installed.

Despite the fact that it's possible to use a WebKit-compatible web browser for the recipes of this book, we suggest to use a real Apple device, such as an iPhone and iPad. However, the iPhone Simulator is a very useful tool that you can use on your Mac for testing applications.

Who this book is for

This book is for web developers interested in applying their knowledge for building web applications for iOS devices. You can develop your own iPhone web applications using nothing but JavaScript combined with XHTML and CSS. You can even give these applications a native look and feel though you wont be able to submit them to the application store. You will develop an application for iOS without having to learn the Objective-C programming language. This is the book for any iPhone developer looking to side step the totalitarian application store regime of Apple.

Conventions

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

Code words in text are shown as follows: "we'll need to add a span element for the `smallfield` label."

A block of code is set as follows:

```
<div id="leftnav">
  <ahref="#">One</a>
  <ahref="#">Two</a>
  <ahref="#">Current</a>
</div>
```

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

```
$ wget http://iui.googlecode.com/files/iui-0.31.tar.gz
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "After clicking on the **Search** button, we'll see our dialog box:"

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1
Frameworks Make Life Easier

In this chapter, we will cover:

- Why frameworks are so good
- Main frameworks for iPhone development
- Installing the iUI framework
- Installing the UiUIKit framework
- Installing the XUI framework
- Installing the *iWebKit* framework
- Installing the WebApp.Net framework
- Installing the PhoneGap framework
- Installing the Sencha Touch framework
- Installing the Apple Dashcode framework

Introduction

Many web applications implement common features independent of the final purpose for which they have been designed. Functionalities and features such as authentication, forms validation, retrieving records from a database, caching, logging, and pagination are very common in modern web applications. As a developer, surely you have implemented one or more of these features in your applications more than once. Good developers and software engineers insist on concepts, such as modularity, reusability, and encapsulation; as a consequence you can find a lot of books, papers, and articles talking about how to design your software using these techniques. In fact, modern and popular methodologies, such as *Extreme Programming*, *Scrum*, and *Test-driven Development* are based on those principles. Although this approach sounds very appealing in theory, it might be complicated to carry it out in practice.

Developing any kind of software from scratch for running in any platform is undoubtedly a hard task. Complexity grows up when the target platform, operating system, or machine has its own specific rules and mechanisms. Some tools can make our job less complicated but only one kind of them is definitely a safe bet. It is here when we meet frameworks, a set of proven code that offers common functionality and standard structures for software development. This code makes our life much easier without reinventing the wheel and gives a skeleton to our applications, making sure that we're doing things correctly. In addition, frameworks avoid starting from scratch once more. From a technical point of view, most frameworks are a set of libraries implementing functions, classes, and methods.

Using frameworks, we can save time and money, writing less code due to its code skeleton, and features implemented on it. Usually, frameworks force us to follow standards and they offer well-proven code avoiding common mistakes for beginners. Tasks such as testing, maintenance, and deployment are easier to do using frameworks due to the tools and mechanisms included. On the other hand, the learning curve could be a big and difficult drawback for beginners.

Through this chapter, we'll learn how to install the main frameworks for JavaScript, HTML, and CSS development for iPhone. All of them offer a base to develop applications with a consistent and native look and feel using different methods. While some of them are focused on the user interface, others allow using AJAX in an efficient and easy way. Even some frameworks allow building native applications from the original code of the web application. We have the chance to choose which is better to fulfill our requirements; it is even possible to use more than one of these solutions for the same application. For our recipes, we'll use the following frameworks:

- **iUI**: This is focused on the look and feel of iPhone and consists of CSS files, images, and a small JavaScript library. Its objective is to get a web application running on the device with a consistent interface such as a native application. This framework establishes a correspondence between HTML tags and conventions used for developing native applications.
- **UiUIKit**: Using a set of CSS and image files, it provides a coherent system for building web applications with a graphic interface such as native iPhone applications. The features offered for this framework are very similar to iUI.
- **XUI**: This is a pure JavaScript library specific for mobile development. It has been designed to be faster and lighter than other similar libraries, such as jQuery, MooTools, and prototype.
- **iWebKit**: This is developed specifically for Apple's devices and is compatible with CSS3 standard; it helps to write web applications or websites with minimum HTML knowledge. Its modular design supports plugins for adding new features and we can build and use the themes for UI customization.

- **WebApp.Net**: This framework comes loaded with JavaScript, CSS, and image files for developing web application for mobile devices that uses WebKit engine in its web browsers. Besides building interfaces, this framework includes functionality to use AJAX in an easy and efficient way.
- **PhoneGap**: This is designed to minimize efforts for developing native mobile applications for different operating systems, platforms, and devices. It is based on the **WORE** (**Write once, run anywhere**) principle and it allows conversion from a web application into a native application. It supports many platforms and operating systems, such as *iOS*, Android, webOS, Symbian, and BlackBerry OS.
- **Sencha Touch**: This is a complete mobile web framework based on HTML5, JavaScript, and CSS standards for developing Android and *iOS*-based applications. It has been developed by Sencha—the owner company of the popular JavaScript framework Ext JS.
- **Apple Dashcode**: Formally, this is a software development tool for Mac OS X included in Leopard and Snow Leopard versions, and focused on widget development for these operating systems. However, the last versions allow you to write web applications for iPhone and other *iOS* devices offering a graphic interface builder.

Installing the iUI framework

This recipe shows how to download and install the iUI framework on different operating systems. Particularly, we'll cover Microsoft Windows, Mac OS X, and GNU/Linux.

Getting ready

The first step is to install and get ready; some tools need to be downloaded and decompressed. As computer users, we know how to decompress files using software such as WinZip, Ark, or the built-in utility on Mac OS X. You will surely have installed a web browser on your computer. If you are a Linux or Mac developer, you already know how to use *curl* or *wget*. These tools are very useful for quick download and you only need to use the command line through applications such as GNOME Terminal, Konsole, iTerm, or Terminal. iUI is an open source project, so you can download the code for free.

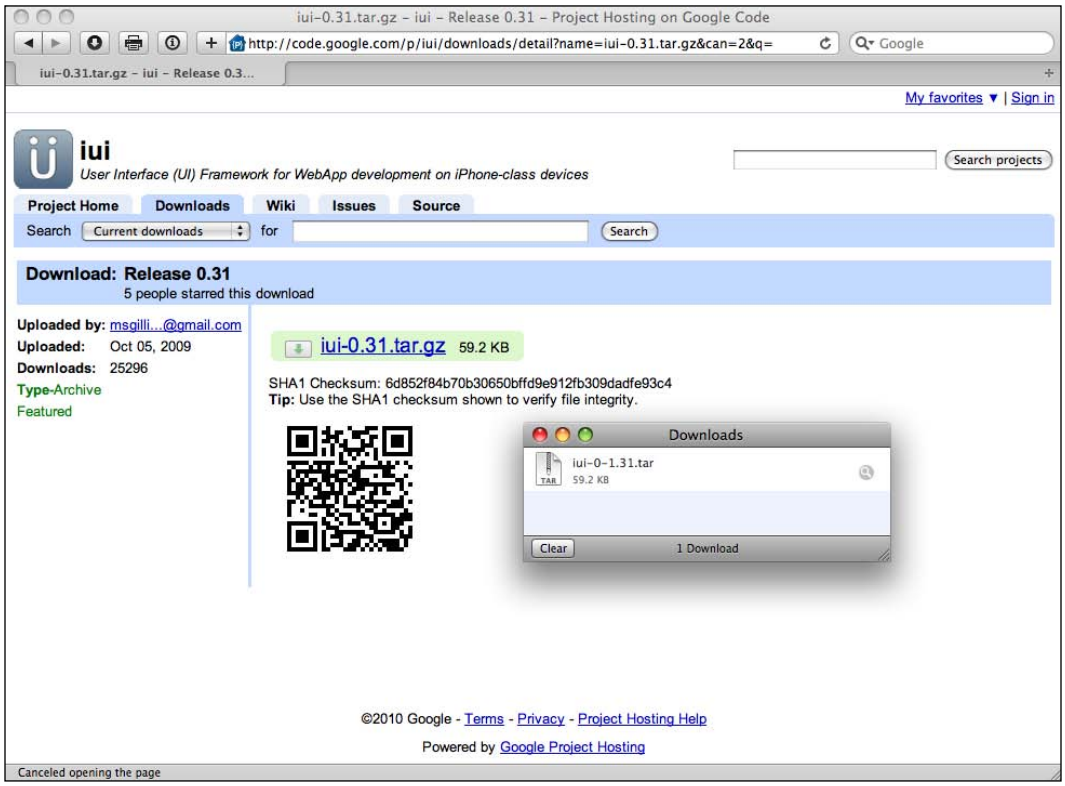
The open source project releases some stable versions packed and ready to download, but it is also possible to download a development version. This one could be suitable if you prefer working with the latest changes made by the official developers contributing to the project. Due to this, developers are using Mercurial version control and thus we'll need to install a client for it to get access to this code.

How to do it...

iUI is an open source project so you can download the code for free. Open your favorite web browser and enter this URL:

```
http://code.google.com/p/iui/downloads/list
```

In that web page, you'll see a list with files that refer to different release versions of this framework. Clicking on the link corresponding to the latest release's drives takes you to a new web page that shows you a new link for the file. Click on it for instant downloading.



If you are a GNU/Linux user or a Mac developer you will be used to command line. Open your terminal application and launch this command from your desired directory:

```
$ wget http://iui.googlecode.com/files/iui-0.31.tar.gz
```

Once you have downloaded the tarball file, it's time to extract its content to a specific folder on our computer. WinZip and WinRAR are the most popular tools to do this task on Windows. Linux distributions, by default, install similar tools such as File Roller and Ark. Double-clicking from the download window of the Safari browser will extract the files directly to your default folder on your Mac, which is usually called `Downloads`. For command-line enthusiasts, execute the following command:

```
$ tar -zxvf iui-0.31.tar.gz
```

How it works...

After decompressing the downloaded file, you'll find a folder with different subfolders and files. The most important is a subfolder called `iui` that contains CSS, images, and JavaScript files for building our web applications for iPhone. We need to copy this subfolder to our working folder where other application files reside.

Sharing this framework across different web applications is possible; you only need to put the iUI at a place where these applications have permissions to access. Usually, this place is a folder under the *DocumentRoot* of your web server. If you're planning to write a high load application, it would be a good idea to use a cloud or **CDN** (**Content Delivery Network**) service such as Amazon Simple Storage Services (Amazon S3) for hosting and serving static HTML, CSS, JavaScript, and image files.

Installing the iUI framework is a straightforward process. You simply download and decompress one file, and then copy one folder into an other, which has permission to be accessed by the web server.

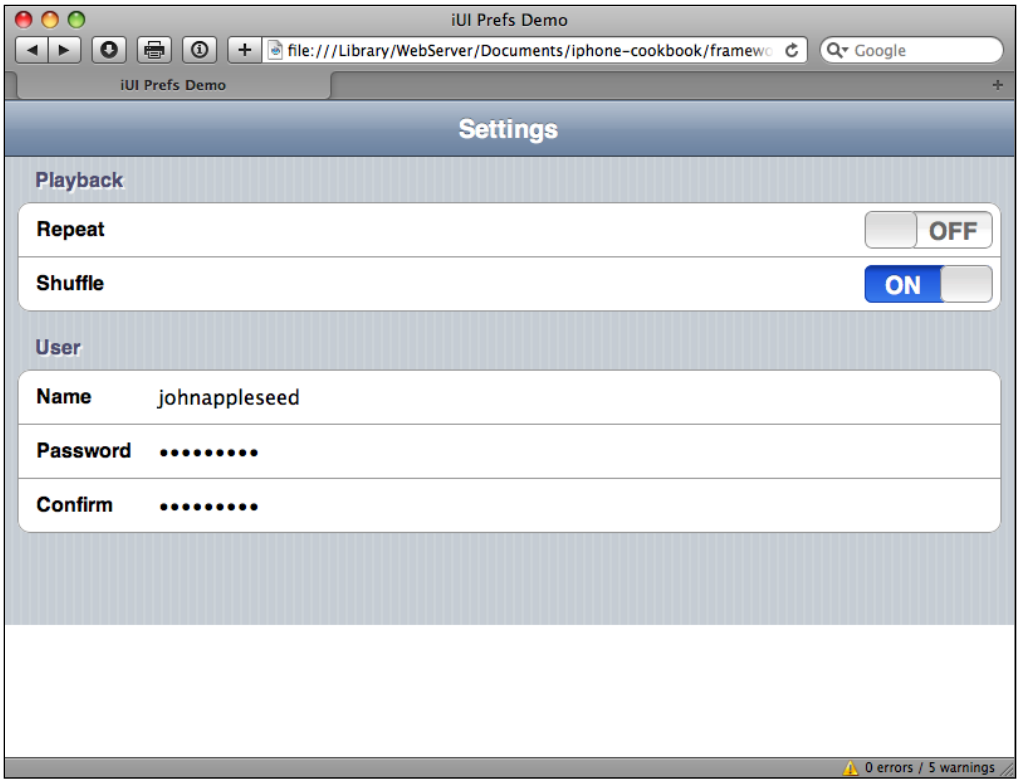
Apache is one of the most used and extended web servers in the world. Other popular options are Internet Information Server (IIS), lighttpd, and nginx. Apache web server is installed by default on Mac OS X; most of the operating systems based on Linux and UNIX offer binary packages for easy installation and you can find binary files for installing on Windows as well. IIS was designed for Windows operating systems, meanwhile, lighttpd and nginx are winning popularity and are used on UNIX systems as Linux's distros, FreeBSD, and OpenBSD. Ubuntu Linux uses `/var/www/` directory as the main *DocumentRoot* for Apache. So, in order to share iUI framework across applications, you can copy the folder to the other folder by executing this command:

```
$ cp -r iui-0.31/ui /var/www/iui
```

If you are a Mac user, your target directory will be `/Library/WebServer/Documents/iui`.

There's more...

Inside the samples subfolder, you'll find some files showing capabilities of this framework, including HTML and PHP files. Some examples need a web server with PHP support but you can test others using Safari web browser or an other WebKit's browser such as Safari or Google Chrome. Open `index.html` with a web browser and use it as your starting point.



If you prefer to use the latest version in development from the version control, you'll need to install a Mercurial client. Most of the GNU/Linux distribution such as Fedora, Debian, and Ubuntu includes binary packages ready to install them. Usually, the name of the binary package is mercurial. The following command will install the client on Ubuntu Linux:

```
$ sudo apt-get install mercurial
```

Mercurial is an open source project and offers a binary file ready to install for Mac OS X and Windows systems. If you're using one of these, go to the following page and download the specific file for your operating system and version:

`http://mercurial.selenic.com/downloads/`

After downloading, you can install the client using the regular process for your operating system. Mac users will find a ZIP file containing a binary package. For Windows, the distributed file is a MSI (Microsoft Installer), ready for self-installation after clicking on it.

Despite that the client of this version control was developed for the command line, we can find some GUI tools online such as TortoiseHG for Windows. These tools are intuitive and user-friendly, allowing an interactive use between the user and the source files hosted in the version control system. TortoiseHG can be downloaded from the same web page as the Mercurial client.

Finally, we'll download the version development of the iUI framework executing the following command:

```
$ hg clone https://iui.googlecode.com/hg/ iui
```

The new `iui` folder includes all files of the iUI framework. We should copy this folder to our *DocumentRoot*.

If you want to know more about this framework, point your browser at the official wiki project:

`http://code.google.com/p/iui/w/list`

Also, taking a look at the complete code of the project may be interesting for advanced developers or just for people wanting to learn more about internal details:

`http://code.google.com/p/iui/source/browse`

Installing the UiUIKit framework

UiUIKit is the short name of the **Universal iPhone UI Kit** framework. The development of this framework is carried out through an open source project hosted in *Google Code* and is distributed under the GNU Public License v3. Let's see how to install it on different operating systems.

Getting ready

As the main project file is distributed as a ZIP file, we'll need to use one tool for decompressing these kind of files. Most of the modern operating systems include tools for this process. As seen in the previous recipe, we can use *wget* or *curl* programs for downloading the files.

If you are planning to read the source code or you'd like to use the current development version of the framework, you'll need a Subversion client as the UiUIKit project is working with this open source version control.

How to do it...

Open your web browser and type the following URL:

`http://code.google.com/p/iphone-universal/downloads/list`

After downloading, click on the link for the latest version from the main list, for instance, the link called `UiUIKit-2.1.zip`. The next page will show you a different link for this file that represents the version 2.1 of the UiUIKit framework. Click on the link and the file will start downloading immediately.

Mac users will see how the Safari browser shows a window with the content of the compressed file, which is a folder called `UiUIKit`, which is stored in the default folder for downloads.

Command line's fans can use these simple commands from their favorite terminal tool:

```
$ cd ~
$ curl -O http://iphone-universal.googlecode.com/files/UiUIKit-2.1.zip
```

After downloading, don't forget to decompress the file on your web-specific directory. The commands given next execute this action on Linux and Mac OS X systems:

```
$ cd /var/www/
$ unzip ~/UiUIKit-2.1.zip
```

How it works...

The main folder of the UiUIKit framework contains two subfolders called `images` and `stylesheets`. The first one includes many images used to get a native look for web applications on the iPhone. The other one offers a CSS file called `iphone.css`. We only need the `images` subfolder with its graphic files and the CSS file.

In order to use this framework in our projects, we need to allow our HTML files access to the images and the CSS file of the framework. These files should be in a folder with permissions for the web server. For example, we'll have a directory structure for our new web application for iPhone as follows:

```
myapp/
  index.html
  images/
    actionButtons.png
    apple-touch-icon.png
    backButton.png

  toolButton.png
  whiteButton.png
```

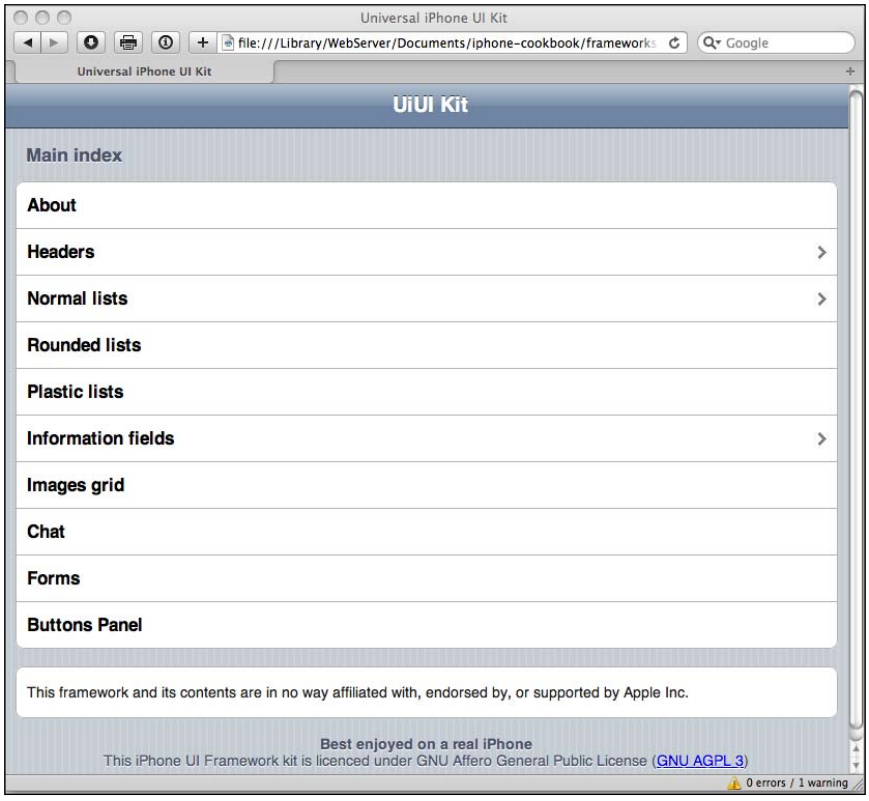
```
first.html
second.html
stylesheets/
  iphone.css
```

Remember that this framework doesn't include HTML files; we only need a bunch of the graphic files and one stylesheet file. The HTML files showed in the previous example will be our own files created for the web application.

We'll also find a lot of examples on different HTML files located in the root directory, outside the mentioned subfolders. These files are not required for development but they can be very useful to show how to use some features and functionalities.

There's more...

For an initial contact with the capabilities of the framework it would be interesting to take a look at the examples included in the main directory of the framework. We can load the `index.html` in our browser. This file is an index to the different examples and offers a native interface for the iPhone. Safari could be used but is better to access from a real iPhone device.



Subversion is a well-proven version control used by many developers, companies, and, of course, open source projects. UiUIKit is an example of these projects using this popular version control. So, to access the latest version in development, we'll need a client to download it. Popular Linux distributions, including Ubuntu and Debian have binary packages ready to install. For instance, the following command is enough to install it on Ubuntu Linux:

```
$ sudo apt-get install subversion
```

The last versions of Mac OS X, including Leopard and Snow Leopard, includes a Subversion client ready to use. For Windows, you can download Slik SVN available for 32-bit and 64-bits platforms; installation programs can be downloaded from: `http://www.sliksvn.com/en/download`.

When you are sure that your client is running, you could execute it for getting the latest development version of the UiUIKit framework. Mac and Linux users will execute the following command:

```
$ svn checkout http://iphone-universal.googlecode.com/svn/trunk/ UiUIKit
```

All information related to the UiUIKit framework project could be found at: `http://code.google.com/p/iphone-universal/`

Installing the XUI framework

Frameworks and libraries such as jQuery, prototype, MooTools, YUI, and Dojo are becoming very popular among web developers. All of them present an easy way for using the DOM model of HTML in addition to the AJAX capabilities and other interesting features such as animations, including support for multiple browsers avoiding the complexity of cross-browser applications. It sounds pretty good but the problem is that they aren't focused on mobile devices. To solve this problem we're introducing XUI, a simple but powerful and lightweight JavaScript framework.

Getting ready

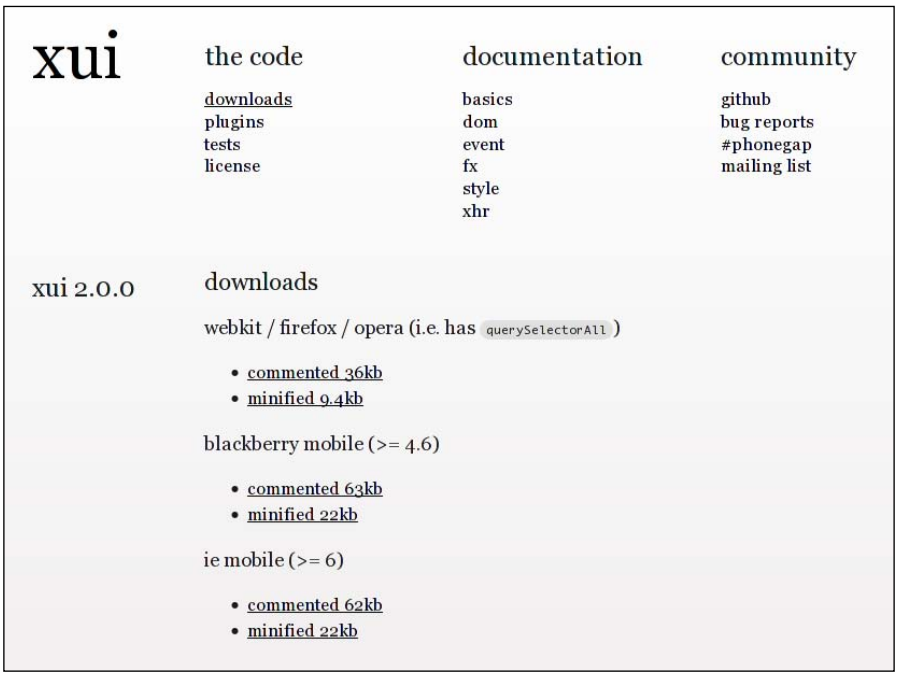
XUI is open source and can be downloaded from the main page of its website (`http://xuijs.com/`). As seen in the previous recipes, you will need a web browser or a command-line utility for downloading. Developers or people interested in development versions will need a client for it, the version control system used by the XUI open source project.

How to do it...

Open your web browser and type this URL: `http://xuijs.com/downloads/`.

In the list, we'll find different versions related to the general mobile devices and specific for BlackBerry and Windows Mobile operating systems. Each version has two different files, one minified and the other commented. The first one should be used for production environments. We'll download the commented version for general mobile devices, marked as `webkit/firefox/opera`. After clicking on the specified link, you'll have a new file called `xui-2.0.0.js`. If you prefer to use command line, you can execute the following command:

```
$ wget http://xuijs.com/downloads/xui-2.0.0.js
```



The last step will be to copy these files to a folder inside our `DocumentRoot` directory. For example, on Fedora Linux:

```
$ sudo mkdir /var/www/html/xui
$ sudo cp xui-2.0.0.js /var/www/html/xui/
```

How it works...

XUI is a pure JavaScript framework integrated only by files written in this programming language. In order to use this lightweight and fast framework, we only need to copy the two JavaScript files into a folder with permissions for our web server, and then include a reference on the HTML files of our web application for iPhone.

Inside the head section of our HTML files, we'll need to add this line:

```
<script type="text/javascript" src="xui/xui-2.0.0.js"></script>
```

A complete reference and documentation for the XUI framework can be found at: `http://xuijs.com/docs`.

All the source code for the XUI can be found at `http://github.com/xui/xui`.

Installing the iWebKit framework

The *iWebKit* is a framework focused on being fast, lightweight, and specifically for developing web applications and websites for Apple's devices. Installation is a straightforward process, as we'll see in this recipe.

Getting ready

This framework can be downloaded for free from its main website. As it is distributed as a ZIP file, we'll need a web browser or command-line utility, as seen in the previous recipes, and a tool for decompressing.

How to do it...

Point your browser at the following URL and download the file through the **Download** button:

`http://snippetspace.com/projects/iwebkit/`

When the download is finished, decompress the file:

```
$ unzip iWebKit5.04.zip
```

Finally, copy the JavaScript, CSS, and graphic files to a folder under `DocumentRoot`. For example, on Ubuntu Linux, we'll execute the following commands:

```
$ mkdir /var/www/iWebKit
$ cd iWebKit5.04/Framework
$ cp -r css /var/www/iWebKit/
$ cp -r images /var/www/iWebKit/
$ cp -r javascript /var/www/iWebKit/
```

Our *iWebKit* copy is ready to use for iPhone web application development.

How it works...

After decompressing the main ZIP file, you'll see different files and folders. Most important is the `Framework` folder that contains the required JavaScript, graphics, and stylesheet files for developing our own applications. *iWebKit* also includes some PDF files, one of them is a practical user's guide showing how to use the framework through many examples. The last folder included in the ZIP file is called `iWebKit demo` and it contains examples, including some PHP files for showing how to establish a communication between the client and the server side of web applications.

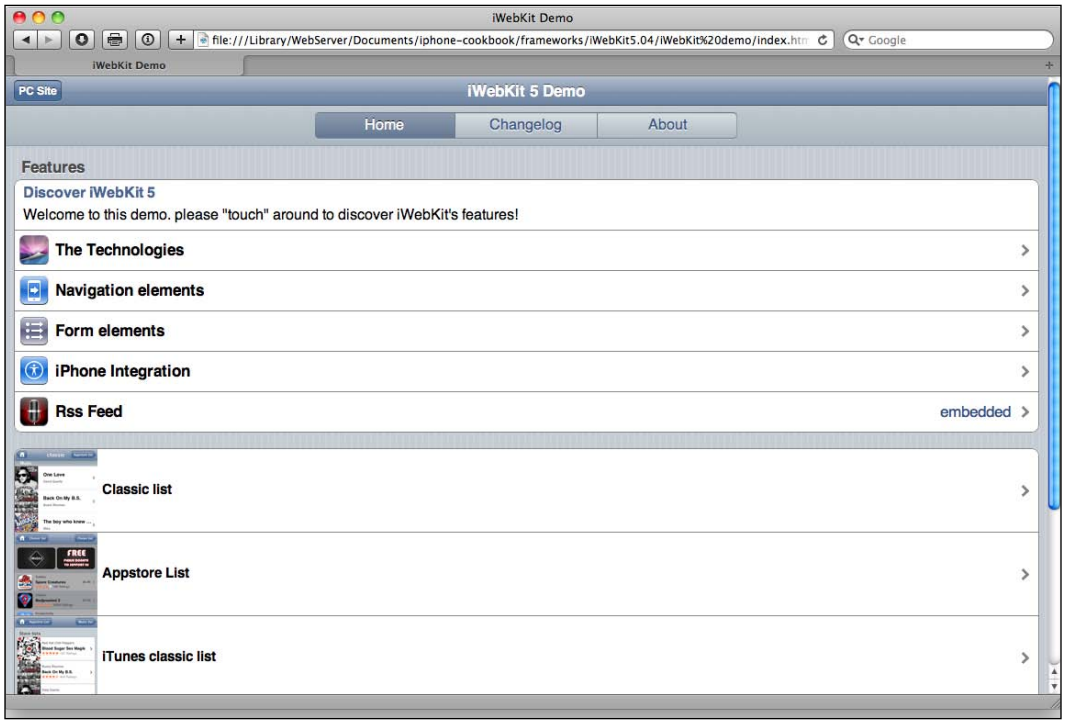
The HTML files of our applications developed using this framework should include the following lines in the head section:

```
<link href="/iWebKit/css/style.css" rel="stylesheet" media="screen"
type="text/css" />
"<script src="/iWebKit/javascript/functions.js" type="text/
javascript"></script>"
```

By observing the preceding lines, we guess `style.css` is the main stylesheet and `functions.js` is the file that has the main JavaScript code used for the framework. Probably, we'll use some images as well. In this case, don't forget to use a reference to the `images` folder.

There's more...

The `Userguide.pdf` is a small tutorial introducing the *iWebKit* framework and it is really useful for an initial contact. This document is focused on practical issues, avoiding internal details. Another good starting point is the included examples; load the `index.html` file on your browser or on your Apple device. This file allows access to different examples showing some features and functionalities.



Before starting the development using this framework, you can take a look at some of the websites and applications developed by others. iWebKit's website has a special page showing these examples:

`http://snippetspace.com/projects/iwebkit/demo/`

Forums allow the developers and users to exchange questions, answers, and knowledge. *iWebkit* has its own forum open to everyone; you only need to register to send your own questions and answers. This interesting forum can be reached at:

`http://snippetspace.com/forum/`

Installing the WebApp.Net framework

This JavaScript framework is not specific for iPhone and other Apple devices. It was designed for different modern mobile devices focused on AJAX functionality, offering functions to avoid the complexity of developing this technology from scratch. In fact, it is very useful due to differences between web browsers of the mobile platforms and operating systems. This recipe explains how to install the WebApp.Net on Linux, Windows, and Mac OS X operating systems.

Getting ready

WebApp.Net is a set of HTML, JavaScript, and CSS files packed and distributed as a ZIP file and can be downloaded for free. We'll need a web browser to download this file and a tool to decompress it.

How to do it...

Type the following URL on your favorite web browser and click on the link located on the right (get it now: v0.5.2): `http://webapp-net.com/`.

When the file has been downloaded, it can be decompressed using a tool such as WinZip on Windows. Safari users will find a new folder called `base-package-v0` on the default folder for downloads. Linux users can execute the following command from a terminal tool:

```
$ unzip base-package-v0.5.2-20100206.zip
```

After decompressing, you will need to copy the required files of the framework to our folder under control of the web server. For instance, you will execute these commands on Mac:

```
$ mkdir /Library/WebServer/Documents/WebAppNet
$ cp -r base-package-v0/WebApp /Library/WebServer/Documents/WebAppNet
```

How it works...

The main ZIP file of the framework contains many files but the most important is the `WebApp` folder that will be needed for developing our own web applications for the Apple devices. In order to use the WebApp.Net framework, at a minimum we will need two files that should be included on the head section of our HTML pages. Actually, the following lines are required:

```
<link rel="stylesheet" href="/WebAppNet/Design/Render.css" />
<script type="text/javascript" src="/WebAppNet/Action/Logic.js"></
script>
```

Inside the `WebApp` folder, we can find different subfolders such as `Design`, `Action`, and `Img`. The first one of them stores two CSS files and some graphic images required by these stylesheets. Action subfolders contain the main JavaScript file called `Logic.js` and the last subfolder is `Img`, responsible for the main graphic files needed to get a native look and feel for the applications.

There's more...

As a part of the commented WebApp main folder, this framework contains other folders inside the ZIP file. These subfolders are `Debug` and `Tools`. The first one offers the JavaScript and CSS files without minimizing them, which is useful for development environments. They are more human readable than the others included on the `WebApp` main folder. The `Tools` subfolder has one PHP file for allowing us to use a proxy with the XML data. In addition, two stylesheets are also included. One of them is specific for the Mozilla Firefox web browser and can be used for testing and developing our web applications.

WebApp.Net has a complete documentation for users and developers that can be found at: `http://webapp-net.com/Doc/`.

Same case as *iWebKit*, you can find a complete and useful forum for WebApp. Net at: `http://webapp-net.com/Forums/`.

Installing the PhoneGap framework

PhoneGap is a web framework designed to build cross-platform for mobile devices. The framework includes a set of tools to deploy your applications on different mobile operating systems without changing the original code written using HTML, CSS, and JavaScript. We'll cover how to install this framework on Mac to be used for iPhone development.

Please keep in mind that the version of PhoneGap for *iOS* development only works on a machine with a recent version of Mac OS X installed, such as *Leopard* or *Snow Leopard*.

Getting ready

We'll need the Xcode tool for building applications for Apple devices using the PhoneGap framework. This tool is an **IDE** (**Integrated Development Environment**) designed for Mac OS X. It can be downloaded for free and it's included on the latest releases of the Apple's operating system including Leopard and Snow Leopard.

In addition to Xcode, you will need to install the **SDK** (**Software Development Kit**) for iPhone, also knows as *iOS* SDK. Apple distributes this software for free; you only need to register as a developer. This process is fast and you can use your own Apple's ID.

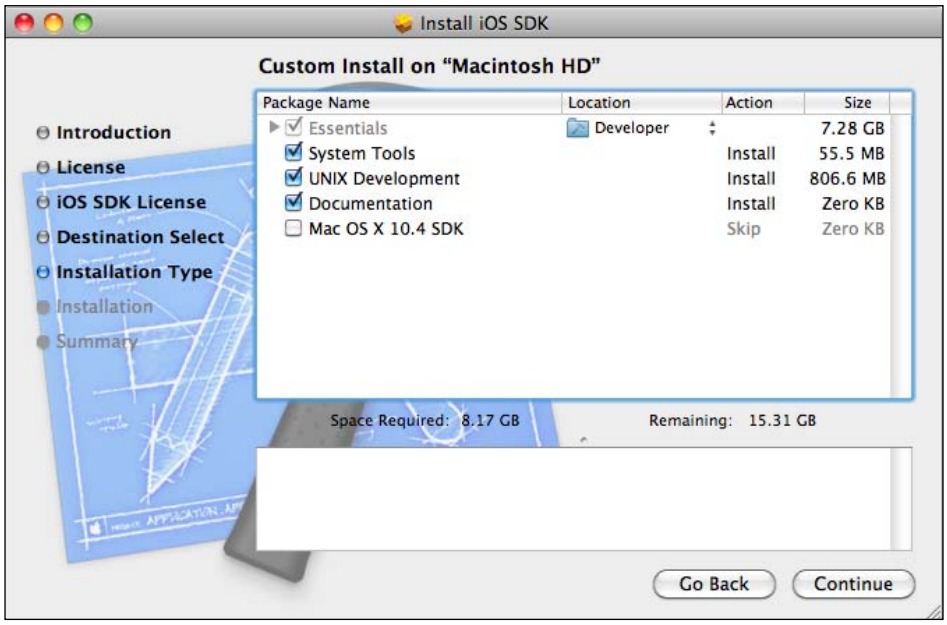
How to do it...

The first step will be to install Xcode and the *iOS* SDK. Apple distributes these tools in one DMG file ready for downloading and installing. For registering as an Apple Developer, you should open your browser and type the following URL:

```
http://developer.apple.com/programs/register/
```

Follow the instructions on the screen and then at the end of the process, you can access the specific link for downloading this SDK. This link appears as *iOS* SDK 4.1; it is located at `http://developer.apple.com/devcenter/ios/`. The next page will show you a list with different links. If you click on the **Downloads** link, it drives you to the other link specific for the latest version of the SDK and Xcode. After clicking, the file will be downloaded to your `Downloads` folder.

When you have the DMG file on your computer, click on it and the installation process will be launched. Note that this process will take a long time because the size of the file is over 3.5 GB. Follow all the instructions on the screen. When the process ends, you will be able to access the applications for launching Xcode. The SDK will be auto configured and ready to use with the installed IDE.



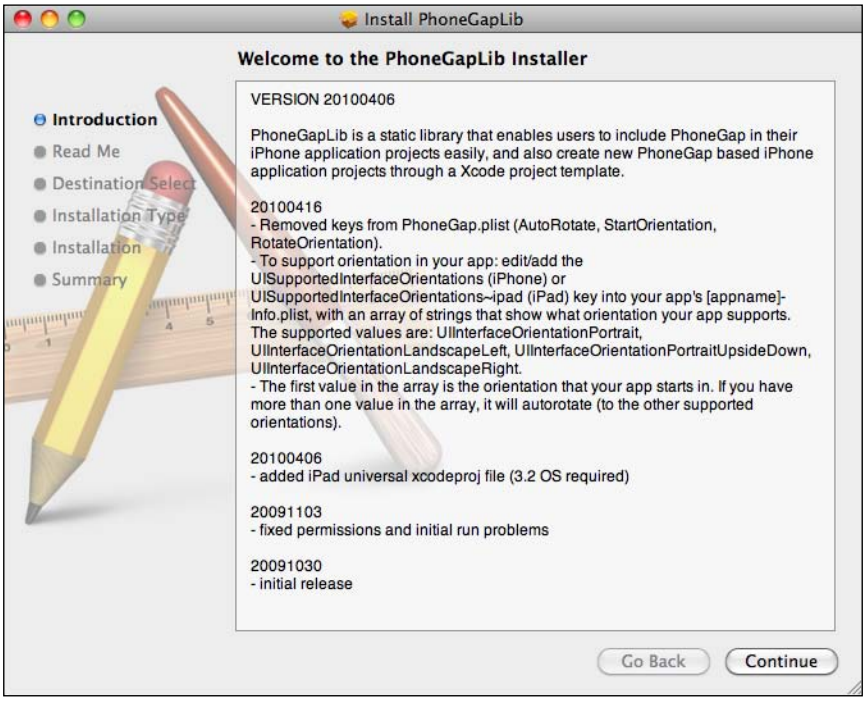
Now, we are going to download the PhoneGap framework. It's distributed as a compressed file in ZIP and tarball format. We can download it using our browser and loading the URL `http://www.phonegap.com/download`. You'll find a button to download it. After downloading, you should decompress the file into your `DocumentRoot` folder:

```
$ cd /Library/WebServer/Documents/
$ unzip ~/Downloads/phonegap-0.9.4.zip
```

It's time to link PhoneGap to Xcode. For this process, we'll need to use the command line through the *Terminal* application. Open this application and execute the following commands:

```
$ cd /Library/WebServer/Documents/phonegap-0.9.4/phonegap-iphone
$ make
```

The `make` command will generate a binary package (`PhoneGapLibInstaller.pkg`) for installing PhoneGap on your computer. To start the installation process, open the Finder and click on this new generated file. Follow the simple instructions onscreen. At the end of this process, you'll be ready to use Xcode to build applications using the PhoneGap framework.

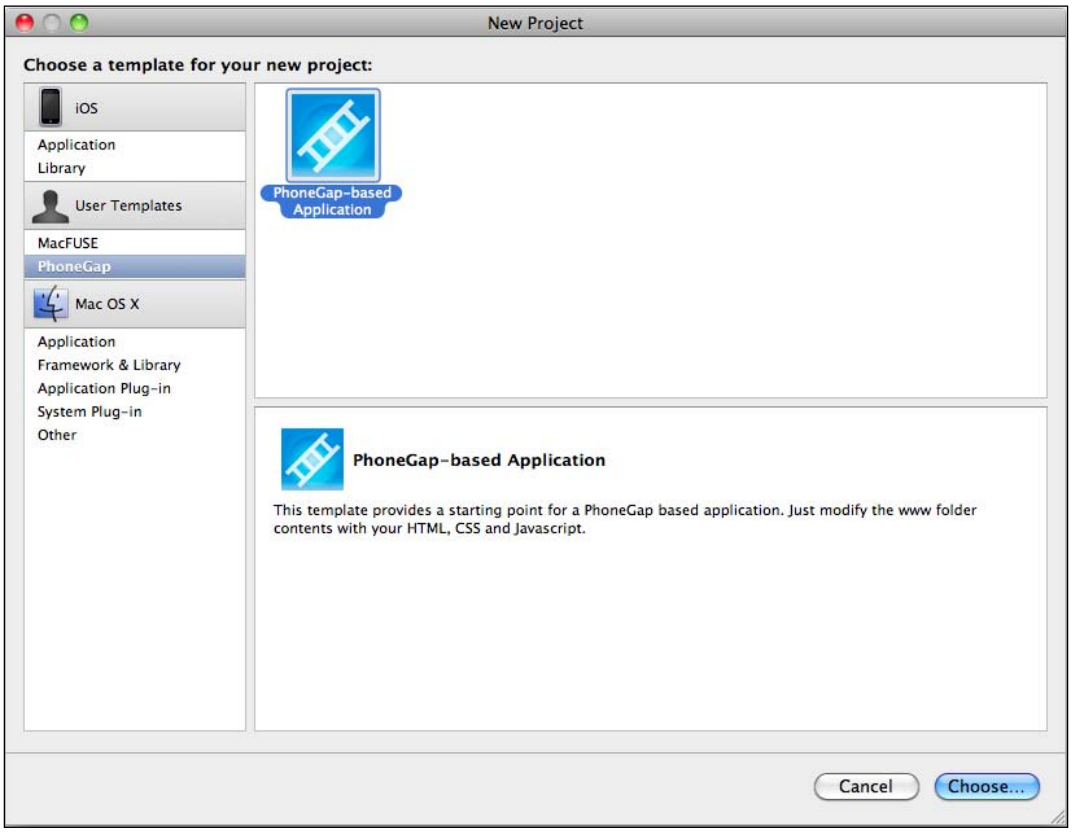


How it works...

PhoneGap uses different tools to create native applications for different mobile platforms. This is the reason to include specific folders for each operating system supported by the framework. We are focused on Apple's devices, so the most important folder for us is the `phonegap-iphone`. This folder was used to build the required tools to use PhoneGap with Xcode and the *iOS* SDK.

Don't forget that PhoneGap was designed to build native applications. In fact, we'll need a Mac Intel-based computer with Snow Leopard and the commented tools installed.

PhoneGap installs a specific template for Xcode ready to create new projects using this template. When you create a new project (`File | New project`), a dialog box shows you a user template called *PhoneGap*. This template generates the skeleton for a PhoneGap application using a Xcode project file. After the new project is created, you can see a folder called `www`. This is the place to put your application files. Remember, we're using web technologies to create native applications.



Inside the `www` folder, a main file called `index.html` is created by default. This file contains the minimum code for a PhoneGap-based application. The head section includes a reference to a main JavaScript file called `phonegap.js`. This file was created during the installation process of the binary package.

For testing our PhoneGap applications, we can use the iPhone simulator—a tool included in the *iOS* SDK. The **Build and Run** button of the IDE will invoke directly to run our applications on this simulator. Before this step, we need to choose **Simulator-4.1** as the platform target on the configuration of Xcode for our development project.

The iPhone simulator can be launched from the `Application` folder as an independent application. Actually, we can use this tool to test our application instead of a real device. Any kind of web application can be tested from the simulator, not only the PhoneGap-based application.

Keep in mind, we'll need to be registered *iOS* developers to install our native applications into our physical devices. The registration process is not free; you need to pay an annual fee to Apple. For more information about the conditions, requirements, and related information about the *iOS Developer Program*, take a look at `https://developer.apple.com/programs/ios/`.

A complete API documentation with useful examples can be found at `http://docs.phonegap.com`.

The wiki site of the PhoneGap project is another resource for documentation and can be found at `http://phonegap.pbworks.com/`.

The PhoneGap project maintains a repository with applications developed by many authors. It could be interesting to take a look at it: `http://www.phonegap.com/apps/`. Maybe you can consider the idea of uploading your own!

If you are interested in contributing to the PhoneGap open source project, you can start reading the contributor agreement located at `http://www.phonegap.com/contributor-agreement/`.

Installing the Sencha Touch framework

This framework allows to build web applications for *iOS* and Android based on devices with a native look and feel. It uses the new HTML5 standard, CSS3, and techniques such as AJAX, DHTML, and DOM scripting. By reading through this recipe, you'll learn how to install it.

Getting ready

Sencha Touch is distributed based on a dual licensing model. If you're planning to develop an open source application with a license compatible with the *GNU GPL License v3*, you can download it for free. Otherwise, you must pay for using a commercial license. Prices and information about commercial licenses of this framework can be found at `http://www.sencha.com/store/touch/`.

How to do it...

Open your browser and type the following URL:

`http://www.sencha.com/products/touch/`

After loading, you'll see a **Download** button at the top of the web page. This button allows to download a ZIP file corresponding to the latest released version of the framework. You can find two different versions for *Sencha Touch*, one for the Open Source version and the other for the Commercial Upgrade. For the recipes of this book, we'll be using the Open Source version. If you need to use the command line to decompress the ZIP file, you can execute a command as follows:

```
$ unzip sencha-touch-1.0.1a.zip
```

Now it's time to copy only the required files to our web server. For instance, to do that on Ubuntu Linux, you should execute the following commands:

```
$ mkdir /var/www/sencha-touch
$ cp sencha-touch-1.0.1a/*.js /var/www/sencha-touch/
$ cp –r sencha-touch-1.0.1a/resources/ /var/www/sencha-touch/
$ cp –r sencha-touch-1.0.1a/src/ /var/www/sencha-touch/
```

How it works...

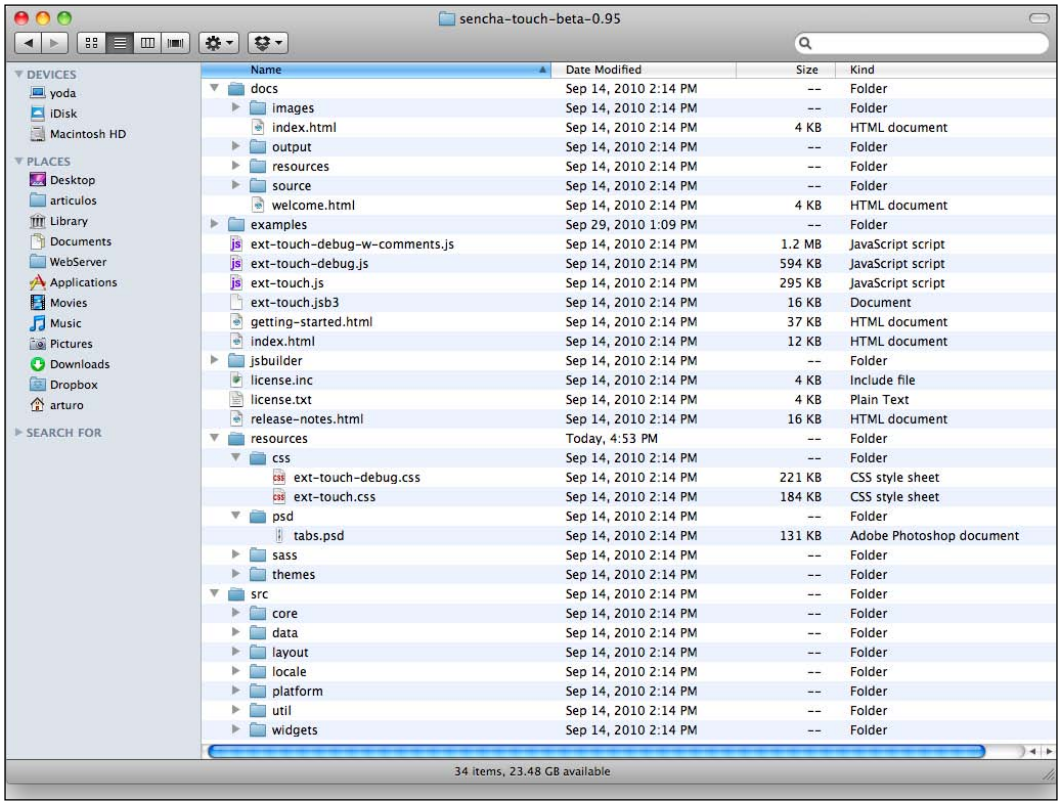
In order to use the Sencha Touch in your web projects for iPhone, you'll need to use two files at least. The two files are `ext-touch.css` and `ext-touch.js`. The first one is located inside the `css` subfolder, which is inside the `resources` folder. The other one can be found in the root directory. Your HTML files should contain these lines inside the head section for loading the required files:

```
<link rel="stylesheet" href="/sencha-touch/resources/css/ext-touch.
css" type="text/css" />
<script type="text/javascript" src="/sencha-touch/ext-touch.js"></
script>
```

Sencha Touch requires an additional JavaScript file for your application. This file will contain the needed code for creating the user interface. For example, if this file is called `main.js`, the reference on the HTML files will be a line as follows:

```
<script type="text/javascript" src="main.js" type="text/javascript"></
script>
```

The developers of this framework recommend using the `ext-touch.js` JavaScript file only for the production environment. Instead of this file, we can use the `ext-touch-debub.js` file, which is located in the same root folder.



The examples folder contains some good examples to learn how to use this framework. You can load the `index.html` file inside the `examples` folder on your WebKit-compatible browser or, of course, directly on the iPhone.

A complete API can be found at `http://dev.sencha.com/deploy/touch/docs/`.

Installing the Apple Dashcode framework

From the technical point of view, Dashcode is not a framework. However, it provides a tool with a set of components and utilities designed for building *iOS* web applications. This is the reason for talking about it in this chapter.

In this recipe, we'll see how to install this development tool designed by Apple.

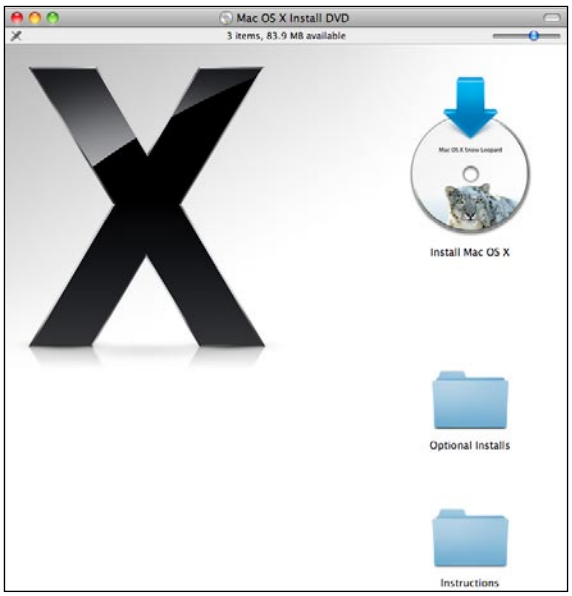
Getting ready

Apple Dashcode is a software tool for development designed for running on Mac OS X. We'll need a recent version of this operating system such as *Leopard* or *Snow Leopard*. It is not possible to install this software on Windows or Linux systems.

How to do it...

Dashcode is a part of the Xcode's development tools included on the Snow Leopard DVD as an optional install. The first step will be to insert the mentioned DVD on the drive of the computer.

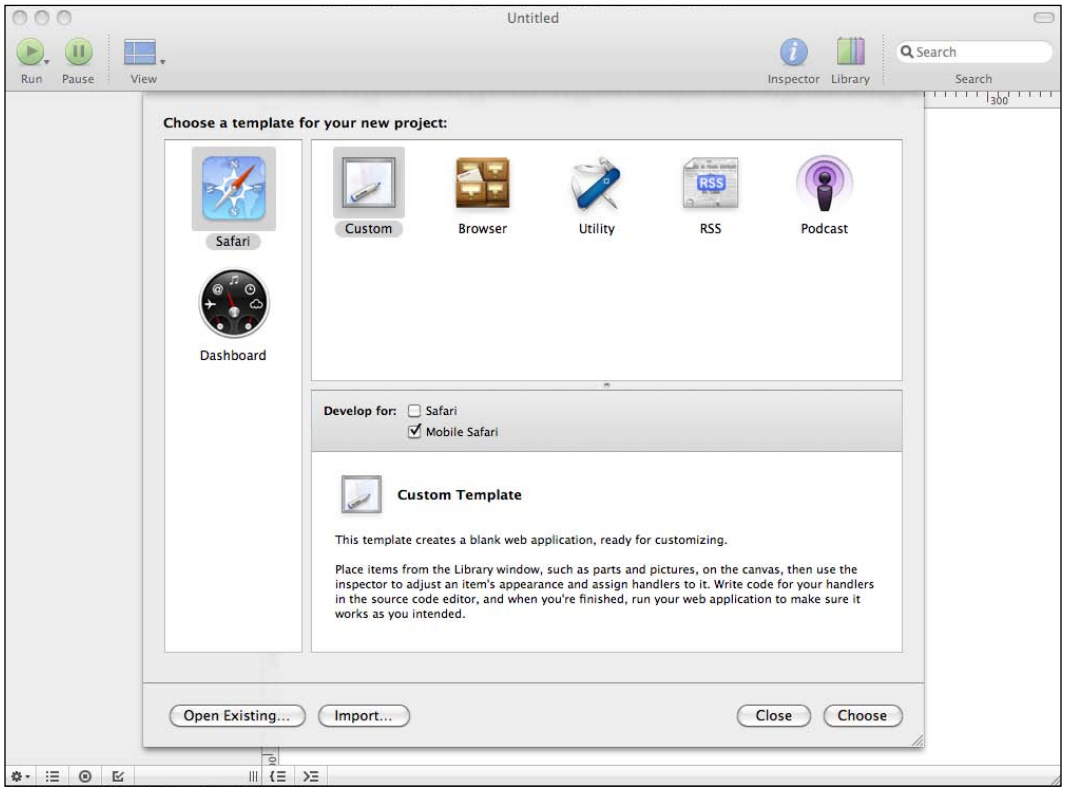
When the DVD is loaded, a new Finder's window will be launched. This window will show two folders, one called **Optional Installs** and other called **Instructions**. Click on the **Optional Installs** folder. Inside this folder, you'll find a binary package called **Xcode**. If you click on it, the installation process will be started. Follow the instructions on the screen provided during this process. After some minutes, Xcode will be installed on your computer.



How it works...

Mac OS X uses a different folder for **Applications** to store specific tools for developers. This folder is called the **Developer** folder and it contains another folder called **Applications**, where you'll find **Dashcode**. For accessing the **Developer** folder, you can click on the folder, which represents your main device; usually it is called **Macintosh HD**. The absolute path of the executable file for Dashcode is `/Developer/Applications/Dashcode`.

Dashcode includes a template for creating web applications for iPhone. When you create a new project (**File** | **New Project**), you must deselect the **Safari** option in the **Develop for** checkbox. Only the **Mobile Safari** option will be checked, as shown in the following screenshot:



This development tool allows creating a graphic interface for our web applications in an easy and intuitive way. We can drag-and-drop defined widgets such as buttons, lists, and image containers inside the builder area. The properties and the behavior of each widget can be set through different windows and dialog boxes. On the other hand, Dashcode includes a text editor for writing the needed JavaScript code to add functionality to our applications. In fact, Dashcode is a tool similar to other development tools designed to build applications with a graphic user interface using a specific toolkit or library.

When you finish your development and the application is ready to run, you can test it using the iPhone simulator. This one will be invoked directly from the **Run** button in the main toolbar of the Dashcode. Once your application is ready for the users, you should deploy it in a configured production environment. Dashcode launches the simulator, which loads your application from an internal web server designed only for development and testing. This internal server uses the port 49853 on the localhost referred to the simulator.

As a part of the documentation reference of Apple, you can find a user guide to use Dashcode at: `http://tinyurl.com/69vgmea`.

2
Building Interfaces

In this chapter, we will cover:

- Creating a toolbar
- Modifying the default status bar
- Creating a footer
- Creating a back button
- Creating a button for the toolbar
- Building a breadcrumb menu
- Building the duo navigation buttons
- Building the lists for items
- Building menus using lists
- Creating the toggle buttons
- Creating a modal box with buttons
- Building a search dialog
- Building the information fields
- Building forms with checkboxes, radio buttons, select fields, and text fields
- Creating and customizing a notification box
- Building a chat-style interface
- Creating a date picker
- Using different tabs

Introduction

The graphical user interface, also knows as the *GUI*, is one of the most important components of modern software applications. It allows users to interact with the software through visual elements called *widgets*. Desktop and web applications use common elements such as buttons, labels, text inputs, checkboxes, and menus. However, the applications designed to run on mobile devices require a special and different GUI. The main reason is that it runs on a machine, which is different from a desktop computer or a laptop.

The iPhone introduced a new GUI designed specifically for this device. It's an exclusive interface that uses its own look and feel. In fact, this is the most distinctive feature of the operating system of this device. Although it's possible to use the native widgets through the Objective-C programming language, we'll see how to do it in an alternative way. Actually, we're going to use HTML, CSS, and some JavaScript code.

If you're not familiarized with the iPhone interface, it is a good idea to take a look at the built-in applications on the device. It gives you a clear idea what we're talking about. The same consideration will be taken for the iPad.

As you're building a graphical interface for your iPhone applications, you'll need to keep in mind some recommendations and concepts for getting an application with a native look and feel. Remember, your target device is not a desktop computer or a laptop. We can summarize these recommendations and concepts as follows:

- The iPhone viewport is a rectangular area that determines how the content of your application is displayed on the device. In fact, this area defines the layout of the application. The viewport can be managed through the specific HTML *meta* tag used by *Safari Mobile*.
- Users will interact with the application through *gestures* such as tap, double tap, drag, and flick. They aren't using a mouse or a keyboard.
- The iPhone always has a visible *statusbar*. This is a small area on the top displaying information about your carrier, the status of the battery, and the current time.
- The applications should contain a *toolbar* with a title at least. Usually, this will be the place to set navigation buttons such as the common *back* button. Also, the *action* button should be displayed in this area of the screen.
- Take care about colors. The iPhone uses specific colors for its graphical interface. In order to build applications with a native *look*, you should try to use the same.
- A *view* is an individual page with content displayed on the screen of the device. Therefore, an iPhone application can contain many views. Navigation between them must be defined using different widgets.

- Including *back* buttons is a good practice. You don't want your user to feel lost. Users should be able to come back to the previous view.
- Navigation is important. Don't forget to include some navigation widgets such as *menus* or *breadcrumbs.*

In this chapter, we find out how to build the main widgets for our iPhone applications using the main HTML/CSS/JavaScript frameworks mentioned in the previous chapter. Let's start building a *toolbar*.

Creating a toolbar

This recipe shows how to create a main toolbar for our application. The main toolbar is located at the top of the graphical interface and is the place to put the main navigation buttons and icons. Usually, this toolbar is the basic widget for iPhone applications.

Getting ready

In order to create our toolbar, we're going to use the *iWebKit* framework. A text editor will be used for creating and editing the main required HTML file. You can use the editor and the operating system of your choice for creating and editing the file. TextMate, Emacs, Vim, and Ultraedit are examples of advanced editors used by many developers for this task. These editors offer useful features such as syntax highlighting, auto-completion, and automatic indenting, making the development easier.

How to do it...

1. The first step will be to ensure that the *iWebKit* framework is installed on our computer, and then we'll create a new HTML file adding the following lines to load the main files of the framework:

```
<linkhref="../iwebkit/css/style.css" rel="stylesheet"
media="screen" type="text/css" />
<scriptsrc="../iwebkit/javascript/functions.js" type="text/
javascript"></script>
```

2. In order to create our main toolbar, we'll use two simple HTML *div* elements:

```
<div id="topbar">
<div id="title">This is the toolbar</div>
</div>
```

3. After loading your HTML file on the iPhone, you can see a screen similar to the next screenshot:

The complete HTML file can be found at `code/ch02/toolbar.html` in the code bundle provided on the Packtpub site.

How it works...

iWebKit offers a predefined *div* element with two CSS classes to build a main toolbar for applications. We only need to create a standard HTML page, adding the references to the HTML and CSS files of the framework, and including the referred *div*. The *id* attribute of the *div* element is used to create the toolbar.

A toolbar should display a label. Usually, this information will be a simple title. For this information we're using a new *div* element with a specific *id*, which is defined in the main CSS file of the *iWebKit* framework.

You can test your new page through a *WebKit* browser such as Google Chrome or Safari, using the *iPhone Simulator* on your Mac or directly on your iPhone device. For the moment, we're not using any server-side code so you can load your page from the filesystem or from your web server. In the first case, you'll need to use the file protocol instead of the common HTML. For instance, if you have installed the *iWebKit* framework under the control of your web server, you can load your page using the URL `http://localhost/toolbar.html`. Otherwise, you need to use the path to your file using the file protocol: `file:///home/user/toolbar.html`.

The *iWebKit* main files are loaded from the `iwebkit` directory, which is located outside of our main HTML file. Remember, we installed the frameworks in a directory under the main *DocumentRoot* of the web server.

One of the main advantages of using web technologies for developing iPhone applications is the independence of the operating system. Remember, we need a Mac OS X system to build native applications. However, we recommend testing our applications on the *iPhone Simulator* because the result is closer to the real iPhone device. Windows users can use Safari or Google Chrome on their local machines. Meanwhile, Linux users should use Google Chrome, although it is possible to use Safari under Wine (`http://www.winehq.org`).

There's more...

The color of the main toolbar is blue by default, but the *iWebKit* allows changing it using a black or transparent color. You only need to add a new CSS class to the main `div` element. This class is defined in the `style.css` file of the framework. The names of these classes are *black* and *transparent*. For example, to change the background of the main toolbar to black, we'll use the following line of code:

```
<div id="topbar" class="black">
```

If we reload our page, we can observe how the background color has changed, as shown in the following screenshot:

See also

- *Installing the iWebKit framework* recipe in *Chapter 1, Frameworks Make Life Easier*

Modifying the default status bar

If you take a look at the application running in your iPhone device, you'll see a small, thin, and light gray bar at the top of the screen. This is the status bar and it shows information, such as the current time, the status of the battery, and the name of the carrier. In this recipe, we'll learn how to modify the default status bar of the iPhone.

Getting ready

For this recipe, we don't need any framework as we can use one of the native properties of the *Safari Mobile* web browser installed in the device.

How to do it...

We only need to add one line in the *meta* section of the HTML file. Instead of creating a new file, you can reuse the file used for the previous recipe. To change the background color to black, we'll add this line:

```
<meta name="apple-mobile-web-app-status-bar-style" content="black" />
```

If we prefer to change the default status bar to translucent black, we can do it by adding the following line:

```
<meta name="apple-mobile-web-app-status-bar-style" content="black-
translucent" />
```

After changing and reloading your file, you'll see something similar to the following screenshot:

How it works...

The *meta* tag provided by the *Safari Mobile* browser allows to modify the default status bar of the iPhone. In fact, this tag only allows changing the background color. If we don't need to modify this bar, we can omit the tag or just use the *default* value for the *content* attribute of this tag.

Keep in mind that the *apple-mobile-web-app-status-bar-style* only works when the application is using the full screen mode and, when an icon for the application has been added to the home screen. Don't be impatient. We'll take a closer look at these issues in the following chapters.

See also

- *Viewing applications in full screen* recipe in *Chapter 3*, *Events and Actions*
- *Choosing an icon image for the application* recipe in *Chapter 4*, *A Picture Speaks a Thousand Words*

Creating a footer

The first recipe of this chapter shows how to create a main toolbar with a title. It seems useful to create a *footer* with some relevant information. Usually, a footer uses a small font and a different color to be different from the rest of the text in the same screen. Let's find out how to create a footer for our applications.

Getting ready

We'll use the *iWebKit* framework as it contains a specific CSS style for creating a footer with a consistent look for our applications.

How to do it...

As we have seen in the previous recipe, we can reuse our initial HTML file. You only need to add one line before the `</body>` tag with this content:

```
<div id="footer">This is the footer</div>
```

The following screenshot shows a simple footer:

Note that it makes sense to add the footer after all the content and widgets of our application are added. The previous example illustrates how to create the footer but the developer must put it in the right place.

Don't forget to check if the lines for loading the *iWebKit* main files are present inside the *head* section of the HTML file, as shown in the first recipe of this chapter.

How it works...

Through the *id* attribute of the *div* tag, we're loading a predefined style present in the *style.css* file of the *iWebKit* framework. The text inside of this tag would be shown centered using a small and gray font.

Many developers use the footer to put a copyright notice or similar information. In addition, we can add one link to our website.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Creating a back button

At this point we have the basic elements for our application. We have learned how to create a toolbar and a footer, and how to modify the main status bar of the device. Probably you're thinking our toolbar is too simple. Yes, you're right. It could be more useful if we add some buttons. One of the most important buttons for this bar is the *back* button. This is a small and blue button allowing the user to come back to the previous page of our application. It's very intuitive for the user and is similar to an arrow pointing to the left. Formally, the *back* button is a navigation widget for the user interface.

Getting ready

We're continuing using the *iWebKit* framework installed on our local machine.

How to do it...

Open your previously created HTML file and add the following line below the *div* element with the *title* value in the *id* attribute:

```
<div id="leftnav"><a href="#">Back</a></div>
```

Take a look at the following screenshot and check how the main toolbar has been changed:

Our **Back** button can be more intuitive and funny when using an icon for it. The *iWebKit* framework contains some images for this functionality. One of these represents a house and can be used to get our user back to the homepage of the application. We'll need to change the text inside the *link* tag for an *image* tag. The complete line of code for it will be as follows:

```
<div id="leftnav"><a href="#" /><img alt="Go Home" src="../iwebkit/
images/home.png /></a></div>
```

Reload your page and observe how the button has changed:

How it works...

iWebKit provides a special class to transform a simple *div* element into a button similar to an arrow. The CSS class is called *leftnav* and it was designed to be used as a *back* button for the main toolbar of applications. Inside this HTML element, we need to put a link to the right page. This *back* button doesn't make sense for a *homepage,* but it will be very useful for secondary pages.

As the back button is a component of the toolbar, the *div* element must be inside the *div* whose *id* is *topbar*. Don't forget the link for this button; a navigation element should drive the user to a specific page or place.

In some cases a *next* button could be useful. This one will be similar to the *back* button and it will use an arrow pointing to the right instead of to the left. To use this kind of button, we only need to change the *id* attribute of the *div* element. Instead of using the *leftnav* class, we must use the *rightnav,* also provided by the *iWebKit* framework.

Probably, you observed that *leftnav* and *rightnav* are values for the *id* attribute instead of names of the CSS classes. The reason is simple; we can only have one button of each kind per page.

There's more...

Obviously, you can use your own icon for the *back* and *next* navigation buttons of the toolbar. Your icon should be a PNG file with 20x20 pixels and it can be located in any directory with permissions to be accessed by the web server.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating a toolbar* recipe

Creating a button for the toolbar

The toolbar bar can contain simple buttons as well, not only navigation buttons. For instance, the *Cancel* button is very common and the status bar can be the right place for it. This kind of button is different; we don't need any arrow or icon. A rounded button with a simple text is enough for the purpose.

Getting ready

Once again, *iWebKit* will be our framework for this new recipe.

How to do it...

It's possible to put our new button on the left or on the right inside the toolbar. We only need to use a specific attribute, which identifies a CSS style defined in the `style.css` file of the framework.

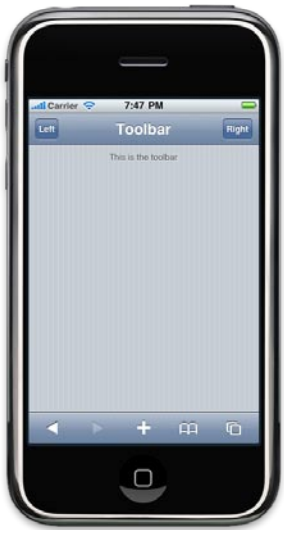
To put the button on the left, we'll add the following line inside the *div* with the *id="topbar"* and below the *titlediv* element:

```
<div id="leftbutton"><a href="#">Left</a></div>
```

If we need a button on the right, we'll add the next line of code:

```
<div id="rightbutton"><a href="#">Right</a></div>
```

When you are reloading your page, you should see a screen similar to what is shown in the following screenshot:



How it works...

The main CSS file of the *iWebKit* framework defines a special style for navigation buttons and another one for common action buttons in the toolbar. Although it is not very common to use one left and one right in the same page, it can be useful for some applications. Usually, developers prefer to use only one of them in the same page. However, it is very common to use an arrow button and a left or right button together in the toolbar. One of them could be used to come back to the home page and the other one for a specific action such as loading a selector widget.

iWebKit has defined a specific style for changing the color of these buttons. In fact, you can change the color of the left button to blue using the `blueleftbutton` value for the *id* of the *div* element. The other style is accessible through the `bluerightbutton`.

As the navigation buttons, these new kind of buttons must be located inside the *topbar* element.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating a toolbar* recipe

Building a breadcrumb menu

A *breadcrumb* is a useful and common widget in modern graphical user interfaces. This navigation widget gives users a way to keep track of their location. Breadcrumbs provide a simple mechanism to navigate from the current page to its parents using different links. Typically, graphical elements are used to separate elements inside the breadcrumb. Arrows are one of these common elements. In this recipe, you'll learn how to build a breadcrumb widget for the iPhone applications using the *iWebKit* framework.

Getting ready

For this recipe, we are going to use the *iWebKit* as it offers a simple way for building breadcrumbs.

How to do it...

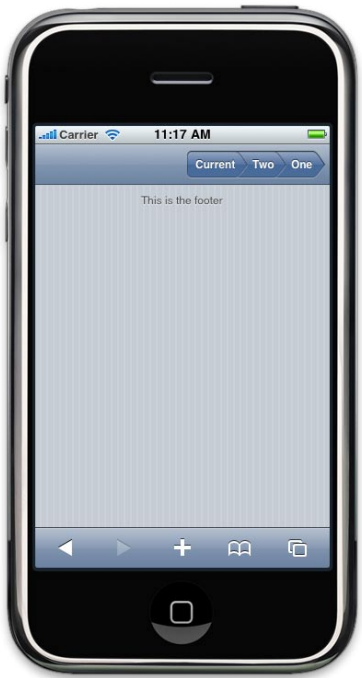
Once again, we're going to use our base file for the *iWebKit* framework created as seen in the first recipe of this chapter. Edit this file and add the following lines below the *div* element with *id* equal to *title*:

```
<div id="leftnav">
  <a href="#">One</a>
  <a href="#">Two</a>
  <a href="#">Current</a>
</div>
```

Let's see our new page on the *iPhone Simulator*. The result on the screen will be similar to the following screenshot:

How it works...

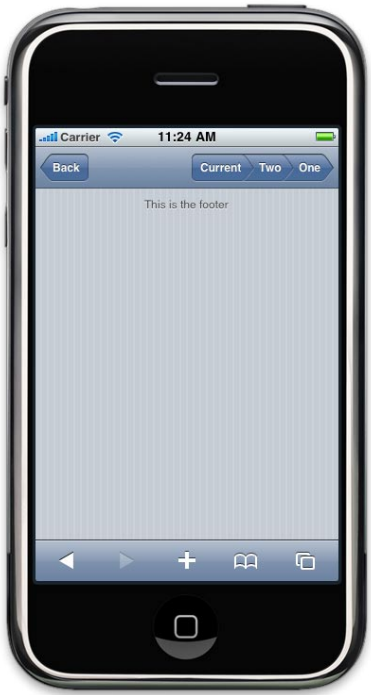
As mentioned in the recipe about how to create a back button, we used the same *leftnav* class. The difference in this new recipe is the use of more than one link. Using different HTML links inside the *leftnavdiv* element, we can build a simple breadcrumb widget. Also, it is possible to get a breadcrumb on the right side of the toolbar; you should simply change the *leftnav* value to *rightnav*. Take a look at the next screenshot to see this breadcrumb located on the right of the main toolbar.



A more complicated case is when you need a back button with a breadcrumb in the same toolbar. This is not very usual, but it can be useful for some applications. To solve this issue, you can add a left and right navigation in the toolbar using the CSS classes provided for the *iWebKit* framework. The following code shows how to do it:

```
<div id="leftnav">
      <a href="#">Back</a>
  </div>
  <div id="rightnav">
      <a href="#">One</a>
      <a href="#">Two</a>
      <a href="#">Current</a>
       </div>
```

After reloading your page, you'll see a screenshot similar to the following:



Don't forget to change the # symbol for the complete name of the referenced HTML page. Keep in mind that the purpose of the back button and the breadcrumb is to offer a navigation tool for the user.

If you're wondering whether it is possible to use an image for the breadcrumb elements, the answer is *yes*. The process is straightforward; simply add an HTML *img* tag inside each link.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating a toolbar* recipe

Building the duo navigation buttons

Another important navigation widget for the graphical user interface of our iPhone applications is the *duo* buttons. These buttons allow us to go to different pages. It's also is possible to use these buttons as triggers for actions. For instance, one of these buttons can be used for cancelling an action. Let's show how to build this navigation widget.

Getting ready

The *iWebKit* framework provides a good set of navigation widgets, so we'll use this one again.

How to do it...

The process for building this navigation widget is very simple; you only need to create a *div* element with a specific CSS class and add two links inside it. Take your work file for the *iWebKit* and replace its content with the following lines inside the *topbardiv*, which indentifies the main toolbar of our application:

```
<div id="duoselectionbuttons">
  <a href="#">A</a>
  <a href="#">B</a>
</div>
```

The new widget is as shown in the following screenshot:

How it works...

Using the *duoselectionbuttons* value for the *id* attribute of the *div* element, the *iWebKit* will be able to create one widget with two different buttons. The style for this widget is defined in the main CSS included in this framework.

As you can check it, the new widget is created inside the toolbar. The *iWebKit* also includes other CSS styles for building a *duo* button outside the main toolbar. We'll need to use a new style and a new *div* element to indicate that the widget is different. The required code to do that is as follows:

```
<div id="duobutton">
  <div class="links">
    <a href="#">One</a>
    <a href="#">Two</a>
  </div>
</div>
```

The preceding code snippet must appear after the *topbardiv* element. This is an important difference to distinguish between these different kinds of duo buttons.

There's more....

In addition to the duo navigation buttons, *iWebKit* offers to let us to build trio navigation buttons. We can build this widget using the following lines of code:

```
<div id="triselectionbuttons">
  <a href="#">A</a>
  <a href="#">B</a>
  <a href="#">C</a>
</div>
```

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Building the lists for items

One of the most used widgets in the graphical user interface for iPhone applications is the *list*. This one shows different items separated by a horizontal thin gray line. The lists can be useful for showing sorted information to the user. For instance, a list is ideal for listing a set of records from a database. Usually, each item of the list is a small text. On the other hand, the items inside the list can contain an image or a comment next to the mentioned text.

Obviously there exists different kinds of lists, the most basic and simple is a set of items containing text with one or two words. All the items of this simple list appear to be separated only by a thin line. Other lists are *rounded* and allow a blank space between other widgets. It is even possible to build lists such as a navigation menu using links, as we'll see in the recipe called *Building menus using lists,* which is explained later in this chapter.

In this recipe, we're going to explain how to build simple lists with text, comments, and images.

Getting ready

Instead of the *iWebKit* framework used in the previous recipes, we'll use the *UiUIKit* for this recipe. The reason is because this framework offers a simple way for building this kind of list with minimum code and effort.

To refresh your memory, take a look at the previous chapter to find out how to install and configure the *UiUIKit*.

How to do it...

As we're going to use the *UiUIKit*, we'll need to add the reference to the required file of this framework to our HTML file:

```
<linkrel="stylesheet" href="../uiuikit/stylesheets/iphone.css" />
```

The process for building our simple list is very intuitive. As in regular HTML files, you only need to create an unordered list using the *ul* tag and then add as many *li* elements as items for the lists you need to create. The following is the required code for creating the simple list:

```
<ul>
  <li>Item 1</a></li>
  <li>Item 2</a></li>
  <li>Item 3</a></li>
  <li>Item 4</a></li>
</ul>
```

When you're ready, you can load your new HTML page on the device for viewing a page, as shown in the following screenshot:

How it works...

In order to get our first simple list running, we'll need to use the *UiUIKit* framework, which allows creating a toolbar and other areas to put inside different widgets for the graphical user interface. This framework only uses a CSS file to create the required layout for our iPhone applications. We're using a standard XHTML page with a reference to the specific CSS file of the framework. The *body* element of the page requires a specific *id* with the *normal* value. This *id* is defined in the mentioned CSS file of the framework. Inside this element, we'll to put a special *div* element to create the main toolbar. This *div* element needs to use other styles defined in the `stylesheet` file called *header*. The *h1* element is used for the title of this toolbar. After this initial *div* element, we can put other widget elements as our first simple list. In the previous screenshot, we can observe a toolbar with the title **Simple list**. The code for this toolbar is given next:

```
<div id="header">
  <h1>Simple list</h1>
</div>
```

You can find the complete HTML file for this recipe at `code/ch02/simple_list.html` in the code bundle provided on the Packtpub site.

The simple list can be modified to insert text on the right side of the item. This one can be useful to show complementary information. For instance, when you access the **Settings** application installed on your iPhone, you can view a navigation list with different options. If you click on the **General** option, a new list appears. The **About** items takes you to a new screen showing a simple list with information about the number of songs, the capacity of the device, the number of installed applications, and other similar information. This data shows how to use a simple list with different items, where each of them is built using a description on the left side and other text in different format on the right side. To imitate this behavior, we can use a simple list provided by the *UiUIKit*. The required code is as follows:

The following screenshot shows the original list of the *Settings* application of the iPhone Simulator:

And our similar list that we built with the help of the *UiUIKit* framework:

There's more...

The *UiUIKit* framework allows the use of links for each item of the simple list. If we put an HTML link inside each *li* element of the list, we'll get a link to other pages.

On the other hand, this framework offers the option to add icons to the items of our simple list. You only need to add an *img* HTML tag inside each *li* tag of the simple list. This new HTML element requires a specific CSS class called `ico`. Take a look at the following code to learn how to do that:

```
<ul>
  <li><imgsrc="../uiuikit/images/list-icon-1.png" class="ico" />Item
1</li>
  <li><imgsrc="../uiuikit/images/list-icon-2.png" class="ico" />Item
2</a></li>
</ul>
```

This new list with icons is shown in the following screenshot:

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Building menus using lists* recipe

Building menus using lists

The menu is one of the most important widgets of the graphical user interface of the software. It allows a structured and hierarchical way for accessing different options and features offered by the application. Desktop applications use a menu inside the main toolbar and web applications apply techniques such as the use of unordered list or tabs. For the iPhone applications, we will need a specific widget to emulate a conventional and typical menu as the common pull-down menus are not used by the iPhone. Usually, this widget is a navigation list, which is similar to the simple list showed in the previous recipe. However, the native look of the iPhone uses a rounded list where each item has an arrow to get access to other pages or to other menu lists.

This recipe helps you to build menus using lists with the elements and styles provided by the *UiUIKit* and the *iWebKit* frameworks.

Getting ready

The *UiUIKit* and the *iWebKit* frameworks offer simple methods for building our desired menus, so we'll use these for our purpose. Make sure you have installed and configured both of them before continuing.

How to do it...

The process for building a menu for our iPhone applications is very easy. Both frameworks provide CSS styles to do it. Let's explore the two different methods.

Using UiUIKit

We'll start using the *UiUIKit* for building a simple menu. First of all, create a new HTML file with the reference to the required CSS file of this framework. Don't forget to include a toolbar using one *div* element with *id* equal to *header*. Instead of adding an *id* attribute for the tag body, leave it blank. To refresh your memory, take a look at the previous recipe.

The next step will be to create an unordered HTML element, where each item has a link to the next page:

```
<ul>
  <li class="arrow"><a href="#">Item 1</a></li>
  <li class="arrow"><a href="#">Item 2</a></li>
  <li class="arrow"><a href="#">Item 3</a></li>
</ul>
```

You can see your new menu loading with the just created HTML file using your iPhone device, the software simulator, or a *WebKit* web browser on your local machine. The result will be as shown in the following screenshot:

Using iWebKit

Take the base file that we created for the first recipe of this chapter. It had the required references to the *iWebKit* files plus a toolbar. After this one, we'll create a new *div* element, which contains an unordered HTML list. The following code shows how to do it:

```
<ul class="pageitem">
    <li class="menu">
        <a href="#" >
          <span class="name">Item 1</span>
          <span class="arrow"></span>
        </a>
    </li>
  <li class="menu">
        <a href="#" >
          <span class="name">Item 2</span>
          <span class="arrow"></span>
        </a>
  </li>
<li class="menu">
        <a href="#" >
          <span class="name">Item 3</span>
          <span class="arrow"></span>
        </a>
 </li>
</ul>
```

Observe the final result on the screen; it is very similar to the list building using the *UiUIKit*:

How it works...

Both frameworks use CSS styles to get a menu working for our application. The method provided for the *UiUIKit* is easier than the other offered by the *iWebKit*. In the first case, we only need to include a link element inside each item and adding a style—called *arrow*—for the *li* element.

The method used by the *iWebKit* is a bit more complicated due to the need of a new *div* element. As a part of this, we need to use two different *span* elements inside each item. One of these is for the item text itself, and the other one is for the small arrow, which appears on the right side. Note each span has its own CSS class, *name* for the text, and *arrow* for the small arrow. Also, the unordered list presents a different style represented by the value *pageitem* of the *id* attribute. The *pageitem* is the most important element of the *iWebKit* framework. Basically, this component or style is the outline of other components and widgets, such as menus or textboxes. Each page of your application can include many *pageitems*, and each of these components will create a group. We can organize widgets using these individual groups.

Keep in mind that the examples shown are using the # symbol for the *href* attribute. In the real world, we'll need to replace it with a reference to a new page.

Surely, you can guess that it is possible to create nested menus. If each item points to a new page that includes a new menu, we'll have the desired effect. Don't forget that interfaces for mobile devices are quite different than those used by desktop and standard web applications.

There's more....

The *iWebKit* allows you to include a small text on the right side of each menu item. You only need to add a new *span* element with the class called `comment`. The following code segment illustrates how to do that:

```
<li class="menu">
  <a href="#">
    <span class="name">Item 2/span>
    <span class="arrow"></span>
    <span class="comment">Click me!</span>
  </a>
</li>
```

On the other hand, you can use the *UiUIKit* for the same task. This issue is very simple; we'll include a *small* tag inside the selected item as shown in the next fragment of code:

```
<li class="arrow"><small>Click me!</small><a href="#.html">Item 3</
a></li>
```

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating a toolbar* recipe
- *Building lists for items* recipe

Creating the toggle buttons

A *toggle* button allows you to choose between two possible states or values. The most common use of this widget is for actions or properties, whose value can be *On* or *Off*. The value can be changed by clicking on the button. It detects the event and automatically changes the value from one to the other.

Getting ready

As in the previous recipe, we're following with the *iUI* framework for this recipe.

How to do it...

Open the file you've created for the previous recipe with a text editor. We only use one row for the panel, so you can delete the other two rows. After the *label* tag for the one and only row, we'll add the following code:

```
<div class="toggle" onclick="return;">
  <span class="thumb"></span>
  <span class="toggleOn">On</span>
  <span class="toggleOff">Off</span>
</div>
```

Reload your modified HTML page for viewing the result, as shown in the next screenshot:

Click on the button and observe how it changes to **On**. Observe that the color of the **On** state is blue, however the **Off** state is using a light gray color.

How it works...

The main *div* element used by the toggle needs a special class called `toggle`. It can only be used inside rows for panels. Also, this HTML element requires three additional *span* tags and each one has its own CSS class. The first one belongs to the `thumb` class and is left blank. The `toggleOn` represents one of the possible values for the widget and the `toggleOff` the other value. Meanwhile, `toggleOn` will be the *activated* value and the `toggleOff` will be the *disabled* value.

The `onclick` JavaScript event handler manages the state for the toggle button. This handler can be used to respond to the event with a defined action using a JavaScript function. By default, the handler is using the `return` JavaScript command, which indicates that the button must change from `On` to `Off` and vice versa. This behavior is managed by the `ui.js` JavaScript file of the *iUI* framework.

Although the predetermined value for the toggle button is **Off**, you can change it by adding a no standard attribute for the *div* element. Specifically, this attribute is called *toggled* and its value should be `true`. The following segment of code shows you the modified *div* element:

```
<div class="toggle" onclick="return;" toggled="true" >
```

If you use the *div* element with the *toggled* attribute and reload your page, you'll see how the toggle button is **On** instead of **Off**.

There's more...

The toggle button is not exclusive to the *iUI* framework. Others such as *iWebKit* and *PhoneGap* allow you to create this kind of widget.

See also

- *Installing the iUI framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Creating a modal box with buttons

A *modal box* is similar to a conventional dialog window used by web and desktop applications. It appears as a response to an event. For instance, when a user clicks on a button then the `click` event is launched. This modal box can contain some buttons to launch different events or actions, and it is useful for offering different choices to the user without leaving the current page. The appearance of the modal box is not the same as the rest of the widget of the page. It is showed with a black background color at the bottom of the page. We will look at the process of building these kind of widgets in this recipe.

Getting ready

We go back to use the *UiUIKit* framework as it includes a specific widget for building this widget.

How to do it...

As we have seen in the previous recipes, we'll include the reference to the CSS file of this framework in our new HTML file. To refresh your memory, here is the line of code to do that:

```
<linkrel="stylesheet" href="../uiuikit/stylesheets/iphone.css" />
```

The second step will be to create a simple button for launching our modal box:

```
<p>
  <a class="green button" href="#" onclick="show_hide_box('optionpanel
');">Show box</a>
</p>
```

The next step will be to create our modal box using the following code:

```
<div id="optionpanel" style="display: none">
  <p>
    <a class="black button" href="#">Black button </a>
    <a class="white button" href="#" onclick="show_hide_box('optionpan
el');">White button</a>
    <a class="red button" href="#">Red button</a>
  </p>
</div>
```

Remember, our new modal box will be launched as a response to an event. We need some JavaScript code to respond to it. Insert the following lines in the *head* section of your HTML file:

```
<script language="javascript">
  <!--
    functionshow_hide_box(div_box) {
      element = document.getElementById(div_box);
      if (element.style.display == 'block') {
        element.style.display = 'none';
      } else {
        element.style.display = 'block';
      }
    }
  //-->
</script>
```

Now it's time to click on the green button to see our modal box working, as shown in the following screenshot:

The complete code for this recipe can be found at `code/ch02/buttons_panel.html` in the code bundle provided on the Packtpub site.

How it works...

In order to create the button for launching our modal box, the *UiUIKit* defines a link element with a CSS class called `button`. You can choose the color for your button using a predefined CSS style. We chose the green color for our recipe, but the *stylesheet* of the framework allows other colors such as white and black.

We need to launch the modal box when the button is pressed. This is the reason for using the `onclick` handler event on the HTML link of our button. We built a simple JavaScript function detecting the current value of the property `display` of the modal box. This one is hidden by default when the user loads the page and will be shown after clicking on the main green button. In fact, the JavaScript function works as a toggle component. The only parameter of this function is the *id* of the modal box and it is used to find out the value of its `style.display` property. You'll observe the use of the `getElementById` method of the `document` JavaScript object, which is defined in the browser following the **DOM** (**Document Object Model**) model. It will be considered a native feature of the *Safari Mobile* browser used by the iPhone and the other Apples devices. Desktop browsers—based on the *WebKit* engine—such as *Google Chrome* and *Safari* contain the same feature for accessing HTML objects through the mentioned DOM model.

The modal box itself is created using a specific *div* element with a predefined *id* value called *optionpanel*. Taking a look at the *style* attribute of this element, you'll find its value indicates that the modal box will be hidden by default. Inside the *div* tag, we're using a set of links for representing different buttons. Each of these buttons has a color defined through the *class* attribute of each link.

In the real world, the *href* attribute of each button should contain a reference to a specific action. For instance, we can get a value to put on a text field. Once again, you'll need some JavaScript code to do that.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Building a search dialog

The iPhone provides a specific way to display dialog boxes for searching. This kind of dialog is different than the modal box that we explained in the previous recipe. Actually, the search dialog box is displayed as a pop up at the top of the current page. Meanwhile, the rest of the same page is grayed out instead of using the black color as the modal box. The search dialog box should contain two different action buttons. One of them will be used for cancelling the search and the other one will launch this search. Let's explore how to build a standard search dialog for our iPhone applications.

Getting ready

The *iUI* framework uses a simple and efficient way for building our required dialog. Check the installation of this framework on your computer before continuing.

How to do it...

To create the mentioned dialog, the *iUI* framework uses a standard HTML form with two buttons, one label and one text input field. The code for this form is the following:

```
<form id="frmDialog" class="dialog" action="">
<fieldset>
  <h1>Search dialog</h1>
  <a class="button leftButton" type="cancel">Cancel</a>
  <a class="button blueButton" type="submit">Send</a>
  <label>Choose item</label>
  <input type="text" name="inputxt"/>
</fieldset>
</form>
```

The form will be hidden until the user clicks on the **Search** button of the main toolbar, but we'll need a specific value for the *href* attribute of the *anchor* element for this button, as shown in the next line of code:

```
<a class="button" href="#frmDialog">Search</a>
```

The next screenshot shows the initial state of the application, where the dialog box is not visible:

After clicking on the **Search** button, we'll see our dialog box:

How it works...

The *dialog* is a predefined CSS class for showing a dialog box using regular HTML form. Obviously, this form will need a value for the attribute *action* to get the value typed for the user in the text field. Usually, it will be a reference to a server-side component such as a PHP page or a CGI script. The links inside the form help us to create two different buttons, one for cancelling our search and other one for submitting the content of the form to the server. *leftButton* and *blueButton* are CSS classes defined to allow us to place the buttons on the left and the right side.

One HTML label is used to set a title for the dialog box and the text field is a standard *input* for HTML forms. No predefined CSS classes are required for these elements.

In order to launch our dialog box, we're using the *id* of the form as the value for the attribute *href* of our button. Observe that the value is preceded by the # character to indicate that the reference is in the same page.

There's more...

In *Chapter 6*, *Exchanging Data: AJAX*, we'll learn how to improve our search dialog box using AJAX to search a database located in the server-side of the application.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Searching with AJAX* recipe in *Chapter 6*, *Exchanging Data: AJAX*

Building the information fields

In previous recipes, we learned how to create simple lists for items and for navigation. This recipe explores the way to build a new widget for displaying a similar list. It contains labels and its associated values and will be very useful for showing information similar to a read-only form typically used in web applications. Actually, this widget shows fields with information related to it. We can establish a correspondence between this kind of field and the name of the rows of a set of data, for instance, a record of a database.

Getting ready

The *UiUIKit* framework will be used again for this recipe.

How to do it...

Be sure to put the required reference to the CSS file of the *UiUIKit* framework in your new HTML file. We can use a toolbar as seen in the previous recipes using this framework. The fragment of code shows how to get a list with some information fields about one user:

```
<ul class="field">
  <li>
    <h3>Name:</h3>
    <a href="#">Mark O'Connor</a>
  </li>
  <li>
    <h3>Phone:</h3>
    <a href="#">555 555 555</a>
  </li>
  <li>
    <h3>Company:</h3>
    <a href="#">ACME</a>
  </li>
  <li>
    <h3>Position:</h3>
    <a href="#">Software developer</a>
  </li>
  <li>
    <h3>e-mail:</h3>
    <a href="#">my@email.com</a>
  </li>
  <li>
    <h3>Address:</h3>
    <a href="#">Cypress Hill St.<br />New York<br />USA<br /></a>
  </li>
</ul>
```

After loading your new page, you'll see something similar to the next screenshot:

How it works...

As you have seen, we're using a conventional unordered HTML for building this kind of widget. The *UiUIKit* framework defines a special CSS class (*field*) for it. Each item—the *li* tags—of this list contains two additional HTML elements. One of these elements is the title of the field and the other one is its value. We'll use a *h3* node for the first element and a simple *anchor* for the value. The *stylesheet* of the framework indentifies the tags and its classes to build the widget.

To distinguish between the name of the field and its value, the *UiUIKit* framework uses different colors for the text. The values in black and the fields in light blue color are used as the native look of the iPhone.

If you need more than one line for your values, the *br* HTML tag can be used for building multiline values. Our example shows one of the most common of this kind of field: the address of one person.

In practice, it's interesting to add a title for our information field's widget. We only need to insert a line of code, before the unordered list, like the following:

```
<h1>User</h1>
```

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Building forms with checkboxes, radio buttons, select fields, and text fields

One of the most common widgets for building graphical interfaces is the form. It is used as an interactive component to get information typed by the user. Modern web and desktop applications make intensive use of this widget. The iPhone applications are not an exception. Forms are composed of different widgets as text fields, checkboxes, radio buttons, and selection fields. All of them have a custom look and behavior designed for the iPhone device. Explore this recipe to learn how to build a form with different widgets.

Getting ready

Building a complete form is a straightforward process using the *iWebKit* framework. Let's see how to do it with the help of this set of JavaScript and CSS files.

How to do it...

For this recipe, we'll use a new blank XHTML file. After the main standard headers, don't forget to include the required lines for loading the main files of the *iWebKit* framework:

```
<linkhref="../iwebkit/css/style.css" rel="stylesheet" media="screen"
type="text/css" />
<scriptsrc="../iwebkit/javascript/functions.js" type="text/
javascript"></script>
```

The next step will be to toolbar through this code:

```
<div id="topbar">
  <div id="title">Form</div>
</div>
```

Our first widget for the form will be a pair of text inputs, one of these allows us to type a password. It will be transforming characters into dots as the user is typing:

```
<span class="graytitle">Text Fields</span>
<ul class="pageitem">
  <li class="bigfield">
    <input placeholder="Username" type="text" />
  </li>
  <li class="bigfield">
    <input placeholder="Password" type="password" />
  </li>
</ul>
```

If you load your page on your iPhone device or on the *iPhoneSimultor*, you can see a screen, as shown in the following screenshot:

The next step is to create a simple checkbox field using the following segment of HTML code:

```
<span class="graytitle">Checkbox</span>
<ul class="pageitem">
  <li class="checkbox">
    <span class="name">Remember Me </span>
    <input name="remember" type="checkbox" />
  </li>
</ul>
```

Reload your page to get a page as shown in the following screenshot:

Another useful widget used by forms is the radio buttons. Let's see how to create a radio button with three possible values:

```
<span class="graytitle">Radio Buttons</span>
<ul class="pageitem">
  <li class="radiobutton">
    <span class="name">Value 1</span>
    <input name="myradiobutton" type="radio" value="value1"/>
  </li>
  <li class="radiobutton">
    <span class="name">Value 2</span>
    <input name="myradiobutton" type="radio" value="value2"/>
  </li>
  <li class="radiobutton">
    <span class="name">Value 3</span>
    <input name="myradiobutton" type="radio" value="value3"/>
  </li>
</ul>
```

In order to test our new radio button, you can comment the previous lines of code of your file referring to the text fields and to the checkbox.

The selected fields allow the user the choice among different predefined values. The other name used for this widget is *combobox*. For creating one widget of this kind with three predefined values, we'll use these lines of HTML code:

```
<span class="graytitle">Selection boxes</span>
<ul class="pageitem">
  <li class="select">
    <select name="my-select">
      <option value="1">Option 1</option>
      <option value="2">Option 2</option>
      <option value="3">Option 3</option>
    </select>
  </li>
</ul>
```

When the previous code is added to your file, reload it to check the new widget on the screen:

Although it is not very common to use long text fields on iPhone interfaces, we can do that using a special widget included in the *iWebKit* framework. Simply add the following segment of code to your file:

```
<span class="graytitle">Textarea</span>
<ul class="pageitem">
  <li class="textbox">
    <span class="header">Long text</span>
    <textarea name="TextArea" rows="4"></textarea>
  </li>
</ul>
```

Take a look at the new screenshot to see how the iPhone displays this widget:

Obviously, a form can contain one or more of the widgets presented. Independent of this issue, we need a HTML form tag before the first *span* tag of our widgets. In addition, a *fieldset* tag is required after the *form* tag. Please, take a look at the complete code for this recipe. It can be reached in the `code/ch02/form.html` folder in the code bundle provided on the Packtpub site.

How it works...

To include one or more widgets in our form, we'll need to use an unordered HTML belonging to the `pageitem CSS` class defined in the `stylesheet` file used by the *iWebKit*. Remember, this class defines one of the most important components of this framework. It should appear after the toolbar declared by the *div* element with the *topbarid* attribute.

In this recipe, we separate the widgets by giving each one a title. The `span` element located before the unordered HTML list uses the class `graytitle` to introduce each item on the form. For instance, the text *Selected boxes* is used for the *combobox* widget, as shown in the following line of HTML code:

```
<span class="graytitle">Selection boxes</span>
```

The text fields use a specific CSS class (`bigfield`) for the *li* item of the list. As a part of this kind of text fields, the frameworks allow using another text field with a label indicating the name of this one. We should use the `smallfield CSS` class instead of *bigfield*. In addition, we'll need to add a *span* element for the `smallfield` label. The following code snippet shows a complete *li* HTML element to create a text field with label:

```
<li class="smallfield">
  <span class="name">Name:</span>
  <input placeholder="Your name here" type="text"/>
</li>
```

Forms are designed to send data from the client to the server. When using a form in our application, one needs a server-side component to retrieve data and process it. Actually, our *form* tag must include an *action* attribute referring to this component. At the moment, we forget this issue and we'll talk about it in *Chapter 6*, *Exchanging Data: AJAX*, to learn how to exchange data with a server. Right now, we only need to know that we'll set a *form* tag for building a set of fields belonging to a form widget. The first line of our form widget will be the following:

```
<form action="" name="frm" method="post">
```

The *UiUIKit* framework requires another HTML tag for building the form component. It will be after the HTML form we just mentioned:

```
<fieldset>
```

Don't forget to close both HTML tags at the end of your fields, otherwise, the form won't work properly.

The text typed by the user will be stored in the *input* component and it will be sent as a part of the values of the form. You can observe that we're using the *placeholder* attribute for the mentioned *input*. The value of this attribute displays a light grey text working as a clue for the user.

When the user clicks on a text field, the iPhone virtual keyboard appears, ready for input, as shown in the next screenshot:

Surely, you can guess that a *checkbox* is similar to the *toggle* button created in a previous recipe. However, the checkbox used by the *iWebKit* framework is designed to be included inside a form. In fact, if we send the form—through a **submit** button—we can pick up the value of the checkbox processing the request and sending a response to the client. The CSS class `checkbox` was used for the *li* HTML element.

Radio buttons are kinds of special buttons displaying a set of options. The user should choose one of these options by clicking on it. After clicking, the option will be selected and the form will be read for sending its value.

In the case of the radio buttons, a new CSS class is used. The name of this class is `radiobutton` and each option of the button needs it. Keep in mind that the attribute *name* of each *input* element must be the same. In our recipe, we're using the value `myradiobutton` for this attribute.

The iPhone has its own way of displaying the different options for a select field widget. When the user clicks on it, these options are displayed at the bottom of the screen. In addition, we find four different buttons. The most important of them is the **Done** button, which allows us to select the value for our select field. It can be found on the right in a different blue color.

Regarding the CSS class, the select field needs another different field called *select*. Inside the *li* element of the unordered HTML, we're using a typical HTML *select*.

Finally, the *textarea* widget is defined by the *iWebKit* framework through the CSS class *textbox*. A *span* with the style *header* is used to set a label for this widget.

There's more...

We know how to build text fields allowing users to type values through the keyboard. However, sometimes it will be interesting to display a keypad instead of the regular keyboard. It's pretty simple; you only need to change the *type* value of the desired *input* element:

```
<input type="tel" name="phone" />
```

Despite using different widgets separated for the same form, it's possible to join them in the same pane. Let's show a simple example for displaying one text field and select field in the same pane:

```
<span class="graytitle">My form</span>
<ul class="pageitem">
  <li class="bigfield">
```

```
      <input placeholder="Username" type="tel" />
    </li>
    <li class="select">
      <select name="my-select">
        <option value="1">Option 1</option>
        <option value="2">Option 2</option>
        <option value="3">Option 3</option>
      </select>
    </li>
  </ul>
```

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Building a search dialog* recipe
- *Creating the toggle buttons* recipe

Creating and customizing a notification box

Without doubt, applications need to send notifications to their users. The iPhone applications are not an exception, and the *Safari Mobile* browser implements a native JavaScript function to do that. Actually, this function is the same as the one used by other web browsers. Surely, you all know this function; it is the famous *alert*. However, if you call directly to this function in your code, the browser will launch a notification box with a title at the top. By default, this title displays the URL of the page, as shown in the following screenshot:

Many developers find this title unprofessional so it's better to build a custom notification box for setting our own title. As part of this title, we can change/add other options as we'll see in this recipe.

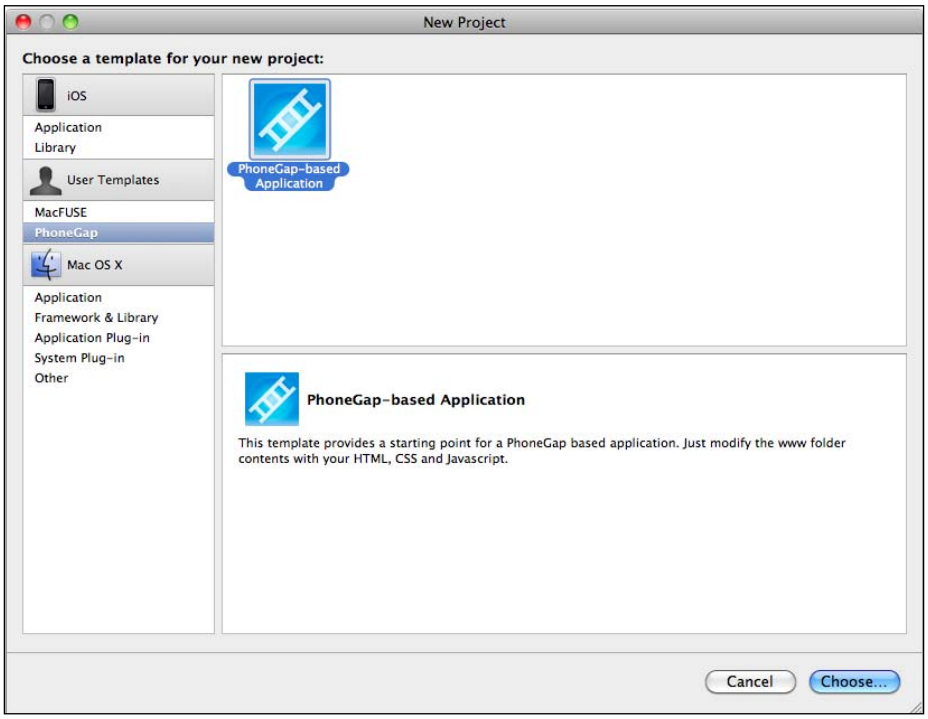
Getting ready

For this recipe, we will need a Mac computer as we're going to use the *Xcode* tool and the *PhoneGap* framework together. To refresh your memory about how to install both programming tools, take a look at the *Installing the PhoneGap framework* recipe in *Chapter 1*, *Frameworks Make Life Easier* of this book.

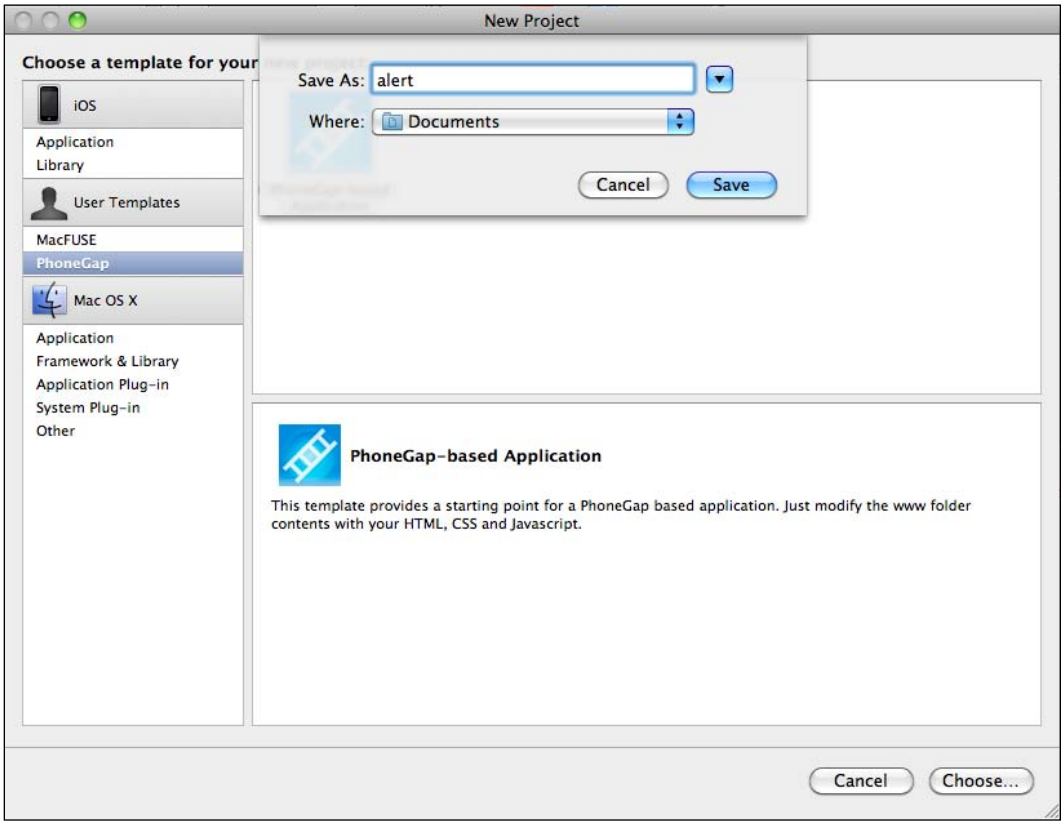
How to do it...

Before continuing, you should launch *Xcode*, which is accessible through **Macintosh HD** | **Developer** | **Applications** | **Xcode**.

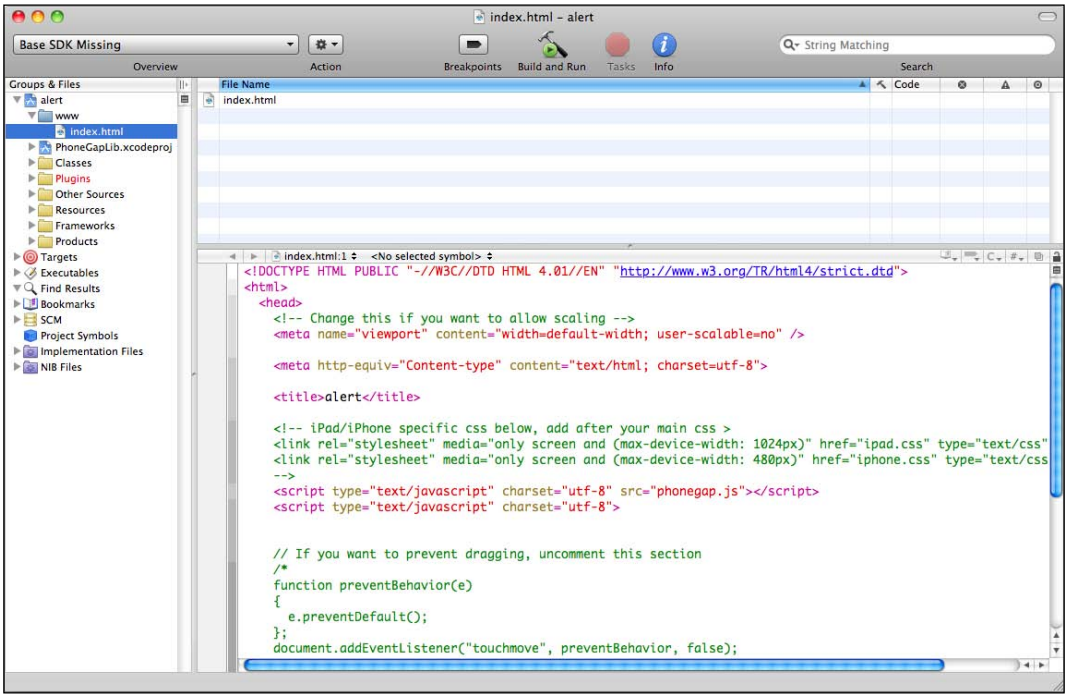
The second step will be to create a new project using the predefined template called *PhoneGap-based Application*. In order to do that, click on the menu option **File** | **New Project**..., and select the mentioned kind of template, as shown in the following screenshot:



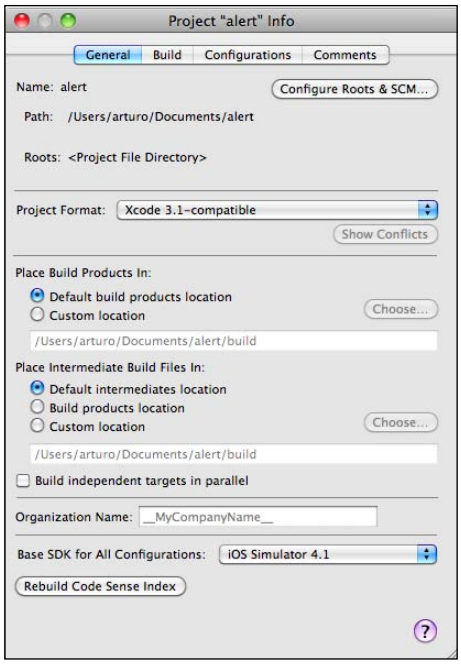
When you're ready, select the **Choose...** button to select a folder for creating your new *PhoneGap* project. You can do that through a simple dialog box:



In the **Save As** field, name the project as `alert`, and click the **Save** button to create the project. *Xcode* will create the required files for your new *PhoneGap* application. It uses a set of predefined files, including a main HTML file with the minimum JavaScript code required.



Now we're going to configure our project to use the *iPhone Simulator* to test and debug our application. In order to do this action, click on the menu option **Project** | **Edit Project Settings**. At the bottom of the new dialog box for the settings of your project, you can see an option called **Base SDK for All Configurations**. Be sure the option **iOS Simulator 4.1** is selected, as shown in the following screenshot:



Making our life easier, we'll take the CSS file defined and used by the *UiUIKit* for this recipe. Copy this file to the `www` folder of your `PhoneGapXcode` project. This CSS file is called `iphone.css` and is located on the `stylesheets` folder of your `UiUIKit` main installation folder.

Let's see the JavaScript and HTML required code for building our notification box. After creating your *PhoneGap* project, *Xcode* opens the main HTML file called `index.html` on the text editor. As you can see, this file has a skeleton and should be modified to build our application. Replace the `onDeviceReady()` JavaScript function with the following code:

```
functiononDeviceReady(){
  showAlert();
}
```

Now it's time to write the code for creating our notification box. Add the following lines after the previous `onDeviceReady()` function:

```
functionshowAlert() {
  navigator.notification.alert(
    'This is the alert box',
    'Your new title',
    'Done'
  );
}
```

At the top of the `index.html` file, you can see two references to the CSS files. Uncomment the line referring to the `iphone.css` file. The code line should be as the next one:

```
<linkrel="stylesheet" media="only screen and (max-device-width:
480px)" href="iphone.css" type="text/css" />
```

Be sure the following line appears on the `head` section of `index.html` file:

```
<script type="text/javascript" charset="utf-8" src="phonegap.js"></
script>
```

The last step is to build a simple toolbar through the following code:

```
<div id="header">
  <h1>Notification box</h1>
</div>
```

In order to test your application and verify that everything is right, click on the **Build and Run** button from the *Xcode* main toolbar.

The complete *Xcode* project containing all the code for this recipe can be found at the `code/ch02/alert/` folder in the code bundle provided on the Packtpub site.

How it works...

The *PhoneGap* framework contains a custom JavaScript class called `Notification`. It requires different parameters to build a custom notification box for your applications. These parameters are a string used as a message for the user, a title for the box, a string for creating one or two buttons, and a JavaScript `callback` function to handle the user's input. Formally, only the first parameter is mandatory, the others being optional. Our recipe is using a button with the label `Done`. After clicking on it, the notification box will be closed. This is the behavior by default.

Obviously, it's possible to add a `callback` function to execute some code when the user clicks on the button of the notification box. In order to do that, you need to write the code for this function and use its name as the second parameter of the constructor of the notification box. For instance, the following code illustrates this technique:

```
functionmyCallback() {
  // code for response after clicking on the notification box main
button
}
functionshowAlert() {
  navigator.notification.alert(
    'This is the alert box',
    myCallback,
    'Your new title',
    'Done'
  );
}
```

We're using the standard *onload* attribute for the *body* element for calling to a predefined JavaScript function when the application is loaded by the web browser. This function call to others is designed to initialize the *PhoneGap* framework. When it happens, the `onDeviceReady()` function is executed. It will be the main entry point for our application. Our recipe is calling to a function called `showAlert()`, which is building the notification box after the application is launched and the framework is loaded.

We have decided to use the `iphone.css` file of the *UiUIKit* for the main layout of our application, which is launching the notification box. The reason to do that is because using this simple file we can create an initial layout with a native look and feel through a few lines of HTML code. Obviously, you can use your own CSS file or other *stylesheet* provided by other framework. This recipe is a simple example combining two different JavaScript/HTML/CSS frameworks for iPhone applications.

One of the advantages of using the *PhoneGap* framework is that it allows you to build a native application for the iPhone using only HTML, CSS, and JavaScript code. In fact, you can use *Xcode* to create it in the same way as building your native application using the Objective-C programming language. Remember, to distribute your native application to the *Apple Store*, you'll need to become a registered *iPhone Developer* as we explained in the *Installing the PhoneGap framework* recipe in *Chapter 1*, *Frameworks Make Life Easier* of this book.

There's more

As a part of the `notification.alert` object used for this recipe, *PhoneGap* contains another kind of notification box called `notification.confirm`. This one allows you to build a box similar to the notification built by the *confirm* JavaScript function. In practice, the constructor for it uses the same parameters as the `notification.alert` object. We'll need two different buttons for allowing the user to choose between two options. In order to do it, you'll pass the label for these buttons separated by commas as the last parameter. In this case, the `callback` function will receive a parameter, which indentifies the index of the clicked button by the user. The next skeleton code shows how to do it:

```
functionafterChoosing(btnIndex) {
  // 'btnIndex' is an integer
}
function confirm() {
  navigator.notificatin.confirm(
    'Are you sure?',
    afterChoosing,
    'Choose',
    'Yes, No'
  );
}
```

See also

- *Installing the PhoneGap framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Building a chat-style interface

The iPhone introduces a special interface for displaying SMS and chat messages. It displays *bubbles* similar to comic balloons with the text of each user who participates in the conversation. These bubbles appear on the right or on the left depending on the user who types the text.

This interface may not be very common, but it can be useful for some kinds of applications needing to display text belonging to different users at the same time.

Getting ready

Let's use the *UiUIKIt* framework because it allows you to build this interface in an easy way. The final result will be awesome and funny!

How to do it...

First of all, we'll create a regular `body` HTML tag with a specific value (`chat`) for its `id` attribute. Don't forget to put this mandatory value, as shown in the following line:

```
<body id="chat">
```

Creating a bubble on the left is very simple:

```
<div class="bubble left">
  <p class="aqua">Hi!</p>
</div>
```

If you need some text below your bubble, add it using a HTML paragraph:

```
<p>mark</p>
```

After adding one initial left bubble, you'll need at least one more on the right:

```
<div class="bubble right">
  <p class="lemon">Hello Mark!</p>
</div>
```

Add as many bubbles as you need for your interface by adding different *right* and *bubbles* block *div* elements.

For instance, we can build a chat interface for displaying a chat between two different people as **mark** and **mary**:

How it works...

The CSS file of the *UiUIKit* provides a style for building the mentioned bubbles using a specific CSS property defined for the *Safari Mobile* and the other web browser based on the *WebKit* engine. This property is called *-webkit-border-image* and is useful for defining an image to be used instead of the regular border of an element. The CSS main file of the *UiUIKit* framework is using different images located in the `images` directory. Some colors such as pink, lime, or purple can be used directly through the name of a CSS predefined class.

For setting a bubble on the right, we are using a simple *div* HTML element with two different CSS classes: *bubble* and *left* or *right*. Inside each of these *div* elements, we need a HTML paragraph element for setting the text. The CSS class of this *p* element decides the color of the bubble using the mentioned `-webkit-border-image` property.

For instance, the following lines of CSS code show how the pink bubble is defined in the *stylesheet* file (`iphone.css`):

```
body#chatdiv.rightp.pink {
  -webkit-border-image: url(../images/chat_bubbles_pink_r.png) 10 19
13 10;
}
body#chatdiv.leftp.pink {
  -webkit-border-image: url(../images/chat_bubbles_pink_r.png) 10 19
13 10;
}
```

If you need to change the dimensions of the bubbles, you can do it by changing the values of the style *bubble* defined on the `iphone.css` file. The next segment of code shows the values defined by default:

```
body#chatdiv.bubble {
  margin: 10px 10px 0 0px;
  width: 80%;
  clear: both;
}
```

There's more...

Despite the fact that the *stylesheet* of the *UiUIKit* framework provides enough colors for the bubbles used in the chat interface, you can add your own. A simple PNG graphic file is used to build each bubble. The framework contains one of these images for each color. You can build your own images using software for handling graphics such as *GIMP* or *Photoshop*. You can even take one of these images provided by the framework and modify it by applying a different color of your selection. We recommend you to take a look at the `iphone.css` file, which is located at the `stylesheets` directory of the *UiUIKit* framework.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Creating a date picker

A *datepicker* is a widget designed to select a specific date choosing a month, a day, and a year from a predefined set of options. Using this widget, the developer can be sure to offer to the user a way for selecting a real date without errors. Another advantage of this widget is that it allows you the use of a predefined format for the date. Usually, some graphic interfaces are confusing for the user because it is difficult to find out which is the correct format required for the application. The date picker avoids this possible confusion using a clear way for selecting the components of a date: month, day, and year. Continue reading to find out how to create and use a date picker in our iPhone applications.

Getting ready

In previous recipes, we used some frameworks such as *UiUIKit*, *iUI*, and *iWebKit*. Now it's time to try out *Sencha Touch*. Be sure to check that you have installed this toolkit before continuing.

How to do it...

In order to get our data picker working, you'll need to follow some steps. We are going to start creating a new directory located inside the `DocumentRoot` of our web server. For instance, Mac users can execute the following lines from `Terminal.app`:

```
$ cd /Library/WebServer/Documents/
$ mkdir picker
```

The next step will be to create a new HTML file with the standard headers and with the required files of the *Sencha Touch* framework:

```
<!DOCTYPE html>
<html>
  <head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <title>Picker</title>
  <linkrel="stylesheet" href="../sencha-touch/resources/css/ext-touch.
css" type="text/css">
  <script type="text/javascript" src="../sencha-touch/ext-touch-debug.
js"></script>
```

It's time to add some references for our own CSS and JavaScript files that are required for building the widget:

```
<linkrel="stylesheet" href="picker.css" type="text/css">
<script type="text/javascript" src="picker.js"></script>
```

We don't need more code for the HTML file, so you can close the open tags by adding the next lines to your file:

```
  </head>
  <body></body>
</html>
```

As you observed, we added two different references so we need to create this file in the same directory as the HTML file. Let's start for the CSS file. Just add the following lines to this new file and save it as `picker.css`:

```
body {
  background: rgb(197,204,211);
  margin: 0;
  padding: 0;
}
```

Now, you are going to create a new JavaScript file called `picker.js` inside the same directory as `picker.css` and your main HTML file. After creating the file, you can add the following lines of JavaScript code:

```
Ext.setup({
  onReady: function(options) {
    vardatePicker = new Ext.DatePicker({
      yearFrom: 2000,
      yearTo: 2020,
      value: {
        day: 13,
        month: 6,
        year: 2011
      }
    });

    var panel = new Ext.Panel({
      fullscreen: true,
      layout: {
        type: 'vbox',
        align: 'center',
        pack: 'center'
      },
      items: [{
        xtype: 'button',
        ui: 'normal',
        text:'Click forDatePicker',
        handler: function() {
          datePicker.show();
        }
      }]
    });
  }
});
```

Surely, you're in a hurry to see how the date picker is displayed by the iPhone. Load your new HTML file and click on the button to see a screen shown as follows:

How it works...

Sencha Touch makes intensive use of JavaScript, so we always need to write code in this programming language for our iPhone applications. In this recipe, we built a button for launching a simple date picker allowing the user to choose a date by selecting month, day, and year for it. The *Sencha Touch* framework includes a special and predefined widget ready for use in our applications. Its name is *DatePicker* and is a regular JavaScript class with different attributes and methods.

All *Sencha Touch* applications require a JavaScript file for defining the user interface. This file must include a mandatory method called `Ext.setup`. The purpose of this method is to initialize the framework creating the widgets for the user interface. Also, it allows you to set many start up behaviors for your application. When the browser loads the main HTML file, the `onReady` method will be executed. Actually, the main JavaScript code should be inside it.

Our date picker widget will be created, thanks to the `DatePicker` object. Inside the `onReady` method, we're creating a new instance of this object, passing parameters as the default selected date for the widget, and the minimum and maximum dates allowed. You can pass other configuration parameters as the maximum and minimum height and width for it. All of them are passed to the constructor of the class, but the object can use other defined public methods for different actions. For instance, you can use the `hide` method for hiding the widget when a user clicks on a button. The `show` method will display your widget again. In order to find out more about this `DatePicker` class, we recommend you to take a look at the complete reference of the API documentation of the framework.

On the other hand, the last lines of our recipe stress creating a mandatory component of the *Sencha Touch* framework. In practice, we need a main *container* to set the widgets used by the application. In fact, this container holds the widgets that the application displays. Our recipe shows you one of these, which is called *Panel*. Inside our defined *Panel*, we need to configure the graphical layout and the required widget for the initial screen of our application. Even you can define some functions working as an event handler for the widgets. In our recipe, we're using a vertical and centred container through the `type` and `align` properties of the `layout` property. Regarding the trigger button for our date picker, we need to use the `item` property, which includes a `callback` function. It will call our date picker when the user clicks on the defined button. The code for this function is as follows:

```
handler: function() {
  datePicker.show();
}
```

Inside the function show, we can see how the method `show` of our `DatePicker` object is launched. It will display the widget on the screen. This widget contains a **Done** button to hide the widget and pick up the data typed by the user.

You can find the complete code for this recipe at the `code/ch02/picker/` folder in the code bundle provided on the Packtpub site.

See also

- *Installing Sencha Touch framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- Complete reference for the `DatePicker` JavaScript class is available at `http://dev.sencha.com/deploy/touch/docs/?class=Ext.DatePicker`

Using different tabs

Despite the fact that *tabs* are more popular for web and desktop applications, we can build a graphic interface with it for our iPhone applications. Usually, tabs are used as a container within a window or a page. It is a navigation widget for switching between a set of widgets used for grouping common functionalities. Formally, the area using tabs is known as **tabbed interface**. This recipe shows you how to build a basic widget with three different tabs.

Getting ready

As seen in the previous recipe for the date picker widget, we're going to use the *Sencha Touch* framework again. The reason for it is very clear; this framework offers a simple and useful widget to do that.

How to do it...

The first few steps for this recipe are very similar to the others shown in the previous recipe for the date picker. Let's start by creating a new directory under the control of the web server. For instance, Ubuntu Linux users will execute this command from the *gnome-terminal* application as follows:

```
$ sudomkdir /var/www/tabs
```

We recommend you to work on the HTML file created in the previous recipe. Just replace the lines referring to the CSS and the JavaScript files with our new widget. The code should be the following:

```
<linkrel="stylesheet" href="tabs.css" type="text/css">
<script type="text/javascript" src="tabs.js"></script>
```

Now we're going to create a new CSS file called `tabs.css` inside the same directory with the following code:

```
body {
  background: rgb(197,204,211);
}
card1, .card2, .card3 {
  background-color: #376daa;
  text-align: center;
  color: #204167;
  text-shadow: #3F80CA 0 1px 0;
  font-size: 1.5em;
  padding-top: 100px;
}
```

The last step for this recipe will be the creation of a new file saved as `tab.js`, located in the same `tabs` subfolder containing the following JavaScript code:

```
Ext.setup({
  onReady: function() {
    newExt.TabPanel({
      fullscreen: true,
      sortable: true,
      items: [{
        title: 'Tab 1',
        html: '<p>Content for tab 1</p>',
        cls: 'card1'
        }, {
        title: 'Tab 2',
        html: '<span>Here is the tab 2</span>',
        cls: 'card2'
        }, {
        title: 'Tab 3',
        html: 'Third tab',
        cls: 'card3'
      }]
    });
  }
});
```

Ready to check how is it displayed on your iPhone? Just open your page on your real iPhone device or on your *iPhone Simulator*. Maybe you're using a desktop web browser as *Google Chrome* or *Safari*. No problem, the final result will be as shown in the next screenshot taken from the *iPhone Simulator* running on *Snow Leopard*:

The complete code for this recipe can be found at the `code/ch02/tabs/` folder in the code bundle provided on the Packtpub site.

How it works...

The `Ext.TabPanel` is a JavaScript class defined in the CSS main file of the *Sencha Touch* framework. It's a container, which can hold other widgets to be accessed in a tabbed interface. We only need a few lines of JavaScript code for building an interface with tabs.

In this case, we used the *items* JavaScript's array object to define our three tabs. Each of them is using three different properties: `title`, `html`, and `cls`. The first one is used to set the title of each tab. This text will always be visible but its background color will be different when it is be selected. Don't worry about that; the widget manages this behavior by default. You don't need to write code to control what happens when the user clicks on each different tab.

The *html* element holds the content of the tab allowing any valid HTML code inside. Our example uses a HTML paragraph tag for one tab and *span* element for another. Even you can only set text without any tag as shown in our last tab. The last property of the array is *cls*. It's designed to set a specific CSS style for each tab. Despite that we're using a different name for the CSS class for each of them, the style used in the CSS file is the same for all. Obviously, you can define different styles for each tab. However, it's better to design a consistent user interface using the same CSS style for all tabs. The code of our recipe is using a bunch of CSS properties to define and configure the aspect of each tab. For instance, the CSS *background-color* property sets a color for the background of each of them. Other is *text-align*, which is used for indicating how the text should be aligned within each tab. In general terms, you can use any CSS style for your tabs without special restrictions.

Another configuration option used by the `Ext.TabPanel` object is the property `sortable`, which enables the sorting functionality of this widget. Thanks to it, the tabs are displayed in the same order as indicated by the *items* array.

The `Ext.TabPanel`class has more configuration options such as *width*—for configuring the width of the widget, *margin*—for setting a specific margin, and *height*—useful for setting the height of the widget. Also, a part of the constructor used in our code, you can execute other methods defined on the original JavaScript class. Some of them allow you to disable the widget or hide it. In order to find out more functionalities and features of this widget, you can visit the documented API offered at the website of the framework.

See also

- *Installing the Sencha Touch framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating a date picker* recipe
- Complete reference for the `Ext.TabPanel` JavaScript class is available at `http://dev.sencha.com/deploy/touch/docs/?class=Ext.TabPanel`

3
Events and Actions

In this chapter, we will cover:

- Identifying the devices
- Viewing applications in full screen
- Detecting full screen or browser mode
- Scaling to device width
- Preventing scaling
- Detecting one-finger events
- Detecting multi-touch events
- Preventing default behavior for events
- Detecting rotation events
- Implementing drag-and-drop
- Adding visual effects
- Running your web application without Internet access

Introduction

This chapter covers topics related to events and actions. Both are an important issue of the user interface because they allow a better control of the interaction between the user and the device.

The first few recipes of this chapter show you how to detect and respond to the main events launched by the browser where our application is a user input. Also, you'll learn how to apply some visual effects in order to get a richer graphic interface, helping to improve the interaction between users and the user's interaction with the device.

As we're focused on the iPhone, most of the recipes of this book can be used for developing applications for other iOS devices, such as the iPad and iPod Touch. Actually, all of these devices use the same operating system known as iOS; however, versions for these systems can be different. One of the recipes of this chapter shows how to indentify the type of the device and the current operating system's version. Maybe you want to provide a specific action based on the kind of the device running your application. For instance, an iPod Touch cannot send an SMS. This means that you can detect the device and inform the user if it is possible or not to send an SMS.

We should not forget that our intention is to develop web applications with a native look and feel. Some recipes will help teach you how to apply techniques to create an interface closer to what a native application provides. For example, providing a full screen mode for our applications will be very useful.

On the other hand, it could be a big advantage to use our web application without Internet access. The last recipe of this chapter shows how to do that using the HTML5 capabilities implemented by *Safari Mobile*.

Identifying the devices

Probably you're reading this book because you're interested in developing applications for the iPhone. However, other devices are running the same operating systems used by this smartphone. Actually, we have two more: iPad and iPod Touch. The first one is the most modern gadget designed by Apple. The other one is an advanced multimedia player with functionalities and features close to smartphones and PDA's. Because the devices have their differences, it could be useful to detect what device is using our application to do specific actions for each of them. This recipe shows you how to do that.

Getting ready

For this recipe, we don't need any framework. We're going to use only a bit of JavaScript code in a plain HTML file.

How to do it...

1. Open your favorite text editor and create a simple HTML file with standard headers. Add the following JavaScript code inside the `<head>` section of your file:

```
<script type="text/javascript" charset="utf-8">
  function init() {
    var agent = navigator.userAgent;
    varregex = new RegExp("^Mozilla/.*(iPhone|iPod|iPad).*(;
      U;).*$");
    var match = regex.exec(agent);
    if (match != null) {
```

```
        alert("This device is an " + match[1]);
      } else {
        alert("Sorry, this is not an iOS device");
      }
    }
</script>
```

For testing purposes, it will be useful to call this function when the page is loading. You can do that modification by adding the `<body>` tag to the `onload` event, as shown in the following line:

```
<body onload="init()";>
```

2. You're ready for loading your new page from your device. A new alert box will be displayed if the device is an iPhone, an iPod Touch, or an iPad.

How it works...

This recipe shows how to detect the kind of device based on the device type through the `navigator` object of the web browser. The property `userAgent` contains information about the client that is loading the web page. Most of the web browsers use this property for storing this kind of information, which will be different for each device. We used a simple regular expression to find the proper string.

Among the information contained in the `userAgent` property, we can find the version of the operating system of the device. If we need to use this information, we can change our regular expression to the following:

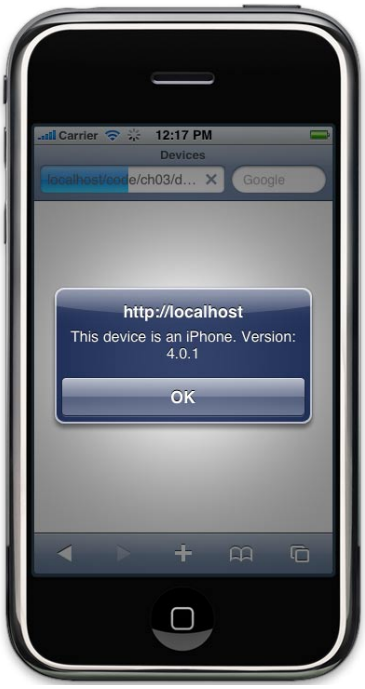
```
varregex = new RegExp("^Mozilla/.*(iPhone|iPod|iPad).*(; U;).*(OS )(\\
d_\\d_?\\d?)( like).*$");
```

For displaying the version, we can use the following code:

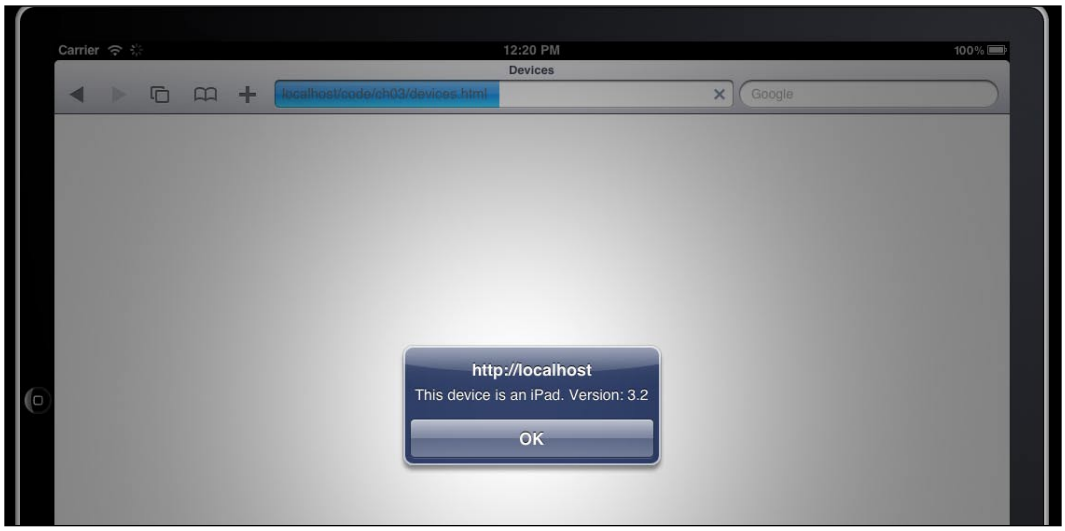
```
alert("Version: " + match[4].replace(/_/g, "."));
```

The code located at `code/ch03/devices.html` in the code bundle provided on the Packtpub site contains a complete example, identifying and displaying the kind of device plus the version of its operating system.

The following screenshots show the different information when the page is loaded by the iPhone and/or the iPad:



A similar information window in an iPad would look like the following screenshot:



Despite the fact that this recipe only shows an alert box with the mentioned information, you can write your own functions to launch different additional actions based on the device.

There's more...

If you're interested in finding out the complete information the `userAgent` property has to offer, you can display it using the following line of JavaScript code:

```
alert(navigator.userAgent)
```

See also

Instead of the standard `alert()` JavaScript method, you can use a customized one, as shown in the *Creating and customizing a notification box* recipe in *Chapter 2*, *Building Interfaces*.

Viewing applications in full screen

Obviously, web applications were designed to be viewed with a web browser. Keep in mind that we're developing non-native applications to look and feel like native applications, by using our knowledge of web technologies, such as HTML, CSS, and JavaScript. But we need a web browser for displaying these applications. In fact, we're using the *Safari Mobile* browser provided by the different devices. The problem is that this browser displays our web application like other conventional web pages. This means that the browser is displaying a URL bar at the bottom. But we don't want this effect. We need our application to be displayed in full screen as native applications. You'll learn how to get full screen mode after reading this recipe.

How to do it...

You can use the file created in the previous recipe or just create another basic HTML file with standard headers and `<body>` structure. Inside the `<head>` section of the file, you should add the following line:

```
<meta name="apple-mobile-web-app-capable" content="yes" />
```

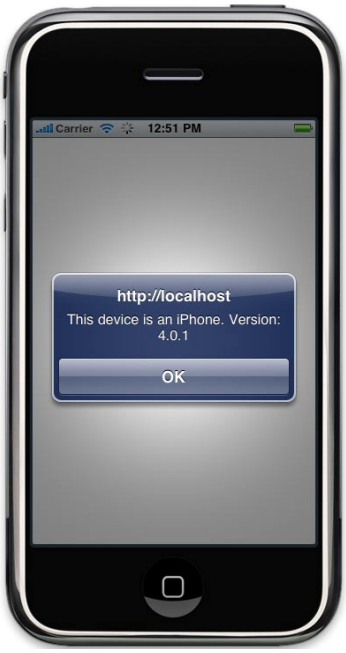
How it works...

If you reload your new page, it will be displayed without any changes by the web browser. In other words, you don't have full screen mode yet. Think about how native applications are loaded. We're using small icons for launching it from the home screen of the device. Well, we want the same effect for our web applications with a native look and feel. Don't worry, you can do it using the bookmark functionality of Safari. Click on the plus icon at the bottom. Now, you can see three different buttons as shown in the following screenshot:

Click on **Add to Home Screen** and you'll see a new dialog screen asking you for a name to identify and access your web page.

After choosing a new name, push the **Add** button at the bottom of the screen. Congratulations! You now have a new icon on the home screen. When the user clicks on the new icon from this screen, the application will be launched by Safari in full screen mode, thanks to the detection of the `apple-mobile-web-app-capable` tag.

After following the earlier mentioned process, you can see your application in full screen mode, as shown in the following screenshot:



In the next chapter, we'll learn how to choose and add our own icon for launching our application from the home screen.

See also

- Choosing an icon image for the application in *Chapter 4*, *A Picture Speaks a Thousand Words*

Detecting full screen or browser mode

As you've learned in the previous recipe, an application can run in full screen or in browser mode. By default, web applications will run on the second one, that is, in the browser mode. Because it's possible to run applications in these two modes, it could be useful to detect which mode is running. Let's explore how to do it.

How to do it...

1. We're going to use only JavaScript code to detect the running mode for our application. You can use a simple HTML page with the following code inside the `<head>` section:

```
<script type="text/javascript" charset="utf-8">
  var mode = 'browser';
  if (navigator.standalone) {
    mode = "standalone";
  }
  alert("App. running in " + mode + " mode");
</script>
```

   For testing our JavaScript code, we'll add the following `onload` attribute to the `<body>` tag:

```
<body onload="init()";>
```

2. Don't forget to close the `<body>` and `<html>` tags properly. The complete code for this recipe can be found at `code/ch03/modes.html` in the code bundle provided on the Packtpub site.

How it works...

The `navigator` JavaScript object provided by Safari contains a special property called `standalone`. It allows you to detect the running mode of a web application. This recipe uses a simple alert box for displaying this information. However, you can write your own JavaScript function and invoke it when you need to detect this mode. Inside the function, we'll do something more sophisticated than displaying an alert box.

See also

- *Creating and customizing a notification box* recipe in *Chapter 2*, *Building Interfaces*
- *Viewing applications in full screen* recipe

Scaling to device width

The previous chapter covered the fundamentals of the graphical user interface for the iPhone. Refreshing your mind, we described the viewport as the rectangular area determining how the content of the application displays on the device. This one is an important concept to keep in mind when you're building web applications that have a native look.

How to do it...

We only need one line of code in our HTML files, inside the `<head>` section:

```
<meta name="viewport" content="width=device-width" />
```

How it works...

The viewport area can be controlled through the `viewport` `meta` tag defined by the *Safari Mobile* browser. The `width` property sets the available width in pixels for the area. Using the value `device-width`, we can use all available space in the viewport.

Instead of the `device-width` value for the `width` property, you can change the width setting to a different numeric value expressed in pixels. By default, iPhone and iPad use 980 pixels for the width so `device-width` is a constant equal to this value.

See also

- *Preventing scaling* recipe

Preventing scaling

By default, Safari allows you to zoom in and out of the page that is displayed inside the mentioned viewport. However, native applications don't allow this behavior by default. Keep reading to learn more about how to avoid it.

How to do it...

Once more, we're going to use a simple HTML line inside the `<head>` section:

```
<meta name="viewport" content="user-scalable=no" />
```

How it works...

The previous recipe showed how to set the width for the `viewport`, but this area can be managed through additional parameters. One of them is `user-scalable`, which controls whether the user can zoom or not. In this case, we're using the `no` value to indicate that scaling zooming is not allowed. Actually, this is the default feel for native applications.

In addition to these properties, other parameters such as `minimum-scale` and `maximum-scale` can be used to control the viewport of the devices. Both of them use a float value and it is `1.0` by default.

In practice, we'll use the `viewport meta` tag with these mentioned values for getting a closer native look and feel for our iPhone web applications. As you have seen in the previous chapters, we used this line of code in most of our recipes. The complete line of code used by our applications is the following one:

```
<meta name="viewport" content="width=device-width, user-scalable=no" />
```

See also

- *Scaling to device width* recipe

Detecting one-finger events

Users interact with the iPhone through a touch screen. No physical keyboard or mouse is available. The fingers serve as both the pointer and the input device. When the user touches the screen, a single event is launched. *Safari Mobile* defines different types of events, which capture and respond to the movements and touches of the user. We can respond to a one-finger or multi-touch finger event because the iPhone is a multi-touch device. In this recipe, we find out how to detect when the user is using one finger to interact with the device.

Getting ready

We are going to use the *iWebKit* framework for building a simple user interface to detect the events.

How to do it...

1. Create a standard XHTML file with the following lines inside the `<head>` section:

```
<meta name="apple-mobile-web-app-capable" content="yes" />
<meta name="viewport" content="width=device-width, user-
scalable=no" />
<linkhref="../iwebkit/css/style.css" rel="stylesheet"
media="screen" type="text/css" />
<scriptsrc="../iwebkit/javascript/functions.js" type="text/
javascript"></script>
```

2. The second step will be to add three different JavaScript functions:

```
functiontouch_start(event) {
  alert("Your finger touched the screen");
}
functiontouch_end(event) {
  alert("Your finger was removed from the screen");
}
functiontouch_move(event) {
  alert("Moving!");
}
```

3. Let's go to the `<body>` tag. We need to add some event handlers as follows:

```
<bodyontouchstart="touch_start(event);"
  ontouchend="touch_end(event);"
  ontouchmove="touch_move(event);">
```

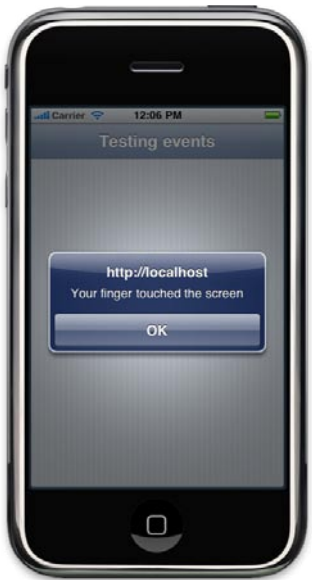
4. The last step for this recipe will be to create a standard toolbar using the styles defined by the *iWebKit* framework. In order to do this, you only need the following lines of HTML code:

```
<div id="topbar">
  <div id="title">Testing events</div>
</div>
```

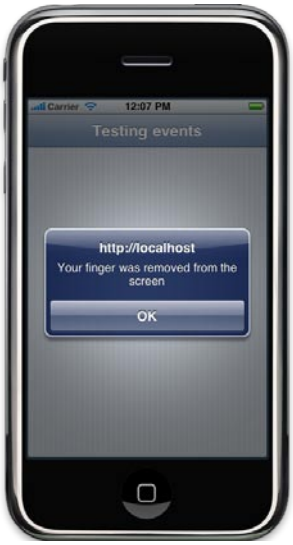
Don't forget to close the `<body>` and `<html>` tags properly. The complete code for this recipe can be found at `code/ch03/one_finger.html` in the code bundle provided on the Packtpub site.

How it works...

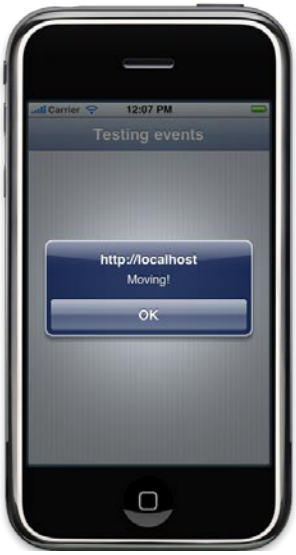
When the user touches the screen, the browser launches a JavaScript event called `touchstart`. In fact, this event happens every time the user places a finger on the screen. Using the `ontouchstart` handler on the body tag, we're indicating that a JavaScript function should be invoked when this event happens. In this case, the `touch_start(event)` function will be launched. Inside this function, we simply have an `alert()` function for displaying a message indicating that the user has touched the screen. Actually, when this action is taken, the user will see a message as shown in the following screenshot:



In a similar way, when the user removes the finger from the screen, the `touch_end(event)` function is called and a new message is displayed. For handling this event, we used the `ontouchend` attribute for the `<body>` tag. The following screenshot shows the message that appears when this action is carried out by the user:



The other attribute we added to the `<body>` tag – `ontouchmove` – handles the callback function, which is launched when the user moves one finger across the screen. The result of invoking this function called `touch_move(event)` will be an alert box with the message **Moving!**, as shown in the following screenshot:



All of these functions are using the same parameter, `event`. This event is a JavaScript object belonging to the `TouchEvent` class, which encapsulates all the information about the touch event.

This recipe is very simple and only shows how to detect the events related to one-finger events. In the real world, you'll want to write some JavaScript code in order to respond to the many type of events.

There's more...

Apart from the events shown in this recipe, *Safari Mobile* defines other events related to one-finger touch. One of these is `touchcancel`, which happens with the systems cancel tracking for the touch. It can be detected in a similar way to others using the `ontouchcancel` handler.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Detecting multi-touch events* recipe
- *Preventing the default behavior for events* recipe

Detecting multi-touch events

The previous recipe shows how to detect events for one-finger events, but one of the most celebrated features of the iPhone is the multi-touch screen. Using more than one finger, users can do different actions in a very intuitive and easy way. For instance, zoom over a picture is one of these actions. This action requires using only two fingers, but other more complicated actions can use more fingers. One example of this could be a game for playing a piano.

This recipe shows the difference between one-finger and multi-touch events and how we can detect it.

Getting ready

For simplicity, we're going to continue using the *iWebKit* framework. You can reuse the file created in the previous recipe.

How to do it...

1. Open your XHTML file—created in the previous recipe—with your favorite text editor and change the body element, replacing the old attributes for new ones, as shown in the following line:

```
<body ongesturestart="gesture_start(event);">
```

2. Now it's time to write a simple JavaScript function to respond to this event:

```
function gesture_start(event)
  alert("Two or more fingers touched the screen");
}
```

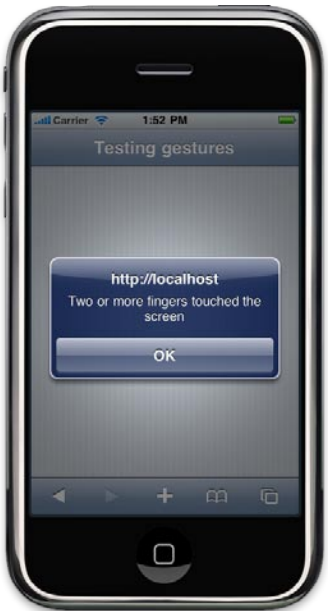
3. At this point, we can detect when the user touches the screen with one or more fingers. Also, you can detect if the user moves the finger across the screen. For this case, we're going to replace the attribute `ongesturestart` for `ongesturemove`. This action is very simple, you only need the next line:

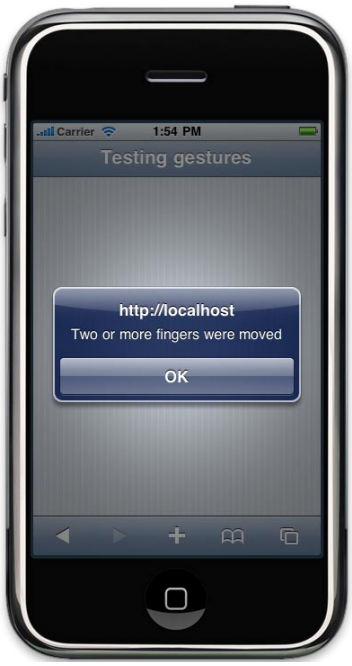
```
<body ongesturechange="gesture_change(event);">
```

4. Obviously, one handler is required to respond to this event, so we'll use the following JavaScript function:

```
function gesture_change(event) {
  alert("Two or more fingers were moved");
}
```

5. The next two screenshots show different messages displayed for each event:





You'll find the code for this recipe at: `code/ch03/multi-touch.html` in the code bundle provided on the Packtpub site.

How it works...

From the technical point of view, the class `GestureEvent` encapsulates the information about a multi-touch event. Also the objects belonging to this class encapsulate `TouchEvent` objects. This situation defines two levels:

1. High for `GestureEvent`
2. Low for `TouchEvent`

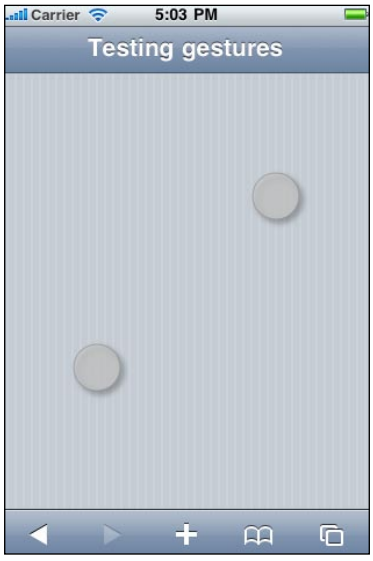
Therefore, both of these are triggered when a multi-touch event occurs. Actually, the sequence for multi-touch events combines both kinds of classes. One clear example of this process is when the user makes use of a two-finger swipe across the screen. In this case, the complete sequence of events is as follows:

1. The user touches the screen using one finger and then the `touchstart` event is launched.
2. The second finger goes to the screen and the `gesturestart` event is launched.
3. Immediately after the `gesturestart` is launched, a new `touchestart` event is launched for the second finger.

4. At this point, both fingers are touching the screen. If the user moves the fingers, then the `gesturechange` event is launched.
5. When the user stops the movement of the fingers, a new `gestureend` event is launched.
6. When one of the fingers is lifted from the screen, the `touchend` event is launched for this finger. This event happens immediately after `gestureend`.
7. Finally, when the second finger is lifted from the screen, the system launches `touchend` for completing the sequence.

The events-launching mechanism offers flexibility to developers because they can program some actions as a response to these events, taking a major fine-grained control over the multi-touch screen. In practice, using the detection and control of these events, we can enable visual effects such as drag-and-drop.

Developers using the iPhone Simulator can use a shortcut to simulate the multi-touch. By clicking on the *Alt + Apple* key, two circles will be displayed on the screen. If you hold these keys and move your mouse simultaneously, then the circles will be moved. You can move the circles across the screen. Release your fingers and the circles will disappear.



There's more...

Another event that you can detect is `gestureend`. It happens when the gesture ends, usually when any finger or only one of them is touching the screen. The handler for the HTML tag is `ongestureend` and it uses `event` as the parameter to identify the object encapsulating the right physical event.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Detecting one-finger events* recipe
- *Preventing the default behavior for events* recipe

Preventing the default behavior for events

Remember that our applications are run in Safari, which is a web browser and events are handled in a different way than in native applications. In order to implement interfaces for our web applications that are similar to native ones, we'll need to disable the default behavior for these kind of events. This recipe explains how to prevent such behaviors.

How to do it...

1. First, we need to write one JavaScript function as follows:

```
function avoid_behavior(evt) {
  evt.preventDefault();
  }
```

2. The second step will be to write some code for invoking the previous JavaScript function. Actually, this function will be called for different events. Use the following `<body>` tag for the HTML pages of your application:

```
<body
  ontouchmove="avoid_behavior(event);"
  ontouchstart="avoid_behavior(event);"
  ongesturestart="avoid_behavior(event);"
  ongesturechange="avoid_behavior(event);">
```

How it works...

Safari Mobile offers a simple method for disabling the default behavior for events. We only need to handle each event calling to a function, which will invoke to this default method. Using the `<body>` tag, we get to avoid this behavior when the page is loaded.

In practice, using the function of this recipe, we avoid effects such as the elastic scrolling. By default, Safari allows you to flick the content of the viewport. The problem is that native applications do not use this behavior and we need to avoid it to achieve a native look and feel similar to these ones.

The approach of this recipe is framework agnostic, so you can use it for your applications with independence of it. This means that it is possible to combine it with any frameworks described in the *Chapter 1*, *Frameworks Make Life Easier*. For instance, you can use the *iWebKit* framework as used in the previous recipe.

See also

- *Chapter 1*, *Frameworks Make Life Easier* and *Chapter 2*, *Building Interfaces* for choosing and using one of the frameworks described.

Detecting rotation events

Thanks to the hardware accelerometer incorporated in the iPhone, it is possible to rotate the content of the viewport. When the user rotates the physical device, by default, the layout of the application rotates as well. Let's see how to detect this kind of movement in our applications.

Getting ready

This recipe doesn't require any framework; despite this, we're going to continue using the *iWebKit* so we may reutilize our code.

How to do it...

1. One more time, take your base XHTML file for *iWebKit* and modify the `<body>` tag to include the `onorientationchange` attribute. The line should be changed to the following:

```
<body onorientationchange="detect_orientation()">
```

2. For detecting and responding to the event rotation, we'll use a simple and useful JavaScript function as follows:

```
function detect_orientation() {
  alert("Orientation has been changed");
}
```

3. Be sure to close your XHTML file properly with the convenient standard tags. Take a look at the complete code for this recipe at `code/ch03/orientation.html` in the code bundle provided on the Packtpub site.

How it works...

The code for this simple recipe captures the event, which is launched when the user rotates the device. For this purpose, the `onorientationchange` handler is used inside the body tag. The `detect_orientation()` callback function will be executed as a response to the event. It will display a simple alert box with a message indicating that the event was launched.

In order to test this recipe, you can load your new page directly on the iPhone. As shown in the following screenshots, each time you rotate the device, a new alert box will be displayed. The following screenshots show different messages responding to these rotations:

We can use a more sophisticated mechanism for displaying this information; despite this, we chose a common and native alert box provided by the web browser.

Maybe you're wondering why it can be useful to detect the rotation event. By default, the device autorotates the content, so for some applications you'll need to adjust the elements of the user interface. In some cases, the automatic adjustment made by the device will be sufficient and the content displayed will be indented properly. On the other hand, detecting the rotation could be required by those applications, which use the features and capabilities of the accelerometer. For instance, many games are making intensive use of this component and it is becoming very important to properly detect the rotation for responding to this action differently and then respond accordingly.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Detecting current orientation* recipe in *Chapter 9*, *Location, Location, Location*

Implementing drag-and-drop

We can implement a graphic interface with drag-and-drop components from scratch, simply by detecting the different events over the screen and writing the code and responding to them. Fortunately, frameworks were designed to make our life easier, so we're going to use one of them for implementing this behavior in our web applications. Let's see how to write a small example with two rectangles. One of them will serve as an area for dropping and the other one will be the draggable component. Each will have a unique color and colors being the draggable component a smaller will be smaller. Obviously, the draggable rectangle can only be dropped inside the bigger one. This recipe explains the required code to implement this effect using the capabilities of the *Sencha Touch* framework.

Getting ready

Be sure you've installed *Sencha Touch* framework under the control of your web and can access it in your browser.

How to do it...

1. The first step for this recipe will be to create a new folder inside the `DocumentRoot` of your web server. This new directory will contain three different files. Let's start writing the first one called `index.html`. It is a XHTML file defining the two main rectangles.

2. The following lines load the required files for the *Sencha Touch* framework, including the standard headers:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<htmlxmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=utf-8">
    <title>Drag and Drop</title>
    <linkrel="stylesheet" href="../sencha-touch/resources/css/ext
      -touch.css" type="text/css" />
    <script type="text/javascript" src="../sencha-touch/ext-touch
      -debug.js"></script>
    <linkrel="stylesheet" href="main.css" type="text/css" />
    <script type="text/javascript" src="main.js"></script>
  </head>
```

3. Now, we need to define the main rectangles inside the `<body>` tag:

```
  <body>
    <div id="droppable">Drop area</div>
    <div id="draggable">Drag me!</div>
  </body>
</html>
```

4. The second file is a CSS stylesheet defining the properties of the rectangles. Actually, we need to choose the position, dimensions, and colors. This is the code for `main.css`, which was referred in the XHTML file:

```
#draggable {
  width: 100px;
  color: white;
  height: 100px;
  background-color: blue;
  top: 170px;
  left: 5px;
  position: absolute;
  padding-top: 40px;
  text-align: center;
  border-bottom: 1px solid;
}
#draggable.x-dragging {
  background-color: black;
}
#droppable {
  text-align: center;
  color: white;
  border-top: 1px solid;
  padding-top: 60px;
  height: 150px;
```

```
  width: 210px;
  background-color: red;
  position: absolute;
  left: 5px;
  top: 5px;
}
```

5. The third file is the `main.js` JavaScript main file required by *Sencha Touch*, which defines the *Sencha Touch* interface. Remember that applications using this framework must define a main custom JavaScript file for initializing it and defining the initial behavior of the components. The code inside our file, named `main.js`, is as follows:

```
Ext.setup({
  onReady: function(){
    newExt.util.Draggable('draggable', {
      revert: true
    });
    newExt.util.Droppable('droppable', {
      validDropMode: 'contains',
      listeners: {
        drop: function(droppable, draggable, e) {
          draggable.el.setHTML('Dropped!');
        }
      }
    });
  }
});
```

6. Now we're ready to test our new web application with drag-and-drop implemented.

7. The required code files for this recipe are located at `code/ch03/drag_drop/` in the code bundle provided on the Packtpub site.

How it works...

The HTML interface for this recipe is pretty simple: two rectangles with different colors and with absolute positioning. The bigger rectangle represents the droppable area where the other rectangle can be dropped. The smaller rectangle is blue and located at the bottom of the page. Both of them are simple `div` standard HTML elements with some CSS styles applied. We defined colors, width, and positioning in the main `.CSS` file for our recipe. Apart from this, the CSS file also defines different colors to be used when the user interacts with the rectangles. For instance, if the user clicks-and-drags the smaller blue rectangle, its color changes to grey and when this element is successfully dropped inside the larger rectangle area, the larger rectangle changes its color from red to blue.

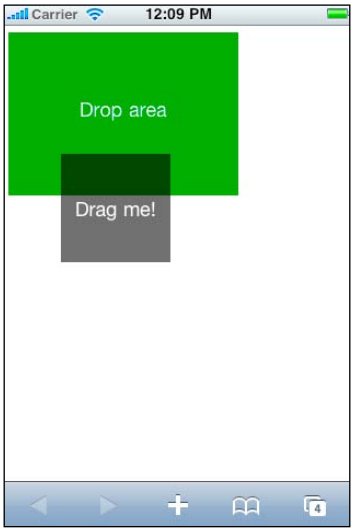
Regarding the `main.js` file, it is using the `Ext.setup` method to initialize the framework and for defining the behavior of our rectangles. We need to indicate which rectangle is draggable and which is droppable. It's simple, thanks to two predefined classes implemented in *Sencha Touch*. These are `Ext.util.Draggable` and `Ext.util.Droppable`. In our code, we're using the `id` of each `div` element as the first parameter of the constructor of each class. This means that the `div` with `id` equal to droppable – the bigger rectangle – will be an object of the `Ext.util.Droppable`. On the other hand, the draggable element will be an instance of the `Ext.util.Draggable` class and the value for the `id` will be `draggable`.

The constructor of the class `Ext.util.Draggable` is using the property `revert` to indicate that the rectangle will come back to its original position, if it isn't properly dropped inside the specific boundaries. You can change this behavior by setting the value to `false` for this property. By default, the action will start when the user holds the element, but we can indicate a delay in milliseconds using the delay property.

With respect to the constructor method for the `Ext.util.Droppable`, we're using a property called `validDropMode` for considering how the drop is used to confirm the placement of the dropped item. With the `contains` value for this property, we're indicating that the drop action will be valid only if the user places the smaller rectangle inside the bigger one. The other value for this property is `intersect`, which is used when we only want to intersect both elements to execute the action. In addition to this property, we used the `listener` object, which contains the event handlers to be added to the object during the initialization of the action. Our recipe simply changes the text of the draggable element to dropped! For executing this action, we're accessing the content of the `div` element through the `setHTML()` method provided by *Sencha Touch*.

The following screenshot shows the starting point:

When the users holds the small rectangle for dropping it, its color changes to grey as shown in the next screenshot:



The final result of the drag-and-drop action is shown the following screenshot:

There's more...

Despite the fact that we only use two simple rectangles for implementing drag-and-drop with basic behavior, the classes provided by *Sencha Touch* offers a set of methods and properties for building more sophisticated interfaces.

See also

- *Installing the Sencha Touch framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- API reference for `Ext.util.Draggable` class at `http://dev.sencha.com/deploy/touch/docs/?class=Ext.util.Draggable`
- API reference for `Ext.util.Droppable` class at `http://dev.sencha.com/deploy/touch/docs/?class=Ext.util.Droppable`

Adding visual effects

Sometimes it can be awesome to add visual effects to our applications. Usually, these kind of effects are implemented using animations. Basically, an animation is a set of movements made by a widget on the screen. Combining these movements with color changes, we can apply visual effects to the user interface of our web applications. The most popular of these animations or effects are fade and slide. Both of them display different elements on the screen showing one element and hiding another one that is applying a transition.

In this recipe, you'll learn how to apply simple visual effects in your web applications to improve the user experience.

Getting ready

We're continuing with *Sencha Touch* because this toolkit implements some visual effects, which are ready-to-use for some elements of the user interface.

How to do it...

1. Let's start creating a new directory called `effects` inside the `DocumentRoot` of our web server. Then you'll create a new XHTML file called `index.html` inside this new directory.
2. The content for this new file is simple and contains the following lines:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<htmlxmlns="http://www.w3.org/1999/xhtml">
  <head>
```

```
    <meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
    <title>Slide Effect</title>
    <linkrel="stylesheet" href="../sencha-touch/resources/css/ext-
touch.css" type="text/css" />
    <script type="text/javascript" src="../sencha-touch/ext-touch-
debug.js"></script>
    <linkrel="stylesheet" href="main.css" type="text/css" />
    <script type="text/javascript" src="main.js"></script>
  </head>
  <body></body>
</html>
```

3. The next step will be to create a new CSS file–`main.css`–with the following lines:

```
body {
  background: rgb(197,204,211);
}
.tab {
  background-color: #78bcf0;
  text-align: center;
  color: #204167;
  font-size: 1.5em;
  padding-top: 100px;
}
```

4. Finally, we need to add a new main JavaScript file required by *Sencha Touch*. For this recipe, the file will define a graphic interface based on tabs. The required code is the next one:

```
Ext.setup({
  onReady: function(){
    newExt.TabPanel({
      fullscreen: true,
      sortable: true,
      items: [
        {title: 'Cu',
        html: '<p>Tab 1. Cube</p>',
        cls: 'tab',
        animation: {
        type: 'cube',
        duration: 400}
```

```
        },
        {title: 'P',
          html: '<p>Tab 2. Pop</p>',
          cls: 'tab',
          animation: {
          type: 'pop',
          duration: 400}
        },
        {title: 'Fa',
          html: '<p>Tab 3. Fade</p>',
          cls: 'tab',
          animation: {
           type: 'fade',
           duration: 600}
         },
         {title: 'Fl',
           html: '<p>Tab 4. Flip</p>',
           cls: 'tab',
           animation: {
           type: 'flip',
           duration: 400}
         },
         {title: 'Sl',
             html: '<p>Tab 5. Slide</p>',
             cls: 'tab',
            animation: {
            type: 'slide',
            duration: 400}
          }
        ]
      });
  }
});
```

5. Load this new XHTML page on your iPhone and click on different tabs for displaying different visual effects applied when each tab is loaded.

6. All files for this recipe can be found at: `code/ch03/effects/` in the code bundle provided on the Packtpub site.

How it works...

Sencha Touch uses a main JavaScript file for defining the user interface and its behavior. This is the reason for using an empty HTML file for this recipe. However, this file needs to load the main JavaScript and CSS files, including our application-specific JavaScript file with the required code for defining our user interface.

The CSS file for our application is very simple. It only contains a specific background color for the page and some properties for the tabs as the color, size, and alignment of the text and the background color.

Regarding the main JavaScript file for our application, it defines a widget based on five different tabs. A different visual effect is applied to change to the active tab each time the user clicks on one of them. Specifically, we're using the effects `cube`, `pop`, `fade`, `flip`, and `slide` predefined by *Sencha Touch*. Each of these effects is defined by the object `animation`, which uses two parameters. The first one indicates the predefined effect and the second one is the duration of the transition expressed in milliseconds. We chose 400 milliseconds except for the `fade` value. For this one, we're using 600, but these are arbitrary values. You can choose the ones you consider the best for your purposes. Inside each tab, we're displaying a simple text indicating the number of the tabs and the effect applied during the transition. On the other hand, the `cls` property of each tab is used to indicate which style should be applied. In this recipe, we used the same value for all tabs.

We must keep in mind that these effects defined and implemented by *Sencha Touch* can only be applied to container objects using a `CardLayout` for the layout. This is the reason for using an object of the class `TabPanel` as our main container, which uses this layout by default. The advanced user can design a customized object using a layout such as this one. Also, it's possible to use whichever class defined by *Sencha Touch* that uses this specific kind of layout. For instance, `NestedList` is another container using `CardLayout`.

There's more...

Apart from the effects shown in this recipe, the *Sencha Touch* framework implements others as a different kind of slide or the wipe effect. If you're interested, you can take a look at the complete documentation offered by this framework.

See also

- *Installing the Sencha Touch framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Using different tabs* recipe in *Chapter 2*, *Building Interfaces*
- Reference about CardLayout class at `http://dev.sencha.com/deploy/touch/docs/?class=Ext.layout.CardLayout`
- Reference about animations defined by *Sencha Touch* at: `http://dev.sencha.com/deploy/touch/docs/?class=Ext.anims`

Running your web application without Internet access

We're building web applications with a native look and feel, we have to keep in mind that they are running over the Internet. This means our potential clients will need an Internet connection to access our applications. However, this is an important difference between web applications and native applications. Here is where cache comes to the rescue. The new HTML5 standard defines a new mechanism for running web applications without Internet access. Using this technique our web iPhone application will behave closer to a native application.

Let's explore the mechanism for running your web application offline.

Getting ready

Be sure you've installed the *iWebKit* framework because we're going to use it to illustrate how to cache the files of our application. Also, we need to use a web server as Apache or lighttpd.

How to do it...

For this recipe, we'll start from scratch instead of reusing a file previously created.

1. Open your favorite text editor and add the following lines:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
  <html manifest="cache-mf"
    xmlns="http://www.w3.org/1999/xhtml">
    <head>
      <meta content="text/html; charset=utf-8" http
        -equiv="Content-Type" />
      <meta name="apple-mobile-web-app-capable" content="yes"
        />
      <meta name="viewport" content="width=device-width, user
        -scalable=no" />
      <linkhref="../iwebkit/css/style.css" rel="stylesheet"
        media="screen" type="text/css" />
      <scriptsrc="../iwebkit/javascript/functions.js"
        type="text/javascript"></script>
      <title>Offline</title>
    </head>
    <body>
      <div id="topbar">
      <div id="title">Offline</div>
      </div>
    </body>
```

2. After adding the code, save the file as `offline.html` inside your `DocumentRoot` directory of the web server. We suppose the required files for the *iWebKit* framework are inside a directory called `iwebkit`, which is inside the `DocumentRoot` directory. For instance, using Ubuntu Linux and Apache, we'll have the following path structure:
   - `/var/www/iwebkit/`: It is the directory containing the required files for the *iWebKit* framework.
   - `/var/www/offline.html`: It is the main XHTML file for this recipe.
3. Now, create a new text file with the following lines and save it as `cache-mf` inside the same `DocumentRoot`:

```
CACHE-MANIFEST
offline.html
../iwebkit/css/style.css
../iwebkit/javascript/functions.js
```

4. Great! Now your application is ready to work offline.

How it works...

We used the `manifest` attribute of the `<html>` tag with the `cache-mf` value. It indicates to the web browser that there exits a file called `cache-mf` in the same directory that the HTML file is located. You can observe we're using a standard XHTML file, but it will work with an HTML file as well.

The `cache-mf` is a plain text file containing information about the file, which should be cached for the web browser. Actually, it will contain one line per file, which will be cached. Each text line is a path to the specified file. By default, the path is relative to the directory where the application is running. Also, it is possible to use absolute path using the complete URL for each file. Don't forget to include the `CACHE-MANIFEST` header at the top of the file.

In this recipe, we used the *iWebKit* framework with a simple HTML file for the main user interface. As we don't need anything more, only `style.css` and `functions.js` are required and they will be cached with `offline.html`. Obviously, in the real world, your application will use and require more files such as images and other CSS and JavaScript files. Caching additional files is pretty easy; just add a new line per file to `cache-mf`. Maybe you want to use another name for the file, no problem. Just use a filename you prefer and set it as `value` in the `manifest` attribute of the `<html>` tag.

It is important to keep in mind what happens when the application files are upgraded. If the application is cached, users may still be running an older version. The browser will detect new files, but it will keep running the old ones, even though the browser detects the new files. To run the new version of the application, users must relaunch it. In fact, the browser downloads the new files in the background, but it is using the old files until the user decides to relaunch the application. You can understand this behavior easily by thinking about desktop applications and updates. When you launch a desktop application, it checks if updates are available and proceeds to download them. In this scenario, your application is still running an older version. Although the updates are downloaded and installed, you need to relaunch your application to use it. The cache mechanism and behavior explained earlier for our web applications will work in a similar way.

One of the main advantages of using cache for web applications is that the mechanism is transparent for the users. They don't need to configure anything; the browser does that automatically by handling the caching for them.

The mechanism used for this recipe is not designed specifically for iPhone or for *iWebKit* browser; it should work on every web browser able to understand and process HTML5 content. In fact, some desktop browsers such as Firefox will ask you for confirmation before storing some files in your local drive in order to enable caching.

There's more...

In addition to `CACHE-MANIFEST` header, you can use another header defined by the HTML5 standard. For instance, using `NETWORK` and a new list of files after it, you could indicate that these files won't be cached. This means that the browser will always ask for a new version of them to the Web when the user visits the page.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- Complete information about the cache mechanism defined by HTML5 can be found at `http://www.w3.org/TR/offline-webapps/`

4

A Picture Speaks a Thousand Words

In this chapter, we will cover:

- Choosing an icon image for the application
- Specifying a splash image
- Displaying an image inside a container
- Creating a grid with images
- Creating a carousel for images
- Rotating images
- Scaling an image by applying animations
- Taking and displaying pictures
- Drawing geometric figures
- Applying colors
- Working with gradients
- Adding an activity indicator

Introduction

Undoubtedly, images and graphics are very important for the user interface of any kind of application. Web applications for iPhone aren't an exception. In fact, we can use many different kinds of images, from photos taken with the iPhone's camera, to geometric figures, or even maps.

This chapter focuses on how to use graphics and images to improve the user's experience with their iPhone and our applications.

The chapter shows the fundamentals of graphic and images for iPhone applications, and then goes ahead to teach you how to build more sophisticated interfaces. Take the chapter as a starting point because the options for building web applications for iPhone, using images and graphics, are infinite.

Choosing an icon image for the application

Users need a method for launching applications from their device. *iOS* offers different panels with icons on each screen. Each of these icons is a direct link for launching the application. When you install a native application, the operating system creates an icon for it. After that, the user can move the icon to a different panel. Usually, developers provide an icon for their application and the system uses this icon; however, we're developing web applications. Remember we are striving for a native look and feel, so we want to imitate the native behavior. In this recipe, we'll learn how to provide an icon for our web application, so that the user can launch and execute our application by simply touching our icon.

Getting ready

We are going to use the *UiUIKit* framework for this recipe, but you actually don't need any framework to create an icon image for your application: *iOS* and *Safari Mobile* offer you a simple method for creating this kind of icon.

How to do it...

The first task will be to create a standard XHTML file with required headers for the *UiUIKit* framework.

1. Open your favorite text editor and add the following lines of code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta content="yes" name="apple-mobile-web-app-capable" />
  <meta content="text/html; charset=utf-8" http-equiv="Content}
        -Type" />
  <meta content="minimum-scale=1.0, width=device-width,
        maximum-scale=0.6667, user-scalable=no" name="viewport" />
  <meta name="apple-mobile-web-app-capable" content="yes" />
  <link rel="stylesheet"
        href="../uiuikit/stylesheets/iphone.css" />
```

2. Save this file as `icon.html` inside a directory; this is accessible by the web server.
3. For our icon we'll need to use a PNG graphic file, so you can choose one or create it using a graphic editor such as GIMP or Adobe Photoshop. Copy your file to the same directory as your XHTML file and rename it as `apple-touch-icon.png`. The next step will be to add the following line to your `icon.html` file:

```
<link rel="apple-touch-icon" href="apple-touch-icon.png" />
```

4. Finally, add the following lines of code and save your file again:

```
</head>
  <body id="normal">
    <div id="header">
      <h1>Icon</h1>
    </div>
  </body>
</html>
```

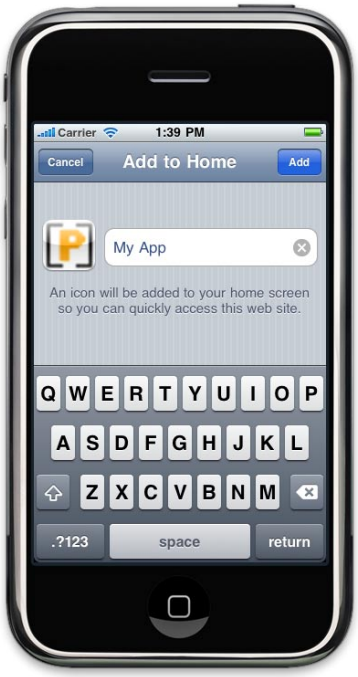
How it works...

In order to use an icon for launching our application from the home screen of our device, we'll use the capabilities of the *iOS* operating system and of *Safari Mobile* web browser. Refresh your mind by taking a look at the previous chapter; there you can find a recipe for getting full screen for our application. This previous recipe describes how to use the **Add to Home Screen** button for adding a link for our application on the home screen. However, we didn't choose any icon, therefore the operating system chooses one by default. It was a blank rectangle with a glossy effect applied, as you can see in the following screenshot:

Because we want to avoid the behavior explained before, we'll need to create or obtain an icon for our application. In any case, the graphic file must be in PNG format and its width and height must be the same. In other words, the image should be squared. Regarding dimensions, we can use 114x114 pixels or 128x128 pixels. Nevertheless, this is only a recommendation because the operating system will automatically resize our icon to the fixed dimensions required by each device. For instance, iPad uses 72x72 pixels while iPhone uses 56x56 pixels. As these devices use different dimensions, it is a good idea to provide a bigger icon that works without any problem in both the iPad and the iPhone. After our icon is ready to use, the `link` tag containing the `rel` attribute and the value `apple-touch-icon` is responsible for loading it on the home screen.

The rest of the HTML lines of code, after the required link for the icon, are used for creating a small header with the text icon through the CSS file of the *UiUIKit* framework.

During the process of adding a new bookmark, we can choose a title for our application. This text will be displayed under our icon on the home screen. Being descriptive and short is a good idea because we want users to identify our application easily.



After the user creates the bookmark through the **Add to Home** button, their screen will change providing a new icon for our application, as shown in the following screenshots:


Carrier
1:39 PM
Game Center
My App
Contacts
Safari


iPhone Simulator
Carrier
5:35 PM
100%
My App
Safari
Photos

Just for simplicity, our icon was called `apple-touch-icon.png`, but you can use your own filename. Of course, if you are using a different filename you must change the link reference for the `apple-touch-icon` tag. It is possible to change the path for this graphic file, simply change the reference and put the icon in another directory accessible from the web server.

- Don't forget that for adding our icon to the home screen of the device, users should use the **Add to Home** button in the **Bookmarks** menu. This means that the process is not absolutely programmatic. On the other hand, the meta tag `apple-mobile-web-app-capable` is required as we had seen in the recipe *Viewing application in full screen* of *Chapter 3*, *Events and Actions*.

There's more...

The `apple-touch-icon` tag allows you to choose a specific size for your icon. For this purpose the `sizes` attribute is used and it expects numeric values with the x letter. Using this attribute it is possible to indicate two icons with different size for each kind of device. For instance, you can use one icon for an iPhone with 57x57 pixels and another for an iPad with 72x72 pixels. In this case, you will need the following lines of code:

```
<link rel="apple-touch-icon" href="small-icon.png" sizes="57x57" />
<link rel="apple-touch-icon" href="big-icon.png" sizes="114x114" />
```

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Viewing applications in full screen* recipe in *Chapter 3*, *Events and Actions*

Specifying a splash image

In a similar way for native applications, you can indicate and use a splash image for your web applications. This kind of image is very useful for displaying something on the screen while the application is loading. Some applications don't need it, but this technique will be especially useful when our web application is offline. By default, a screenshot of the web application the last time it was launched will be displayed. In order to avoid it, our splash image comes to the rescue.

This recipe shows you how to use a splash image for your *iOS* devices.

Getting ready

For this recipe, we don't need any framework, but for simplicity you can use the same file of the previous recipe.

How to do it...

If you're using the same file of the previous recipe, just add the following line before closing the head section:

```
<link rel="apple-touch-startup-image" href="splash.png" />
```

The previous line is referencing a file called `splash.png`, so you'll need to create a PNG file for it. The only requirement for this file is its size, which must be set to 320x460 pixels. You could use a graphic editor such as GIMP or Adobe Photoshop for this action. Save your new graphic file as `splash.png` inside the same directory as that of your XHTML file.

For testing our new splash image, you could launch the application for the home screen to see something such as the following screenshot:

How it works...

As in the previous recipe, we used an HTML tag provided by *Safari Mobile* browser to indicate which image will be loaded when the application is launched. The `link` tag uses two attributes. The `href` attribute indicates the path to the image file relative to the directory of the web server where the application is running.

The image used for this recipe has the size required by iPhone and iPod Touch, but if you want your application to run on iPad devices as well, you need to use a different size for the graphic file. In this case, the size will be 1004x768 pixels. For keeping compatibility, is a good idea to use two different images: one for iPhone and iPod Touch and another for iPad. For instance, if we can have an additional image called `splash-ipad.png`, we'll add a new line of code to our XHTML file:

```
<link rel="apple-touch-startup-image" href="splash-ipad.png" />
```

Surely, you've realized we used an image with 1004 pixels instead of 1024 pixels for the iPad, which is the height of the device. The reason for it is simple and clear: the other 20 pixels are the size of the status bar. This case is analogous for the iPhone, whose height is 480 pixels and we used a height of 460 pixels for our splash image.

Actually, we're using two lines of code referring to two different splash images, but the device will use only one of these. Therefore, an iPad device will load `splash-ipad.png` and an iPhone or iPod Touch will use `splash.png`.

Keep in mind that if the size of the splash image is different than the specified size, the device will ignore it and the image won't be displayed on the screen.

See also

- *Choosing an icon image for the application*

Displaying an image inside a container

This recipe explains how to load an image on the iPhone screen. Normally, images have different sizes and we need to adjust it for loading each image inside a specific container or element in our web page. The process for taking this action is straightforward, thanks to the *iWebKit* framework.

The different files required by this recipe can be found at: `code/ch04/display/` in the code bundle provided on the Packtpub site.

Getting ready

We'll use the *iWebKit* framework for this recipe. Be sure you have this framework ready to use.

How to do it...

1. Open your favorite text editor and create a new file called `display.html`. The main lines of code will be the following:

```
<link href="../iwebkit/css/style.css" rel="stylesheet"
          media="screen" type="text/css" />
```

```
<script src="../iwebkit/javascript/functions.js"
type="text/javascript"></script>
<title>Image</title>
</head>
<body>
  <div id="topbar">
  <div id="title">Image</div>
</div>
```

2. For this recipe we're going to use a simple PNG image with 320x356 pixels. Actually, this image will be a simple red rectangle. You can find this image at: `code/ch04/display/red.png`.
3. Coming back to our `display.html` file, it's time to add the required lines for loading our image inside a `div` element. Simply copy the following code after your `div` element with `id` equal to `topbar`:

```
<div id="content">
    <img src="red.png" alt="red rectangle" />
</div>
```

4. Load `display.html` in your iPhone to see a screen similar to the next screenshot:

How it works...

The *iWebKit* framework automatically adjusts images to containers or elements with independence of image size. In fact, our original image is smaller than the size defined for the container.

In this recipe, we're defining a simple `div` as the container for our image. The web page has two main elements: one toolbar with a title and the container with the image. Between them you can see a blank space due to *iWebKit* establishing a margin for our container, which is a `div` element with `id="content"`. Actually, the `style.css` file of this framework is using the `margin-top` property for setting the mentioned space.

What happens if we don't use any framework? In this case, the image won't be scaled. Consider this example. You can create a new HTML file with the following lines:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta content="text/html; charset=utf-8" http-equiv="Content
        -Type" />
  <meta name="apple-mobile-web-app-status-bar-style"
        content="default" />
  <title>Image</title>
</head>
<body>
  <div><img src="red.png" alt="Sky"/></div>
</body>
```

After this action, reload your file and see that the image is displayed for the iPhone, as shown in the next screenshot:

When we work with images, it is important to keep in mind the `viewport` meta tag defined by *Safari Mobile*. As we learned in *Chapter 3*, *Events and Actions*, this tag defines a rectangular area for determining how the content is displayed on the screen of the device. It's possible to change the default viewport using properties such as `width`, `initial-scale`, `minimum-scale`, and `maximum-scale`. By default, it's a good idea to use the value `device-width` for the width `attribute` in order to work with the appropriate size for each device. However, by applying this value and `1.0` for the others we'll get a different behavior because the image will be scaled to the width of each device. For illustrating this situation, you can change the previous HTML file created by adding the following line into the head section:

```
<meta name="viewport" content="width=device-width, initial-
        scale=1.0, minimum-scale=1.0, maximum-scale=1.0, user
         -scalable=no"  />
```

If you load your page again, the result will be as follows:

In general terms, frameworks such as *iWebKit* make our job easier by dealing with images and managing the scaling automatically, but if you need more control for specific purposes, the `viewport` tag can help you.

There's more...

Alternately, *iWebkit* proposes to use the `background-image` CSS property. Also, through this property we can adjust an image inside an element. However, developers should use this property explicitly in the HTML code or in their customized CSS file for the application.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Scaling to device width* recipe in *Chapter 3*, *Events and Actions*

Creating a grid with images

In the previous recipe, we learned how to load and display an image inside a container. Obviously, some applications require displaying more than one image on the screen at the same time. It's very interesting to use a grid for displaying images in different rows and columns.

In this recipe, you'll learn how to display a few images using a simple grid with four columns, where different rows are created automatically based on the number of images used. All the file's images for this recipe and the required HTML code are available at: `code/ch04/grid` in the code bundle provided on the Packtpub site.

Getting ready

This recipe requires the use of the *UiUIKit* framework. Please, ensure you've installed this framework in your machine.

How to do it...

1. As you've seen in previous recipes, we're going to create a new HTML file with the required lines for loading the CSS file of the *UiUIKit* framework. The main line is the following one:

```
<link rel="stylesheet"
        href="../../uiuikit/stylesheets/iphone.css" />
```

2. Now it's time for the specific lines for creating our grid and for loading our images inside it:

```
<body id="images">
  <div id="header">
    <h1>Grid</h1>
  </div>
```

```
<ul>
  <li><img src="./image1.png" /></li>
  <li><img src="./image2.png" /></li>
  <li><img src="./image3.png" /></li>
  <li><img src="./image4.png" /></li>
  <li><img src="./image5.png" /></li>
  <li><img src="./image6.png" /></li>
  <li><img src="./image8.png" /></li>
  <li><img src="./image9.png" /></li>
  <li><img src="./image10.png" /></li>
</ul>
```

3. Finally, you can save the file as `grid.html`.
4. We're using ten PNG files with 75x75 pixels, which are in the same directory as `grid.html`. Load your new page in your iPhone to see a result similar to next screenshot:

How it works...

The *UiUIKit* provides a special style for creating grid for images. Actually, we defined the body element with `id=images`, because the style will be applied for elements with this value for the `id`. Additionally, the grid is created thanks to a standard unordered HTML list, where each item is a reference to a different image. For each of them, a regular `img` tag is used. The number of images displayed in each row will depend on the size of each image. Usually, we'll work with images with same the size in order to stay consistent.

Space between each item of the list is defined by the CSS style (`iphone.css`) of the framework. By default, each item has 73x73 pixels and 4 pixels for the bottom and left margin. Keep in mind that if your graphic files are bigger, you'll need to adjust these values using your own CSS file or by modifying the original one provided by the framework.

If you prefer, each item of the unordered list can be changed including a link element. Applying this change, you can offer additional functionality when the users click on each image. Don't forget that links inside items have a specific style defined in the CSS file (`iphone.css`) as well.

The CSS style applied for the *UiUIKit* sets the background of the grid to white. Obviously, you can change it by using a different CSS style for it. The easiest way to do that is setting the `background-color` property to your favorite color inside the body tag.

Sometimes images are not loaded and displayed immediately. Considering this, it can be helpful to display something which indicates that the process was launched, but some time is required for loading images. Fortunately, *UiUIKit* does the job for us automatically by displaying a simple animation until the image is loaded and we don't need to add anything to our application. The specific style defined in the CSS file of the framework uses a GIF animation file called `image-loading.gif`. Take a look at the following screenshot to check how it works:

There's more...

If you're interested in applying different styles for your grid, take a look at the `iphone.css` file provided by *iWebKit*. You can find the styles for the element involved in the grid from line number 1047 at the time of publishing. One alternative is to create your own CSS file and redefine the values for these styles and create a reference to this new file inside your HTML file.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Creating a carousel for images

In this recipe, you'll learn to create a carousel. A carousel is a widget that allows the user to slide back and forth between different elements. In our recipe, these elements will be pictures. This method provides a simple and intuitive mechanism for displaying pictures, allowing users to skip between them with a simple touch gesture. Optionally, the carousel can be configured to work as an automatic slideshow.

You can use as many images as you want, but we're only going to work with three pictures for this recipe.

The complete code for this recipe, including the pictures, can be found at `code/ch04/carousel/` in the code bundle provided on the Packtpub site.

Getting ready

Sencha Touch provides a predefined widget that creates a carousel containing HTML elements, so we'll use this framework for this recipe. Check your computer to ensure that this framework is installed.

How to do it...

You'll need three different code files and three pictures. For the best fit on the iPhone screen, we'll use pictures sized at 300x400 pixels.

1. The first code file for our recipe will be a simple XHTML file containing references to the required CSS and JavaScript files for the *Sencha Touch* framework, and two additional references to our new files. The complete code for this initial file called `index.html` is the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf
       -8">
  <title>Carousel</title>
  <link rel="stylesheet" href="../../sencha
       -touch/resources/css/ext-touch.css" type="text/css">
  <script type="text/javascript" src="../../sencha-touch/ext-touch
       -debug.js"></script>
<link rel="stylesheet" href="main.css" type="text/css">
  <script type="text/javascript" src="main.js"></script>
</head>
<body>
</body>
</html>
```

2. The second step is to create our main JavaScript file with the user interface definition. Create a new file with the following lines and save it as `main.js`.

```
Ext.setup({
     onReady: function() {
        var carousel = new Ext.Carousel({
            defaults: {
                cls: 'card'
            },
            items: [{
                html: '<img src="pic01.png"/>'
            },
            {
                html: '<img src="pic02.png"/>'
            },
            {
                html: '<img src="pic03.png"/>'
            }]
        });
        new Ext.Panel({
            fullscreen: true,
            layout: {
                type: 'vbox',
                align: 'stretch'
            },
            defaults: {
                flex: 1
            },
            items: [carousel]
        });
    }
});
```

3. The last step is to add some styles to our widget. The `Tfinal` file will be a CSS file named `main.css` containing the following code:

```
body {
  background: rgb(197,204,211);
  }
  card {
  text-align: center;
  color: #204167;
  text-shadow: #3F80CA 0 1px 0;
  font-size: 72px;
  padding: 80px 40px;
 }
 .x-phone .card {
  padding: 10px;
  font-size: 36px;
  }
```

Don't forget to copy all files, including the pictures, to the same directory under the control of your web server. For example, create a new folder inside the `/Library/WebServer/Documents/` folder on Mac OS X. Ubuntu users should use `/var/www/` folder.

How it works...

As we've mentioned before, *Sencha Touch* provides a specific widget for creating a carousel. From the technical point of view, it's a JavaScript class called `Ext.Carousel`. The items array is used to indicate the components of the carousel for which our recipe is using a HTML `img` tag for each item of this array.

In addition, we can use a specific class for applying some styles. This recipe is using a class called `cls`, which is defined in our `main.css` file. Our stylesheet sets a different background color instead of using the default white background. We have also applied a `padding` to our pictures so that they fit better inside the container. In order to do that we override the `x-phone` style, which is predefined in the main CSS file (`ext-touch.css`) of the framework used in this recipe.

The *Sencha Touch* framework requires a main container in which you enable the widgets to use the application. We used a simple panel through the `Ext.Panel` class—a first level container—and pass it some properties to configure our widget to the `fullscreen` property that is set to `true`. The `layout` property establishes how the items are arranged into the container. As you can observe, we are using vertical box object by setting the type property to `vbox`. The carousel is displayed inside the referred container through the items array.

At the bottom of the page, three small circles indicate we're using three elements for our carousel. Also, each circle changes color when it is active. For example, when a user launches our application, the device displays the first picture and the first small circle is a different color from the rest. Take a look at the following screenshot to see this effect:

When the user slides to the second picture, the second small circle is colored as active, as shown in the following screenshot:

There's more...

In order to get a complete understanding of all the options available for the `Ext.Carousel` `widget` it is a good idea to take a look at the API documentation of the framework.

See also

- *Installing the Sencha Touch framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*.

Rotating images

Applying some effects to our images could be useful in order to achieve better user experience. One of the simplest effects that can be applied to images is rotation. We can rotate and grade an image on the screen of the device. This recipe explains how to apply this effect with only a little bit of CSS code.

Getting ready

The WebKit engine used by *Safari Mobile* defines specific properties for applying effects and simple transitions to HTML elements. We're going to use them directly; however, the *iWebKit* framework will be useful for defining an initial layout for our image. Be sure that you've installed this framework in your computer before continuing.

How to do it...

As you've seen in the previous recipe about how to display an image inside a container, we'll use a standard XHTML file with some references to the JavaScript and CSS files provided by *iWebKit*.

1. The main lines for loading this framework are the following:

```
<link href="../iwebkit/css/style.css" rel="stylesheet"
       media="screen" type="text/css" />
<script src="../iwebkit/javascript/functions.js"
       type="text/javascript"></script>
```

2. After the lines shown earlier, we're going to add some lines for applying our rotation using CSS. Specifically, these are the lines for the effect:

```
<style type="text/css" media="screen">
  #picture {
  -webkit-transform: rotate(35deg);
}
</style>
```

3. Additionally, you need to include an `img` tag for the image, which will be rotated. The rest of the HTML code will be the following:

```
<title>Rotate</title>
</head>
<body>
  <div id="content">
  <img src="pic01.png" id="picture" />
  </div>
</body>
</html>
```

4. This recipe assumes you have an image or picture file called `pic01.png` in the same directory or folder as your `HTML` file.
5. We are ready for loading and checking our rotation. The result should be as shown in the next screenshot:

How it works...

The simple rotation done for this recipe is possible thanks to the `-webkit-transform` property included in the WebKit engine of *Safari Mobile*. Using the value `rotated` followed by `35deg` string, we can rotate our picture by 35 degrees. Obviously, it's possible to choose other values for the rotation expressed in degrees. The mentioned property belongs to CSS so we need to declare it through the `style` tag. Alternately, you can create a new CSS and add a reference to it inside your HTML file.

Our picture is displayed on the screen using a specific `div` with a regular `img` tag inside. The style of this block element is `content`, which is declared in the CSS file provided by the *iWebKit* framework.

In addition to rotation, other transformations can be applied using the `-webkit-transform` such as translation and scaling.

There's more...

Visit the **Visual Effects** section of the Safari Reference library at: `http://developer.apple.com/library/safari/documentation/AppleApplications/Reference/SafariCSSRef/Articles/StandardCSSProperties.html#//apple_ref/doc/uid/TP30001266-VisualEffects` to get more information about which effects and transformations can be applied through CSS properties and parameters.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Displaying an image inside a container* recipe

Scaling an image by applying animations

In the previous recipe, you learned how to apply a simple rotation transformation to rotate an image, but *Safari Mobile* allows us to create more sophisticated transformations and effects with a few lines of CSS and JavaScript. This recipe shows you how to create a simple animation for scaling a picture.

Our goal will be to display an image, as you have learned in previous recipes, then scaling using a 0.2x factor through an animation. Actually, the animation applies the transformation with a time delay for a visual effect. The animation will start when the user clicks on the picture.

Getting ready

Just for simplicity, we're going to use the *iWebKit* framework one more time. However, no specific framework is required for applying the effect.

Also, you'll need an image File; a good size for the image of this recipe will be 300x400 pixels. We are using PNG files, but other formats such as JPEG and GIF are valid as well.

How to do it...

1. We'll start by creating a new XHTML file with the standard headers including the references for the JavaScript and CSS files required for *iWebKit*:

```
<link href="../iwebkit/css/style.css" rel="stylesheet"
      media="screen" type="text/css" />
<script src="../iwebkit/javascript/functions.js"
      type="text/javascript"></script>
```

2. The next step is to write a simple JavaScript function for responding to the `onclick` event:

```
<script type="text/javascript" charset="utf-8">
    function animate(theStyle) {
        theStyle.webkitTransform = 'scale(.2)';
  }
</script>
```

3. Now, it's time for the CSS style of our image; add the following lines to `animation.html`:

```
<style type="text/css" media="screen">
  #picture {
    padding-left: 10px;
    -webkit-transition: all 4s;
  }
</style>
```

4. The last step will be to add the reference to our image including an event handler for the click:

```
<body>
  <div id="topbar">
  <div id="title">Scaling</div>
  </div>
  <div id="content">
    <img src="pic-rot.png" id="picture"
    onclick="animate(this.style)"/>
  </div>
```

5. Now you're ready to test how the animation works.

How it works...

When the application is launched, we display the picture with its original size. Simply, we define an `img` tag with a reference to our image. Additionally, this tag is using an event handler for calling a JavaScript function when the user clicks on the image. Keep in mind that this *click* is equivalent to a touch gesture with a finger on the device. Our JavaScript function is called `animate`. As a parameter, we'll need the style object of the image. This function receives the style object and modifies the `-webkit-transform` attribute through the `webkitTransform` method of the object. Actually, we use the scale transformation with a `.2` value to indicate that the image will be reduced to half its original size.

Regarding the initial style of our image, we're using two different properties for setting up the configuration. One of these is `padding-left`, which is used for centering the image on the screen. The other one is `-webkit-transition`, which indicates how the transition should work. Choosing a value of `4s` for this property, we establish that the duration of the transition is 4 seconds. The `all` value is used for all styles of the element. Instead of `-webkit-transition` with these values, you can use two different properties with two values. We're referring to `-webkit-transition-property` and to `-webkit-transition-duration`. In order to do it, simply replace the `-webkit-transition: all 4s` line for the following lines:

```
-webkit-transition-property: all;
-webkit-transition-duration: 4s;
```

The initial state of the animation happens when the picture is displayed using its original size, as shown in the next screenshot:

After clicking on the image, the transition will start. It will begin over the course of four seconds. The image shrinks with a smooth animation. For example, the following screenshots show you one of the phases of the process:

Finally, when the animation is finished, our image will be displayed much smaller:

You should try to change the value of `-webkit-transition` for changing the duration of the animation. Also, it's possible to include other transformations such as rotation. Combining different transformations and animations, we can achieve great visual effects and improve the overall experience for our users.

There's more...

In order to learn more about which kinds of animations can be used, take a look at: `http://developer.apple.com/library/safari/documentation/AppleApplications/Reference/SafariCSSRef/Articles/StandardCSSProperties.html#//apple_ref/doc/uid/TP30001266-VisualEffects`.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Rotating images* recipe

Taking and displaying pictures

You'll surely find it useful to use the camera of your iPhone. By default, a basic application is included with the phone that allows you to take pictures and save them automatically to the internal memory. You can then access your pictures through the Photo Library application.

However, it could be interesting to develop our own applications that can take and display pictures inside a container or other kind of element. For example, we can implement a feature for our application which allows taking and displaying pictures in a specific area of the screen delimited by a container element.

Most of the recipes in this book are focused on web applications instead of native applications. It is possible to build native applications using CSS, HTML, JavaScript, and the PhoneGap framework. Some features of the *iOS* operating system are inaccessible from a pure web application, such as taking and displaying pictures. The process we'll use to interact with the iPhone's camera and gallery will be explained in this recipe.

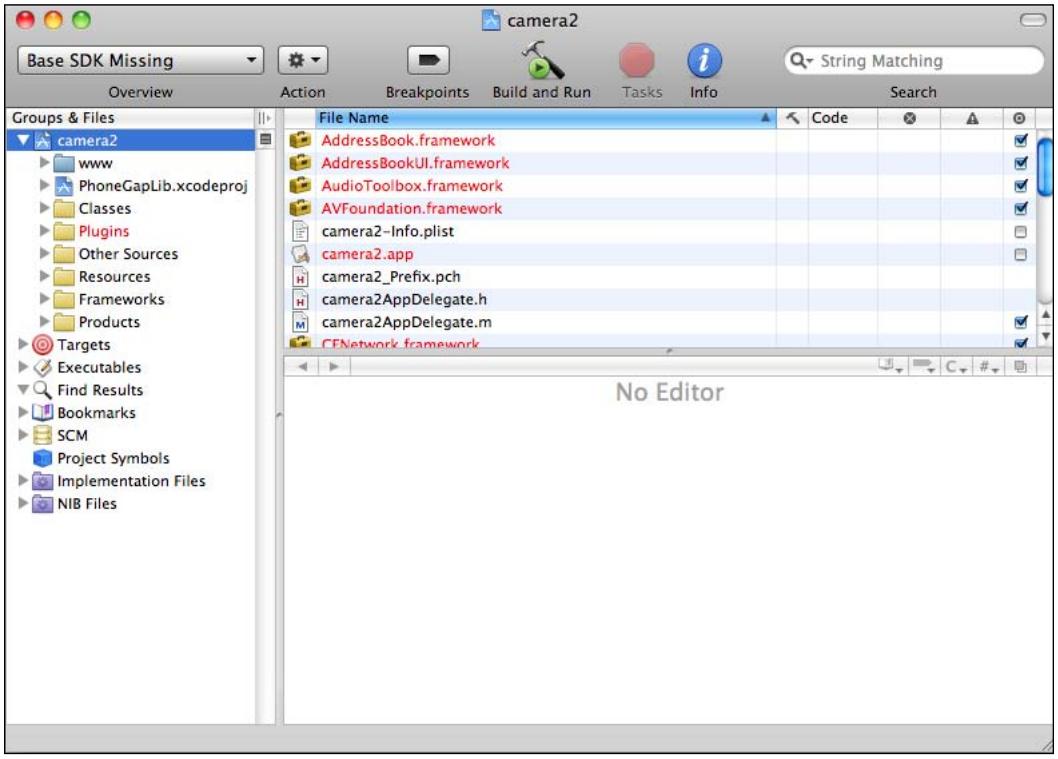
Getting ready

Remember, the use of PhoneGap for iPhone requires the Mac OS X operating system. You must have a computer with a recent version of this operating system. Additionally, we need to install the iPhone SDK and Xcode, as you've seen in the recipe *Installing PhoneGap* in *Chapter 1*.

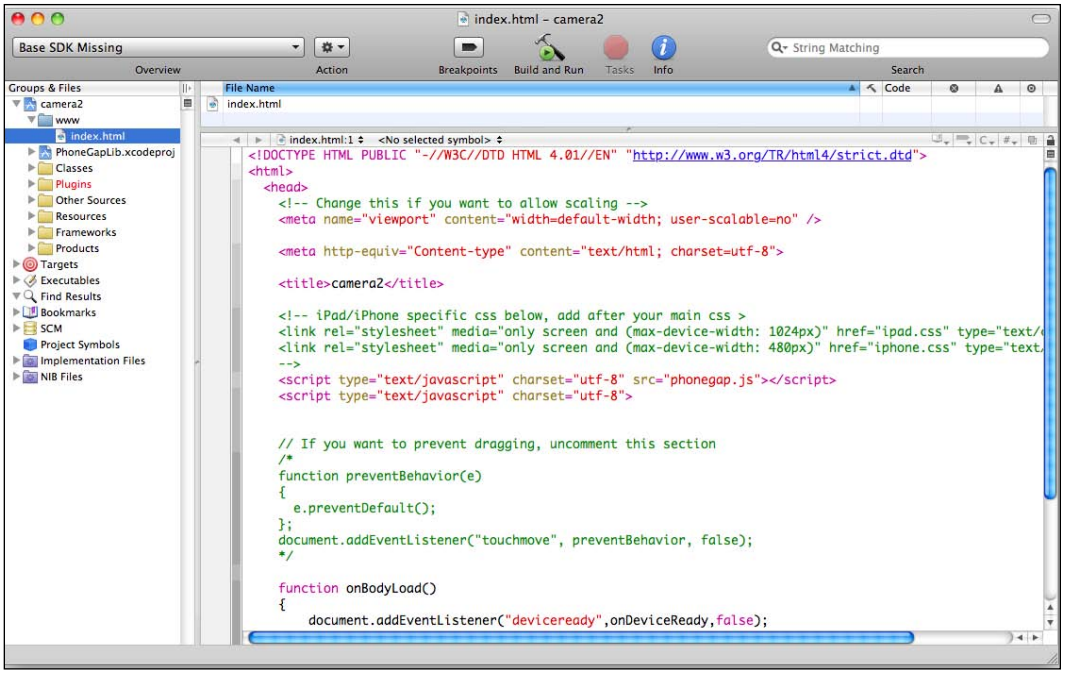
On the other hand, we'll need a real iPhone device for testing this recipe because the iPhone simulator doesn't offer a way to test the camera. Also, for installing your final native application on the device, you'll need to belong to the *iOS* developer program. Bare in mind that this process isn't free; you must pay an annual fee.

How to do it...

1. Launch your Xcode application and create a new project using the **File | New Project...** menu option. Immediately, you'll see a new dialog box on your screen. Select **PhoneGap-based Application** in **User Templates** and click on the **Choose...** button.
2. The next step will be to choose a folder and name for your new project. After clicking on the **Save** button, Xcode will create a new project for you.



3. Select your `index.html` file located inside the `www` folder. By default, Xcode will open the file in the editor.



4. Now it's time for adding our JavaScript code for taking and displaying pictures. Add the following line of Javascript to your `index.html` file:

```
function isOK(data) {
  var strData = "";
  var img = document.getElementById("img");
  img.src = strData + data;
  }
function isError(error) {
  navigator.notification.alert("Error: " + error, "Ooops!", "OK");
  }
function takePicture() {
  navigator.camera.getPicture(isOK, isError, {quality: 30});
  }

function onDeviceReady(
  takePicture();
  }
```

5. Additionally, we're going to add some HTML code for displaying our picture on the screen. Copy the following code inside the body section of the `index.html` file:

```
<img src="" id="img" />
```

6. Finally, we'll add CSS code for hiding our tag `img` after the picture is taken. The following lines will be inside your `head` section of the `index.html` file:

```
<style type="text/css" media="screen">
       #img {
              display: none;
}
</style>
```

7. Now, we're ready for building and deploying our application on the device. Click on the **Build and Run** button in Xcode.

How it works...

PhoneGap offers direct access to the camera through a specific object called `navigator.camera`. The method `getPicture` allows us to directly take a picture and manage its data. In other words, after taking a picture we can display it on the screen. Regarding the default camera application, it will be open and once photo is taken it will be closed and the control will be returned to our application.

The code for this recipe implements a simple application for calling to the internal photo application when our application is launched from the device. After taking the picture, automatically it will be displayed through the `img` tag inside the body section of the `index.html` file, which identifies our main screen. If some error happens, the `isError` function will be invoked and it will display a simple alert indicating the current error. Surely, you have discovered that isn't the standard alert JavaScript method. You're right. We're using a customized box for it, as we explained in the recipe *Creating and customizing an alert box* in *Chapter 2*, *Building Interfaces*.

The `takePicture()` function is launched when the application and the framework are ready. This JavaScript function is a wrapper to our functionality. Actually, `takePicture()` directly invokes `navigator.camera.getPicture` method. This method uses three different parameters, one callback function to indicate what will happen after taking a picture, an other callback for managing errors, and a third parameter, which is an array with different options. Our recipe is only using the `quality` option for indicating the quality of the taken picture. This parameter is an integer between `0` and `100`, where a bigger number means more quality. In order to avoid memory problems and for compatibility with older models of iPhone, we are setting `30` as the value for this parameter.

By default, the `isOK` callback function will pass a parameter, which contains the raw data of the picture. Actually, it is a string for the `Base64` encoded image. We're concatenating this kind of data with an other string (`strData`), which indicates the format of the image. Finally, we take our `img` tag and modify its `src` attribute by assigning our new string representing the picture. The picture is displayed on the screen after this action.

Instead of using the `Base64` encoded string for our pictures, it's a good idea to use the file **URI** (**Uniform Resource Identifier**). PhoneGap allows us to do it through one specific option (`destinationType`) for the array passed as a third parameter of the `getPicture` method. If you're interested in this approach, you can replace the call to `getPicture` method of the recipe for this one:

```
navigator.camera.getPicture(isOK, isError, {quality: 30,
      destinationType: navigator.camera.DestinationType.FILE_URI});
```

Also, you should modify our `isOK` JavaScript callback function. Simply, replace it with this new code:

```
function isOK(imgU) {
    var img = document.getElementById("img");
    img.src = imgU;
}
```

There's more...

Complete information about how to enroll in the *iOS* Developer Program can be found at: `http://developer.apple.com/programs/ios/`.

If you're interested in distributing a native application, it would be worthwhile to take a look at: `http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iphone_development/145-Distributing_Applications/distributing_applications.html#//apple_ref/doc/uid/TP40007959-CH10-SW2`.

See also

- *Installing the PhoneGap framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating and customizing a notification box* recipe in *Chapter 2*, *Building Interfaces*

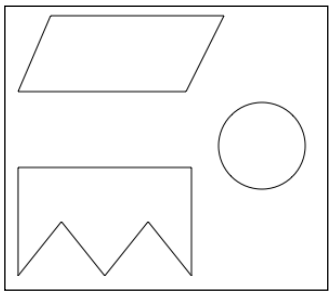
Drawing geometric figures

The WebKit engine provides an interesting element which allows drawing. Its name is **canvas** and it was originally designed for Apple to build Mac OS X dashboard applications. Sometime ago, the Cupertino company decided to extend its support to Safari and *Safari Mobile* web browsers. Regardless, canvas is fully supported under the recent HTML5 standard. This means that other web browsers such as Firefox or Opera can render a canvas component.

Basically, canvas defines an area inside the screen where we can draw geometric figures or simply display images. One of the most important benefits of the canvas element is that we don't require additional libraries or frameworks; all we need is a modern web browser Traditionally, technologies, such as Adobe Flash and Java applets were necessary to implement similar functionalities.

JavaScript helps us to interact with the canvas. We can draw different figures and shapes using some JavaScript objects and methods defined by *Safari Mobile*. Additionally, CSS3 allows us to apply new properties, which give us better control over how the content is displayed inside the canvas element. Fortunately, *Safari Mobile* implements this version of the CSS specification.

In order to learn how to use the `canvas` element for our JavaScript iPhone applications, we'll explain in this recipe how to draw some geometric figures. Concretely, more specifically our goal is to draw three different figures, as shown in the following image:



The complete code for this recipe can be found at: `code/ch04/geometric/` in the code bundle provided on the Packtpub site.

Getting ready

Although we don't need any specific framework to work with the `canvas`, we're going to use *iWebKit* just to achieve a look and feel closer to a native iPhone application.

How to do it...

1. We'll start by creating a simple XHTML file called `figures.html` with the required headers to load JavaScript and CSS files of the *iWebKit* framework:

```
<link href="../iwebkit/css/style.css" rel="stylesheet"
         media="screen" type="text/css" />
<script src="../iwebkit/javascript/functions.js"
         type="text/javascript"></script>
```

2. The following step is to add the required JavaScript code for drawing our figures. Simply add the following lines to our new file:

```
<script type="text/javascript" charset="utf-8">
    function drawFigure() {
        var canv = document.getElementById("thecanvas");
        var ctx = canv.getContext('2d');
       /* First figure */
  ctx.beginPath();
```

```
            ctx.moveTo(10, 80);
            ctx.lineTo(40, 10);
            ctx.lineTo(200, 10);
            ctx.lineTo(200, 10);
            ctx.lineTo(165, 80);
            ctx.closePath();
            ctx.stroke();
           /* Second figure */
           ctx.beginPath();
           ctx.moveTo(10, 150);
           ctx.lineTo(10, 250);
           ctx.lineTo(50, 200);
           ctx.lineTo(90, 250);
           ctx.lineTo(130, 200);
           ctx.lineTo(170, 250);
           ctx.lineTo(170, 150);
           ctx.closePath();
           ctx.stroke();
           /* Third figure */
           ctx.beginPath();
           ctx.arc(235, 130, 40, 0, 360*Math.PI/180, true);
           ctx.stroke();
       }
   </script>
```

3. After adding the JavaScript code, we only need to define our `canvas` element:

```
<div id="content">
  <canvas id="thecanvas" width="300" height="300"></canvas>
</div>
```

4. We're ready to display our first `canvas` containing three different geometric figures.

How it works...

When we work with `canvas`, the first action is to define it using the specific HTML5 tag. Two attributes (`height` and `width`) are required to define the size of the canvas. It's important to close the tag with `</canvas>`. Now our canvas is ready to be managed by JavaScript code.

The `onload` event handler of the body tag is used to invoke directly a main JavaScript function (`drawFigure`), which immediately starts accessing the canvas and drawing the figures inside it. Obviously, when the page is loaded the `onload` event handler is responsible to invoke our function.

Inside our main JavaScript function, you can find a first line, which gets a reference to our `canvas` object through the `getElementById` method. This provides access using the **DOM (Document Object Model**) of the HTML page. The second line of code retrieves the `context` of our canvas. Actually, the `context` object is very important because it gives access to all methods for drawing inside the canvas.

We used `2d` as parameter for the `getContext` method, which offers a two-dimensional space for our drawing.

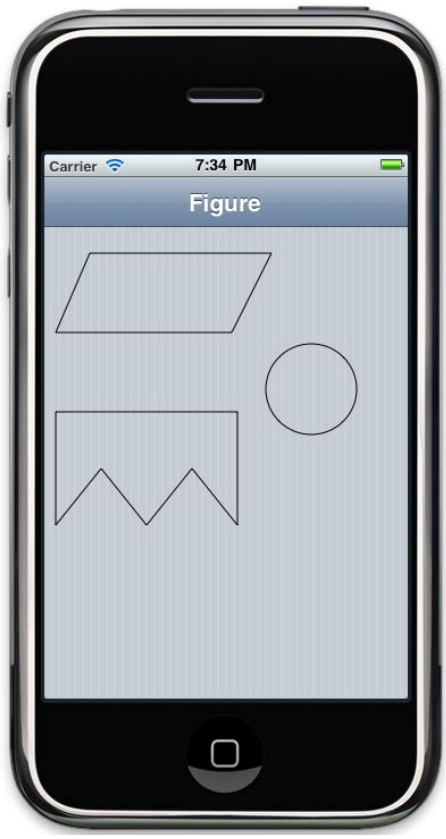
In order to use canvas properly, it is very useful to imagine that we have a virtual pencil or brush for painting on it. Instead of moving our pencil with our hand, we need to use a coordinates system based on X and Y axes. This means we need to keep in mind that we'll be working with a Cartesian coordinate system inside our canvas.

The secret of canvas is the use of paths. Basically, these *paths* are a set of lines and arcs and connections between them. In other words, by applying these basic elements, we can draw tons of figures based on infinite combinations of them. Only the imagination sets the limits. As you can guess, JavaScript appears on the scene to handle these basic elements.

It's important to understand how *paths* are used. Two of the most important methods of the context object are `beginPath()` and `closePath()`. The first one sets the origin of our new path to `0,0` coordinates. `closePath()` indicates the ending of the current path. On the other hand, you will need to indicate that the path should be displayed on the screen. This action is done thanks to the `stroke()` method of the context object. Apart from these methods, we only use two additional methods: `lineTo` and `moveTo`. Both of them need coordinates as parameters. Surely, you must have guessed how these methods work. `lineTo` draws a line between two points that are the parameters received for it. The `moveTo` method moves our virtual pencil to the given coordinate of the canvas.

Our circle is not using `lineTo` and `moveTo`, instead it uses a method called `arc`. This method needs six different parameters: `X` and `Y` points, `radius`, `starting angle`, `ending angle`, and the `direction` in which the circle is drawn from the start angle. Actually, this last parameter is a Boolean and it can be `true` or `false`. In the first case, the direction will be clockwise. It's very important to take care about the unit used by the arc method as it uses *radians*. Remember, the equivalence between angles and radians is that 360 degrees is the same as 2π radians. This is the reason we use `360*Math.PI/180` in the code of our recipe, because we'll have to multiply π/180 for converting between units.

When you view the file on your iPhone or browser the final result will be similar to the next screenshot:

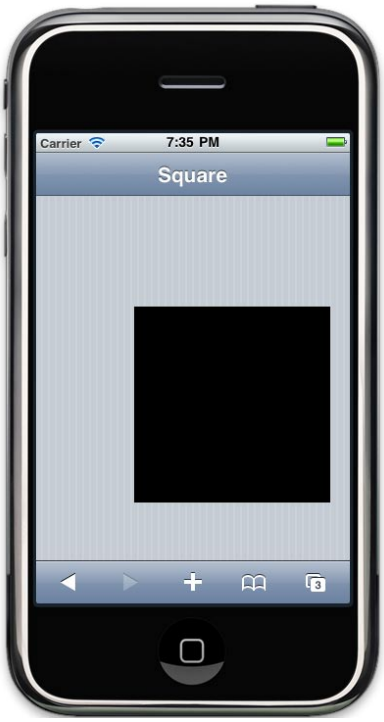


There's more...

Using paths is the better way to draw complex figures on the screen, but *Safari Mobile* provides other JavaScript methods to draw simple shapes. One of these is `fillRect`, which allows us to draw a rectangle. For instance, we can draw a simple square using the following JavaScript function:

```
<script type="text/javascript" charset="utf-8">
    function drawSquare() {
      var canv = document.getElementById("thecanvas");
      var ctx = canv.getContext('2d');
      ctx.fillRect(100, 100, 200, 200);
  }
</script>
```

The following screenshot shows the result of executing the previously described function:



The `canvas` tag has a lot of options so you can take a look at online documentation provided by Apple at: `http://developer.apple.com/library/mac/#documentation/AppleApplications/Conceptual/SafariJSProgTopics/Tasks/Canvas.html`.

The other useful resource is a tutorial offered by *Mozilla Developer Network* at: `https://developer.mozilla.org/en/canvas_tutorial`.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Applying colors* recipe

Applying colors

In the last recipe, you learned how to draw geometric figures on the screen using paths through lines and movements. You can go further by applying some styles such as colors. Fortunately, *Safari Mobile* provides a few JavaScript functions for applying these styles, making our job easier.

This recipe describes how to apply background colors to figures inside a `canvas` element.

Getting ready

In order to make things easier, we're going to use our code generated for the previous recipe as a starting point. This means we'll need to have the *iWebKit* framework installed in our computer.

How to do it...

1. We'll start by saving the file generated for the previous file as `colors.html`. Then, let's go to the line `ctx.stroke()` for the first figure. Delete it and insert the following two lines:

```
ctx.fillStyle = "#3eac3e";
ctx.fill();
```

2. Repeat the process for the other figures. You can even change the color assigned for the `fillStyle` property. For example, we can use `#295ea0` for the figure at the bottom and `#ffffff` for the circle.
3. When you're ready, reload your new page in your device. The result will be as shown in the next screenshot:

4. You can take a look at the complete code used by this recipe at: `code/ch04/colors.html`.

How it works...

Basically, we need one JavaScript method and one property for coloring our figures and shapes inside a canvas. We're talking about `fillStyle` and `fill()`, which are provided by the context object.

`stroke()` and `fill()` method have similar behavior; both are used for painting inside the canvas.

We used the HTML notation for setting colors, but you can also use the string notation or the *rgba* notation introduced by HTML5. For instance, using string notation, you can set `white` string for the white color. Alternatively, you can use (255,255,255, 0) for the *rgba* notation.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Drawing geometric figures* recipe
- *Working with gradients* recipe

Working with gradients

iPhone's user interface makes intensive use of gradients and shading for achieving a good looking user experience. For example, a glossy effect can be created by applying gradients of different colors. Through colors and gradients we can get a closer interface to how native applications look and feel.

A gradient can be applied to a figure as either *linear* or *radial* to our shapes and figures, the difference being the direction in which the gradient is applied. Linear gradients need a starting and ending point. On the other hand, radial gradients need to define applied circle to be. From the geometric point of view, linear gradients are applied through a simple vector and radial gradients are defined using a two-dimensional array of vectors.

The required HTML files for this recipe can be reached at: `code/ch04/gradients/` in the code bundle provided on the Packtpub site.

Getting ready

As you've seen in other recipes regarding canvas element, we don't need to use any specific framework. We'll continue using the *iWebKit* framework to keep a consistent look and feel for iPhone.

How to do it...

For this recipe, we're going to create and use two different files: one for linear gradient and another for radial gradient. Let's start with linear gradients.

Applying linear gradients

To avoid starting from scratch, we can use the file created in the previous recipe. Actually, for learning how to apply a linear gradient, we're going to target one of the geometric figures from the previous recipe. You can start by saving your `colors.html` as `linear.html`.

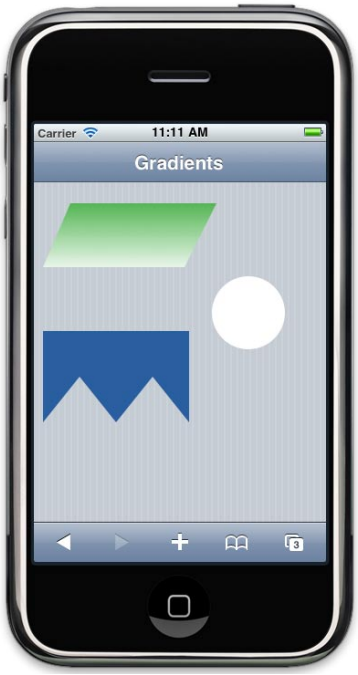
1. The first step will be to delete the line of JavaScript code about applying colors for the first figure. In practice, you can delete the following line:

   ```
   ctx.fillStyle = '#3eac3e';
   ```

2. In its place, we'll add the following lines to create a gradient fade between two colors:

   ```
   var grad = ctx.createLinearGradient(0,0,0,150);
   grad.addColorStop(0, '#3eac3e');
   grad.addColorStop(0.6, '#ffffff');
   ctx.fillStyle = grad;
   ```

3. If you load your new `linear.html` file on your iPhone, you'll see a final result, as shown in the following screenshot:



Using radial gradients

1. For our first contact with radial gradients, we're going to create a new file called `radial.html`. We can reuse the code from the previous example for linear gradients. Just replace the complete `drawFigure` JavaScript function for the following one:

```
function drawFigure() {
    var canvas = document.getElementById("thecanvas");
    var ctx = canvas.getContext("2d");
    var grad = ctx.createRadialGradient(300, 300, 0,300, 300,
          300);
    grad.addColorStop(0, "8a084b");
    grad.addColorStop(0.2, "8a084b");
    grad.addColorStop(0.25, "#33aaee");
    grad.addColorStop(0.45, "#fe9a2e");
    grad.addColorStop(0.55, "#eeeeee");
    grad.addColorStop(0.8, "#f7fe2e");
    grad.addColorStop(0.95, "8a084b");
    grad.addColorStop(1, "8a084b");
    ctx.fillStyle = grad;
    ctx.fillRect(0, 0, 300, 300);
}
```

2. Now you're ready to load your new `radial.html` file on your device.

How it works...

Each gradient is created through a method provided by the `context` object. Specifically, the `createLinearGradient` method is used for linear gradients, while the `createRadialGradient` is for a radial gradient. In the first case, the `createLinearGradient` defines a vector using the coordinates for two points. Each of these points will need the X and Y coordinates based on the Cartesian axes. For a radial gradient the `createRadialGradient` will need two additional parameters for a total of six. The first three parameters define the start of the gradient and the other three define the end point. Each point uses the X and Y coordinates plus a value for the radius.

After we have created our linear or radial gradient, we'll need to use the method `addColorStop` provided by the gradient object. In our recipe, `grad` is an object of this JavaScript class. The `addColorStop` method requires a starting position plus a color to be applied. In both kinds of gradients, `addColorStop` uses these parameters. Each color is defined for the gradient. It is defined between `0` and `1` point so the first parameter for the `addColorStop` method is a float number between them.

For displaying our figures with gradients, we're assigning our grad object to the `fillStyle` property of the context of the canvas. Finally, the `fill()` method is called to display the complete figure on the screen.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Drawing geometric figures* recipe
- *Applying colors* recipe

Adding an activity indicator

Some actions take a long time and it's very useful to indicate to our users that an action or activity is in progress. Therefore, users will stay patient while they wait as they are being informed that the application is working properly. Usually, an application displays a special widget on the screen, which uses a simple animation to indicate that the application is still working. We can even use a simple text such as Loading, Working, or Waiting.

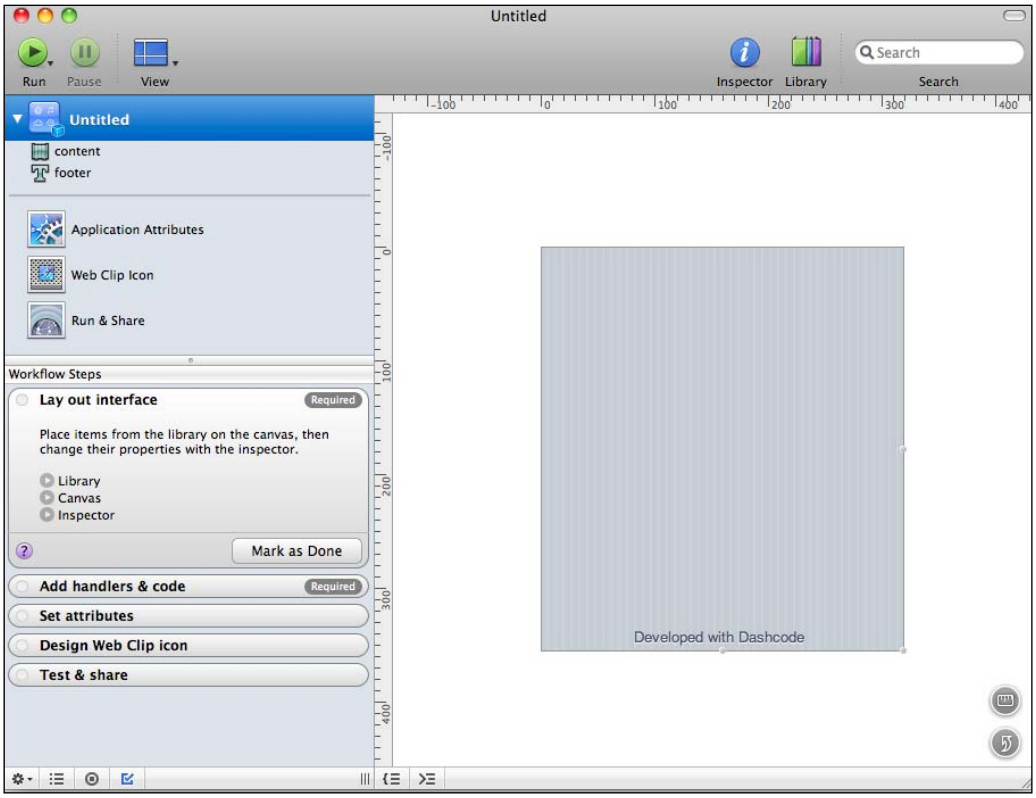
In this recipe, you'll learn how to use the activity indicator widget provided by *Dashcode*. The complete project can be reached at: `code/ch04/activity`.

Getting ready

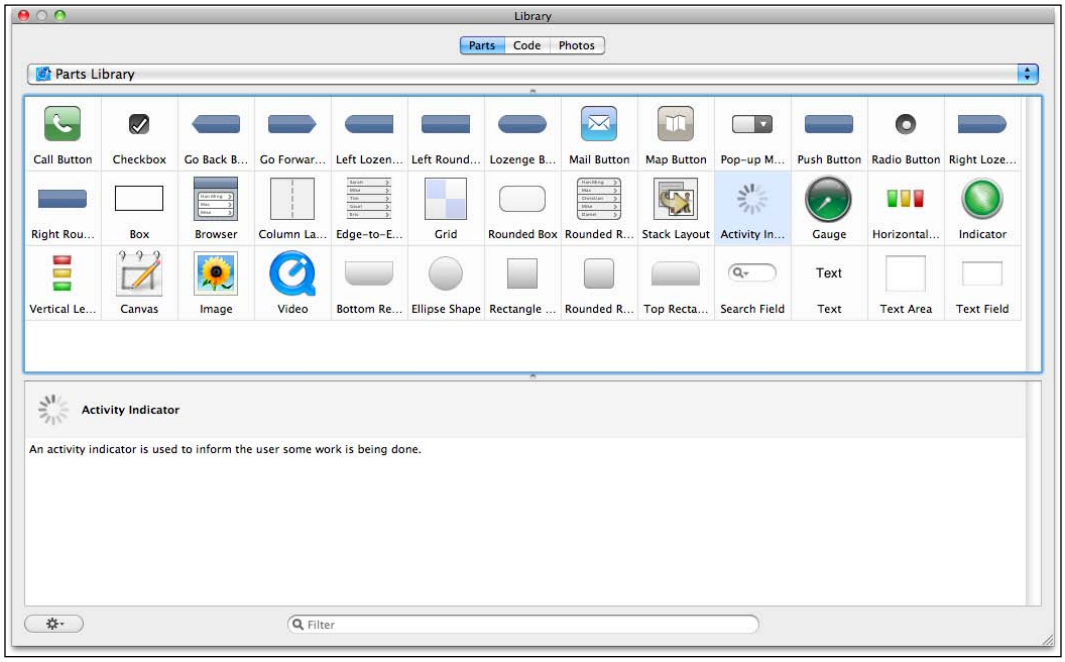
This recipe requires Apple *Dashcode*, so we need a computer with the Mac OS X operating system. It's highly recommendable to have installed the *iOS* SDK, which includes the iPhone Simulator. Apple *Dashcode* will allow us to test and debug our web application for the iPhone.

How to do it...

1. Launch the Dashcode application from the folder `Developer/Applications/` on your system. Now, you can see a dialog box for choosing a template for a new project.
2. Select **Safari** and **Custom** and only mark the value **SafariMobile** in the **Develop for** checkbox.
3. *Dashcode* will show you a new window with two main areas: One on the left for handling all the options about our new project and another area on the right showing a workspace widget for constructing our application from prebuilt parts.

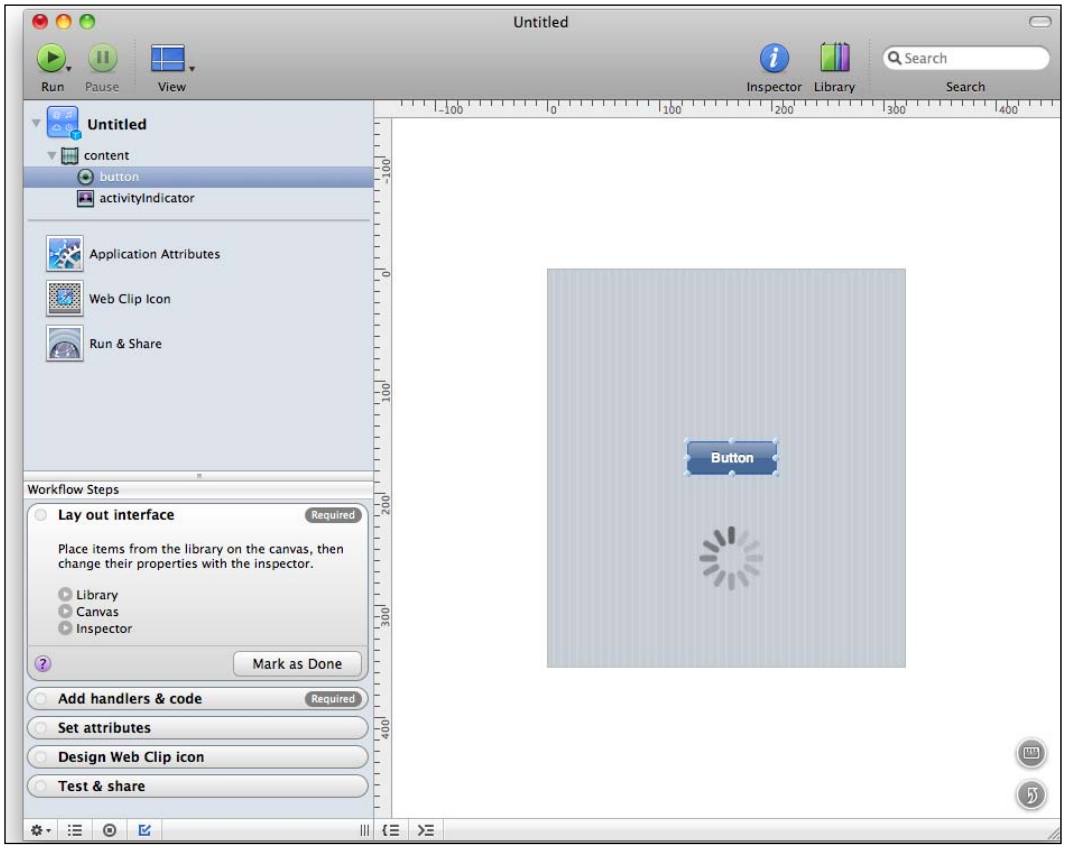


4. Click on the **Library** button, located at the top of the window, to see the parts that are available to choose from. Select the **Activity Indicator** icon, as shown in the following screenshot:

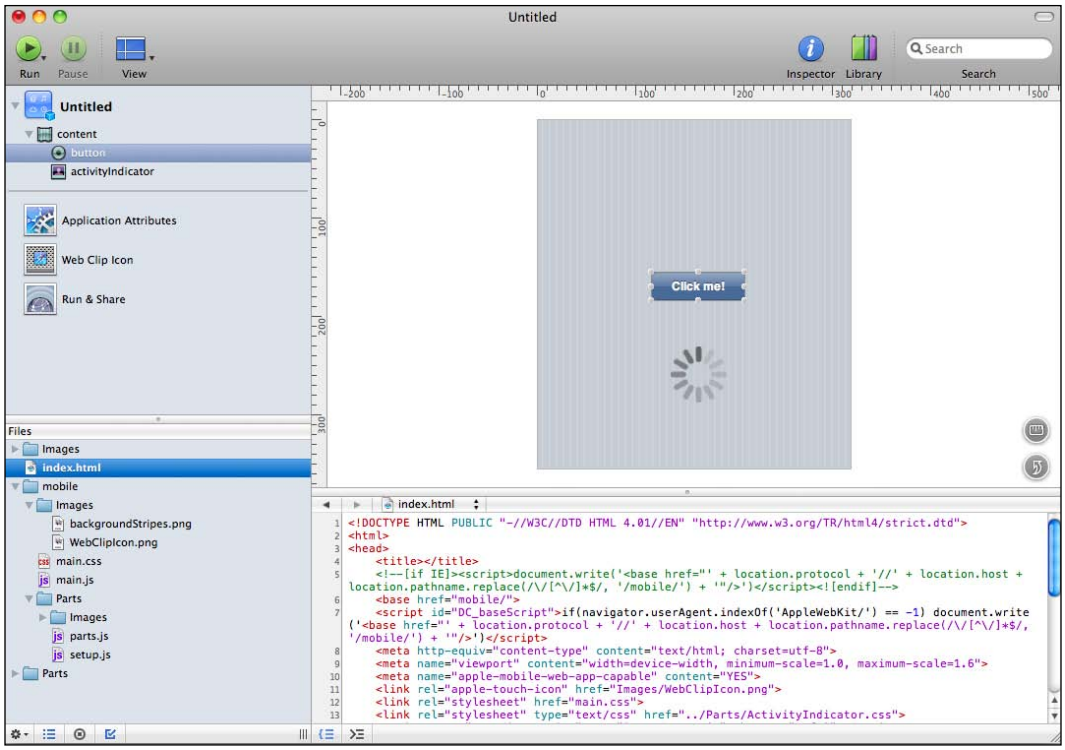


5. The next step will be to drag-and-drop the mentioned icon to our workspace located on the right area of the main window.
6. Now, we're going to delete the label developed with *Dashcode* by selecting it and pressing the *Delete* key on your keyboard. We'll use a simple button for loading our indicator, so you should look for the **Push** button, which also can be found on the **Library** panel.

7. Drag this new **Button** to our workspace and drop it above our widget indicator. Your workspace will look as shown in the following screenshot:



8. Our new **Button** has a default label, so it's a good idea to change it for something more appropriate. Click on the **Inspector** button and a new window will appear. Change the **Button** label to **Click me!**. Close the **Inspector** window and the label for our button will be modified.
9. The user interface for our small application is ready. Now, it's time for adding some code to render the activity indicator when the user clicks on the button.
10. At the bottom left of the main window of *Dashcode*, you can find some icons. Click on the window that looks like a bullet list, titled **Show or hide this project's files**.
11. The **Workflow Steps** area will be replaced by a tree with all the files that were created automatically along with our project. Click on `index.html` and its source code will appear in our workspace.



12. The next step will be to look for the `img` tag with `id="activityIndicator"` and add this new value inside the style attribute: `display: none;`
13. Now, you should add the following `onclick` event handler to the `div` element that has the attribute `id="button"`: `onclick="showIndicator()"`
14. Finally, we need a JavaScript function for showing our indicator when the button is clicked. Add the following JavaScript code inside the `head` section of the `index.html` file:

```
<script type="text/javascript" charset="utf-8">
    function showIndicator() {
      var indicator =
      document.getElementById("activityIndicator");
      indicator.style.display = "inline";
  }
</script>
```

15. We have now finished our simple application and we can execute it by clicking on the **Run** button located at the top of the main window of *Dashcode*. The *iPhone Simulator* will start automatically.

16. If the user clicks on the button, the activity indicator will be seen in action:

How it works...

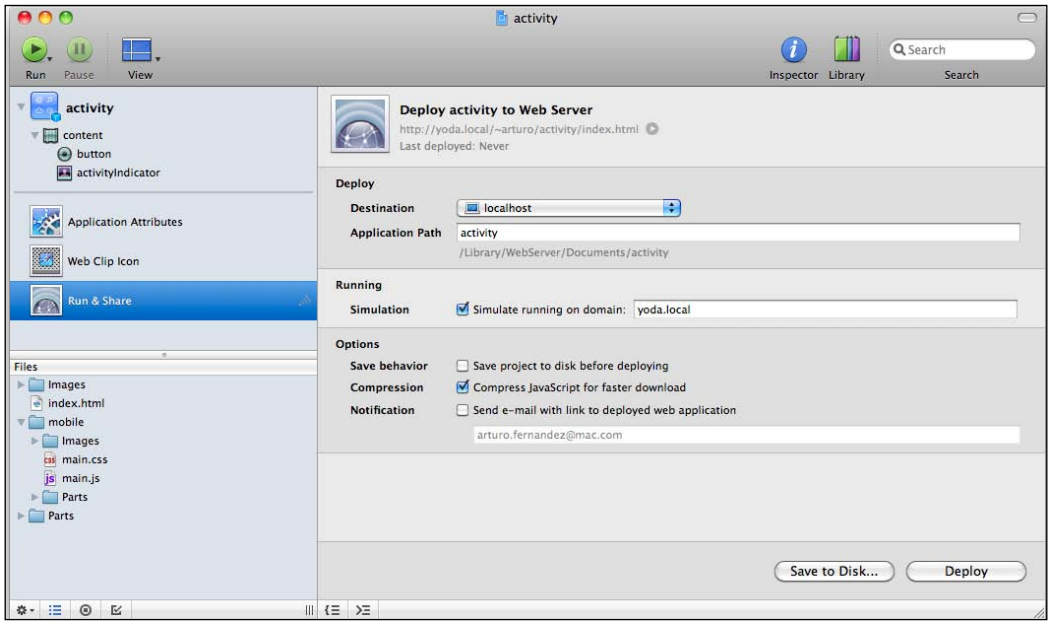
Dashcode provides a library of ready-to-use parts, including an indicator for activities in progress. Thanks to the user interface of Dashcode, we can drag-and-drop the activity indicator and the button directly to the main workspace.

For this recipe, we inserted a button that triggers and displays our activity indicator. Obviously, you should use this widget for those activities or actions that may need some time to complete. For instance, loading files from a server over the Internet could result in a delayed response in the user interface, especially over a mobile data connection.

In order to show and hide our activity indicator, we used CSS styles. Specifically, the `display` property was widget set to none in order to initially hide the activity indicator. After clicking on the button, the `showIndicator` JavaScript function selects our activity indicator using the `getElementById` method and makes the indicator visible by using the setting of CSS value `inline` for the `display` property. The `onclick` event handler of the button invokes the JavaScript function. We don't need to do anything more for displaying widget; the activity indicator *Dashcode* will do the rest automatically.

There's more...

Dashcode allows us to deploy our web application for iPhone to a specific server or location. In order to do that, you can click on the **Run & Share** button located on the left area of the main window of Dashcode. There you'll find a set of properties on the right side of the main window for choosing where to deploy your web application. For example, if we're going to deploy to a folder on our web server, the first step will be to select **localhost** as **Destination**. By default, the application will be deployed to a new folder with the same name of the project inside our `DocumentRoot` folder. Also note that Dashcode compresses all JavaScript files by default, but if you don't want this to happen unselect the checkbox **Compress JavaScript for faster download**. When you're ready, click on the button **Deploy** and the application will be deployed to your chosen destination.



Once your application is deployed, you can access it from a real iPhone device using the proper path. For our recipe, the URL is: `http://localhost/activity/`.

See also

- *Installing the Apple Dashcode framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

5
Mastering Sound and Music

In this chapter, we will cover:

- Making a beep alert
- Making a vibrate alert
- Creating an iPod playlist and playing a specific item
- Loading an iTunes playlist
- Playing an audio file
- Playing a video
- Recording an audio

Introduction

This chapter introduces you to a set of recipes about how to utilize the audio and video capabilities of the iPhone. Many users decide to use their smartphone as a multimedia player; therefore, developers will find how to build applications for playing video and audio files very interesting.

We'll start to find out how to make a simple beep sound from our application. Then, we'll discover another kind of notification for vibrating the device. After learning this, we'll delve into more advanced features related to iPod and iTunes Store. The last recipes of this chapter show how to play audio and video files with a few lines of JavaScript and HTML code.

Making a beep alert

The first recipe for this chapter is focused on how to play a simple beep sound as a response to an event or an action. Specifically, we'll play a sound when the user clicks on a simple button.

Usually, the beep sound is used for alerting the user. For example, you may want to launch an acoustic signal to indicate that pushing the button is an invalid action.

Instead of beeping, we can use another kind of notification such as an alert box as you've learned in *Chapter 2*, *Building Interfaces.*

Getting ready

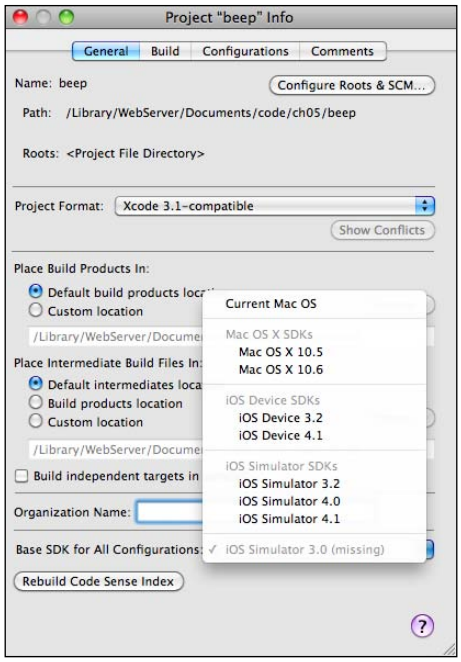
This recipe requires the use of two different frameworks, *UiUIKit* and *PhoneGap*. *UiUIKit* will be used for building a consistent interface. *PhoneGap* will allow us to interact with the native hardware and play the beep sound.

Remember, you'll need to use a Mac OS X computer that has Xcode and the *iOS* SDK installed for building iPhone applications with *PhoneGap*.

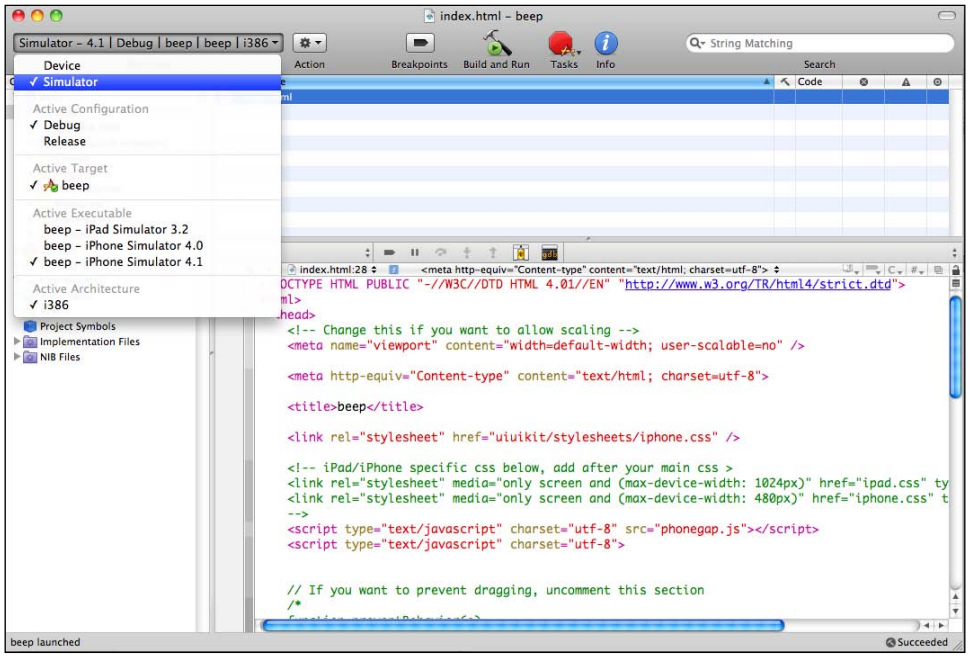
The complete code for this recipe can be found at `code/ch05/beep/` in the code bundle provided on the Packtpub site.

How to do it...

1. Our first task will be to create a new *PhoneGap* project in Xcode. In order to do that, launch Xcode and click on **File** | **New project**. Don't forget to select the **PhoneGap-based Application** template and click on the **Choose...** button. Then, choose a name for your new project and click on the **Save** button when you're ready.
2. The second task is focused on selecting the appropriate options for the build and run environments. Select the **Project** | **Edit Project Settings** menu option for the required configuration of these environments. You'll see a new dialog box, where you can select the base SDK for your project. Choose the last version for the *iOS* Simulator. The following screenshot displays the mentioned dialog box for these options:



3. Before continuing, be sure to select the **Simulator** option on the combobox located at the upper-left side of the main window of Xcode:



4. We're going to integrate the required code for the *UiUIKit* framework with our *PhoneGap* project. This action is pretty simple; you should only copy the files of the *UiUIKit* framework inside the `www` folder of your project. For example, suppose you've installed the *UiUIKit* framework in the `/Library/WebServer/Documents/uiuikit` and your new project is located at `/Library/WebServer/Documents/ch05/beep`. In this scenario, you only need to execute the following command from the terminal:

   ```
   $ cp -r /Libray/WebServer/Documens/uiuikit/
     /Library/WebServer/Documents/ch05/beep/www
   ```

5. Click on the **www** item of the **Group & Files** pane and click again on the `index.html` item. After this action, the main HTML file of our project will be opened inside the Xcode's editor. We're going to add a new code line after the `title` meta tag:

   ```
   <link rel="stylesheet"
             href="uiuikit/stylesheets/iphone.css" />
   ```

6. A new JavaScript function should be added inside the JavaScript section of `index.html`, so we'll set the following section before the `</script>` tag:

   ```
   function make_beep() {
     navigator.notification.beep(4);
   }
   ```

7. Let's go to the body section of our main file and replace it in the following code:

   ```
   <div id="header">
     <h1>Beep</h1>
   </div>
   <h1>Making beep</h1>
   <ul class="data">
     <li>
       <p>Click on the button for beeping</p>
     </li>
   </ul>
   <p>
   <a href="#" onclick="make_beep()" class="button
              white">Click me!</a>
   </p>
   ```

8. Before testing our new application, we'll need to save our changes by clicking on the **File | Save** menu option. After this operation, click on the **Build and Run** button for launching our application through the iPhone simulator tool.

How it works...

The *PhoneGap* framework offers a simple function for invoking a beep sound through the `notification` object. The `beep` method uses the number of times as a parameter to repeat the beep sound.

In order to play the beep, we're using a simple button with the `onclick` handler for invoking a specific JavaScript function, which calls the `beep` method. As we used `4` as a parameter, the beep will be repeated four times.

For designing a graphical user interface, which is consistent with the native look and feel of iPhone, we're applying some elements of the *UiUIKit* framework. Actually, we defined a `div` element with `id="header"` for building a main toolbar with a simple title. *UiUIKit* contains a specific unordered list (`<ul class="data">`) for displaying some text. Finally, our button for playing the beep was built through an anchor element applying two different CSS styles, `white` and `button`.

There's more...

Keep in mind that *PhoneGap* will build a native iPhone application, so you'll need to be a member of the iPhone Developer Program to install and run your application in a real device. The audio does not work in the *iOS* simulator.

In order to prevent dragging of our application, it could be interesting to uncomment the specific code located in the JavaScript section of the `index.html` file. Actually, the following JavaScript lines should appear in the mentioned section:

```
function preventBehavior(e) {
  e.preventDefault();
};
document.addEventListener("touchmove", preventBehavior, false);
```

A part of the `beep` method, the `notification` object of *PhoneGap* contains two more methods for launching notifications to the user. One of these is `alert` and the other one is `vibrate`. We've learned how to create alerts in the *Creating and customizing a notification box* recipe in *Chapter 2*, *Building Interfaces* and we'll find out how to use the `vibrate` method in the next recipe.

See also

- *Installing the PhoneGap* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the UiUIKit* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating and customizing a notification box* recipe in *Chapter 2*, *Building Interfaces*
- *Triggering a vibration* recipe

Making a vibrate alert

In the previous recipe, you've learned how to create a simple notification using a beep sound. Continuing with notifications, we're going to find out how to vibrate our device with JavaScript.

This recipe is very similar to the previous one. In fact, we'll only need to change one line of JavaScript code for testing our new kind of notification.

You can find the complete code for this recipe at: `code/ch05/vibrate/` in the code bundle provided on the Packtpub site.

Getting ready

As we've seen in the previous recipe, we'll need the *PhoneGap* and *UiUIKit* frameworks plus the Xcode IDE. Remember that this IDE requires a Mac OS X operating system installed on your computer. Also, it's important to keep in mind that you need to be a iPhone Developer Program member for installing and running your iPhone's *PhoneGap* application in a real and physical device.

How to do it...

1. Let's start reusing our code written for the previous recipe. Replace the `make_beep()` JavaScript function with the following new one:

```
function make_vibrate() {
  navigator.notification.vibrate(4000);
}
```

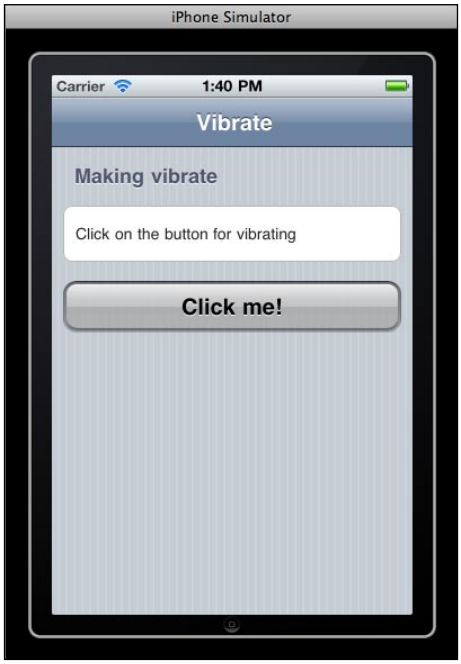
2. The next change will be to replace the name of the JavaScript function with the `onclick` handler of the anchor element. This element should refer to our new JavaScript function:

```
<a href="#" onclick="making_vibrate()" class="button
          white">Click me!</a>
```

3. In order to be consistent, we'll change our header and the unordered list:

```
<div id="header">
  <h1>Vibrate</h1>
</div>
<h1>Making vibrate</h1>
<ul class="data">
  <li>
    <p>Click on the button for vibrating</p>
  </li>
</ul>
```

4. Now we're ready for testing. Save your new project and run it by clicking on the **Build and Run** button located at the top of the main Xcode window.



5. Push the **Click me!** button for testing the device's vibration effect.

How it works...

Triggering a vibration is pretty simple, thanks to the vibrate method of the `notification` object offered by *PhoneGap*. This method expects as parameter the number of milliseconds to vibrate the device. We used the value `4000` for four seconds.

As we've seen in the previous recipe, a simple button is used, which calls the code to trigger the vibration. An additional benefit of using *UiUIKit*, is that it helps us to build the user interface through a simple `div` element, an unordered list and an anchor HTML element.

See also

- *Installing PhoneGap* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing UiUIKit* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating and customizing a notification box* recipe in *Chapter 2*, *Building Interfaces*
- *Making a beep alert* recipe

Creating an iPod playlist and playing a specific item

As other modern smartphones, iPhone has the capability to reproduce audio files in different formats as MP3 and AAC. Also, other devices running the *iOS* operating systems—as iPad and iPod Touch—include the same feature. Keeping in mind this fact, it would be interesting to create a list with some items in a similar way to the iPod application included in iPhone.

This recipe shows how to create a simple list with some items that can be played when the user clicks on each one. For each item in the list, we'll display information about it as the title and duration of each song.

For simplicity, we're going to use some preview songs from iTunes as items for our list.

Getting ready

Check whether you've installed the *iWebKit* framework on your computer or not. We're going to use this framework for creating our playlist.

How to do it...

1. Open your favorite editor and create a new file called `ipod_playlist.html`. The first task is to add a standard header and the CSS and JavaScript files required for *iWebKit*. Specifically, these lines are as follows:

```
<link href="../iwebkit/css/style.css" rel="stylesheet"
          media="screen" type="text/css" />
<script src="../iwebkit/javascript/functions.js"
          type="text/javascript"></script>
```

2. Before closing our head tag, we'll insert a `<title>` tag for our application:

```
<title>iPod Playlist</title>
</head>
```

3. Now, it's time to create the body with a specific class for it:

```
<body class="ipodlist">
```

4. As you've learned in the previous recipes related to *iWebKit*, we'll include a top bar inserting the next code:

```
<div id="topbar">
  <div id="title">iPod Playlist</div>
</div>
```

5. The required code for our complete list with a few items is the following:

```
<div id="content">
<ul>
  <li>
    <a class="noeffect" href="javascript:document.song01.Play();">
      <span class="number">1</span>
      <span class="auto"></span>
      <span class="name">Beautiful Day</span>
      <span class="time">4:08</span>
    </a>
  </li>
  <li>
    <a class="noeffect" href="javascript:document.song02.Play();">
      <span.class="number">2</span>
      <span class="auto"></span>
      <span.class="name">Stuck in a Moment You Can't Get Out</
span>
      <span class="time">4:32</span>
    </a>
  </li>
  <li>
```

```
    <a class="noeffect" href="javascript:document.song03.Play();">
      <span.class="number">3</span>
      <span class="auto"></span>
      <span class="name">Elevation</span>
      <span class="time">3:47</span>
    </a>
  </li>
  <li>
    <a class="noeffect" href="javascript:document.song08.Play();">
      <span class="number">8</span>
      <span class="auto"></span>
      <span class="name">Peace on Earth</span>
      <span class="time">4:48</span>
    </a>
  </li>
</ul>
</div>
```

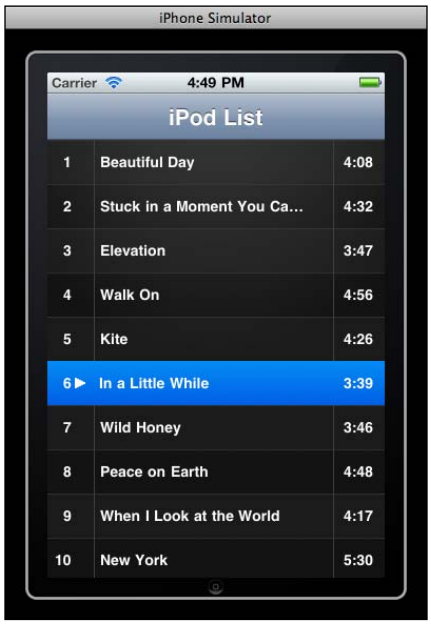
6. Additionally, we'll need the reference to the location of each song providers for iTunes. The next code should be added to our new file:

```
<embed enablejavascript="true" autoplay="false"
height="0"name="song01"
width="0"src="http://a1.phobos.apple.com/us/r1000/014/
Music/38/30/97/mzm.owzkqqkq.aac.p.m4a"/>
<embed enablejavascript="true" autoplay="false" height="0"
          name="song02" width="0"
src="http://a1.phobos.apple.com/us/r1000/042/Music/ea/06/a
          e/mzm.xxnpunuc.aac.p.m/>
<embed enablejavascript="true" autoplay="false" height="0"
name="song03" width="0"
src="http://a1.phobos.apple.com/us/r1000/035/Music/2f/a7/0
          f/mzm.gvewsftw.aac.p.m4a"/>
<embed enablejavascript="true" autoplay="false" height="0"
          name="song11" width="0"
src="http://a1.phobos.apple.com/us/r1000/020/Music/ef/29/e
          5/mzm.kuaoqjps.aac.p.m4a"/>
```

7. Load your new application in your device or iPhone simulator for testing our list. After this action, your screen will display a list, as shown in the following screenshot:

8. If you click on one of the items of the list, the iPhone's player will be launched and the songs will be played. The following screenshot shows the launched player:

9. When the user clicks on the **Done** button, our list will be displayed selecting the recent played item:



How it works...

The *iWebKit* offers a predefined CSS class. When added to the body element, it styles a playlist similar to the original one used by the iPod application included in the iPhone. The name of the mentioned CSS class is `ipodlist` and it defines how the playlist should be displayed, including colors and lines working as separators between items. Lastly, we need to define the details of each song in an unordered HTML list.

Each element of the unordered list contains an anchor element and different `span` tags for displaying some attributes such as the number, title, and duration of each song. The most important of these elements is the anchor, which refers to a specific embed object. At the end of the file, you can find one of these embed objects per song. Actually, we're using this specific tag of *Safari Mobile* to establish a reference to the original audio file served by iTunes. The `embed` tag allows you to play content through the QuickTime infrastructure included in the iPhone. In fact, we can consider each of these tags as a pointer for playing content in QuickTime. The tag contains different attributes such as `autoplay`, `height`, and `width`. For our recipe, we used `autoplay="false"` because the content will be played only when the user clicks on an item in the unordered list. On the other hand, we want to hide each `embed` tag, so we put `0` value for `width` and `height`. The last and an important attribute is `src`, which is a hyperlink to the physical file served by iTunes.

Once we've completed our list and all the links for the items, we should add a handler to each of these items. We chose a simple `onclick` handler to indicate what will happen when the user clicks on the item. Our recipe is using one line of JavaScript code for playing each of these items. Therefore, we only need to select each item and play it using the `Play()` method. This action happens thanks to the `document.<name of element>.Play()` line. For example, if we want to play the third item, we only need to call `document.song03.Play()`, where `song03` is the value for the name attribute of the third item of our list.

Inside each `<li>` element, you'll find a blank `span` tag with a CSS class called `auto`. This span tag will add an arrow when a specific sound is playing. You can change the default behavior using two different classes, such as stop and play. The second one shows the mentioned arrow and the first one doesn't show anything.

Even though we have used a default toolbar, you can apply a CSS style to be more consistent using a black color for it. Just add `class="black"` for the `div` element with `id="topbar"`. The result of this change is visualized in the following screenshot:

There's more...

Perhaps you're wondering how to get direct links from iTunes. Fortunately, Apple offers a web page for searching songs; take a look at it by directing your web browser to: `http://itunes.apple.com/linkmaker`.

See also

- *Installing the iWebKit* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Loading an iTunes playlist

In this recipe, we're going to learn how to create a playlist with the elements which point to the iTunes Store. Instead of the standard iTunes application installed in iPhone, we can build our own customized playlist, which offers a way to access some related items inside the iTunes Store.

This kind of playlist can be very useful for a specific author, distribution company, or record label interested in building a specific iPhone application such as a catalogue for selling songs or albums through iTunes.

The HTML file for this recipe can be found at `code/ch05/itunes.html` in the code bundle provided on the Packtpub site.

Getting ready

We continue using the *iWebKit* framework because it offers specific CSS classes for helping us.

How to do it...

1. As in the previous recipe, we'll need to create a standard HTML file, adding the required references for the *iWebKit* framework. You can reuse the file created in the previous recipe. In order to do that, save the `itunes_playlist.html` as `itunes.html` and delete the complete body tag. After taking this action, add the following lines for the new body tag:

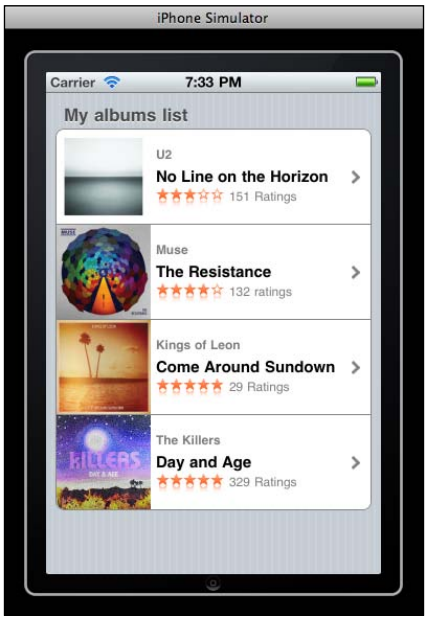
```
<body>
<div id="content">
  <span class="graytitle">My albums playlist</span>
  <ul class="pageitem">
    <li class="store">
      <a class="noeffect"
               href="http://itunes.apple.com/us/album/no-line-
               on-the-horizon/id305352505">
        <span class="image" id="album01"></span>
        <span class="comment">U2</span>
        <span class="name">No Line on the Horizon</span>
        <span class="stars3"></span>
        <span class="starcomment">151 Ratings</span>
```

```
            <span class="arrow"></span>
          </a>
        </li>
        <li class="store">
          <a class="noeffect"
                   href="http://itunes.apple.com/us/album/the
                   -resistance/id326492721">
            <span class="image" id="album02"></span>
            <span class="comment">Muse</span>
            <span class="name">The Resistance</span>
            <span class="stars4"></span>
            <span class="starcomment">132 ratings</span>
            <span class="arrow"></span>
          </a>
        </li>
        <li class="store">
          <a class="noeffect"
                   href="http://itunes.apple.com/us/album/come-around
                   -sundown/id390973055">
            <span class="image" id="album03"></span>
            <span class="comment">Kings of Leon</span>
            <span class="name">Come Around Sundown</span>
            <span class="stars5"></span>
            <span class="starcomment">29 Ratings</span>
            <span class="arrow"></span>
          </a>
        </li>
        <li class="store">
          <a class="noeffect"
                   href="http://itunes.apple.com/us/album/day-age
                   -deluxe-version/id296905600">
            <span class="image" id="album04"></span>
            <span class="comment">The Killers</span>
            <span class="name">Day and Age</span>
            <span class="stars5"></span>
            <span class="starcomment">329 Ratings</span>
            <span class="arrow"></span>
          </a>
        </li>
      </ul>
    </body>
```

2. Come back to the head section of your new file and add the new CSS style code:

```
<style type="text/css" media="screen">
  #album01 {
    background-image:
url('http://a1.phobos.apple.com/us/r1000/005/Music/82/b4/94/mzi.
wnujigmg.170x170-75.jpg')
  }
  #album02 {
    background-image: url('http://a1.phobos.apple.com/us/
r1000/007/Music/79/84/54/mzi.alpkhceo.170x170-75.jpg')
  }
  #album03 {
    background-image: url('http://a1.phobos.apple.com/us/
r1000/041/Music/fb/aa/da/mzi.aoktfsej.170x170-75.jpg')
  }
  #album04 {
    background-image: url('http://a1.phobos.apple.com/us/
r1000/034/Music/6b/da/4e/mzi.cpebjtsm.170x170-75.jpg')
  }
</style>
```

3. Your new application is ready to use. It's better to test our new application in a real device such as the iPhone or iPod Touch instead of using the iPhone simulator. The reason for it is simple; the simulator cannot access the iTunes Store. After loading your new file, you can see our playlist, as shown in the next screenshot:



4. If you click on one item, the iTunes application will be opened and it takes you to the specific iTunes Store page, where the user can purchase the album. For example, if the user clicks on the second item of our playlist, the device will display the album's page, as shown in the following screenshot:

How it works...

We built a simple playlist with four different items displaying an album's art and some of its details, which can be found in the iTunes Store. In fact, we only have some links ready to directly open the iTunes application installed in the device for displaying the specific page of the store for each one. From this page, the user can purchase a complete album or only some songs.

The *iWebKit* framework is used for building a consistent interface with a native look and feel that is closer to native applications. An unordered list gives the *iWebKit* defined CSS class (`pageitem`). The albums are the items of this list and each are given a CSS class called `store`. This *iWebKit* defined style divides the area for each album in two different areas: one for displaying the artwork and another for showing information such as the album's title and author.

Each `<li>` element of our unordered list contains an anchor element where its `href` attribute points to the direct link of the iTunes Store. Also, we can find some `span` elements inside each of these links. Specifically, we are using the following six `span` tags:

1. The first one is used for the artwork image of the album, which is a link to the physical image stored in the iTunes Store. The CSS class called `image` is used for this `span` element.
2. Album's author or artist: The CSS class `comment` should be used for displaying this information.
3. Name of the album: For this `span`, we're using the name class defined by *iWebKit*.
4. An empty `span`, which indicates the rating of the album through the `start<number>` class, where `<number>` is an integer between 1 and 5.
5. Number of ratings for this album: A new `span` is used for this purpose using the `starcomment` class and some text inside the element.
6. Each item has a small arrow on the right of the information displayed for each album. The `arrow` class manages this graphic arrow.

Surely, you've realized that we're not using an explicit link for each artwork image inside the `span` for it. However, each `span` class using the image `class` has an `id` attribute with a value. We used CSS for setting a background image for each of these `span` elements.

By default, *Safari Mobile* will open the related links to iTunes Store—belonging to the `itunes.apple.com` domain—using the specific iTunes application installed in the device.

See also

- *Installing the iWebKit* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating an iPod playlist and playing a specific item* recipe

Playing an audio file

This recipe will describe the process of playing an audio file from a web application designed specifically for the *iOS* operating system. We'll learn three different ways for playing a file with our various frameworks.

This recipe shows a simple method for playing audio, but you can integrate it with your own code or application for building more sophisticated applications.

We're going to play an audio file in MP3 format; however, other audio formats, such as WAF and AAC-LC, can be played as well.

Our recipe describes how to display a simple player with controls for playing and pausing a specific and preloaded MP3 file.

For one of the three approaches explained in our recipe, we don't need any framework because we'll take advantage of the HTML5 support of the *Safari Mobile* browser installed in the *iOS* devices.

The *UiUIKit* framework will be used for defining a simple layout for our *PhoneGap* and HTML5 approaches. The reason for it is pretty simple; we get a consistent graphic interface with a native look and feel for our web application using a few lines of HTML code.

The complete code for each of the approaches explained in this recipe can be found in the following directories in the code bundle provided on the Packtpub site:

- `code/ch05/play_audio/`
- `code/ch05/play_sencha/`
- `code/ch05/play_phonegap/`

Getting ready

As we're going to use multiple frameworks in this recipe, you should make sure that *PhoneGap*, *Sencha Touch*, and *UiUIKit* are installed in your machine. Don't forget that the *PhoneGap* application for iPhone requires Xcode installed in a Mac OS X computer. On the other hand, you'll need an audio file in WAV, MP3, AAC-LC, or HE-ACC format. For example, our recipe works with an MP3 file.

How to do it...

First, we'll describe the process for playing an audio file using the native capabilities of HTML5 implemented by *Safari Mobile*. In the second example, we're going to the next stage using *Sencha Touch*. Finally, you'll learn how to do the same with *PhoneGap*.

Using HTML5

1. Create a new folder called `play_audio` inside your `DocumentRoot` directory. Remember, usually this directory is `/Library/WebServer/Documents/` under Mac OS X. Other operating systems, such as Ubuntu, use a different folder, such as `/var/www/`.
2. Inside your new folder, create a new file called `play_audio.html` and add the following lines of HTML code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta content="yes" name="apple-mobile-web-app-capable"/>
  <meta content="text/html; charset=utf-8" http
              -equiv="Content-Type" />
```

```
  <meta content="minimum-scale=1.0, width=device-width,
                maximum-scale=0.6667, user-scalable=no"
                name="viewport" />
  <link rel="stylesheet"
            href="../../uiuikit/stylesheets/iphone.css" />
  <link rel="apple-touch-icon"
            href="../../uiuikit/images/apple-touch-icon.png"/>
</head>
<body>
  <div id="header">
    <h1>Playing audio</h1>
  </div>
  <ul class="data">
    <li><audio src="frogs.mp3" controls></audio></li>
  </ul>
</body>
</html>
```

3. After saving your new file, load it from your device. The screen will display something similar to the next screenshot:

4. If you play the arrow controller, the audio will start to play and the widget player will display the duration of the songs and a progress indicator as shown in the next screenshot:

Using Sencha Touch

1. As you've learned in previous recipes using *Sencha Touch*, we need to create three different files inside a new directory, which will be located inside the `DocumentRoot` of your web server. For example, if you're using Ubuntu, execute the following command for creating our new directory:

   ```
   $ sudo mkdir /var/www/play_sencha
   ```

2. After creating our new directory, open your favorite editor, copy the following code and save the file as `index.html`:

   ```
   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
   www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
   <html xmlns="http://www.w3.org/1999/xhtml">
   <head>
     <meta http-equiv="Content-Type" content="text/html; charset=utf-
   8">
     <title>Playing audio</title>
     <link rel="stylesheet" href="../../sencha
               -touch/resources/css/ext-touch.css" type="text/css">
     <script type="text/javascript" src="../../sencha
                  -touch/ext-touch-debug.js"></script>
     <link rel="stylesheet" href="main.css"
               type="text/css">
     <script type="text/javascript"src="main.js"></script>
   </head>
   <body>
   </body>
   </html>
   ```

3. Now, it's time to create our main JavaScript file called `main.js` with the next code:

```
Ext.setup({
  onReady: function() {
    new Ext.Panel({
      fullscreen: true, items: [{
        xtype: 'audio', url: "frogs.mp3",
             html: '<p id="title">Playing Audio</p>}],
      layout: {
        type: 'vbox', pack: 'center'
      },
    });
  }
});
```

4. The next step will be to create a new file for applying some style to our application. Create a file called `main.css` and add the following lines of code:

```
body {
  background: rgb(197,204,211);
}
#title {
  margin-left: 46px;
  margin-bottom: 8px;
  padding: 0px;
  font-family: Arial;
  font-size: 1em;
  font-weight: bold;
}
```

5. Your application is ready for testing. Load the `play_sencha/index.html` on your device and you can see a screen similar to the next screenshot:

6. If you click on the arrow icon of the player widget, the song will start to play and the progress indicator will change, as shown in the following screenshot:

Using PhoneGap

Let's start by creating a new *PhoneGap* project through Xcode using the specific template for this kind of project. The process is the same as explained in the previous *PhoneGap* recipes. Basically, we need to choose the appropriate template and a name for our project. After taking this action, we'll configure our project to use the simulator as our base SDK configuration.

The main HTML file `index.html` for our project is located in the `www` directory. Open this file in the editor and take the following actions:

1. Delete the commented lines with the CSS references in the head section.
2. Uncomment the JavaScript function called `preventBehaviour` and the line of code above this function.
3. Add the following line inside the `onDeviceReady` JavaScript function:

```
myAudio = new Media("media/frogs.mp3");
```

4. Add the next JavaScript function inside the JavaScript section:

```
function play() {
  myAudio.play();
  var btnObj = document.getElementById("btn");
  btnObj.innerHTML = "Stop";
  btnObj.onclick = function (){ myAudio.stop(); };
}
```

5. Insert the next lines of HTML code inside the `body` section:

```
<div id="header">
  <h1>Playing Audio</h1>
</div>
<ul class="data">
  <li><p>Click on the button for playing</p></li>
</ul>
<p>
  <a href="#" id="btn" onclick="play()" class="button
    white">Play!</a>
</p>
```

6. Your code is ready at this point, but we need to copy the required files for the *UiUIKit* to the `www` directory and your audio file to a new folder named `media`, which should be inside the `www` folder. After copying these files, we're ready for loading our new application. The **Build and Run** button of Xcode will launch the application in the iPhone Simulator. The main screen of our application will look similar to the following screenshot:

7. If the user clicks on the **Play!** button, the audio file starts to play and the title for our button will be changed to **Stop**. Click on the button again to stop playing:

How it works...

In order to explain properly how each of the approaches for our recipe work, we're going to use three different sections as follows:

HTML5 approach

Using this approach, we only need one file plus the required files for the *UiUIKit* framework since *UiUIKit* defines the interface. Because we take advantage of the HTML5 capabilities of *Safari Mobile*, we actually don't need any framework to play audio in the browser.

The `<audio>` tag handles our file allowing to play it. We used only two attributes: `src` and `controls`. The first one is a link to our physical MP3 file and the other sets icons for playing and stopping our audio.

Regarding the graphic interface, we selected a simple toolbar plus an unordered list for our player widget. The *UiUIKit* framework provides a consistent look and feel, thanks to its `iphone.css` file.

Remember we used a MP3 file, but *Safari Mobile* supports other audio formats such as WAF, AIF, AAC-LC, and HE-AAC. Apart from these audio formats, the iPhone, iPod Touch, and iPad can play different types of files.

HTML5 introduces other attributes for the `<audio>` tag. Among these, we can find `autoplay` and `loop`. The `autoplay` attribute can be used for playing the audio automatically after loading the application. On the other hand, `loop` allows playing audio indefinitely. In order to understand how it works, you can add the attribute `autoplay` to our audio tag. This attribute doesn't need any value, it means that if the attribute is present, the audio will play automatically.

Sencha Touch approach

A *Sencha Touch* application requires two files at least: one for the JavaScript code defining the user interface and the behavior of the application and another one with the HTML code for invoking the mentioned JavaScript file. In addition to these files, our application uses one more file for applying some style to our user interface. Actually, the `main.css` file applies a background color and sets the style for playing an audio file.

For playing audio, *Sencha Touch* contains a specific widget called `Ext.Audio`. Instead of calling it explicitly, we're using the items array inside a main `Ext.Panel`. The `xtype` with the `audio` value indicates this kind of object. Additionally, we need the `url` attribute for indicating which file will be played. Our recipe is using the attribute `html` for adding a small title above our widget. On the other hand, the `layout` property of our main panel establishes the use of a `vbox` container, where all the elements inside it should be aligned at the center.

As you can observe, the `index.html` file for this recipe is a simple XHTML file with the required references to `main.js`, `main.css`, and the other files required by *Sencha Touch*.

From the technical point of view, the `Ext.Audio` class encapsulates an HTML5 audio container. This is the reason the widget in this example looks identical to the HTML5 example.

The `Ext.Audio` component that is defined by *Sencha Touch* can use other different properties. For example, we can choose the height and width of the widget or apply a border to it. It's even possible to start playing our file automatically after loading the widget through the `autoPause` property. Apart from these properties, some methods can be invoked for instantiated object of this component. The most useful are `play` and `pause`, both of them can be used dynamically.

PhoneGap approach

Creating a new *PhoneGap* project requires the same initial steps. The process is pretty simple and you should do most of the job inside the main `index.html` file. In this case, we're creating a new JavaScript function for handling our audio file. As you learned in other *PhoneGap* related recipes, the `onDeviceReady` function is run after the framework is loaded. Therefore, we declare our main object inside this function. Specifically, we're using an object of the `Media` class predefined by *PhoneGap.* The constructor of this class expects the URL for our audio physical file.

The graphical user interface of our application is defined using the style provided by the *UiUIKit* framework. Actually, we built a simple button through an anchor element, which has an event handler for invoking the `play()` function. This JavaScript function will be responsible for playing and stopping our audio. When the user clicks on the main button of our application, the audio will start to play. Additionally, the function changes the label of the button to `Stop` and replaces the event handler for it, adding a new call for invoking the `stop()` method of our `myAudio` object.

The constructor of the `Media` object can receive two different callbacks as parameters. One of them is for taking some actions if an error occurs. The other one allows you to execute specific code when the component is created. In practice, it's very useful to use the callback for errors to indicate to the user that something unexpected is happening. The complete syntax for this constructor method is `new Media(src, OKfunction, errorFunction)`. Obviously, we'll need to define these JavaScript callback functions in our HTML main file or inside other JavaScript files, which should be referenced from the main HTML file. The callback function that handles the possible error receives as parameter a predefined object called `MediaError`, which has two main properties: `code` and `message`. The first one contains a predefined code by *PhoneGap* and the other property provides a text describing the error. From the technical point of view, `code` is a constant defined through a simple string.

In addition to the `play()` and `stop()` main methods, the JavaScript `Media` class provides another useful method called `pause()`, allowing to pause our audio file.

There's more...

Apple offers a basic syntax reference for the audio tag at: `http://developer.apple.com/library/safari/documentation/AudioVideo/Conceptual/Using_HTML5_Audio_Video/AudioandVideoTagBasics/AudioandVideoTagBasics.html#//apple_ref/doc/uid/TP40009523-CH2-SW4`.

The complete reference for the `Ext.Audio` of the *Sencha Touch* can be found at: `http://dev.sencha.com/deploy/touch/docs/output/Ext.Audio.html`.

Don't forget that you need to become a member of the iPhone Developer Program in order to install your *PhoneGap* application in a real Apple device.

See also

- *Installing the PhoneGap* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the Sencha Touch* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the UiUIKit* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating and customizing a notification box* recipe in *Chapter 2*, *Building Interfaces*
- *Playing a video* recipe

Playing a video

This recipe is focused on playing video in our web applications for *iOS* devices. You'll learn how to display a widget that has the ability to play a video file. We're going to use a video in MPEG-4 format, but other video formats such as H.264 can be used as well.

For our goal, you'll learn two different methods. The first method only uses the HTML5 capabilities of *Safari Mobile* and the other method requires the *Sencha Touch* framework.

The code for the HTML5 method can be found at `code/ch05/video5/`. Regarding the *Sencha Touch* approach, the code can be found at the `code/ch05/video_sencha/` folder in the code bundle provided on the Packtpub site.

Getting ready

For this recipe, you'll need to install the *Sencha Touch* and the *UiUIKit* frameworks on your computer.

How to do it...

Let's start taking the advantages of the HTML5 standard and then we'll continue using the *Sencha Touch* framework.

Using HTML5

Create a new file with the help of your favorite text editor and save the file as `video5.html`. Add the following lines of HTML code to your just created file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
<meta content="yes" name="apple-mobile-web-app-capable" />
<meta content="text/html; charset=utf-8" http-equiv="Content-Type" />
<meta content="minimum-scale=1.0, width=device-width, maximum-
scale=0.6667, user-scalable=no" name="viewport" />
<link rel="stylesheet" href="../../uiuikit/stylesheets/iphone.css" />
<link rel="apple-touch-icon" href="../../uiuikit/images/apple-touch-
icon.png" />
<style type="text/css" media="screen">
  #myVideo {
    margin-top: 10px
  }
</style>
```

```
</head>
<body>
  <div id="header">
    <h1>Playing Video</h1>
  </div>
  <video id="myVideo" src="video.m4v" controls width="300"
height="240"></video>
</body>
</html>
```

Before continuing, you should check that your video file is called `video.m4v` and it resides in the same directory as your `video5.html` file. Load your new file on your device to see a result, as shown in the next screenshot:

When the user clicks on the arrow, the video will start playing.

Using Sencha Touch

Our first task will be to create a new directory under your `DocumentRoot` folder. Inside this new directory, you should create a new file called `index.html` with the following code:

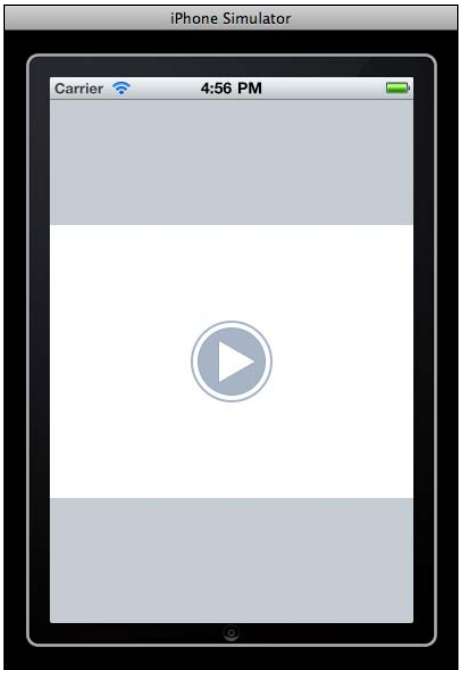
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <title>Playing audio</title>
```

```
    <link rel="stylesheet" href="../../sencha-
    touch/resources/css/ext-touch.css" type="text/css">
    <script type="text/javascript" src="../../sencha-touch/ext-touch-
    debug.js"></script>
    <style type="text/css" media="screen">
      body {
        background: rgb(197,204,211);
      }
    </style>
    <script type="text/javascript" src="main.js"></script>
  </head>
  <body>
  </body>
  </html>
```

Now, it's time to create the JavaScript file required by *Sencha Touch*. Its name will be `main.js` and it will contain the following lines of JavaScript code:

```
  Ext.setup({
    onReady: function() {
      new Ext.Panel({
        fullscreen: true,
        items: [{
          xtype: 'video',
          url: "video.m4v",
          width: 320,
          height: 240}],
        layout: {
          type: 'vbox',
          pack: 'center'
        }
      });
    }
  });
```

Test your new application by loading the file in your device, where you'll see a screen similar to the next screenshot:



As in the previous method for this recipe, after clicking on the arrow, the video will start playing.

How it works...

Despite the differences among frameworks, this recipe uses the capabilities of HTML5 for video playing in both methods. The reason for this fact is simple: *Sencha Touch* offers an encapsulated object for the HTML5 `<video>` tag container through the `Ext.Video` class. In fact, we declared an object of this class inside our main `Ext.Panel` object using the items array where the value for `xtype` is `video`. The `width` and `height` properties passed to the same array are used for defining the video size on the screen. These are the same names used by the `<video>` tag of HTML5, where the size is defined through attributes. The other common property is the location of the video file. `Ext.Video` uses the `url` property for it and the same job is done by the `src` attribute of the `<video>` tag.

In addition to properties and attributes mentioned before, others can be used with different purposes. For example, the `controls` attribute allows the user to display the typical arrow for playing. The other example is the `poster` attribute of the `<video>` tag, which is used for displaying an image before playing the video. The `posterUrl` property of `Ext.Video` has the same purpose. If we add `poster="quicktime-logo.gif"` to our `<video>` tag in `video5.html` file and reload our page again, we'll see the following result:

When the video starts to play, a toolbar with a set of controls will automatically appear at the bottom of the screen. At the top of the screen, we can see another toolbar with information about the duration of the file, a progress bar, and a standard **Done** button. After a few seconds, these toolbars will disappear but if the user clicks on the screen, the toolbars appear again. Developers don't need to worry about implementing this functionality; the operating system and the browser handle it automatically. The next two screenshots show you these different states during the playback:


iPhone Simulator
Reset
Retry
Restart
Reinstall
Restore
Press and hold Center and Menu buttons for 6 seconds


iPhone Simulator
Carrier
4:57 PM
Done
0:27
-0:58
Reset
Retry
Restart
Reinstall
Restore
iMac

Regarding the video formats, *Safari Mobile* can play MPEG-4, H.264, and QuickTime video encoded with H.264. It's important to keep in mind that the browser requires MP3, WAV, or ACC formats for the audio of these different video formats.

There's more...

Apple offers a set of video samples in different formats such as MPEG-4 and H.624. You'll find these samples at `http://support.apple.com/kb/HT1425?viewlocale=en_US%20Video`.

If you're interested in the complete options of the `Ext.Video` class of *Sencha Touch*, you can find the complete reference at `http://dev.sencha.com/deploy/touch/docs/output/Ext.Video.html`.

See also

- *Installing the Sencha Touch* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the UiUIKit* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating and customizing a notification box* recipe in *Chapter 2*, *Building Interfaces*
- *Playing an audio file* recipe

Recording an audio

As you probably know, the iPhone can be used as a sound and voice recorder. Surely, some developers need to integrate this functionality inside their applications. In this recipe, you'll learn how to build a native application using JavaScript, CSS, and HTML code with the ability of recording sound. Yes, you guessed it; we're going to use the *PhoneGap* framework because it has the advantages of compiling to a native application.

The application for this recipe will display three different buttons: one for recording, an other for stopping, and a third for playing our new recording. For simplicity, we will only write the code for accepting these actions in this order. In a production application, we would have to consider what happens if the user clicks on the **Play** button before the sound is recorded.

You'll find the Xcode's project code for this recipe at the `code/ch05/recording/` folder in the code bundle provided on the Packtpub site.

Getting ready

The requisites for this recipe are the *PhoneGap* and *UiUIKit* frameworks, the Xcode IDE, and the *iOS* SDK installed in a Mac OS X computer. As we're going to build a native application, you will need to become a member of the *iPhone Developer Program* to install it in real Apple devices, such as the iPhone or iPod Touch.

How to do it...

Launch *Xcode* and create a new project named `recording` through the *PhoneGap* template provided by the main wizard of the IDE. Before continuing, copy the files of the *UiUIKit* inside the `www` folder of your *PhoneGap* project.

1. Open your `index.html` file located at `www` folder and the following JavaScript methods inside the JavaScript section of the mentioned HTML main file:

```
function record() {
  myAudio.startAudioRecord();
}

function stop() {
  myAudio.stopAudioRecord();
}

function play() {
  myAudio.play();
}
```

2. Let's go to the head section of `index.html` file and add the next line:

```
<link rel="stylesheet" href="uiuikit/stylesheets/iphone.css" />
```

3. The next step will be to uncomment the JavaScript lines for preventing the default scroll behavior.

4. Right now, it's time to add our HTML code for defining the graphical user interface including the mentioned buttons. Just add the following code between the opening and closing `<body>` tags:

```
<div id="header">
  <h1>Recording</h1>
</div>
<p>
  <a href="#" id="btn" onclick="record()" class="button
    white">Record</a>
  <a href="#" id="btn" onclick="stop()" class="button
white">Stop</a>
  <a href="#" id="btn" onclick="play()" class="button
white">Play</a>

</p>
```

5. Finally, we can test our new *PhoneGap* application for *iOS* devices. After building and running the application on your device, you'll see a screen similar to the next one:

How it works...

For this recipe, we used the `Media` class defined by *PhoneGap* for handling audio files. In one of the previous recipes in this chapter, we also used the same JavaScript class for playing an audio file. However, for this new recipe, we're applying different methods allowing us to start and stop a sound record.

As you can observe, our three buttons have different calls in their event handlers. They all utilize the `onclick` event handler and each invokes a different JavaScript function when clicked. The **Record** button will call the record function, which is defined in the JavaScript section in the head area of the `index.html` file. This function calls to the `startAudioRecord()` method of the `Media` object provided by *PhoneGap*. For stopping our record, we only need to click on the **Stop** button, which will invoke the `stopAudioRecord()` method defined inside our `stop()` function. Finally, the `play()` function will be invoked through the **Play** button. We built this simple JavaScript function to encapsulate the call to the `play()` method of our `Media` object called `myAudio`. Obviously, this object exists because we created it in the `onDeviceReady()` initial function required for the *PhoneGap* applications.

As we mentioned previously, we don't administer the potential combinations of button clicks, meaning that you must test the application by clicking in the following order:

1. Click on the **Start** button.
2. After a few seconds, click on the **Stop** button.
3. When you're ready to listen to your record, click on the **Play** button.

Despite the fact that we don't need the *UiUIKit* framework, it is very useful for defining our interface with a few lines of HTML code. Perhaps, you prefer to use another framework to do that. No problem, it's your application and, of course, your choice.

See also

- *Installing the PhoneGap* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the UiUIKit* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating and customizing a notification box* recipe in *Chapter 2*, *Building Interfaces*
- *Playing an audio file* recipe

6
Exchanging Data: AJAX

In this chapter, we will cover:

- How to send HTTP requests
- Processing JSON responses
- Sending cross-domain requests

Introduction

Modern web applications make intensive use of AJAX. This is a concept that can be defined as a technique of using the JavaScript programming language for sending the HTTP request to a remote server without the need to reload the page. The main advantage of this technique is the speed, no doubt. We don't need to reload all the content of the page causing many extra requests. Other times, reloading a page implies that we carry out only one request but if this requests returns a lot of information – for example, 50 records with 10 fields for each one, we need more time. Apart from the speed, another clear advantage of AJAX is the savings in bandwidth consumption. If we make few requests, we're saving bandwidth. In fact, the user can perceive this savings bandwidth as a faster response.

AJAX is a set of technologies such as XML, XHTML, and the DOM model where JavaScript acts as glue between them. Thanks to JavaScript, developers can invoke server-side code from the client side avoiding unneeded additional requests.

One of the first problems for web developers is how to deal with AJAX when users work with different web browsers. Inspite of many efforts for using a standard way, the current web browsers employ different JavaScript objects for handling AJAX requests and responses. Within this scenario it could be very useful to use a framework for encapsulating this behavior. For iPhone applications we can use different frameworks, which offer us components for working with AJAX in an easier manner. Specifically, we're going to work with three of them: Sencha Touch, XUI, and WebApp.Net.

In this chapter, you'll learn the core of AJAX: how to send different HTTP requests and how to process responses. Pertaining to these responses, you find out how to deal with different and common data formats such as **JSON** (**JavaScript Object Notation**) and XML. One of the recipes of this chapter handles an important aspect of AJAX. We're talking about how to do cross-domain requests.

The server side plays a fundamental role in the AJAX technology because we need a component to respond to the request. In practice, we can use any of the current server-side technologies in our iPhone applications. This means that we'll need to get a set of backend or server-side components ready to interact with the client side. Currently, some of the most famous and extended of these technologies are PHP, Ruby on Rails, Django, and ASP.NET. A lot of providers offer hosting services based on one or more of these technologies. Probably, the production machine for serving your iPhone web applications has already installed the required software for applying the mentioned technologies.

The three recipes covered, simply attempt to show you the fundamentals of using AJAX in your iPhone web applications. The presented concepts are only the foundation of building complex and interactive applications.

How to send HTTP requests

In this recipe, you'll learn one of the most important issues of AJAX: how to send an HTTP request from the client to the server. Our client will be a simple HTML page with a button. When the user clicks on the button, we'll make a HTTP request to the server. After the request is sent, we'll display a simple message box. In order to keep our recipe as simple as possible, we're going to imagine that we have a backend – the server side – ready to respond to our request. Later in this chapter, you'll learn how to receive and process the client-side response offered by the server.

Using AJAX, we can make requests to the server over HTTP. This network protocol offers different methods such as GET, POST, PUT, and DELETE. Typically, the GET method is used for getting data or information and the POST method helps to produce persistent changes in the backend. For example, if we need to read data from the backend we'll use GET and for saving a new record, we'll do a request using the POST method. On the other hand, the use of PUT and DELETE methods is less common, although the **REST** (**Representational State Transfer**) architecture proposes the use of these methods for some specific operations affecting to

update and delete information in the backend. Currently, the REST architecture is widespread for developing web services. Our recipe will explain how to do a request using the GET method. However, it's simple to use other HTTP methods, as you'll see in this recipe.

The Sencha Touch and the XUI frameworks offer a simple and effective way for doing HTTP requests so we'll use both for this recipe. Actually, you'll find two different approaches based on these frameworks.

The complete code required for this recipe can be reached at `code/ch06/xui_get` and at `code/ch06/sencha_get` in the code bundle provided on the Packtpub site.

Getting ready

You need Sencha Touch, XUI, and UIUiKit installed on your machine. We won't process the request but it will be useful to get installed and ready a backend engine such as PHP, ASP.NET, Rails, or Django. As a web developer, you'll probably already have experience with one of these technologies. For this recipe, we'll use a simple PHP page so you'll need to install an interpreter of this programming language on your machine. Apache is a web server with an excellent support for PHP and one of the most widespread web technologies of the world.

How to do it...

We're going to start our recipe by describing the XUI approach. Then, we'll continue with the approach offered by Sencha Touch:

1. **Using XUI**: The first step is to create a new folder called `xui_get` under the control of your web server. Later, open your favorite text editor and create a new file called `index.html`. Then add the following lines of HTML code to your new file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html;
      charset=utf-8" />
    <meta name="apple-mobile-web-app-capable" content="yes" />
    <meta id="viewport" name="viewport" content="width=device-
      width, user-scalable=0;" />
    <title>GET request</title>
    <link rel="stylesheet"
      href="../../uiuikit/stylesheets/iphone.css" />
```

2. Now, we need to add some JavaScript code inside the header section:

```
    <script type="text/javascript" charset="utf-8"
      src="../../xui/xui-2.0.0.min.js"></script>
    <script type="text/javascript" charset="utf-8">
```

```
    var my_callback = function() {
      alert('Request was done!')
    }

    function do_request() {
      x$(window).xhr('remote.php', {callback:my_callback});
    }
    </script>
  </head>
```

3. Finally, we'll complete our main HTML file adding the following lines of code:

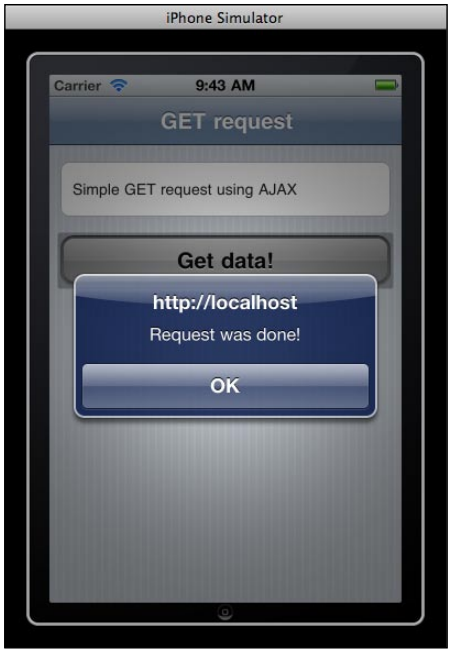
```
  <body>
    <div id="header">
      <h1>GET request</h1>
    </div>

    <ul class="data">
      <li>
        <p>Simple GET request using AJAX</p>
      </li>
    </ul>

    <p>
      <a href="#" onclick="do_request()" class="button white">Get
        data!</a>
    </p>

  </body>
</html>
```

4. The backend component will be a simple PHP page called `remote.php` so you need to create a new file with the next line of code:

```
<?php echo "done"; ?>
```

5. You can test it on your new application by pointing the browser of your iPhone at `http://<IP>/xui_get/` where `<IP>` is the IP address of your machine. The following screenshot shows you the new application:

6. After the user clicks on the main button, the HTTP request will be launched and a simple message box will be displayed as is shown in the next screenshot:



7. **Using Sencha Touch**: As you learned in previous recipes of this book, a Sencha Touch application requires at least two different files: one for HTML and another one for the JavaScript code. Before continuing, you should create a new folder called `sencha_get` under the `DocumentRoot` of your web server. Our main HTML file, `index.html`, will contain the following lines of code:

```
<link rel="stylesheet" href="../../sencha-touch/resources/css/ext-
touch.css" type="text/css">
<script type="text/javascript" src="../../sencha-touch/ext-touch-
debug.js"></script>
<style type="text/css">
  body {
    background: rgb(197,204,211);
    margin: 0;
    padding: 0;
  }
</style>
<script type="text/javascript" src="main.js"></script>
```

8. For the required JavaScript code, we're going to create a new file called `main.js` with the following lines:

```
Ext.setup({
  onReady: function() {
    var panel = new Ext.Panel({
      fullscreen: true,
      layout: {
        type: 'vbox',
        align: 'center',
        pack: 'center'
      },
      items: [{
        xtype: 'button',
        ui: 'normal',
        text: 'Get data!',
        handler: function() {
          Ext.Ajax.request({
            url: 'remote.php',
            success: function(response, opts) {
              alert("Request was done!");
            }
          });
        }
      }]
    });
  }
});
```

For simplicity, we'll use the same PHP file for this approach, so you can copy `remote.php` to your `sencha_get` directory. After taking this action, your new application will be ready and you can load it on your iPhone. The main page will be displayed as is shown in the next screenshot:

If you click on the button, the application will display a simple alert box as seen in the previous approach of this recipe.

How it works...

Let's explain how each one of the different approaches applied for this recipe work.

XUI approach:

The XUI framework uses a predefined method called `xhr` for carrying out HTTP/AJAX requests. This object is used in our `do_request` JavaScript function, which invokes the `remote.php` through the GET method. When the request is finished our `my_callback` function will be invoked. This function calls to the standard alert method with a simple message. You can observe how the `callback` option of the `xhr` method defines which function will be invoked. In this case, we're defining a JavaScript function (`my_callback`) as a variable. Actually, the `xhr` uses a callback mechanism to handle the response of the server to our request.

Despite the fact that we don't indicate which method should be used to make our request, the `xhr` method will do the job through the GET method. However, you can use the `method` option for indicating another HTTP kind of request. For example, if you need to use the POST method, simply modify the `xhr` method by adding the following option:

```
x$(window).xhr('remote.php', {callback:my_callback, method: 'post'});
```

Usually, the POST requests need to pass some parameters to the backend. For instance, let's think about an HTML form. The XUI framework offers an extra parameter in the `xhr` method for these cases. Actually, we can use the `data` option followed by the name and values required for the backend. For example, if we need to pass two parameters as `id` and `name`, we can use the following code:

```
x$(window).xhr('remote.php', {
   callback:my_callback,
   method: 'post',
   data: 'id=1&name = myname
});
```

By default, this framework uses `x$` as a selector for accessing the HTML objects. This is the reason for invoking to the `xhr` method through the selector for the window object.

On the other hand, we used the UIUiKit framework for defining a consistent interface for iPhone. As you can see in other recipes of this book, this framework helps us to create a button and a small box with text inside it.

Sencha Touch approach:

The core component for handling AJAX request to the Sencha Touch framework is `Ext.Ajax.request`. This component works with different parameters, `url` and `success` being the most important. The first one indicates the path to the server component and `success` requires a function for handling the result after the request is complete. In our recipe, we used a simple JavaScript function for calling to the standard `alert()` method. By default, the GET method is used for the `Ext.Ajax.request` component, however you can use other HTTP methods through the `method` public property of the aforementioned component. For example, if you want to use the POST method, you should add the `method:'POST'` string as parameter. In addition, you can pass additional parameters through the `params` option, which expects a set of names and values separated by a comma.

Regarding the main component of the user interface of our application, we used the `Ext.Panel` object, which contains a simple button. This button defines a handler function with the AJAX request and it will be invoked when the user clicks on the button.

The main HTML file contains a standard header plus a few lines of CSS code for setting the color of the background. Remember, Sencha Touch does the hard job through the main JavaScript file so the `index.html` file for our recipe is quite simple.

There's more...

For taking control over possible errors, `Ext.Ajax` offers the `requestexception` and `requestcomplete` methods. The complete reference for the `Ext.Ajax` component of Sencha Touch can be reached at `http://dev.sencha.com/deploy/touch/docs/output/Ext.Ajax.html`.

Interesting information for installing PHP in different web servers and operating systems can be found at `http://php.net/manual/en/install.php`.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the Sencha Touch framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the XUI framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Processing JSON responses

In the previous recipe we learned how to make HTTP requests. Usually, after we send a request we receive a response from the server and we need to process it. One of the most supported formats used with AJAX for exchanging data is JSON, so this recipe shows you how to process JSON responses using the Sencha Touch and XUI frameworks.

As in the previous recipe, we're going to use two different approaches: one for Sencha Touch and another for XUI. The functionality of both will be slightly different, although our goal of reading the JSON response is the same. For the Sencha Touch approach, we'll build a simple application with one button. When the user clicks on the button, a new HTTP request will be executed and the response will be processed. Finally, the result will be displayed using a standard JavaScript alert box. On the other hand, the XUI approach will display the result by populating two fields.

All the code needed for this recipe can be located at `code/ch06/sencha_json` and `code/ch06/xui_json` folders in the code bundle provided on the Packtpub site.

Getting ready

Review your web server and verify that you've installed the Sencha Touch, XUI, and UIUiKit frameworks. To keep it simple, we're going to reuse the same HTML files that we used in the previous recipe.

We don't need to write a server-side code for simulating a response; instead we may use a simple text file containing some JSON data. We assume that the reader has the minimum required knowledge for understanding the syntax of the JSON format.

How to do it...

Our first task is to create a new text file with the JSON response. This file will be used for the two mentioned approaches. Open your favorite editor and create a file called `data.json` with this line:

```
{"username":"Michael", "age":"35"}
```

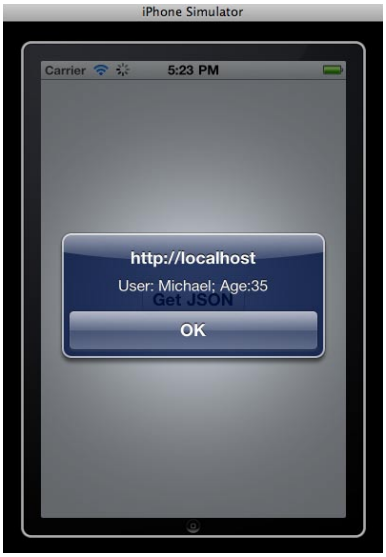
We're starting to get focus on the Sencha Touch approach. Later, we'll continue with the XUI framework.

Sencha Touch approach:

1. Create a new directory called `sencha_json` under the control of your web server. Then, copy the `index.html` file and the `main.js` file used in the previous recipe for the Sencha Touch approach in this new directory. Right now, take the `main.js` file and replace the content of the function in the `success` option for the following code snippet:

   ```
   var data = Ext.util.JSON.decode(response.responseText.trim());
   alert("User: " + data.username + "; Age:" + data.age);
   ```

2. Then change the value of the `text` option for `'Get JSON'`. Finally, you should replace the value for the `url` option as well. The new value will be `'data.json'`. You're ready for using your new application. When the user clicks on the **Get JSON** button, a new alert box will display the data read from our JSON file.



XUI approach:

1. First, we need to use a specific plugin for the XUI framework. Actually, this plugin is a JavaScript file called `xhrjson.js`. Point your browser to the following URL and download the mentioned file through the specific link:

   `https://github.com/xui/xui-plugins`

2. After downloading the file, you should copy it to the same directory where `xui-2.0.0.min.js` resides.

3. For this approach we'll use the same `index.html` file that we used in the previous recipe. However, we're going to create a new directory called `xui_json`. Start by editing the HTML file, adding the following line for loading our mentioned plugin:

   ```
   <script type="text/javascript" charset="utf-8" src="../../xui/
   xhrjson.js"></script>
   ```

4. After taking this action, replace the original JavaScript code used in the previous recipe with the following lines:

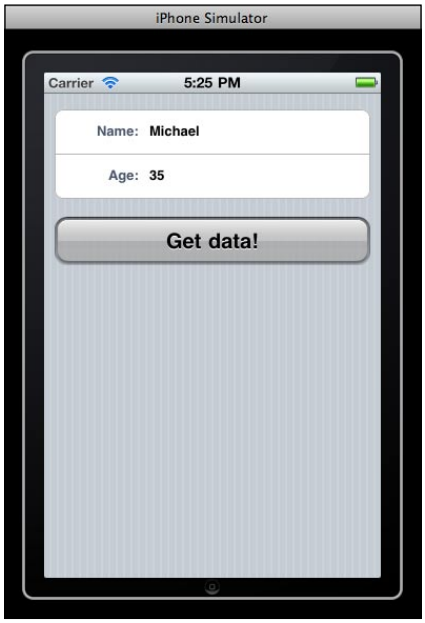
   ```
   function get_data() {
     x$(window).xhrjson('data.json',
       {map:{'username':'#name',
       'age':'#age'}})
     }
   ```

5. Now, it's time to modify the code for the user interface, so replace the code inside the `<body>` tag with the following:

   ```
   <ul class="field">
     <li>
       <h3>Name:</h3>
       <a href="#" id="name"></a>
     </li>
     <li>
       <h3>Age:</h3>
       <a href="#" id="age"></a>
     </li>
   </ul>
   <p>
     <a href="#" onclick="get_data()" class="button white">Get
     data!</a>
   </p>
   ```

6. We have finished our job and you can now load your new application. The next screenshot shows the first state where the fields are blank:

7. If you click on the **Get data!** button, a new request will be granted and the result will be displayed in each of the fields:



How it works...

Our simple JSON file contains two variables, one of them defines the name of a user and the other defines the age. We defined only two attributes but you could add another by separating it with a comma. In fact, the JSON format defines more complicated structures such as arrays and lists.

Each approach uses different code and functionality. Let's find out how they work.

Using Sencha Touch:

The most important piece of code for this recipe is the `Ext.util.JSON` object found in the *Sencha Touch* framework. Thanks to the `decode` method, we can transform simple text in a JavaScript object with different properties. Actually, in our example we are sending a GET request to the JSON file and then the response goes to the `data` object. Finally, the `alert` method is accessing the `username` and `age` properties of the mentioned JavaScript object.

By default, Sencha Touch defines a `response` object which encapsulates the response offered by the server after a HTTP request. In this case, we only need to access the `resposeText` for getting our response. Remember that we're requesting directly one text file without any additional processing in the server side. In fact, we don't require a web server for testing with our local machine.

Using XUI:

The XUI framework offers a specific method for making HTTP requests, which returns JSON responses. Additionally, this object can replace directly the content of some HTML tags for the received values. Our recipe is carrying out this effective action. The first step is to make the request to a specific URL and then using an array to indicate which HTML elements will be replaced. Each of these nodes is referred through its `id` attribute, which is used as value in the mentioned array inside the `xhrjson` method. For example, our recipe needs to populate the fields after the user clicks on the button. This widget has a handler to invoke to the `get_data()` function, which calls directly to `xhrjson`. The `#name` value of this method references to the value of the `username` property of our JSON file. Additionally, our field is using the `name` value for its `id` attribute.

For the user interface we built two simple information fields as shown in the *Chapter 2*, *Building Interfaces*. The UIUiKit offers an effective way to do that.

There's more...

It is possible to write a server-side component for generating the JSON file. For instance, let's suppose that we have a set of data stored as records in a database. We can query the database and generate a JSON file containing this data. In fact, this technique is common across modern web applications using AJAX. Some server-side technologies such as Ruby on Rails (Ruby) and Django (Python) offer a mechanism to simplify applying this technique.

A lot of information about the JSON format can be found at `http://www.json.org`.

If you're interested in more details about the `Ext.util.JSON` object contained in Sencha Touch, you can find the complete reference at `http://dev.sencha.com/deploy/touch/docs/output/Ext.util.JSON.htmlhttp://dev.sencha.com/deploy/touch/docs/output/Ext.util.JSON.html`.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the Sencha Touch framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the XUI framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Building the information fields* recipe in *Chapter 2*, *Building Interfaces*
- *How to send HTTP requests* recipe

Sending cross-domain requests

The previous recipes in this chapter show you how to make HTTP requests and how to process responses in JSON format. But our examples assume that we're using a request from and to the same domain. At the moment, we only use localhost as our domain but it can be replaced by your public domain name. Keep in mind that it could be really useful to make requests to other domains. For example, let's suppose that we need to get some data from a domain other than the one where our application is running. Handling this situation requires a specific technique, which is different from a regular HTTP request. In order to help, the *WebApp.Net* offers us a specific object plus a PHP script. Actually, this recipe will show you how to make a cross-domain request and how to process the response in XML format.

We're going to build a small application for reading some data from an external domain and displaying the result through a standard alert JavaScript dialog box.

Getting ready

We'll be using the WebApp.Net framework, so you should make sure that this software is installed on your computer. For testing this recipe, you'll need two different machines located at different domains. Maybe you use a hosting service or you have access to an external machine at your office or your university. To keep things simple, one of the machines will be a local machine under the localhost domain. For our example, we'll be using a domain owned by the author called `bsnux.com`. You can use it for testing or you can replace the reference to it with your own.

Our XML file will represent a simple record with a username and the age of one person. In fact, we're going to use the same data that we used in the previous recipe.

The interpreter of PHP programming language must be installed and running in your local domain.

How to do it...

1. The first step for this recipe is to create a new file called `data.xml` in the machine under a different domain that is localhost. In our case, this file resides under a directory called `static` in the `bsnux.com` domain. The mentioned XML file will contain the following lines:

```
<?xml version="1.0" ?>
<person>
  <username>Michael</username>
  <age>35</age>
</person>
```

2. Let's come back to our local machine for creating a new directory under our `DocumentRoot` folder of the web server. This new directory will be called `cross-webapp`. Open your favorite text editor, copy the following lines, and save it as `index.html`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html;
      charset=utf-8" />
    <meta name="apple-mobile-web-app-capable" content="yes" />
    <meta name="viewport" content="width=device-width; initial-
      scale=1.0; maximum-scale=1.0; user-scalable=0;" />
    <title>Cross-domain</title>
    <link rel="stylesheet"
      href="../../WebAppNet/Design/Render.css" />
    <script type="text/javascript"
      src="../../WebAppNet/Action/Logic.js"></script>
    <script type="text/javascript" src="main.js"></script>
  </head>
  <body>
    <div id="WebApp">
      <div id="iHeader">
        <span id="waHeadTitle">Cross-domain</span>
      </div>
      <div id="iGroup">
        <div class="iLayer" id="waHome" title="Home">
          <div class="iBlock">
            <h1>Cross-domain request</h1>
```

```
            <input type="button" class="iPush iBClassic"
              value="Get data!" onclick="get_cross()" />
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

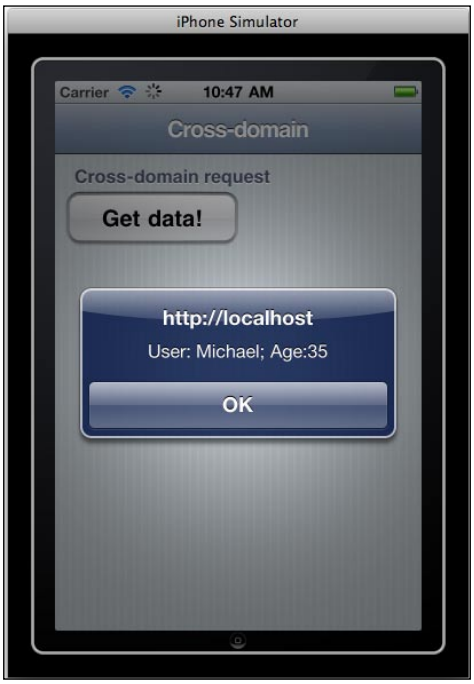
3. After saving the HTML file, create a new file called `main.js` inside our `cross-webapp` folder and add the next lines of JavaScript code:

```
WA.Proxy("../../WebAppNet/Proxies/ProxyXML.php");

function get_cross(o) {
  if (!o) {
    WA.Request("http://www.bsnux.com/static/data.xml",
      null, get_cross, true);
    return;
  }
  var data = o.responseXML;
  var username =
    data.getElementsByTagName("username").item(0).firstChild.data;
  var age =
    data.getElementsByTagName("age").item(0).firstChild.data;
  alert("User: " + username + "; Age:" + age);
}
```

4. At this point make sure that you have a file called `ProxyXML.php` inside your `WebAppNet` folder, where the WebApp.Net framework is installed. If you don't have this file, just copy it from the original folder created after uncompressing the ZIP file of the framework.

5. When you're ready for testing, load the new application in your iPhone and you'll see a screen similar to the next screenshot:

6. By clicking on the **Get data!** Button, the cross-domain request will be executed and the alert box will display the final processed result:



How it works...

Instead of using a JSON response, we used a XML response to express our data. Similarly to the JSON file used in the previous recipe, the XML file for this recipe contains plain text. Web developers are familiarized with these kinds of files because both are common formats for exchanging data via AJAX.

The WebApp.Net framework provides a PHP file for making cross-domain requests without problems and restrictions. From the technical point of view, this file works as a proxy and uses the cURL library of PHP for making the request. PHP and cURL library are extremely common on servers all over the world. Probably your hosting provider is offering you this software in your remote server. Regardless, you can install it on your local machine for testing this recipe. However the technique used in this recipe has a main disadvantage: PHP is required as a backend and a server-side component.

Regarding the user interface of our small application, we applied some CSS classes to achieve a native interface for the iPhone. For example, the `iBlock` CSS class helps us to define a block for inserting some widgets. In our application, we inserted a button inside a `div` element with this CSS class. As you can observe, our button is a simple HTML anchor tag with the `onclick` handler added. When the user clicks on it, the `get_cross()` JavaScript function will be invoked. In fact, this function resides in a file called `main.js`, which contains a line of code for invoking the `Proxy` method of the `WA` predefined object of WebApp.Net. Even though we didn't pass any arguments to this function, the `res` parameter will be populated with the response from the server. In fact, this event will happen after invoking the `Request` method of the `WA` mentioned object. This method is using the remote URL for getting an XML response. All of this behavior is transparent for the user and our proxy will take care of it.

When the request is completed successfully, we'll have an object called `res` along with our response. At this point, it's time to process our response for displaying its result. The `responseXML` method helps us to handle the response as an XML format. Safari Mobile implements the required technology to work with it. In addition, the `getElementsByTagName` method provides access to the XML file and it allows us to get the value of each node. Finally, we used two different variables for reading the values of our XML nodes and the final result is displayed to the user through a standard JavaScript alert box.

There's more...

Despite the fact that we used an XML file for our recipe, it's possible to use other formats as response to JSON. For this different approach, you'll need to write your own proxy instead of using the original provided by WebApp.Net. In this case, PHP is not required and you can write your own proxy using other server-side technology.

Cross-domain requests are very useful for mashup applications because this technique allows us to get data for different domains without reloading the page and avoiding the problems of using plain HTTP requests from the JavaScript code.

If you need to install PHP in your machine, take a look at the official documentation at `http://php.net/manual/en/install.php`.

The PHP website offers complete reference and documentation for the language and for its different libraries. cURL is not an exception and you'll find all the related information of this library at `http://php.net/manual/en/book.curl.php`.

See also

- *Installing the WebApp.Net framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Processing JSON responses* recipe

7
Working with Data: Storage and SQL

In this chapter, we will cover:

- Creating a database
- Creating a table
- Inserting records
- Searching and selecting records
- Deleting records
- Saving and reading preferences
- Storing data in session

Introduction

Usually, storage is an important issue for modern applications. Many of them require saving and reading data in a persistent storage. It means that the data will remain after closing the application or turning off the device. Traditional web applications use a remote database for carrying out these actions. Despite the fact that we can follow this approach in our web applications for the iPhone, we have a clear alternative: the use of local storage capabilities of *Safari Mobile*. Actually, this functionality was defined in the technical specifications of the HTML5 standard, which the web browser of iPhone implements, so we are free to take advantage of it.

Apart from the mentioned local databases, *Safari Mobile* offers us another mechanism for storing and handling data. In fact, we can talk about two different kinds of it. The first kind of storage is called `localStorage` and the second is `sessionStorage`. Main differences between them are the persistence and the life cycle of data. **localStorage** will keep data in a safe state after closing our application or turning off our iOS device. However, data stored using **sessionStorage** will be deleted after performing one of these two operations. Obviously, developers should decide which data would be stored using each storage system.

For simple storage information, it's very recommendable to use `localStorage` or `sessionStorage`. Both of them work with the property equal to value schema. On the other hand, more complex data should use local databases.

Readers interested in working with local databases should be familiarized with the SQL language. If you aren't yet, it could be recommendable to take a look at some documentation about fundamentals of databases and SQL. It's pretty easy to find tons of documents about this on the Internet.

Despite, we don't need any specific framework for the recipes of this chapter. We'll work with the *UiUIKit* for building simple user interfaces. Of course, you can use the framework of your choice for building the GUI of your own applications working with local databases and/or `sessionStorage` and `localStorage`. The main reason for using the mentioned framework for the following recipes is the simplicity. Remember, utilizing these storage techniques doesn't require any specific framework or external component.

This chapter teaches you how to work with the internal capabilities of *Safari Mobile* and how to handle data in your iPhone's applications. You'll learn:

- How to create and open local databases
- How to deal with records
- How to store and access data using `localStorage` and `sessionStorage`

Let's get started by creating a new local database.

Creating a database

The goal of this recipe will be to create a local database for our data. Inside the new database, we can create tables and execute different operations for inserting, reading, saving, and deleting records. Fortunately, we don't need to learn any new language because it's possible to use regular SQL for the mentioned operations.

Instead of using a file or other different resources for storing our data, we'll work with the database provided by the browser. This has two immediate consequences; one of them is avoiding the use of a remote database in the server side. The other one is the flexibility of using a local database engine embedded inside the browser. *Safari Mobile* will do the hard job for us; developers only need to take care about executing the right SQL statements. Internal mechanisms are transparent for uses and developers.

For the sake of simplicity, we're going to create a simple application for creating a new database when the user clicks on a button. The complete `HTML` file for this recipe can be reached at `code/ch07/create_db.html` in the code bundle provided on the Packtpub site.

Getting ready

In this recipe we're going to use the *UiUIKit* for defining the look and feel of our application. This is not a formal requirement; you can use another framework for this definition. We decided to use this framework just for simplicity, so don't forget to check if the *UiUIKit* is installed and working on your computer before continuing.

How to do it...

1. As you've learned in the previous recipes of this book, we're going to start using a standard XHTML header plus the statement needed for loading and activating our chosen framework. The required lines of code for loading the *UiUIKit* are the following:

```
<link rel="stylesheet"
href="../uiuikit/stylesheets/iphone.css" />
```

2. The most important part of this recipe is the following JavaScript function, which has the responsibility of creating our database:

```
<script type="text/javascript" charset="utf-8">
  function create_db() {
    var db_kb_size = 256;
    var db_size = 1024 * db_kb_size;
    var db = openDatabase('firstDB', '1.0', 'firstDB',
    db_kb_size);
    if(db) {
      alert("Database created!");
    } else {
      alert("Error!");
    }
  }
</script>
```

3. Finally, we'll need to define our user interface by placing the following HTML elements inside of our body tags:

```
<div id="header">
  <h1>Database</h1>
</div>

<ul class="data">
  <li>
```

```
    <p>How to create a new database</p>
  </li>
</ul>

<p>
<a href="#" onclick="create_db()" class="button white">Create
it!</a>
</p>
```

4. Before testing our new application, you should save the new file. After loading the new HTML file on your iPhone or on the iPhone Simulator application, you'll see a result similar to the next screenshot:

How it works...

For this recipe we defined one JavaScript function, which creates a new local database called `firstDB()`. The main function to do that is `openDatabase()` and it is provided by *Safari Mobile*, so we don't need any specific framework or library. Different parameters are required by this function. Specifically, we must use four parameters related to our new database. The first parameter will be the name, the second one will be a simple string for identifying the version, the third is a description, and the last parameter is the estimated space available for the database. In our recipe, we use the same value for description and for the name of the database. However, the description could be a long textarea with more information about the

database and its goal is to inform the user about the purpose of our new database. Regarding the size, it is expressed in bytes so we used the `db_kb_size` and `db_size` variables to set it. In this case, the browser will reserve 256 KB for our database. You don't need to worry about it; if you need it, you can change this size later.

It's very important to keep in mind that the `openDatabase()` function returns a connection handler to the database identified by the given name. If this database doesn't exist, it will be created.

Our JavaScript function called `create_db()` will be called when the user clicks on the **Create it!** button. As you can observe, we defined an `onclick` handler for our button represented by the specific HTML anchor element. Finally, our method is using a simple and effective Javascript *alert* function to send a message to the user informing him/her about the success or error of the operation.

There's more...

Safari Mobile offers us a mechanism for checking how many databases were created. Also, we can see how much disk space was defined and the current size for each one. This information is available though the Settings application of iPhone. To see it, you can click on the **Settings** | **Safari** | **Databases** menu option.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Creating a table

This recipe shows you how to create a simple table in a local database using a standard `CREATE TABLE` sentence of SQL language. Our table will have four different fields representing the data (first name, last name, e-mail, and age) of one student. This table will be called `alumni`. All of the fields will be declared as `VARCHAR`, except the age that will be an `INTEGER`.

We're going to display a button for creating a table, and if everything is fine, we'll launch an alert box with a message informing that the table was created. Otherwise, our box will display an error message.

All the code required for this recipe can be found at `code/ch07/create_table.html` in the code bundle provided on the Packtpub site.

Getting ready

As in the previous recipe of this chapter you'll need to get the *UiUIKit* framework installed and ready. Instead of writing code from scratch, we'll reuse the code used in the previous recipe.

How to do it...

1. Save the HTML file created in the previous recipe as `create_table.html`, and then open it with your favorite text editor. After the `create_db()` JavaScript code, you should add the following new functions:

```
function onError(tx_table,error) {
  alert("Error: " + error.message + "; Code: " + error.code);
  return true;
}

function onSuccess(tx_table, res) {
  alert("Table 'alumni' created!");
  return true;
}

function create_table() {
  db = create_db();
  if (db) {
    db.transaction(
      function(tx_table) {
        tx_table.executeSql(
          'CREATE TABLE IF NOT EXISTS alumni ' +
          ' (id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT, ' +
          ' first_name VARCHAR(255) NOT NULL, ' +
          ' last_name VARCHAR(255) NOT NULL, ' +
          ' email VARCHAR(255) NOT NULL, ' +
          ' age INTEGER NULL)',
          [],
          onSuccess,
          onError
        );
      }
    );
  }
}
```

2. In order to be consistent, the next step will be to change the text of some components of the user interface. For example, replace the content of the `<h1>` HTML tag with a new string:

```
<h1>New table</h1>
```

3. Our button requires to call to the new `create_table()` JavaScript function:

```
<a href="#" onclick="create_table()"
class="button white">Create it!</a>
```

4. Save changes to your new file and load it on your device. Then click on the button. An alert will be launched as shown in the next screenshot:

How it works...

The main JavaScript function of this recipe is `create_table()`. It calls our previously defined `create_db()` function and then uses the `transaction` method of the returned database connection. Remember that the database will only be created the first time, otherwise it will be opened. In any case, a reference to it will be returned by our `create_db()` function.

The predefined `transaction` method will deal with our database using a specific function passed as parameter. This function is anonymous and it will invoke the predefined `executeSql` method, which requires four different parameters. The first one is the SQL statement, in this case, it is our `CREATETABLE` passed as a string. The second parameter is an empty string, which will be used for other kind of SQL sentences, as we'll learn in next recipes of this chapter. Finally, the last parameters are the `callback` functions. The `onSuccess()` callback will be called if everything works properly and the `onError()` callback will be called if an error is found. `onSucess()` will expect an object with the result of the performed action. On the other hand, `onError()` will receive an error object with properties such as message and code.

You probably observed the name of the JavaScript database method we named as transaction. This means we're executing a database transaction as an atomic operation.

The `IF NOT EXISTS` condition prevents to execute the `CREATE TABLE` statement more than once. Also, we apply the `AUTOINCREMENT` property to our `PRIMARYKEY` to indicate that we'll use a different and unique ID for each record. This behavior will be transparent for users and developers.

Regarding the user interface, we used the main HTML as in the previous recipe. The `click` event is controlled using the `onclick` specific handler of the `anchor` element.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating a database* recipe

Inserting records

We've learned how to create a local database and a table associated to it. The next logical step will be to insert records in our recent table. For our new goal, we're going to use a simple form where a user can type the values for our fields. Traditional web applications are using the same method. Obviously, our form will contain a button for doing the action.

You can insert records as you desire, but remember the database has size restrictions. We set this value through a parameter passed in the function, which opens or creates our database. The form introduced in this recipe allows you to save many records and the application will display a simple alert box after each insertion.

The code for the application of this recipe contains only one HTML file located at `code/ch07/insert_rec.html` in the code bundle provided on the Packtpub site.

Getting ready

Be sure the *UiUIKit* is working on your computer. For making our life easier, we're going to use the same file created for the previous recipe. This current recipe doesn't create the required table in the database, so you need to create it as you've seen before.

How to do it...

1. Save your previous file `create_table.html` as `insert_rec.html` in a folder inside the *DocumentRoot* of your favorite web server. Open the file just created with your text editor and delete the `create_table()` JavaScript function. Instead of it, you should add the following new function:

```
function insert_record() {
  var first_name = document.frm.first_name.value;
  var last_name = document.frm.last_name.value;
  var email = document.frm.email.value;
  var age = document.frm.age.value;
  db.transaction(
    function(tx) {
      tx.executeSql(
        'INSERT INTO alumni (first_name, last_name, email, age)'+
        'VALUES (?, ?, ?, ?);',
        [first_name, last_name, email, age],
        function(tx, res) {
          alert("New record added!");
        },
      onError
      );
    }
  );
}
```

2. At this point, we need to do some small changes. Replace the `tx_table` parameter of the `onError()` function for `tx` and change the content of the `<h1>` tag for inserting record. Look for the `<body>` element and insert a new handler:

```
<body onload="create_db();">
```

3. Now, replace the `<ul>` and the `<p>` HTML element for the following form:

```
<form name="frm">
  <ul class="form">
    <li>
      <input type="text" name="first_name"
      placeholder="First name" id="first_name"
      value=""/>
    </li>
    <li>
      <input type="text" name="last_name"
      placeholder="Last name" id="last_name" value=""/>
    </li>
    <li>
      <input type="text" name="email"
      placeholder="e-mail" id="email" value=""/>
    </li>
    <li>
      <input type="text" name="age"
      placeholder="age" id="age" value=""/>
```

```
    </li>
  </ul>

  <p>
    <a href="#" onclick="insert_record()"
    class="button white">Insert</a>
  </p>
</form>
```

4. Well, you're ready for testing our new application for inserting records. The next screenshot shows you how the application will display our new form:

How it works...

We're supposing our database and table are created. You only need to call the `create_db()` function. In this case, the function is called only one time when the application is loaded by the browser. For this reason, we used the `onload` event handler.

The function managing insertion is called `insert_records()` and it picks up values of each field directly using the `document.frm` accessor and the `value` property. This main function is using the `transaction` method of the db object in the same way the `create_table()` function used in the previous recipe. As you can observe, we used an array as the second parameter in the `executeSql` method. The mentioned array will contain the values collected for each field of the form. Regarding the SQL sentence, it lists parameters using the `?` character. Each one of these will be replaced for each value of our array. This technique keeps our application safe from a dangerous and malicious SQL injection exploit.

If each insertion works properly, the application will display a message using a standard alert box. Otherwise, the error value and code will be shown to the user.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating a database* recipe
- *Creating a table* recipe

Searching and selecting records

Retrieving records from our database is very important. Actually, we've been storing values because we need to read them back later.

To illustrate the process and find out how it works, we'll build a simple application for searching records by its `first name` field. Our goal will be to provide a text input where a user can type some text for finding students.

After clicking on one button, we'll execute a simple SQL query looking for the student whose first name contains the value typed by the user. The results will be displayed using an unordered list. But, what happens if an error occurs? In this case, we'll launch an alert box reporting the error.

To take a look at the file for this recipe, you can see `code/ch07/select_rec.html` in the code bundle provided on the Packtpub site.

Getting ready

As a part of the *UiUIKit* for defining the graphic interface of our application, we're going to use the *XUI* framework for this recipe. Before continuing, check if both frameworks are installed and ready to use. Just for simplicity, we'll use some of the code from the previous recipe. Also, we'll suppose our database and the alumni table are created. It's important to populate the mentioned table with some data using the application developed for the previous recipe.

How to do it...

1. Copy the `insert_rec.html` file to `select_rec.html`. Of course, you can use the same folder for both files. Before the `<title>` tag, you should insert the following line for loading the *XUI* framework:

```
<script type="text/javascript"
src="../xui-2.0.0.min.js"></script>
```

2. Inside the JavaScript section of our HTML file, we'll need to add the next JavaScript function:

```
function onSuccess(tx, res) {
  var html_res = "";
  if (res.rows.length > 0) {
    for (var i=0; i < res.rows.length; i++) {
      var row = res.rows.item(i);
      html_res += '<li><a href="">' + row.first_name + '</a></
li>';
      }
    } else {
      html_res = "<li>No records were found</li>";
    }
  x$('#rec_list').html('inner', html_res);
}
```

3. After inserting the `onSuccess()` function, you should add this new function for executing the required `SELECT` statement:

```
function select_records() {
  var first_name = document.frm.first_name.value;
  db.transaction(
    function(tx) {
      tx.executeSql(
        'SELECT * FROM alumni WHERE first_name LIKE ?;',
        ["%" + first_name + "%"],
        onSuccess,
        onError
      );
    }
  );
}
```

4. User needs to type some text for searching, so we're going to use a simple form:

```
<form name="frm">
  <ul class="form">
    <li>
      <input type="text" name="first_name" placeholder="First
name"
        id="first_name" value="" />
    </li>
  </ul>
</form>
```

5. Finally, we need a button and an unordered list to display records found:

```
<p id="p_btn">
  <a href="#" onclick="select_records()"
  class="button white">Search</a>
</p>

<ul id='rec_list'></ul>
```

6. Testing time. Load your new application in your device. The initial screen is shown in the following screenshot:

7. Introduce some text and click on the **Search** button for getting a list with records extracted from our database:

How it works...

You may already know that `executeSql` is the most important method for working with our local database. This method allows you to run various SQL statements. In this case, we chose a simple `SELECT` query using the `LIKE` operator. This query helps us find students whose first name contains the text introduced by the user.

If there are no errors, the `onSuccess()` callback function will be executed. Instead of displaying a message, we're going to read all returned values creating a new `<li>` element for each one. By default, the unordered list is empty and each new element will be added using the `html` method provided by the *XUI* framework. Actually, we're handling the DOM of our web page thanks to the functionalities included in this framework. The `x$('#rec_list)` acts as a selector of the unordered list whose ID is equal to `rec_list`. The `html` method allows us to use two different parameters: one for content to be inserted and another indicating where. In our example, `html_res` is a simple `<li>` element containing the value for each record and `inner` indicates that the element requires to be inserted inside the unordered list.

The `res` parameter received by `onSucess` contains an array called `rows`, which contains all information about our retrieved records. The `item` method provides access to each one of these records and it requires a number working as the index for the mentioned array. Each of the objects retrieved for `item` will have one property for each field of the record in the table. In fact, `row.first_name` returns the `first_name` field of the `alumni` table of the `firstDB` database.

If our query doesn't get any result, the `onSucess()` function will return a `<li>` element with a message reporting this situation. On the other hand, keep in mind that if the user clicks on the **Search** button without typing any text, we'll get all records.

Apart from the main form, we're using a simple unordered list that UiUIKit transforms into a list with a consistent look and feel for the iPhone. You can use another framework for building these widgets if your prefer it. It's even possible to use other widgets for displaying results.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the XUI framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Building the lists for items* recipe in *Chapter 2*, *Building Interfaces*
- *Creating a database* recipe
- *Creating a table* recipe
- *Inserting records* recipe

Deleting records

Sometimes we'll need to delete some records of our table. This is a common operation in applications using databases. For this recipe, we're going to modify the application developed in the previous recipe by adding a link for each field. This new link will be useful to invoke a new JavaScript function to do the job.

If the selected record is deleted, we'll display an alert box with a message reporting about the success of the operation. Otherwise, another alert box will launch an error message.

Getting ready

For our example application we'll need *UiUIKit* and *XUI* frameworks. Once again we're going to reuse the code specified in the previous recipes about dealing with local databases.

How to do it...

1. The first step will be to save `select_rec.html` as `deleting_rec.html`. A small change for the `onSucces()` Javascript function is required. Just replace the line where `<li>` is assigned to `html_res`:

```
html_res += '<li class="arrow"><small>Delete</small><a href="#"
onclick="delete_record(' + row.id + ')">' + row.first_name + '</
a></li>';
```

2. Our specific function for deleting records is the following one:

```
function delete_record(id_rec) {
  if (confirm("Are you to delete it?")) {
    db.transaction(
      function(tx) {
        tx.executeSql(
          'DELETE FROM alumni WHERE id=?;',
          [id_rec],
          onSuccessDel,
          onError
        );
      }
    );
  }
}
```

3. As you can observe, we have a new callback called `onSuccessDel`. The required code for this function is the following:

```
function onSuccessDel(tx, res) {
  alert(res.rowsAffected + " records deleted!");
}
```

4. We don't need any more changes for our new recipe. After loading the new HTML file on your device, you will see the same form from the last recipe. If you type some text and click on the **Search** button, a new list will appear with a new link called **Delete**, as shown in the next screenshot:

5. When one of the links is clicked, a dialog box appears asking for confirmation from the user for proceeding to delete the current record. After executing this operation, a new alert box will appear reporting about the success of deleting.

How it works...

Deleting records is pretty easy; we only need to execute a `DELETE` query using standard SQL through the `executeSql` method. This recipe shows you how to delete a record using its own ID. Remember that this field is unique for each record of our table and it's assigned automatically.

For preventing accidental deletion of records, we use a simple confirm dialog box requesting permission from the user.

The `onSuccesDel()` callback function receives an object, which contains a property called `rowsAffected`. This is useful for finding out how many records were deleted. In our recipe, only one record will be deleted at a time. But if you use a different SQL statement, the `rowsAffected` value will not be equal to one.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the XUI framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Building the lists for items* recipe in *Chapter 2*, *Building Interfaces*
- *Creating a database* recipe
- *Creating a table* recipe
- *Inserting records* recipe
- *Searching and selecting records* recipe

Saving and reading preferences

This recipe will teach you how to save and read values from the `localStorage` system. We're going to build a simple application for saving and reading a selected language as an internal preference. Actually, we'll ask the user to select one value for language and it will be stored in a property called `lang`. This property could be useful for deciding which other values should be used. In fact, the mentioned property helps us to choose and set another property in `sessionStorage` as you can learn in the next recipe of this chapter.

Obviously, you can store different properties in `localStorage`. This storage system can work as a central place for storing user's preferences avoiding the use of a local SQL-based database.

Getting ready

Take a look at your server to check if *iWebKit* is installed and ready to use.

How to do it...

1. We're going to start our recipe creating a new XHTML file called `preferences.html` inside a folder under the control of *DocumentRoot* of your web server. Our first lines of code will be a standard XHTML header followed by the required lines for loading the *iWebKit* framework:

```
<link href="../iwebkit/css/style.css" rel="stylesheet"
media="screen" type="text/css" />
  <script src="../iwebkit/javascript/functions.js"
    type="text/javascript"></script>
```

2. The second step is to add our new JavasScript functions:

```
<script type="text/javascript" charset="utf-8">
  function get_lang() {
    if (localStorage.lang != undefined) {
      document.frm.sel_lang.value = localStorage.lang;
    }
  }

  function save_pref() {
    localStorage.lang = document.frm.sel_lang.value;
    alert("Preference was saved!");
  }
</script>
```

3. After adding the JavaScript code, we'll modify our `<body>` element by adding an event handler:

```
<body onload="get_lang()">""
```

4. For the next step, you need to build our user interface adding the following `<div>` elements including our main HTML form and one simple button:

```
<div id="topbar">
  <div id="title">Preferences</div>
</div>

<div id="content">
  <form name="frm">
    <fieldset>
      <span class="graytitle">Language</span>
      <ul class="pageitem">
        <li class="select">
          <select name="sel_lang">
            <option value="English">English</option>
            <option value="German">German</option>
            <option value="Spanish">Spanish</option>
          </select>
        </li>
      </ul>
      <ul class="pageitem">
        <li class="button">
          <input type="button" value="Save"
          onclick="save_pref()"/>
        </li>
      </ul>
    </fieldset>
  </form>
</div>
```

5. The last step will be to close the `<body>` and `<html>` tags and save all the changes to our new file. Then, you can load it on your iPhone and the expected result is showed in the next screenshot:

How it works...

After the browser is ready and it loads our new application, it will call the `get_lang()` JavaScript function. This function gets the `lang` property from the `localStorage` system and displays its value in the `<select>` element. If the property is not defined then it doesn't do anything and the `<select>` element will display the first option (**English**).

When the user clicks on the **Save** button, the event *click* for the button is launched and the `save_pref()` JavaScript function is executed. This function gets the value from the `<select>` element and sets the `lang` property of `localStorage`. After taking this action, we'll launch a simple alert box reporting that our value has been set.

Remember that the value for our `lang` preference will remain after closing your browser or deleting the HTPP session.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Building forms with checkboxes, radio buttons, select fields, and text fields* recipe in *Chapter 2*, *Building Interfaces*
- *Storing data in session* recipe

Storing data in session

Thanks to this recipe, you'll know how to use `sessionStorage` to store properties with values. It's very important to keep in mind the lifecycle of these properties. If the user closes the browser or cleans the HTTP session, our values stored in `sessionStorage` will be deleted.

The application for this recipe shows you how to store and read one property in `sessionStorage`. We are going to start by setting and displaying a property called `today`, which stores the current date using a format based on the value of another property set previously in `localStorage`. As you've seen in the previous recipe, `lang` is a property storing the language selected by the user. In this recipe, our application will read this property from `localStorage` and will set the suitable format for the `today` property based on it. For example, if the current date is June, 13th 2011 and the selected language is **English**, the value for `today` will be **6/13/2011**. Otherwise, the result will be **13/6/2011**.

Getting ready

Two frameworks are required for this recipe. One of this is *XUI* and another is *iWebKit*. Both frameworks help us build a simple user interface for our new application. Please, bear in mind that from a strict point of view, we don't need any framework for handling `localStorage` and `sessionStorage`.

How to do it...

1. The first step will be to create a new XHTML file with a simple header including the required lines for using *iWebKit* and *XUI*:

```
<link href="../iwebkit/css/style.css" rel="stylesheet"
media="screen" type="text/css" />
<script src="../iwebkit/javascript/functions.js"></script>
<script src="../xui-2.0.0.min.js"></script>
```

2. For the next step, we're going to add our JavaScript code inside the `<head>` section:

```
<script type="text/javascript" charset="utf-8">
  function get_today() {

    if (sessionStorage.today == undefined) {
      var toD = new Date();
      var day = toD.getDate();
      var year = toD.getFullYear();
      var month = toD.getMonth() + 1;

      if (localStorage.lang != undefined)
        if (localStorage.lang == 'English') {
          sessionStorage.today = month + "/" + day + "/" + year;
        } else {
          sessionStorage.today = day + "/" + month + "/" + year;
        }
      else {
        sessionStorage.today = day + "/" + month + "/" + year;
      }
    }
    x$("#today").html("inner", "Today is "+sessionStorage.today);
  }
</script>
```

3. Finally, you should add the HTML code required for defining our user interface:

```
<body onload="get_today()">
  <div id="topbar">
    <div id="title">Today</div>
  </div>
  <div id="content">
    <span class="graytitle" id="today"></span>
  </div>
</body>
```

4. Save your new file as `session_date.html` and load it on your iPhone. Your screen will display a message with the current date based on the format chosen previously and store it in `localStorage`:

How it works...

The small application developed for this recipe displays the current date after it is loaded. This happens thanks to the `onload` handler, which calls the `get_today()` JavaScript function. As `sessionStorage` uses a simple storage system based on property equals value, we can assign a value directly to a property using the syntax `sessionStorage.property = value`. In fact, our JavaScript function uses this mechanism in the `sessionStorage.today = month + "/" + day + "/" + year` code line.

It's easy to access the properties stored in `sessionStorage`. You only need to use the syntax `sessionStorage.propety` as you have seen in the `x$("#today").html("inner", "Today is " + sessionStorage.today)` line of our `get_today()` function.

On the other hand, for displaying the current date on the screen, we used the `html` method provided by *XUI* for inserting text inside the selected `<span>` element. The inner value for this method indicates where the text should be added.

The current date is calculated using the JavaScript `Date` class, which provides different methods for getting the current day, year, and month. If you create an object of this class and don't pass any parameters for its constructor, it will use the current date.

The format for our date is chosen based on the selected language stored previously in a property under the control of `localStorage`. Our `get_today()` method only sets the `today` property once because it first checks if the property is defined. If the user loads the application and the property is set, the current day will be displayed reading its value directly from `sessionStorage`. But remember, if you close your browser or delete your HTTP session, the value will disappear and the function will need to set the property again. What happens if the `localStorage` is not defined? Simple answer: Our function will assume that the default language is not **English** and it will assign the right value to `sessionStorage.today` based on this assumption.

Regarding the user interface, we're using the *iWebKit* framework instead of *UiUIKit*. There isn't any special reason for it. You can use another framework for building your graphic user interface. In this case, *iWebKit* helps us to build a simple top bar with a title and specific area for displaying the current date.

There's more...

Instead of using the syntax `sessionStorage.property = value`, it's possible to use an alternative method. It consists of using the `setItem()` method of `sessionStorage`. For example, you could write `sessionStorage.setItem(today, month + "/" + day + "/" + year)` instead of `sessionStorage.today = month + "/" + day + "/" + year`. Respectively, the `getItem()` method will read our value. Therefore, `var current_date = localStorage.getItem('today')` will assign our value to the new `current_date` JavaScript variable.

Apart from `getItem()` and `setItem()`, `sessionStorage` and `localStorage` provide another method called `removeItem()`. Yes, you guessed it. This method is used for deleting the property for the storage system. You can even call another method called `clear()` for deleting all properties and values stored in `localStorage` and `sessionStorage`.

Don't forget that `getItem()`, `setItem()`, `removeItem()`, and `clear()` can be applied to `localStorage` and `sessionStorage`.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks make life easier*
- *Installing the XUI framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Saving and reading preferences* recipe

8
This is a Phone

In this chapter, we will cover:

- Calling a number
- Sending an SMS to a number
- Selecting contacts
- Creating a new contact
- Searching and displaying contacts

Introduction

Although iPhone is a smartphone, we shouldn't forget its main functionality of making phone calls and sending text messages in addition to its other rich features. This chapter will teach you how to use the contacts stored in the internal address book of the device.

At the time of publication, working with contacts requires us to develop native applications. But we can use JavaScript with the *PhoneGap* framework for building these kinds of applications for the iPhone. This is the main reason for applying this framework for the recipes included in the current chapter.

This chapter focuses on issues related to calls, SMS, and contacts. We'll learn how to handle contacts and how to send an SMS or place a phone call simply by interacting with the user interface.

Calling a number

In this recipe, you'll learn how to call a number when a user clicks on a button in the user interface. Specifically, we're going to build a simple list containing our contacts where each represents a person and their phone number. After clicking on one of these elements the dial screen will be opened and the user will only need to click on the green button for calling the specified number.

We only need to build a simple XHTML file for this recipe. It can be found at `code/ch08/call.html` in the code bundle provided on the Packtpub site.

Code files can be downloaded at the Packt website.

Getting ready

This recipe requires the use of the *iWebKit* framework.

How to do it...

Open your favorite text editor or IDE and create a new file called `call.html`.

1. Add the standard XHTML headers and the required lines for invoking the JavaScript and CSS files provided by *iWebKit*:

```
<link href="../iwebkit/css/style.css" rel="stylesheet
  media="screen" type="text/css" />
<script src="../iwebkit/javascript/functions.js"
  type="text/javascript"></script>
```

2. Add the main HTML code for defining our user interface:

```
<body class="list">

<div id="topbar">
  <div id="title">Call</div>
</div>

<div id="content">
  <ul>
    <li class="title">Contacts</li>
    <li><a class="noeffect" href="tel:555-666-777">
      <span class="name">Aaron Stone</span></a></li>
    <li><a class="noeffect" href="tel:555-888-999">
      <span class="name">Ben Jackson</span></a></li>
```

```
    <li><a class="noeffect" href="tel:555-222-333">
      <span class="name">Bob McKenzie</span></a></li>
    <li><a class="noeffect" href="tel:555-444-666">
      <span class="name">Luke Johnson</span></a></li>
    <li><a class="noeffect" href="tel:555-333-666">
      <span class="name">Michael Sterling</span></a></li>
  </ul>
</div>
```

3. Now, save your new file and load it on your device. The following screenshot shows you the result of loading the file built for this recipe:

How it works...

The most important part of the code in this recipe is the *anchor* element inside each `<li>` tag. The `href` attribute of the anchor element is using a string, which represents a specific protocol followed by a number. In fact, `tel` identifies the protocol and the iPhone, understanding its meaning displays the dial screen allowing the user to call a number. On the other hand, the class `noeffect` was applied to each anchor for opening the dial screen in *fullscreen* mode.

From a strict point of view, *iWebkit* is not a must for calling numbers. *Safari Mobile* indentifies the `tel` protocol by default. However, we used the mentioned framework because it helps us to easily construct a list with many items.

What happens if you are using an iOS device other than the iPhone?

It is not possible to call numbers from an iPad or iPod touch as of now. The code developed for this recipe works differently in these devices. Instead of calling numbers, iPad and iPod touch will ask you if you want to save numbers in the address book.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Sending an SMS to a number* recipe

Sending an SMS to a number

The previous recipe explained how you can call a number from your applications. This recipe will show you how to send an SMS to a selected number. For simplicity we're going to use the same approach. We'll develop a simple XHTML file displaying a list with different contacts. When the user clicks on one of the contacts the iPhone will display the screen for sending an SMS.

Getting ready

As in the previous recipe we will use the *iWebKit* framework. As both recipes are similar, we're going to use the same XHTML file developed for the previous recipe.

How to do it...

1. Open the `call.html` file and replace the content of the `href` attribute of each `<li>` item for a new string starting with `sms:` instead of `tel`. For example, the first item will be the following:

```
<li>
  <a class="noeffect" href="sms:555-666-777">
    <span class="name">Aaron Stone</span>
  </a>
</li>
```

2. Save the new content of the original file as a new file called `sms.html`, then you'll be ready for testing it on your iPhone.

How it works...

As you've learned in the previous recipe, *Safari Mobile* identifies the `tel` protocol for calling numbers. In the same way, `sms` is used for sending SMS's. Also for this recipe the *iWebKit* is used for building the user interface.

See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Calling a number* recipe

Selecting contacts

Thanks to this recipe we'll learn how to select a contact from the address book of the iPhone and to display its related information. Our goal is quite simple; when the user clicks on a button, a new screen will display all available contacts. After clicking on one of these elements, the complete name of the selected contact will be displayed in the main screen.

Although this recipe is very simple, you can apply it for building complex applications that require dealing with contacts available through the address book of the iPhone.

The complete code for this recipe can be reached at `code/ch08/queryAddress` in the code bundle provided on the Packtpub site.

Getting ready

For this recipe we'll use three different frameworks: *PhoneGap*, *XUI*, and *UiUIKit*. Before continuing, make sure you have these frameworks installed on your machine. Also, remember you'll need a Mac OS X computer with the *iOS SDK* and *Xcode* installed.

How to do it...

1. The first step is to create a new *PhoneGap* project through *Xcode*. If you don't know how to do it, you can follow steps explained in previous chapters.
2. After creating your new project, the next step will be to open the main `index.html` file for adding our own code for this recipe. First, you should add the following lines inside the `head` section of the mentioned HTML file:

```
<script type="text/javascript" src="xui-2.0.0..min.js"></script>
<link rel="stylesheet" href="uiuikit/stylesheets/iphone.css" />
<style type="text/css">
  #mybtn {
    margin-right: 12px;
  }
</style>
```

3. The next step is to comment out the JavaScript function `preventBehavior()` and the line below this function. Then, we're going to add our JavaScript code:

```
function searchContact() {
  navigator.contacts.chooseContact(onSucessContact);
}

function onSucessContact(contact) {
  var selectedContact = contact.name;
  x$('#ul_contacts').html( 'inner', "<li>" + selectedContact +
    "</li>");
}
```

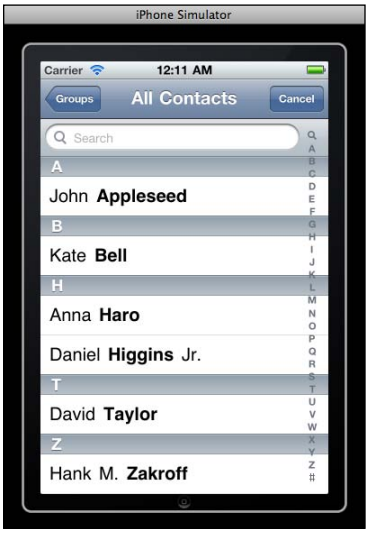
4. Finally, we'll modify the original `body` section adding the following lines of HTML code:

```
<div id="header">
  <h1>Contacts</h1>
</div>

<h1>Working with contacts</h1>

<p id="p_btn">
  <a href="#" id="mybtn" onclick="searchContact()"
    class="button white">Select</a>
</p>

<ul id='ul_contacts'></ul>
```

Before continuing and after applying all changes inside the code, we need to copy some files to the `www` directory of our project. Specifically, we're going to copy the `uiuikit` directory and the `xui-2.0.0.min.js` file. Remember, these components belong to the *UiUIKit* and *XUI* framework respectively.

Save your project and click on the **Build and Run** button of *Xcode* for testing your new application. After loading the application in the iPhone Simulator, you can see a screen similar to next screenshot:

When the user clicks on the button **Select**, a list with all the available contacts will be displayed as shown in the following screenshot:



After selecting one of the items showed in the list, our application will display the data for the selected contact:

How it works...

PhoneGap provides a JavaScript function for accessing the predefined screen of the iPhone for selecting contacts. The name of this function is `ChooseContact()` and it is invoked through the `navigator.contacts` predefined object. It encapsulates the access to the address book of the iPhone. As a parameter, this function requires a callback method that will be invoked when a contact is selected. In our case, we implemented a function called `onSuccessContact()`, which captures the complete name of the contact displaying the result in a simple list. `onSuccesContact` uses a parameter called `contact`, which represents an object containing the information related to the selected contact from the address book.

The *XUI* framework allows us to add new items to our `<ul>` list by manipulating the DOM of the HTML page. On the other hand, the *UiUIKit* framework was used for building the user interface for our small application.

We used a bit of CSS for centering our button. Actually, we modified the right margin of this widget. The CSS style for this appears before the JavaScript code, also inside the head section.

See also

- *Installing the PhoneGap framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Making a beep alert* recipe in *Chapter 5*, *Mastering Sound and Music*

Creating a new contact

The goal for this recipe will be to add a new contact inside the address book of the iPhone. We'll use a simple form with the basic data for a new contact: first name, last name, and phone number. The user can type this data inside the mentioned form. By clicking on a button the contact will be added. A simple alert box will notify this action to the user.

All code required for this recipe can be found at `code/ch08/newContact` in the code bundle provided on the Packtpub site.

Getting ready

Make sure you've installed *PhoneGap* and *UiUIKit* on your Mac computer.

How to do it...

1. Our first task will be to create a new *PhoneGap* project using *Xcode* and the *PhoneGap-based* template. Edit the `index.html` file stored in the `www` folder. Before changing the code of this file, we need to copy the folder `uiuikit` from the *UIUIKit* framework inside the mentioned `www` folder of our new project.
2. Comment out the JavaScript lines of the function called `preventBehavior()`. Then, add the following lines inside the `head` section and before the JavaScript code:

```
<link rel="stylesheet" href="uiuikit/stylesheets/iphone.css" />
<style type="text/css">
  #mybtn {
    margin-right: 12px;
  }
</style>
```

3. Now, it's time to add our own code inside the JavaScript section of `index.html`:

```
function saveContact() {
  var dataContact = {'firstName':
                       document.frm.first_name.value,
                     'lastName':
                       document.frm.last_name.value,
                     'phoneNumber':
                       document.frm.phone.value};
  navigator.contacts.newContact(dataContact, onSucessContact);
}

function onSucessContact(contact) {
  navigator.notification.alert('Contact created',
    'Contacts', "Done");
}
```

4. Finally, we'll add the required HTML lines for building the user interface for our application. The following code will be added inside the body section:

```
<div id="header">
  <h1>Contacts</h1>
</div>

<h1>New contact</h1>

<form name="frm">
  <ul class="form">
    <li>
      <input type="text" name="first_name" value=""
        placeholder="First name" id="first_name"/>
```

```
    </li>
    <li>
      <input type="text" name="last_name" value=""
        placeholder="Last name" id="last_name"/>
    </li>
    <li>
      <input type="text" name="phone" value=""
        placeholder="Phone" id="phone"/>
    </li>
  </ul>
</form>

<p id="p_btn">
  <a href="#" id="mybtn" onclick="saveContact()"
    class="button white">Save</a>
</p>
```

5. For testing our new application we click on the **Build and Run** button of *Xcode*. After loading the application, the result will be similar to the following screenshot:

6. If we type some text inside each element of the form and click on the **Save** button, the new contact will be added to our address book:

7. You can check if the new contact was added. For taking this action, we can access our address book list and look for the new contact. You will see that it exists, and by clicking on it you'll see the information just added:

How it works...

The main JavaScript function used in this recipe is `saveContact()`. This function takes two actions: getting data from the form and encapsulating the call to the `newContact()` function provided by *PhoneGap* through the `navigator.contacts` object. As you can see, this method requires a dictionary with three main keys which represent the first name, second name, and phone number for the new contact. In this case, we're picking up these values directly from the form using the attribute `value`.

`onSuccessContact()` is a function working as callback, which displays a simple alert box using the predefined `alert` method of the `navigator.notification` object. This callback function will be invoked after saving the new contact inside the address book.

For building our form we used the *UiUIKit* framework, which provides a simple and consistent look and feel using a HTML form. Our button has an `onclick` handler for invoking the `saveContact()` function, which is enabled after clicking on it.

Finally, the functionality for displaying our new contact is provided directly with iOS, so we don't need to use any additional code.

See also

- *Installing the PhoneGap framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Selecting contacts* recipe

Searching and displaying contacts

Sometimes it could be very interesting to build our own interface for searching contacts saved in the address book and for displaying the results. We'll build a simple form with a textbox where the user can type text. A button will be used for inputting the typed text and then we'll look for coincidences in our address book. The field chosen for our search will be the **First name**.

After finding contacts, we'll display a simple list with the results. When the user clicks on one of the items, the data for the selected contact will be displayed on a new page.

Getting ready

As with the other recipes in this chapter, we're going to use *PhoneGap*, *UiUIKit,* and *XUI*. Please, make sure you've installed these frameworks before continuing.

How to do it...

1. As you've learned, create a new *PhoneGap* project in *Xcode*. Open the main `index.html` file and add the required lines for loading *UiUIKit* and *XUI*:

```
<link rel="stylesheet" href="uiuikit/stylesheets/iphone.css" />
<script type="text/javascript" src="xui-2.0.0.min.js"></script>
```

2. Now, we need to add our own JavaScript code:

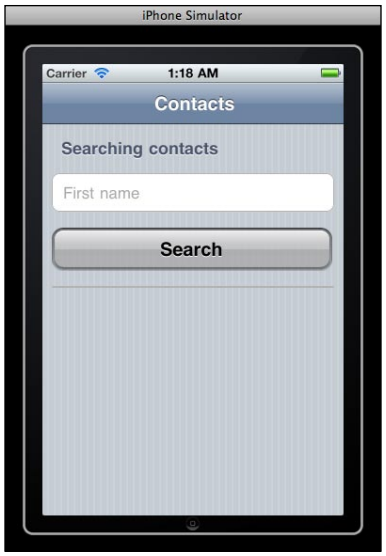
```
function onError(contactError) {
  navigator.notification.alert('Error!', 'Error', 'Error');
}


function searchContacts() {
  var options = {};
  options.nameFilter = document.frm.first_name.value;

  navigator.contacts.getAllContacts(onSucessContact, onError,
    options);
}

function displayContact(contactID) {
  navigator.contacts.displayContact(contactID, null,
    {"allowsEditing": false});
}

function onSucessContact(contacts) {
  var options = {};
  options.allowsEditing = true;
  var contactsLen = contacts.length;
  var names = "";
  if (contactsLen <= 0)
    names = "<li>No contacts founded</li>";
    for (var i = 0; i < contactsLen; i++) {
      var firstName = contacts[i].firstName;
      var lastName = contacts[i].lastName;
      names += '<li class="arrow"><a href="#"
        onclick="displayContact(' + contacts[i].recordID + ');' +
        '">' + firstName + " " + lastName + "</a></li>";
    }
  x$('#ul_contacts').html( 'inner', names);
}
```

Don't forget to comment out the `preventBehaviour(e)` JavaScript function and the line below it.

3. The last step will be to build our user interface using a simple form and a **Search** button:

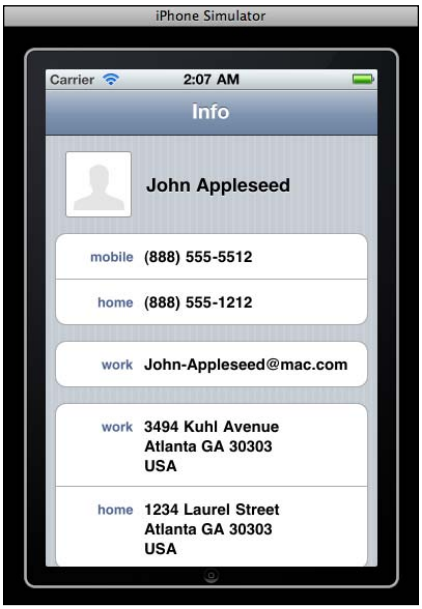
```
<div id="header">
  <h1>Contacts</h1>
</div>

<h1>Searching contacts</h1>

<form name="frm">
  <ul class="form">
    <li>
      <input type="text" name="first_name" value=""
        placeholder="First name" id="first_name"/>
    </li>
  </ul>
</form>

<p id="p_btn">
  <a href="#" id="mybtn" onclick="searchContacts()"
    class="button white">Search</a></p>

<ul id='ul_contacts'></ul>
```

4. When you're ready, load the new application for getting the main screen as shown in the following screenshot:



5. After typing a name inside the textbox and clicking on the **Search** button, a list with the result of the search will be displayed:

6. When the user clicks on one of the items of the list, the data for the selected contact will be displayed on a new page:



How it works...

The user interface for this recipe is quite simple. We only need a form with a text input, a button for searching, and an unordered list for displaying results. *UiUIKit* provides all of these widgets, so we used this framework.

Our button is using a handler for calling the `searchContact()` JavaScript function, which will invoke the method `getAllContacts()` provided by the `navigation.contacts` object of *PhoneGap*. This method is using three parameters: two callback functions and one options dictionary for setting the field criteria for searching. One of the callbacks will be launched if something is wrong and the other if everything is fine. The mentioned options dictionary is using a predefined variable called `nameFilteriable` and its value is retrieved up from our text input.

`onSuccessContact()` received as a parameter an array, containing all the matching contacts from the search results. We're iterating over this array for getting the first and last name for each contact. Both of them are assigned to different JavaScript variables and are used for building items for our unordered list. Also, we're adding an `onclick` handler for each one, which will invoke the `displayContact()` function after clicking on it.

Inside our `onSuccesContact()` function we checked if results are found for displaying a specific message if we don't find contacts that match our query.

The `displayContact()` JavaScript function encapsulates the call to the function with the same name predefined inside the `navigator.contacts` object of *PhoneGap*. The responsibility of this function is to display all the data for the selected contact. As you can observe, we used an additional dictionary passed as a parameter to this function. Actually, the value `false` for `allowsEditing()` is set when we don't want the user to be able to change any value of our contact. Otherwise, you can set this property to `true`.

Finally, *XUI* allows us to create a list with different items at in runtime.

See also

- *Installing the PhoneGap framework* recipe in *Chapter 1*, *Frameworks make life easier*
- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Selecting contacts* recipe

9
Location, Location, Location

In this chapter, we will cover:

- Detecting current orientation
- Identifying the current location
- Opening Google Maps at a specific location
- Calculating distances between two points

Introduction

This chapter introduces you to the world of *geolocation* and how to take advantage of it using our mobile devices. Basically, **geolocation** is a term used for defining the action of getting an exact location using the latitude and longitude coordinates. HTML5 and some of our frameworks offer us functionalities and features for utilizing geolocation. Thanks to it, we can add these seemingly complex features to our web applications for *iOS* devices.

Google Maps is one of the most popular available web services and many people consider it as a de facto standard for different issues related to geolocation on the Web. One of our recipes for this chapter uses the services provided by Google Maps for displaying a map and adding a map marker at a specific location.

Native applications for the iPhone are able to access the functionalities and features of the internal GPS receptor of the device directly. However, this is not possible for web applications. Luckily, some of these features can be replaced for those provided by HTML5. Also, some frameworks, such as *PhoneGap* and *Sencha Touch*, offer these native functionalities.

As you know, iPhone includes an accelerometer for detecting the orientation of the device. This feature is simple in concept but it can be a very powerful enhancement for certain applications. You will find that the first recipe in this chapter is related to the accelerometer.

Detecting current orientation

In this recipe, we'll learn how to detect the current orientation of our device. The iPhone includes an accelerometer sensor, which is capable to detect its orientation. For example, if we move the device from its original position to the right, the current orientation will be changed. Despite this recipe being very simple, you can build a complex application based on the detection of the current orientation of the device.

The XHTML file required for this recipe can be located at `code/ch09/current_orientation.html` in the code bundle provided on the Packtpub site.

Getting ready

This recipe doesn't require any specific framework; however, we're going to use *iWebKit* for designing a simple layout for a basic user interface. Make sure you have this framework installed and ready to use.

How to do it...

1. As we're going to use the *iWebKit*, we need to create a new XHTML file, which includes the required lines for loading this framework:

```
<link href="../iwebkit/css/style.css" rel="stylesheet"
  media="screen" type="text/css" />
<script src="../iwebkit/javascript/functions.js"
  type="text/javascript"></script>
```

2. Our user interface will be defined by the following lines of HTML code:

```
<body onorientationchange="show_orientation()">
  <div id="topbar">
    <div id="title">Show orientation</div>
  </div>
</body>
```

3. For detecting and responding to the event rotation, we'll use a simple and useful JavaScript function like the following:

```
function show_orientation() {
  var orientation = window.orientation;
  var message = "Current orientation: ";
  switch(orientation) {
```

```
        case 0:
          message += "Portrait";
          break;
        case 90:
          message += "Landscape, counterclockwise";
          break;
        case -90:
          message += "Landscape, clockwise";
          break;
      }
      alert(message);
    }
```

4. Make sure to close your XHTML file properly with the proper tags.

How it works...

When the user rotates the device, an event is launched, and the code in this recipe captures the event. For this purpose, the `onorientationchange` handler is used inside the body tag. Then our `show_orientation()` callback function will be executed in response to the event, simply displaying an alert box with a message indicating the current position of the device.

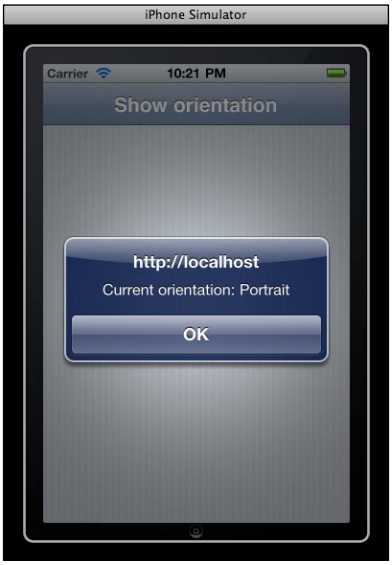
For detecting this orientation, *Safari Mobile* offers the `orientation` property, which belongs to the main window object. The value of this property is an integer and it is based on the degrees of the physical rotation.

In order to test this recipe, you can load your new page directly on the iPhone. Each time you rotate the device, a new alert box will be displayed. The next screenshots show you different messages responding to these rotations.

Even though we could have chosen a more sophisticated mechanism for displaying this information, our simple alert box keeps the example simple.

Maybe you're wondering why it can be useful to detect the rotation event. By default, the device autorotates the content, so for some applications you'll need to adjust the elements of the user interface. Otherwise, the automatic adjust made by the device will be enough and the content will be properly displayed. On the other hand, detecting the rotation could be required for those applications which use the features and capabilities of the accelerometer. For instance, many games require an intensive use of this component and it is very important to detect the rotation for responding to this action in different ways.

After changing the orientation of the device, you will get a result, as shown in the next screenshot:



See also

- *Installing the iWebKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Identifying the current location

iPhone, iPod Touch, and iPad share an important feature: All of them are mobile devices. Taking advantage of the built-in GPS sensor, we can identify the device's current location. This kind of information can be very useful for those developers interested in geolocation.

Let's find out how to develop a simple application for retrieving our current location using our mobile *iOS* device. Our application will display our current location on the screen using the standard latitude and longitude coordinates. This information will be displayed after the user clicks the provided button.

Getting ready

For this recipe, we'll use the *UiUIKit* framework for building the graphical user interface of our application. Additionally, the *XUI* framework will be used for manipulating the DOM model of the page for displaying our result. Check to make sure that both the frameworks are installed on your machine.

How to do it...

1. First, we're going to create an XHTML file with the required lines for loading the mentioned frameworks:

```
<script type="text/javascript"
  src="../xui/xui-core-1.0.0.js"></script>
<link rel="stylesheet"
  href="../uiuikit/stylesheets/iphone.css" />
```

2. The next task will be to add some CSS code to make our button more stylish:

```
<style type="text/css">
  #mybtn {
    margin-right: 12px;
  }
</style>
```

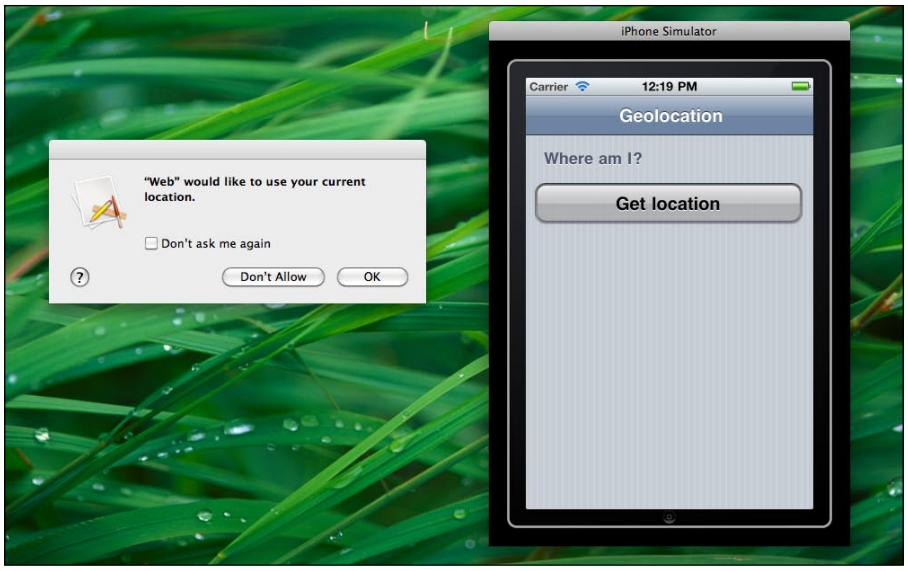
3. Regarding the JavaScript code for our recipe, add the following lines:

```
<script type="text/javascript" charset="utf-8">

  function onSuccess(pos) {
    var lat = pos.coords.latitude;
    var lon = pos.coords.longitude;
    var ele = "<p><strong>Your location is:</strong></p>";
    ele += "<p>Lat: " + lat + "</p>";
    ele += "<p>Lon: " + lon + "</p>";
    x$('#btn').html('after', ele);
  }

  function onError(msg) {
    alert("Error!: " + msg);
  }

  function get_location() {
    navigator.geolocation.getCurrentPosition(onSuccess, onError);
  }
</script>
```

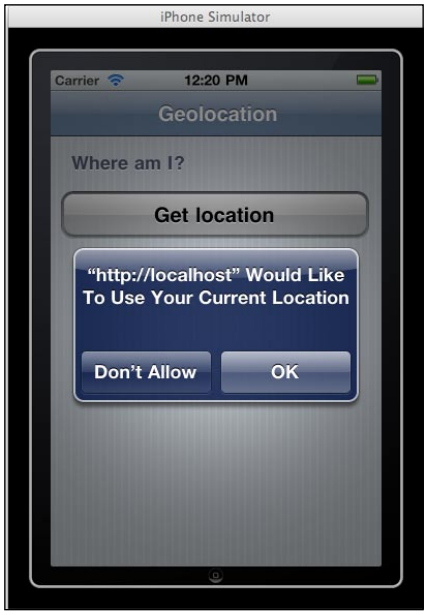
4. Finally, add the HTML code to define the user interface for the application:

```
<body>
  <div id="header">
    <h1>Geolocation</h1>
  </div>

  <h1>Where am I?</h1>

  <p id="btn">
    <a href="#" onclick="get_location()"
      class="button white" id="mybtn" >Get location</a>
  </p>
</body>
```

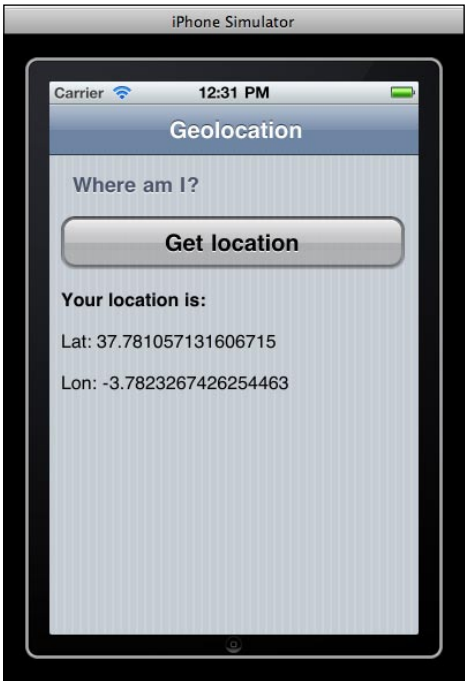
Save your new file and load it on your device. When the application is loaded, you can click on the **Get location** button. A new window will be displayed asking you to confirm if the browser could use your current location. For example, if you're using the iPhone Simulator, you'll see the two different windows as shown in the following screenshot:



If you're testing your application directly using the iPhone, you will see an *alert* box asking you the same question:



After clicking on the **OK** button to authorize the action, the application will display your current location:



How it works...

UiUIKit and *XUI* are used together in this recipe for building our user interface. The *UiUIKit* framework is used for defining a simple layout including a small button for getting our current location. With the *XUI* framework, we modify the DOM of the page to display the result of the `onSuccess()` JavaScript function. Actually, more specifically, we're inserting a paragraph element after our button.

The main JavaScript function of this recipe is called `get_location()`, which is invoked after clicking on the main button. The `onclick` handler of the button has the responsibility to call the function. *Safari Mobile* offers us a special object called `navigator.geolocation`, which has access to geolocation, through the `getCurrentPosition()` method. It works using two different parameters working like callback functions. The first one is called `onSucces()` and will be executed if everything is right and the position can be reached, the other `onError()` will be called if something goes wrong.

The `onSucess()` JavaScript function receives a parameter with the related information about our current location. Actually, the main object `coords` offers two properties: one for latitude and one for longitude. Both of them are required for the GPS systems to determine a specific location.

HTML5 defines a new API for working with geolocation. In fact, `navigator.geolocation` is an object introduced by this standard. *Safari Mobile* includes the ability to harness the native API of HTML5. Due to this fact, we used this object for getting our current location.

In addition to `latitude` and `longitude`, the `coords` object offers us other properties related to our current location. One of these is `altitude`, which represents the distance in metres with regards to the sea level.

There's more...

Surely, you will find the complete reference of HTML5 related to geolocation very interesting. It can be found at: `http://dev.w3.org/geo/api/spec-source.html`.

Some frameworks like *Sencha Touch* and *PhoneGap* include specific objects and methods for working with geolocation as an alternative to the native API of HTML5. However, some of the features offered by these frameworks just work as simple wrappers. This means that they encapsulate the geolocation API in objects with a higher level of abstraction. The aim of this technique is to make the job easier.

If you're interested in using *Sencha Touch*, you can take a look at the object `Ext.util.Geolocation`.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the XUI framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Opening Google Maps at a specific location* recipe

Opening Google Maps at a specific location

The code for this recipe shows you how to create a small application for loading a Google Map at a specific location and adding a map marker to it. For simplicity, we are going to use the current location. You've learned how to find the current location in the previous recipe. However, if you want, you can use another location of your choice.

For marking our location on the map, we'll use a *red pushpin*. Actually, it's the default map marker provided by Google Maps.

Even though the application developed for this recipe is quite simple, it sets up an important foundation for building complex applications, which require such functionality.

The files for this recipe can be found at `code/ch09/location_map/` in the code bundle provided on the Packtpub site.

Getting ready

The main requirement for this recipe is the *Sencha Touch* framework. Before continuing, make sure you have this framework installed on your computer or server.

Also, we need to use the Google Maps JavaScript API, but this JavaScript library can be loaded directly from the website of Google Maps, as you'll learn in this recipe.

How to do it...

1. Let's start by creating a new directory or folder inside the `DocumentRoot` directory of our web server. This new folder will be called `location_map`. After creating this folder, we need to create a new XHMTL file called `index.html`. In addition to the standard headers for this kind of file, we'll add the required lines for *Sencha Touch*:

```
<link rel="stylesheet" type="text/css"
  href="../../sencha-touch/resources/css/ext-touch.css"/>
<script type="text/javascript"
  src="../../sencha-touch/ext-touch-debug.js">
</script>
```

2. Our recipe requires the API offered by Google. Add the following lines to load this library:

```
<script type="text/javascript"
  src="http://maps.google.com/maps/api/js?sensor=true">
</script>
```

3. Add the following CSS to apply some styles to our user interface:

```
<style type="text/css">
  body {
    background: rgb(197,204,211);
    margin: 0;
    padding: 0;
  }
</style>
```

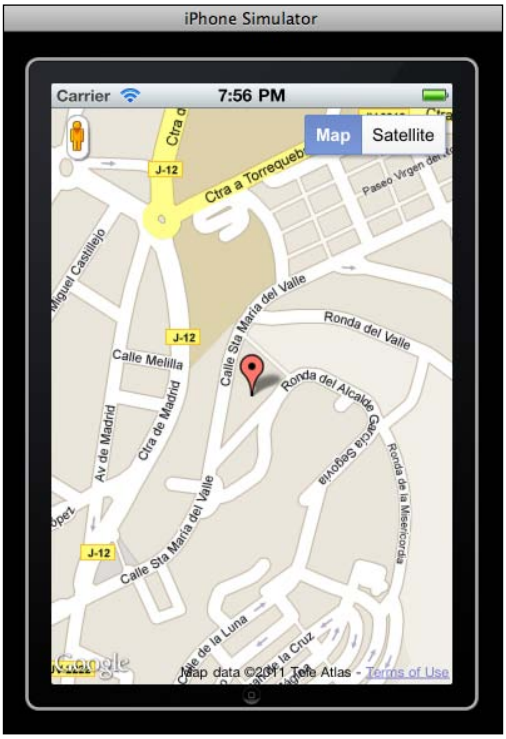
4. Don't forget to include a reference to the main JavaScript file of our *Sencha Touch* project:

```
<script type="text/javascript" src="main.js"></script>
```

5. The last lines of code for our `index.html` file will be the required lines for closing our `head`, `body`, and `html` tags. After saving this XHTML file, we're going to create a new one called `main.js`, which contains the required JavaScript code for our recipe. The code lines for this file are the following:

```
Ext.setup({
  onReady: function() {

    var map = new Ext.Map({
      title: 'Map',
      getLocation: true,
      mapOptions: {
        zoom: 16
      }
    });

    var panel = new Ext.Panel({
      fullscreen: true,
      items: [map]
    });

    var refresh = function() {
      var coords = map.geo.coords;
      var position = new
        google.maps.LatLng(coords.latitude, coords.longitude);
      setMarker(position);
    };

    var setMarker = function(position) {
      var marker = new google.maps.Marker({
        map: map.map,
        position: position
      });
    };

    map.geo.on('update', refresh);
  }
});
```

6. When you save `main.js`, you are ready to run and test our new application, which displays a Google Map showing our current location through a simple pin (marker):



How it works...

Thanks to the `Ext.Map` provided by *Sencha Touch*, we can load a Google Map for a specific location. This recipe is using our current location, but we can just as easily use another one. The parameter `getLocation` with the value `true` directs `Ext.Map` to use our current location. Another parameter is used for setting the level of zoom. In the example, we're using `16` to get a higher zoom level but you can use a custom value. For example, it's very common to use 12.

Our map is loaded using full screen. This is the reason for using a main object called `panel`, which represents a main widget for loading our map inside it. The parameter `fullscreen` with the value `true` is used for this purpose.

The `refresh()` function is a callback invoked by the method `map.geo.on()`. This JavaScript function gets our position and then uses `google.maps.LatLng` to create a specific object representing our current location. The `map.geo.coords` object encapsulates the information about the point, the most important properties being `latitude` and `longitude`.

Finally, the `setMarker()` method is used for displaying a mark inside the map for our location. As you can guess, `marker` is a new object belonging to the class `google.maps.Marker`. Actually, the Google Maps JavaScript API provides this class and `google.maps.LatLng` both of them are used for displaying our mark.

There's more...

A good resource for searching information related to the Google Maps API for JavaScript can be found at `http://code.google.com/apis/maps/documentation/javascript/`.

Sencha Touch offers us complete reference documentation for the object `Ext.Map` at `http://dev.sencha.com/deploy/touch/docs/output/Ext.Map.html`.

See also

- *Installing the Sencha Touch framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Identifying the current location* recipe

Calculating distances between two points

In this chapter, we've already leaned how to find out our current location and how to pinpoint it on Google Maps. A step further could be to calculate the distance between two points. For getting these points, we can use the two different points through the current location. This means that we'll need to change the physical location of our iPhone.

For this recipe, we're going to develop a simple application, which captures two points and displays the distance between them. Although this recipe is very simple, you can extend its functionality by adjusting it to your requirements. For example, you can write an application that receives two points from a widget. In this case, the user can type the data for both points. Also, your application can get two points automatically by using a timer or something like that.

The complete code for the XHTML file for this recipe can be found at `code/ch09/distances.html` in the code bundle provided on the Packtpub site.

Getting ready

Safari Mobile supports the HTML5 capabilities of geolocation, but we'll additionally use two frameworks to define the user interface for the application for this recipe, *XUI* and *UiUIKit*.

How to do it...

1. Create a new file called `distances.html` with the help of your favorite editor. Add the standard lines for setting up an XHTML file followed by the lines to load the *XUI* and *UiUIKit* frameworks:

```
<script type="text/javascript"
  src="../../xui/xui-2.0.0.min.js"></script>
<link rel="stylesheet"
  href="../../uiuikit/stylesheets/iphone.css" />
```

2. The next step will be to add the required JavaScript code for our recipe:

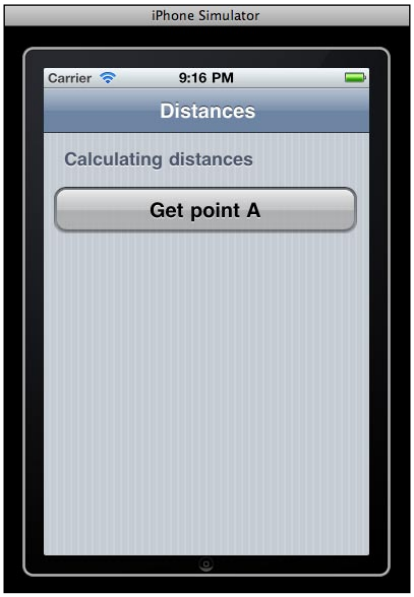
```
<script type="text/javascript" charset="utf-8">

  var pA;
  var pB;

  function onSuccessA(pos) {
    pA = pos.coords;
  }

  function onSuccessB(pos) {
    pB = pos.coords;
  }

  function onError(msg) {
    alert("Error!: " + msg);
  }

  function get_distance(pointA, pointB) {
    var distanceLat = (pointB.latitude -
      pointA.latitude).toRad();
    var distanceLong = (pointB.longitude –
      pointA.longitude).toRad();
    var a = Math.sin(distanceLat/2) *
      Math.sin(distanceLat/2) +
      Math.cos(lat1.toRad()) * Math.cos(lat2.toRad()) *
      Math.sin(distanceLong/2) * Math.sin(distanceLong/2);
    var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
    var distance = radius * c;
    var ele = "<p>Distance between A and B is: " + distance +
      " km.</p>";
    x$('#btn_distance').html('after', ele);
  }

  function get_distance_points() {
```

```
    get_distance(pA, pB);
  }

  function get_point_a() {
    x$('#btn_lnk_stop').setStyle('display', 'block');
    navigator.geolocation.getCurrentPosition(onSuccessA,
      onError);
  }

  function get_point_b() {
    x$('#btn_lnk_distance').setStyle('display', 'block');
    navigator.geolocation.getCurrentPosition(onSuccessB,
      onError);
  }
</script>
```

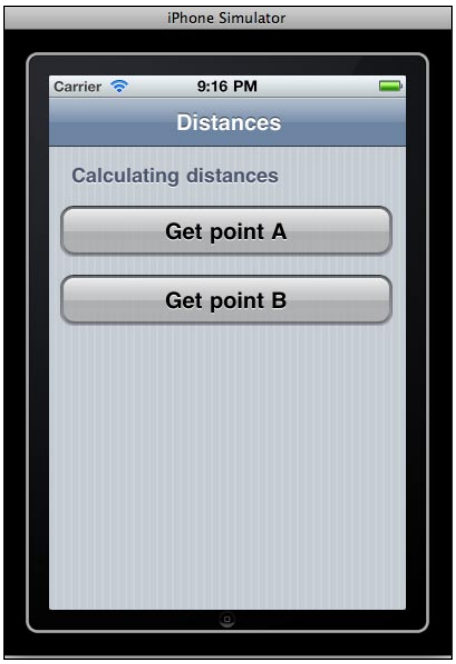
3. Finally, we're going to add the HTML code to define our user interface:

```
<body>
  <div id="header">
    <h1>Distances</h1>
  </div>

  <h1>Calculating distances</h1>

  <p id="btn_start">
    <a href="#" onclick="get_point_a()"
      class="button white">Get point A</a>
  </p>
  <p id="btn_stop">
    <a href="#" onclick="get_point_b()"
      class="button white" style="display: none"
    id="btn_lnk_stop">Get point B</a>
  </p>
  <p id="btn_distance">
    <a href="#" onclick="get_distance_points()"
      class="button white" style="display: none"
      id="btn_lnk_distance">Get distance</a>
  </p>
</body>
```

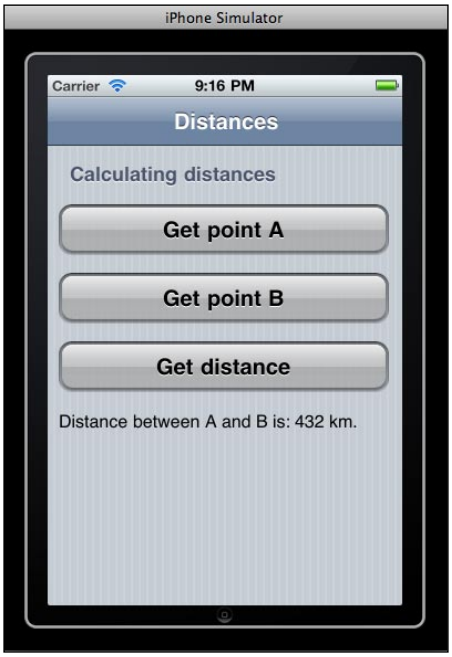
4. Save the new file and you'll be ready to test our new application. After loading it, you'll see the initial **Get point A** button, as shown in the following screenshot:



5. When the user clicks on the **Get point A**, a new button will be displayed:



6. After clicking on the **Get point B** button, a new **Get distance** button is added to calculate the distance between both points. Finally, when the user clicks on the **Get distance** button, the total distance is displayed:



How it works...

With regards to the user interface, we decided to use the *UiUIKit* framework because it helps us to easily build an attractive interface with buttons. Obviously, if you prefer to use another framework for this issue, you can do so without any problems. Each one of these buttons has an `onclick` handler, which invokes a different JavaScript function. The first one makes the **Get point B** button visible and captures our current location. When we're ready to select the second point for calculating the distance, we should click on the **Get point B** button. This button will get the new current location and make the third button visible. Finally, the third button (**Get distance**) will calculate the distance between the two points.

The method `navigator.geolocation.getCurrentPosition()` is used for getting our current location and the *XUI* framework helps us to make the button visible and to add a new paragraph element after the **Get distance** button.

For calculating the distance between the two points, we used a function called `get_distance()`. This JavaScript function can be found at `http://www.movable-type.co.uk/scripts/latlong.html`.

There's more...

Another interesting practical use of calculating distances—using the geolocation functionalities provided by HTML5—could be tracking different points by incrementing the distance from the last one. For this purpose, you can use the `navigator.geolocation.watchPosition()` method, which captures a new point each time the device changes its current location. When you desire to calculate the distance, you should invoke the `navigator.geolocation.clearWatch()` method. This JavaScript method will stop the tracking. Of course, you can use the `get_distance()` function for this recipe, but you'll need to apply some modifications for getting an incremental counter.

A complete reference for the `watchPosition()` method can be found at `http://dev.w3.org/geo/api/spec-source.html#watch-position`.

See also

- *Installing the XUI framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Identifying the current location* recipe

10
Web 2.0 Integration

In this chapter, we will cover:

- Embedding an RSS feed
- Opening a YouTube video
- Posting on your Facebook wall
- Retrieving recent tweets from Twitter
- Displaying photos from Flickr

Introduction

The last chapter of this book is dedicated to *mashup* applications. These kind of applications allow us to exchange data with other web applications or services. Web 2.0 applications provide this feature through different mechanisms.

Currently, some of the most popular websites like YouTube, Flickr, and Twitter provide a way for exchanging data through their API's. From the point of view of the user interfaces, *mashups* allow us to build rich interfaces for our application.

This chapter shows simple and effective ways to connect with the most important websites on the Internet. The techniques covered here shall provide a basis for building more complex applications. Our goal is to get a good understanding of each mechanism implemented in our recipes.

The first recipe for this chapter will cover embedding a standard RSS feed information. Later in our application we'll delve into YouTube, Facebook, Twitter, and Flickr and build some interesting *mashup* web applications for the iPhone.

Embedding an RSS feed

Our goal for this recipe will be to read an RSS feed and present the information provided in our application. In practice, we're going to use a feed offered by *The New York Times* newspaper, where each item provides a summary and a link to the original web page where the information resides. You could also choose another RSS feed for testing this recipe.

The code for this recipe can be found at `code/ch10/rss.html` in the code bundle provided on the Packtpub site.

Getting ready

Make sure *iWebKit* is installed in your computer before continuing.

How to do it...

1. As you've learned in the previous recipes, you need to create an XHTML file with the required headers for loading the files provided by *iWebKit*:

```
<link href="../iwebkit/css/style.css" rel="stylesheet"
  media="screen" type="text/css" />
<scriptsrc="../iwebkit/javascript/functions.js"
  type="text/javascript"></script>
```

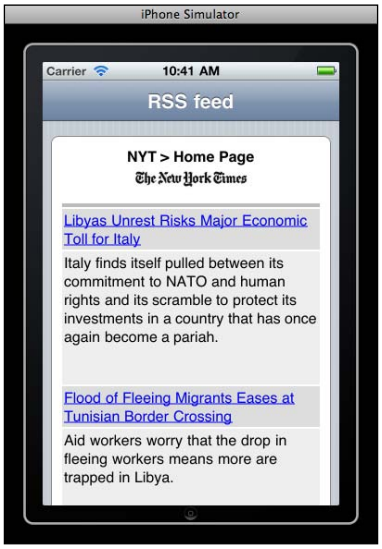
2. The second step will be to build our simple user interface containing only a top bar and an unordered list with one item. The top bar is added with the following lines:

```
<div id="topbar">
  <div id="title">RSS feed</div>
</div>
```

3. To add the unordered list, use the following code:

```
<div id="content">
  <ul class="pageitem">
    <li class="textbox">
      <p>
        <script src="http://rssxpress.ukoln.ac.uk/lite/
          viewer/?rss=http://www.nytimes.com/services/xml/
          rss/nyt/HomePage.xml" type="text/javascript"></script>
      </p>
    </li>
  </ul>
</div>
```

4. Finally, you should add the code for closing the body and html tags and save the new file as `rss.html`. After loading your new application, you will see a screen, as shown in the screenshot:



5. If you click on one of the items, *Safari Mobile* will open the web page for the article, as shown in the following screenshot:

How it works...

For avoiding complexity and keeping our recipe as simple as possible, we used a free web service provided by *RSSxpress*. This service is called *RSSxpressLite* and it works by returning a chunk of JavaScript code. This code inserts an HTML table, containing a summary and a link for each item provided by the referenced RSS feed. Thanks to this web service, we don't need to parse the response of the original feed; *RSSxpressLite* does the job for us.

If the mentioned web service returns the code that we need, you should only write a small line of JavaScript code referring to the web service through its URL and pass as a parameter the RSS feed for displaying information.

There's more...

For learning more about *RSSxpressLite*, take a look at `http://rssxpress.ukoln.ac.uk/lite/include/`.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Opening a YouTube video

It is safe to say that everyone who uses the Internet knows of YouTube. It is one of the most popular websites in the world. Millions of people use YouTube to watch videos through an assortment of devices, such as PC's, tablets, and smartphones. Apple's devices are not an exception and of course we can watch YouTube videos on the iPhone and iPad.

In this case, we're going to load a YouTube video when the user clicks on a specific button. The link will open a new web page, which allows us to play it.

The simple XHTML recipe can be found at `code/ch10/youtube.html` in the code bundle provided on the Packtpub site.

Getting ready

This recipe only requires the *UiUIKit* framework for building the user interface for this application. You can use your favorite YouTube video for this recipe. By default, we're using a video provided by Apple introducing the new iPad 2 device.

How to do it...

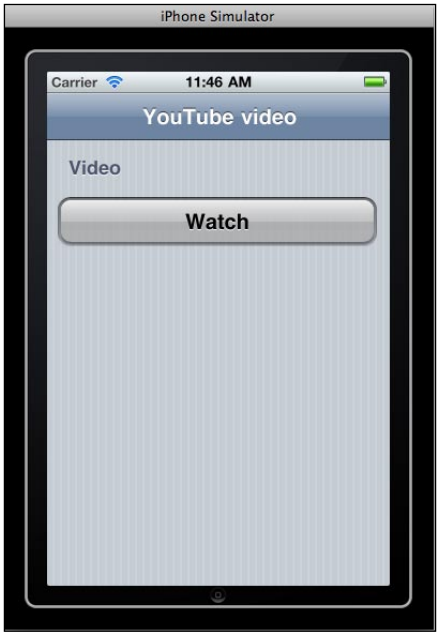
1. Following the example from the previous recipe, create a new XHTML file called `youtube.html` and insert the standard headers for loading the *UiUIKit* framework. Then add the following CSS inside the `<head>` section to style the main button:

```
<style type="text/css">
  #btn {
    margin-right: 12px;
  }
</style>
```

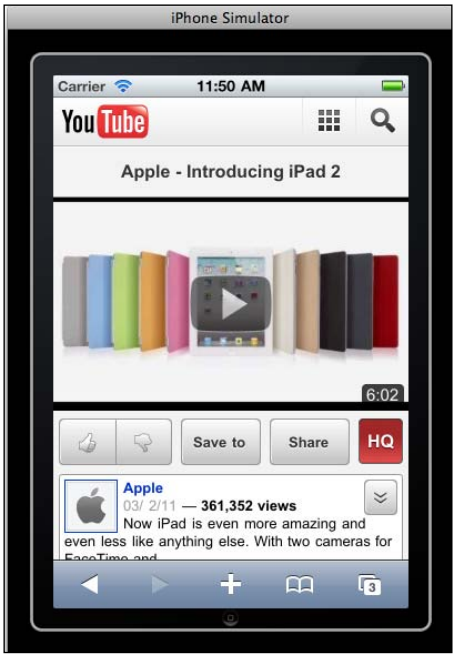
2. Our graphical user interface will be completed by adding the following XHTML code:

```
<div id="header">
  <h1>YouTube video</h1>
</div>

<h1>Video</h1>

<p id="btn">
  <a href="http://m.youtube.com/watch?v=Z_d6_gbb90I"
    class="button white">Watch</a>
</p>
```

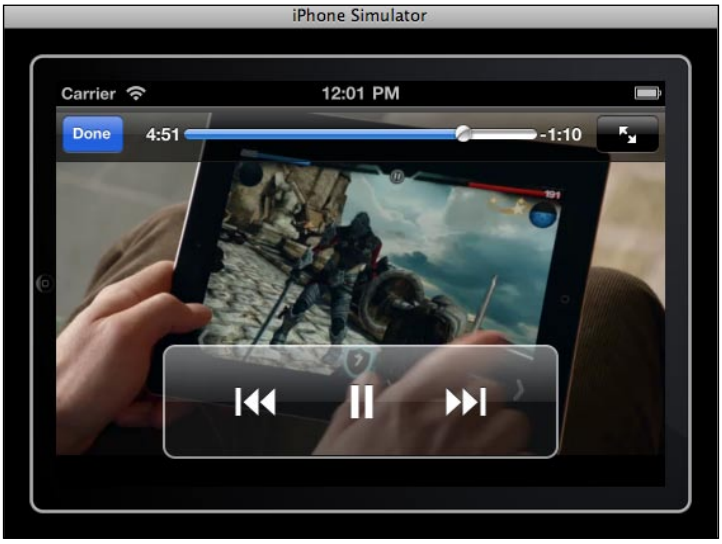
3. After loading the new application on your device, you will see a screen similar to the following screenshot:



4. When the user clicks on our main button, *Safari Mobile* will go to the web page of the video at YouTube, as shown in the following screenshot:



5. After clicking on the play button, the video will start playing. We can rotate our device for a better aspect ratio, as shown in the following screenshot:



How it works...

This recipe is pretty simple; we only need to create a link to the desired YouTube video. The most important thing to keep in mind is that we'll use a mobile-specific domain for loading our video to mobile devices. The URL is simply `http://m.youtube.com`, instead of the regular URL `http://www.youtube.com`.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Posting on your Facebook wall

The application developed for this recipe shows how to authenticate with Facebook and how to write a post on your public wall. If everything is successful, an alert box is used to report it to the user.

Although there are myriad complex applications with better functionalities that can be built for Facebook, we will focus on simple posting in this recipe. This application will only allow you to post on your wall. For this you need to hardcode your Facebook account for posting. This is to keep the recipe as simple as possible and to get a good understanding of all the complex processes involved in dealing with the `OAuth` protocol used by Facebook. However, thanks to this open protocol, it is easier to allow secure authentication of APIs from web applications.

Also, this recipe requires using a real Internet domain and a server with public access. Thus, you cannot test this application on your local machine. Our application needs to use a server-side language for which we'll use PHP. Currently, it's very easy to find hosting services for PHP applications. You can also find very cheap services for hosting your PHP code.

You can find the complete code for this recipe at `code/ch10/facebook/` in the code bundle provided on the Packtpub site.

Getting ready

To build the application for this recipe, you need to have a public server with an Internet domain linked to it. Also, you must install a web server with a PHP interpreter and have the *UiUIKit* framework ready to use.

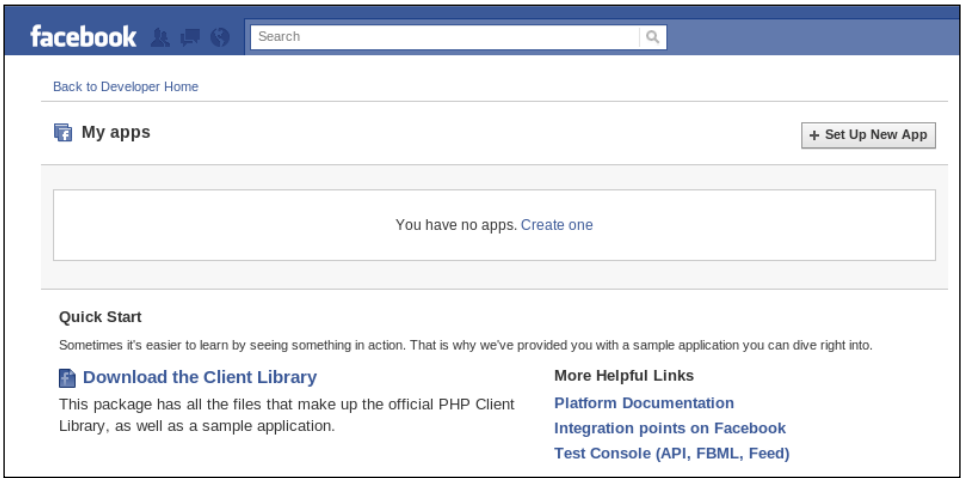
You need to install the `cURL` library, which allows PHP to connect and communicate to many different types of servers. Two interesting resources for this issue are:

- `http://www.php.net/manual/en/book.curl.php`
- `http://www.php.net/manual/en/curl.setup.php`

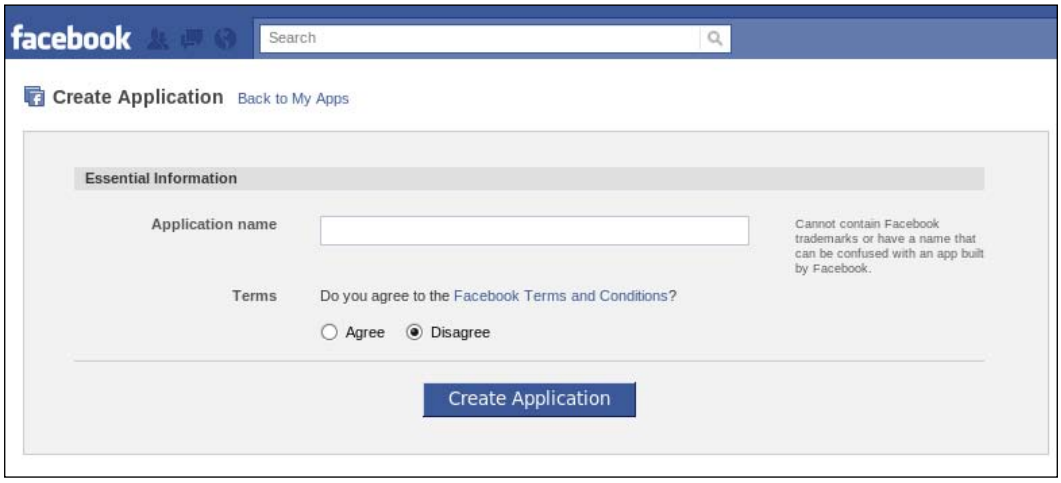
How to do it...

1. The first step required for this recipe is to make sure that you have a public server with PHP and a web server installed and running. As a web developer, you may already have one. At least, you're capable of configuring this server-side environment quickly and without too much effort. We're going to assume that your server environment is ready before continuing.
2. The next step is easier than the first one. Just check if your *UiUIKit* framework is installed in your server.
3. As we're going to use the Facebook Graph API, we'll need to find out the specific Facebook ID (identifier) for our user. In order to do that, we go to Facebook's home page. After authenticating, you can click on the **Profile** link and a new URL will appear on your browser's navigation bar. This URL ends with a string after a slash character. Actually, the mentioned URL has this structure:

   `http://www.facebook.com/home.php#!/<profile_name>`, where `<profile_name>` is your Facebook ID. Now your ID is strictly a number, but we'll use the string mentioned earlier, which works as an alias to the ID.
4. Another important and required step will be to register our application with Facebook and fortunately, this process is straigtforward. Since Facebook needs to verify that you're the owner of the profile and can create applications, you'll be asked to provide your mobile phone or credit card information. It's simpler and faster to use your mobile phone. After typing your number, you'll receive an SMS with a code. Then, type this code in the required field of the form. The process starts by accessing `http://developers.facebook.com`. Then, click on the **My apps** link. When the new web page is loaded, push the **+ Set Up New App**. You can find a form, fill it and click on the **Create Application** button. Now, you've got your application registered and have received a unique set of values like Application ID, API Key, and App Secret. We'll use some of these values later.



5. After clicking on the **Create one** link, you'll see a form as shown in the next screenshot:



6. Now it's time to build our main page for allowing the user to write a post. For taking this action, we're going to create a new XHTML page (`iposting.html`) inside a new folder called `facebook` under the control of your web server. We'll use the standard XHTML headers plus the required lines for loading the *UiUIKit* framework. Then, we'll add the following lines of code for building the user interface:

```
<div id="header">
  <h1>iPosting</h1>
</div>
<form name="frm" method="post" action="posting.php">
  <ul class="form">
    <li>
      <textarea name="ptext" rows="4"
        placeholder="Type here"></textarea>
    </li>
  </ul>
  <p>
    <a href="javascript: document.frm.submit()"
      class="button white">Post to your wall</a>
  </p>
</form>
```

7. When this file is ready, we need to create our PHP page for calling Facebook and posting on our wall. Our PHP is named `posting.php` and we'll start by adding the following lines:

```
session_start();

$ptext = '';
if (empty($_REQUEST['ptext'])) {
  $ptext = $_SESSION['ptext'];
} else {
  $ptext = $_REQUEST['ptext'];
  $_SESSION['ptext'] = $ptext;
}
```

8. Then, we need to hardcode some values:

```
$app_id = your_app_id;
$app_secret = "your app secret here";
$my_url = "your redirect page";
$scope = "offline_access,publish_stream";
$url_fb = "https://graph.facebook.com/<ID>/feed";
```

   Don't forget to change the values for the `$app_secret` and `$url_fb` variables. You must use your own values.

9. The following lines handle communication and authentication with Facebook:

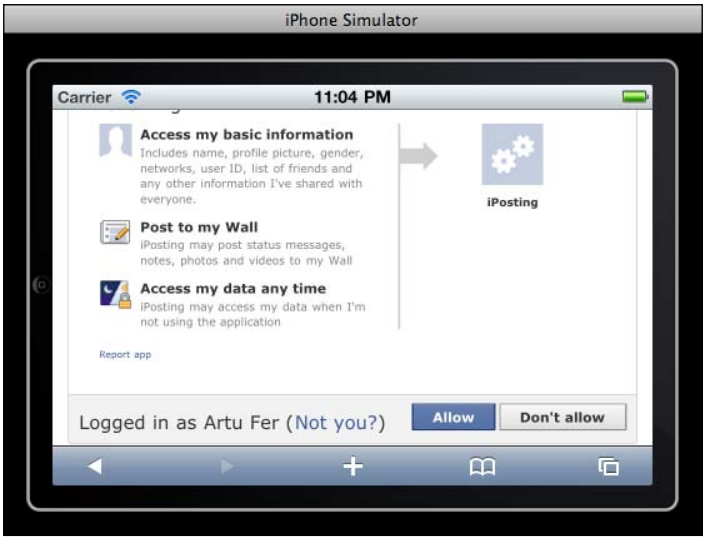
```
$code = $_REQUEST["code"]

if(empty($code)) {
  $dialog_url =
    "http://www.facebook.com/dialog/oauth?client_id="
    $app_id . "&redirect_uri=" . urlencode($my_url) .
    "&scope=". $scope;

  echo("<script>top.location.href='" . $dialog_url . "'
    </script>");
}

$token_url =
  "https://graph.facebook.com/oauth/access_token?client_id="
    $app_id . "&redirect_uri=" . urlencode($my_url) .
    "&client_secret=". $app_secret . "&code=" . $code;

$access_token = file_get_contents($token_url);

$arr_token = split("access_token=", $access_token);
$acc_token = $arr_token[1];

$graph_url = "https://graph.facebook.com/me?" . $access_token;

$user = json_decode(file_get_contents($graph_url));
```

10. Once we have permission for posting, we need to get a POST request using `cURL`:

```
$fields = array(
  'access_token'=>urlencode($acc_token),
  'message'=>urlencode($ptext)
);
$fields_string = '';

foreach($fields as $key=>$value) { $fields_string .=
  $key.'='.$value.'&'; }
  rtrim($fields_string,'&');

$ch = curl_init();
curl_setopt($ch,CURLOPT_URL,$url_fb);
curl_setopt($ch,CURLOPT_POST,count($fields));
curl_setopt($ch,CURLOPT_POSTFIELDS,$fields_string);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

curl_exec($ch);
curl_close($ch);
```

11. Finally, we close our PHP script adding the same XHTML code used in `iposting.html`. After taking this action, we'll add the following JavaScript code before closing the `<body>` tag:

```
<script type='text/javascript'>
  alert('Text was published in your wall');
</script>
```



12. The result of posting on your wall is shown in the following screenshot:

13. Now we have finished coding. To test that everything works, you'll need to upload the application to your server. Then, you can load the following URL in your device: `http://yourdomain/iposting.html`.

How it works...

Once we have our new application registered with Facebook, we can start developing it. The first page (`posting.html`) works as the starting point and it offers a `textarea` widget for writing the post, which will be sent to our wall. After clicking on the main button, a standard HTTP POST request is sent to our `iposting.php` page. This PHP script checks and stores the content of our widget in a session variable. We need to do this action because `iposting.php` will call the Facebook API in order to authenticate our application. Without storing this value in the session, it will be lost and will not be authenticated. Actually, our PHP script first calls a web page specifically designed by Facebook for using the `OAuth` protocol. If the user is not logged in our application will redirect to a Facebook login page. Once the user has been authenticated by entering the e-mail and password, the control will be returned to our `posting.php`. This is the reason for using the following code line:

```
$my_url = "http://yourdomain/posting.php";
```

After our PHP recovers the control, it will do a HTTP POST request using `cURL`. This request will write our post on the wall.

The values `offline_access` and `publish_stream` are used by the variable `$scope` for indicating which kind of permissions we require for our application. Actually, these two values are required for posting from our iPhone application. Thanks to these parameters, Facebook knows which questions will be asked to the user for exchanging data with our application. This means that the user always knows what kind of actions could be taken for the application.

We just copy the XHTML code inside our PHP scripts to make it easier. The complex code could be confusing so it is better to copy this code for understanding how it works. Obviously, you should refactor this code for better performance.

For a real production application, we recommend to check for errors before making the HTTP POST to Facebook. Remember, some changes will be required to get this application production ready. Don't forget our goal was to keep it simple.

There's more...

It could be very interesting to take a look at the Facebook Graph API documentation at `http://developers.facebook.com/docs/reference/api/`, where you can find a lot of information for building more complex web applications for Facebook interaction.

More information about how the Facebook's authentication process works can be located at `http://developers.facebook.com/docs/authentication`.

If you're interested in finding out which is the `OAuth` protocol and how it works, you can visit its official website at `http://oauth.net/`.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Retrieving recent tweets from Twitter

Undoubtedly, Twitter is one of the most popular Web 2.0 applications on the Net. Many developers have found ways to connect this service to their iPhone applications .This recipe shows you how to load *n* tweets from a specific Twitter user. All of these tweets will be displayed in a simple list.

The required code for this recipe is a simple XHTML file called `code/ch10/twitter.html` in the code bundle provided on the Packtpub site.

Getting ready

First you need a Twitter account name; you may already have one. Then, make sure you've installed *XUI* and *UiUIKit* frameworks on your machine.

How to do it...

1. After creating a standard XHTML file with the required lines for loading both our frameworks, add the following lines to set up the user interface:

```
<body id="normal">
       <div id="header">
              <h1>Last tweets</h1>
       </div>
       <ul id="tlist"></ul>
```

2. Then, we're going to add some JavaScript code for retrieving our last tweets:

```
<script>
  functionmy_tweets(tweets) {
    varele = "";
    for (i=0; i<tweets.length; i++) {
      ele += "<li>" + tweets[i].text + "</li>";
    }
    x$('#tlist').html(ele);
  }
</script>

<script src="http://twitter.com/statuses/user_timeline/
  bsnux.json?callback=my_tweets&count=5"></script>
```

3. Finally, close and save your new file. After loading it on your device, you will see the result as is shown in the following screenshot:

How it works...

The *UiUIKit* framework is used in this recipe for building our simple user interface. On the other hand, *XUI* helps us to process the result response sent by Twitter. In order to do that, we only need to use a JavaScript line for making a JSON request through AJAX. Actually, our recipe is using the Twitter username *bsnux* and we're loading the latest five tweets posted by this user. For using your own user, just replace the mentioned username with a user of your choice. Also, you can change the number of the requested tweets by changing the value of the `count` variable.

The JavaScript `my_tweets()` function works as a callback, receiving the tweets and processing the response for creating our simple list.

There's more...

We can modify our original recipe for displaying the profile picture of the owner of each tweet. The change is pretty simple, just replace the JavaScript line:

```
ele += "<li>" + tweets[i].text + "</li>";
```

With the following line:

```
ele += "<li><img class='ico' src='" +
  tweets[i].user.profile_image_url + "' />" + tweets[i].text +
  "</li>";
```

The result of loading this new application is shown in the next screenshot:

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the XUI framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*

Displaying photos from Flickr

The last recipe of this chapter covers the Flickr service. This website is one of the most popular services for uploading and sharing photos online. We'll build an application for loading the last 20 photos published by a user. Later, the photos will be displayed using a *grid* widget.

Getting ready

Also with the previous recipes in this chapter, we're going to use the XUI and UIUiKit frameworks. Make sure both of them are installed and ready to use on your machine.

How to do it...

1. After creating your new `flickr.html` file with the standard XHTML headers and the required lines for loading both our frameworks, we'll be ready to add the code for this recipe.
2. First we'll set CSS styles for a consistent background color for our application:

```
<style type="text/css">
body {
  background: rgb(197,204,211) url(../images/stripes.png)
!important;
}
</style>
```

3. Then we'll add JavaScript code so we can connect to the Flickr API and get our photos:

```
function buildPhotoURL(photo, psize) {
  psize = (typeof(psize) == 'undefined') ? '_s' : psize;
  var url = "http://farm" + photo.farm + ".static.flickr.com/" +
   photo.server +"/" + photo.id + "_" + photo.secret +  psize +
   ".jpg";
  return url;
}

function get_data() {
  x$(window).xhr('http://api.flickr.com/services/rest/?
   method=flickr.people.getPublicPhotos&per_page=20&page=1
   &user_id=41152606@N03&jsoncallback=?
   &api_key=1a66f4e75c44f13656d97a712a443740&format=json',
  {
    callback: function() {
      var jsonData =
        this.responseText.substring(14,
        (this.responseText.length-1));
      var data = eval("(" + jsonData + ')');
      var ele = "";
      for (var i=0; i < data.photos.photo.length; i++) {
        var photoURL =
          buildPhotoURL(data.photos.photo[i]);
```

```
            ele += "<li><imgsrc=" + photoURL + " />";
          }
          x$('#grid').html(ele);
        }
      });

    }
```

4. Finally, we're going to add the html elements for building our user interface with the grid widget:

```
<body id="images" onload="javascript: get_data()">
  <div id="header">
    <h1>Flickr Album</h1>
  </div>

  <ul id="grid">
  </ul>
</body>
```

5. The final result is shown in the next screenshot:

How it works...

The Flickr API provides a method for retrieving the latest tweets for a specific user. We only need to call to make a JSON request. Our application is using the `on_load` handler for retrieving the pictures directly after the application loads.

Using *XUI*, we're adding a new `<li>` element to our main `<ul>` tag to represent our grid widget. The response sent by Flickr is a JSON object containing different information about our photos. This object is an array where each element offers us information about each photo.

Additionally, our `buildPhotoURL()` JavaScript function is used for building an absolute URL to each one of the photos. Thanks to this technique, when the user clicks on one picture, *Safari Mobile* will load the original photo from Flickr.

See also

- *Installing the UiUIKit framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Installing the XUI framework* recipe in *Chapter 1*, *Frameworks Make Life Easier*
- *Creating a grid with images* recipe in *Chapter 4*, *A Picture Speaks a Thousand Words*

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Y

Thank you for buying
iPhone JavaScript Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

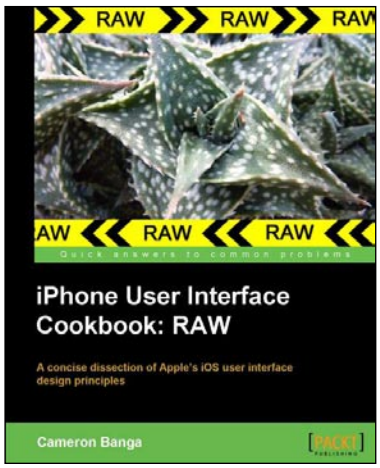
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] PUBLISHING



iPhone User Interface Cookbook: RAW

ISBN: 978-1-849691-14-7 Paperback: 500 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

---

iPhone Applications Tune-Up

ISBN: 978-1-84969-034-8 Paperback: 312 pages

High performance tuning guide for real-world iOS projects

1. Tune up every aspect of your iOS application for greater levels of stability and performance
2. Improve the users' experience by boosting the performance of your app
3. Learn to use Xcode's powerful native features to increase productivity
4. Profile and measure every operation of your application for performance