Quick answers to common problems

Sencha Touch Cookbook

Second Edition

Over 100 hands-on recipes to help you understand the complete Sencha Touch framework and solve your day-to-day problems

Ajit Kumar

[PACKT] open source* community experience distilled PUBLISHING

Sencha Touch Cookbook

Second Edition

Over 100 hands-on recipes to help you understand the complete Sencha Touch framework and solve your day-to-day problems

Ajit Kumar

[PACKT] open source
PUBLISHING
community experience distilled

BIRMINGHAM - MUMBAI

Sencha Touch Cookbook

Second Edition

First published: December 2011

Second edition: August 2013

Production Reference: 1130813

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

ISBN 978-1-78216-918-5

`www.packtpub.com`

Cover Image by Abhishek Pandey (`abhishek.pandey1210@gmail.com`)

Credits

Author
Ajit Kumar

Reviewers
Joseph Khan
Michael McCurrey

Acquisition Editor
Usha Iyer

Lead Technical Editor
Anila Vincent

Technical Editors
Shali Sasidharan
Hardik Soni
Sharvari H. Baet

Copy Editors
Brandt D'Mello
Gladson Monteiro
Insiya Morbiwala
Aditya Nair
Alfida Paiva
Adithi Shetty
Laxmi Subramanian

Project Coordinator
Apeksha Chitnis

Proofreader
Stephen Copestake

Indexer
Monica Ajmera Mehta

Graphics
Valentina Dsilva

Production Coordinator
Nitesh Thakur

Cover Work
Nitesh Thakur

About the Author

Ajit Kumar started his IT career with Honeywell, Bangalore in the field of embedded systems and moved on to enterprise business applications (such as ERP) in his 13 years' career. From day one, he has been a staunch supporter and promoter of open source and believes strongly that open source is the way for a liberal, diversified, and democratic setup, such as India. He dreams, and continuously strives to ensure, that architecture, frameworks, and tools must facilitate software development at the speed of thought.

Ajit holds a Bachelor's degree in Computer Science and Engineering from the Bihar Institute of Technology, Sindri. He co-founded Walking Tree, which is based out of Hyderabad, India where he plays the role of CTO and works on fulfilling his vision.

Prior to writing this book, he worked on the following titles by *Packt Publishing*:

- *ADempiere 3.6 Cookbook*
- *Sencha Touch Cookbook*
- *Sencha MVC Architecture*

I would like to thank my wife Priti for her untiring support, my family, and my team at Walking Tree for their constant motivation, the readers of the first edition of this book for their encouraging feedback, and the team behind the Sencha Touch framework.

About the Reviewers

Joseph Khan is a Senior Web Developer at GoldSpot Media where he specializes in HTML5 standard mobile web apps, JavaScript/CSS3 standard rich media apps, and other Rich Internet Applications (RIA). Before moving into mobile web development, he was working with Adobe Flex, Action Script, and Flash technologies and developed data visualization and enterprise dashboard-based applications for clients such as Cisco, The World Bank, AADI, and other global organizations. His liking for mobile web development occurred recently and he has been hooked ever since. He also likes Phonegap, PHP, Drupal, and Python.

He has a Bachelor's degree in Computer Science from N.I.T Silchar, India and has been working on the Web and related technologies for six years.

Besides his regular work he also likes to design cars and motorbikes, ride his Yamaha, and look for good food. Find out more about him and all his work at `http://jbkflex.wordpress.com/`.

He is also the author of *Instant Adobe Edge Inspect Starter*, *Packt Publishing*.

> I would like to dedicate this book to my parents and my wife Nilofer without whom I would not have been here, and especially to my 5-month-old baby boy Ayaan.

Michael McCurrey has been working in the software development industry for over 15 years. He has been party to the success of many notable software startups including SalesLogix and Trans-soft. Besides technical editing titles, he works as the Software Development Manager at Ping Golf in Phoenix, Arizona. He lives in Arizona with his wife, Sunni, and their three children Mickie, Zachary, and Daimon.

Table of Contents

Preface

Sencha touch is a versatile HTML5-based framework for developing mobile web applications that look and feel native on touchscreen devices; with it, you can write your code once and deploy it to both iOS and Android, saving you both time and money.

The *Sencha Touch Cookbook Second Edition* has a comprehensive selection of recipes covering everything from installation right through to HTML5 geolocation.

This book really is your one-stop resource for cross-platform HTML5 application development. It covers the basics such as setting up an iOS, Android, and Blackberry development environment right through to much more complex development issues such as animation, rich media, geolocation, and device features. Every recipe is practically focused. Maximum action. Minimum theory.

What this book covers

Chapter 1, *Gear Up for the Journey*, covers the steps required to set up the environment to develop, deploy, and test Sencha Touch-based applications.

Chapter 2, *Catering to Your Form-related Needs*, explains how to create a form using Sencha Touch and the different form fields that can be used to work effectively with the forms.

Chapter 3, *Containers and Layouts*, covers all the layouts that the framework offers and how one can use them to structure their widgets on the screen.

Chapter 4, *Building Custom Views*, explains how one can create very custom-looking, data-driven, interactive view, and which classes need to be used for the same.

Chapter 5, *Dealing with Data and Data Sources*, covers every aspect of loading data from local or remote data sources, dealing with XML/JSON data, data translation, building client-side caches, and sharing the data across different UI components.

Chapter 6, *Adding Components*, covers components such as lists, buttons, the picker, action sheets, and tab panels. It also explains how to create new components or extend the existing ones.

Chapter 7, *Adding Audio/Visual Appeal*, covers how to work with audio and video components. Also, it covers the different charts that one can use to present data, visually, to the user.

Chapter 8, *Taking Your Application Offline*, introduces the concept of offline support and covers the steps that an application developer shall take to make their application work in online as well as offline mode.

Chapter 9, *Increased Relevance Using Geolocation*, covers how to get geolocation details inside the application and integrate it with Google Map.

Chapter 10, *Device Integration*, explains the different device features across different devices (for example, cameras, contacts, back button, and home button) and how one can integrate them inside the application to give a better user experience to the device user.

What you need for this book

To run the samples, provided in the book, you need the following software:

- Sun JDK Version 1.5 or above
- Android Developer Tools (ADT) Bundled Eclipse
- Apache Cordova 2.4.0
- Apache Ant 1.8.4 or above
- Sencha Touch 2.2.1 library
- XCode 4
- BlackBerry WebWorks SDK

Who this book is for

This book is ideal for anyone who wants to gain the practical knowledge involved in using Sencha Touch mobile web application framework to make attractive web applications for mobile phones. If you have some familiarity with HTML and CSS, then this book is for you. This book will give designers the skills they need to implement their ideas and provide developers with creative inspiration through practical examples. It is assumed that you know how to use touchscreens, touch events, WebKit on mobile systems, Apple iOS, Google Android for mobile phones, and BlackBerry.

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: "Create and open a new file `ch01_05.js` in the `ch01` folder and paste the following code in it."

A block of code is set as follows:

```
Ext.application({
    name: 'MyApp',
    launch: function() {
```

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

```
Ext.application({
    name: 'MyApp',
    launch: function() {
```

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

```
ant blackberry build
```

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: "Launch Xcode and go to **Preferences**."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

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

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

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

Customer support

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

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you. You can also find the code at: `https://github.com/ajit-kumar-azad/SenchaTouch2Cookbook`

Errata

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

Piracy

Piracy of 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
Gear Up for the Journey

In this chapter we will cover:

- Setting up the Android-based development environment
- Setting up the iOS-based development environment
- Setting up the BlackBerry-based development environment
- Setting up a browser-based development environment
- Detecting the device
- Finding features that are supported in the current environment
- Letting your application configure itself using profiles
- Responding to orientation changes

Introduction

As with any other development, the first and foremost thing that is required before we embark on our journey is setting up the right environment so that development, deployment, and testing become easier and effective. And this calls for a list of tools that are appropriate in this context. In this chapter, we will cover topics related to setting up the environment using the right set of tools. Sencha Touch is an HTML5-based JavaScript framework to build applications for touch devices. An application built using Sencha Touch can be accessed from a web browser or can be packaged for the target touch device and run on it. Sencha Touch 2.2 supports WebKit, the IE 10 browser on Windows 8, and the following platforms:

- Android
- iOS
- BlackBerry
- Windows Phone 8

For each of these platforms, we will see what steps we need to follow to set up the complete development and deployment environment. We will be packaging our Sencha Touch-based application using Apache Cordova. Cordova is another JavaScript framework, which provides two important capabilities:

- The APIs needed to access the device features, such as camera and address book
- A build mechanism for writing the code once (in the form of JS, HTML and CSS) and packaging them for different platforms such as iOS and Android

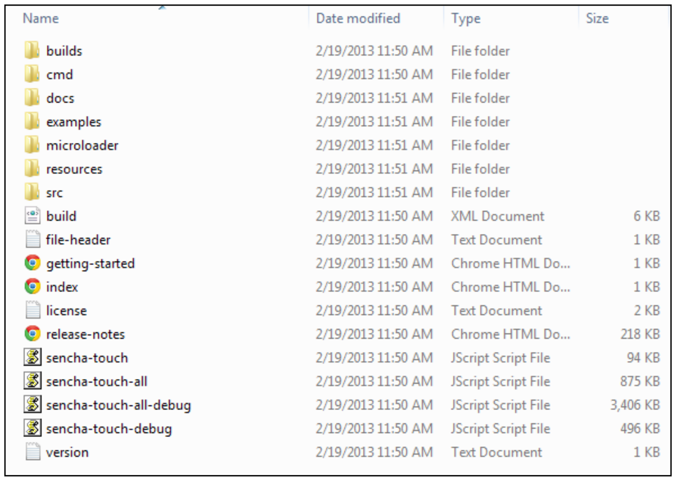
Throughout the book we will be using the following software:

- Sun JDK version 1.5 or above (required to run Eclipse)
- Android Developer Tools (ADT-bundled Eclipse), which is required for Android development
- Apache Cordova 2.4.0 (required for using device features and packaging the application for different platforms)
- Apache Ant 1.8.4 or above (required to run a project build in Eclipse and Cordova tools)
- Sencha Touch 2.2.1 library (Sencha Touch SDK)
- Xcode 4 (required for iOS development)
- BlackBerry WebWorks SDK (required for BlackBerry development)

Before we get any further, you should download and install the following, which will act as a common base for all our discussions:

- Sun JDK 1.5 or above
- Android Developer Tools (ADT-bundled Eclipse)
- Sencha Touch 2.2.1 library

After downloading Sencha Touch library, extract it to a folder, say, `C:\sencha-touch-cookbook\softwares\touch-2.2.1`. After the extraction, you should see the folders as shown in the following screenshot:

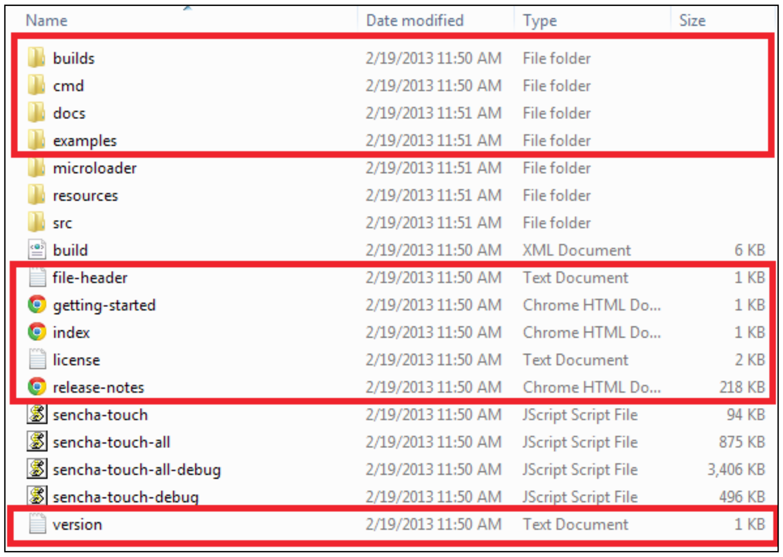


Name	Date modified	Type	Size
builds	2/19/2013 11:50 AM	File folder	
cmd	2/19/2013 11:50 AM	File folder	
docs	2/19/2013 11:51 AM	File folder	
examples	2/19/2013 11:51 AM	File folder	
microloader	2/19/2013 11:51 AM	File folder	
resources	2/19/2013 11:51 AM	File folder	
src	2/19/2013 11:51 AM	File folder	
build	2/19/2013 11:50 AM	XML Document	6 KB
file-header	2/19/2013 11:50 AM	Text Document	1 KB
getting-started	2/19/2013 11:50 AM	Chrome HTML Do...	1 KB
index	2/19/2013 11:50 AM	Chrome HTML Do...	1 KB
license	2/19/2013 11:50 AM	Text Document	2 KB
release-notes	2/19/2013 11:50 AM	Chrome HTML Do...	218 KB
sencha-touch	2/19/2013 11:50 AM	JScript Script File	94 KB
sencha-touch-all	2/19/2013 11:50 AM	JScript Script File	875 KB
sencha-touch-all-debug	2/19/2013 11:50 AM	JScript Script File	3,406 KB
sencha-touch-debug	2/19/2013 11:50 AM	JScript Script File	496 KB
version	2/19/2013 11:50 AM	Text Document	1 KB

There are many files that are not required for the development and testing.

The `docs` folder contains the documentation for the library and is very handy when it comes to referring to the properties, configs, methods, and events supported by different classes. You may want to copy it to a different folder so that you can refer to the documentation whenever needed.

Since there are many files inside the SDK, let us clean it up a bit so that we remove the files that are not required for development. Delete the files and folders enclosed within the rectangles in the following screenshot:



Name	Date modified	Type	Size
builds	2/19/2013 11:50 AM	File folder	
cmd	2/19/2013 11:50 AM	File folder	
docs	2/19/2013 11:51 AM	File folder	
examples	2/19/2013 11:51 AM	File folder	
microloader	2/19/2013 11:51 AM	File folder	
resources	2/19/2013 11:51 AM	File folder	
src	2/19/2013 11:51 AM	File folder	
build	2/19/2013 11:50 AM	XML Document	6 KB
file-header	2/19/2013 11:50 AM	Text Document	1 KB
getting-started	2/19/2013 11:50 AM	Chrome HTML Do...	1 KB
index	2/19/2013 11:50 AM	Chrome HTML Do...	1 KB
license	2/19/2013 11:50 AM	Text Document	2 KB
release-notes	2/19/2013 11:50 AM	Chrome HTML Do...	218 KB
sencha-touch	2/19/2013 11:50 AM	JScript Script File	94 KB
sencha-touch-all	2/19/2013 11:50 AM	JScript Script File	875 KB
sencha-touch-all-debug	2/19/2013 11:50 AM	JScript Script File	3,406 KB
sencha-touch-debug	2/19/2013 11:50 AM	JScript Script File	496 KB
version	2/19/2013 11:50 AM	Text Document	1 KB

This prepares us to get started. Since Sencha Touch is a JavaScript library, you may want to configure your Eclipse installation for JavaScript development. You may install the VJET and Sencha Eclipse plugin, which comes with Sencha Complete, and configure it for Sencha Touch development. Steps to do so are detailed on the VJET website (`http://eclipse.org/vjet/`) and instructions are provided as part of the Sencha Complete package; hence, they have been excluded from this book. To learn more about Sencha Complete, visit `http://www.sencha.com/products/complete`.

Download and install Apache Ant from `http://ant.apache.org/` to a folder of your choice, say, `C:\sencha-touch-cookbook\softwares\apache-ant-1.8.4`, and add this as the `ANT_HOME` environment variable. Also, add `%ANT_HOME%\bin` to the `PATH` environment variable.

To learn about device and platform support, you may refer to `http://www.sencha.com/products/touch/features/` to see if your device and platform are listed.

Setting up the Android-based development environment

This recipe describes the detailed steps we need to follow to set up the environment for the Android-based development. The steps do not include setting up the production environment.

Getting ready

Install JDK and set the following environment variables correctly:

- `JAVA_HOME`
- `PATH`

Install Apache Ant and set the following environment variables correctly:

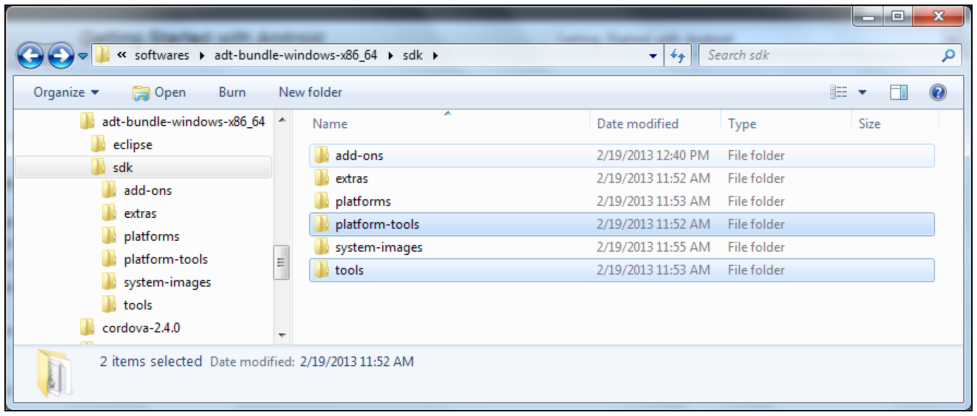
- `ANT_HOME`
- `PATH`

How to do it...

Follow these steps to set up your Android-based development environment:

1. Download **Android SDK ADT Bundle** for your platform from `http://developer.android.com/sdk/index.html` and install it inside the `softwares` folder, `C:\sencha-touch-cookbook\softwares\adt-bundle-windows-x86_64`.

2. The installed folder contains two folders, `sdk` and `eclipse`. `sdk` contains Android SDK whereas `eclipse` contains ADT-bundled Eclipse.
3. Add the following folder paths to the `PATH` environment variable:
    - `platform-tools`
    - `tools`

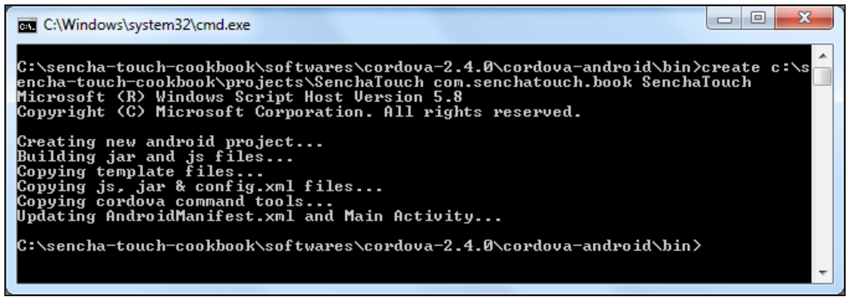

« softwares ▸ adt-bundle-windows-x86_64 ▸ sdk ▸
Search sdk
Organize ▾
Open
Burn
New folder
adt-bundle-windows-x86_64
eclipse
sdk
add-ons
extras
platforms
platform-tools
system-images
tools
cordova-2.4.0
2 items selected Date modified: 2/19/2013 11:52 AM

Name	Date modified	Type	Size
add-ons	2/19/2013 12:40 PM	File folder	
extras	2/19/2013 11:52 AM	File folder	
platforms	2/19/2013 11:53 AM	File folder	
platform-tools	2/19/2013 11:52 AM	File folder	
system-images	2/19/2013 11:55 AM	File folder	
tools	2/19/2013 11:53 AM	File folder	

4. Download **Apache Cordova** from `http://cordova.apache.org/` and extract it to a folder or your choice, say, `C:\sencha-touch-cookbook\softwares\cordova-2.4.0`. We will refer to this as `CORDOVA_HOME`.
5. Go to the `CORDOVA_HOME` folder and extract the following in it:
    - `cordova-android`: This is an Android application library that allows for Cordova-based projects to be built for the Android platform
    - `cordova-cli`: This is the command-line tool to build, deploy, and manage Cordova-based applications
    - `cordova-js`: This contains a unified JavaScript layer for Cordova-based projects
6. Launch the command prompt and go to `<CORDOVA_HOME>\cordova-android\bin`.
7. Run the following command to create our project:

```
create c:\sencha-touch-cookbook\projects\SenchaTouch com.
senchatouch.book SenchaTouch
```

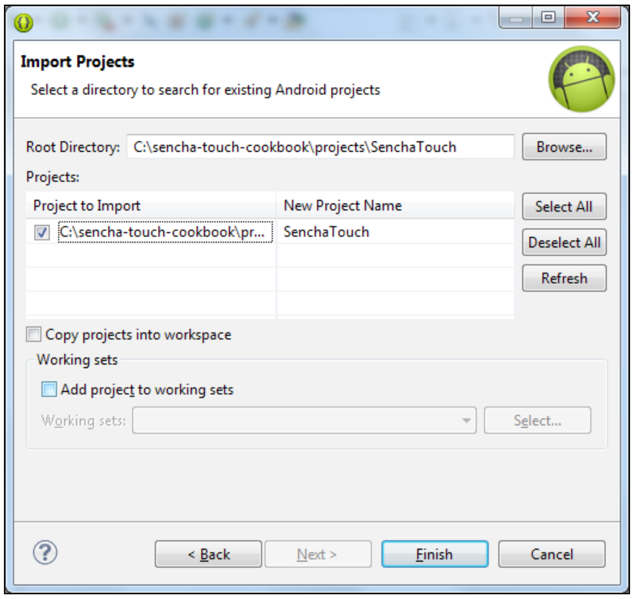
You shall see the following messages, which shows that everything went well:



```
C:\sencha-touch-cookbook\softwares\cordova-2.4.0\cordova-android\bin>create c:\s
encha-touch-cookbook\projects\SenchaTouch com.senchatouch.book SenchaTouch
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

Creating new android project...
Building jar and js files...
Copying template files...
Copying js, jar & config.xml files...
Copying cordova command tools...
Updating AndroidManifest.xml and Main Activity...

C:\sencha-touch-cookbook\softwares\cordova-2.4.0\cordova-android\bin>
```

The syntax of the command `create <project folder> <default package> <project name>`.

8. This will create a `SenchaTouch` folder inside the `projects` folder.
9. Go to the `eclipse` folder inside the ADT installed folder.
10. Launch Eclipse and use `C:\sencha-touch-cookbook\workspace` as the workspace folder.
11. Click on the **File** menu and select **Import**. Select **Existing Android Code Into Workspace** under the **Android** section.
12. Click on the **Next** button; this will take you to the **Import Projects** window. Click on the **Browse...** button, next to **Root Directory**, and select the `SenchaTouch` project that we created in the previous step.


Import Projects

Select a directory to search for existing Android projects

Root Directory: C:\sencha-touch-cookbook\projects\SenchaTouch

Projects:

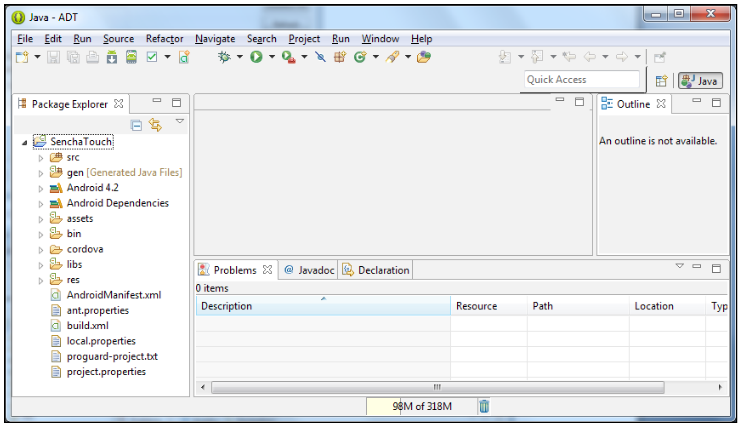
Project to Import	New Project Name
☑ C:\sencha-touch-cookbook\pr...	SenchaTouch

☐ Copy projects into workspace

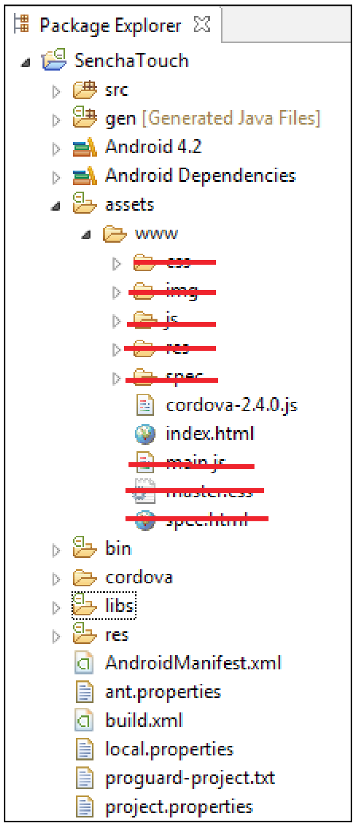
Working sets

☐ Add project to working sets

13. Click on the **Finish** button to import the project.



14. Expand the `assets\www` folder and delete the files as shown in the following screenshot:



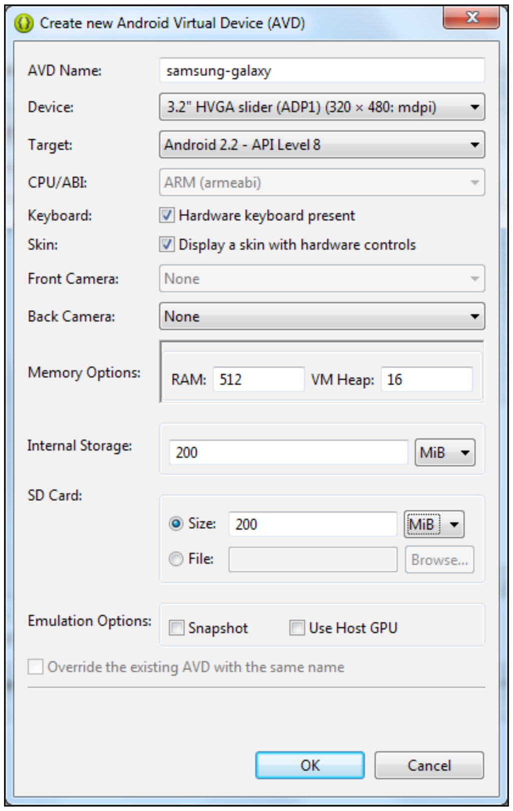
15. Copy the `www` directory and located at `C:\sencha-touch-cookbook\softwares\touch-2.2.1` folder to the `assets` directory and rename it to `touch`.
16. Create and open a new file named `ch01_01.js` file in the `assets/www/ch01` directory. Paste the following code in it:

```
Ext.application({
    name: 'MyApp',

    launch: function() {
    Ext.Msg.alert("INFO", "Welcome to the world of Sencha
Touch!");
    }
});
```

17. Now create and open a new file named `index.html` in the `assets\www` directory. Paste the following code in it:

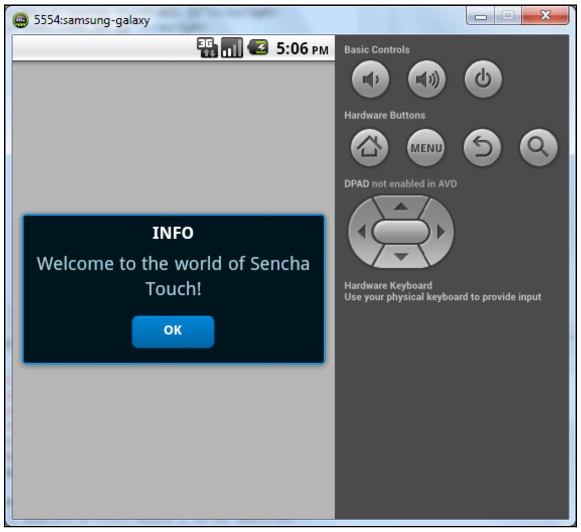
```
<!DOCTYPE HTML>
<html>
<head>
<title>Sencha Touch Cookbook - Sample</title>
<link rel="stylesheet" href="touch/resources/css/sencha-touch.css"
type="text/css">
<script type="text/javascript" charset="utf-8" src="cordova-
2.4.0.js"></script>
<script type="text/javascript" charset="utf-8" src="touch/sencha-
touch-all-debug.js"></script>
<script type="text/javascript" charset="utf-8" src="ch01/ch01_01.
js"></script>
</head>
<body></body>
</html>
```

18. Deploy the project to the simulator:
    1. Right-click the project, go to **Run As**, and click on **Android Application**.
    2. Eclipse will ask you to select an appropriate AVD. If there isn't one, you'll need to create it. To create an AVD, follow these steps:
        1. In Eclipse, go to **Window** | **Android Virtual Device Manager**.
        2. Select the **Android Virtual Devices** tab and click on the **New...** button.
        3. Enter your virtual device detail. For example, the following screenshot shows the virtual device detail for the Samsung Galaxy Ace running Android 2.2:


Create new Android Virtual Device (AVD)

AVD Name: samsung-galaxy
Device: 3.2" HVGA slider (ADP1) (320 × 480: mdpi)
Target: Android 2.2 - API Level 8
CPU/ABI: ARM (armeabi)
Keyboard: [✓] Hardware keyboard present
Skin: [✓] Display a skin with hardware controls
Front Camera: None
Back Camera: None
Memory Options: RAM: 512 VM Heap: 16
Internal Storage: 200 MiB
SD Card: [✓ Size: 200 MiB] File: Browse...
Emulation Options: [ ] Snapshot [ ] Use Host GPU
[ ] Override the existing AVD with the same name

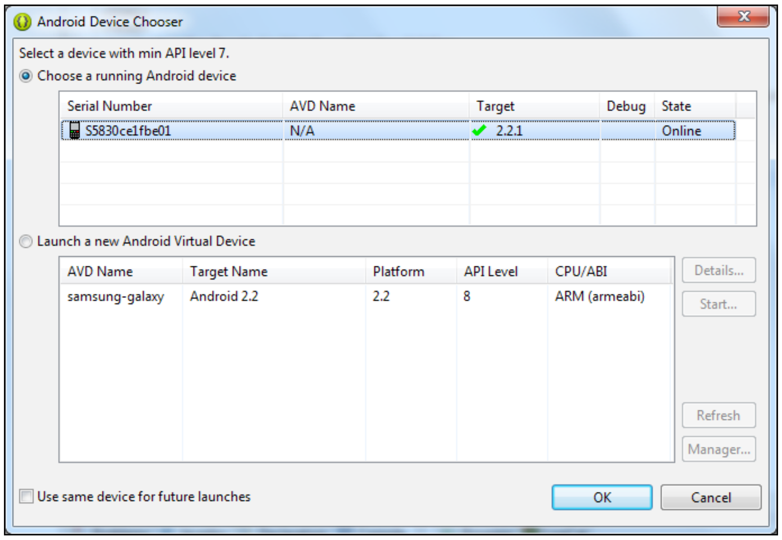
4. Click on the **OK** button.


5554:samsung-galaxy

INFO
Welcome to the world of Sencha Touch!

19. Deploy the project to the device:

    1. Make sure USB debugging is enabled on your device and plug it into your system. You may enable it by going to **Settings** | **Applications** | **Development**.
    2. Right-click on the project, go to **Run As**, and click **Android Application**. This will launch the **Android Device Chooser** window.
    3. Select the device and click on the **OK** button.



Select a device with min API level 7.

[✓ Choose a running Android device]

Serial Number	AVD Name	Target	Debug	State
S5830ce1fbe01	N/A	✓ 2.2.1		Online

Launch a new Android Virtual Device

AVD Name	Target Name	Platform	API Level	CPU/ABI
samsung-galaxy	Android 2.2	2.2	8	ARM (armeabi)

☐ Use same device for future launches

With these steps, now you will be able to develop and test your application.

How it works...

From steps 1 to 5, we downloaded and installed Android SDK, the ADT-bundled Eclipse plugin, and Apache Cordova, which are required for the development of the Android-based application. The SDK contains the Android platform-specific files, an Android emulator, and various other tools required for the packaging, deployment, and running of Android-based applications. The ADT plugin for Eclipse allows us to create Android-based applications and to build, test, and deploy them using Eclipse.

In steps 6 and 7, we created an Android project using Apache Cordova's command-line utility. It creates a project that is completely compatible with the ADT plugin.

From steps 9 to 13, we imported the Cordova-created project into the ADT-bundled Eclipse.

In steps 14 and 15, we removed the unwanted files and folders and copied Sencha Touch SDK to the `www` folder of the project. This is required to run the touch-based applications. We kept the Cordova JavaScript file, which contains the implementation of the Cordova APIs. You shall do this if you intend to use the Cordova APIs in your application (for example, to get the contacts list in your application). For this book, this is an optional step; however, interested readers may find details about the API at `http://docs.phonegap.com/en/2.4.0rc1/index.html`.

In step 16, we created the `ch01_01.js` JavaScript file that contains the entry point for our Sencha Touch application. The `Ext.application` class registers the `launch` function as the entry point to the application, which the framework calls during the application startup.

In step 17, we modified the `index.html` file to include the Sencha-Touch-related JavaScript (`sencha-touch-all-debug.js`), CSS files (`sencha-touch.css`) and our application-specific JavaScript files (`ch01_01.js`). `sencha-touch-all-debug.js` is very useful during development as it contains well-formatted code, which can be used to analyze application errors during development. You also need to include the Cordova JS file in case you intend to use its APIs in your application. The Cordova-related JS and CSS files need to be included first, then come Sencha-Touch-related JS and CSS files, and finally the application-specific JS and CSS files.

In step 18, we created an Android virtual device (an emulator) and deployed and tested the application on it.

Finally, in step 19, we deployed the application on a real Android 2.2-compatible device.

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.

Setting up the iOS-based development environment

This recipe outlines the steps to set up the development environment for the iOS-based (for example, iPhone, iPad, and iPod) development.

Getting ready

Install JDK and set the following environment variables correctly:

- `JAVA_HOME`
- `PATH`

Install Apache Ant and set the following environment variables correctly:

- `ANT_HOME`
- `PATH`

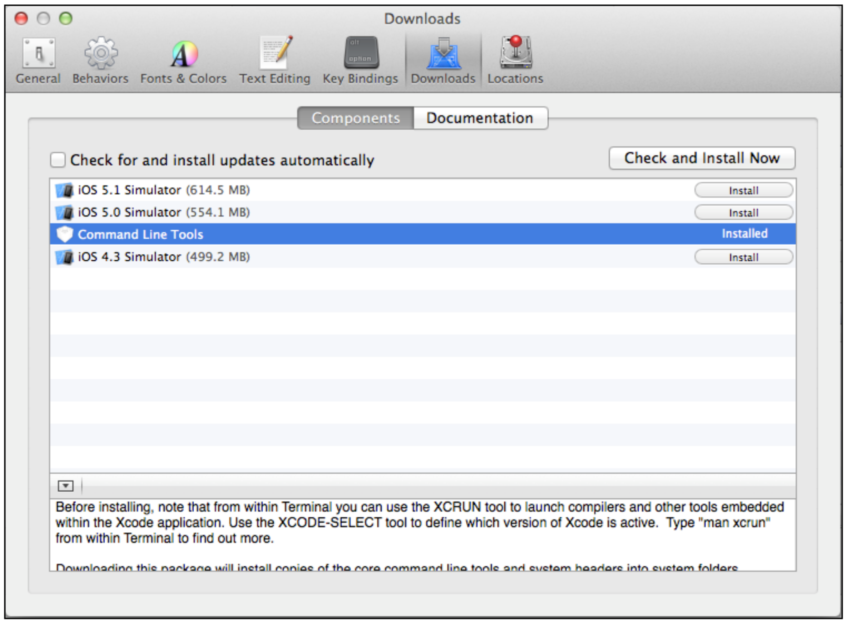
You should have created the `ch01_01.js` file as mentioned in the previous recipe.

Download and install Xcode from Apple Developer portal at `http://developer.apple.com`. This requires you to have membership of the iOS and Mac developer programs.

How to do it...

Follow these steps to set up your iOS-based development environment:

1. Launch Xcode and go to **Preferences**. Click on the **Downloads** tab on the top and install **Command Line Tools**, as shown in the following screenshot:



Check for and install updates automatically

iOS 5.1 Simulator (614.5 MB)	Install
iOS 5.0 Simulator (554.1 MB)	Install
Command Line Tools	Installed
iOS 4.3 Simulator (499.2 MB)	Install

Before installing, note that from within Terminal you can use the XCRUN tool to launch compilers and other tools embedded within the Xcode application. Use the XCODE-SELECT tool to define which version of Xcode is active. Type "man xcrun" from within Terminal to find out more.

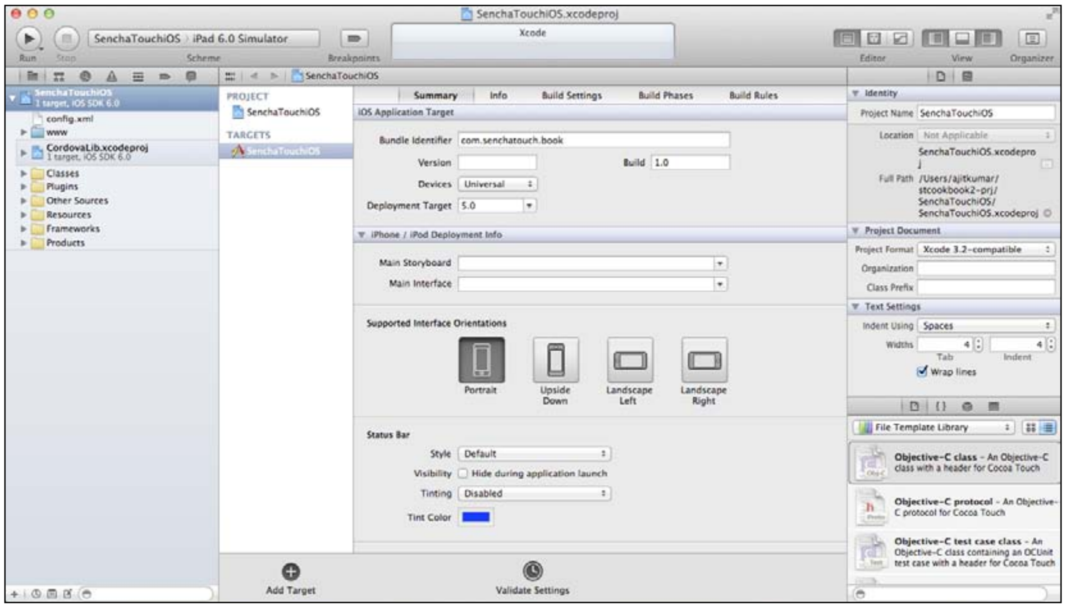
Downloading this package will install copies of the core command line tools and system headers into system folders

2. Exit Xcode.

3. Download Apache Cordova from `http://cordova.apache.org/` and extract it to a folder or your choice, say, `C:\sencha-touch-cookbook\softwares\cordova-2.4.0`. We will refer to this as `CORDOVA_HOME`.
4. Go to the `CORDOVA_HOME` folder and extract the following in it:
   - `cordova-ios`: This is an iOS application library that allows for Cordova-based projects to be built for the iOS platform
   - `cordova-cli`: This is the command-line tool to build, deploy, and manage Cordova-based applications
   - `cordova-js`: This contains a unified JavaScript layer for Cordova-based projects
5. Launch the **Terminal** window and go to `<CORDOVA_HOME>\cordova-ios\bin`.
6. Run the following command to create your project:

   ```
   create ~\sencha-touch-cookbook\projects\SenchaTouchiOS com.
   senchatouch.book SenchaTouchiOS
   ```

   This will create a project inside the `~\sencha-touch-cookbook\projects\SenchaTouchiOS` folder.
7. Use **Finder** and get to the `SenchaTouchiOS` folder created in the previous step.
8. Double-click on `SenchaTouchiOs.xcodeproj` to open the project in Xcode, as shown in the following screenshot:



iOS Application Target

Bundle Identifier: com.senchatouch.book
Version:
Build: 1.0
Devices: Universal
Deployment Target: 5.0

iPhone / iPod Deployment Info

Main Storyboard:
Main Interface:

Supported Interface Orientations: Portrait, Upside Down, Landscape Left, Landscape Right

Status Bar

Style: Default
Visibility: ☐ Hide during application launch
Tinting: Disabled
Tint Color: 🟦

Identity

Project Name: SenchaTouchiOS
Location: Not Applicable
SenchaTouchiOS.xcodeproj
Full Path: /Users/ajitkumar/stcookbook2-prj/SenchaTouchiOS/SenchaTouchiOS.xcodeproj

Project Document

Project Format: Xcode 3.2-compatible
Organization:
Class Prefix:

Text Settings

Indent Using: Spaces
Widths: 4 (Tab) 4 (Indent)
✓ Wrap lines

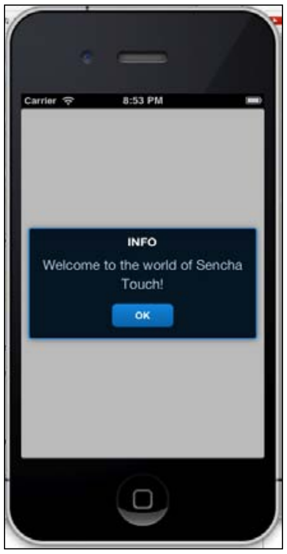
In **Finder**, you should see the `www` directory inside your project.

9. Copy the `touch-2.2.1` folder inside `www` and rename it to `touch`.
10. Add the `ch01` folder to `www` and copy the `ch01_01.js` file in it, which was created in the previous recipe.
11. Open the folder named `www` and paste the following code in the `index.html` file:

```
<!DOCTYPE HTML>
<html>
<head>
<title>Sencha Touch Cookbook - Sample</title>
<link rel="stylesheet" href="touch/resources/css/sencha-touch.css"
type="text/css">
<script type="text/javascript" charset="utf-8" src="cordova-
2.4.0.js"></script>
<script type="text/javascript" charset="utf-8" src="touch/sencha-
touch-all-debug.js"></script>
<script type="text/javascript" charset="utf-8" src="ch01/ch01_01.
js"></script>
</head>
<body></body>
</html>
```

12. Deploy the project to simulator:

    Make sure you change the simulator on the top-left menu and hit **Run** in your project window header. This will build and run the project in the simulator, as shown in the following screenshot:

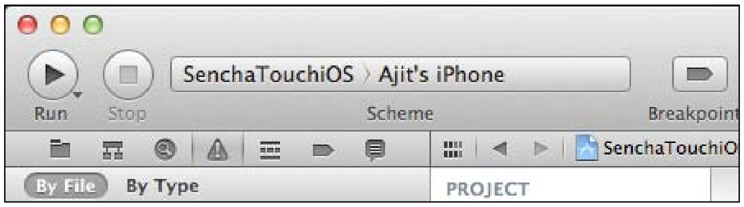


INFO

Welcome to the world of Sencha Touch!

13. Deploy the project to a device:

    1. Create a provisioning profile on `http://developer.apple.com/iphone/manage/overview/index.action`.
    2. Verify that the **Code Signing Identity** option in the **Code Signing** sections of the **SenchaTouchiOS** Xcode project settings has been set with your provisioning profile name.
    3. Connect your device using USB or Thunderbolt.
    4. Make sure to select the device on the top-left corner, as shown in the following screenshot:



    5. Hit **Run** in your project window header.

How it works...

In steps 2 and 3, we installed the required tools and libraries in Xcode. It is the IDE provided by Apple for the iOS-based application development.

From steps 3 to 6, we created a Cordova-based iOS project using Cordova's command-line utility.

From steps 7 to 10, we prepared the `www` folder for the application. Its content is described in the *Setting up the Android-based development environment* recipe.

In step 11, we included the Sencha Touch-related files and the application-specific JS file (`ch01_01.js`) in the `index.html` file.

In step 12 and 13, we deployed and tested the application in the simulator as well as a real iOS device (for example, iPhone).

See also

- The *Setting up the Android-based development environment* recipe

Setting up the BlackBerry-based development environment

So far, we have seen how to set up the environments for Android and iOS development. This recipe walks us through the steps required to set up the environment for BlackBerry-based development.

Getting ready

Install JDK and set the following environment variables correctly:

- `JAVA_HOME`
- `PATH`

Install Apache Ant and set the following environment variables correctly:

- `ANT_HOME`
- `PATH`

You should have created the `ch01_01.js` file as mentioned in the previous recipe.

How to do it...

Follow these steps to set up your BlackBerry-based development environment:

1. Download **BlackBerry WebWorks SDK** from `http://developer.blackberry.com/html5/download/` and install it to, say, `C:\sencha-touch-cookbook\softwares\RIM\BlackBerry10WebWorksSDK1.0.4.11`.
2. Download and install the BlackBerry simulator from `http://developer.blackberry.com/develop/simulator/simulator_installing.html`.
3. Go to the `CORDOVA_HOME` folder and extract the following into it:
   - `cordova-blackberry`: This is a BlackBerry application library that allows for Cordova-based projects to be built for the BlackBerry platform
   - `cordova-cli`: This is the command-line tool to build, deploy, and manage Cordova-based applications
   - `cordova-js`: This contains a unified JavaScript layer for Cordova-based projects
4. Launch the command prompt and go to `<CORDOVA_HOME>\cordova-ios\bin`.

5. Run the following command to create your project:

   ```
   create ~\sencha-touch-cookbook\projects\SenchaTouchBB com.
   senchatouch.book SenchaTouchBB
   ```

   This will create a project inside the `~\sencha-touch-cookbook\projects\SenchaTouchBB` folder.

6. Go to `c:\sencha-touch-cookbook\projects\SenchaTouchBB`, edit `project.properties` in an editor of your choice, and set the following properties:

   ```
   blackberry.bbwp.dir=C:\\sencha-touch-cookbook\\softwares\\RIM\\
   BlackBerry10WebWorksSDK1.0.4.11
   ```

7. Copy the `touch-2.2.1` folder inside `www` and rename it to `touch`.
8. Add the `ch01` folder to `www`, and copy the `ch01_01.js` file inside it, which was created in the previous recipe.
9. Open the folder named `www` and paste the following code in the `index.html` file:

   ```
   <!DOCTYPE HTML>
   <html>
   <head>
   <title>Sencha Touch Cookbook - Sample</title>
   <link rel="stylesheet" href="touch/resources/css/sencha-touch.css"
   type="text/css">
   <script type="text/javascript" charset="utf-8" src="cordova-
   2.4.0.js"></script>
   <script type="text/javascript" charset="utf-8" src="touch/sencha-
   touch-all-debug.js"></script>
   <script type="text/javascript" charset="utf-8" src="ch01/ch01_01.
   js"></script>
   </head>
   <body></body>
   </html>
   ```

10. Run the command window and go to the `SenchaTouchBB` project folder.
11. Run the following command to build the project:

    ```
    ant blackberry build
    ```

    This will create the `.bar` file for the device as well as a simulator; these are kept under the `build` folder inside the `SenchaTouchBB` folder.

12. Deploy the project to the simulator:
    1. Download and install the VMware player from `http://www.vmware.com/products/player/`.

2. Run the simulator file, based on your platform, using the VMware player. Make a note of the IP address of the simulator.
3. Go to `C:\sencha-touch-cookbook\softwares\RIM\BlackBerry10WebWorksSDK1.0.4.11\dependencies\tools\bin` and run the following command on the command prompt:

```
blackberry-deploy –installApp –device <ip address of
simulator> -package C:\sencha-touch-cookbook\projects\
SenchaTouchBB\build\simulator\SenchaTouchBB.bar
```

4. This will deploy the application on the simulator and you shall be able to launch it by tapping on the icon.

13. Deploy the project to the device:

1. You have to have your signing keys from RIM.
2. Plug in your device to the computer.
3. While in your project directory, in the command prompt, type `ant blackberry load-device`.

How it works...

In steps 1 and 2, we downloaded and installed the BlackBerry SDK and Apache Cordova, required for the BlackBerry-based development.

In steps 3 to 5, we created a Cordova-based project for BlackBerry.

In step 6, we set up the BlackBerry SDK path, which will be used by the Cordova build and deploy scripts.

From steps 7 to 9, we prepared the `www` folder by creating and copying our application-specific folders and files. Also, we modified the `index.html` file to make it look exactly like the one created in the *Setting up the Android-based development environment* recipe.

In steps 10 and 11, we built the project and created the deployable packages for the BlackBerry simulator as well as the device.

In steps 12 and 13, we deployed and tested the application in the simulator as well as a real BlackBerry device.

See also

- The *Setting up the Android-based development environment* recipe

Setting up a browser-based development environment

In the previous recipes, we saw how we can make use of Cordova to build, package, and deploy the Sencha Touch applications directly on the device. Another very popular kind of application is the browser-based one. All the devices, which Sencha Touch supports come along with Internet browsers. In this recipe, we will see how we can develop a Sencha Touch application, access it, and test it using Internet browsers.

Sencha Touch is moving towards using HTML5 features and we will require a browser that runs on WebKit engine: Opera, Safari, and Chrome or the IE 10 browser. We can also test most of the things on a browser running on your desktop/workstation (except items such as orientation change).

> Sencha Touch applications do not work on the browsers using the Gecko engine, which includes Mozilla Firefox.

We will be using this environment for this book to demonstrate the capabilities of Sencha Touch.

Getting ready

Make sure your device has a WebKit-compatible browser, such as Opera, Safari, and Chrome or IE 10.

Verify that you have your GPRS or Wi-Fi enabled and working on your device so that you are able to access the Internet.

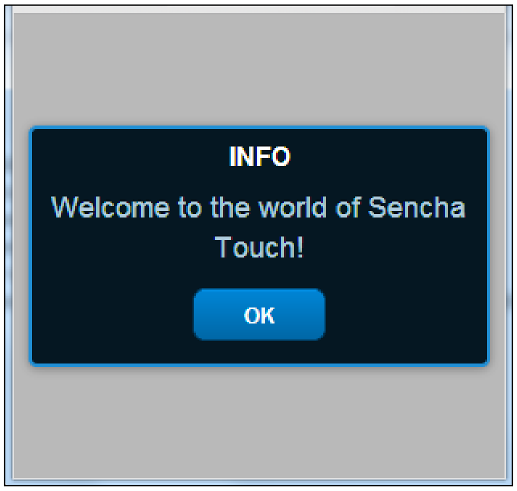
You should have a web server (such as Apache or nginx) deployed on a server, which is accessible on the Internet. For example, I have my web server running on `http://walkingtree.in`.

How to do it...

Follow these steps to set up your browser-based development environment:

1. Create a folder `touch-cookbook` in your web server's deployment/root folder, say, `public_html` or `htdocs`.
2. Copy the content of the `assets\www` folder, prepared in the *Setting up the Android-based development environment* recipe, to the `touch-cookbook` folder.

3. Go to the Internet browser on your device and enter the `http://<your domain or ip address>:<port>/touch-cookbook` URL (for example, `http://walkingtree.in/touch-cookbook`) in the address bar and hit **Go**.
   You should have the application running inside the browser, as shown in the following screenshot:



INFO

Welcome to the world of Sencha Touch!

How it works...

In step 1, we created the `touch-cookbook` folder as a placeholder to keep our application code inside it. This would help us avoid polluting the web server's root folder.

In step 2, we copied the contents from the `assets\www` folder, which we prepared in the *Setting up the Android-based development environment* recipe. In step 3, we removed the `<script>` tag including the Cordova JS file, as we are not going to use its APIs in this book.

See also

- The *Setting up the Android-based development environment* recipe

Detecting the device

Different devices offer different capabilities; hence, for an application developer, it becomes important to identify the exact device so that it can respond to the events in the most appropriate way. Sencha Touch offers the `Ext.os` class to help us detect the platform or device. This recipe describes how we can detect the device on which the application is being run using this class.

How to do it...

Follow these steps:

1. Create and open a new file `ch01_02.js` in the `ch01` folder and paste the following code into it:

```
Ext.application({
    name: 'MyApp',
    launch: function() {

  if (Ext.os.is.Android)
  Ext.Msg.alert("INFO", "Welcome Android user!");

  if (Ext.os.is.Blackberry)
  Ext.Msg.alert("INFO", "Welcome Blackberry user!");

  if (Ext.os.is.iPad)
  Ext.Msg.alert("INFO", "Welcome iPad user!");

  if (Ext.os.is.Windows) {
    var str = "Welcome Windows user!";
    if (Ext.os.deviceType === "Desktop")
      str += "Looks like you are running this sample on Desktop";

    Ext.Msg.alert("INFO", str);
  }
    }
});
```

2. Remove the following line from `index.html`:

```
<script type="text/javascript" charset="utf-8" src="ch01/ch01_01.
js"></script>
```

3. Include the following line in `index.html`:

```
<script type="text/javascript" charset="utf-8" src="ch01/ch01_02.
js"></script>
```

4. Deploy and run the application. Based on the device on which the application is being run, you will see a corresponding message.

How it works...

The Ext.os class is instrumental in detecting your target device platform on which your application is being run. It uses the JavaScript's navigator object to detect the browser details, including the platform/device. For example, if the platform property in the navigator object has iPhone in it, the target platform is iPhone; whereas, if the userAgent property in the navigator object has Android, the platform is Android.

See also

- The *Setting up a browser-based development environment* recipe

Finding features that are supported in the current environment

Each device and platform offers a rich set of functionality. However, it is difficult to identify a set of features available across all devices and platforms. And even if we happen to find out a list of common features, there may be reasons why you may want to use a feature on a device that is not present on other devices; here, you would make your application work on those devices by creating the best approximation of that specific feature. For example, on a device where SVG is supported, you may want to make use of that feature in your application to render the images so that they are scalable. However if another device does not support SVG, you may want to fall back to rendering your image into JPEG/PNG so that the image will be visible to the user. This recipe describes how an application can detect the different features that a device supports. This comes in very handy to enable/disable certain application features based on the device's supported features.

How to do it...

Follow these steps:

1. Create and open a new file ch01_03.js in the ch01 folder and paste the following code in it:

```
Ext.application({
    name: 'MyApp',

    launch: function() {

      var featuresStr = "";
      for (var key in Ext.feature.has) {
        if (Ext.feature.has.hasOwnProperty(key))
        featuresStr += key + " - " + Ext.feature.has[key] + "\n";
      }
```

```
            var browserStr = "";
            for (var key in Ext.browser.is) {
              if (Ext.browser.is.hasOwnProperty(key))
              browserStr += key + " - " + Ext.browser.is[key] + "\n";
            }

            alert(featuresStr);
            alert(browserStr);
          }
      });
```

2. Remove the following line from `index.html`:

   ```
   <script type="text/javascript" charset="utf-8" src="ch01/ch01_02.
   js"></script>
   ```

3. Include the following line in `index.html`:

   ```
   <script type="text/javascript" charset="utf-8" src="ch01/ch01_03.
   js"></script>
   ```

4. Deploy and run the application. Based on the device on which the application is being run, you will see a corresponding message.

How it works...

Check that the support for different features is encapsulated inside the Sencha Touch's `Ext.feature` class. This class applies various different mechanisms to find out whether a requested feature is supported by the target platform/device. For example, to find out whether the device supports geolocation, this class checks whether `geolocation` is present in the `window` object. Another example to find out whether SVG is supported on the target platform is that it tries to add an SVG element (removed after successful creation, setting the flag to indicate that the device supports SVG) to the document.

The `Ext.browser` class helps us detect which browser we are running on.

In the code, we are iterating through the various properties of the `Ext.feature` and `Ext.browser` class and showing the value of each property, based on the device that we are running our application on.

See also

- The *Setting up a browser-based development environment* recipe

Letting your application configure itself using profiles

This recipe describes how to set up multiple profiles for an application and let the application configure itself using the profile.

How to do it...

Follow these steps:

1. Create and open a new file `ch01_04.js` in the `ch01` folder and paste the following code in it:

```
Ext.application({
    name: 'MyApp',
   profiles: ['Android', 'Desktop', 'TabletPortrait'],
    launch: function() {
  Ext.Viewport.add(Ext.create('Ext.Panel', {
            items : [
                {
                    html: 'Welcome to My App!' + ' - profile - ' +
this.getCurrentProfile().getName()
                }
            ]
        }));
    }

});
```

2. Create an `app` folder inside the `www` folder and a `profile` folder under it.
3. Create and open a new file `Android.js` in the `profile` folder and paste the following code in it:

```
Ext.define('MyApp.profile.Android', {
    extend: 'Ext.app.Profile',

    isActive: function() {
        return Ext.os.is.Android;
    }
});
```

4. Create and open a new file `Desktop.js` in the `profile` folder and paste the following code in it:

```
Ext.define('MyApp.profile.Desktop', {
    extend: 'Ext.app.Profile',

    isActive: function() {
        return Ext.os.deviceType === 'Desktop';
    }
});
```

5. Create and open a new file `TabletPortrait.js` in the `profile` folder and paste the following code in it:

```
Ext.define('MyApp.profile.TabletPortrait', {
    extend: 'Ext.app.Profile',

    isActive: function() {
        return Ext.os.deviceType === 'Tablet' && Ext.viewport.
Default.getOrientation() === 'portrait';
    }
});
```

6. Remove the following line from `index.html`:

```
<script type="text/javascript" charset="utf-8" src="ch01/ch01_03.
js"></script>
```

7. Include the following line in `index.html`:

```
<script type="text/javascript" charset="utf-8" src="ch01/ch01_04.
js"></script>
```

8. Deploy and run the application. Based on the device profile, you shall see a message.

How it works...

The `Application` class provides a property `profiles`, which is used to set up multiple profiles, as shown in the previous code. When the application is launched, the framework calls the `isActive` method of each of the `profile` classes; whichever method returns `true`, that profile becomes active and is used in the application.

In the previous code, we defined three profile classes `Android`, `Desktop`, and `TabletPortrait`, where the `isActive` method does a profile-specific check and returns `true` or `false`.

The current profile can be fetched from the application; you can use the `getCurrentProfile()` method, which returns the instance of `Ext.app.Profile`; then we call the `getName()` method on it to print the name of the profile (for example, `Android`). The `name` property is the name of the profile class name.

See also

- The *Setting up a browser-based development environment* recipe

Responding to orientation changes

It is possible to change the orientation from portrait mode to landscape by turning your device. Many applications make use of this to provide better usability to the user. For example, when we are working with the virtual keyboard and change the orientation from portrait to landscape, the keyboard gets bigger and it becomes easier to type. Most of the devices support orientation change; based on your application, you may want to make use of this feature to change your application layout or behavior. Sencha Touch automatically watches for this and notifies all the application components by sending them the `orientationchange` event. The framework fires the `orientationchange` event on the `Ext.Viewport` class that needs to be handled in case we want to modify the behavior of our application when the orientation of the device changes.

If the application or any component of it needs to change its behavior, the corresponding component shall register a handler for the `orientationchange` event.

How to do it...

Follow these steps:

1. Create and open a new file `ch01_05.js` in the `ch01` folder and paste the following code in it:

```
Ext.application({
    name: 'MyApp',
    launch: function() {
  Ext.Viewport.add(Ext.create('Ext.Panel', {
            items : [
                {
                    html: 'Welcome to My App!'
                }
            ],
  listeners: {
    orientationchange: function(vp, newOrientation, width, height,
eOpts) {
      Ext.Msg.alert("INFO","Orientation: " + newOrientation + " : 
width:" + width + ":height:" + height);
    }
  }
        }));
    }

});
```

2. Remove the following line from `index.html`:

   ```
   <script type="text/javascript" charset="utf-8" src="ch01/ch01_04.
   js"></script>
   ```

3. Include the following line in `index.html`:

   ```
   <script type="text/javascript" charset="utf-8" src="ch01/ch01_05.
   js"></script>
   ```

4. Deploy and run the application. Based on the device orientation, you shall see a message.

How it works...

The `Viewport` class in Sencha Touch framework registers the `orientationchange` event and resizes the event handler if the target platform supports it. These handlers are invoked when the device orientation is changed. It also checks the `autoMaximize` property, which is by default set to `false`, and auto maximizes the viewport (and the application) when the orientation changes. The handlers, after resizing or maximizing the application, fire the `orientationchange` event that is available to the application developers on the `Viewport` class to handle and further modify the application behavior, if needed.

See also

- The *Setting up a browser-based development environment* recipe

2
Catering to Your Form-related Needs

In this chapter we will cover:

- Getting your form ready with form panels
- Working with the search field
- Applying custom validation in the e-mail field
- Working with dates using the date picker
- Making a field hidden
- Working with the select field
- Changing a value using slider
- Spinning the number wheel using spinner
- Toggling between your two choices
- Checkbox and checkbox groups
- Text and text area
- Grouping fields with fieldset
- Validating your form

Introduction

Most of the useful applications not only present data but also accept inputs from their users. And when we think of having a way to accept inputs from the user, send them to the server for further processing, and allow the user to modify them, we think of forms and form fields. If our application requires users to enter some information, we go about using the HTML form fields, such as `<input>` and `<select>`, and wrap them inside a `<form>` element. Sencha Touch uses these tags and provides convenient JavaScript classes to us to work with the form and its fields. It provides classes, such as `Url`, `Toggle`, `Select`, and `Text`. Each of these classes provides the properties to initialize the field and handle the events and utility methods to manipulate the behavior and the values of the field. On the other side, the form takes care of the rendering of the fields and also handles the data submission.

Each field can be created using the **JavaScript Object Notation** (**JSON** – `http://www.json.org`) or by creating an instance of the class. For example, a text field can be constructed using the following JSON notation:

```
{
xtype: 'textfield',
name: 'text',
label: 'My Text'
}
```

Alternatively, we can use the following class constructor:

```
var txtField = new Ext.form.Text({
name: 'text',
label: 'My Text'
});
```

The first approach relies on the `xtype` property, which is a type assigned to each of the Sencha Touch components. It is used as shorthand for the component class. The basic difference between the two is that the `xtype` approach is used more for lazy initialization and rendering. The object gets created only when it is required. In any application, we would use a combination of these two approaches.

In this chapter, we will go through all the form fields and understand how to make use of them and learn about their specific behaviors. Also, we will see how to create a form using one or more form fields, and handle form validation and submission.

Getting your form ready with form panels

This recipe shows how to create a basic form using Sencha Touch and implement some of the behaviors such as how to submit the form data and how to handle the errors during the submission.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

How to do it...

Carry out the following steps to create a form panel:

1. Create a `ch02` folder in the same folder where we had created the `ch01` folder.
2. Create and open a new file `ch02_01.js` and paste the following code into it:

```
Ext.application({
    name: 'MyApp',

    requires: ['Ext.MessageBox'],

    launch: function() {
var form;

//form and related fields config
var formBase = {

//enable vertical scrolling in case the form exceeds the page
height

scrollable: 'vertical',
standardSubmit: false,
submitOnAction: true,
url: 'http://localhost/test.php',
items: [{//add a fieldset
xtype: 'fieldset',
title: 'Personal Info',
instructions: 'Please enter the information above.',

//apply the common settings to all the child items of the fieldset
defaults: {
```

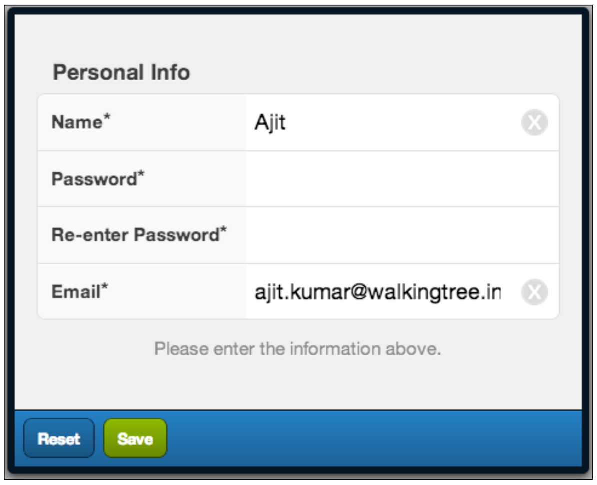
```
required: true,//required field
labelAlign: 'left',
labelWidth: '40%'
},

items: [
{//add a text feild
xtype: 'textfield',
name : 'name',
label: 'Name',
clearIcon: true,//shows the clear icon in the field when user
types
autoCapitalize : true
},

{ //add a password field
 xtype: 'passwordfield',
 name : 'password',
 label: 'Password',
     clearIcon: false
  }, {
     xtype: 'passwordfield',
     name : 'reenter',
     label: 'Re-enter Password',
     clearIcon: true
  }, { //add an email field
    xtype: 'emailfield',
    name : 'email',
    label: 'Email',
    placeHolder: 'you@sencha.com',
    clearIcon: true
    }]
}, {

//items docked to the bottom of the form
  xtype: 'toolbar',
  docked: 'bottom',
  items: [
        {
          text: 'Reset',
          handler: function() {
     form.reset();  //reset the fields
```

```
          }
         },
          {
          text: 'Save',
          ui: 'confirm',
          handler: function() {

        //sumbit the form data to the url
        form.submit({
       success: function(form, result) {Ext.Msg.alert("INFO", "Form
       submitted!");},
       failure: function(form, result) {Ext.Msg.alert("INFO", "Form
       submission failed!");}
         });
        }
         }
          ]
        }
        ]

            };

            if (Ext.os.is.Phone) {
            formBase.fullscreen = true;
            } else { //if desktop
            Ext.apply(formBase, {
                   modal: true,
                   centered: true,
                   hideOnMaskTap: false,
                   height: 385,
                   width: 480
                });
             }
       //create form panel
       form = Ext.create('Ext.form.Panel', formBase);

       Ext.Viewport.add(form);

           }
       });
```

3. Include the following line of code in the `index.html` file:

```
<script type="text/javascript" charset="utf-8"
src="ch02/ch02_01.js"></script>
```

4. Deploy and access it from the browser. You will see a screen as shown in the following screenshot:



Personal Info	
Name*	Ajit
Password*	
Re-enter Password*	
Email*	ajit.kumar@walkingtree.in

Please enter the information above.

How it works...

The code creates a form panel with a fieldset inside it. The fieldset has four fields specified as part of its child items. The `xtype` config mentioned for each field tells the Sencha Touch component manager which class to use to instantiate them.

`form = new Ext.form.FormPanel(formBase);` creates the form and the other field components using the config defined as part of the `formBase. The form.show();` code renders the form to the body, and that's how it will appear on the screen. `url` contains the URL where the form data will be posted upon submission. The form can be submitted in two ways:

- By hitting **Go** on the virtual keyboard, or *Enter* on a field, which ends up generating the action event
- By clicking on the **Save** button, which internally calls the `submit()` method on the form object

`form.reset()` resets the status of the form and its fields to the original state. So, if you had entered the values in the fields and clicked on the **Reset** button, all the fields would be cleared.

`form.submit()` posts the form data to the specified URL. The data is posted as an Ajax request using the `POST` method.

Use of `useClearIcon` on the field tells Sencha Touch whether it should show the clear icon in the field when the user starts entering values in it. On clicking this icon, the value in the field is cleared.

There's more...

In the preceding code, we saw how to construct a form panel, add fields to it, and handle events. Let us see what other non-trivial things we may have to do in the project and how we can achieve these using Sencha Touch.

Standard submit

This is an old and traditional way for posting form data to the server URL. If your application's need is to use the standard form submit rather than Ajax, you will have to set the `standardSubmit` property to `true` on the form panel. This is set to `false` by default. The following code snippet shows the usage of this property:

```
var formBase = {
            scroll: 'vertical',
            standardSubmit: true,
...
```

After this property is set to `true` on the form panel, `form.submit()` will load the complete page specified in the `url` property.

Submitting on field action

As we saw earlier, the form data automatically gets posted to the URL if the action event occurs (when the **Go** button or the *Enter* key is hit). In many applications, this default feature may not be desirable. To disable this feature, you will have to set `submitOnAction` to `false` on the form panel.

Post-submission handling

Say we posted our data to the URL. Now, either the call may fail or it may succeed. To handle these specific conditions and act accordingly, we will have to pass additional config options to the form's `submit()` method. The following code shows the enhanced version of the `submit` call:

```
form.submit({
        success: function(form, result) {
        Ext.Msg.alert("INFO", "Form submitted!");
        },
        failure: function(form, result) {
        Ext.Msg.alert("INFO", "Form submission failed!");
        }
});
```

In case the Ajax call (to post form data) fails, the `failure()` callback function is called and if it's successful, the `success()` callback function is called. This works only if the `standardSubmit` property is set to `false`.

Reading form data

To read the values entered into a form field, form panel provides the `getValues()` method, which returns an object with field names and their values. It is important that you set the `name` property on your form field otherwise that field value will not appear in the object returned by the `getValues()` method:

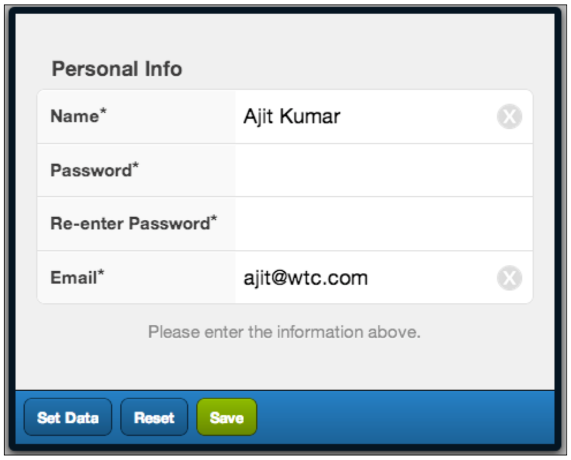
```
handler: function() {
   console.log('INFO', form.getValues());

   //sumbit the form data to the url
   form.submit({
...
...
```

Loading data in the form fields

To set the form field values, the form panel provides `record` config and two methods, `setValues()` and `setRecord()`. The `setValues()` method expects a config object with name-value pairs for the fields. The following code shows how to use the `setValues()` method:

```
{
      text: 'Set Data',
      handler: function() {
           form.setValues({
                name:'Ajit Kumar',
                email: 'ajit@wtc.com'
           });
      }
         },
         {
               text: 'Reset',

...
...
```

The preceding code adds a new button named **Set Data**; by clicking on it, the form field data is populated as shown in the following screenshot. As we had passed values for the **Name** and **Email** fields they are set:


Personal Info

Name*	Ajit Kumar
Password*	
Re-enter Password*	
Email*	ajit@wtc.com

Please enter the information above.

The other method, `setRecord()`,expects an instance of the `Ext.data.Model` class. The following code shows how we can create a model and use it to populate the form fields:

```
,
{
     text: 'Load Data',
     handler: function() {
          Ext.define('MyApp.model.User', {
               extend: 'Ext.data.Model',
               config: {
                    fields: ['name', 'email']
               }
           });
var ajit = Ext.create('MyApp.model.User', {
                     name:'Ajit Kumar',
                     email:'ajit@wtc.com'
                });
form.setRecord(ajit);
     }
},
{
     text: 'Reset',
...
...
```

We shall use `setRecord()` when our data is stored as a model, or we will construct it as a model to use the benefits of the model (for example, loading from a remote data source, data conversion, data validation, and so on) that are not available with the JSON presentation of the data.

While the methods help us to set the field values at runtime the, `record` config allows us to populate the form field values when the form panel is constructed. The following code snippet shows how we can pass a model at the time of instantiation of the form panel:

```
var ajit = Ext.create('MyApp.model.User', {
               name:'Ajit Kumar',
               email:'ajit@wtc.com'
       });
var formBase = {
      scroll: 'vertical',
      standardSubmit: true,
      record: ajit,
...
```

More about the model will be discussed in *Chapter 5*, *Dealing with Data and Data Sources*.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Working with the search field

In this and subsequent recipes in the chapter, we will go over each of the form fields and understand how to work with them. This recipe describes the steps required to create and use a search form field.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panels* recipe.

How to do it...

Carry out the following steps:

1. Copy `ch02_01.js` to `ch02_02.js`.
2. Open a new file `ch02_02.js` and replace the definition of `formBase` with the following code:

   ```
   var formBase = {
               items: [{
                       xtype: 'searchfield',
                       name: 'search',
                       label: 'Search'
                       }]
   };
   ```

3. Include `ch02_02.js` in place of `ch02_01.js` in `index.html`.
4. Deploy and access the application in the browser. You will see a form panel with a search field.

How it works...

A search field can be constructed using the `Ext.field.Search` class instance or using the `xtype: 'searchfield'` approach. A search form field implements the HTML5 `<input>` element with `type="search"`. However, the implementation is very limited. For example, the search field in HTML5 allows us to associate a data list that it can use during the search, whereas this feature is not present in Sencha Touch. Similarly, the W3 search field defines a `pattern` attribute to allow us to specify a regular expression against which a user agent is meant to check the value, which is not supported yet in Sencha Touch. For more detail, you may refer to the W3 search field (`http://www.w3.org/TR/html-markup/input.search.html`) and the source code of the `Ext.field.Search` class.

There's more...

In the application, we often do not use a label for the search fields. Rather, we would like to show text, such as **Search...**, inside the field that will disappear when the focus is on the field. Let us see how we can achieve this.

Using a placeholder

Placeholders are supported by most of the form fields in Sencha Touch using the `placeholder` property. Placeholder text appears in the field as long as there is no value entered in it and the field does not have the focus. The following code snippet shows the typical usage of it:

```
{
xtype: 'searchfield',
   name: 'search',
   label: 'Search',
   placeHolder: 'Search...'
}
```

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

Applying custom validation in the e-mail field

This recipe describes how to make use of the e-mail form field provided by Sencha Touch, and how to validate the value entered into it to find out whether the entered e-mail passes the validation rule or not.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panels* recipe.

How to do it...

Carry out the following steps:

1. Copy `ch02_01.js` to `ch02_03.js`.
2. Open a new file `ch02_03.js` and replace the definition of `formBase` with the following code:

```
var formBase = {
       items: [{
       xtype: 'emailfield',
       name : 'email',
       label: 'Email',
       placeHolder: 'you@sencha.com',
       clearIcon: true,
       listeners: {
              blur: function(thisTxt, eventObj) {
              var val = thisTxt.getValue();

//validate using the pattern
if (val.search("[a-c]+@[a-z]+[.][a-z]+") == -1)
Ext.Msg.alert("Error", "Invalid e-mail address!!");
              else
                  Ext.Msg.alert("Info", "Valid e-mail address!!");

              }
        }
}]
};
```

3. Include `ch02_03.js` in place of `ch02_02.js` in `index.html`.
4. Deploy and access the application in the browser.

How it works...

The `Email` field can be constructed using the `Ext.field.Email` class instance or using the `xtype` value as `emailfield`. The e-mail form field implements the HTML5 `<input>` element with `type="email"`. However, similar to the search field, the implementation is very limited. For example, the e-mail field in HTML5 allows us to specify a regular expression pattern, which can be used to validate the value entered in the field.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

Working with dates using the date picker

This recipe describes how to make use of the date picker form field provided by Sencha Touch, which allows the user to select a date.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panels* recipe.

How to do it...

Carry out the following steps:

1. Copy `ch02_01.js` to `ch02_04.js`.
2. Open a new file `ch02_04.js` and replace the definition of `formBase` with the following code:

```
var formBase = {
            items: [{
                    xtype: 'datepickerfield',
                    name: 'date',
                    label: 'Date'
}]
};
```

3. Include `ch02_04.js` in place of `ch02_03.js` in `index.html`.
4. Deploy and access the application in the browser.

How it works...

The date picker field can be constructed using the `Ext.field.DatePicker` class instance or using the `xtype`: `datepickerfield` approach. The date picker form field implements the HTML `<select>` element. When the user tries to select an entry, it shows the date picker component with the slots for the month, day, and year for selection. After selection, when the user clicks on the **Done** button, the field is set with the selected value.

There's more...

Additionally, there are other things that can be done, such as setting a date to the current date or a particular date, or changing the order of appearance of month, day, and year. Let us see what it takes to accomplish this.

Setting the default date to the current date

To set the default value to the current date, the `value` property must be set to the current date. The following code shows how to do it:

```
var formBase = {
            items: [{
                    xtype: 'datepickerfield',
                    name: 'date',
                    label: 'Date',
                    value: new Date(),
…
```

Setting the default date to a particular date

The default date is January 01, 1970. Let's suppose that you need to set the date to a different date but not the current date. To do so, you will have to set the `value` property using the `year`, `month`, and `day` properties, as follows:

```
var formBase = {
            items: [{
                    xtype: 'datepickerfield',
                    name: 'date',
                    label: 'Date',
                    value: {year: 2011, month: 6, day: 11},
…
```

Changing the slot order

By default, the slot order is month, day, and year. You can change it by setting the `slotOrder` property of the `picker` property of date picker, as shown in the following code:

```
var formBase = {
            items: [{
                    xtype: 'datepickerfield',
                    name: 'date',
                    label: 'Date',
                    picker: {slotOrder: ['day', 'month', 'year']}
                    }]
};
```

Setting the picker date range

By default, the date range shown by the picker is from 1970 till the current year. For our application need, if we have to alter the year range to a different range, then we can do so by setting the `yearFrom` and `yearTo` properties of the `picker` property of the date picker, as follows:

```
var formBase = {
            items: [{
                    xtype: 'datepickerfield',
                    name: 'date',
                    label: 'Date',
                    picker: {yearFrom: 2000, yearTo: 2013}
                    }]
        };
```

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

Making a field hidden

Often in an application, there would be a need to hide the fields that are not needed in a particular context but are required, and hence they need to be shown. In this recipe, we will see how to make a field hidden and show it conditionally.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panel* recipe.

How to do it...

Carry out the following steps:

1. Edit `ch02_04.js` and modify the code, as follows, by adding the `hidden` property:

   ```
   var formBase = {
               items: [{
                       xtype: 'datepickerfield',
                       id: 'datefield-id',
                       name: 'date',
                          hidden: true,
                       label: 'Date'}]
           };
   ```

2. Deploy and access the application in the browser.

How it works...

When a field is marked as hidden, Sencha Touch uses the DOM's `hide()` method on the element to hide that particular field.

There's more...

Let's see how we can programmatically show/hide a field.

Showing/hiding a field at runtime

Each component in Sencha Touch supports two methods, `show()` and `hide()`. The `show()` method shows the element and the `hide()` method hides the element. To call these methods, first we will have to find the reference to the component, which can be achieved by either using the object reference or by using the `Ext.getCmp()` method. Given a component ID, the `getCmp()` method returns us the component. The following code snippet demonstrates showing an element:

```
var cmp = Ext.getCmp('datefield-id');
cmp.show();
```

To hide an element, we will have to call `cmp.hide()`.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

Working with the select field

This recipe describes the use of the select form field, which allows the user to select a value from a list of choices, such as a combobox.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panels* recipe.

How to do it...

Carry out the following steps:

1. Copy `ch02_01.js`to `ch02_05.js`.
2. Open a new file `ch02_05.js` and replace the definition of `formBase` with the following code:

```
        var formBase = {
            items: [{
                    xtype: 'selectfield',
                    name: 'select',
                    label: 'Select',
                    placeHolder: 'Select...',
options: [
        {text: 'First Option',  value: 'first'},
        {text: 'Second Option', value: 'second'},
        {text: 'Third Option',  value: 'third'}
    ]
    }]
        };
```

3. Include `ch02_05.js` in place of `ch02_04.js` in `index.html`.
4. Deploy and access the application in the browser.

How it works...

The preceding code creates a select form field with three options for selection. The select field can be constructed using the `Ext.field.Select` class instance or using the `xtype: 'selectfield'` approach. The select form field implements the HTML `<select>` element. By default, it uses the `text` property to show the text for selection.

There's more...

It may not always be possible or desirable to use `text` and `value` properties in the date to populate the selection list. In case we have a different property in place of `text`, then how do we make sure that the selection list is populated correctly without any further conversion? Let's see how we can do this.

Using a custom display value

We shall use `displayField` to specify the field that will be used as text, as shown in the following code:

```
{
                    xtype: 'selectfield',
                    name: 'select',
                    label: 'Second Select',
                    placeHolder: 'Select...',
                    displayField: 'desc',
options: [
        {desc: 'First Option',  value: 'first'},
        {desc: 'Second Option', value: 'second'},
        {desc: 'Third Option',  value: 'third'}
    ]
}
```

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

Changing a value using slider

This recipe describes the use of the slider form field, which allows the user to change the value by mere sliding.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panels* recipe.

How to do it...

Carry out the following steps:

1. Copy `ch02_01.js` to `ch02_06.js`.
2. Open a new file `ch02_06.js` and replace the definition of `formBase` with the following code:

```
var formBase = {
            items: [{
                    xtype: 'sliderfield',
                    name : 'height',
                    label: 'Height',
                    minValue: 0,
                    maxValue: 100,
                    increment: 10
                    }]
        };
```

3. Include `ch02_06.js` in place of `ch02_05.js` in `index.html`.
4. Deploy and access the application in the browser.

How it works...

The preceding code creates a slider field with `0` to `100` as the range of values, with `10` as the `increment` value; this means that, when a user clicks on the slider, the value will change by `10` on every click. The `increment` value must be a whole number.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

Spinning the number wheel using spinner

This recipe describes the use of the spinner form field, which allows the user to change the value by clicking on the wheel.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panels* recipe.

How to do it...

Carry out the following steps:

1. Copy `ch02_01.js` to `ch02_07.js`.
2. Open a new file `ch02_07.js` and replace the definition of `formBase` with the following code:

   ```
   var formBase = {
               items: [{
                       xtype: 'spinnerfield',
                       name : 'spinner',
                       label: 'Spinner',
                       minValue: 0,
                       maxValue: 100,
                       stepValue: 10,
                       cycle: true
   }]
   };
   ```

3. Include `ch02_07.js` in place of `ch02_06.js` in `index.html`.
4. Deploy and access the application in the browser.

How it works...

Spinner is a wrapper around the HTML5 number field. The spinner field can be constructed by instantiating the `Ext.field.Spinner` class or using the `xtype` value as `spinnerfield`. `minValue` sets the initial value, which will be displayed in the field when the field is rendered. `maxValue: 100` is the maximum value that will be displayed in this field. `stepValue` tells the framework that, on every click, the value will be incremented/decremented by `10` based on the direction in which the user is moving.

There's more...

In the spinner, it may be a more sensible thing to be able to recycle the value. The following section shows how to do this.

Recycling the values

By default, when the user reaches `maxValue` or `minValue`, he/she cannot move further. In this case, we may want to recycle the values. To do this, the `Spinner` class provides a `cycle` property; setting its value to `true` will ensure that the value is set to `minValue` when the user clicks after the field value has reached `maxValue` and vice versa. The following code snippet shows how to set this property:

```
items: [{
                    xtype: 'spinnerfield',
                    name : 'spinner',
                    label: 'Spinner',
                    minValue: 0,
                    maxValue: 100,
                    stepValue: 10,
                    cycle: true
                    }]
```

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

Toggling between your two choices

This is a specialized slider with only two values. In this recipe we will see how to make use of the toggle field.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panels* recipe.

How to do it...

Carry out the following steps:

1. Copy `ch02_01.js` to `ch02_08.js`.
2. Open a new file `ch02_08.js` and replace the definition of `formBase` with the following code:

```
var formBase = {
            items: [{
                    xtype: 'togglefield',
                    name : 'toggle',
                    label: 'Toggle'
                    }]
};
```

3. Include `ch02_08.js` in place of `ch02_07.js` in `index.html`.
4. Deploy and access the application in the browser.

How it works...

This creates a slider field with `minValue` set to `0` and `maxValue` set to `1`.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

Checkbox and checkbox groups

Checkboxes permit the user to make multiple selections from a number of available options. It is a convenient way to learn about user choices. For example, in an application we may have a checkbox asking the user if he/she stayed in Hyderabad. And if we are capturing details about multiple cities where the user had stayed, then we would group multiple checkboxes under one name and use them as a checkbox group. In this recipe, we will see how we can create a checkbox and a checkbox group using Sencha Touch, and how to handle the values when you want to set them or when the form data is posted.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panels* recipe.

How to do it...

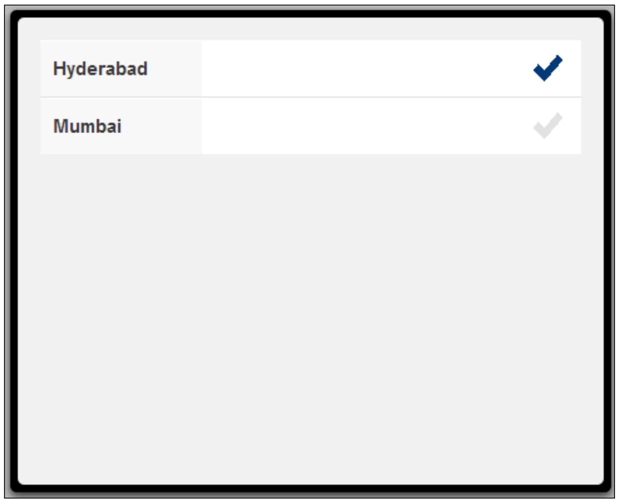
Carry out the following steps:

1. Copy `ch02_01.js` to `ch02_09.js`.

2. Open a new file `ch02_09.js` and replace the definition of `formBase` with the following code:

   ```
   var formBase = {
               items: [{
                       xtype: 'checkboxfield',
                       name: 'city',
                       value: 'Hyderabad',
                       label: 'Hyderabad',
                       checked: true
                       }, {
                       xtype: 'checkboxfield',
                       name: 'city',
                       value: 'Mumbai',
                       label: 'Mumbai'
   }]
   };
   ```

3. Include `ch02_09.js` in place of `ch02_08.js` in `index.html`.
4. Deploy and access the application in the browser. You shall see the checkboxes as shown in the following screenshot:



How it works...

The preceding code creates two checkboxes inside the form panel. `checked:true` checks the checkbox when it is rendered. When a form is submitted, the checkbox values are returned as an array. For example, given the previous code, when the user clicks on the **Submit** button, `city` would have two values, as follows:

```
city: ['Hyderabad', 'Mumbai']
```

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

Text and text area

A text field is one of the initial fields that allows the user to enter data in a form. Text area allows entering multiple lines of text. In this recipe we will make use of the text and text-area-related classes.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panels* recipe.

How to do it...

Carry out the following steps:

1. Copy `ch02_01.js` to `ch02_10.js`.
2. Open a new file `ch02_10.js` and replace the definition of `formBase` with the following code:

```
var formBase = {
            items: [
 {
                    xtype: 'textfield',
                    name : 'firstName',
                    label: 'First Name'
                },
                {
                    xtype: 'textfield',
                    name : 'lastName',
                    label: 'Last Name'
                },
                {
                    xtype: 'textareafield',
                    name : 'detail',
                    label: 'Detail'
                }]
};
```

3. Include `ch02_10.js` in place of `ch02_09.js` in `index.html`.
4. Deploy and access the application in the browser.

How it works...

In the preceding code, we created two text fields and a text area. A text field can be constructed using the `Ext.field.Text` class instance or using the `xtype` value as `textfield`. Similarly, a text area can be constructed using the `Ext.field.TextArea` class instance or using `xtype: textareafield`. Internally, the text form field implements the HTML `<input>` element with `type="text"` whereas text area implements the HTML `<textarea>` element. There is no validation on these fields; hence, the user is allowed, by default, to enter any kind of value.

There's more...

By default, a text field or a text area allows entering any number of characters. However, in some specific scenario, we may have to limit this to a particular value in our application. Let us see how we can limit this.

Limiting the number of input characters

Both text fields and text areas support a property named `maxLength` that controls how many characters the user can enter. If this property is set to `20`, the user can only enter 20 characters. The following code snippet shows how to do this:

```
{
xtype: 'textfield',
name : 'firstName',
maxLength: 20,
label: 'First Name'
},
{
xtype: 'textareafield',
name : 'detail',
maxLength: 80,
label: 'Detail'
}
```

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

Grouping fields with fieldset

Fieldset is used to logically group together elements in a form, an example of which we saw in the first recipe of this chapter. This recipe shows what Sencha Touch class can be used to create and how it groups the items under a fieldset.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panels* recipe.

How to do it...

Carry out the following steps:

1. Copy `ch02_01.js` to `ch02_11.js`.
2. Open a new file `ch02_11.js` and replace the definition of `formBase` with the following code:

```
var formBase = {
            items: [
        {
            xtype: 'fieldset',
            title: 'About Me',
            items: [
                {
                    xtype: 'textfield',
                    name : 'firstName',
                    label: 'First Name'
                },
                {
                    xtype: 'textfield',
                    name : 'lastName',
                    label: 'Last Name'
                }
            ]
        }
    ]
```

3. Include `ch02_12.js` in place of `ch02_10.js` in `index.html`.
4. Deploy and access the application in the browser.

How it works...

Fieldsets can be constructed using the `Ext.field.FieldSet` class instance or the `xtype` value as `fieldset`. All the elements, which must be grouped under the fieldset, must be added to the fieldset as its child `items`. The `FieldSet` class implements the HTML `<fieldset>` tag and uses the `legend` element to show the title.

There's more...

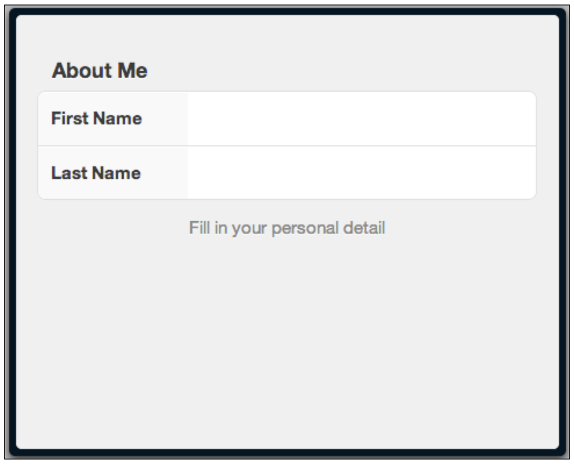
Say, when you are grouping the elements under the fieldset, you also want a way to add some instructions for it to give more information to the user. The `FieldSet` class supports this and lets us see how to do it.

Adding instructions

The `Ext.field.FieldSet` class provides a property named `instructions`, which we can use to add additional instructions. The following code snippet shows how to set this property:

```
xtype: 'fieldset',
title: 'About Me',
instructions: 'Fill in your personal detail',
…
```

The specified instruction gets added to the bottom of the fieldset, as shown in the following screenshot:



See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

Validating your form

So far, we have looked at how to create a form and make use of the different form fields offered by Sencha Touch. Different form fields provide different kinds of information a user can enter. Now, some of them may be valid and some may not be. It is a common practice to validate the form and the entered values at the time of posting. Now, based on your application architecture, you may choose to apply all kinds of validations in the frontend UI or you may choose to handle them in the backend server code, or a combination of the two. All of them are valid approaches. However, for this chapter we would assume that we want to validate the form on the frontend to make sure that the values entered are valid.

Sencha Touch does not offer a mechanism to do form validation. As of now, it has no direct support for validating the inputs. If we intend to validate the form, the code has to be written to do so. There are various approaches to building the form validation capability, depending upon what level of abstraction and reusability we want to achieve. One can write specific code in each form to carry out the validation, or one can enhance the `Ext.Component` class, which is the base class for all the Sencha Touch components, or the `Ext.field.Field` classes to handle the validation in a more generic way. Alternatively, one can enhance the form panel as well to implement a nicely encapsulated form and field validation functionality. In this recipe, we will see how we can write the specific validation code to take care of our need. The author hopes that there will be a more streamlined validation in a future version of Sencha Touch.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have followed the *Getting your form ready with form panels* recipe.

How to do it...

Carry out the following steps:

1. Copy `ch02_01.js` to `ch02_12.js` and modify the handler function with the following code:

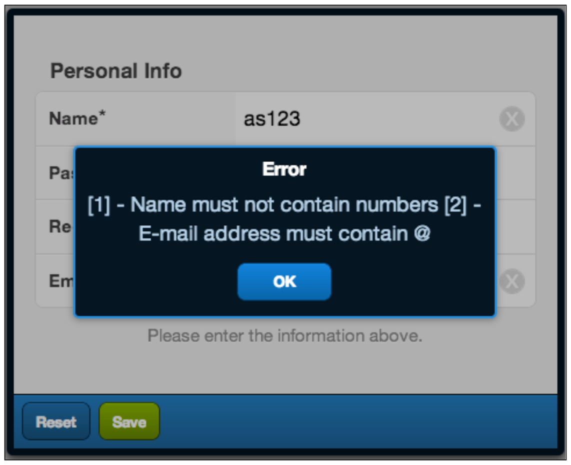
```
handler: function() {
var isValid = true;
var errors = new Array();

var fieldValMap = form.getValues();
var email = fieldValMap['email'];
var name = fieldValMap['name'];

//validate the name
if (name.search(/[0-9]/) > -1) {
isValid = false;
errors.push({field : 'name',
reason : 'Name must not contain numbers'});
}

//validate e-mail
if (email.search("@") == -1) {
isValid = false;
errors.push({field : 'email',
reason : 'E-mail address must contain @'});
}

//show error if the validation failed
if (!isValid) {
var errStr = "";

Ext.each(errors, function(error, index){
errStr += "[" + (index+1) + "] - " + error.reason + "\n";
});
```

```
            Ext.Msg.alert("Error", errStr);
        } else {//form is valid
        form.submit();
        }
        }
```

2. Include `ch02_13.js` in place of `ch02_12.js` in `index.html`.
3. Deploy and access the application in the browser.

How it works...

The `handler()` function gets called when the user clicks on the **Save** button. The handler validates the name and the e-mail address field values. `name.search(/[0-9]/)` checks if the name entered contains any number and `email.search("@")` verifies if the e-mail address contains `@` or not. In case of any error, we add an error object to the `errors` array with two properties, `field` and `reason`. The `field` property stores the field on which the validation had failed and the corresponding reason is stored in the `reason` property. After all the fields have been validated, we check the `isValid` flag to see if any of the field validation had failed; if so, we show up a message box with the list of errors, as shown in the following screenshot:



Error

[1] - Name must not contain numbers [2] - E-mail address must contain @

If there are no field validation errors, the form is submitted.

Another mechanism to validate form data is using a model, which we will see in *Chapter 5*, *Dealing with Data and Data Sources*.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe

3
Containers and Layouts

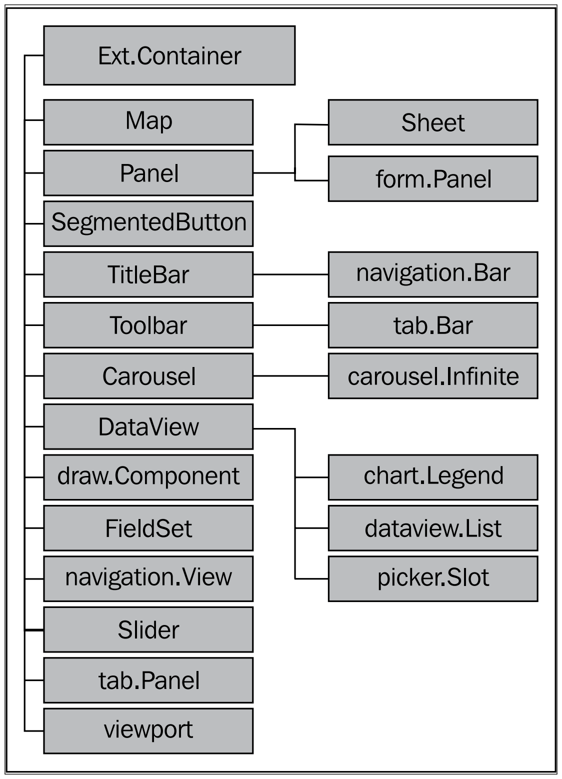
In this chapter we will cover:

- Keeping your container lightweight
- Working with Panel
- Adding items to a container at runtime
- Building wizards using CardLayout
- Docking items
- Fitting into the container using FitLayout
- Arranging your items horizontally using HBoxLayout
- Arranging your items vertically using VBoxLayout
- Mixing layouts
- Easing view navigation with the NavigationView class

Introduction

Containers in Sencha Touch are components that can contain other components as their child items. They handle the basic behavior of adding and removing items. In the previous chapter we talked about the form panel and different form fields. `FormPanel` is a container that contains form fields, buttons, toolbars, and so on. The other containers include `Panel`, `TabPanel`, `Sheet`, `NestedList`, `Carousel`, `FieldSet`, `Toolbar`, and so on. All container classes extend the `Ext.Container` class.

The following diagram depicts the various container classes and their relationship with each other:



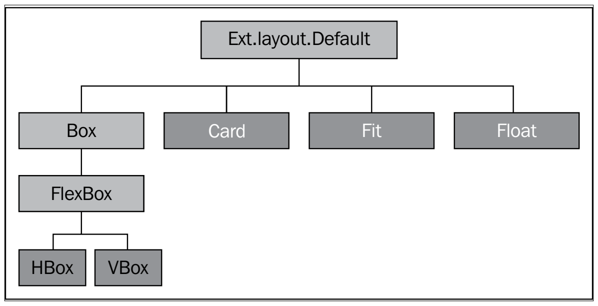
`Ext.Container` is the base class and provides the basic common functionalities related to a container; it is extended further by different classes that implement certain specific behaviors. For example, `Toolbar` takes care of showing various buttons in the form of a toolbar and `Media` takes care of playing the audio/video.

[In order to implement a new container, you may extend one of the existing specific container classes, such as `TabPanel` extending `Panel`, that is very close to your requirements. In the worst case scenario, you will have to extend the `Ext.Container` class.]

When we go on adding items (fields, panels, and so on) to a container, an obvious question that strikes our mind is, How will these items be rendered and positioned on the page? Will they be rendered one after another, vertically? Will they be rendered horizontally? Will they be resized when we resize the page? The answer to all these questions is **layout**. The layout takes care of the sizing, resizing, and positioning of the child items of a container. Every container in Sencha Touch has a config property, `layout` that accepts the name of the layout that needs to be used to calculate the sizing and positioning of the child items. The following are the predefined values and how they lay out the child items:

Layout	**Description**
`auto`	Renders one item after another
`card`	Renders each item as a card; only one item is visible at any given time
`fit`	Renders a single item and automatically expands to fill the layout's container
`hbox`	Arranges items horizontally across a container
`vbox`	Arranges items vertically down a container

The following diagram depicts the different layout-related classes and how they are related to each other:



The top-level layout, `Default`, is also referred to as the `auto` layout.

[`Ext.Container` is the default container class used by Sencha Touch if no `xtype` is specified, and the `auto` layout is used if no layout property is specified on a container class.]

We have already used some of these containers in the previous chapter, and we will use the other ones in this and subsequent chapters. In this chapter, we will look into the different containers and use layouts to position the items inside the containers.

Keeping your container lightweight

We saw earlier in this chapter that `Ext.Container` is the base class for all the containers. It gives the basic building block and the specific behaviors are implemented in the respective containers. `Ext.Panel` acts as a generic container class with the support for overlay; they can appear on top of an existing component. In case your application only needs a container in which you can add items to be rendered, you should go for `Ext.Container` rather than using `Ext.Panel`. In this recipe, we will see how to make use of `Ext.Container` to contain our item.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder named `ch03` in the `www` folder. We will be using this new folder to store the code.

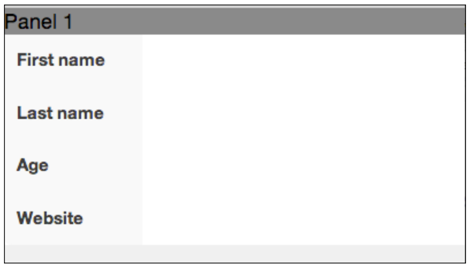
How to do it...

Carry out the following steps to keep your container lightweight:

1. Create and open a new file named `ch03_01.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var pnl = Ext.create('Ext.Container', {
      fullscreen : true,
      items : [ {
        style : 'background:grey;',
        html : '<p>Panel 1</p>'
      }, {
        xtype : 'textfield',
        name : 'first',
        label : 'First name'
       }, {
        xtype : 'textfield',
        name : 'last',
        label : 'Last name'
      }, {
        xtype : 'numberfield',
        name : 'age',
        label : 'Age'
      }, {
        xtype : 'urlfield',
        name : 'url',
        label : 'Website'
      } ]
    });
  }
});
```

2. Include the `ch03_01.js` file in the `index.html` file.
3. Deploy and access it from the browser. You will see the panel with the items as shown in the following screenshot:



How it works...

The preceding code creates a container with a panel and four form fields. The following is the code for adding the first child item to the container:

```
{
  style: 'background:grey;',
  html: '<p>Panel 1</p>'
}
```

As there is no `xtype` specified, Sencha Touch creates `Ext.Container` and sets `style` and `html` on it.

The default layout used is `auto`; hence we see the items rendered one after another.

There's more...

While we are using the `Ext.Container` class for its lightweight nature, we may need our items to be laid out differently. Let's see how we can do this.

Using layout

The `Ext.Container` class supports the `layout` property, which we can set to request the container to position and calculate sizing accordingly. The following are the layouts that can be used with `Ext.Container`:

- `auto`
- `fit`
- `card`
- `hbox`
- `vbox`
- `float`

For example, adding the following additional properties on `Ext.Container` will show the first panel on the whole screen:

```
layout: 'card',
activeItem: 0
```

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panel* recipe of *Chapter 2*, *Catering to Your Form-related Needs*

Working with Panel

The `Ext.Panel` is a specific implementation of a generic container by extending the `Ext.Container`. The main functionality offered on top of `Ext.Container` is the support for overlay, which makes it float over the application. This recipe describes how to make use of the `Ext.Panel` class to create an application.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch03` folder inside the `www` folder.

How to do it...

Carry out the following steps to work with `Panel`:

1. Create and open a new file named `ch03_02.js` and copy-paste the following code into it:

```
Ext.setup({
    onReady: function() {
  var pnl = new Ext.Panel({
    fullscreen: true,
    items: [{
        bodyStyle: 'background:grey;',
        html: '<p>Panel 1</p>'
    },{
          xtype: 'textfield',
          name : 'first',
          label: 'First name'
      },
        {
            xtype: 'textfield',
            name : 'last',
            label: 'Last name'
        },
        {
            xtype: 'numberfield',
            name : 'age',
            label: 'Age'
        },
        {
            xtype: 'urlfield',
            name : 'url',
            label: 'Website'
    }]
});
    }
});
```

2. Include the `ch03_02.js` file in the `index.html` file.
3. Deploy and access it from the browser.

How it works...

The preceding code creates a panel that contains four form fields. The default layout used is `auto`.

There's more...

Additionally, we can use a different layout and also have docking items with a panel. Let's see how we can make use of these features.

Docking items

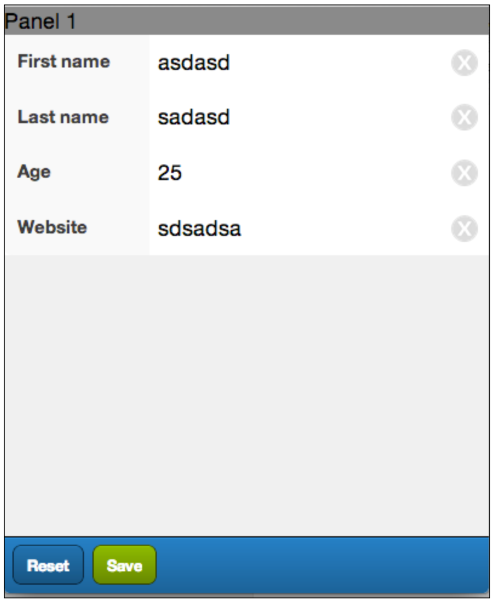
As we discussed earlier, one of the major advantages of using `Ext.Panel` over `Ext.Container` is that it can dock one or more items. This is driven by the config named `docked`. The items that need to be docked must have this property set on it. This config property tells `Ext.Panel` that these items need to be docked and it uses the `docked` value to position them and calculate their sizing.

The following code is used for adding a toolbar with two buttons, **Save** and **Reset**, as the dock items:

```
items: [
  {
    xtype: 'toolbar',
    docked: 'bottom',
      items: [
        {
          text: 'Reset',
          handler: function() {

          }
        },
        {
          text: 'Save',
          ui: 'confirm',
          handler: function() {
            Ext.Msg.alert("INFO", "In real implementation, this
              will be saved!");
```

```
                    }
                }
            ]
        },
        {
                    style : 'background:grey;',
...
... }
]
```

`docked: 'bottom'` is a dock-layout-specific property telling us that the toolbar needs to be positioned at the bottom of the panel. In the dock panel, we have added two buttons, **Reset** and **Save**. The following screenshot show how the screen will look:



Panel 1

First name	asdasd
Last name	sadasd
Age	25
Website	sdsadsa

Using layouts

Just as with `Ext.Container`, `Ext.Panel` supports the `layout` property, which can be used to set the appropriate layout. The following are the layouts that can be used with `Ext.Panel`:

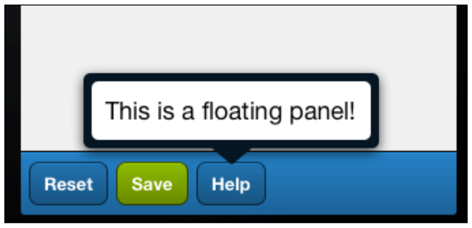
- `auto`
- `fit`
- `card`
- `hbox`
- `vbox`

Panel used as an overlay

`Panel` has a useful feature that we can use to show it as a floating panel so that it appears as an overlay panel. The following code adds one more button, **Help**; on clicking it, the handler shows the overlay panel.

```
{
  text: 'Help',
  handler: function(btn) {
    Ext.create('Ext.Panel', {
      html: 'This is a floating panel!',
      left: 0,
      padding: 10
    }).showBy(btn);
  }
}
```

The following screenshot shows how the floating panel will appear:



See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Docking items* recipe

Adding items to a container at runtime

In an application, there will be numerous scenarios where we will have to add components (ranging from a simple field to a panel) at runtime as part of the response to a user event. For example, your application may have a payment panel where you may want to show the payment-specific detail panels, which depend on the payment method. If a user selects **Credit Card** as the payment method, you may want to show a panel asking the user to enter their credit card details. This requires us to add components dynamically to an existing container. In this recipe, we will see how to work with the components at runtime.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch03` folder inside the `www` folder.

How to do it...

Carry out the following steps for adding items to a container at runtime:

1. Create and open a new file named `ch03_03.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var pnl = Ext.create('Ext.Panel', {
      fullscreen : true,
      items : [{
        xtype: 'toolbar',
        docked: 'bottom',
        items: [{
          text: 'Reset',
          handler: function() {

          }
        },
        {
          text: 'Add',
          ui: 'confirm',
          handler: function() {
            pnl.add([{
              xtype: 'emailfield',
              name : 'email',
              label: 'E-mail'
            },
            {
              xtype : 'toolbar',
              docked: 'top',
              items: [{
                text: 'Dummy'
              }]
            }]);
```

```
                }
              }]
            },
            {
              style : 'background:grey;',
              html : '<p>Panel 1</p>'
            },
            {
              xtype : 'textfield',
              name : 'first',
              label : 'First name'
            },
            {
              xtype : 'textfield',
              name : 'last',
              label : 'Last name'
            },
            {
              xtype : 'numberfield',
              name : 'age',
              label : 'Age'
            },
            {
              xtype : 'urlfield',
              name : 'url',
              label : 'Website'
             } ]
          });
        }
      });
```

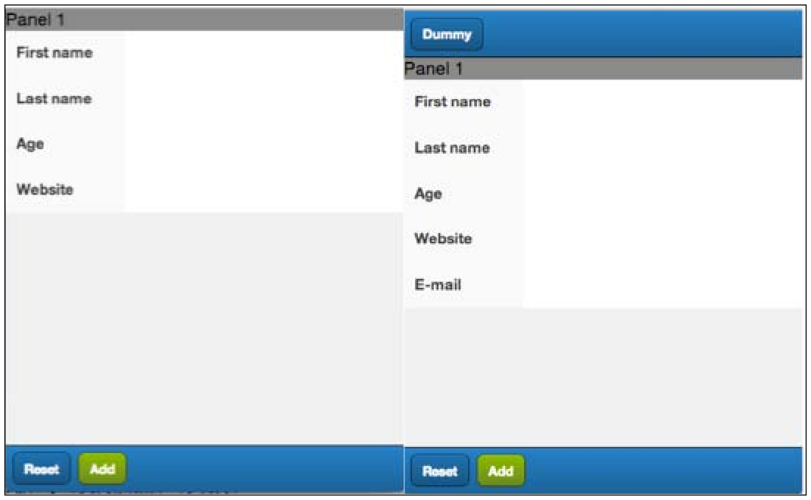
2. Include the `ch03_03.js` file in the `index.html` file.
3. Deploy and access it from the browser.

How it works...

The code creates a panel with three form fields and two buttons, **Add** and **Reset**. When the user clicks on the **Add** button, a new **E-mail** field is added to the panel and a new toolbar with a **Dummy** button is added on the top as a docked item. The following code in the **Add** button handler adds an **E-mail** field and a docked item to the panel:

```
pnl.add([{
  xtype: 'emailfield',
  id: 'email-id',
  name : 'email',
  label: 'E-mail'
  }, {
  xtype : 'toolbar',
  docked: 'top',
  items: [{
    text: 'Dummy'
  }]
}]);
```

When the items are added to the panel, they get rendered immediately on the screen and we can see the changes made to the panel. This is primarily due to the framework automatically notifying the layout manager, which renders the new items. The following screenshot shows how the screen will look before and after clicking on the **Add** button:



There's more...

Additionally, we can use a different layout and also have docking items with a panel. Let's see how we can make use of these features.

Inserting at a specific position

The `insert` method allows us to insert a child item at a desired position. For example, the following code will add the **E-mail** field before the **Website** field:

```
pnl.insert(5, [{
      xtype: 'emailfield',
      name : 'email',
      label: 'E-mail'
    },
    {
      xtype : 'toolbar',
      docked: 'top',
      items: [{
        text: 'Dummy'
      }]
    }

]);
```

Removing an item

In order to remove an item, the container provides the `remove` and `removeAll` methods to remove one or all components respectively. In order to remove a particular component, we either need its ID or its object reference. In the following code snippet, we have added an ID, `email-id`, to the **E-mail** field that we are creating; when the user clicks on the **Reset** button, we are removing it from the panel:

```
items : [{
  xtype: 'toolbar',
  docked: 'bottom',
  items: [{
    text: 'Reset',
    handler: function() {
      pnl.remove(Ext.getCmp('email-id'));
      var tb = pnl.down('toolbar[docked=top]');
      pnl.remove(tb);
    }
  },
  {
    text: 'Add',
    ui: 'confirm',
    handler: function() {
    pnl.add([{
        xtype: 'emailfield',
        id: 'email-id',
        name : 'email',
        label: 'E-mail'
      },
      {
        xtype : 'toolbar',
        docked: 'top',
        items: [{
          text: 'Dummy'
        }]
      }
    ]);
...
...
```

The `Ext.getCmp` method accepts a component ID and searches its component hierarchy to return the matching component. In the preceding code, we used the ID of the **E-mail** field, `email-id`, to get the object reference of the **E-mail** field and then used it to remove the field.

To find out the top toolbar that we want to remove from the panel, we use the following code:

```
var tb = pnl.down('toolbar[docked=top]');
```

The container's `down` method will go down searching in the component hierarchy of the panel to find out the first descendant component matching the specified selector. If a matching component is not found, the `down` API will return null. Hence, you may want to check the returned value for null using the `Ext.isEmpty` method. Also, the selector string is case-sensitive. You can read more about the selector in the `Ext.ComponentQuery` class documentation.

Hiding/showing

Sometimes, the user will be seeing a field based on some condition. Moreover, if your application was doing this repeatedly, `add` and `remove` may not be an efficient set of methods to use. Rather, we should use the `show` and `hide` methods to control the visibility of a component. The following code snippet shows how a component can be hidden and shown again:

```
items : [{
    xtype: 'toolbar',
    docked: 'bottom',
    items: [{
      text: 'Reset',
      handler: function() {
        pnl.remove(Ext.getCmp('email-id'));
        var tb = pnl.down('toolbar[docked=top]');
        tb.hide();
      }
    },
    {
      text: 'Add',
      ui: 'confirm',
      handler: function() {
        pnl.add([{
          xtype: 'emailfield',
          id: 'email-id',
          name : 'email',
          label: 'E-mail'
        },
        {
          xtype : 'toolbar',
          docked: 'top',
          items: [{
            text: 'Dummy'
          }]
        }]);
...
...
```

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Working with Panel* recipe
- The *Docking items* recipe

Building wizards using CardLayout

This recipe describes how to use a `card` layout as a container layout. `CardLayout` lays items in the form of playing cards and shows only one item at a time. We will implement a wizard application to understand the usage of this layout.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch03` folder inside the `www` folder.

How to do it...

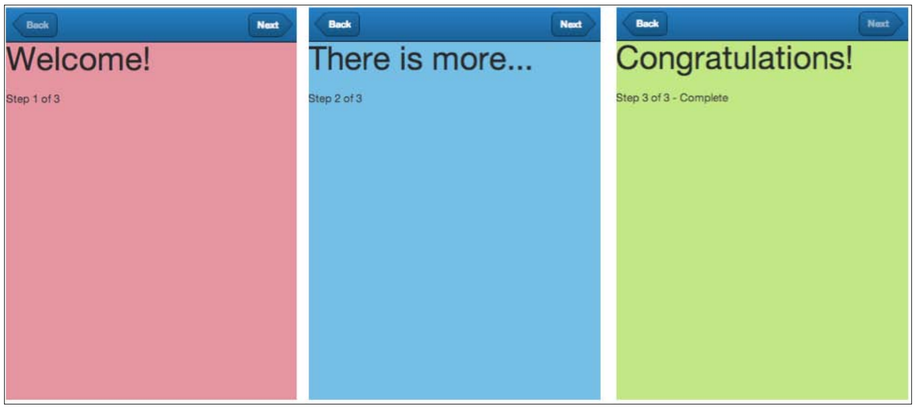
Carry out the following steps for building wizards using CardLayout:

1. Create and open a new file named `ch03_04.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    Ext.define('Wtc.tux.CardPanel', {
      extend: 'Ext.Panel',
      config: {
        layout: 'card'
      },
      getCurrentItemIdx: function() {
        var curr = this.getActiveItem();
        var items = this.getInnerItems();//exclude docked
          items
        var idx = -1;
        var l = items.length;
        for (var i=0; i<l; i++) {
          if (items[i].id == curr.id) {
            idx = i;
            break;
          }
        }
        return idx;
      },
      prev: function() {
        var idx = this.getCurrentItemIdx();
        var items = this.getInnerItems();
        var l = items.length;
        var o = {
          next:--idx<l-1,
          prev:idx>0
        };
        this.setActiveItem(items[idx]);
        return o;
      },
      next: function() {
        var idx = this.getCurrentItemIdx();
        var items = this.getInnerItems();
        var l = items.length;
        var o = {
          next:++idx<l-1,
          prev:idx>0
        };
        this.setActiveItem(items[idx]);
        return o;
      }
    });
  var navigate = function(panel, direction){
      var o = panel[direction]();
      Ext.getCmp('move-prev').setDisabled(!o.prev);
      Ext.getCmp('move-next').setDisabled(!o.next);
  };
  var pnl = Ext.create('Wtc.tux.CardPanel', {
    title: 'Wizard',
    fullscreen: true,
    styleHtmlContent: true,
    items: [
      {
        docked : 'top',
        xtype: 'toolbar',
        items: [
          {
            id: 'move-prev',
            text: 'Back',
            ui: 'back',
            handler: function(btn) {
              navigate(btn.up("panel"), "prev");
            },
            disabled: true
          },{xtype: 'spacer'},
          {
            id: 'move-next',
            text: 'Next',
            ui: 'forward',
            handler: function(btn) {
              navigate(btn.up("panel"), "next");
            }
          }
        ]
```

```
            },
            {
              id: 'card-0',
              html: '<h1>Welcome to the Wizard!</h1><p>Step 1 of
                3</p>'
            },
            {
              id: 'card-1',
              html: '<p>Step 2 of 3</p>'
            },
            {
              id: 'card-2',
              html: '<h1>Congratulations!</h1><p>Step 3 of 3 -
                Complete</p>'
            }
            ]
          });
        }
    });
```

2. Include the `ch03_04.js` file in the `index.html` file.
3. Deploy and access it from the browser. You will see the following screens on clicking on the **Next** buttons:



How it works...

The preceding code creates a panel with three child panels and a docked panel with two buttons, **Back** and **Next**. The `layout: 'card'` property indicates that `CardLayout` will be used to lay out the items.

We defined a new class, `Wtc.tux.CardPanel`, to encapsulate the functionality of moving to the next or previous item in the card by implementing the `next()` and `prev()` methods. The class extends the existing `Ext.Panel` class, indicated by `extend:'Ext.Panel'`, and defaults the layout to `card`. It borrows all the behaviors from the `Ext.Panel` class and adds the `prev()` and `next()` methods to it.

The `navigation` function, based on the specified `direction` value, moves to the next/previous item in the card and enables and disables the appropriate panel.

```
var navigate = function(panel, direction){
  var o = panel[direction]();
  Ext.getCmp('move-prev').setDisabled(!o.prev);
  Ext.getCmp('move-next').setDisabled(!o.next);
};
```

`Panel` contains the `next()` and `prev()` methods that can set the active panel based on the card stack. The `panel[direction]()` method calls `next()` or `prev()` depending on the `direction` value. These methods return an object containing two properties, `prev` and `next`, that indicate if there is any item after or before the currently set active item.

Let's take a look at the other two lines:

```
Ext.getCmp('move-prev').setDisabled(!o.prev);
Ext.getCmp('move-next').setDisabled(!o.next);
```

In these lines, we are disabling the **Back** button if we have reached the first panel; otherwise it remains enabled. Similarly, we will disable the **Next** button if we have reached the last panel.

There are different ways to access the **Back** and **Next** buttons and you may use them based on your application design. For example, you may pass the button reference to the `navigate` method or you may use the `down` API on the panel object with the appropriate selector, as described in the previous recipe.

The button handler calls the `navigate` method, where it passes the reference of the panel object, `(btn.up("panel"))`, and the direction text, `next`, using the following line:

```
navigate(btn.up("panel"), "next");
```

There's more...

By default, `CardLayout` sets the first item as the active item and the user will see that on the screen when the application comes up. However, there might be a situation where we would like a different item to remain active by default. Let's see what functionality `CardLayout` provides.

Changing the default active item

`CardLayout` provides a property named `activeItem` that can be used to set the item that will be active by default. The default value of this property is `0`. To show the second item as the default panel when the container is initialized, set `activeItem` to `1` on the container panel. The following code snippet shows the use of this property:

```
var pnl = new Ext.Panel({
    title: 'Wizard',
    fullscreen: true,
    styleHtmlContent: true,
    layout: 'card',
    activeItem: 1,
})
```

Alternatively, you may call the `setActiveItem` method on the panel object reference, which we have done inside the `prev` and `next` handlers, to set the active item at runtime.

Animating cards

When the card moves from one item to another, you can enable animation by setting the `animation` property on the `layout` config as shown in the following code snippet:

```
Ext.define('Wtc.tux.CardPanel', {
    extend: 'Ext.Panel',
    config: {
      layout: {
        type: 'card',
        animation: 'reveal'
      }
    },
    getCurrentItemIdx: function() {
...
```

The following is the list of valid values that we can pass as animation:

- `cover`
- `cube`
- `fade`
- `flip`
- `pop`
- `reveal`
- `scroll`
- `slide`

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Working with Panel* recipe
- The *Docking items* recipe

Docking items

The dock panel is used to position the child content along the edge of a layout container. Sencha Touch provides the mechanism to dock items along any of the four edges, `top`, `left`, `bottom`, or `right`. In this recipe we will see what needs to be done to use a `docked` property.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch03` folder inside the `www` folder.

How to do it...

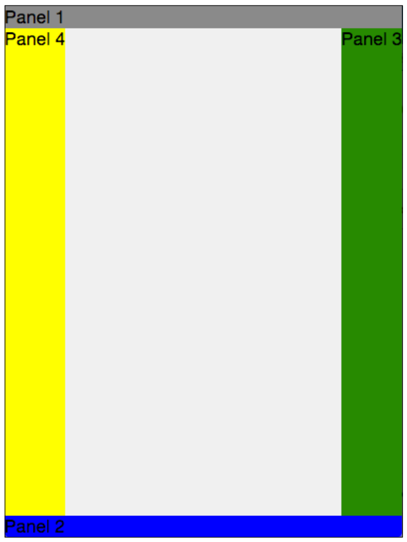
Carry out the following steps to dock items:

1. Create and open a new file named `ch03_05.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var pnl = Ext.create('Ext.Panel', {
      fullscreen: true,
      styleHtmlContent: true,
      items: [{
        docked : 'top',
        style: 'background:grey',
        html: '<p>Panel 1</p>'
      },
      {
        docked : 'bottom',
        style: 'background:blue',
        html: '<p>Panel 2</p>'
```

```
            },
            {
              docked : 'right',
              style: 'background:green',
              html: '<p>Panel 3</p>'
            },
            {
              docked : 'left',
              style: 'background:yellow',
              html: '<p>Panel 4</p>'
            }]
          });
        }
      });
```

2. Include the `ch03_05.js` file in the `index.html` file.
3. Deploy and access it from the browser. The following screenshot shows how the view will look:



How it works...

The preceding code creates a panel with four docking panels along the four different edges using the `docked` property. Also, irrespective of the value of the `layout` config, if a container has items with the `docked` property defined, they will be rendered using the position mentioned in the `docked` property.

Docked items can be used with any layout.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Working with Panel* recipe

Fitting into the container using FitLayout

The `FitLayout` class is used for the container that contains a single item that automatically expands to fill the layout's container. `CardLayout` utilizes `FitLayout` to fit an item into a card. In this recipe, we will learn about the usage of the `FitLayout` class.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch03` folder inside the `www` folder.

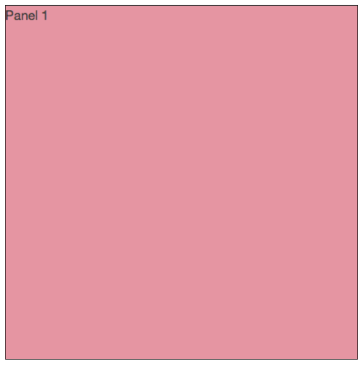
How to do it...

Carry out the following steps to fit an item into the container using `FitLayout`:

1. Create and open a new file named `ch03_06.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var pnl = Ext.create('Ext.Panel', {
      fullscreen: true,
      styleHtmlContent: true,
      layout: 'fit',
      items: [{
        style: 'background:#E58A99',
        html: '<p>Panel 1</p>'
      }]
    });
  }
});
```

2. Include the `ch03_06.js` file in the `index.html` file.
3. Deploy and access it from the browser. The following screenshot shows the view:



How it works...

The `layout:'fit'` property initializes the `FitLayout` class and associates it with the panel, which will then be used to render the child items. There is no other config specific to `FitLayout`.

> If the container with `FitLayout` has multiple panels, only the first one will be displayed.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Working with Panel* recipe

Arranging your items horizontally using HBoxLayout

The `HBoxLayout` class arranges items horizontally across a container. It optionally divides the available horizontal space between child items containing a `flex` configuration, which is a numeric. The `flex` option is a ratio that distributes the width after any items with explicit widths have been accounted for. We can either use the `width` property to specify a fixed width or use `flex`. This recipe describes how we can arrange our items using `HBoxLayout`.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1, Gear Up for the Journey*.

Make sure you have created the `ch03` folder inside the `www` folder.

How to do it...

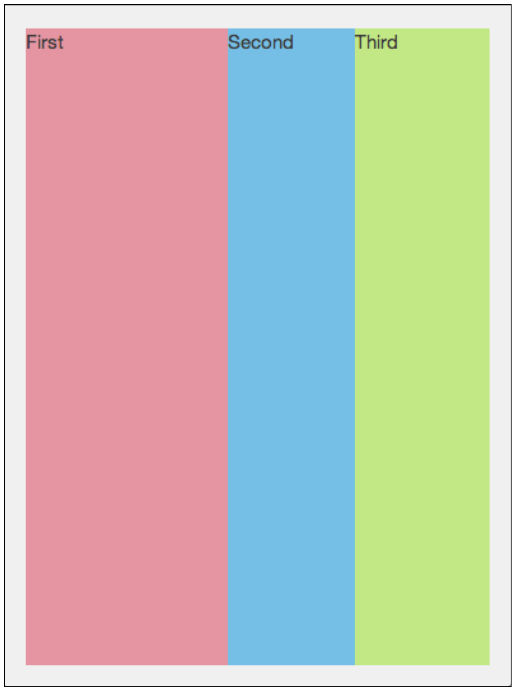
Carry out the following steps for arranging your items horizontally using `HBoxLayout`:

1. Create and open a new file named `ch03_07.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var pnl = Ext.create('Ext.Panel', {
      fullscreen: true,
      styleHtmlContent: true,
      layout: 'hbox',
      items: [{
        flex: 3,
        html: 'First',
        style: 'background:#E58A99'
      },
      {
        width: 100,
        html: 'Second',
        style: 'background:#65B9E0'
      },
      {
        flex: 2,
        html: 'Third',
        style: 'background:#B7E488'
      }]
    });
  }
});
```

2. Include the `ch03_07.js` file in the `index.html` file.

3. Deploy and access it from the browser. The following screenshot shows the view:



How it works...

The preceding code creates a panel of 400 pixels width and 300 pixels height. Also, it has three child panels, where one panel has a fixed width of 100 px and others are using `flex`. This is how the `hbox` layout will calculate the width of each item:

1. The fixed width item is subtracted, leaving us with 300 px width.
2. The total flex number is counted; in this case, it is 5.
3. The ratio is then calculated; 300/5 = 60.
4. The first item has a `flex` value of 3, so its width is set to 3*60 = 180 px.
5. The other remaining item is set to 2*60 = 120 px.

There's more...

Additionally, the `HBoxLayout` class provides options such as controlling the vertical and horizontal alignment of the item.

Aligning the component vertically

If there is no height specified for the items, you will notice that the items occupy the complete container height. In some cases, you may have the need to make the item appear in the middle of the container. To achieve this, set the `align` property to `middle`, as shown in the following code snippet:

```
layout: {
    type: 'hbox',
    align: 'middle'
}
```

Refer to the `hbox` layout documentation for other valid values.

Aligning the component horizontally

While the `align` property represents the vertical axis for `HBoxLayout` and helps us align the items with respect to it, the `pack` property represents the horizontal axis; we can align the item in the center by setting the `pack` property to `center`, as shown in the following code snippet:

```
layout: {
    type: 'hbox',
    pack: 'center'
}
```

Refer to the `hbox` layout documentation for other valid values.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1, Gear Up for the Journey*
- The *Working with Panel* recipe

Arranging your items vertically using VBoxLayout

The `VBoxLayout` class arranges items vertically down a container. It optionally divides available vertical space between child items containing a `flex` configuration, which is a numeric. The `flex` option is a ratio that distributes height after any items with explicit heights have been accounted for. We can either use the `height` property to specify a fixed height or use `flex`. This recipe describes how we can arrange our items using the `VBoxLayout` class.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch03` folder inside the `www` folder.

How to do it...

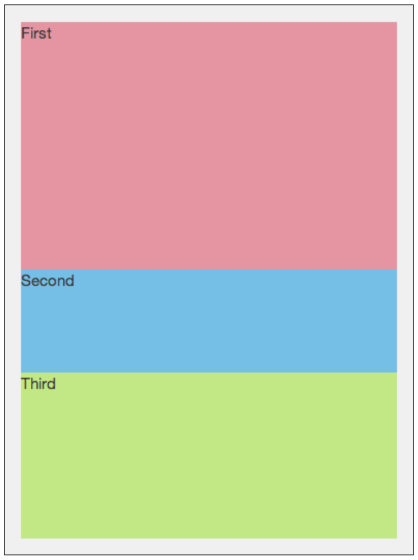
Carry out the following steps for arranging your items vertically using `VBoxLayout`:

1. Create and open a new file named `ch03_08.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var pnl = Ext.create('Ext.Panel', {
      fullscreen: true,
      styleHtmlContent: true,
      layout: {
        type: 'vbox'
      },
      items: [{
        flex: 3,
        html: 'First',
        style: 'background:#E58A99'
      },
      {
        height: 100,
        html: 'Second',
        style: 'background:#65B9E0'
      },
      {
        flex: 2,
        html: 'Third',
        style: 'background:#B7E488'
      }]
    });
  }
});
```

2. Include the `ch03_08.js` file in the `index.html` file.

3. Deploy and access it from the browser. The following screenshot shows the view:



How it works...

The preceding code creates a panel of 400 pixels width and 400 pixels height. Also, it has three child panels, where one panel has a fixed height of 100 px and the others are using `flex`. This is how the `vbox` layout will calculate the height of each item:

1. The fixed height item is subtracted, leaving us with 300 px height.
2. The total flex number is counted; in this case, it is 3.
3. The ratio is then calculated; 300/3 = 100.
4. The first item has a `flex` value of 2, so its height is set to 2*100 = 200 px.
5. The other remaining item is set to 1*100 = 100 px.

There's more...

Additionally, the `vbox` layout provides options such as controlling the vertical and horizontal alignment of the item.

Aligning the component horizontally

If there is no width specified for the items, you will notice that the items occupy the complete container width. In some cases, you may have a need to make the item appear in the middle of the container width. To achieve this, set the `align` property to `middle`, as shown in the following code snippet:

```
layout: {
  type: 'vbox',
  align: 'middle'
}
```

Refer to the `vbox` layout documentation for other valid values.

Aligning the component vertically

While the `align` property represents the horizontal axis for the `VBox` layout and helps us align the items with respect to it, the `pack` property represents the vertical axis; we can align the item in the center by setting the `pack` property to `center`, as shown in the following code snippet:

```
layout: {
  type: 'vbox',
  pack: 'center'
}
```

Refer to the `vbox` layout documentation for other valid values.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1, Gear Up for the Journey*
- The *Working with Panel* recipe

Mixing layouts

In the previous recipes we looked at the different container layouts that are available with Sencha Touch. Given the variety of layouts, a question that arises is whether these layouts are compatible with each other to an extent where they can be nested. For example, is it valid to use the `hbox` layout at the parent container level but use `vbox` inside the subcontainer? The answer is yes. Technically, it is feasible to combine multiple layouts to create complex-looking views. For example, we can have a panel with a `card` layout in which each item has an `hbox` layout, each of its items has a `vbox` layout, and the final container has an `auto` layout with few docked items defined.

In this recipe we will see how we can mix different layouts and the important points that we need to keep in mind when we use these combinations.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch03` folder inside the `www` folder.

How to do it...

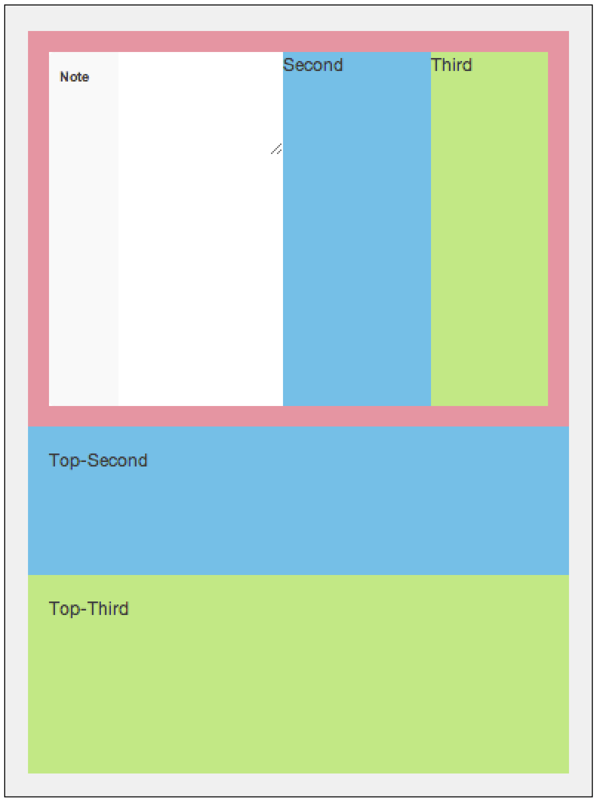
Carry out the following steps for mixing the layouts:

1. Create and open a new file named `ch03_09.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var pnl = Ext.create('Ext.Panel', {
      fullscreen: true,
      styleHtmlContent: true,
      layout: {
        type: 'vbox'
      },
      defaults: {
        styleHtmlContent: true
      },
      items: [{
        flex: 2,
        style: 'background:#E58A99;',
        layout: {
          type: 'hbox'
        },
      items: [{
        flex: 2,
        style: 'background:#E58A99;',
        layout: 'fit',
        items: [{
          xtype: 'textareafield',
          name : 'url',
          label: 'Note'
        }]
      },
      {
        width: 100,
        html: 'Second',
        style: 'background:#65B9E0;'
      },
      {
        flex: 1,
        html: 'Third',
        style: 'background:#B7E488;'
      }]
      },
      {
        height: 100,
        html: 'Top-Second',
        style: 'background:#65B9E0;'
      },
      {
        flex: 1,
        html: 'Top-Third',
        style: 'background:#B7E488;'
      }]
    });
  }
});
```

2. Include the `ch03_09.js` file in the `index.html` file.

3. Deploy and access it from the browser. The following screenshot shows the view:



How it works...

The preceding code creates a top-level panel with the `vbox` layout and one of its items having an `hbox` layout. The subitem of the panel with the `hbox` layout has an item with the `fit` layout.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1, Gear Up for the Journey*
- The *Working with Panel* recipe

Easing view navigation with the NavigationView class

In the *Building wizards using CardLayout* recipe, we saw how we can implement view navigation by adding items to a container with `CardLayout` and using the `setActiveItem` method to switch between the views. This is very handy when we have to implement a workflow, which consists of multiple steps and user switches between the previous or next view to complete the workflow. But all that was manual. Sencha Touch framework offers the new `NavigationView` class, which combines a container with a `card` layout and allows us to go back to the previous view by offering a default **Back** button. Using its `push` method, we push a new view to it and the `NavigationView` class automatically shows it as the active item.

In this recipe, we will see how we can build a view navigation using the `NavigationView` class.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch03` folder inside the `www` folder.

How to do it...

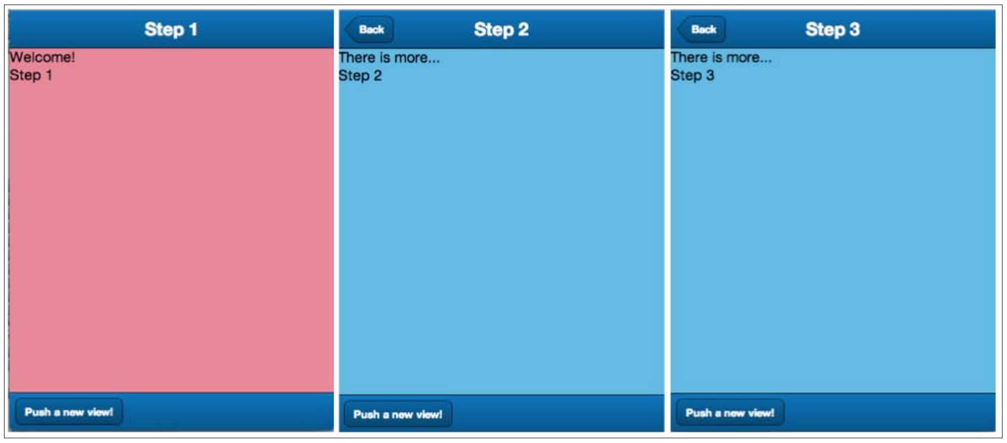
Carry out the following steps to see how to perform easy navigation with the `NavigationView` class:

1. Create and open a new file named `ch03_10.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var cnt = 1;
    var view = Ext.create('Ext.navigation.View', {
      title : 'Wizard',
      fullscreen : true,
      items : [{
        title : 'Step ' + cnt,
        id : 'card-0',
        style : 'background:#E58A99',
        html : '<h1>Welcome!</h1><p>Step ' + cnt + '</p>'
```

```
            },
            {
              docked : 'bottom',
              xtype: 'toolbar',
              items : [ {
                text : 'Push a new view!',
                handler : function() {
                  cnt++;
                  view.push({
                    title : 'Step ' + cnt,
                    style : 'background:#65B9E0',
                    html  : '<h1>There is more...</h1><p>Step '
                      + cnt + '</p>'
                  });
                }
              }]
            } ]
          });
        }
      });
```

2. Include the `ch03_10.js` file in the `index.html` file.
3. Deploy and access it from the browser. The following screenshot shows the view:



How it works...

In the preceding code, we created a navigation view with a container as its child item and also added a toolbar to it with a button. The handler of the **Push a new view!** button pushes a container on the view by calling the `push` API of the `NavigationView` class.

There's more...

In the preceding output, notice that the **Back** button appears automatically on the navigation bar of views (except on the first one); on clicking the button, we were able to go back to the previous item of the navigation view. These are offered by default. In some cases, we may want to customize things, such as showing different text for the **Back** button and showing more buttons on the navigation bar. Let us see how we can modify these defaults and do more with the `NavigationView` class.

Doing some work when the Back button is tapped

When the **Back** button is tapped/clicked, the `NavigationView` class fires the `back` event. Let us see how we can make use of it to build custom application logic.

In the preceding code, we have a logical issue where the `cnt` variable is not being set correctly when the user switches to the previous view. So, when you go back and push another view, you can see the `cnt` variable giving an undesired result. Therefore, let us register the `back` event handler on the `NavigationView` class and decrement the `cnt` variable every time the user clicks/taps on the **Back** button, as shown in the following code snippet:

```
listeners: {
  back: function() {
    cnt--;
  }
}
```

Randomly jumping to a particular view

Sometimes you may have the need to jump to a specific view that may not be the immediate previous item. For example, you can show a **Jump to first** button that the user can tap on to jump to the beginning of the navigation view from any stage of the navigation. To achieve this, we will add another button to the bottom toolbar and inside the handler we will use the `pop` method of the `NavigationView` class to jump to a view using the specific index; this is shown here:

```
, {
    text: 'Jump to first',
    handler: function() {
      view.pop(cnt-1);
      cnt = 1;
    }
}
```

Showing different text for the Back button

By default, the text used for the **Back** button is "Back". To show different text, we will have to set the `defaultBackButtonText` config on the `NavigationView` instance; this is shown here:

```
var view = Ext.create('Ext.navigation.View', {
    title : 'Wizard',
    defaultBackButtonText: '<<',
...
```

Showing the item title as Back button text

Rather than showing fixed text, **Back**, if we have to show the item title as the button text, we will have to set the `useTitleForBackButtonText` config to `true` on the `NavigationView` instance; this is shown here:

```
var view = Ext.create('Ext.navigation.View', {
    title : 'Wizard',
    useTitleForBackButtonText: true,
...
```

Customizing the navigation bar

The `Ext.navigation.Bar` class represents the bar that appears on top of the navigation view and shows the **Back** button. Now, let us say we want to show a **Home** button on the navigation bar; on tapping/clicking on it, the user should jump to the beginning of the view navigation. To achieve this, we will have to pass the additional button configuration as part of the `navigationBar` config on the `NavigationView` instance; this is shown here:

```
...
navigationBar: {
      items: [{
        text: 'Home',
        handler: function() {
          view.pop(cnt-1);
          cnt = 1;
        }
      }]
},
fullscreen : true,
...
```

In the preceding code, we added an additional **Home** button to the navigation bar, and in the handler we are jumping to the first view by calling the `pop` method. The `NavigationView` instance will show the **Home** button on the navigation bar after the **Back** button because the navigation bar uses `HBoxLayout` to align the child items.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Working with Panel* recipe
- The *Building wizards using CardLayout* recipe

4
Building Custom Views

In this chapter we will cover:

- Basic HTML templating using Template
- Using XTemplate for advanced templating
- Conditional view rendering using XTemplate
- Designing custom views using DataView
- Showing filtered data
- Responding to user actions

Introduction

In *Chapter 2*, *Catering to Your Form-related Needs*, and *Chapter 3*, *Containers and Layouts*, we saw how to make use of the different form fields, containers, and layouts to create a view of our choice. The out of the box layouts provided by Sencha Touch have a predefined way to position the components and calculate their sizes. Many a time, there may be situations in the application where the view cannot be created directly using the available containers, components, and layouts. For example, if we wanted to create a photo album where the view shows the photos in a matrix based on the dimension of the device. Alternatively, suppose we wanted to design a view similar to the Facebook feed. There is no out of box Sencha Touch layout that supports these custom view needs. And if we were to try to achieve them by mixing different layouts, it would become a heavy view that would use multiple containers. We would have to work with the styles to do some tweaking on top of what the layouts provide to align the information properly. Alternatively, Sencha Touch provides us with a way to create templates using the HTML fragments and to use them along with the data set to render custom views.

There are two types of templates provided: `Template` and `XTemplate`. `Template` provides us with basic template functionality where we can use an HTML fragment with placeholders for data. On the other hand, `XTemplate` is a more advanced template that allows us to use logical operators and mathematical calculations, execute inline code, and so on, along with an HTML fragment and data placeholders. Additionally, Sencha Touch provides **DataView**, which uses `XTemplate` to render the view, and a store for the data. It also provides events that can be used to respond to user actions.

In this chapter, we will learn about each one of these options to render the custom view and understand their specific usage.

Basic HTML templating using Template

`Template` provides a way to create templates using HTML fragments. It contains the HTML elements and various placeholders that are replaced with the values of the fields present in the data that is given to the template API to use in conjunction with the template text. For example, we may have a `<div>` element present in the body; based on the data, we may add the `<ul>` and `<li>` elements to it at runtime.

In this recipe we will look at typical usage of templates and understand what it takes to define and use one.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder named `ch04` in the same folder in which we created the `ch01` and `ch02` folders. We will be using this new folder to keep the code.

How to do it...

Carry out the following steps:

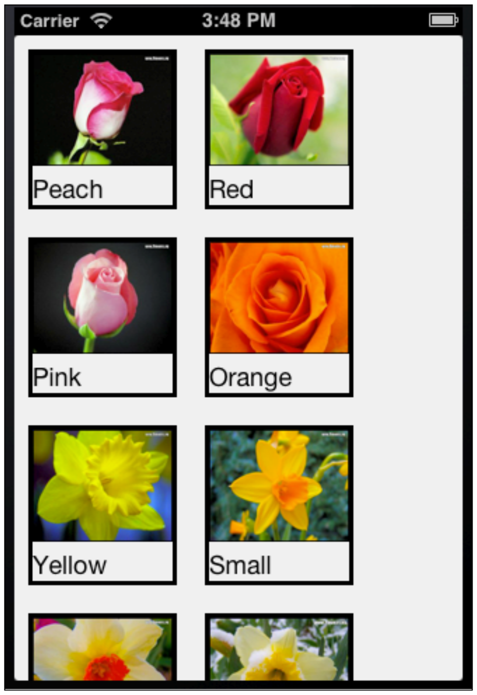
1. Create and open a new file, `ch04_01.js`, and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var data = [{
      album:'rose',
      url:'http://images.flowers.vg/250x300/rdroses01.jpg',
      title:'Rose 1',
      about:'Peach'
```

```
        },
        {
          album:'rose',
          url:'http://images.flowers.vg/250x300/roses-
            maroon3.jpg',
          title:'Rose 2',
          about:'Red'
        },
        {
          album:'rose',
          url:'http://images.flowers.vg/250x300/roses-dark-
            pink.jpg',
          title:'Rose 3',
          about:'Pink'
        },
        {
          album:'rose',
          url:'http://images.flowers.vg/250x300/roses-bright-
            orange.jpg',
          title:'Rose 4',
          about:'Orange'
        },
        {
          album:'daffodil',
          url:'http://images.flowers.vg/250x300/daffodil.jpg',
          title:'Daffodil 1',
          about:'Yellow'
        },
        {
          album:'daffodil',
          url:'http://images.flowers.vg/250x300/daffodil-
            yellow.jpg',
          title:'Daffodil 2',
          about:'Small'
        },
        {
          album:'daffodil',
          url:'http://images.flowers.vg/250x300/daffodil-white-
            orange.jpg',
          title:'Daffodil 2',
          about:'Orange'}, {
          album:'daffodil',
          url:'http://images.flowers.vg/250x300/winter_flowers_
            daffodil_white.jpg',
  title:'Daffodil 2',
  about:'Winter'}, {
  album:'hibiscus',
  url:'http://images.flowers.vg/250x300/hibiscus-
    peach.jpg',
 title:'Hibiscus 1',
  about:'Peach'
},
{
  album:'hibiscus',
  url:'http://images.flowers.vg/250x300/
    hibiscusred.jpg',
  title:'Hibiscus 1',
  about:'Red'
},
{
  album:'hibiscus',
  url:'http://images.flowers.vg/250x300/hibiscus-pink-
    pink.jpg',
  title:'Hibiscus 1',
  about:'Pink'
},
{
  album:'hibiscus',
  url:'http://images.flowers.vg/250x300/hibiscus-red-
    maroon.jpg',
  title:'Hibiscus 1',
  about:'Maroon'
},
{
  album:'hibiscus',
  url:'http://images.flowers.vg/250x300/hibiscus-pink-
    pink.jpg',
  title:'Hibiscus 1',
  about:'Pink'
},
{
  album:'hibiscus',
  url:'http://images.flowers.vg/250x300/hibiscus-red-
    bright.jpg',
  title:'Hibiscus 1',
  about:'Bright Red'
}];
```

```
        var t = new Ext.Template('<div
          style="float:left;margin:10px;border:solid;">',
          '<img border="0" src={url} title={title} width="100"
            height="80" />',
            '<p>{about}</p>',
          '</div>');

        var pnl = Ext.create('Ext.Panel', {
          fullscreen: true,
          tplWriteMode: 'append',
          tpl: t
        });
        Ext.each(data, function(item, index, allItems) {
          pnl.setData(item);
        });
      }
    });
```

2. Include `ch04_01.js` in the `index.html` file.
3. Deploy and access the index file from the browser. You may also run it using the emulator. You will see the screen shown in the following screenshot:



How it works...

In the preceding code, we have a JSON object stored in data. Each item of the data array has album, url, title, and about fields, which are referred to in the HTML fragment given to the template.

```
{
      album:'rose',
      url:' 'http://images.flowers.vg/250x300/rdroses01.jpg',
      title:'Rose 1',
      about:'Peach'
}
```

The following code instantiates Ext.Template with the HTML fragment containing the placeholders {url}, {title},and {about}, which are then replaced with the real values from the data set on the template. The placeholder used in the template name must match the field name in the data array:

```
var t = new Ext.Template
  ('<div style="float:left;margin:10px;border:solid;">',
  '<img border="0" src={url} title={title} width="100"
    height="80" />',
  '<p>{about}</p>',
  '</div>');
```

In the following code, we are calling the panel's setData method to pass the data to the template. The HTML fragment created after applying the data to the template is appended to the panel's body, which is controlled by the tplWriteMode property. We have set the property to append to instruct the framework that the HTML fragment needs to be appended:

```
Ext.each(data, function(item, index, allItems){
    pnl.setData(item);
});
```

There's more...

Template uses placeholders.These placeholders can either be a field name or an index in the data. Internally, a template goes through the stage of compilation and then starts applying the data to the template to get the final HTML fragment that is appended to the element (in this case, the panel body). Additionally, it also provides us with a way to use different inbuilt formats and apply them to the data before displaying.

Compiling Template

Compilation of a template is a costly affair because, at this stage, the framework parses the template string and replaces the placeholders with the appropriate function references (that are generated at runtime) to get the values for the placeholders. If we are creating a template once in our code and reusing it to render a view at different stages in the code, it makes sense to minimize the time spent in the compilation because now the template can be compiled only once and used multiple times. `Ext.Template` provides the option as well as a method to compile the template. The property named `compiled`, when set to `true` at the time of instantiating a template, will be instantiated and then compiled. However, if we want to compile the template on demand, we can call the `compile` method on the `Template` instance. The following code snippet shows the use of the property for an immediate compilation:

```
var t = new Ext.Template
  ('<div style="float:left;margin:10px;border:solid;">',
  '<img border="0" src={url} title={title} width="100"
    height="80" />',
  '<p>{about}</p>',
  '</div>',
  {
    compiled: true    // compile immediately
  }
);
```

The following code snippet shows the usage of the `compile` method:

```
t.compile();
```

Formatting values

In some cases, there may be a need to cook the incoming data before it is displayed on the screen. For example, you may want to format the date properly or you may want to end the long texts with ellipses. The `Ext.Template` class allows us to use the formats defined in the `Ext.util.Format` class to format the values. The following code snippet shows typical usage of a format:

```
var t = new Ext.Template([
    '<div name="{id}">',
    '<span class="{cls}">{name:trim} {value:ellipsis(10)}</span>',
    '</div>',
]);
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Using XTemplate for advanced templating

Conceptually, `XTemplate` provides functionality similar to what `Template` provides. However, it also provides certain advanced functionalities to work with the template and its data quickly. This recipe describes `XTemplate` and demonstrates the difference between `XTemplate` and `Template`.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch04` folder created inside the `www` folder.

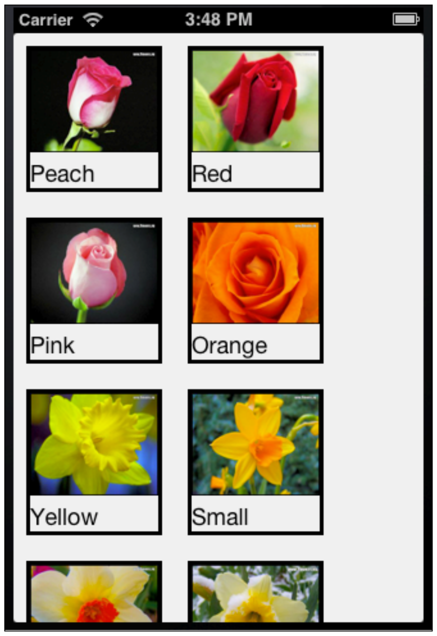
How to do it...

Carry out the following steps:

1. Create and open a new file `ch04_02.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var data = [{
      album:'rose',
      url:'http://images.flowers.vg/250x300/rdroses01.jpg',
      title:'Rose 1',
      about:'Peach'},
      ...
      ...
    {
      album:'hibiscus',
      url:'http://images.flowers.vg/250x300/hibiscus-pink-
        pink.jpg',
      title:'Hibiscus 1',
      about:'Pink'},
     {
      album:'hibiscus',
      url:'http://images.flowers.vg/250x300/hibiscus-red-
        bright.jpg',
      title:'Hibiscus 1',
      about:'Bright Red'}];
    var t = new Ext.XTemplate('<tpl for=".">',
      '<div style="float:left;margin:10px;border:solid;">',
      '<img border="0" src={url} title={title} width="100"
        height="80" />',
      '<p>{about}</p>',
      '</div></tpl>');

    var pnl = Ext.create('Ext.Panel', {
      fullscreen: true,
      tpl:t,
      data: data
    });
  }
});
```

2. Include `ch04_02.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see the screen shown in the following screenshot:



How it works...

The preceding code uses the same `JSON` data that we had used in the previous recipe.

We are then instantiating the `Ext.XTemplate` class with a similar HTML fragment that we used with the `Ext.Template` class:

```
var t = new Ext.XTemplate('<tpl for=".">',
  '<div style="float:left;margin:10px;border:solid;">',
  '<img border="0" src={url} title={title} width="100"
    height="80" />',
  '<p>{about}</p>',
  '</div></tpl>');
```

One difference is that, now, in the HTML fragment, we have enclosed the original HTML within `<tpl>`, which is specific to `XTemplate`. `<tpl for=".">` is an `XTemplate` shortcut to say that this HTML fragment inside this template will be evaluated for each item in the `data` array that is passed to the template `append` method. As the looping construct is part of the `XTemplate` class, we don't have to loop through the `data` array as compared to `Template`.

There's more...

Similar to `Ext.Template`, `Ext.XTemplate` also provides compilation and formatting capabilities.

Compiling Template

`XTemplate` also has a property named `compiled` and a method, `compile`, to accomplish the compilation task. The following code snippet shows the use of the property for immediate compilation:

```
var t = new Ext.XTemplate
  ('<div style="float:left;margin:10px;border:solid;">',
  '<img border="0" src={url} title={title} width="100"
    height="80" />',
  '<p>{about}</p>',
  '</div>',
  {
    compiled: true    // compile immediately

  }
);
```

The following code snippet shows the usage of the `compile` method:

```
t.compile();
```

Formatting values

`XTemplate` has formatting functionality similar to that which is available with `Template`. Refer to the *Basic HTML templating using Template* recipe for more details.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Basic HTML templating using Template* recipe

Conditional view rendering using XTemplate

In the previous recipe, we saw how to use `XTemplate` but did not utilize its capabilities such as using auto-filling arrays, conditional processing, and math functions to build the view by making different decisions on the incoming data. For example, in the previous recipe, we are showing all kind of flowers in our view. What if we just want to show roses? This is where `XTemplate` helps us to put the constructs inside the template definition and not make any change to the data.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch04` folder created inside the `www` folder.

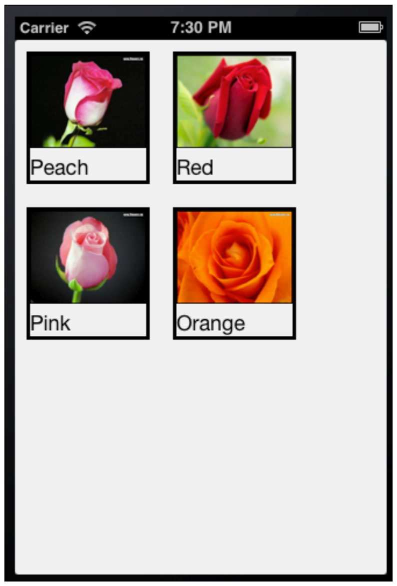
How to do it...

Carry out the following steps:

1. Create and open a new file, `ch04_03.js`, and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var data = [{
      album:'rose',
      url:'http://images.flowers.vg/250x300/rdroses01.jpg',
      title:'Rose 1',
      about:'Peach'},
    {
      album:'rose',
      url:'http://images.flowers.vg/250x300/roses-
        maroon3.jpg',
      title:'Rose 2',
      about:'Red'},
      ...
      ...
    {
      album:'hibiscus',
      url:'http://images.flowers.vg/250x300/hibiscus-pink-
        pink.jpg',
      title:'Hibiscus 1',
      about:'Pink'},
    {
      album:'hibiscus',
      url:'http://images.flowers.vg/250x300/hibiscus-red-
        bright.jpg',
      title:'Hibiscus 1',
      about:'Bright Red'}
    ];

    var pnl = Ext.create('Ext.Panel', {
      fullscreen: true,
      tpl: new Ext.XTemplate('<tpl>',
      '<tpl for="items">',
      '<tpl if="album==parent.filter &&
        this.matchFound()">',
      '<div style="float:left;margin:10px;border:solid;">',
      '<img border="0" src={url} title={title} width="100"
        height="80" />',
      '<p>{about}</p>',
      '</div></tpl></tpl>',
      '<tpl if="this.isMatchNotFound()">',
      '<h1>No match found!!',
      '</tpl></tpl>',
      {
        found: false,
        matchFound: function(){
          this.found = true;
          return this.found;
        },
      isMatchNotFound: function(){
        return this.found ? false: true;
      }
    }),
    data: {filter: 'rose', items: data}
    });
  }
});
```

2. Include `ch04_03.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see the screen shown in the following screenshot:



How it works...

The preceding code uses the additional capabilities, such as conditions, loops, inline functions, and inbuilt variables, offered by `XTemplate` to show the filtered items. Based on the value specified in the `filter` property of the `data`, which is passed to the template, it checks whether there are images whose `album` name matches with the `filter`. The matching ones are shown in the view; otherwise, the **No match found!!** message is displayed.

The data that is being passed to the template has the following structure:

```
{filter: 'rose', items: data}
```

`<tpl for="items">` tells us that the content inside this `<tpl>` tag is applied to each item of the `items` array passed as part of `data`.

Look at this line:

```
<tpl if="album==parent.filter && this.matchFound()">
```

We are comparing the `album` field on the incoming data with the `filter` value (`'rose'`, in this case) and calling an inline function `matchFound()` to set a member property, `found`, to `true` indicating that a matching item has been found, as shown in the following code:

```
matchFound: function(){
  this.found = true;
  return this.found;
}
```

For the entire matching item, the following HTML fragment is used to render the item:

```
'<div style="float:left;margin:10px;border:solid;">',
'<img border="0" src={url} title={title} width="100" height="80"
  />',
'<p>{about}</p>',
'</div>'
```

The following template fragment checks whether there are any matches found. If not, it displays `No match found!!`:

```
'<tpl if="this.isMatchNotFound()">',
  '<h1>No match found!!',
'</tpl>
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Designing custom views using DataView

`Template` and `XTemplate` provide the way to create elements using the template, placeholders, and data. There is one thing that is not really straightforward and this is the support for events. For example, if you want to handle the click on a rose to show a bigger picture of it, this is not very straightforward. We will have to work with the elements and register handlers for the different DOM events that we may be interested in. Also, there is no way to leverage the store. A Store is covered in more detail in the next chapter. For now, we can say that a store is a data structure that can hold a collection of records and that can be associated with components, such as `DataView`, to provide it the required data to render their view. Sencha Touch provides a convenient way to create views using `XTemplate` and link it with a data store. It also provides events that can be handled to respond to the user action using `DataView`.

This recipe describes the steps to use `DataView`.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch04` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. Create and open a new file, `ch04_04.js`, and paste the following code into it:

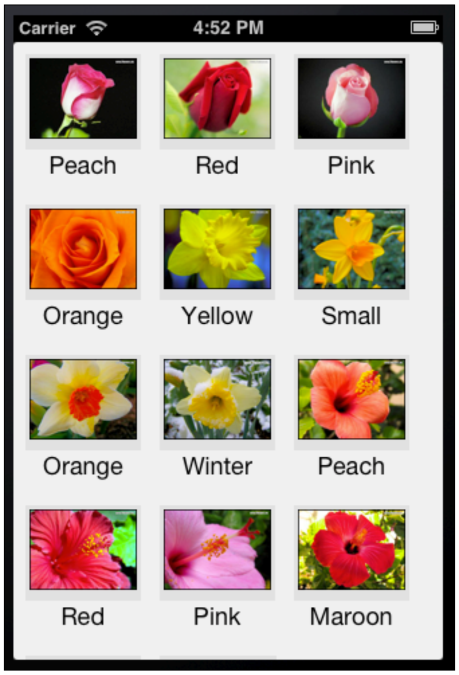
```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var data = [{
      album:'rose',
      url:'http://images.flowers.vg/250x300/rdroses01.jpg',
      title:'Rose 1',
      about:'Peach'},
    {
      album:'rose',
      url:'http://images.flowers.vg/250x300/roses-
        maroon3.jpg',
      title:'Rose 2',
      about:'Red'},
    ...
    ...
    {
      album:'hibiscus',
      url:'http://images.flowers.vg/250x300/hibiscus-red-
        bright.jpg',
      title:'Hibiscus 1',
      about:'Bright Red'}];

    var store = Ext.create('Ext.data.Store', {
      data: data,
      fields: [
        'url', 'title','about', 'album'
      ]
    });
```

```
        var tpl = new Ext.XTemplate(
          '<div class="thumb-wrap" id="{title}">',
          '<div class="thumb"><img src="{url}" title="{title}">
            </div>',
          '<span>{about}</span></div>',
          '<div class="x-clear"></div>');

        Ext.create('Ext.dataview.DataView', {
          id:'images-view',
          fullscreen: true,
          store: store,
          itemTpl: tpl,
          emptyText: 'No images to display'
        });
      }
    });
```

2. Create and open a new file, `ch04.css`, and paste the following style code into it:

```
#images-view .thumb{
  background: #dddddd;
  padding: 3px;
}

#images-view .thumb img{
  height: 60px;
  width: 80px;
}

#images-view .thumb-wrap{
  float: left;
  margin: 4px;
  margin-right: 0;
  padding: 5px;
}

#images-view .thumb-wrap span{
  display: block;
  overflow: hidden;
  text-align: center;
}

#images-view .x-item-selected .thumb-wrap{
  background:#64C6FF;
}
```

3. Update the `index.html` file by including the `.css` and `.js` files.
4. Deploy and access it from the browser. You may also run it using the emulator. You will see the screen shown in the following screenshot:



How it works...

The preceding code uses the `Ext.dataview.DataView` class to create a custom view. We have created a panel with `DataView` as its child item.

First, we created a `JSON` store to hold the data. You may refer to *Chapter 5*, *Dealing with Data and Data Sources* for detailed discussion about different types of stores:

```
var store = new Ext.data.Store({
    data: data,
    fields: [
      'url', 'title','about', 'album'
    ]
});
```

`url`, `title`, `about`, and `album` are the fields that will be present in the record stored within `store`. The `data` property is used to pass the array to the store that is used to replace the placeholder values used in `XTemplate` and produce the net HTML fragment.

Then we instantiated `XTemplate` in the following part of the code:

```
var tpl = new Ext.XTemplate(
  '<div class="thumb-wrap" id="{title}">',
  '<div class="thumb"><img src="{url}" title="{title}"></div>',
  '<span>{about}</span></div>',
  '<div class="x-clear"></div>'
);
```

The template is using `title`, `url`, and `about` as the placeholders. The CSS classes used in the template are defined in the `ch04.css` file.

Next, we created `DataView` with `fullscreen` set to `true` so that it occupies the complete viewport.

To the panel, we are adding `DataView` as follows:

```
Ext.create('Ext.dataview.DataView', {
  id:'images-view',
  fullscreen: true,
  store: store,
  itemTpl: tpl,
  emptyText: 'No images to display'
});
```

`store:store` is where we associated our store object with `DataView`. And the `itemTpl` property helps us in associating `XTemplate` with `DataView`, which it will use to render the items in the view. We have used the `id` attribute and used the same in the CSS definition.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with Panel* recipe in *Chapter 3*, *Containers and Layouts*
- The *Working with store* recipe in *Chapter 5*, *Dealing with Data and Data Sources*

Showing the filtered data

In the previous recipe, we saw how to create a `DataView` component and use `XTemplate` and a store to generate the view. In this recipe, we will see whether we have to show only the relevant items in the view, and how we go about approaching it.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch04` folder created inside the `www` folder.

How to do it...

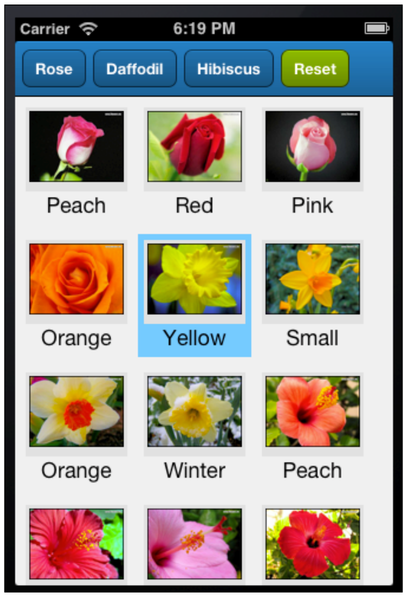
Carry out the following steps:

1. Create and open a new file, `ch04_05.js`, and paste the following code into it:

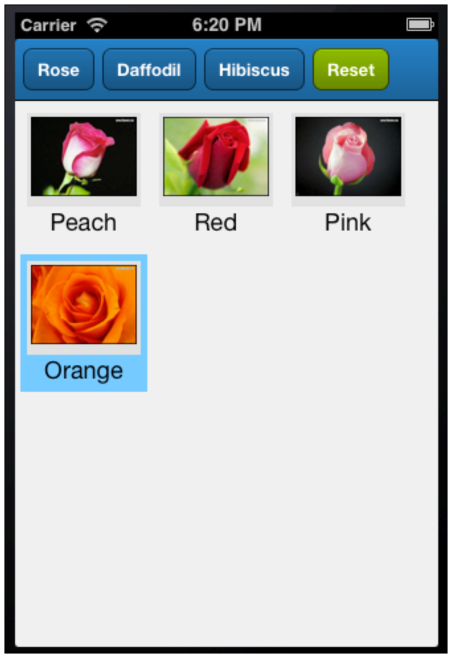
```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var data = [{
      album:'rose',
      url:'http://images.flowers.vg/250x300/rdroses01.jpg',
      title:'Rose 1',
      about:'Peach'},
...
...
    {
      album:'hibiscus',
      url:'http://images.flowers.vg/250x300/hibiscus-red-
        bright.jpg',
      title:'Hibiscus 1',
      about:'Bright Red'}
    ];

    var store = Ext.create('Ext.data.Store', {
      data: data,
      fields: [
        'url', 'title','about', 'album'
      ]
    });

    var filter = function(criteria) {
      store.clearFilter();
      return store.filterBy(function(record, id){
        if (record.get('album') === criteria ||
          Ext.isEmpty(criteria))
          return true;
        else
          return false;
      });
    };

    var tpl = new Ext.XTemplate(
      '<div class="thumb-wrap" id="{title}">',
      '<div class="thumb"><img src="{url}" title="{title}">
        </div>',
      '<span>{about}</span></div>',
      '<div class="x-clear"></div>');

    Ext.create('Ext.dataview.DataView', {
      id:'images-view',
      fullscreen: true,
      store: store,
      itemTpl: tpl,
      emptyText: 'No images to display',
      items: [{
        xtype: 'toolbar',
        docked: 'top',
        items: [
          {
            text: 'Rose',
            id: 'rose-button',
            handler: function() {
              filter('rose');
            }
          },
          {
            text: 'Daffodil',
            id: 'daffodil-button',
            handler: function() {
              filter('daffodil');
            }
          },
          {
            text: 'Hibiscus',
            id: 'hibiscus-button',
            handler: function() {
              filter('hibiscus');
            }
          },
          {
            text: 'Reset',
            id: 'reset-button',
            ui: 'confirm',
            handler: function() {
              Ext.getCmp('images-view').setActiveItem(0);
              filter('');
```

```
                    }
                  },
                  {
                    text: 'Back',
                    id: 'back-button',
                    ui: 'back',
                    hidden: true,
                    handler: function() {
                      Ext.getCmp('images-view').setActiveItem(0);
                      this.hide();
                      Ext.getCmp('rose-button').show();
                      Ext.getCmp('daffodil-button').show();
                      Ext.getCmp('hibiscus-button').show();
                    }
                  }
                ]
              }]
            });
          }
        });
```

2. Include `ch04_05.js` in the `index.html` file.
3. Deploy and access the `index.html` file from the browser. You may also run it using the emulator. You will see the screen shown in the following screenshot:



4. Click on the **Rose** button. You will see roses on the screen as shown in the following screenshot:



How it works...

In the preceding code, besides `DataView`, we have also added a docked toolbar with a button for each album (**Rose**, **Daffodil**, and **Hibiscus**) and a **Reset** button. We have then registered the click handler for all the buttons and each handler is calling the `filter` function with the filter criteria as follows:

```
{
  text: 'Hibiscus',
  id: 'hibiscus-button',
  handler: function() {
    filter('hibiscus');
  }
}
```

In case of the **Reset** click handler, `filter('')` is called, which ensures that all the items are displayed in the view as shown in the following code snippet:

```
var filter = function(criteria) {
    store.clearFilter();
```

```
        return store.filterBy(function(record, id){
          if (record.get('album') === criteria ||
            Ext.isEmpty(criteria))
            return true;
          else
            return false;
          });
    };
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with Panel* recipe in *Chapter 3*, *Containers and Layouts*
- The *Designing custom views using DataView* recipe
- The *Working with store* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Filtering data* recipe in *Chapter 5*, *Dealing with Data and Data Sources*

Responding to user actions

So far, we have seen how to create `DataView`, bind it to `XTemplate` and `store`, and apply certain filtering on the data. In this recipe, we will see how to handle the events generated as part of the user action, for example, when a user selects an item in the view.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch04` folder created inside the `www` folder.

How to do it...

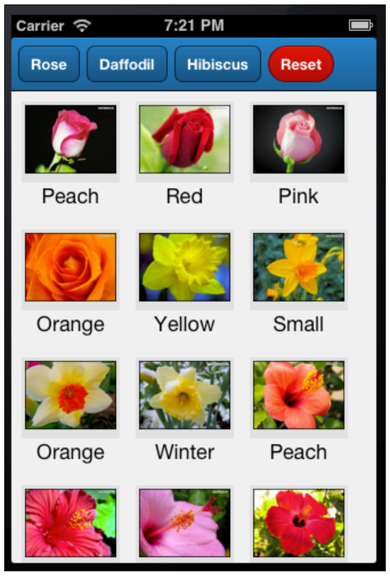
Carry out the following steps:

1. Create and open a new file, `ch04_06.js`, and paste the following code into it:

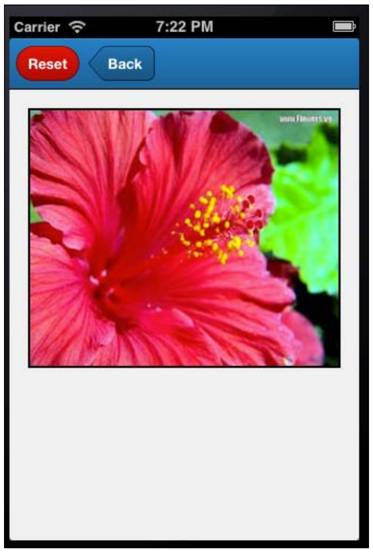
```
Ext.application({
  name : 'MyApp',
  launch : function() {
    var data = [{
      album:'rose',
      url:'http://images.flowers.vg/250x300/rdroses01.jpg',
      title:'Rose 1',
      about:'Peach'},
...
...
    {
      album:'hibiscus',
      url:'http://images.flowers.vg/250x300/hibiscus-red-
        bright.jpg',
      title:'Hibiscus 1',
      about:'Bright Red'}];

    var store = Ext.create('Ext.data.Store', {
      data: data,
      fields: [
          'url', 'title','about', 'album'
      ]
    });

    var filter = function(criteria) {
      store.clearFilter();
      return store.filterBy(function(record, id){
        if (record.get('album') === criteria ||
          Ext.isEmpty(criteria))
         return true;
        else
          return false;
    });
  };

    var tpl = new Ext.XTemplate(
      '<div class="thumb-wrap" id="{title}">',
      '<div class="thumb"><img src="{url}" title="{title}">
        </div>',
      '<span>{about}</span></div>',
      '<div class="x-clear"></div>');

    var pnl = new Ext.Panel({
      id:'images-view',
      fullscreen: true,
      layout: 'card',
      items: [Ext.create('Ext.dataview.DataView', {
        store: store,
        scrollable: 'vertical',
        itemTpl: tpl,
        emptyText: 'No images to display',
        listeners: {
          selectionchange: function(model, recs) {
            if (recs.length > 0) {
              Ext.getCmp('detail-panel').setHtml('<img
                src="' + recs[0].data.url + '">');
              Ext.getCmp('images-view').setActiveItem(1);
              Ext.getCmp('back-button').show();
              Ext.getCmp('rose-button').hide();
              Ext.getCmp('daffodil-button').hide();
              Ext.getCmp('hibiscus-button').hide();
            }
          }
        }
      }),
      Ext.create('Ext.Panel', {
        id: 'detail-panel',
        styleHtmlContent: true,
        scrollable: 'vertical'
      }),
      {
        xtype: 'toolbar',
        docked: 'top',
        items: [
          {
            text: 'Rose',
            id: 'rose-button',
            handler: function() {
              filter('rose');
            }
          },
          {
            text: 'Daffodil',
            id: 'daffodil-button',
            handler: function() {
              filter('daffodil');
            }
          },
          {
            text: 'Hibiscus',
            id: 'hibiscus-button',
            handler: function() {
              filter('hibiscus');
          }
        },
        {
          text: 'Reset',
          id: 'reset-button',
          ui: 'decline-round',
          handler: function() {
            Ext.getCmp('images-view').setActiveItem(0);
            filter('');
          }
        },
        {
          text: 'Back',
          id: 'back-button',
          ui: 'back',
          hidden: true,
          handler: function() {
            Ext.getCmp('images-view').setActiveItem(0);
            this.hide();
            Ext.getCmp('rose-button').show();
            Ext.getCmp('daffodil-button').show();
            Ext.getCmp('hibiscus-button').show();
          }
        }
      ]
      }]
    });
  }
});
```

2. Include `ch04_06.js` in the `index.html` file.

3. Deploy and access it from the browser. You may also run it using the emulator. You will see screen shown in the following screenshot:



4. Click on an item. You will see the bigger image with the **Reset** and **Back** button on the toolbar, as shown in the following screenshot:



How it works...

The preceding code makes changes on top of the functionality that we built in the previous recipe. We changed the layout of the main container panel from `fit` to `card` and added `DataView` to the first card and another panel to the second card to show the bigger image of the selected flower. Also, we added a **Back** button to the docked toolbar so that users can come back to the multiple images view from the detail view.

A `selectionchange` listener is registered to show the bigger image of the flower on the second card panel, switch the active panel to the second, and show/hide the toolbar buttons, appropriately, as follows:

```
emptyText: 'No images to display',
        listeners: {
          selectionchange: function(model, recs) {
            if (recs.length > 0) {
              Ext.getCmp('detail-panel').setHtml('<img src="' +
                recs[0].data.url + '">');
              Ext.getCmp('images-view').setActiveItem(1);
              Ext.getCmp('back-button').show();
              Ext.getCmp('rose-button').hide();
              Ext.getCmp('daffodil-button').hide();
              Ext.getCmp('hibiscus-button').hide();
            }
          }
        }
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Working with Panel* recipe in *Chapter 3, Containers and Layouts*
- The *Designing custom views using DataView* recipe
- The *Building wizards using CardLayout* recipe in *Chapter 3, Containers and Layouts*

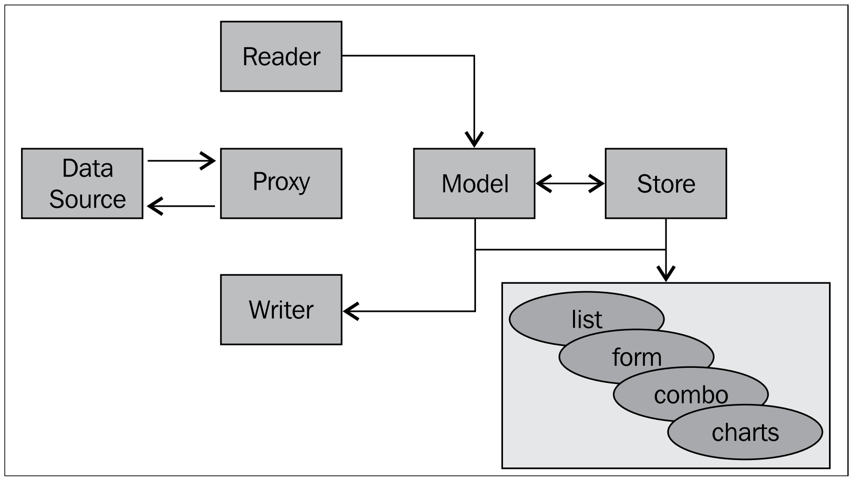
5
Dealing with Data and Data Sources

In this chapter we will cover:

- Creating a model
- Loading the form using a data model
- Working with a store
- Converting incoming JSON data into a model using JsonReader
- Converting incoming XML data into the model using XmlReader
- Validations in models
- Defining your custom validation
- Relating models using association
- Persisting session-specific data using the SessionStorage proxy
- Persisting data using the LocalStorage proxy
- Accessing in-memory data using the Memory proxy
- Loading data through AJAX using the Ajax proxy
- Sorting data
- Data grouping
- Filtering data
- Using a cross-domain URL in your application
- Working with Web SQL databases

Introduction

Imagining an application without the need for data is impossible in today's world. Almost every application has a need for data and some way to store and work with them effectively and efficiently. Sencha Touch provides a rich set of classes to work with varied data sources, represent structured data, and store it locally; it can then be fed to different data-centric components, such as lists, forms, comboboxes, charts, and so on. It also provides classes and APIs to validate, filter, sort, and group data. The following diagram depicts the different classes that are part of the data infrastructure provided by Sencha Touch:

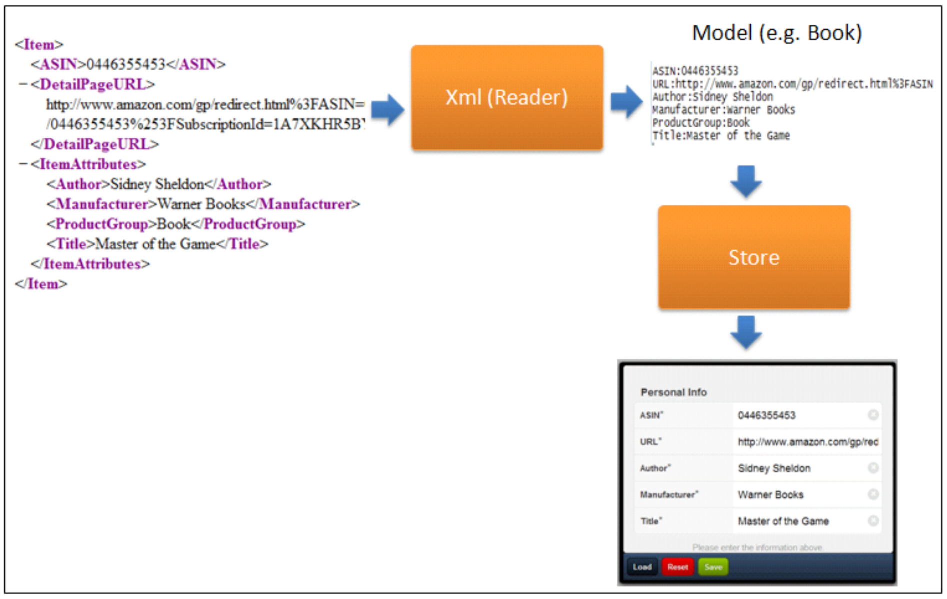


These classes are explained as follows:

- **Proxy**: Proxies allow us to interface with different data sources such as REST services, Servlet, in-memory array, HTML5-based storage, and so on to read data from or save data to.
- **Reader**: Readers are used when the data is being loaded. They interpret the data into a model or a store. Based on the type of data we have to deal with, the respective reader is used; for example, for JSON type data, `JsonReader` is used whereas for XML data, `XmlReader` is used.
- **Writer**: Writers are used when the data is being saved. Similar to readers, an appropriate writer is used based on the type of data we deal with, JSON or XML.
- **Model**: Models represent the object that our application uses and works with. For example, a user, payment objects used by the application containing application specific fields, and methods manipulating those fields. A store contains a collection of such models.

- **Store**: Stores are the collection that contains the models, and are used by the different components. This is the class that helps us reuse the collection across multiple components. For example, the same store can be used for populating a grid as well as a chart.

The following diagram depicts a typical flow involving different concepts to show how the raw data from a data source is rendered in a grid:



```
<Item>
  <ASIN>0446355453</ASIN>
 - <DetailPageURL>
    http://www.amazon.com/gp/redirect.html%3FASIN=
    /0446355453%253FSubscriptionId=1A7XKHR5B
  </DetailPageURL>
 - <ItemAttributes>
    <Author>Sidney Sheldon</Author>
    <Manufacturer>Warner Books</Manufacturer>
    <ProductGroup>Book</ProductGroup>
    <Title>Master of the Game</Title>
  </ItemAttributes>
</Item>
```

```
ASIN:0446355453
URL:http://www.amazon.com/gp/redirect.html%3FASIN
Author:Sidney Sheldon
Manufacturer:Warner Books
ProductGroup:Book
Title:Master of the Game
```

Personal Info

ASIN*	0446355453
URL*	http://www.amazon.com/gp/red
Author*	Sidney Sheldon
Manufacturer*	Warner Books
Title*	Master of the Game

Please enter the information above.

In this chapter, we will learn about every aspect of the data infrastructure provided by Sencha Touch. We will work through the models to represent our data structure, use it to render the views, and also make use of the stores and different proxies to load and save data.

Creating a model

Let's start with understanding how we can represent a data structure using a model and create objects using it.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1, Gear Up for the Journey*.

Create a new folder `ch05` in the `www` folder where we created the `ch01` and `ch02` folders. We will be using this new folder in which the code will be kept.

How to do it...

Carry out the following steps:

1. Create and open a new file named `ch05_01.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {
    Ext.define('User', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          {name: 'name',  type: 'string'},
          {name: 'age',   type: 'int'},
          {name: 'phone', type: 'string'},
          {name: 'email', type: 'string'},
          {name: 'alive', type: 'boolean', defaultValue:
            true}
        ]
      }
    });

  var user = Ext.create('User', {
    name : 'Ajit Kumar',
    age  : 24,
    phone: '555-555-5555',
    email: 'ajitkumar@walkingtree.in'
});

  Ext.Msg.alert('INFO',user.get('name'));

  }
});
```

2. Include `ch05_01.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator.

How it works...

`Ext.define` defines and registers a model with the model manager named `ModelMgr`.

Once registered, using the `Ext.create` method we created a model with specific data. To the `create` method, we also passed the model name, `User`, that we had registered with `ModelMgr`. Based on this, `ModelMgr` constructs an object for us, copies the field values, and returns it to us.

Once a model is created, the class system, by default, provides getter and setter methods for every field. `user.get('name')` returns the value stored in the `name` field of the model. To set a field value, we call the setter method named `user.set('age', 33)`.

Each field in the `fields` array represents `Ext.data.Field`. There are various useful properties supported by the field. However, in the preceding code, we have used `name`, `type`, and `defaultValue`. The `type` property, when specified, is used by the framework to do the conversion and formatting of the incoming value based on the specified type. The `Ext.data.Field` class supports various types such as `auto`, `string`, `int`, `float`, `boolean`, `date`, and `defaultValue`.

`auto` allows every kind of value. If no type property is specified, `auto` is selected as the default. `defaultValue` helps us to set the value that will be used as default for a field.

For the date type field, we can also use the `dateFormat` property to specify the format in which the date will be converted.

Another important property on a field is `convert`. This accepts a function that can be used to convert the value provided by `Reader` into an object that will be stored in the model.

There's more...

We can imagine a model as an object and, due to its very nature, it also allows us to define methods inside it to implement certain logic. Additionally, it also allows us to create a model by extending another model. Let's see how to make use of these functionalities.

Adding methods to a model

The following code snippet shows how a `changeName` method is defined inside the `User` model, which is appending an additional text to the user's name:

```
Ext.define('User', {
  extend: 'Ext.data.Model',
  config: {
    fields: [
      {name: 'name',  type: 'string'},
      {name: 'age',   type: 'int'},
      {name: 'phone', type: 'string'},
      {name: 'email', type: 'string'},
      {name: 'alive', type: 'boolean', defaultValue: true}
    ]
  },

  changeName: function() {
    var oldName = this.get('name'),
    newName = oldName + " Azad";

    this.set('name', newName);
  }
});
```

Once the method is defined, calling it is as easy as calling a method on any object, as shown in the following code snippet:

```
user.changeName();
Ext.Msg.alert('INFO', user.get('name'));
```

Extending a model

Sencha Touch follows object-oriented approaches and methodologies. As part of this, it has also provided a mechanism to extend one class from another, though it is not something offered by JavaScript directly. And the same has been applied to models as well, which allows us to create a model by extending an existing model. The following code snippet shows that we are defining a model named `MyUser` that is extending the `User` model and adding a new field named `dob`:

```
Ext.define('MyUser', {
    extend: 'User',
    config: {
      fields: [
        {name: 'dob',  type: 'string'}
      ]
    }
});
```

The following code shows instantiating `MyUser`, which ensures that the properties from the `User` class are available on `MyUser` as part of the extend mechanism:

```
var myuser = Ext.create('MyUser', {
  name : 'Ajit Kumar',
  age  : 24,
  phone: '555-555-5555',
  email: 'ajitkumar@walkingtree.in',
  dob: '04-04-1978'
});
Ext.Msg.alert('INFO', myuser.get('name') + ' : dob : ' +
  myuser.get('dob'));
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Loading the form using a data model

In this recipe, we will see how to make use of the model that we created in the previous recipe to populate the fields in a form.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

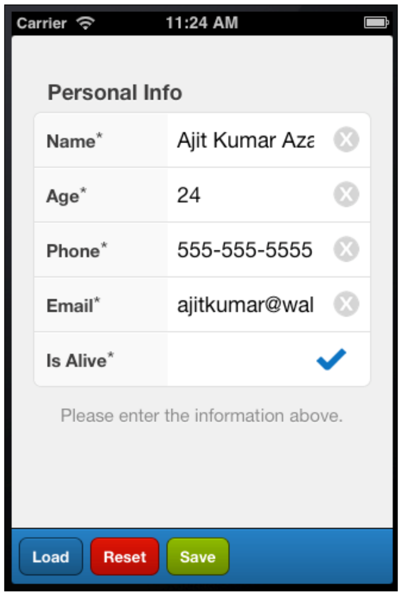
How to do it...

Carry out the following steps:

1. Create and open a new file `ch05_02.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',

  launch: function() {

    Ext.define('User', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          {name: 'name',  type: 'string'},
          {name: 'age',   type: 'int'},
          {name: 'phone', type: 'string'},
          {name: 'email', type: 'string'},
          {name: 'alive', type: 'boolean', defaultValue:
            true}
        ]
      },

      changeName: function() {
        var oldName = this.get('name'),
        newName = oldName + " Azad";

        this.set('name', newName);
      }
    });

  var user = Ext.create('User', {
    name : 'Ajit Kumar',
    age  : 24,
    phone: '555-555-5555',
    email: 'ajitkumar@walkingtree.in'
  });

  user.changeName();

    var form;

    var formBase = {
      scrollable: 'vertical',
      items: [{
        xtype: 'toolbar',
        docked: 'bottom',
        items: [
          {
            text: 'Load',
            handler: function() {
              form.setRecord(user);
            }
          },
          {
            text: 'Reset',
            ui: 'decline',
            handler: function() {
              form.reset();
            }
          },
          {
            text: 'Save',
            ui: 'confirm',
            handler: function() {
              Ext.Msg.alert("INFO", "In a real
                implementation,this will be saved!");
            }
          }
        ]
      },
      {
        xtype: 'fieldset',
        title: 'Personal Info',
        instructions: 'Please enter the information
          above.',
        defaults: {
          required: true,
          labelAlign: 'left',
          labelWidth: '40%'
        },
        items: [
          {
            xtype: 'textfield',
            name : 'name',
            label: 'Name',
            useClearIcon: true,
            autoCapitalize : false
          },
          {
            xtype: 'numberfield',
            name : 'age',
            label: 'Age',
            useClearIcon: false
          },
          {
            xtype: 'textfield',
            name : 'phone',
            label: 'Phone',
            useClearIcon: true
          },
          {
            xtype: 'emailfield',
            name : 'email',
            label: 'Email',
            placeHolder: 'you@sencha.com',
            useClearIcon: true
          },
          {
            xtype: 'checkboxfield',
            name : 'alive',
            label: 'Is Alive',
            useClearIcon: true
          }]
        }
      ],
      listeners : {
        submit : function(form, result){
          console.log('success', Ext.toArray(arguments));
        },
        exception : function(form, result){
          console.log('failure', Ext.toArray(arguments));
        }
      }
    };

    if (Ext.os.is.Phone) {
      //phone specific configuration
      formBase.fullscreen = true;
    }
    else
    {
      //desktop specific configuration
      Ext.apply(formBase, {
        autoRender: true,
        floating: true,
        modal: true,
        centered: true,
        hideOnMaskTap: false,
        height: 385,
        width: 480
      });
    }

    form = Ext.create('Ext.form.FormPanel', formBase);
    Ext.Viewport.add(form);

    }
});
```

2. Include `ch05_02.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator.
4. Click on the **Load** button to load model data into the fields. You will see the screen as shown in the following screenshot:



How it works...

In the preceding code, we created a model, a form panel with fields **Name**, **Age**, **Phone**, **Email**, and **Is Alive** as well as buttons **Load**, **Save**, and **Reset** in the docked toolbar. On clicking the **Load** button, the following handler code loads the form field with the values from the `user` model:

```
form.setRecord(user);
```

For this to work, the field name in `form` must match with the field name in `model`.

There's more...

When we use the model to update a view, such as form, that has editable fields whose values can be changed by the user, a natural need arises where we question whether the model will be updated automatically The answer is, no. If we intend to get the updated model and then work with it to, say, save, we need to do some work. Let's see what exactly we will have to do if we have to use the model to save the updated form data.

Saving form data using the associated model

The `getRecord` method of `Ext.form.FormPanel` returns the model instance currently loaded into the `form`. However, the model is not automatically updated when the field value changes. If we want to get an updated model at any instance, the following piece of code should be written inside the **Save** button handler:

```
var formValues = form.getValues();
user.set(formValues);
user.save();
```

Once we have updated our model, the `save` method takes care of saving it to the appropriate data source. The detail about how it identifies which data source and how it knows whether the data needs to be sent in XML or JSON format, and so on, will be covered in the subsequent recipes.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe in *Chapter 2*, *Catering to Your Form-related Needs*
- The *Working with the select field* recipe in *Chapter 2*, *Catering to Your Form-related Needs*
- The *Creating a model* recipe

Working with a store

So far we saw how to define and create a model and use it to populate a form. There are various other components that work with a collection of models that need to be saved in a store. In this recipe, we will look at the steps required to define a store, use it to contain models, and populate the data in a component.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

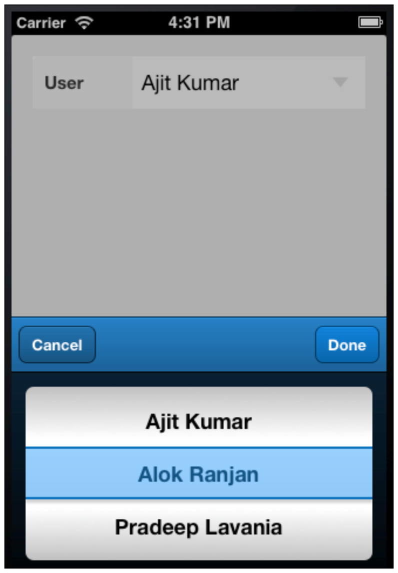
How to do it...

Carry out the following steps:

1. Create and open a new file named `ch05_03.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',

  launch: function() {

    Ext.define('User', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          {name: 'name',  type: 'string'},
          {name: 'age',   type: 'int'},
          {name: 'phone', type: 'string'},
          {name: 'email', type: 'string'},
          {name: 'alive', type: 'boolean', defaultValue:
            true}
        ]
      }
    });

    var store = Ext.create ('Ext.data.Store', {
      model: 'User',
      data : [{
        name : 'Ajit Kumar',
        age  : 32,
        phone: '555-555-5555',
        email: 'ajit@walkingtree.in'
      },
      {
        name : 'Alok Ranjan',
        age  : 32,
        phone: '123-456-7890',
        email: 'alok@walkingtree.in'
      },
      {
        name : 'Pradeep Lavania',
        age  : 34,
        phone: '987-654-3210',
        email: 'pradeep@walkingtree.in'
      }
      ]
    });

    var form;

    var formBase = {
      scrollable: 'vertical',
      items: [{
        xtype: 'selectfield',
        name : 'user',
        label: 'User',
        store: store,
        valueField: 'name',
        displayField: 'name'
      }]
    };

    if (Ext.os.is.Phone) {
      formBase.fullscreen = true;
    } else {
      Ext.apply(formBase, {
        autoRender: true,
        floating: true,
        modal: true,
        centered: true,
        hideOnMaskTap: false,
        height: 385,
        width: 480
      });
    }

    form = Ext.create('Ext.form.FormPanel', formBase);
    Ext.Viewport.add(form);

  }
});
```

2. Include `ch05_03.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see the screen as shown in the following screenshot:



How it works...

In the preceding code, we created a form panel with a combobox that has a store associated with it. The following code creates a `store` object using the inline data containing the user information:

```
var store = Ext.create ('Ext.data.Store', {
    model: 'User',
    data : [{
    name : 'Ajit Kumar',
    age  : 32,
    phone: '555-555-5555',
    email: 'ajit@walkingtree.in'
    }, {
    name : 'Alok Ranjan',
    age  : 32,
    phone: '123-456-7890',
    email: 'alok@walkingtree.in'
    }, {
    name : 'Pradeep Lavania',
    age  : 34,
    phone: '987-654-3210',
    email: 'pradeep@walkingtree.in'
    }
    ]
});
```

The model property on store instructs the store that each item in the data array will be converted into a User model. After this, we created a combobox using the store in the following code:

```
items: [{
    xtype: 'selectfield',
    name : 'user',
    label: 'User',
    store: store,
    valueField: 'name',
    displayField: 'name'
}]
```

valueField and displayField contain the model field names whose value will be read to populate the combobox. While displayField is used to show the drop-down values to the user, valueField is sent to the URL when a form containing the combobox is posted. In our code, both the fields are using the name field, so we see the username appearing in the selection list.

There's more...

In this recipe, we saw how to make use of the inline data to populate a store and subsequently the combobox. In an application, we may have the need to add the records dynamically to the store based on certain application logic. Let's see how we can do it.

Adding a record to the store at runtime

There are multiple options depending upon what exactly we want to do. Let's visit each of the options and understand what their specific usage is.

To add a record at the end of the existing record set, we need to call the add method.

```
store.add({
  name : 'Priti',
  age  : 30,
  phone: '987-654-3210',
  email: 'priti@walkingtree.in'
});
```

In case we want to insert the new record at a specific position in `store`, then the `insert` method can be used, as shown in the following code snippet, where we are adding the new record at index `1`:

```
store.insert(1, {
  name : 'Priti',
  age  : 30,
  phone: '987-654-3210',
  email: 'priti@walkingtree.in'
});
```

Last but not the least, if we have models and want to use them to add records to `store`, we will use the `add` method on `store` as follows:

```
var user = Ext.create('User', {
  name : 'Pratyush Kumar',
  age  : 5,
  phone: '987-654-3210'
});

  store.add([user]);
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Get your form ready with form panels* recipe in *Chapter 2*, *Catering to Your Form-related Needs*
- The *Working with the select field* recipe in *Chapter 2, Catering to Your Form-related Needs*
- The *Creating a model* recipe

Converting incoming JSON data into a model using JsonReader

As we saw earlier in the chapter, a reader helps us in data loading and converting the incoming data into a model, which can then be added to a store. Based on the type of data, Sencha Touch provides two readers namely `JsonReader` and `XmlReader`. In this recipe, we will see how to make use of the `JsonReader` to read the JSON data and prepare a model out of it.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1, Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. Create and open a new file `users.json` and paste the following code into it:

```
{"users": [{
    "id": "1",
      "name" : "Ajit Kumar",
      "age"  : "32",
      "phone": "555-555-5555",
      "email": "ajit@walkingtree.in"
  }, {
    "id": "2",
      "name" : "Alok Ranjan",
      "age"  : "32",
      "phone": "123-456-7890",
      "email": "alok@walkingtree.in"
  }, {
      "id": "3",
      "name" : "Pradeep Lavania",
      "age"  : "34",
      "phone": "987-654-3210",
      "email": "pradeep@walkingtree.in"
}]
}
```

2. Create and open a new file named `ch05_04.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',

  launch: function() {
    Ext.define('User', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          'id',
          {name: 'name',  type: 'string'},
          {name: 'age',   type: 'int'},
          {name: 'phone', type: 'string'},
          {name: 'email', type: 'string'},
          {name: 'alive', type: 'boolean', defaultValue:
            true}
        ]
      }
    });

    var store = Ext.create('Ext.data.Store', {
      model: 'User',
      autoLoad: true,
      proxy: {
        type: 'ajax',
        url : 'users.json',
        reader: {
          type: 'json',
          rootProperty: 'users'
        }
      }
    });

    var form;

    var formBase = {
      scrollable: 'vertical',
      items: [{
        xtype: 'selectfield',
        name : 'user',
        label: 'User',
        store: store,
        valueField: 'name',
        displayField: 'name'
      }]
    };

    if (Ext.os.is.Phone) {
        formBase.fullscreen = true;
    } else {
      Ext.apply(formBase, {
        autoRender: true,
        floating: true,
        modal: true,
        centered: true,
        hideOnMaskTap: false,
        height: 385,
        width: 480
      });
    }

    form = new Ext.form.FormPanel(formBase);
    Ext.Viewport.add(form);
  }
});
```

3. Include `users.json` and `ch05_04.js` in the `index.html` file.
4. Deploy and access it from the browser. You may also run it using the emulator.

How it works...

The code creates a form panel with a selection field that shows the list of users loaded by `store` from the `users.json` file. The `users.json` file contains the user information in the JSON encoded form.

The following code creates a store using an Ajax proxy and `url` pointing to the `users.json` file.

```
var store = Ext.create('Ext.data.Store', {
    model: 'User',
    autoLoad: true,
    proxy: {
      type: 'ajax',
      url : 'users.json',
      reader: {
        type: 'json',
        rootProperty: 'users'
      }
    }
});
```

The proxy uses the JSON reader, which is indicated by the `type` property of `reader`. The other important property on `reader` is `rootProperty`, which needs to be set to the property in the `users.json` file that contains the data array. Hence, it is set to `users`.

Once the `proxy` and `reader` are set up on `store`, the store knows from where it has to load the data (proxy detail) and how the data needs to be interpreted (reader detail) to construct the model. We also need to consider when to load the data. For this, we set the `autoLoad` property on `store` to `true`. This will instruct the store to start loading the data as soon as it is initialized.

More about the proxy and reader is covered in the recipes to follow.

There's more...

There are different properties provided by `proxy` and `reader` to help us deal with different incoming data structures. In the next section, we will see how to deal with some of the data structures, such as nested data and metadata.

Fetching records from a nested data

Say our data contains some metadata about each data such that the actual record is nested. This is shown in the following data format:

```
{"users": [
  {
    "id": "1234",
    "count": "1",
    "user" : {
      "id": "1",
      "name" : "Ajit Kumar",
      "age"  : "32",
      "phone": "555-555-5555",
      "email": "ajit@walkingtree.in"
    }
  }, {
    "id": "1234",
    "count": "1",
    "user" : {
      "id": "2",
      "name" : "Alok Ranjan",
      "age"  : "32",
      "phone": "123-456-7890",
      "email": "alok@walkingtree.in"
    }
  }, {
    "id": "1234",
    "count": "1",
    "user" : {
      "id": "3",
      "name" : "Pradeep Lavania",
      "age"  : "34",
      "phone": "987-654-3210",
      "email": "pradeep@walkingtree.in"
    }
  }]
}
```

To fetch the actual user information out from the preceding structure, we will have to make use of the `record` property on `reader` to indicate the nested field that contains the user information as follows:

```
reader: {
    type: 'json',
    rootProperty: 'users',
    record: 'user'
}
```

Working with response metadata

Sometimes the response contains the metadata and the actual data. These metadata contain application-specific information, which can be used by the client-side code to exhibit certain behaviors. For example, one of the important pieces of information that helps our application to implement pagination is the total record count returned along with the page data so that the frontend would be able to derive the number of pages of data it will have to deal with and, accordingly, render the page information or handle the previous/next functionality. Similarly, the server-side application may have to indicate whether the request was processed successfully or if there was an error. This may be achieved on the server side by returning a property in the metadata and setting it to true/false to indicate success/error. The following code shows the record structure where `totalRecords` and `success` are two metadata properties being returned from the server, besides the actual data, `users`:

```
{
  "totalRecords" : "20",
  "success" : "true",
  "users": [{
    "id": "1",
    "name" : "Ajit Kumar",
    "age"  : "32",
..
  }]
}
```

There are two additional properties provided by `reader`: `totalProperty` and `successProperty` to map the field on the server response, which contains the total number of records available with the server (although it is returning only three in a read) and the field that would indicate whether there was any application level error while processing the request. For example, if the application failed to get users from its database, it can make use of the `success` field to convey the error to the frontend. The following code shows the changes that we will have to make to `reader` to accommodate these two additional metadata properties:

```
reader: {
    type: 'json',
    rootProperty: 'users',
    totalProperty: 'totalRecords',
    successProperty: 'success'
}
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe in *Chapter 2*, *Catering to Your Form-related Needs*
- The *Working with the select field* recipe in *Chapter 2*, *Catering to Your Form-related Needs*
- The *Creating a model* recipe
- The *Working with a store* recipe
- The *Loading data through AJAX using the Ajax proxy* recipe

Converting incoming XML data into the model using XmlReader

Similar to `JsonReader`, `XmlReader` exists for us to work with XML data efficiently. It provides the XPath kind of notation to quickly access the elements of the incoming XML data.

In this recipe, we will see how to work with the XML data and use `XmlReader` to construct the model, which can be used within the application.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. Create and open a new file named `users.xml` and paste the following code into it:

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user>
    <id>1</id>
    <name>Ajit Kumar</name>
    <age>33</age>
    <phone>123-456-7890</phone>
    <email>ajit.kumar@walingtree.in</email>
    <alive>true</alive>
  </user>
  <user>
    <id>2</id>
    <name>Alok Ranjan</name>
    <age>34</age>
    <phone>123-456-7890</phone>
    <email>alok@walkingtree.in</email>
    <alive>true</alive>
  </user>
</users>
```

2. Create and open a new file named `ch05_05.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {
    Ext.define('User', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          {name: 'name',  type: 'string'},
          {name: 'age',   type: 'int'},
          {name: 'phone', type: 'string'},
          {name: 'email', type: 'string'},
          {name: 'alive', type: 'boolean', defaultValue: true}
        ]
      }
    });
    var store = Ext.create('Ext.data.Store', {
      model: 'User',
      autoLoad: true,
      proxy: {
        type: 'ajax',
        url : 'users.xml',
        reader: {
          type: 'xml',
          record: 'user'
        }
      }
    });

    var form;

    var formBase = {
      scrollable: 'vertical',
      items: [{
        xtype: 'selectfield',
        name : 'user',
        label: 'User',
        store: store,
        valueField: 'name',
        displayField: 'name'
      }]
    };

    if (Ext.os.is.Phone) {
      formBase.fullscreen = true;
    } else {
      Ext.apply(formBase, {
        autoRender: true,
        floating: true,
        modal: true,
        centered: true,
        hideOnMaskTap: false,
        height: 385,
        width: 480
      });
    }

    form = Ext.create('Ext.form.FormPanel', formBase);
    Ext.Viewport.add(form);
  }
});
```

3. Include `users.xml` and `ch05_05.js` in the `index.html` file.
4. Deploy and access it from the browser. You may also run it using the emulator.

How it works...

The code loads the data from the `users.xml` file and populates the items in the selection field of the form panel. The `store` is modified to use `proxy` with `url` set to the `users.xml` file and `reader` is configured on proxy with `type` set to `xml`, so that it can interpret the incoming XML data into the model.

```
var store = Ext.create('Ext.data.Store', {
    model: 'User',
    autoLoad: true,
    proxy: {
      type: 'ajax',
      url : 'users.xml',
      reader: {
        type: 'xml',
        record: 'user'
      }
    }
});
```

For the XML reader, we have used the property `record` to tell which element in the incoming XML represents the user information.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe in *Chapter 2*, *Catering to Your Form-related Needs*
- The *Working with the select field* recipe in *Chapter 2*, *Catering to Your Form-related Needs*
- The *Working with a store* recipe
- The *Loading data through AJAX using the Ajax proxy* recipe

Validations in models

A model definition represents the structure of the data that has one or more fields. For example, a payment model containing a `paymentDate` field to store the date when the payment was made. Now, when we construct the models using the incoming data, there may be certain rules that we would like to apply to make sure that the model represents a valid data. For example, on a payment model, it may be required to have a `paymentDate` field and also it may be required that the value in this field is in the past (prior to today's date). This kind of mechanism helps us to build robust applications.

Sencha Touch provides support for this using the validations on `Model`. There are predefined lists of validations that we can use to set up the validation rules on our model. The following are the predefined validations supported:

- `presence`: It validates that a given property value is present, that is, it is not null
- `length`: It validates whether the given value is between the specified min and max
- `inclusion`: It validates that the value is present in the specified list
- `exclusion`: It validates that the value is not present in the specified list
- `format`: It validates that the value matches with the specified regular expression
- `email`: It validates that the value is a valid e-mail format

In this recipe, we will see how to make use of these validations.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. Create and open a new file named `ch05_06.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {
    Ext.define('User', {
      extend: 'Ext.data.Model',
        config: {
          fields: [
            {name: 'name',  type: 'string'},
            {name: 'age',   type: 'int'},
            {name: 'phone', type: 'string'},
            {name: 'email', type: 'string'},
            {name: 'alive', type: 'boolean', defaultValue:
              true}
          ],
          validations: [
            {type: 'presence',  field: 'age'},
            {type: 'length',    field: 'name',     min: 2}
          ]
        }
    });

    var user = Ext.create('User', {
      name : '',
      phone: '555-555-5555',
      email: 'ajitkumar@walkingtree.in'
    });

    var errors = user.validate();
      if (!errors.isValid()) {
        var errStr = '';
        Ext.each(errors.items, function(error, index,
          allErrors){
          errStr += error.getField() + ' : ' +
            error.getMessage() + '\n';
        });
        alert(errStr);
      }
  }
});
```

2. Include `ch05_06.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator.

How it works...

The preceding code sets up the validation rules on the model and validates the model objects using them.

```
validations: [
  {type: 'presence',  field: 'age'},
  {type: 'length',    field: 'name',     min: 2}
]
```

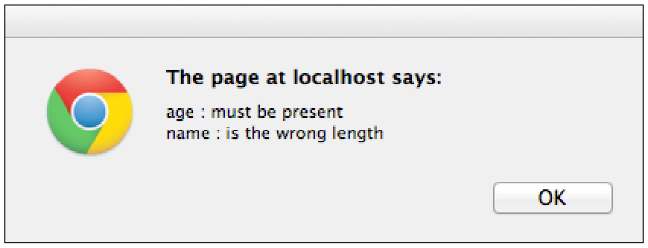
Using the preceding code, we configured two validation rules: `presence` and `length` on the `age` and `name` fields, respectively. The rules suggest that we want to make sure that a model must have the `age` field in it and `name` must be at least two characters long.

```
var errors = user.validate();
```

The preceding line validates the `user` model where it applies all the validations that we had configured on the `User` model. The `validate` method returns `Ext.data.Errors` as the error object. The `errors.isValid()` method returns `true` if the model had passed the validations.

```
if (!errors.isValid()) {
  var errStr = '';
  Ext.each(errors.items, function(error, index, allErrors){
    errStr += error.getField() + ' : ' + error.getMessage() +
      '\n';
  });
  alert(errStr);
}
```

In case of an error, the `validate` method returns an array of error items. Each will have a field indicating the model field that has failed the validation and the corresponding message. In the preceding code, we iterated through the `errors.items` array, concatenated all the error fields and their messages, and displayed them, as shown in the following screenshot:



The page at localhost says:

age : must be present
name : is the wrong length

There's more...

In the following sections, we will see how to make use of the other validations.

Inclusion

Inclusion works with the `list` property, which contains an array of strings. The validation logic for `inclusion` checks if the value is present in the specified list.

The following code shows the typical usage of the `inclusion` validation:

```
{type: 'inclusion',    field: 'gender',   list: ['Male',
  'Female']}
```

Exclusion

Exclusion works as a complement of `inclusion`, where it returns `true` (validation passed) if the value does not belong to the specified list. The usage is exactly the same as `inclusion`, except that the `type` will be `'exclusion'`.

```
{type: 'exclusion', field: 'username', list: ['Admin',
  'Operator']}
```

Format

Format helps us to verify if the value matches with the specified regular expression. We can use the JavaScript regular expressions to create any matchers. This validation rule works on the `matcher` property. The following is a sample usage:

```
{type: 'format',field: 'username', matcher: /([a-z]+)[0-9]{2,3}/}
```

Changing the default message

By default, the `Ext.data.validations` class defines the messages for each type of validation rules. For example, if the `presence` validation fails, **must be present** appears for the field for which it had failed. If the default message is not the desired one, we can change it by specifying the `message` property for the validation, as follows:

```
{type: 'presence',  message: ' property not found', field: 'age'}
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Creating a model* recipe
- The *Validating your form* recipe in *Chapter 2, Catering to Your Form-related Needs*

Defining your custom validation

In the previous recipe, we saw that the out of the box validations are available in Sencha Touch. However, for various practical reasons we may have a need to create additional validation rules and use them across the application. For example, the payment amount must not be negative, the date must be prior to today's date, and so on.

In this recipe, we will go through the steps to create a new validation rule and use it in the application.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. Create and open a new file named `ch05_07.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {

    Ext.apply(Ext.data.validations, {
      checkdate: function(config, value) {
        if (value === undefined) {
          return false;
        }
      var graceDays = Ext.isEmpty(config.grace) ? 1 :
        config.grace;
      var date = Ext.Date.parse(value, 'd-m-Y');
      var currDate = new Date();
      currDate = Ext.Date.clearTime(currDate);
      if (Ext.Date.between(date, currDate,
        Ext.Date.add(currDate, Ext.Date.DAY, graceDays)))
        return true;
      else
        return false;
      },
      getCheckdateMessage: function() {
        return 'date is not within the allowed range';
```

```
          }
        });

        Ext.define('User', {
          extend: 'Ext.data.Model',
          config: {
            fields: [
              {name: 'name',  type: 'string'},
              {name: 'age',   type: 'int'},
              {name: 'phone', type: 'string'},
              {name: 'email', type: 'string'},
              {name: 'effectiveDate', type: 'string'},
                //format d-m-Y
              {name: 'alive', type: 'boolean', defaultValue:
                true}
            ],
            validations: [
              {type: 'presence', field: 'age'},
              {type: 'length',    field: 'name',     min: 2},
              {type: 'checkdate', field: 'effectiveDate',
                grace: 2}
            ]
          }
        });

        var user = Ext.create('User', {
          name : '',
          phone: '555-555-5555',
          email: 'ajit.kumar@walkingtree.in',
          effectiveDate: '18-03-2013'
        });

        var errors = user.validate();
        if (!errors.isValid()) {
          var errStr = '';
          Ext.each(errors.items, function(error, index,
            allErrors){
            errStr += error.getField() + ' : ' +
              error.getMessage() + '\n';
          });
          alert(errStr);
        }
      }
    });
```

2. Include `ch05_07.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator.

How it works...

Consider the following code:

```
Ext.apply(Ext.data.validations, {
```

`Ext.data.validations` contains all the six validation rules and is a singleton class. Using `Ext.apply`, we are adding additional validation to it:

```
getCheckdateMessage: function() {
  return 'date is not within the allowed range';
}
```

`getCheckdateMessage`, once defined, will be used by the `validate` method to show the message in the errors when a field with the new validation, `checkdate`, fails. The syntax for this property is `get<name of the validation rule with first character in uppercase>Message`. So, for our new validation rule, `checkdate`, this has been named `checkdateMessage`.

```
  checkdate: function(config, value) {
    if (value === undefined) {
      return false;
    }

    var graceDays = Ext.isEmpty(config.grace) ? 1 :
      config.grace;

    var date = Ext.Date.parse(value, 'd-m-Y');
    var currDate = new Date();
    currDate = Ext.Date.clearTime(currDate);
    if (Ext.Date.between(date, currDate, Ext.Date.add(currDate,
      Ext.Date.DAY, graceDays)))
      return true;
    else
      return false;

  }
```

The preceding code defines the core logic of the new validation rule `checkdate`. All it is doing is returning `true` if the passed date value is within the specified `grace` days from today. If the function returns `true`, it means the validation has passed. If `false` is returned, then the framework adds the field name and the corresponding message to the `errors` array.

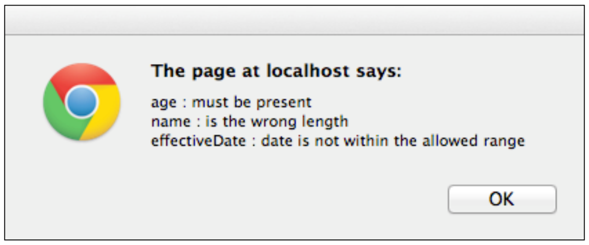
Once the new validation rule is defined, we add it to the `validations` function:

```
{type: 'checkdate', field: 'effectiveDate',   grace: 2}
```

Here we mentioned that the `effectiveDate` must be between today and today + 2 days, and then we passed the `effectiveDate` on the model:

```
effectiveDate: '23-07-2011'
```

When the validation fails, the following errors show up:



The page at localhost says:

```
age : must be present
name : is the wrong length
effectiveDate : date is not within the allowed range
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Creating a model* recipe
- The *Validations in models* recipe

Relating models using association

In an application we generally deal with multiple types of models, for example, user, address, order, payment, account, and so on. Some models are self-sufficient. However, there will be some models that are related to each other and there is an association that exists between them. For example, a user can have one or more addresses, a user may place one or more orders, an order may have one or more payments made against it, a user may have a marital status, and so on. In a typical relational database, we have entities and the relationship between them. The same can be achieved with models using the association mechanism provided by Sencha Touch. This recipe will demonstrate how to define associations between the models, which are used by the reader, internally, to populate nested models for us.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

Caryy out the following steps:

1. Create and open a new file `ch05_08.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',

  launch: function() {
    Ext.define('User', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          {name: 'name',  type: 'string'},
          {name: 'age',   type: 'int'},
          {name: 'phone', type: 'string'},
          {name: 'email', type: 'string'},
          {name: 'alive', type: 'boolean', defaultValue:
            true}

        ],
        hasMany: {model: 'Address', name: 'addresses'}
      }
    });

    Ext.define('Address', {
      extend: 'Ext.data.Model',
      config: {
        fields: ['id', 'line1', 'line2', 'zipcode',
          'state', 'country']
      }
    });

    var user = Ext.create('User', {
      name : 'Ajit Kumar',
      age  : 24,
      phone: '555-555-5555',
      email: 'ajit.kumar@walkingtree.in',
      addresses: [{
        id: 1,
        line1: 'Flat# 101, Plot# 101, Elegance Apartment',
        line2: 'New SBH Colony, East Maredpally,
          Hyderabad',
        zipcode: '500023',
        state: 'AP',
        country: 'India'
      }, {
        id: 2,
        line1: 'Janapriya Utopia',
        line2: 'Hyderguda, Hyderabad',
        zipcode: '500081',
        state: 'AP',
        country: 'India'
      }]
    });

    alert('Number of addresses: ' +
      user.addresses().getCount());
  }

});
```

2. Include `ch05_08.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator.

How it works...

In the preceding code, we defined two models: `User` and `Address` and established `hasMany` association between them such that a user can have multiple addresses. The `hasMany` association helps us establish a one-to-many relationship:

```
hasMany: {model: 'Address', name: 'addresses'}
```

The preceding line in the `User` model definition indicates that the `User` model has the `hasMany` association with the `Address` model where `addresses` is a reference with which we can access the addresses associated with a particular user.

Then we added `addresses` to the user while creating an instance of the `User` model:

```
var user = Ext.create('User', {
    name : 'Ajit Kumar',
    age  : 24,
    phone: '555-555-5555',
    email: 'ajit.kumar@walkingtree.in',
    addresses: [{
      id: 1,
      line1: 'Flat# 101, Plot# 101, Elegance Apartment',
      line2: 'New SBH Colony, East Maredpally, Hyderabad',
      zipcode: '500023',
      state: 'AP',
      country: 'India'
    }, {
      id: 2,
      line1: 'Janapriya Utopia',
      line2: 'Hyderguda, Hyderabad',
      zipcode: '500081',
      state: 'AP',
      country: 'India'
    }]
});
```

`user.addresses()` returns the array of addresses associated with the `User` model.

There's more...

As with `hasMany`, Sencha Touch provides two more association mechanisms named `belongsTo` and `hasOne`. Let's see what they are and where we can use them.

Many-to-one association

`belongsTo` is used to establish a many-to-one association. This is a way to access the parent/owner model from the child. For example, a user can have multiple addresses.
For a user, if we have to get the addresses, we can do this by putting a `hasMany` association between `User` and `Address` in the `User` model. However, if we want to get the corresponding `User` for an address,we can define the `belongsTo` association in the `Address` model with the `User` model. A point to remember in this association is that we must establish a foreign key relationship with the parent/owner model. The following code shows the modified `Address` model with the `belongsTo` association:

```
    Ext.regModel('Address', {
        fields: ['id', 'line1', 'line2', 'zipcode', 'state',
          'country', 'user_id']
        ,belongsTo: 'User'
    });
```

Moreover, when we instantiate the address, we will specify `user_id`, as shown in the following code snippet:

```
{
    id: 1,
    user_id: 1,
    line1: 'Flat# 101, Plot# 101, Elegance Apartment',
    line2: 'New SBH Colony, East Maredpally, Hyderabad',
    zipcode: '500023',
    state: 'AP',
    country: 'India'
}
```

The framework is intelligent enough to generate the `getter` and `setter` methods for us based on the association. On the `address` model, we can use the `getUser()` and `setUser()` methods to work with the model; based on the proxy setup on the model, it will load/save the `user` model for us.

One-to-one association

`hasOne` is used to establish one-to-one association between the models. Let us say there is a `Marital` model that would represent the marital status of a user. The following code shows how we can establish a one-to-one relationship between the `User` and `Marital` models:

```
Ext.define('Marital', {
    extend: 'Ext.data.Model',
    config: {
      fields: [
        'id',
        'status'
      ]
    }
});

Ext.define('User', {
    extend: 'Ext.data.Model',
    config: {
      fields: [
        {name: 'name',  type: 'string'},
        ...,
        hasMany: {model: 'Address', name: 'addresses'},
        hasOne: [
          {
            model: 'Marital'
          }
        ]
    }
});

var user = Ext.create('User', {
    name : 'Ajit Kumar',
    age  : 24,
    phone: '555-555-5555',
    email: 'ajit.kumar@walkingtree.in',
    addresses: [
...
...
    ],
    marital: {
      id: 1,
      status: 'Married'
    }
});
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Creating a model* recipe

Persisting session-specific data using the SessionStorage proxy

So far in this chapter, we have learned how to create a model, store it as a collection in a store, establish relationships, and carry out the validations. However, all this was happening in memory. One page refresh and all our models will be re-initialized and stores reconstructed. It would be a lot better if we could persist them and use them for a longer interval. A proxy provides this persistence capability. Now let's see how to work with the specific proxies to load and save models. The following are types of proxies supported by Sencha Touch:

- `Client`: It helps us to persist a model on the client browser and load it from that storage. Following are the types of `Client` proxies offered by the framework:
    - `Memory`: It uses an in-memory storage
    - `SessionStorage`: It uses HTML5 session storage
    - `LocalStorage`: It uses HTML5 local storage
    - `Sql`: It helps us work with a database

- `Server`: It helps us to persist a model on the server and load it from the remote server. Following are the `Server` proxies offered by the framework:
    - `Ajax`: It is used with the server in the same domain where the application is being accessed.
    - `JsonP`: It is used to connect to a server that is deployed in a domain different from the application domain.
    - `Direct`: It is used to work with the remote functions/methods directly like a remote procedure call. Considering the steps involved with the end-to-end setup and demonstration of direct remoting, discussion of this proxy is out of the scope of this book.

In this recipe, we will see how to make use of the `SessionStorage` proxy to persist the model and restore it from the storage.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

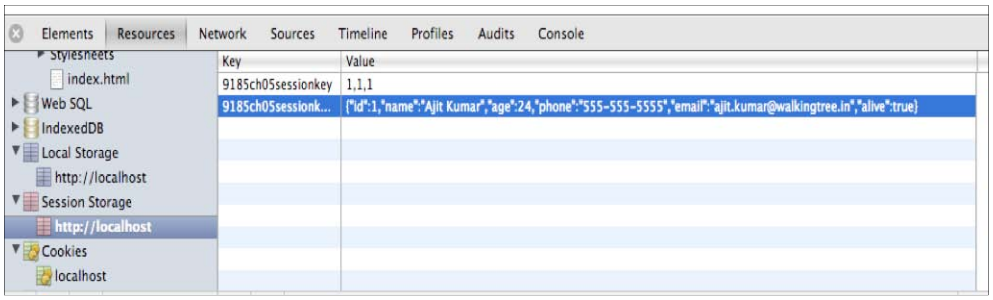
Carry out the following steps:

1. Create and open a new file `ch05_09.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {

    Ext.define('User', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          {name: 'id',  type: 'int'},
          {name: 'name',  type: 'string'},
          {name: 'age',   type: 'int'},
          {name: 'phone', type: 'string'},
          {name: 'email', type: 'string'},
          {name: 'alive', type: 'boolean', defaultValue:
            true}

        ],
        hasMany: {model: 'Address', name: 'addresses'},
        proxy: {
          type: 'sessionstorage',
          id : '9185ch05sessionkey'
        }
      }
    });

    Ext.define('Address', {
      extend: 'Ext.data.Model',
      config: {
        fields: ['id', 'line1', 'line2', 'zipcode',
          'state', 'country', 'user_id'],

        belongsTo: 'User'
      }
    });

    var user = Ext.create('User', {
      id: 1,
      name : 'Ajit Kumar',
      age  : 24,
      phone: '555-555-5555',
      email: 'ajit.kumar@walkingtree.in',
      addresses: [{
        id: 1,
        line1: 'Flat# 101, Plot# 101, Elegance Apartment',
        line2: 'New SBH Colony, East Maredpally,
          Hyderabad',
        zipcode: '500023',
        state: 'AP',
        country: 'India'
      }, {
        id: 2,
        line1: 'Janapriya Utopia',
        line2: 'Hyderguda, Hyderabad',
        zipcode: '500081',
        state: 'AP',
        country: 'India'
      }]
    });

    user.save({
      success: function() {
```

```
            console.log('The User was saved');
        }});

        User = Ext.ModelMgr.getModel('User');
        User.load(1, {
          success: function(record, operation) {
            console.log('The User was loaded');
            alert('Name: ' + record.get('name') + ' : Addresses
              : ' + record.addresses().getCount());
        }});
      }
    });
```

2. Include `ch05_09.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see the following content in the **Session Storage** on the `Resources` tab of Google Chrome:



Key	Value
9185ch05sessionkey	1,1,1
9185ch05sessionk...	{"id":1,"name":"Ajit Kumar","age":24,"phone":"555-555-5555","email":"ajit.kumar@walkingtree.in","alive":true}

How it works...

In the preceding code, we set up `proxy` on the model and configured `sessionstorage` as the `proxy type`, which allows us to persist data in the browser provided HTML5-based `SessionStorage`. `id` is an important property and the value in it must be unique within the session. If the IDs are not unique, we will run into the risk of one part of the application overwriting the data stored by some other part of the application.

```
proxy: {
  type: 'sessionstorage',
  id : '9185ch05sessionkey'
}
```

After `proxy` is set up on the model, it is persisted by calling the `save` method:

```
user.save({
    success: function() {
      console.log('The User was saved');
}});
```

Then, we are loading the persisted model from the `sessionstorage` where the user ID is `1`:

```
User.load(1, {
  success: function(record, operation) {
    console.log('The User was loaded');
    alert('Name: ' + record.get('name') + ' : Addresses : ' +
      record.addresses().getCount());
}});
```

On successful load of the model from `sessionstorage`, the callback registered for `success` is called.

[If this proxy is used in a browser where `sessionstorage` is not supported, the constructor will throw an error.]

There's more...

The code that we saw in the recipe uses the model and the associated proxy to save it in the storage. However, alternatively, we can also use `store` to save the models contained by `store`. In the following section, we will see how to make use of the store to do so.

Working through the Store

To go through the store, replace:

```
//save the model
  user.save({
    success: function() {
      console.log('The User was saved');
  }});
```

With the following:

```
store.add(user);
store.sync();
```

This will ensure that the `user` model is saved in the storage.

Similarly, to read the data from the storage, we can use `store.load()` to read all the stored models:

```
store.load({
  callback: function(records, operation, success) {
    console.log(records);
  }
});
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Creating a model* recipe

Persisting data using the LocalStorage proxy

This recipe describes the usage of HTML5 provided `localstorage`. This persists the data across sessions.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. Create a new file named `ch05_10.js` and copy the content of `ch05_09.js` into it
2. Replace the `proxy` configuration on the `User` model with the following:

```
proxy: {
  type: 'localstorage',
  id : '9185ch05localkey'
}
```

3. Include `ch05_10.js` in the `index.html` file.
4. Deploy and access it from the browser. You may also run it using the emulator. For example, you will see the user information saved in the **Local Storage** on the **Resources** tab of Google Chrome

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Creating a model* recipe
- The *Persisting session-specific data using the SessionStorage proxy* recipe

Accessing in-memory data using the Memory proxy

The simplest form, yet very temporary, is to save and load data in an in-memory variable. In the *Working with Store* recipe of this chapter, we used the inline data to load records in the store. However, that does not utilize `reader`. In order to use the capabilities of the reader, we have to use `proxy` and `Memory` for this purpose. This recipe shows how to use the `Memory` proxy.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. Create and open a new file named `ch05_11.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {
    Ext.define('User', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          {name: 'name',  type: 'string'},
          {name: 'age',   type: 'int'},
          {name: 'phone', type: 'string'},
          {name: 'email', type: 'string'},
          {name: 'alive', type: 'boolean', defaultValue:
            true}
        ]
      }
    });

    var data = {users: [{
      id: 1,
      name : 'Ajit Kumar',
      age  : 32,
      phone: '555-555-5555',
      email: 'ajit@walkingtree.in'
    }, {
      id: 2,
      name : 'Alok Ranjan',
      age  : 32,
      phone: '123-456-7890',
      email: 'alok@walkingtree.in'
    }, {
      id: 3,
      name : 'Pradeep Lavania',
      age  : 34,
      phone: '987-654-3210',
      email: 'pradeep@walkingtree.in'
    }]
    };

    var store = Ext.create('Ext.data.Store', {
      model: 'User',
      autoLoad: true,
      data: data,
      proxy: {
        type: 'memory',
        reader: {
          type: 'json',
          rootProperty: 'users'
        }
      }
    });

    var form;

    var formBase = {
      scroll: 'vertical',
      items: [{
        xtype: 'selectfield',
        name : 'user',
        label: 'User',
        store: store,
        valueField: 'name',
        displayField: 'name'
      }]
    };

    if (Ext.os.is.Phone) {
```

```
          formBase.fullscreen = true;
        } else {
          Ext.apply(formBase, {
            autoRender: true,
            floating: true,
            modal: true,
            centered: true,
            hideOnMaskTap: false,
            height: 385,
            width: 480
          });
        }

        form = Ext.create('Ext.form.FormPanel', formBase);
        Ext.Viewport.add(form);
      }
    });
```

2. Include `ch05_11.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator.

How it works...

Consider the following code:

```
var store = Ext.create('Ext.data.Store', {
    model: 'User',
    autoLoad: true,
    data: data,
    proxy: {
      type: 'memory',
      reader: {
        type: 'json',
        rootProperty: 'users'
      }
    }
});
```

Setting `type` to `memory` sets up the `Memory` proxy on the store. The `Memory` proxy works only on the in-memory data, which we have stored in the `data` variable. As `data` represents a JSON format of the data; we configured the `json` type `reader` and used the `root` to point the property in the data that contains the actual user information.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe in *Chapter 2, Catering to Your Form-related Needs*
- The *Working with the select field* recipe in *Chapter 2, Catering to Your Form-related Needs*
- The *Working with a store* recipe

Loading data through AJAX using the Ajax proxy

In the last three recipes, we saw the usage of the different types of client-side proxies, which help us persist the data on the client browser. Now, we will see how to work with the server proxies to persist the data on a remote server.

In this recipe, we will see what it takes to use the `Ajax` proxy to persist and load the model.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1, Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. Create and open a new file named `ch05_12.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {

    Ext.define('User', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          {name: 'name',  type: 'string'},
          {name: 'age',   type: 'int'},
          {name: 'phone', type: 'string'},
          {name: 'email', type: 'string'},
          {name: 'alive', type: 'boolean', defaultValue:
            true}
        ]
      }
    });

    var store = Ext.create('Ext.data.Store', {
      model: 'User',
      autoLoad: true,
      proxy: {
        type: 'ajax',
        url: 'users.json',
        reader: {
          type: 'json',
          rootProperty: 'users'
        }
      }
    });

    var form;

    var formBase = {
      scroll: 'vertical',
      items: [{
        xtype: 'selectfield',
        name : 'user',
        label: 'User',
        store: store,
        valueField: 'name',
        displayField: 'name'
      }]
    };

    if (Ext.os.is.Phone) {
      formBase.fullscreen = true;
    } else {
      Ext.apply(formBase, {
        autoRender: true,
        floating: true,
        modal: true,
        centered: true,
        hideOnMaskTap: false,
        height: 385,
        width: 480
      });
    }

    form = Ext.create('Ext.form.FormPanel', formBase);
    Ext.Viewport.add(form);
  }
});
```

2. Include `ch05_12.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator.

How it works...

Consider the following code:

```
var store = Ext.create('Ext.data.Store', {
  model: 'User',
  autoLoad: true,
  proxy: {
    type: 'ajax',
    url: 'users.json',
    reader: {
      type: 'json',
      rootProperty: 'users'
    }
  }
});
```

Setting `type` to `ajax` sets up the `Ajax` proxy on the store. The `Ajax` proxy works only if the specified URL is in the domain in which the application is running. `users.json` contains the JSON formatted data that we saw in the *Converting incoming JSON data into a model using JsonReader* recipe in this chapter. As `data` represents a JSON format of the data, we configured the `json` type `reader` and used `root` to point the property in `data` which contains the actual user information.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Getting your form ready with form panels* recipe in *Chapter 2, Catering to Your Form-related Needs*

- The *Working with the select field* recipe in *Chapter 2*, *Catering to Your Form-related Needs*
- The *Working with a store* recipe
- The *Converting incoming JSON data into a model using JsonReader* recipe

Sorting data

The store supports filters, sorting, and grouping. These are very important functionalities, that make the Sencha Touch data classes so useful. One can sort data on one or more fields, apply one or more filters, and group the data on certain fields. All this is available on the client side as well as server side. On the client side, the framework applies the sorting, filtering, and grouping on the models stored within a store whereas, on the server side, the information is passed to the remote server so that the server side application/script can handle them and provide the desired sorted, filtered, and grouped data.

In this recipe, we will see how to sort the data, send the sorting information to the server, and customize the information sent to the server.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

Add the following code to `ch05_12.js` after we have defined the `User` model:

```
var store = Ext.create('Ext.data.Store', {
  model: 'User',
  autoLoad: true,
  proxy: {
    type: 'ajax',
    url: 'users.json',
    reader: {
      type: 'json',
      rootProperty: 'users'
    }
  }
});
```

```
store.sort([
  {
    property : 'age',
    direction: 'DESC'
  },
  {
    property : 'name',
    direction: 'ASC'
  }
]);
```

How it works...

The code in this recipe is based on the previous recipe, *Loading data through AJAX using the Ajax proxy*. In the preceding code, we are using the `Ajax` proxy to load data from the `users.json` file and using the `sort` method, we are sorting the data in the store:

```
store.sort([
  {
    property : 'age',
    direction: 'DESC'
  },
  {
    property : 'name',
    direction: 'ASC'
  }
]);
```

The `sort` method accepts an object containing `property` and sort `direction`. `property` instructs the model field on which record the sorting should be done and the `direction` instructs whether the records should be sorted in ascending or descending order. The `direction` name is optional and the framework sorts the data in ascending order if it is not specified for a property. Also, the direction name is case-sensitive and we should always use the uppercase versions.

As we have added sorting on two fields, `age` and `name`, the order of sorting is the order in which the sorting information is added. Therefore, in our case, the records will be first sorted on `age` and then on `name`.

There's more...

Alternatively, the sorting can happen on the server side. This is generally done in case the UI application does not have the complete data to apply the sorting on the client end or we want to leverage the server capability for better performance. Let's see how to enable the server-side sorting and send the sorting information to the server so that the server-side code can return the sorted data using the specified information.

Sending the sorting information to the server

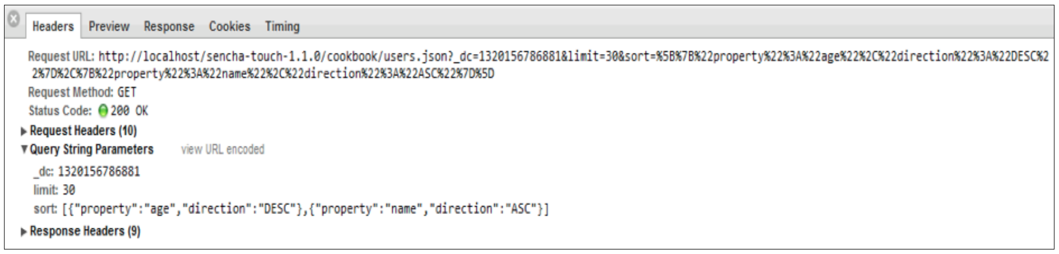
We send the sorting information to the server so that the server-side application can sort the data before returning it to the client-side application. To ask the framework to send the sorting information to the server, we will have to set `remoteSort` to `true` on the store.

```
var store = Ext.create('Ext.data.Store', {
  model: 'User',
  remoteSort: true,
  proxy: {
      type: 'ajax',
      url: 'users.json',
      reader: {
        type: 'json',
        rootProperty: 'users'
      }
  }
});
```

We need to call the `load` method on `store` to send the sorting information to the backend server.

```
store.load();
```

Once this is set, the sorting information will be passed as part of the query parameter as shown in the following screenshot:



```
Request URL: http://localhost/sencha-touch-1.1.0/cookbook/users.json?_dc=1320156786881&limit=30&sort=%5B%7B%22property%22%3A%22age%22%2C%22direction%22%3A%22DESC%22%7D%2C%7B%22property%22%3A%22name%22%2C%22direction%22%3A%22ASC%22%7D%5D
Request Method: GET
Status Code: 200 OK
▶ Request Headers (10)
▼ Query String Parameters
  _dc: 1320156786881
  limit: 30
  sort: [{"property":"age","direction":"DESC"},{"property":"name","direction":"ASC"}]
▶ Response Headers (9)
```

Customizing the sort information being sent to the server

By default, the sort information is sent to the server using the parameter named `sort`. If we want to change this default, we shall use the `sortParam` property on `proxy` to set it to the desired name, say, `searchCritera`.

In case we don't want the sorting information to be sent to the server, we can achieve it by setting the `sortParam` property to `undefined`.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with a store* recipe
- The *Loading data through AJAX using the Ajax proxy* recipe

Data grouping

In this recipe, we will see how to group the data and how to send the grouping information to the server application.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

Pass the additional properties to the `store` instance in `ch05_12.js` as highlighted in the following code snippet:

```
var store = Ext.create('Ext.data.Store', {
  model: 'User',
  autoLoad: true,
  remoteGroup: true,
  groupField: 'age',
  groupDir: 'DESC',
  proxy: {
    type: 'ajax',
    url: 'users.json',
    reader: {
      type: 'json',
      rootProperty: 'users'
    }
  }
});
```

How it works...

The preceding code shows how to specify the grouping information and send it to the server. The related properties are `groupField` and `groupDir`. The `groupField` property instructs the model field on which the data needs to be grouped and the `groupDir` property instructs the direction, ascending or descending. Grouping information is treated in a similar way to sort operations. Grouping information is passed as the first sort information.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with a store* recipe
- The *Loading data through AJAX using the Ajax proxy* recipe

Filtering data

Filtering is a great way to remove unwanted records based on certain criteria. A store allows us to specify the filters and additional properties to send the filter information to the server application. One or more filters can be applied. In this recipe, we will see how to do this.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

How to do it...

Pass the additional properties to the `store` instance in `ch05_12.js` as highlighted in the following code snippet:

```
var store = new Ext.data.Store({
  model: 'User',
  autoLoad: true,
  proxy: {
    type: 'ajax',
    url: 'users.json',
    reader: {
      type: 'json',
      rootProperty: 'users'
    },
    remoteFilter: true
  }
});

  store.filter([
    {
      property: 'name',
      value   : /Aj/
    }
  ]);
```

How it works...

The code in this recipe is based on the previous recipe, *Loading data through AJAX using Ajax proxy*. In the preceding code, we are using the `Ajax` proxy to load data from the `users.json` file and using the `filter` method, we are filtering the data in `store`:

```
store.filter([
  {
    property: 'name',
    value   : /Aj/
  }
]);
```

The `filter` method accepts an object containing `property` and `value` to compare with. `property` instructs the model field on which the filtering should be done and `value` instructs the value/pattern, which shall be used to filter the records.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with a store* recipe
- The *Loading data through AJAX using the Ajax proxy* recipe

Using a cross-domain URL in your application

Besides `Ajax`, `JsonP` is the other server proxy that helps us to persist the model on a remote server and load it from the same. The only catch is that this proxy is used only when the domain where the server is running is different from the domain where the application is running; for example, loading the search detail from the Google Custom search API. This recipe outlines the usage of the `JsonP` proxy to make cross-domain URL calls.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

Get the API key for Google's Custom Search by following the instructions available at `https://developers.google.com/custom-search/v1/using_rest`.

How to do it...

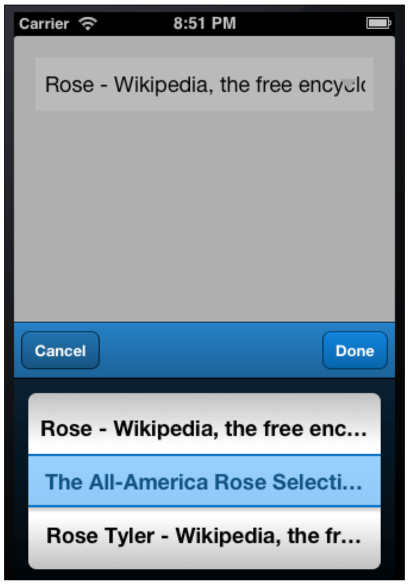
Carry out the following steps:

1. Create and open a new file named `ch05_13.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    Ext.define('SearchResult', {
      extend: 'Ext.data.Model',
      config: {
        fields : [ {name : 'kind', type : 'string'},
          {name : 'title',type : 'string'},
          {name : 'htmlTitle',type : 'string'},
          {name : 'displayLink',type : 'string'},
          {name : 'snippet',type : 'boolean',defaultValue :
            true}
        ]
      }
    });

    var store = Ext.create('Ext.data.Store', {
      model : 'SearchResult',
      autoLoad : true,
      proxy : {
        type : 'jsonp',
        pageParam: null,
        startParam: null,
        limitParam: null,
        noCache: false,
        url : 'https://www.googleapis.com/customsearch/
          v1?key='+ 'AIzaSyD8nxb7bFwURb6gXqHWz9dFMQw8-
            bZCvPw'+ '&cx=013036536707430787589:
              _pqjad5hr1a&q=rose&alt=json',
        reader : {
          type : 'json',
          rootProperty : 'items'
        }
      }
    });

    var form;

    var formBase = {
      scrollable : 'vertical',
      items : [ {
        xtype : 'selectfield',
        name : 'user',
        store : store,
        valueField : 'title',
        displayField : 'title'
      } ]
    };

    if (Ext.os.is.Phone) {
```

```
          formBase.fullscreen = true;
        } else {
          Ext.apply(formBase, {
            autoRender : true,
            floating : true,
            modal : true,
            centered : true,
            hideOnMaskTap : false,
            height : 385,
            width : 480
          });
        }

        form = Ext.create('Ext.form.FormPanel', formBase);
        Ext.Viewport.add(form);
      }
    });
```

2. Include `ch05_13.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see the screen as shown in the following screenshot:


Carrier
8:51 PM
Rose - Wikipedia, the free encyclo
Cancel
Done
Rose - Wikipedia, the free enc...
The All-America Rose Selecti...
Rose Tyler - Wikipedia, the fr...

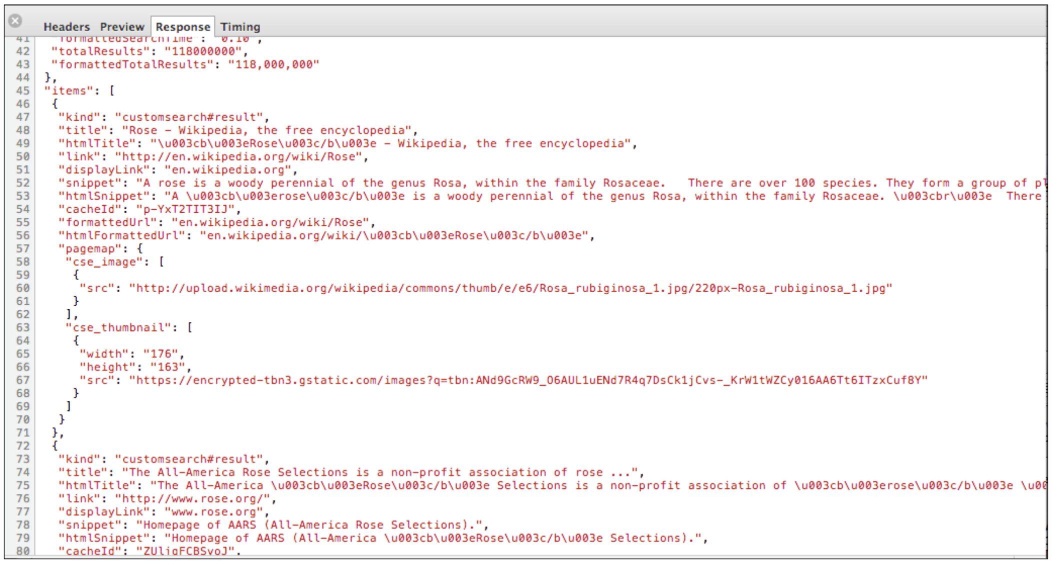
How it works...

In the preceding code, we have used the `jsonp` proxy to read search data from Google's search API.

```
var store = Ext.create('Ext.data.Store', {
  model : 'SearchResult',
  autoLoad : true,
  proxy : {
    type : 'jsonp',
    pageParam: null,
    startParam: null,
    limitParam: null,
    noCache: false,
    url : 'https://www.googleapis.com/customsearch/v1?key='
      + 'AIzaSyD8nxb7bFwURb6gXqHWz9dFMQw8-bZCvPw'
        + '&cx=013036536707430787589:
          _pqjad5hr1a&q=rose&alt=json',
    reader : {
      type : 'json',
      rootProperty : 'items'
    }
  }
});
```

By default, the proxy sends paging information: `start`, `limit`, and `page`, as part of the query string, which causes issues with the Google's API. Also, in Sencha Touch, script caching is disabled, by default. This causes a `_dc` parameter to be sent as part of the query string. We set `limitParam`, `startParam`, and `pageParam` to `null` so that they are not sent to the URL. And we, enable caching by setting `noCache` to `false`, which will instruct the framework not to send the `_dc` parameter to the URL.

Google API returns data in the form of JSON. The following screenshot shows the response received from the Google API call:



```
  "formattedSearchTime": "0.10",
  "totalResults": "118000000",
  "formattedTotalResults": "118,000,000"
 },
 "items": [
  {
   "kind": "customsearch#result",
   "title": "Rose - Wikipedia, the free encyclopedia",
   "htmlTitle": "\u003cb\u003eRose\u003c/b\u003e - Wikipedia, the free encyclopedia",
   "link": "http://en.wikipedia.org/wiki/Rose",
   "displayLink": "en.wikipedia.org",
   "snippet": "A rose is a woody perennial of the genus Rosa, within the family Rosaceae.   There are over 100 species. They form a group of pl
   "htmlSnippet": "A \u003cb\u003erose\u003c/b\u003e is a woody perennial of the genus Rosa, within the family Rosaceae. \u003cbr\u003e  There
   "cacheId": "p-YxT2TIT3IJ",
   "formattedUrl": "en.wikipedia.org/wiki/Rose",
   "htmlFormattedUrl": "en.wikipedia.org/wiki/\u003cb\u003eRose\u003c/b\u003e",
   "pagemap": {
    "cse_image": [
     {
      "src": "http://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/Rosa_rubiginosa_1.jpg/220px-Rosa_rubiginosa_1.jpg"
     }
    ],
    "cse_thumbnail": [
     {
      "width": "176",
      "height": "163",
      "src": "https://encrypted-tbn3.gstatic.com/images?q=tbn:ANd9GcRW9_O6AUL1uENd7R4q7DsCk1jCvs-_KrW1tWZCy016AA6Tt6ITzxCuf8Y"
     }
    ]
   }
  },
  {
   "kind": "customsearch#result",
   "title": "The All-America Rose Selections is a non-profit association of rose ...",
   "htmlTitle": "The All-America \u003cb\u003eRose\u003c/b\u003e Selections is a non-profit association of \u003cb\u003erose\u003c/b\u003e \u00
   "link": "http://www.rose.org/",
   "displayLink": "www.rose.org",
   "snippet": "Homepage of AARS (All-America Rose Selections).",
   "htmlSnippet": "Homepage of AARS (All-America \u003cb\u003eRose\u003c/b\u003e Selections).",
   "cacheId": "ZUliqFCBSvoJ",
```

`items` contains the actual response data that we are interested in. Hence, the `reader` is configured with `type` set to `json` and `items` as `rootProperty`.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Getting your form ready with f orm panels* recipe in *Chapter 2, Catering to Your Form-related Needs*
- The *Working with the select field* recipe in *Chapter 2, Catering to Your Form-related Needs*
- The *Working with a store* recipe

Working with Web SQL databases

Besides `LocalStorage` and `SessionStorage`, Sencha Touch offers an additional client-side proxy named `Sql`, to manage the data in a JavaScript relational database such as `Sqllite`. The proxy provides the same APIs to save or load a model. However, internally, it creates the appropriate SQL query to carry out its task. In this recipe, we will see how to work with the `Sql` proxy.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch05` folder created inside the `www` folder.

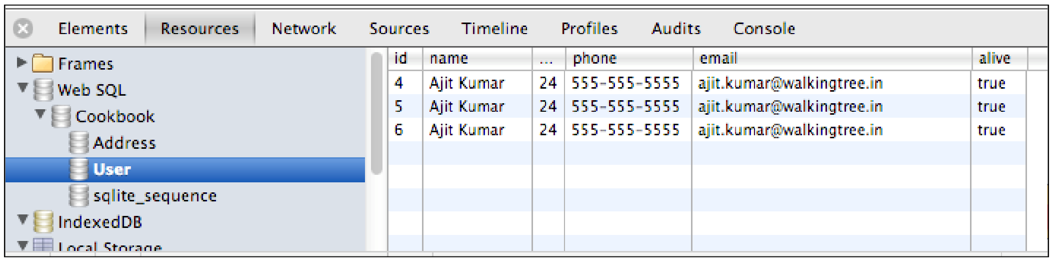
How to do it...

Carry out the following steps:

1. Create and open a new file named `ch05_14.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {

    Ext.define('User', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          {name: 'id',  type: 'int'},
          {name: 'name',  type: 'string'},
          {name: 'age',   type: 'int'},
          {name: 'phone', type: 'string'},
          {name: 'email', type: 'string'},
          {name: 'alive', type: 'boolean', defaultValue:
            true}

        ],
        hasMany: {model: 'Address', name: 'addresses'},
        proxy: {
          type: 'sql',
          database: 'Cookbook',
          model : 'User'
        }
      }
    });

    Ext.define('Address', {
      extend: 'Ext.data.Model',
      config: {
        fields: ['id', 'line1', 'line2', 'zipcode',
          'state', 'country', 'user_id'],

        belongsTo: 'User',
        proxy: {
          type: 'sql',
          database: 'Cookbook',
          model : 'Address'
        }
      }
    });


    var user = Ext.create('User', {
      name : 'Ajit Kumar',
      age  : 24,
      phone: '555-555-5555',
      email: 'ajitkumar@walkingtree.in'
    });

    //create a user in the database
    user.save({
      success: function(record) {
        console.log('The User was saved');

        var userId = record.get('id');

        //create an address in the database for the created
          user
        var addr = Ext.create('Address', {
          user_id: userId,
          line1: 'Flat# 101, Plot# 101, Elegance
            Apartment',
          line2: 'New SBH Colony, East Maredpally,
            Hyderabad',
          zipcode: '500023',
          state: 'AP',
          country: 'India'
        });

        addr.save({
          success: function() {
            console.log('The Address was saved');
            User = Ext.ModelMgr.getModel('User');
            //load the recently added user from database
            User.load(userId, {
              success: function(record, operation) {
                console.log('The User was loaded');
                alert('Name: ' + record.get('name') + ' :
                  Addresses : ' + record.addresses()
                    .getCount());
              }});
          }
        });
    }});
  }
});
```

2. Include `ch05_14.js` in the `index.html` file.
3. Deploy and access it from the browser.
4. Go to the **Resources** tab on the Chrome Developer to see that the `Cookbook` database is created under `Web SQL` with two tables namely `User` and `Address`; the data is stored in those tables, as shown in the following screenshot:



id	name	...	phone	email	alive
4	Ajit Kumar	24	555-555-5555	ajit.kumar@walkingtree.in	true
5	Ajit Kumar	24	555-555-5555	ajit.kumar@walkingtree.in	true
6	Ajit Kumar	24	555-555-5555	ajit.kumar@walkingtree.in	true

How it works...

In the preceding code, we have used the `sql` proxy to save data in the JavaScript database. The proxy uses the browser supported `Web SQL` implementation of the W3 specification outline available at `http://www.w3.org/TR/webdatabase/`.

The proxy, by default, creates a database by the name `Sencha`. As we wanted to have a different database name, we set the `database` config on the proxy. Also, the proxy creates tables and columns based on the model that we set on it.

You must verify the support for the Web SQL standard in your browser/emulator/device before testing this.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Creating a model* recipe

6
Adding Components

In this chapter we will cover:

- Working with the button component
- Creating a sheet of buttons with ActionSheet
- Carousel
- Managing a list of data using List
- Grouping items in a list
- Navigating through a list of data using IndexBar
- Working with a list of nested data using NestedList
- Picking up your choice using Picker
- Switching between multiple views using SegmentedButton
- Working with Tab panels
- Getting quicker access to application features using Toolbar
- Creating a new component
- Extending an existing component's capability
- Overriding a component's behavior
- Adding behavior to an existing component using plugins

Introduction

So far, we have seen the usage of various components such as `FormPanel`, `DataView`, and `Panel`. There were some components that we had used in the previous recipes, such as `toolbar`, but not discussed in detail. Besides, there are some more components that are worth discussing to understand the purpose of their existence and use them accordingly. Also, this chapter goes beyond the existing components and covers how to create a new component, extend an existing component, and build plugins and use them in enhancing the capabilities of a component.

Working with the button component

This recipe introduces the `button` component and shows how to make use of the button in our applications, how to have a different look and feel, and handle user action.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch06`, in the `www` folder where we had created the `ch01` and `ch02` folders. We will be using this new folder in which the code will be kept.

How to do it...

Carry out the following steps:

1. Create and open a new file named `ch06_01.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',

  launch: function() {

    var buttons = [
      {
        text: 'Normal',
        handler: function() {
          Ext.Msg.alert('Info', 'You have clicked: ' +
            this.getText());
        }
      },
      {
        ui   : 'round',
        text: 'Round'
      },
      {
        ui   : 'small',
        text: 'Small'
      }
    ];

    Ext.create('Ext.Panel', {
      fullscreen: true,
      layout: {
        type : 'hbox',
        pack : 'center',
        align: 'middle'
      },
      defaults: {
        xtype: 'button'
      },
      items: buttons
    });
  }
});
```

2. Include `ch06_01.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator.

How it works...

The preceding code creates a panel with three buttons laid out using the `hbox` layout. The following code sets the `xtype` property for all the items of the panel to `button`:

```
defaults: {
    xtype: 'button'
},
```

This shortcut allows us to set `xtype` for all the child items in one go.

The `ui` property defines the look and feel of a button. This acts as a shortcut to a group of CSS styles in the Sencha Touch theme.

We have specified an inline function as `handler` on the button. This function is called when a button is clicked or tapped.

There's more...

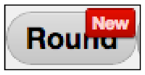
Additionally, the `button` component provides the mechanism to specify the badge and also use the icons along with the text. The following section describes how to make use of these functionalities.

Using badges

A badge is the text that appears on top of the button. This may be useful to highlight a button; for example, if the badge has the text **New**, it will indicate to the user that this button is newly added. This helps us grab the user's attention. The following code snippet shows how to render a badge with the text **New** on the button:

```
{
        ui   : 'round',
        text: 'Round',
        badgeText: 'New'
}
```

When we run the code, we will see the **Round** button with a badge having the text **New**, as shown in the following screenshot:



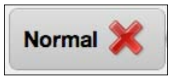
The framework uses the predefined CSS to show the badge. In case you want to define and use a different style, you can do so by setting the `badgeCls` property on the button.

Using icons

It is generally considered good practice to use an icon along with the text while creating a button, as it gives both textual as well as visual meaning to it icon. People who have difficulty in reading and understanding text may find it easier to remember the. For this, the `button` component supports multiple properties: `icon`, `iconCls`, and `iconAlign`. In case you want to use an image directly as an icon, you can do it by setting the `icon` property. However, it is better if we define a CSS class and use it. For this, we shall use the `iconCls` property. The following code snippet shows the usage of these properties:

```
{
        ui   : 'normal',
        text: 'Normal',
        //icon: 'ch06/delete.png',
        iconCls: 'cancel-icon',
        iconAlign: 'right'
}
```

iconAlign allows us to align the icon with respect to the text. The valid values are top, bottom, right, and left; left is the default alignment. The following screenshot shows how the icon will appear on the button:



cancel-icon is defined in the ch06.css file as shown in the following code snippet:

```
.cancel-icon {
  background: url(images/delete.png) no-repeat;
}
```

Make sure ch06.css is included in the index.html file before ch06_01.js.

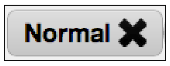
Using pictos icons

Pictos icons are a great way to show icons because they gel very well with the application theme. It works using webkit-mask-image. The black parts of the image hides what it is over, the white parts of the image show what is underneath, and the gray is partially transparent.

You will have to use the iconMask property along with the iconCls property to use one of the predefined pictos icons.

```
{
    ui   : 'normal',
    text: 'Normal',
    iconCls: 'delete',
    iconMask: true,
    iconAlign: 'right'
}
```

The list of supported iconCls properties is documented in the API documentation of the Ext.Button class. The following shows the output that we will see when the iconMask property is used:

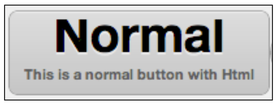


Using custom HTML as button content

In the preceding example, we have used the `text` property to specify the label for the button, which is used to render the button body. Along with that, we have the icon-related properties and their positions, which we can use to show a button. If you want to show custom content inside the button body, we can use the `html` config available on the `button` component. The following code snippet shows how we have created a button with two lines of text using the `html` config:

```
{
        ui  : 'normal',
        //text: 'Normal',
        html: '<span class="first-line">Normal</span><p></p><span
class="second-line">This is a normal button with Html</span>'
}
```

The following screenshot shows how the button will look:



The CSS `first-line` and `second-line` classes are defined in `ch06.css` as shown in the following code snippet:

```
.first-line {
    font-size: 2.1em;
    text-shadow: 0px 2px 0px rgba(255,255,255,.5);
}

.second-line {
    font-size: 0.6em;
    color:#555;
}
```

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1, Gear Up for the Journey*

Creating a sheet of buttons with ActionSheet

In an application, multiple actions can be performed on an entity. For example, on an inbox item, a user can reply to the sender, reply to all, delete the e-mail, and view the complete mail. Moreover, these actions may vary based on the entity in the context. To handle this kind of scenario, the Sencha Touch framework provides the `ActionSheet` component, which allows us to show a sheet of buttons that can help the user trigger different actions. This recipe shows us how to create `ActionSheet` and use it in an application.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

How to do it...

Carry out the following steps:

1. Create and open a new file named `ch06_02.js` and copy the contents of `ch04/ch04_07.js` inside it.
2. Add the following code at before the `launch` function:

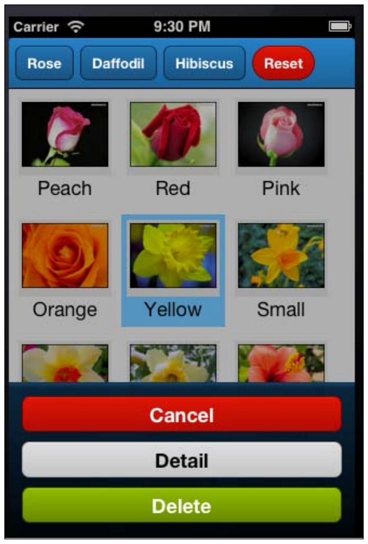
```
var actionSheet = Ext.create('Ext.ActionSheet', {
hidden: true,
  items: [
    {
      text: 'Cancel',
      ui  : 'decline',
      handler: function() {
        actionSheet.hide();
      }
    },
    {
      text: 'Detail',
      handler: function() {
        var recs = selRecs;
        Ext.getCmp('detail-panel').setHtml('<img src="' +
          recs[0].data.url + '" title="' +
            recs[0].data.title + '">');
        Ext.getCmp('images-view').setActiveItem(1);
        Ext.getCmp('back-button').show();
        Ext.getCmp('rose-button').hide();
        Ext.getCmp('daffodil-button').hide();
        Ext.getCmp('hibiscus-button').hide();
        actionSheet.hide();
      }
    },
    {
      text: 'Delete',
      ui  : 'confirm',
      handler: function(){
        Ext.Msg.confirm("Confirmation", "Are you sure you
          want to delete the picture?", function(btn){
            if (btn == "yes") {
              var dview = Ext.getCmp('images-data-view');
              var recs = selRecs;
              dview.getStore().remove(recs);
            }
            actionSheet.hide();
        });
      }
    }
  ]
});
```

3. Change the `selectionchange` handler as per the following code:

```
selectionchange: function(model, recs) {
  if (recs.length > 0) {
    selRecs = recs;
    actionSheet.show();
  }
},
```

4. Include the `ch04/ch04.css` file in the `index.html` file.
5. Include the `ch06_02.js` file in the `index.html` file.

6. Deploy and access it from the browser. You can also run it using the emulator. You will see the flowers on the screen; when you click/tap on a particular flower, you will see the buttons docking in from the bottom, as shown in the following screenshot:



How it works...

The preceding code creates a sheet of three buttons, namely, **Cancel**, **Detail**, and **Delete**. The `DataView` component shows the photos; when a user selects a photo, the `selectionchange` event is fired and its handler shows the button sheet to the user by calling the `show` method on the `ActionSheet` instance.

When the **Cancel** button is clicked on, we will hide the button sheet by calling the `hide` method. When the user clicks on the **Delete** button, the following handler code seeks user confirmation; upon confirmation, it removes the selected photo from the view's store.

```
Ext.Msg.confirm("Confirmation", "Are you sure you want to
  delete the picture?", function(btn){
  if (btn == "yes") {
    var dview = Ext.getCmp('images-data-view');
    var recs = selRecs;
    dview.getStore().remove(recs);
  }
  actionSheet.hide();
});
```

When the user clicks on the **Detail** button, the handler shows the bigger image of the selected photo, updates the toolbar to show the appropriate buttons, and hides the sheet.

There's more...

By default, the sheet appears at the bottom of the viewport and it slides in and out when it is shown or hidden. Let's see how to change these defaults.

Changing the position and animation

The `ActionSheet` container provides different properties to control these defaults.

- `enter`: It is the viewport side from which the sheet is anchored.
- `exit`: It is the viewport side used as the exit point when the sheet is hidden. This is applicable only for the `slide` animation, which is the default animation.
- `showAnimation`: The animation to be used when the sheet is being shown.
- `hideAnimation`: The animation to be used when the sheet is being hidden.

The following code snippet shows the usage of these fields to make sure that the sheet enters from the left-hand side and the animation it uses is "fade":

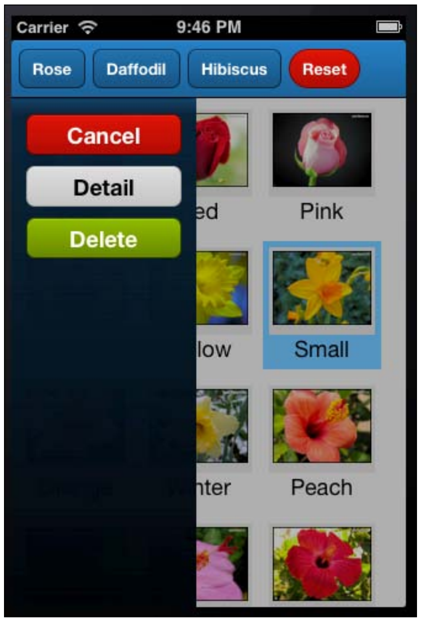
```
var actionSheet = Ext.create('Ext.ActionSheet', {
  hidden: true,
  enter: 'left',
  showAnimation: 'fadeIn',
  hideAnimation: 'fadeOut',
  items: [
    {
...
...
  }]
});
```

ActionSheet as a cross-cut menu

To display the `ActionSheet` container as a cross-cut menu, we will set the following additional configs on `ActionSheet`:

```
var actionSheet = Ext.create('Ext.ActionSheet', {
  hidden: true,
  width: 150,
  top: 0,
  enter: 'left',
  exit : 'left',
  hideOnMaskTap : true,
        items: [
          {
    ...
    ...
        }]
      });
```

The `enter` and `exit` configs indicate the position of the sheet when it is being shown and hidden. We set `top` to `0` so that the sheet occupies the complete container height. We fixed the `width` property to `150` so that it does not occupy the complete container width. `hideOnMaskTap` is set to `true` so that the sheet is hidden even if the user taps anywhere outside the `ActionSheet` container.



If you want to align the button in the middle, you can set the `layout` config in `ActionSheet` as follows:

```
    layout: {
      type: 'vbox',
       pack: 'middle'
    }
```

Using items other than buttons

By default, the `xtype` property for child items of `ActionSheet` is `button`. We can use other `xtype` properties as the default type by setting the `defaultType` config. For example, `defaultType: 'panel'` will indicate that every child item is of type `panel`. This is useful only if we pass the configuration object as the child item. If we pass the instances of objects as the child items, `defaultType` is ignored and the `type` property of the object instance is used instead. This allows us to use components other than buttons as the child items of the sheet.

See also

- The *Setting up a browser based-development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- *Chapter 4*, *Building Custom Views*

Carousel

The carousel is an extension of `Ext.Container` and provides the ability to slide back and forth between different child items. The carousel, internally, uses the `card` layout to render items and allows the user to slide back and forth by setting the active item appropriately.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

How to do it...

Carry out the following steps:

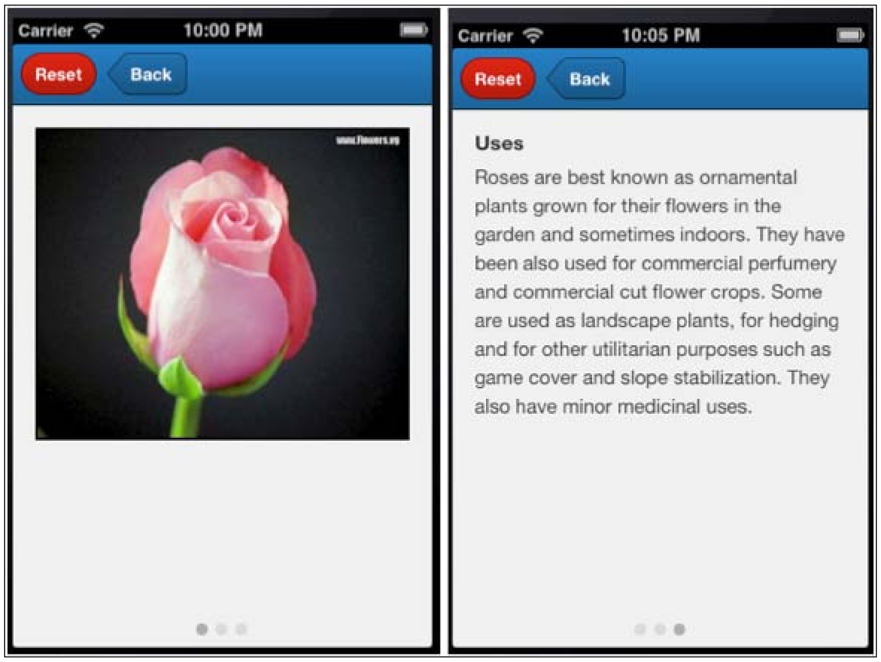
1. Create and open a new file named `ch06_03.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',

  launch: function() {

    var selRecs;

    var actionSheet = Ext.create('Ext.ActionSheet', {
```

```
  ...... //code is same as the one in - Creating a
    sheet of buttons with ActionSheet - recipe
});

var data = [{
  album:'rose',
  url:'http://images.flowers.vg/250x300/rdroses01.jpg',
  title:'Rose 1',
  about:'Peach'
},
  ...... //code is same as the one in - Creating a
    sheet of buttons with ActionSheet - recipe
{
  album:'hibiscus',
  url:'http://images.flowers.vg/250x300/hibiscus-red-
    bright.jpg',
    title:'Hibiscus 1',
    about:'Bright Red'
}];

var store = Ext.create('Ext.data.Store', {
  data: data,
  fields: [
    'url', 'title','about', 'album'
  ]
});

var filter = function(criteria) {
  store.clearFilter();
  return store.filterBy(function(record, id){
    if (record.get('album') === criteria ||
      Ext.isEmpty(criteria))
      return true;
    else
      return false;
  });
};

var tpl = new Ext.XTemplate(
  '<div class="thumb-wrap" id="{title}">',
  '<div class="thumb"><img src="{url}"
    title="{title}"></div>',
  '<span>{about}</span></div>',
    '<div class="x-clear"></div>');
  );
  var carousel = Ext.create('Ext.carousel.Carousel', {
    items: [{
      id: 'detail-panel',
      styleHtmlContent: true,
      scrollable: 'vertical'
    },
    {
      styleHtmlContent: true,
      html: '<h1 style="font-size:16px;"><b>About
        Roses</b></h1><p>The leaves are borne alternately
        on the stem. In most species they are 5 to 15
        centimetres (2.0 to 5.9 in) long, pinnate, with
        (3ñ) 5ñ9 (ñ13) leaflets and basal stipules;
        the leaflets usually have a serrated
        margin, and often a few small prickles on
        the underside of the stem. Most roses
        are deciduous but a few (particularly
        from South east Asia) are evergreen
        or nearly so.</p>'
    },
    {
      styleHtmlContent: true,
      html: '<h1 style="font-size:16px;"><b>Uses</b>
        </h1><p>Roses are best known as ornamental plants
        grown for their flowers in the garden and
        sometimes indoors. They have been also used
        for commercial perfumery and commercial cut
        flower crops. Some are used as landscape
        plants, for hedging and for other
        utilitarian purposes such as game
        cover and slope stabilization. They
        also have minor medicinal uses.</p>'
    }]
  });

  Ext.create('Ext.Panel', {
    id:'images-view',
    fullscreen: true,
    layout: 'card',
    items: [actionSheet, Ext.create
      ('Ext.dataview.DataView', {
```

```
                id: 'images-data-view',
                ... //code is same as the one in - Creating a
                  sheet of buttons with ActionSheet - recipe
              }), carousel, {
                xtype: 'toolbar',
                docked: 'top',
                items: [
                  ... //code is same as the one in - Creating a
                    sheet of buttons with ActionSheet - recipe
                ]
            }]
          });

        }
    });
```

2. Include `ch06_03.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see something similar to the following screenshot on the screen:



Uses

Roses are best known as ornamental plants grown for their flowers in the garden and sometimes indoors. They have been also used for commercial perfumery and commercial cut flower crops. Some are used as landscape plants, for hedging and for other utilitarian purposes such as game cover and slope stabilization. They also have minor medicinal uses.

How it works...

The preceding code modifies the code in the previous recipe in such a way that the panel to show the large photo is moved from the main panel to the `Carousel` class. The `Carousel` class has two more panels: **About** and **Uses**, which contain more information about the topic at hand.

The `Ext.carousel.Carousel` class implements the complete `Carousel` functionality. Internally, it uses the `card` layout to render its children.

There's more...

Different applications may have different needs based on orientation. Some may like it to be horizontal whereas some may like it to be vertical. In the next section, we will see how to achieve this.

Changing direction

By default, the Carousel class's direction is horizontal. Alternatively, if required, we can set it to vertical as well. This behavior is provided by the `direction` property of the `Carousel` class. The following code snippet shows how to set this property on `Carousel`:

```
var carousel = Ext.create('Ext.carousel.Carousel', {
  direction: 'vertical',
  items: [
```

Turning off the indicator

By default, the `Carousel` class's components shows dots as indicators, and the number of dots are shown based on the number of child items inside the Carousel. Sometimes, we may want to hide the indicator. This can be achieved by setting the `indicator` config to `false` while instantiating `Carousel`.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- *Chapter 4*, *Building Custom Views*

Managing a list of data using List

Let us assume that there is a data set in your application that needs to be presented to the user in the form of a list that the user can scroll through and make their selection. For example, a list of contacts, places, or matching words. Sencha Touch provides a `List` component to handle any list-related needs. This recipe shows how to use it to present the contact list to the user.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

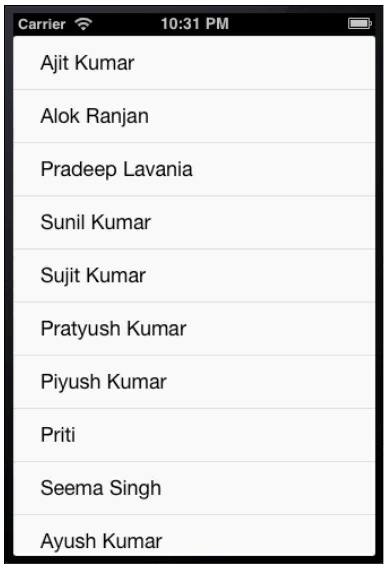
How to do it...

Carry out the following steps:

1. Create and open a new file named `ch06_04.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',

  launch: function() {

    Ext.define('MyApp.model.Contact', {
      extend: 'Ext.data.Model',
      config: {
        fields: ['firstName', 'lastName']
      }
    });

    var store = Ext.create('Ext.data.Store', {
      model  : 'MyApp.model.Contact',

      data: [
        {firstName: 'Ajit',   lastName: 'Kumar'},
        {firstName: 'Alok',   lastName: 'Ranjan'},
        {firstName: 'Pradeep',lastName: 'Lavania'},
        {firstName: 'Sunil',  lastName: 'Kumar'},
        {firstName: 'Sujit',  lastName: 'Kumar'},
        {firstName: 'Pratyush',lastName: 'Kumar'},
        {firstName: 'Piyush', lastName: 'Kumar'},
        {firstName: 'Priti', lastName: ''},
        {firstName: 'Seema',  lastName: 'Singh'},
        {firstName: 'Ayush',  lastName: 'Kumar'},
        {firstName: 'Ayush',  lastName: 'Ranjan'},
        {firstName: 'Alisha', lastName: 'Lavania'},
        {firstName: 'Deepak',   lastName: 'Sinha'},
        {firstName: 'Sheela',   lastName: 'Kejawani'},
        {firstName: 'Srikanth',    lastName: 'Reddy'},
        {firstName: 'Suman', lastName: 'Ravuri'},
        {firstName: 'Ranjit', lastName: ''},
        {firstName: 'Jay', lastName: 'Sharma'}
      ]
    });
    var list = Ext.create('Ext.dataview.List', {
      fullscreen: true,
      itemTpl: '<div>{firstName} {lastName}</div>',
      store: store
    });
  }
});
```

2. Include `ch06_04.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see something similar to the following screenshot:



Ajit Kumar
Alok Ranjan
Pradeep Lavania
Sunil Kumar
Sujit Kumar
Pratyush Kumar
Piyush Kumar
Priti
Seema Singh
Ayush Kumar

How it works...

The preceding code creates a list of contact names and allows the user to select an entry.

```
Ext.define('MyApp.model.Contact', {
  extend: 'Ext.data.Model',
  config: {
    fields: ['firstName', 'lastName']
  }
});
```

This code defines and registers a `Contact` model with the model manager. The model is used on the store in conjunction with the `data` array to convert the `data` array into the model and populate the store.

Since `List` extends `DataView`, it inherits the capabilities and behaviors of `DataView`. The view is refreshed as soon as the models are loaded into the store, which is associated with the list. Each record in the list is rendered using the template defined as `itemTpl`.

There's more...

Sorting is one need that arises naturally when we are dealing with information in a list. Let us see how we can have sorted data inside a list.

Sorting the entries

The list does not provide any method to sort the entries in it. Rather, we shall set up `sorters` on the associated `store` component, as shown in the following code snippet:

```
var store = Ext.create('Ext.data.Store', {
    model  : 'MyApp.model.Contact',
    sorters: 'firstName',
```

`sorters: 'firstName'` will sort the records by their first name and in ascending order. In case we want to sort the data on multiple fields and stipulate the specific way (ascending/descending) the data needs to be sorted, we will expand the `sorters` property value as follows:

```
sorters: [{property: 'firstName', direction: 'ASC'},
    {property: 'lastName', direction: 'DESC'}],
```

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Working with a store* recipe of *Chapter 5*, *Dealing with Data and Data Sources*

Grouping items in a list

In a list, you may want to see the items grouped based on certain criteria; for example, in our contact list, we may want to see the names grouped alphabetically. For this, the `List` class allows us to group data using on criteria and this recipe is going to show exactly how this can be achieved.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

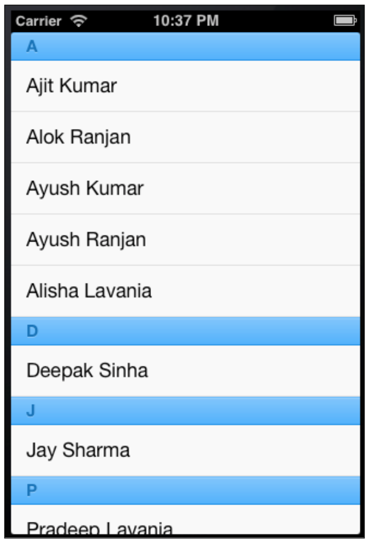
How to do it...

Carry out the following steps:

1. Create and open a new file named `ch06_05.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {

    Ext.define('MyApp.model.Contact', {
      extend: 'Ext.data.Model',
      config: {
        fields: ['firstName', 'lastName']
      }
    });

    var store = Ext.create('Ext.data.Store', {
      model   : 'MyApp.model.Contact',
      grouper: {
        groupFn: function(record) {
          return record.get('firstName')[0];
        }
      },
      data: [
        ... //code from ch06_04.js of - Managing a list of
          data using List - recipe
      ]
    });

    var list = Ext.create('Ext.dataview.List', {
      fullscreen: true,
      grouped: true,
      itemTpl: '<div>{firstName} {lastName}</div>',
      store: store
    });
  }
});
```

2. Include `ch06_05.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see something similar to the following screenshot:



How it works...

The preceding code builds on top of the code mentioned in the previous recipe. It adds the grouping capability to the list by setting the `grouped` property on the `List` class to `true` and implementing the method `groupFn` on the `store` class, which is called by the framework to group the information as per the specified field; in this case, `firstName`. In the code, we are returning the first character of the first name from `groupFn` and hence the data will be grouped on the returned character. However, we can group the data using the entire first name by returning the value of the `firstName` field by changing the function body to `return record.get('firstName')`.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Working with a store* recipe of *Chapter 5*, *Dealing with Data and Data Sources*
- The *Data grouping* recipe of *Chapter 5*, *Dealing with Data and Data Sources*
- The *Managing a list of data using List* recipe

Navigating through a list of data using IndexBar

Imagine there is big book that we are reading and we want to quickly locate the topic of our interest. The very first thing we look for will be the index page, which can tell us the topics and their page numbers. Similarly, in the list, if the items are huge, we can use the index bar functionality to quickly go to the item of our choice. And this recipe will walk us through how to do that.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

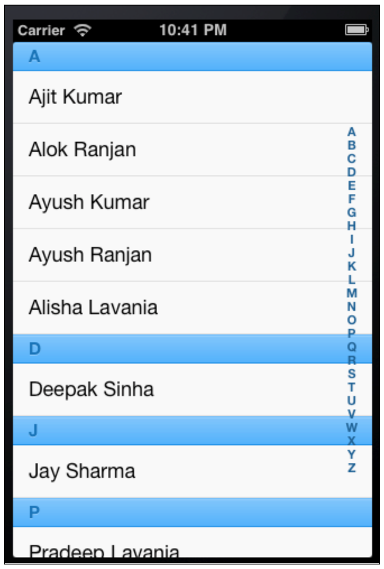
How to do it...

Carry out the following steps:

1. Create and open a new file named `ch06_06.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',

  launch: function() {
    ... //code from ch06_05.js of - Grouping items in a
      List - recipe
    var list = Ext.create('Ext.dataview.List', {
      fullscreen: true,
      grouped: true,
      indexBar    : true,//use IndexBar
      itemTpl: '<div>{firstName} {lastName}</div>',
      store: store
    });
  }
});
```

2. Include `ch06_06.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see something similar to the following screenshot:



How it works...

In the preceding code, the index bar is enabled by setting the `indexBar` property to `true` on `List`. This property tells the framework to generate an index bar (similar to the index at the end of the book) with letters from A-Z, and allows the user to jump to the matching entries when he/she clicks on a particular index.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Working with a store* recipe of *Chapter 5*, *Dealing with Data and Data Sources*
- The *Managing a list of data using List* recipe

Working with a list of nested data using NestedList

Imagine you have a nested data structure that you would like to present to the user in the form of a list and allow him/her to drill down it. In this recipe, we will understand how to achieve this using the `NestedList` component.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

How to do it...

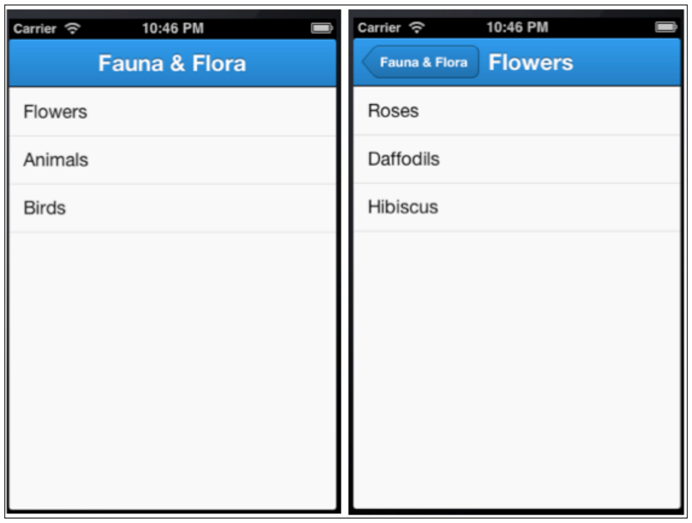
Carry out the following steps:

1. Create and open a new file named `ch06_07.js` and copy-paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {
    var data = {
      items: [{
        text: 'Flowers',
        items: [{
          text: 'Roses',
          items: [{
            text: 'Red',
              leaf: true
          },{
            text: 'Peach',
            leaf: true
          },{
            text: 'Yellow',
            leaf: true
          }]
        },{
          text: 'Daffodils',
          leaf: true
        },{
          text: 'Hibiscus',
          leaf: true
        }]
      },{
        text: 'Animals',
        items: [{
          text: 'Lion',
          leaf: true
      },{
        text: 'Elephant',
        leaf: true
      }]
    },{
      text: 'Birds',
        items: [{
          text: 'Eagle',
          leaf: true
        },{
          text: 'Hamsa',
          leaf: true
        },{
          text: 'Pegion',
          leaf: true
        }]
    }]
  };
```

```
            Ext.define('MyApp.model.ListItem', {
              extend: 'Ext.data.Model',
              config: {
                fields: [{name: 'text', type: 'string'}]
              }
            });
            var store = Ext.create('Ext.data.TreeStore', {
              model: 'MyApp.model.ListItem',
              root: data,
              defaultRootProperty: 'items'
            });
            var nestedList = Ext.create('Ext.dataview.NestedList',
              {
              fullscreen: true,
              title: 'Fauna & Flora',
              store: store
            });
        }
    });
```

2. Include `ch06_07.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see something similar to the following screenshot:



How it works...

The preceding code creates a NestedList component using the data array. The NestedList component uses TreeStore, which expects the data to follow a particular tree structure. The data structure shows that, at the top level, data has three nodes: **Flowers**, **Animals**, and **Birds**. Each one of them has child items. For example, **Flowers** has immediate children **Roses**, **Daffodils**, and **Hibiscus**. For **Daffodils** and **Hibiscus**, the leaf property is set to true, indicating they are the leaf nodes of the tree and will not have any child items. The nesting can go up to any level. Each node has a property text that TreeStore uses to show them on the screen.

NestedList extends the container; based on the nesting of data and at what level the user is, it creates a docked toolbar on the top and displays the buttons on the toolbar to allow the user to navigate through the hierarchy. The button labels are generated using the text property of the nodes.

There's more...

There are additional features available with a tab panel and the subsequent sections cover them.

Using a property other than text

By default, NestedList uses the text property of the node to display it on the screen and generate the button labels. In case our data has a different property (say, label), we shall use the displayField property on NestedList and set it to 'label' as follows:

```
var nestedList = Ext.create('Ext.dataview.NestedList', {
    fullscreen: true,
    title: 'Fauna & Flora',
    displayField: 'label',
    store: store
});
```

Showing the Back button

Say that, in our application, we want to have the **Back** label for the button rather than the text of the parent node of the current level. This can be achieved by setting the useTitleAsBackText property to false.

No toolbar, please!

By default, NestedList generates a top toolbar, adds a **Back** button to it, and handles its click event. In case we do not want to see this toolbar, we shall set the useToolbar property on NestedList to false.

Using different text for the Back button

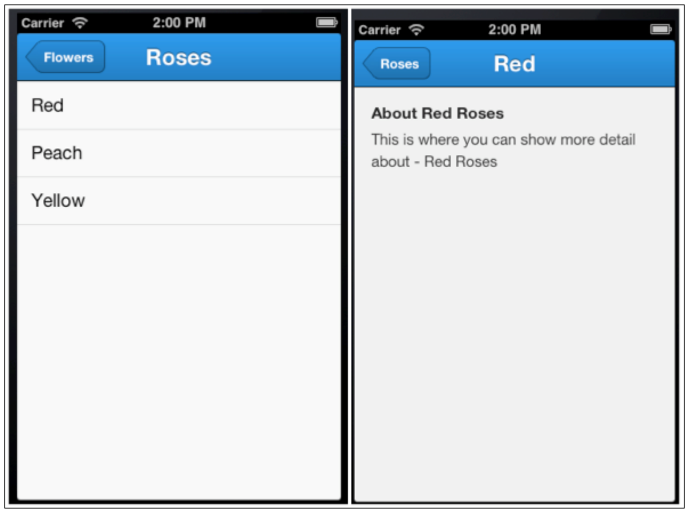
Rather than showing **Back** as the button label, if we want to show custom text (say, **Prev**), we will have to set `useTitleAsBackText` to `false` and `backText` to your custom text; that is, `Prev`.

Showing leaf node detail

By default, `NestedList` stops drilling down when it reaches the leaf node. In some cases, you may want to go down one more level and show detail for the leaf node. For example, when the user taps on the **Red** leaf node under **Roses**, we want to show details about red roses in text. To achieve this, we will have to use the `detailCard` config of `NestedList`. The following code shows that we have used `detailCard` as the container and then we are handling the `leafitemtap` event on `NestedList`, fired when a leaf node is tapped, to show the details related to the tapped leaf node:

```
var nestedList = Ext.create('Ext.dataview.NestedList', {
  fullscreen: true,
  title: 'Fauna & Flora',
  detailCard: {
    styleHtmlContent: true
  },
  store: store,
  listeners: {
    leafitemtap: function(list, subList, idx, t, rec, e, eOpts) {
      var parentTxt = rec.parentNode.data.text;
      var detailCard = list.getDetailCard();
      detailCard.setHtml(Ext.String.format('<h1 style="font-
        size:16px;"><b>About {0} {1}</b></h1><p>This is where you
         can show more detail about - {0} {1}</p>',
           rec.data.text, parentTxt));
    }
  }
});
```

When we run the example and tap on the **Red** leaf node under **Roses**, we will see the following output:



About Red Roses

This is where you can show more detail about - Red Roses

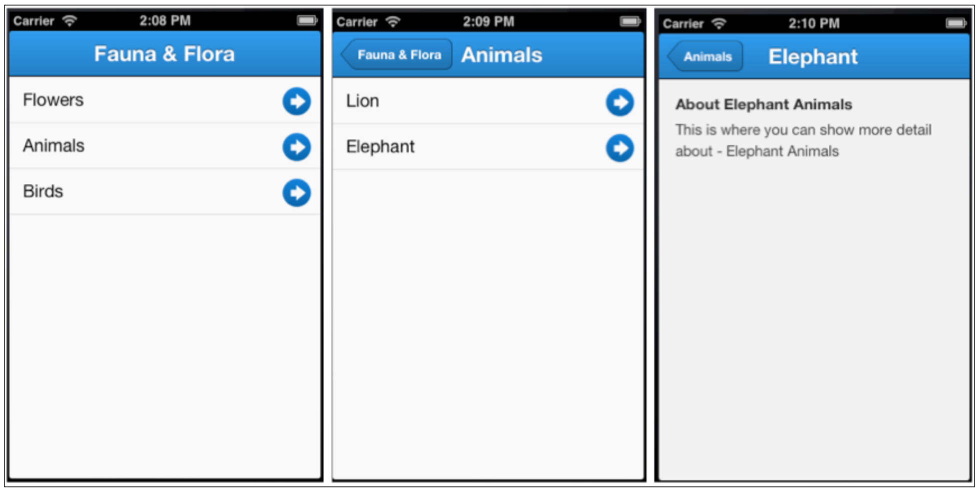
Using disclosure

A user will never be able to discover that there are child items under **Roses** unless they tap on it. Sometimes, this may cause a usability issue, and you may want to show explicit indicators hinting to the user that there is more under an item. This can be enabled on a list using the `onItemDisclosure` config, which shows a disclosure arrow icon on the right-hand side of a list item and also offers the `disclose` event for us to implement the behavior. The following code shows how we have set the `onItemDisclosure` config and handled the `disclose` event to have the same behavior that we had on tapping:

```
var nestedList = Ext.create('Ext.dataview.NestedList', {
  fullscreen: true,
  title: 'Fauna & Flora',
  onItemDisclosure: true,
  listConfig: {
    listeners: {
      disclose: function(list, record, target, index, e, eOpts) {
        nestedList.onItemTap( list, index, target, record, e );
      }
    }
  },
  detailCard: {
...
});
```

Note that we have passed `listConfig` and inside it we have specified the listener for the `disclose` event. This is required because each stage of the view in `NestedList` is rendered as `List`, which offers the `onItemDisclosure` property and also the `disclose` event. The `disclose` event is not fired on `NestedList`. `onItemDisclosure` is set on `NestedList`, which sets this config on each of the inner lists.

When we run the example with these changes, we see the disclosure icon appearing; tapping on the icon will result in the same behavior as tapping on the item, as shown in the following screenshot:



About Elephant Animals

This is where you can show more detail about - Elephant Animals

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1, Gear Up for the Journey*

Picking up your choice using Picker

In *Chapter 2*, *Catering to Your Form-related Needs*, we had talked about `DatePicker`, which shows the dates in the form of slots and allows us to pick a date. `DatePicker` is a specialized version of the `Picker` class. In this recipe we will see how to make use of this class.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

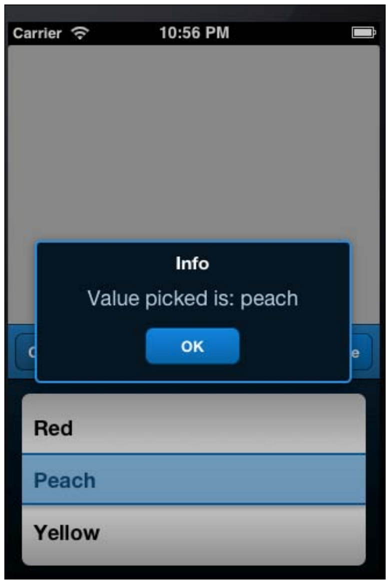
How to do it...

Carry out the following steps:

1. Create and open a new file named `ch06_08.js` and copy-paste the following code into it:

```
Ext.application({
  launch: function() {
    var picker = Ext.create('Ext.picker.Picker', {
      slots: [{
        name : 'color',
        data : [
          {text: 'Red', value: 'red'},
          {text: 'Peach', value: 'peach'},
          {text: 'Yellow', value: 'yellow'},
          {text: 'White', value: 'white'}
        ]
      }],
      listeners: {
        pick: function(picker, pickedObj, slot) {
          Ext.Msg.alert('Info', 'Value picked is: ' +
            pickedObj.color);
        }
      }
    });
    Ext.Viewport.add(picker);
    picker.show();
  }
});
```

2. Include `ch06_08.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see something similar to the following screenshot:



How it works...

The preceding code creates a `Picker` class to allow the user to choose a color of their choice. Each color detail is added as a slot to the picker. Every slot contains two properties, `text` and `value`. The `text` property is used to display the name in the slot and the `value` property is given back to the program when the user picks up a slot.

We then register a handler for the `pick` event on the picker. The handler is fired when the user selects a slot. The parameter `pickerObj` contains information about the slot that is selected. This object contains the name of the slots as its property and the value of the selected slot is set as its value. For example, in our case the `pickerObj` parameter will look like the following when the user selects the **Peach** slot:

```
{
  color: "peach"
}
```

There's more...

As with any other component, there are certain defaults defined by `Picker`. However, we may have to deviate from them; for example, the position, animation, and alignment. The following section shows how to do this.

Changing the position and animation

The `Picker` class extends `Sheet` and inherits the positioning and animation properties from it. We can use the four properties, `enter`, `exit`, `showAnimation`, and `hideAnimation`, to indicate the position of the picker with regard to the viewport when it is shown or hidden and what kind of animation shall be used. The following code snippet shows the usage of these properties to show the `picker` component on the top of the screen and uses the `fadeIn` animation when it is being shown and the `fadeOut` animation when it is being hidden:

```
var picker = Ext.create('Ext.picker.Picker', {
  enter: 'top',
  showAnimation: 'fadeIn',
  exit: 'top',
  hideAnimation: 'fadeOut',
```

By default, the picker uses the `bottom` position and the `slide` animation.

Aligning the slot text

By default, the slot shows the text in the center. However, by using the `align` property on the slots, we can align the text to the left or right as follows:

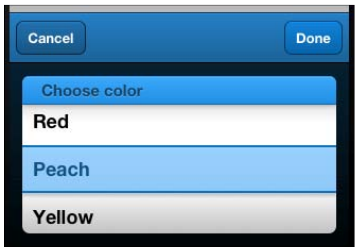
```
slots: [
  {
  name : 'color',
  align: 'left',
```

Using title

To show the title for a slot, we need to specify the `title` config on the slot and set `useTitles` to `true` on the `picker` object, as shown in the following code snippet:

```
var picker = Ext.create('Ext.picker.Picker', {
  useTitles: true,
  slots: [{
    title: 'Choose color',
    name : 'color',
```

The following screenshot shows how the title will appear:



Overriding the button text

By default, the **Cancel** and **Done** buttons appear on the `picker` component. Using the `cancelButton` and `doneButton` configs on the `picker` object, we can replace the default button text with our text. The following code snippet shows the usage of these configs:

```
var picker = Ext.create('Ext.picker.Picker', {
  cancelButton: 'Skip',
  doneButton: "I'm Done!",
  slots: [{
    name : 'color',
```

Hiding buttons

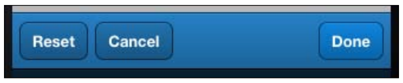
Sometimes, you may want to hide the **Cancel** or **Done** buttons. Setting `cancelButton` to `false` will not show the **Cancel** button and setting `doneButton` to `false` will not show the **Done** button. The following shows how we can do this:

```
var picker = Ext.create('Ext.picker.Picker', {
  cancelButton: false,
  doneButton: false,
  slots: [{
    name : 'color',
```

Customizing the toolbar

If we want to customize the toolbar at the top to show additional items, we set the `toolbar` config on the `picker` object as shown in the following code snippet, where we are adding one more button to the toolbar:

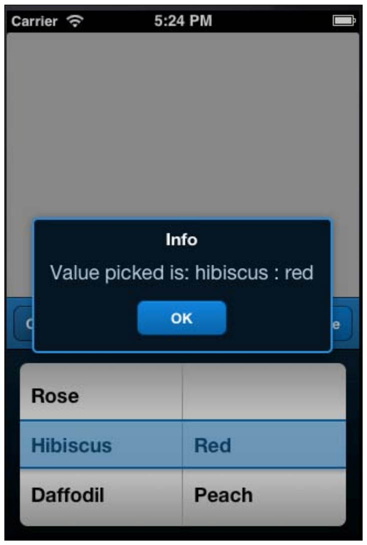
```
var picker = Ext.create('Ext.picker.Picker', {
  toolbar: {
    items: [{
      text: 'Reset'
    }]
  },
  slots: [{
    name : 'color',
```



Showing multiple slots

The `slots` config accepts an array, and each entry represents a slot. Slots are arranged horizontally and their order is determined based on the order in which they have been added to the `slots` config. The following code shows how we can add one more slot to select a flower, in addition to the slot to select a color:

```
var picker = Ext.create('Ext.picker.Picker', {
  slots: [
    {
      title: 'Choose flower',
        name : 'flower',
        data : [
          {text: 'Rose', value: 'rose'},
          {text: 'Hibiscus', value: 'hibiscus'},
          {text: 'Daffodil', value: 'daffodil'},
          {text: 'Daisy', value: 'daisy'}
        ]
      },{
        itle: 'Choose color',
        name : 'color',
        Ext.Msg.alert('Info', 'Value picked is: ' +
          pickedObj.flower + ' : ' + pickedObj.color);
```



Info

Value picked is: hibiscus : red

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*

Switching between multiple views using SegmentedButton

This recipe describes the usage of the `SegmentedButton` component, which is generally a part of the toolbar and is useful in switching between different views.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

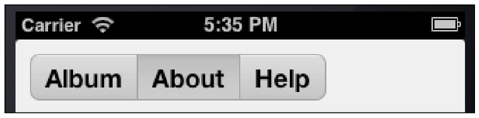
How to do it...

Carry out the following steps:

1. Create and open a new file named `ch06_09.js` and copy-paste the following code into it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
    var segmentedButton = Ext.create('Ext.SegmentedButton',
      {
      items: [
        {
          text: 'Album'
        },
        {
          text   : 'About',
          pressed: true
        },
        {
          text: 'Help'
        }
      ],
      listeners: {
        toggle: function(container, button, pressed){
          console.log("User toggled the '" +
            button.getText() + "' button: " + (pressed ?
              'on' : 'off'));
        }
      }
    });
    Ext.Viewport.add({
      xtype: 'container',
      padding: 10,
      items: [segmentedButton]
    });
  }
});
```

2. Include `ch06_09.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see something similar to the following screenshot:



How it works...

The preceding code creates `SegmentedButton` with three buttons, **Album**, **About**, and **Help**, and allows the user to select one of them. Setting `pressed: true` on the **About** button ensures that it will be selected by default. A selected button is deselected only if the user selects another button.

The `SegmentedButton` component fires the `toggle` event every time a button is selected and deselected. Our handler for the `toggle` event shows a message informing us which button is selected/deselected. For example, when the **Album** button is pressed, we see two messages appearing on the console: one saying **About button: off** and other one saying **Album button: on**.

Finally, we added the `SegmentedButton` instance to a container, which is added to the viewport.

There's more...

SegmentedButton also allows us to press multiple buttons. Let's see how to do it.

Keeping multiple buttons pressed

If we need the capability of keeping multiple buttons pressed, we set the allowMultiple property to true. Setting this property allows us to deselect an already selected button.

Aligning buttons in the middle

By default, the buttons are aligned to the left. To change their alignment and show them in the middle, we will have to set the pack property on the layout config as shown in the following code snippet:

```
var segmentedButton = Ext.create('Ext.SegmentedButton', {
  layout: {
    type: 'hbox',
    pack: 'middle'
 },
  items: [
...
```

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*

Working with Tab panels

The Tab panel is a popular UI component that can hold other components and be accessed in a tabbed fashion using a tab bar. In this recipe, we will learn about the tab panel and the different options that we may use to build our application.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the ch06 folder inside the www folder.

How to do it...

Carry out the following steps:

1. Create and open a new file named `ch06_10.js` and copy-paste the following code into it:

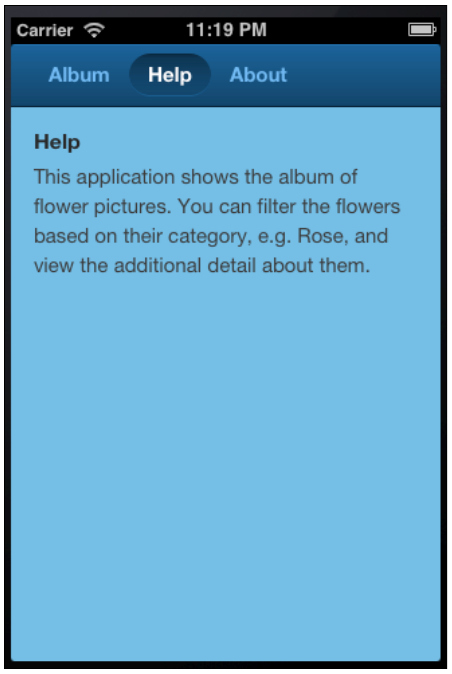
```
Ext.application({
  name: 'MyApp',
  launch: function() {
    Ext.create('Ext.tab.Panel', {
      fullscreen: true,
      items: [
        {
          title: 'Album',
          styleHtmlContent: true,
          html: 'Contains the photos!',
          cls: 'tab1'
        },
        {
          title: 'Help',
          styleHtmlContent: true,
          html: '<h1 style="font-size:16px;"><b>Help</b>
            </h1><p>This application shows the album of
            flower pictures. You can filter the flowers
            based on their category, e.g. Rose, and view
            the additional detail about them.</p>',
          cls  : 'tab2'
        },
        {
          title: 'About',
          styleHtmlContent: true,
          html : '<h1 style="font-size:16px;"><b>About this
            app!</b></h1><p>Version 0.1</p>',
          cls  : 'tab3'
        }
      ]
    });
  }
});
```

2. Include `ch06_10.js` in the `index.html` file.

3. Add the following styles to the `ch06.css` file:

```
.tab1 {
  background-color: #E58A99;
}

.tab2 {
  background-color: #65B9E0;
}

.tab3 {
  background-color: #B7E488;
}
```

4. Deploy and access it from the browser. You may also run it using the emulator. You should see something similar to the following screenshot:



Help

This application shows the album of flower pictures. You can filter the flowers based on their category, e.g. Rose, and view the additional detail about them.

How it works...

The preceding code creates a `tab` panel with three panels. Internally, the `tab` panel generates a tab bar using the `title` property of each panel item; with that it allows the user to switch between different tabs.

There's more...

There are a few more interesting options available with the `tab` panel. The following section describes one of them.

Positioning the tab bar at the bottom

By default, the tab bar is positioned on the top. To show it at the bottom, we shall set the `tabBarPosition` property to `bottom`.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*

Getting quicker access to application features using Toolbar

The `Toolbar` class is a great way of getting single-click access to application features. It can have buttons, dropdowns, a text field, and so on. Sencha Touch provides a `Toolbar` component and this recipe will show us how to use it and work with its options.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

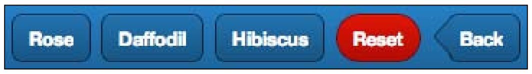
How to do it...

Carry out the following steps:

1. Create and open a new file named `ch06_11.js` and copy-paste the following code into it:

```
Ext.application({
  launch: function() {
    var myToolbar = Ext.create('Ext.Toolbar', {
      items: [{
        text: 'Rose'
      },
      {
        text: 'Daffodil'
      },{
        text: 'Hibiscus'
      },{
        text: 'Reset',
        ui: 'decline-round'
      }, {
        text: 'Back',
        ui: 'back'
      }
      ]
    });

    var myPanel = Ext.create('Ext.Panel', {
      items: [myToolbar],
      styleHtmlContent: true,
      fullscreen : true,
      html       : 'Test Panel'
    });
  }
});
```

2. Include `ch06_11.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see something similar to the following screenshot:



How it works...

The preceding code creates a toolbar with five buttons inside it and the toolbar is added to the panel. The **Reset** button is using `decline-round` as the value for the `ui` property and the **Back** button is using `back` as the value for the `ui` property for a different look and feel.

There's more...

What if we want to have non-button components in our toolbar? Well, let us see.

Adding non-button components

The defaultType config property on the toolbar defines the xtype property that shall be used for each item being added to it. Unless xtype is specified on an item, xtype defaults to button, which is the default value for defaultType. That's the reason we did not have to specify xtype for buttons in the preceding code. To add a component of some other xtype, we will have to set the xtype property on the particular item. For example, the following code shows adding a search field to the toolbar:

```
items: [{
  xtype: 'searchfield'
}, {
  text: 'Rose'
}]
```

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1, Gear Up for the Journey*

Creating a new component

So far, we have seen the various components the Sencha Touch framework offers and how to use them to model our application. However, there may be a need to create new components or extend the capability of an existing component. This recipe walks us through the steps to create a new component.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1, Gear Up for the Journey*.

Make sure you have created the ch06 folder inside the www folder.

How to do it...

Carry out the following steps:

1. Create and open a new file named `PhotoAlbum.js` and copy-paste the following code into it:

```
Ext.define('Touch.book.ux.PhotoAlbum', {
  extend: 'Ext.dataview.DataView',
  xtype: 'photoalbum',
  config: {
    store: null,
    itemTpl : new Ext.XTemplate(
      '<div class="thumb-wrap" id="{title}">',
      '<div class="thumb"><img src="{url}"
        title="{title}"></div>',
        '<span>{about}</span></div>',
        '<div class="x-clear"></div>'
    ),
    scrollable: 'vertical',
    emptyText: 'No images to display'
  },

  initialize: function() {
    this.callParent(arguments);

    this.on('selectionchange', function(dv, recs) {

      if (recs.length > 0) {
        Ext.Msg.alert('Info', 'Selected: ' +
          recs[0].data.album + ' : ' + recs[0].data.about);
      }
    });
  }
});
```

2. Create and open a new file named `ch06_12.js` and copy-paste the following code into it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
    var data = [{
      album:'rose',
      url:'http://images.flowers.vg/250x300/rdroses01.jpg',
      title:'Rose 1',
      about:'Peach'
    },
    ...
    ...
    {
      album:'hibiscus',
      url:'http://images.flowers.vg/250x300/hibiscus-red-
        bright.jpg',
      title:'Hibiscus 1',
      about:'Bright Red'
    }];

    var store = Ext.create('Ext.data.Store', {
      data: data,
      fields: [
          'url', 'title','about', 'album'
      ]
    });

    Ext.create('Touch.book.ux.PhotoAlbum', {
      id: 'images-view',
      fullscreen: true,
      store: store
    });

  }
});
```

3. Include `PhotoAlbum.js` and `ch06_12.js` in the `index.html` file.
4. Deploy and access it from the browser. You may also run it using the emulator.

How it works...

In the preceding code, we defined a new component named `PhotoAlbum` in the `Touch.book.ux` namespace. `PhotoAlbum` extends the `DataView` component and defines its own template to render its items, and other common properties are defined inside it. The `Ext.define` method provides us with a way to define a new component by extending an existing one. You may also extend `Object` using this method. Additionally, one method that has been added to it is `initialize`. This method acts as a hook into the overall component management lifecycle of Sencha Touch, discussion of which is beyond the scope of this book.

The `initialize` method is called by the component manager to give a chance to the component to take care of its specific initialization. This is called during the initialization of a component. Our `PhotoAlbum` component registers the handlers for the `selectionchange` event.

`this.callParent(arguments)` calls the corresponding method of the super class, which is `DataView`. This is required for the parent class to be initialized properly.

> The `Ext.Component` class contains the code related to the component lifecycle; understanding that may give you more insight into writing your own component.

The properties `itemTpl`, `scrollable`, and others that are being set inside the component, can be overridden by the value specified by the user at the time of constructing an instance of `PhotoAlbum`. For example, if you want to have a different template for `PhotoAlbum`, you can pass the `itemTpl` property during the instantiation as follows:

```
Ext.create('Touch.book.ux.PhotoAlbum', {itemTpl: ….});
```

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Working with a store* recipe of *Chapter 5*, *Dealing with Data and Data Sources*

Extending an existing component's capability

In the previous recipe, we defined a new component and used it in our application. However, in some cases, the choice may not be to define a new component. Rather, we may have to see if we can add the capability to the existing component. For example, `String` is a standard object in JavaScript and we would like to add a new method, `formatWithWordBreak`, in such a way that, once it is added, it is available to the complete application code to make use of this new method without defining a `MyString` class and using it wherever we need the `formatWithWordBreak` method. This recipe will take us through the steps to achieve this requirement.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

How to do it...

Carry out the following steps:

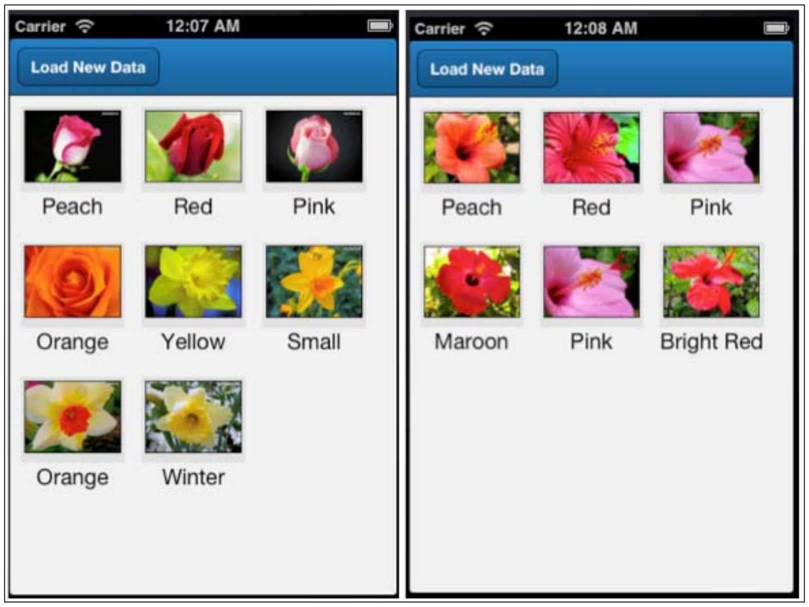
1. Create and open a new file named `ch06_13.js` and copy-paste the following code into it:

```
Touch.book.ux.PhotoAlbum.prototype.loadData =
  function(data) {
  this.getStore().setData(data);
};

Ext.application({
  name: 'MyApp',

  launch: function() {
    var data = [{
      album:'rose',
      url:'http://images.flowers.vg/250x300/rdroses01.jpg',
      title:'Rose 1',
      about:'Peach'
    }, {
      album:'rose',
      url:'http://images.flowers.vg/250x300/roses-
        maroon3.jpg',
      title:'Rose 2',
      about:'Red'
    }, {
      album:'rose',
      url:'http://images.flowers.vg/250x300/roses-dark-
        pink.jpg',
      title:'Rose 3',
      about:'Pink'
    }, {
      album:'rose',
      url:'http://images.flowers.vg/250x300/roses-bright-
        orange.jpg',
      title:'Rose 4',
      about:'Orange'
    }, {
      album:'daffodil',
      url:'http://images.flowers.vg/250x300/daffodil.jpg',
      title:'Daffodil 1',
      about:'Yellow'
```

```
}, {
  album:'daffodil',
  url:'http://images.flowers.vg/250x300/daffodil-
    yellow.jpg',
  title:'Daffodil 2',
  about:'Small'}, {
  album:'daffodil',
  url:'http://images.flowers.vg/250x300/daffodil-white-
    orange.jpg',
  title:'Daffodil 2',
  about:'Orange'
}, {
  album:'daffodil',
  url:'http://images.flowers.vg/250x300/winter_
    flowers_daffodil_white.jpg',
  title:'Daffodil 2',
  about:'Winter'
}];

var newData = [{
  album:'hibiscus',
  url:'http://images.flowers.vg/250x300/hibiscus-
    peach.jpg',
  title:'Hibiscus 1',
  about:'Peach'
}, {
  album:'hibiscus',
  url:'http://images.flowers.vg/250x300/
    hibiscusred.jpg',
  title:'Hibiscus 1',
  about:'Red'
}, {
  album:'hibiscus',
  url:'http://images.flowers.vg/250x300/hibiscus-pink-
    pink.jpg',
  title:'Hibiscus 1',
  about:'Pink'
}, {
  album:'hibiscus',
  url:'http://images.flowers.vg/250x300/hibiscus-red-
    maroon.jpg',
  title:'Hibiscus 1',
        about:'Maroon'
      }, {
        album:'hibiscus',
        url:'http://images.flowers.vg/250x300/hibiscus-pink-
          pink.jpg',
        title:'Hibiscus 1',
        about:'Pink'
      }, {
        album:'hibiscus',
        url:'http://images.flowers.vg/250x300/hibiscus-red-
          bright.jpg',
        title:'Hibiscus 1',
        about:'Bright Red'}];

      var store = Ext.create('Ext.data.Store', {
        data: data,
        fields: [
          'url', 'title','about', 'album'
        ]
      });

      var photoPnl = Ext.create('Touch.book.ux.PhotoAlbum', {
        id: 'images-view',
        fullscreen: true,
        store: store,
        items: [{
          xtype: 'toolbar',
          docked: 'top',
          items:[{
            text: 'Load New Data',
            handler: function() {
              photoPnl.loadData(newData);
            }
          }]
        }]
      });
    }
  });
```

2. Include `ch06_13.js` in the `index.html` file.

3. Deploy and access it from the browser. You may also run it using the emulator. You will see something similar to the following screenshot:



How it works...

`prototype` is the standard JavaScript mechanism to extend an existing JavaScript object. For example, adding the `printWithLineBreak` method to the exiting `String` object so that the new method is accessible across the application code. *JavaScript: The Good Parts*, a book by *Douglas Crockford*, is an excellent resource on JavaScript. The preceding code uses the same mechanism to add a new method named `loadData` to the existing `PhotoAlbum`, which loads data into the `DataView` store. When the user clicks on the **Load New Data** button, `photoPnl` is loaded with the new data array by calling the newly added `loadData` method on the `PhotoAlbum` class.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1, Gear Up for the Journey*
- The *Creating a new component* recipe

Overriding a component's behavior

This recipe will show us how to override the existing behavior of a component and use the modified behavior in the code.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

How to do it...

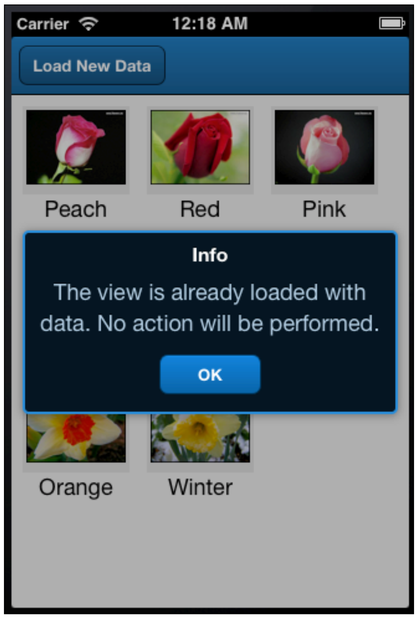
Carry out the following steps:

1. Create and open a new file named `PhotoAlbumOverride.js` and copy-paste the following code into it:

   ```
   Ext.define('Touch.book.ux.PhotoAlbumOverride', {
     override: 'Touch.book.ux.PhotoAlbum',
     loadData : function(data) {
       if (this.getStore().getCount() > 0) {
         Ext.Msg.alert('Info', 'The view is already loaded
           with data. No action will be performed.');
       } else {
         this.getStore().setData(data);
       }
     }
   });
   ```

2. Update the `index.html` file to include the `PhotoAlbumOverride.js` file. We will use the existing `ch06_13.js` file. Make sure you include the `PhotoAlbumOverride.js` file after `ch06_13.js`.

3. Deploy and access it from the browser. You may also run it using the emulator. You will see something similar to the following screenshot:



Info

The view is already loaded with data. No action will be performed.

How it works...

`Ext.define` with the `override` config allows us to override the existing behavior of a class. It allows us to override class properties and/or methods in a convenient way using the Touch API programming model. The good part is that you don't have to use the newly defined class, `PhotoAlbumOverride`. Rather, you can override a property or method of the class and continue to use it with different behavior. The method checks whether there is data already loaded into the view; if so, it shows a message to the user and skips the loading of new data. Otherwise, it loads the new data.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1, Gear Up for the Journey*
- The *Creating a new component* recipe
- The *Extending an existing component's capability* recipe

Adding behavior to an existing component using plugins

Plugins are another mechanism through which we can enhance/customize the behavior of an existing component. The new behavior is effective only if the plugin is added to the component. Otherwise, the base behavior remains intact. In this recipe, we will understand how to create a new plugin and use that on an existing component.

Getting ready

Make sure you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure you have created the `ch06` folder inside the `www` folder.

How to do it...

Carry out the following steps:

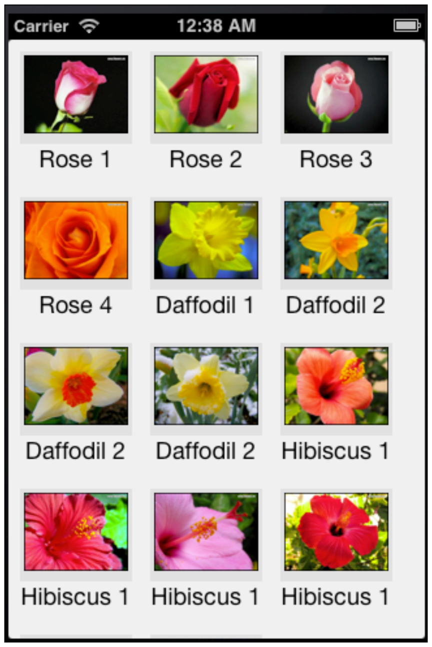
1. Create and open a new file named `PhotoAlbumPlugIn.js` and copy-paste the following code into it:

```
Ext.define('Touch.book.ux.PhotoAlbumPlugIn', {
  extend: 'Ext.Component',
  alias: 'plugin.photoalbum-view',

  init: function(viewCmp) {
    viewCmp.setItemTpl(new Ext.XTemplate(
      '<tpl for=".">',
        '<div class="thumb-wrap" id="{title}">',
        '<div class="thumb"><img src="{url}"
          title="{title}"></div>',
        '<span>{title}</span></div>',
      '</tpl>',
      '<div class="x-clear"></div>'
    ));
  }
});
```

2. Copy `ch06_12.js` as `ch06_14.js` and make the following change to the photo album instantiation code:

```
  Ext.create('Touch.book.ux.PhotoAlbum', {
    id: 'images-view',
plugins: ['photoalbum-view'],
    fullscreen: true,
    store: store
  });
```

3. Include `PhotoAlbumPlugIn.js` and `ch06_14.js` in the `index.html` file.
4. Deploy and access it from the browser. You may also run it using the emulator. You will see something similar to the following screenshot:



How it works...

The Sencha Touch framework provides support for plugins and most of the components have a `plugins` property that can accept one or more plugins that need to be initialized for the component. Plugins are a great way to enhance the capabilities of a component without modifying its core behavior; for example, using a plugin to make the label editable for a `DataView` component. In the preceding code, we defined a plugin named `PhotoAlbumPlugIn` within the `Touch.book.ux` namespace. The plugin extends the `Ext.Component` class so that, in case we have to deal with events, our plugin will be capable of doing it. However, it is not mandatory to extend `Component`. You may also extend `Object` to define a plugin. The important thing is that a plugin must have an `init` method defined, that accepts the component reference to which the plugin was added in its `plugins` property. In our case, the plugin gets the reference to `PhotoAlbum` and sets `itemTpl` to a new template. The `alias` name is defined as a name for the plugin that we can use in our code instead of using the class instance. It is important to use `plugin` as a prefix for the `alias` name. This way, the class is registered as a plugin in the framework and will load the class if the plugin name were used in the code.

After the plugin is defined, `plugins: ['photoalbum-view']` associates the plugin with `PhotoAlbum`; thus, when the application runs, we see the template set by the plugin used.

> A plugin initializes after the `initialize` method of the component is called. So you can rest assured that, when your plugin code is running, the complete component has been initialized.

See also

- The *Setting up a browser-based development environment* recipe of *Chapter 1*, *Gear Up for the Journey*
- The *Creating a new component* recipe

7 Adding Audio/Visual Appeal

In this chapter we will cover:

- Animating an element
- Ding-dong! You have got a message
- Working with videos
- Creating your drawing
- Working with an area chart
- Working with a bar chart
- Working with a column chart
- Showing a group of bars and columns
- Highlighting and displaying an item detail
- Working with a gauge chart
- Working with a line chart
- Working with a pie chart
- Rotating the pies
- Highlighting a pie
- Working with a 3D pie chart
- Working with a radar chart
- Working with a scatter chart
- Working with a candlestick/OHLC chart

Introduction

So far we have worked with components that present data either in the form of lists, form fields, or custom views. However, there is always a need in an application to present the information visually. Also, notification is another key need in an application where you may want to notify the user that a certain event has occurred in the system; for example, a new sales inquiry has arrived and a request has come for your approval. This chapter starts with introducing how to animate elements in Sencha Touch and the different types of in-built animations supported by the framework. Next, we will see how to use the audio control in our application to get notifications, audio help, and more. After audio, we will look into the video components and see how to use them in our application. In subsequent recipes, we will learn how to set up the chart support in our application, what different types of charts are available with the framework, how to use them, and also understand ways to build interactive charts that can respond to user actions.

Animating an element

A Sencha Touch application is built using the elements, represented by `Ext.dom.Element`, and every element of it can be animated. In this recipe, we will see how to animate an element, what different types of animations are available, and how to change animation properties.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to store the code.

How to do it...

Perform the following steps to animate an element:

1. Create and open a new file, `ch07_01.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
  var ct = Ext.create('Ext.Container', {
      fullscreen: true,
      style: 'background-color:gold;',
      height: 100,
      width: 100
  });

  Ext.Anim.run(ct.element, 'cube', {});
    }
});
```

2. Include `ch07_01.js` in the `index.html` file.
3. Deploy and access it from the browser or the device of your choice.

How it works...

In the preceding code, we created a container instance that was rendered to the document body. After the instantiation of it, we animated the container by calling the `run` method of the `Ext.Anim` class. The first argument to the `run` method indicates what element needs to be animated; the second argument, `'cube'`, indicates the type of animation that needs to be applied to the element; and the third argument is used to pass the animation-specific configuration object that is, in this case, empty. That means the default configuration will be applied. You may pass a Sencha Touch component instance, an HTML element, or an `Ext.Element` instance as the first parameter to the `run` method.

The following is the list of animations supported by Sencha Touch and is defined in the `Ext.anims` class:

- `cube`
- `fade`
- `flip`
- `pop`
- `slide`
- `wipe`

Internally, each animation type corresponds to some calculation and then uses the appropriate WebKit CSS properties to animate the element. A list of WebKit CSS properties can be found at `http://css-infos.net/properties/webkit.php`.

There's more...

In the preceding code, we saw that the third argument to the `run` method is the animation-specific configuration. There are various options that can be passed and they are outlined in the `Ext.Anim` class. Let us look at some of the important ones.

Working with different animation durations

By default, the animation lasts for 250 milliseconds. If this is not the desired length, then we can change it by passing the `duration` config option to the `run` method. It accepts a value in milliseconds. The following code snippet shows how to pass the `duration` value:

```
Ext.Anim.run(pnl.getEl(), 'cube', {
  duration : 2000
});
```

Setting the direction of animation

Most of the animations use a default direction. It is useful in deciding the side from which the element shall enter into the scene. For example, cube animation uses left direction, by default. We can change this by passing the `direction` config to the `run` method. The following are the possible values:

- `left`
- `right`
- `up`
- `down`

Reversing the animation

If you want to reverse the direction of animation, setting the `reverse` property to `true` and passing the same to the `run` method can do it. The following code snippet shows how to pass this configuration:

```
Ext.Anim.run(pnl.getEl(), 'cube', {
      reverse: true,
    duration : 2000
  });
```

Postponing animation

If you do not want the animation to start immediately (but rather after a certain time), we can achieve it by using the `delay` option. This option accepts a value in milliseconds. The following code snippet shows how to pass this configuration:

```
Ext.Anim.run(pnl.getEl(), 'cube', {
      delay: 2000,
    ...
  });
```

Calling a function after the animation is over

Let's say you want to execute a piece of code as soon as the animation is over. For example, in the famous word game, Hangman, you would like to present a word to the user to complete as soon as the animation of the hangman is over. To achieve this, the class provides an `after` property to which we can pass a function and it will be called as soon as the animation is over. The following code snippet shows how we can do this:

```
Ext.Anim.run(pnl.getEl(), 'cube', {
  after : function() {
       alert('Animation is over!');
   }
});
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Ding-dong! You have got a message

Say we are building an application for the Sales force that allows them to look at the orders placed in the ERP system from their touch device. And, you want to notify your user by playing a notification sound as soon as a new order arrives in the system. This can be achieved using the audio component provided by Sencha Touch. In this recipe we will see how to use the audio component to play a sound.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

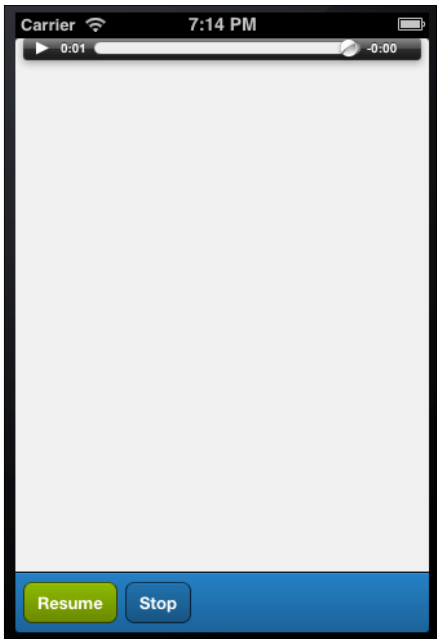
How to do it...

Follow the mentioned steps to use the audio component:

1. Create and open a new file, `ch07_02.js`, and paste the following code in it:

```
Ext.application({
 name: 'MyApp',
 launch: function() {
 var pnl = Ext.create('Ext.Container', {
     fullscreen: true,
     items: [
         {
           id: 'audio-pnl',
             xtype: 'audio',
             url  : "ch07/here-it-is.mp3"
         }, {
               xtype: 'toolbar',
               docked: 'bottom',
               items: [
                   {
                       text: 'Resume',
                       ui: 'confirm',
                       handler: function() {
                       var audioPnl = Ext.getCmp('audio-pnl');
             audioPnl.play();
                       }
                   },
                   {
                       text: 'Stop',
                       handler: function() {
                       var audioPnl = Ext.getCmp('audio-pnl');
                         audioPnl.pause();
                       }
                   }
               ]
           }],
     listeners: {
       painted: function() {
      //auto-play the audio as soon as the component is rendered
         var audioPnl = Ext.getCmp('audio-pnl');
         audioPnl.play();
       }
     }
 });

   }
});
```

2. Save your MP3 file inside the `ch07` folder and update the `url` property based on the MP3 filename.
3. Include `ch07_02.js` in place of `ch07_01.js` in the `index.html` file.
4. Deploy and access it from the browser or the device of your choice. You shall see the following screen when it is run:



How it works...

The preceding code creates an audio component using `xtype:'audio'` and the important property, `url`, is set to the path of the MP3 file that needs to be played. This, internally, uses the HTML5 audio field. By default, the audio component does not play the MP3 file. To get that working, we registered a handler for the `painted` event on the container panel and called the `play` method, explicitly, on the audio component.

Additionally, we create a docked toolbar with the **Resume** and **Stop** buttons to play and stop the audio.

[The recommended file types are: uncompressed WAV and AIF audio, MP3 audio, and AAC-LC or HE-AAC audio.]

There's more...

The audio component offers various other functionalities to control the way the audio component is rendered and the audio is played. For example, how to disable the default controls and use our own controls, how to play the audio in a loop, and so on. Let us see how we can do these things.

Hiding controls

In the previous screenshot we saw that, by default, the component shows the controls to play/pause the audio and also a slider bar that shows the audio timeline. In case you do not want these controls to appear because you want to play the audio in the background, you can set the `enableControls` config to `false`, as shown in the following code snippet:

```
...
xtype: 'audio',
enableControls: false,
...
```

Looping

Say you have an audio that you want to play as a background score as long as the user is working with your application. Since, the duration for which the user would use the application is not determined and hence you cannot create audios of that duration, you may want to play that audio in a loop. To do that, you need to set the `loop` config to `true`, as shown in the following code snippet:

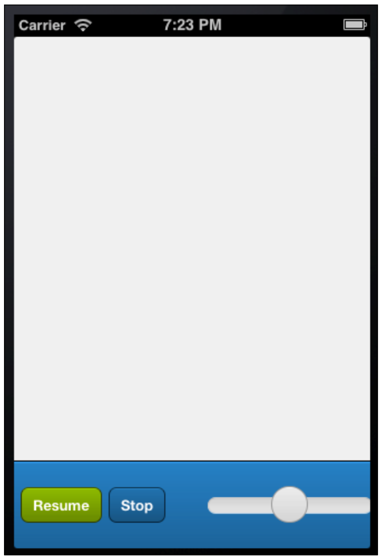
```
...
xtype: 'audio',
enableControls: false,
loop: true,
...
```

Letting the user control the volume

In case you need to have controls in your application by which the user can play/pause the audio or control the volume level, and you do not want to use the default controls because their position and their style do not match with your application requirement, then we can use the Sencha Touch components and link them with the actions on the audio. For example, the following code snippet shows that we are adding a slider for the volume control. When the slider value changes, the handler calculates the `volume` value using the slider's current position and sets it on the audio component by calling the `setVolume` method:

```
...
xtype: 'audio',
enableControls: false,
loop: true,
volume: 0.5, //default volume level
...
{
     text: 'Stop',
     handler: function() {
        var audioPnl = Ext.getCmp('audio-pnl');
        audioPnl.pause();
     }
}, {
     xtype: 'sliderfield',
     width: 200,
     value: 5,  //to match it with the volume config value
     minValue: 0,
     maxValue: 10,
     listeners: {
         change: function(thisSl, sl, thumb, newVal, oldVal) {
            var audioPnl = Ext.getCmp('audio-pnl');

            audioPnl.setVolume(newVal/10);
      }
 }
}
...
```

This is demonstrated in the following screenshot:



See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Working with videos

In this recipe, we will look at the video component to see how to use it to add video-playing capability to our application.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

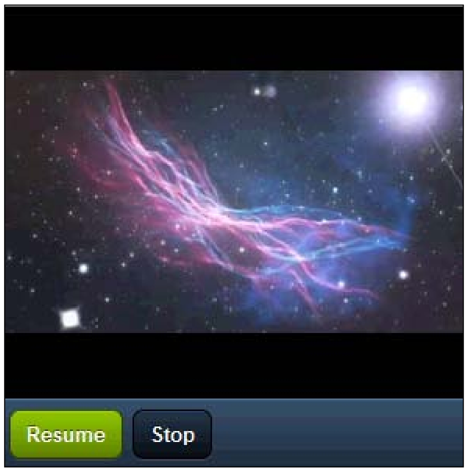
How to do it...

Follow the ensuing steps to use the video component for adding video-playing capability in our application:

1. Create and open a new file, `ch07_03.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
  var pnl = Ext.create('Ext.Container', {
    fullscreen: true,
    items: [
        {
            xtype    : 'video',
            id       : 'video-pnl',
            enableControls: false,
            url      : "ch07/space.mp4",
            posterUrl: "ch07/Screenshot.png"
        }, {
            xtype: 'toolbar',
            docked: 'bottom',
            items: [
                {
                    text: 'Resume',
                    ui: 'confirm',
                    handler: function() {
                    var videoPnl = Ext.getCmp('video-pnl');
                    videoPnl.play();
                    }
                },
                {
                    text: 'Stop',
                    handler: function() {
                    var videoPnl = Ext.getCmp('video-pnl');
                    videoPnl.pause();
                    }
                }
            ]
        }]
  });

    }
});
```

2. Save your MP4 file inside the `ch07` folder and update the `url` property based on the MP4 filename.
3. Include the `ch07_03.js` file in the `index.html` file.
4. Deploy and access it from the browser or the device of your choice and you shall see the following screen:

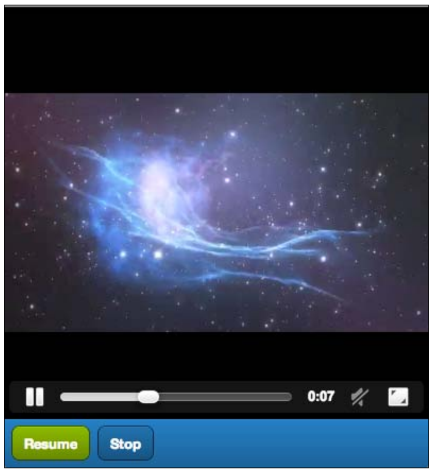


How it works...

The preceding code creates a video component using `xtype:'video'` and the important property, `url`, is set to the path of the MP4 file that needs to be played. This, internally, uses the HTML5 video field.

Additionally, we created a docked toolbar with the **Resume** and **Stop** buttons to play and stop the video.

`enableControls` allows us to control whether the control panel (with play/pause button, slider and sound buttons) shall be displayed or not. Since, in our case, we play and pause the video on a click of the toolbar buttons, **Resume** and **Stop**, we have set the property to `false`. By default, the controls are enabled. When the controls are enabled, they can be seen as shown in the following screenshot:



The video component offers other capabilities, similar to the audio component, to loop and link Sencha Touch components to the actions on the video component (for example, using slider for volume control).

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Ding-dong! You have got a message* recipe

Creating your drawing

Sencha Touch offers a drawing surface that one can use to draw any arbitrary drawing using sprites. Sprites are the basic drawing constructs—circle, rectangle, path, text and more—using which we can create any drawing. Subclasses of `Ext.draw.sprite.Sprite` are the list of sprites supported in Sencha Touch that we will use to create a drawing in this chapter.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

How to do it...

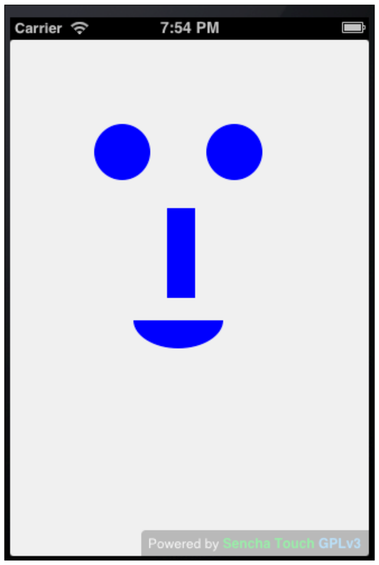
Follow the ensuing steps to create your drawing:

1. Create and open a new file, `ch07_04.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
    Ext.create('Ext.draw.Component', {
      fullscreen: true,
      items: [{
        type: 'circle',
        cx: 100,
        cy: 100,
        r: 25,
        fillStyle: 'blue'
      }, {
        type: 'circle',
        cx: 200,
        cy: 100,
        r: 25,
        fillStyle: 'blue'
      }, {
```

```
                type: 'rect',
                x: 140,
                y: 150,
                width: 25,
                height: 80,
                fillStyle: 'blue'
              }, {
                type: 'ellipticalArc',
                cx: 150,
                cy: 250,
                rx: 40,
                ry: 25,
                fillStyle: 'blue',
                startAngle: 0,
                endAngle: Math.PI,
                anticlockwise: false //shows the arc upside down
              }]
            });

          }
        });
```

2. Include `ch07_04.js` in the `index.html` file.
3. Deploy and access it from the browser or the device of your choice and you shall see the following screen:



How it works...

The preceding code creates a simple face by adding two circle, one rectangle, and one elliptic arc sprites to the drawing surface. Each sprite has got its specific config that we set while instantiating them. For example, a circle requires us to set the radius, a rectangle requires us to specify the width and height, and an elliptic arc requires us to specify the x radius (`rx`) and y radius (`ry`). We used `cx` and `cy` as the center co-ordinates to position the circle and the elliptic arc, whereas we used `x` and `y` as the top-left corner co-ordinates of the rectangle to position it on the screen.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Working with an area chart

In this recipe, we will learn about the area chart provided by Sencha Touch. This creates a stacked area chart and is useful in displaying multiple aggregated layers of information.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1, Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

In all the chart-related recipes, we will be using the following store definition to feed the data to our charts:

```
var store = Ext.create('Ext.data.Store', {
          fields: ['name', 'data1', 'data2', 'data3', 'data4',
'data5'],
          data: [
              {'name':'House Rent', 'data1':10, 'data2':12,
'data3':14, 'data4':8, 'data5':13},
              {'name':'Books', 'data1':7, 'data2':8, 'data3':16,
'data4':10, 'data5':3},
              {'name':'Petrol', 'data1':5, 'data2':2, 'data3':14,
'data4':12, 'data5':7},
              {'name':'Grocery', 'data1':2, 'data2':14, 'data3':6,
'data4':1, 'data5':23},
              {'name':'Loans & Deposits', 'data1':27, 'data2':38,
'data3':36, 'data4':13, 'data5':33}
          ]
      });
```

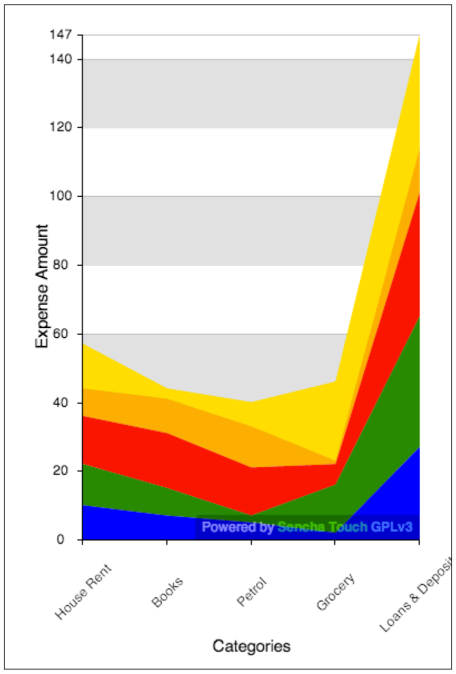
How to do it...

Follow the ensuing steps to create an area chart:

1. Create and open a new file `ch07_05.js` and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
   var store = Ext.create(...); //defined above

   var chart = Ext.create('Ext.chart.CartesianChart', {
        store: store,
          insetPadding: 30,
        axes: [{
            type: 'numeric',
            position: 'left',
            fields: ['data1'],
            title: {
                text: 'Expense Amount',
                fontSize: 15
            },
            grid: {
                  odd: {
                      opacity: 1,
                      fill: '#ddd',
                      stroke: '#bbb',
                      'stroke-width': 1
                  }
              },
            minimum: 0
        }, {
            type: 'category',
            position: 'bottom',
            fields: ['name'],
            title: {
                text: 'Categories',
                fontSize: 15
            },
            label: {
                  rotate: {
                      degrees: 315
                  }
              }
        }],
        series: [{
            type: 'area',
            title:['Month 1','Month 2','Month 3','Month
                   4','Month 5'],
            subStyle: {
                fill: ['blue', 'green', 'red', 'orange',
                       'gold']
            },
            xField: 'name',
            yField: ['data1', 'data2', 'data3', 'data4',
                     'data5']
        }]
    });
    Ext.Viewport.setLayout('fit');
    Ext.Viewport.add(chart);

    }
});
```

2. Include `ch07_05.js` in the `index.html` file.
3. Deploy and access it from the browser or the device of your choice. You will see the following screen:



How it works...

The preceding code creates the high-level `Ext.chart.CartesianChart` instance, which provides the capability to visualize the data using the Cartesian co-ordinate system based on the `x` and `y` coordinates. This object accepts four important properties—`store`, `legend`, `axes`, and `series`. `store` binds a data source to the chart so that the chart can be updated dynamically. `legend` displays a list of legend items, each of them related to a series being rendered (this is optional). `axes` contains the definition of the Cartesian axis and the field from the dataset shall be used to render the x and y axes. `series` indicates the kind of chart that needs to be rendered using the data stored in the `store` and `axes` definitions.

In the code, we have defined two axes: one of type `numeric` and the other of type `category`. The `Ext.chart.axis.Axis` class represents each entry in the axes.

For the `numeric` axis, we defined the `grid` as a config object containing the information about how the odd rows in the grid shall be rendered.

The other property that the `axes` property supports is `label`, which allows us to provide the information about how the label shall be displayed. In the preceding code, we have mentioned that the label shall be displayed at a 315 degree angle with respect to the value for the `category` axis.

Then we added the series of type `area` to create an area chart. The `xField` and `yField` properties provide the mapping of the field in the data to the axis where they should be displayed. `subStyle` is used to specify the style that needs to be applied to each sample in the series. For example, in the preceding code we have specified the color that shall be used to render each sample.

There's more...

Having the `legend` property in a chart is almost a necessity and Sencha Touch does support this in its chart functionality. In the previous discussion, we talked about the optional property on the `legend` chart object. Let us see how to use it.

Showing a legend

To show a legend on a chart, we need to do the following:

1. Add the `legend` config to the `CartesianChart` object as shown in the following code snippet:

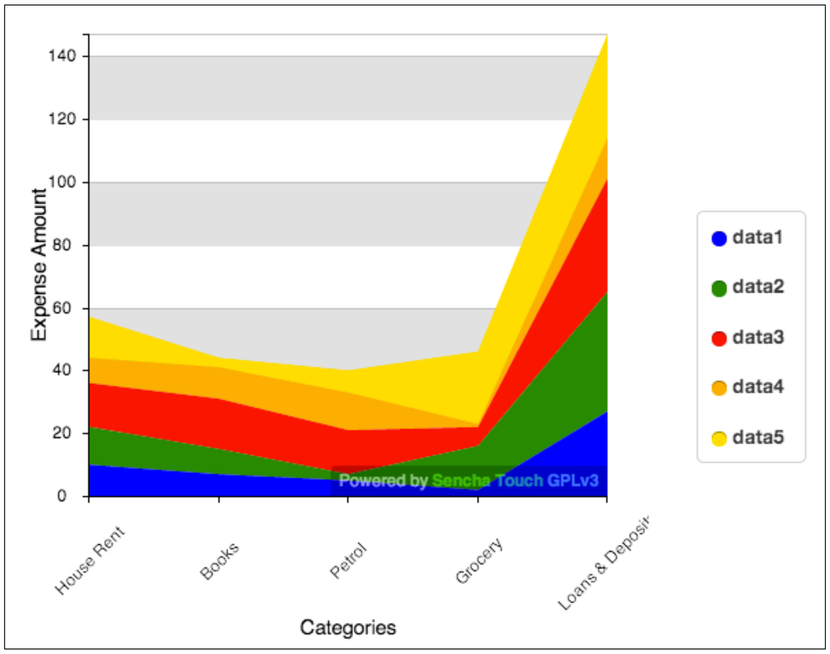
```
legend: {
  docked: Ext.os.is.Phone?'bottom':'right'
},
```

In the preceding code, we have defined a `legend` metadata where the `name` field from the dataset will be used to generate the legend and, if the view is in the `Phone` mode, the legend will be displayed at the bottom of the chart; otherwise, the legend will be displayed on the right-hand side of the chart.

2. Set the `showInLegend` property to `true` in the series.

   Though `true` is the default value set by the framework, I have mentioned it here to indicate what config can be used to indicate on a series whether it shall appear in the legend or not.

   These changes to the code will ensure that the legend is generated for the chart as shown in the following screenshot:

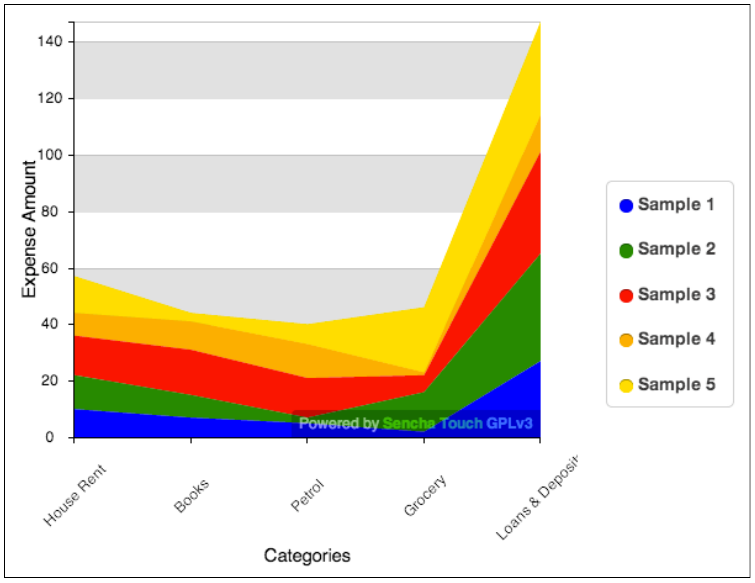


Changing the legend text

By default, the chart library uses the field name mentioned in the `yField` property in the series. In order to have a different legend text, add the `title` config to the series as shown in the following code line:

```
title:['Sample 1','Sample 2','Sample 3','Sample 4','Sample 5'],
```

In the preceding code, we have defined a title for each sample data that the chart library will use to show the legend text as shown in the following screenshot:



See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with store* recipe in *Chapter 5*, *Dealing with Data and Data Sources*

Working with a bar chart

Bar chart is another series that can be used to help a user visualize and compare data. In this recipe, we will see how to use the bar series to get a bar chart generated.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

How to do it...

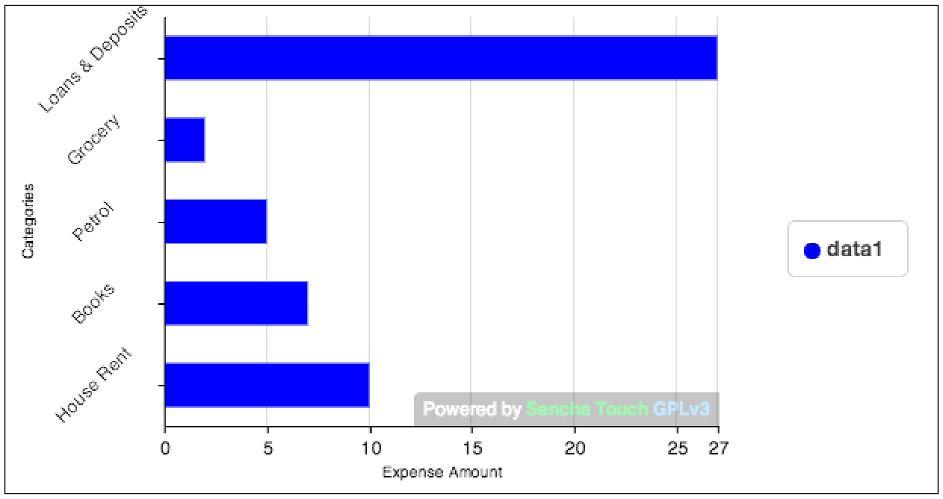
Follow these steps to generate a bar chart:

1. Create and open a new file, `ch07_06.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
   var store = Ext.create(...); //defined in Area Chart
                                   recipe

   var chart = Ext.create('Ext.chart.CartesianChart', {
        store: store,
        flipXY: true,
        legend: {
            docked: Ext.os.is.Phone?'bottom':'right'
        },
        axes: [{
            type: 'numeric',
            position: 'bottom',
            fields: ['data1'],
            title: 'Expense Amount',
            grid: true,
            minimum: 0
        }, {
            type: 'category',
            position: 'left',
            fields: ['name'],
            title: 'Categories',
            label: {
                  rotate: {
                      degrees: 315
                  }
              }
        }],
        series: [{
```

```
                    type: 'bar',
                    xField: 'name',
                    yField: ['data1'],
                    subStyle: {
                        fill: ['blue']
                    }

                }]
            });

            Ext.Viewport.setLayout('fit');
            Ext.Viewport.add(chart);

        }
    });
```

2. Include `ch07_06.js` in the `index.html` file.
3. Deploy and access it from the browser or the device of your choice. You will see the following screen:



How it works...

In the preceding code, we created an instance of the `CartesianChart` object with the axis and series information. In the series, we set the `type` value to `bar` to get the bar chart generated from the dataset. By default, the framework produces a column chart when a bar series is used where the bars are shown vertically. To show a column chart, we have to set `flipXY` to `true` on the chart; this will flip the x and y axis to show horizontal bars.

There's more...

The default spacing between the bars is derived by the bar series. However, if there is a need to increase or decrease the gap between them, here is the way to do that.

Limiting the bar width

The bar's minimum width is defaulted to 2 pixels and the maximum width is defaulted to 100 pixels. In case we want to modify these defaults, we can do that by setting `minBarWidth` and `maxBarWidth` on the bar series as part of the `style` config, as shown in the following code snippet:

```
series: [{
    type: 'bar',
    xField: 'name',
    yField: ['data1'],
    subStyle: {
        fill: ['blue']
    },
    style: {
        maxBarWidth: 30
    }
 }]
```

Using rounded corners

In case you want to show rounded corners for the bars, you can do so by specifying the radius on the `style` config of the series. It accepts a numeric value in pixels. For example, if radius is set to `5` in the series' `style` configuration, the corners of the bar are rounded to 5 pixels.

Changing the spacing between the bars

By default, 5 pixels is the minimum gap between the two bars. To increase/decrease the gap between the bars, you can set the `minGapWidth` property in the series' `style` configuration. It accepts a numeric value in pixels. For example, if `minGapWidth:2` is set in the series configuration, the spacing between two bars will be `2` pixels.

You may refer to the `Bar` sprite class (`Ext.chart.series.sprite.Bar`) to learn more about the different configurations that you can pass as part of the `style` config in the series.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with store* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Working with an area chart* recipe

Working with a column chart

Column chart is extended from the bar series and displays the chart in the form of vertical bars. In this recipe, we will see how to create a column chart.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

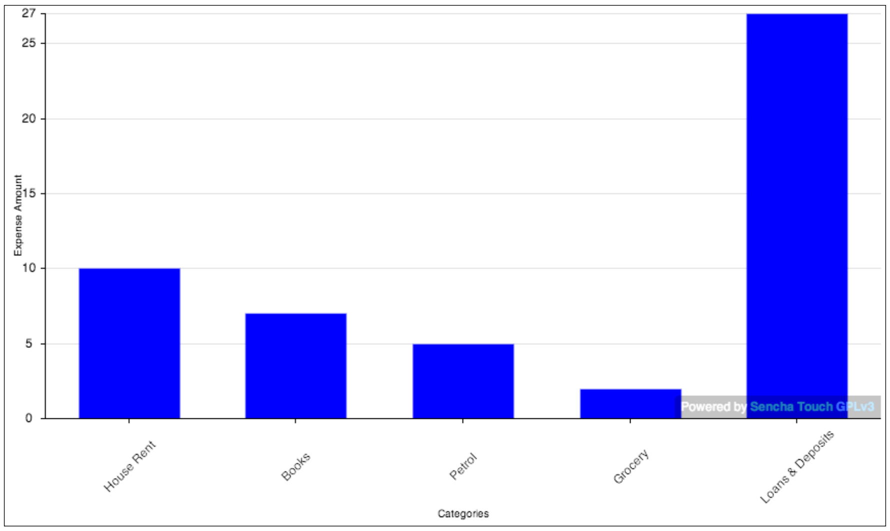
How to do it...

Follow the ensuing steps to create a column chart:

1. Create and open a new file, `ch07_07.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
   var store = Ext.create(...); //defined in Area Chart recipe

   var chart = Ext.create('Ext.chart.CartesianChart', {
          store: store,
          axes: [{
              type: 'numeric',
              position: 'left',
              fields: ['data1'],
              title: 'Expense Amount',
              grid: true,
              minimum: 0
            }, {
              type: 'category',
              position: 'bottom',
              fields: 'name',
              title: 'Categories',
            label: {
                  rotate: {
                      degrees: 315
                  }
              }
            }],
            series: [{
              type: 'bar',
              xField: 'name',
              yField: ['data1'],
              colors: ['blue']
            }]

  });
      Ext.Viewport.setLayout('fit');
      Ext.Viewport.add(chart);

    }
});
```

2. Include the `ch07_07.js` in the `index.html` file.
3. Deploy and access it from the browser or the device of your choice. You will then see the following screen:



How it works...

The preceding code creates an instance of a chart with the axis and a series of type `column`.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with store* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Working with an area chart* recipe

Showing a group of bars and columns

Our dataset contains the `data1`, `data2`, `data3`, `data4`, and `data5` fields, besides the `name` field. Suppose, for every month for your monthly expenses, `data1` represents the actual expense, whereas `data2` represents the estimated expense, and we want to see the visuals for both the values being presented for each of the expense categories. That is, we need to show a group of bars for each category. In the bar and column charts, Sencha Touch supports showing a group of bars in the place of a single bar. This recipe will show you how to do that.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*. Also, complete the *Working with a column chart* recipe.

How to do it...

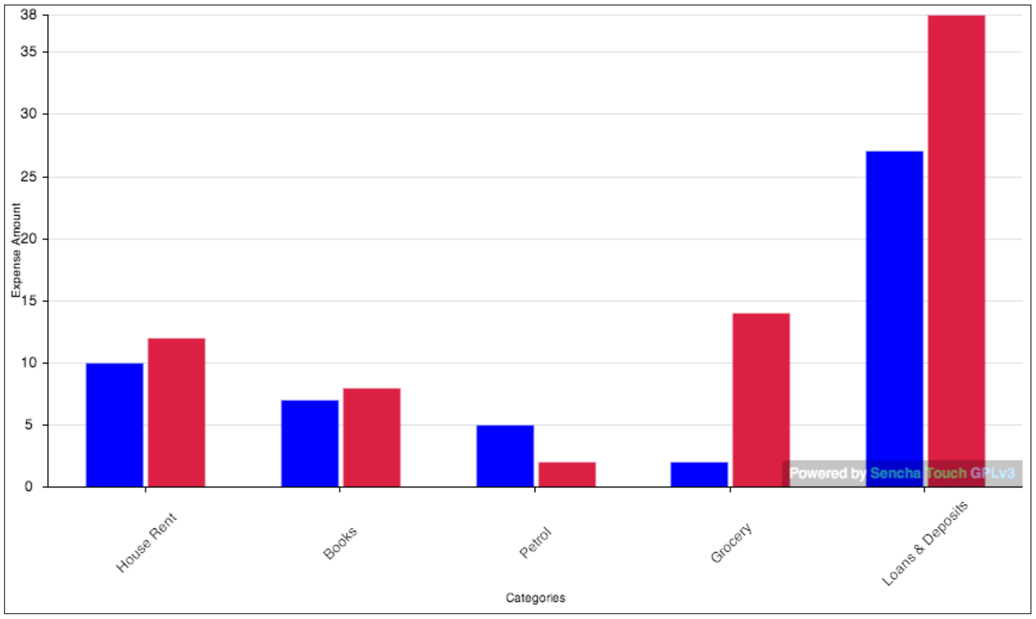
Follow the ensuing steps to show a group of bars and columns:

1. Edit the `ch07_07.js` file and add the properties, highlighted in bold, as mentioned in the following code snippet:

```
axes: [{
              type: 'numeric',
              position: 'left',
              fields: ['data1'],
              title: 'Expense Amount',
              grid: true,
              minimum: 0
            }, {
```

```
  type: 'category',
  position: 'bottom',
  fields: 'name',
  title: 'Categories',
label: {
      rotate: {
          degrees: 315
      }
  }
}],
series: [{
  type: 'bar',
     stacked: false,
  xField: 'name',
     yField: ['data1', 'data2'],
     colors: ['blue', 'crimson']
}]
```

2. Deploy and access the application from the browser or the device of your choice. You will see the following screen:



How it works...

In the preceding code, we set the `stacked` property to `false` so that the bars appear as a group rather than stacked. We had to list the additional sample `data2` to the `yField` property so that the bars are generated for this sample. And finally, we added `crimson` as the color for the bar representing the `data2` sample.

There's more...

The default spacing between the bars is derived by the bar series. However, if there is a need to increase or decrease the gapping between them, here is the way to do that.

Changing the spacing between grouped bars

The bar sprite provides an `inGroupGapWidth` config to specify the gap that we would like to have between the bar groups. The value can be specified as numeric in pixels. The following code snippet shows how to set the gap to `0` pixel:

```
series: [{
            ...
          },
           style: {
                  inGroupGapWidth: 0
                }
```

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with store* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Working with an area chart* recipe
- The *Working with a column chart* recipe

Highlighting and displaying an item detail

The next level of interaction is that, when the user clicks on a chart item, we may want to highlight that item and show the item detail corresponding to it. In this recipe, we will see how to achieve this.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*. Also, complete the *Working with a column chart* recipe.

How to do it...

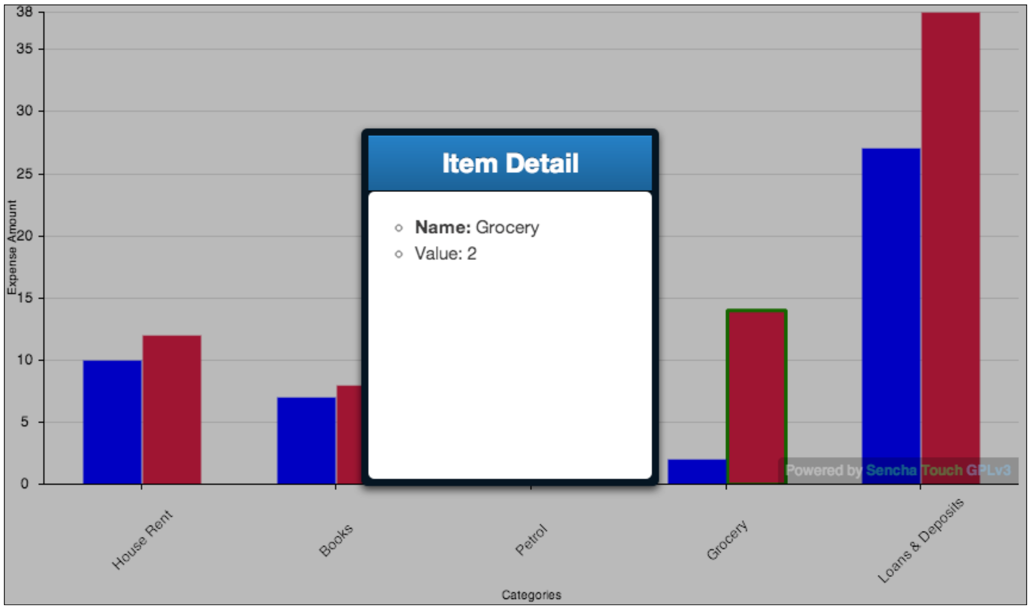
Follow these steps to highlight and display an item detail:

1. Edit the `ch07_07.js` file.
2. Add the following interactions to the chart's `interactions` array:

```
interactions: [{
        type: 'iteminfo',
        gesture: 'itemtaphold',
        listeners: {
           show: function(me, item, panel) {
                 var rec = item.record;

                 var str = '<ul><li><span style="font-weight:
bold">Name: </span>' + rec.get('name') + '</li>';
                 str += '<li>Value: ' + rec.get(item.field) + '</
li></ul>';
                 panel.setHtml([str].join(''));
            }
        }
},
        'itemhighlight']
```

3. Save the change.
4. Deploy and access it from the browser or the device of your choice.

5. Perform a single tap on a bar. This will highlight the bar.
6. Tap on the bar and hold it for a while. You will see a pop up showing the item detail as shown in the following screenshot:



Item Detail

- **Name:** Grocery
- Value: 2

How it works...

The code uses two of the in-built interactions—`itemhighlight` and `iteminfo`. The default handler of the `iteminfo` interaction shows a pop-up panel with **Item Detail** as the title. This panel is passed to the `show` event handler as the third argument, which we updated with the selected item detail. The `field` property on the item contains the bar on which the `itemtaphold` event has occurred. For example, it will contain `data1` or `data2` as values, based on the bar where the event has occurred. We used this field information to fetch the value from the record.

There's more...

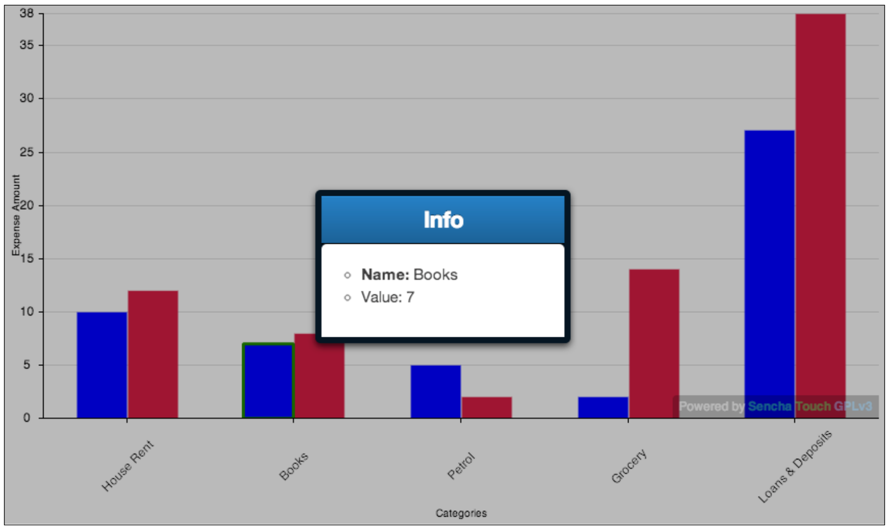
The `panel` object that is passed to the show event, in the preceding code, has the default title, style, items, and dimension. Say we want to show a different title or we want to show some other component inside the body of the panel. Let us see how we can customize this panel configuration.

Customizing the Item Detail panel

The `iteminfo` interaction supports the `panel` config that allows us to specify our own configuration for the panel and it is merged with the default configuration. The following code snippet shows how we can set a different title and override the default height of the panel:

```
interactions: [
                {
                  type: 'iteminfo',
                  gesture: 'itemtaphold',
                  panel: {height: 150, items: [{docked: 'top',
                          xtype: 'toolbar', title: 'Info'}]},
                     listeners: {
```

This is demonstrated in the following screenshot:



- **Name:** Books
- Value: 7

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with store* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Working with an area chart* recipe
- The *Working with a column chart* recipe

Working with a gauge chart

Gauge charts are used to show progress in a certain variable. In this recipe, we will walk through the steps to create a gauge chart.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

How to do it...

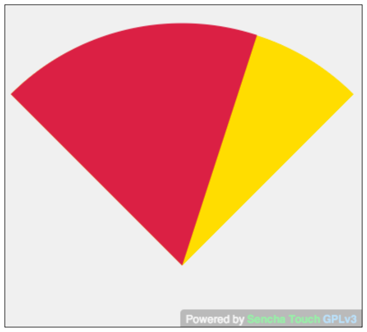
Follow these steps to create a gauge chart:

1. Create and open a new file, `ch07_08.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
   var store = Ext.create(...); //defined in Area Chart recipe

      var chart = Ext.create('Ext.chart.SpaceFillingChart', {
          fullscreen: true,
          store: store,
        series: [{
            type: 'gauge',
            maximum: 20,
            field: 'data3',
            colors: ['crimson', 'gold']
        }]
  });

    }
});
```

2. Include `ch07_08.js` in the `index.html` file.

3. Deploy and access it from the browser. You may also run it using the emulator. You will then see the following screen:



How it works...

The preceding code creates a `SpaceFillingChart` object with series of type `gauge`.

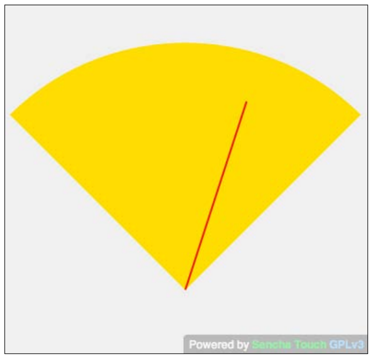
In the series, after the `type` property, the next important property is `field`, which contains the field name of the record that is used for the gauge angles. The value must be a positive real number. `colors` is used to specify the colors that will be used to render the sections/pies of the gauge chart.

There's more...

There are some more useful properties of the gauge chart that are worth a discussion.

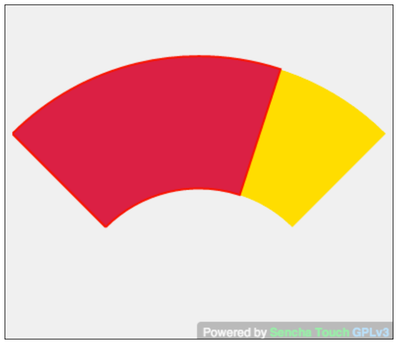
Showing a needle

On a gauge, sometimes you may want to see a needle to show something like a dial chart. This can be achieved by setting the `needle` property to `true` on the gauge series.



The donut effect

The donut effect can be created by setting the `donut` property on the gauge series to a value that is used as the radius of the inner circle. For example, this is how the gauge will look if we set the `donut` property to `50`:



See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with store* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Working with an area chart* recipe

Working with a line chart

This recipe is all about creating a line chart using the Sencha Touch chart library.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

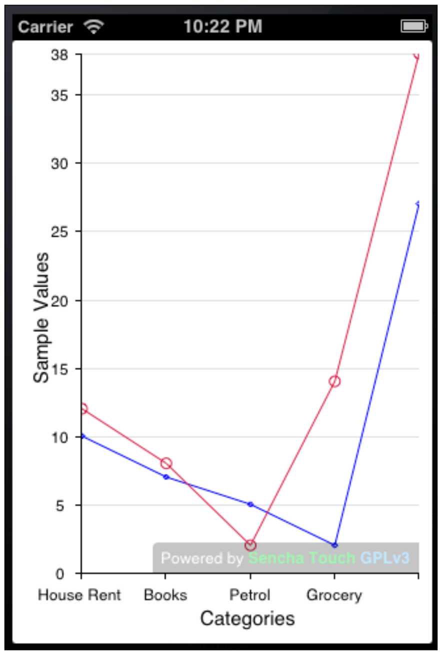
How to do it...

Follow these steps to create a line chart:

1. Create and open a new file, `ch07_09.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',
  launch: function() {
   var store = Ext.create(...); //defined in Area Chart recipe

   var chart = Ext.create('Ext.chart.CartesianChart', {
        store: store,
        axes: [{
            type: 'numeric',
            position: 'left',
            fields: ['data1', 'data2'],
            title: {
                text: 'Sample Values',
                fontSize: 15
            },
            grid: true,
            minimum: 0
        }, {
            type: 'category',
            position: 'bottom',
            fields: ['name'],
            title: {
                text: 'Categories',
                fontSize: 15
            }
        }],
        series: [{
            type: 'line',
            style: {
                stroke: 'blue'
            },
            xField: 'name',
            yField: 'data1',
            marker: {
                type: 'path',
                path: ['M', -2, 0, 0, 2, 2, 0, 0, -2, 'Z'],
                stroke: 'blue',
                lineWidth: 0
            }
        }, {
            type: 'line',
            highlightCfg: {
                size: 7,
                radius: 7
            },
            style: {
                stroke: 'crimson'
            },
            xField: 'name',
            yField: 'data2',
            marker: {
                type: 'circle',
                radius: 4,
                lineWidth: 0
            }
        }],
          interactions: ['itemhighlight']
    });
    Ext.Viewport.setLayout('fit');
    Ext.Viewport.add(chart);

    }
});
```

2. Include `ch07_09.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will then see the following screen:



How it works...

The preceding code creates a `CartesianChart` object with a `line` type series with various interactions. In the series, the `highlightCfg` config object defines the configuration, which is used to highlight the line and the nodes when the user taps on a particular line series. It will show the circle with a radius of `7` pixels. We configured the `itemhightlight` interaction to highlight the series.

We used the `marker` config to set up different markers for the nodes on each series. On one series we used a `path` sprite to create a marker, whereas on the other one we used a `circle` sprite to create the marker.

There's more...

Let us look at some of the additional useful properties of the line chart.

Filling the area

To fill the area under a line series, we shall set the `fill` property to `true` on that particular line series. For example, if we want to show the color under our first line series, we will set the property on it as shown in the following code snippet:

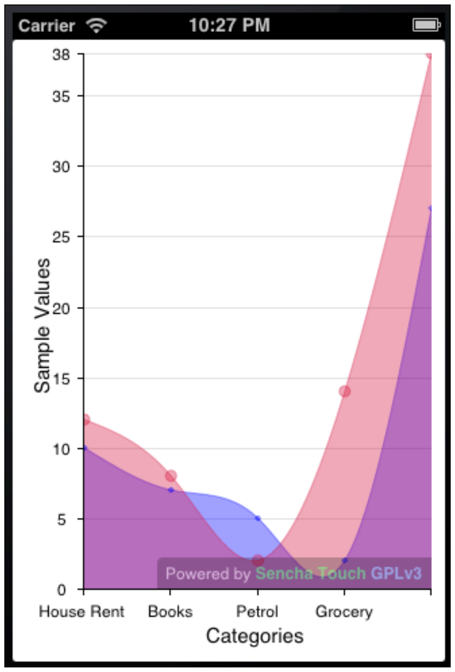
```
series: [{
            type: 'line',
            fill: true,
            highlight: {
                size: 7,
                radius: 7
            …
        }
```

Smoothing curves

By default, curves will have edges. In case we want smooth curves resembling the ones drawn by Bezier or B-Spline curves, we must set the property `smooth` to `true` on the desired line series. The following code snippet shows how to set this property:

```
series: [{
            type: 'line',
            fill: true,
            smooth: true,
            highlight: {
                size: 7,
                radius: 7
            …
        }
```

Once these properties are set, you will see the chart as shown in the following screen:



Using cross-zoom to see more detail

To enable cross-zoom on the line chart, we need to add `crosszoom` to the `interactions` array since it is offered as one of the interactions.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Working with stores* recipe in *Chapter 5, Dealing with Data and Data Sources*
- The *Working with an area chart* recipe

Working with a pie chart

This recipe shows how to create a pie chart and work with some interesting features offered by the framework.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1, Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

How to do it...

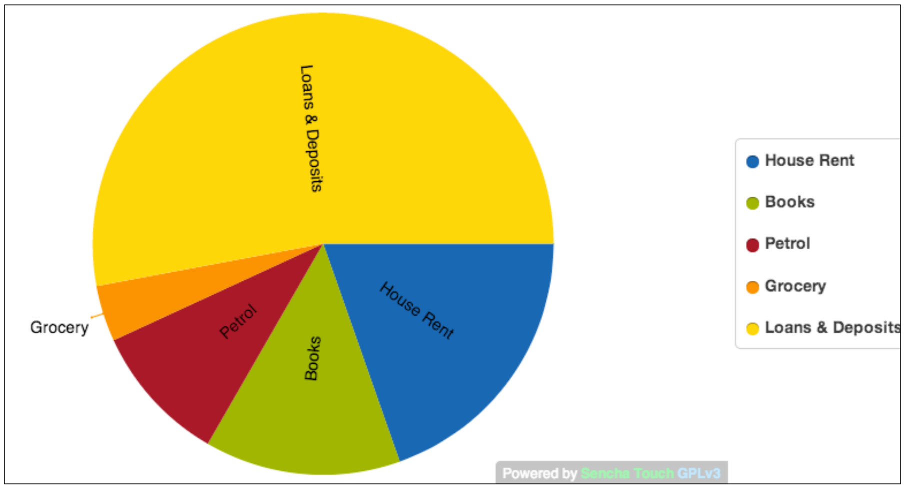
Follow these steps to create a pie chart:

1. Create and open a new file, `ch07_10.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
   var store = Ext.create(...); //defined in Area Chart recipe

      var chart = Ext.create('Ext.chart.PolarChart', {
        store: store,
        colors: ["#115fa6", "#94ae0a", "#a61120",
                   "#ff8809", "#ffd13e"],
        legend: {
              docked: Ext.os.is.Phone ? 'bottom' : 'right'
          },
          series: [{
            type: 'pie',
            labelField: 'name',
            xField: 'data1',
        }]
  });
      Ext.Viewport.setLayout('fit');
    Ext.Viewport.add(chart);

    }
});
```

2. Include `ch07_10.js` in the `index.html` file.

3. Deploy and access it from the browser. You may also run it using the emulator. You will then see the following screen:



How it works...

The preceding code creates a `PolarChart` object with the series of type `pie`. The `xField` property is the record field, which is used to calculate the angle. To show the legend, the `legend` configuration is specified in the chart. `labelField` is the record field name whose value shall be used as the label text.

There's more...

Let us look at some of the other properties that might be of interest.

The donut effect

Similar to the gauge chart, setting the `donut` property on the `pie` series with a positive numeric value can create the donut effect.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Working with store* recipe in *Chapter 5, Dealing with Data and Data Sources*
- The *Working with an area chart* recipe

Rotating the pies

Coming to interactions with a pie chart, rotation allows the user to rotate the pie chart to view a pie from a particular position. In this recipe, we will learn how to achieve this.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*. Also, complete the *Working with a pie chart* recipe.

How to do it...

Follow these steps to rotate a pie chart:

1. Edit the `ch07_10.js` file.
2. Add the following interaction item to it:
   ```
   interactions: ['rotate']
   ```
3. Save the changes.
4. Deploy and access the application from the browser. You may also run it using the emulator.
5. Use single-finger or mouse drag, based on your device, around the center of the series. You will see the pie chart rotating.

How it works...

This is taken care of by the default `rotate` interaction of the Sencha Touch chart framework.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with store* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Working with an area chart* recipe
- The *Working with a pie chart* recipe

Highlighting a pie

One cool feature the pie chart supports is highlighting a pie; when it is selected, it actually stands out distinctly from other pies. In this recipe, we will see how to make use of this feature.

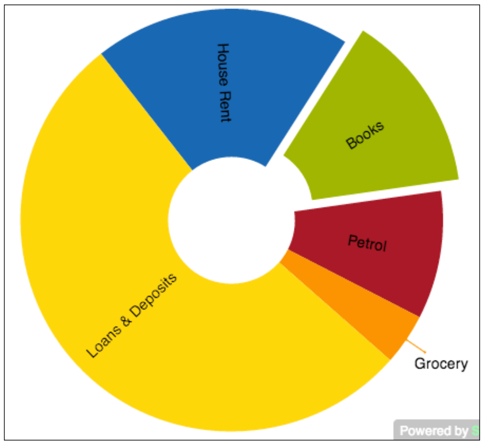
Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*. Also, complete the *Working with a pie chart* recipe.

How to do it...

Follow these steps to highlight a pie chart:

1. Edit the `ch07_10.js` file.
2. Add the following interaction item to it:
   ```
   interactions: ['itemhighlight']
   ```
3. Set the following config on the `pie` series:
   ```
   highlightCfg: {
                     margin: 20
                 }
   ```
4. Save the changes.
5. Deploy and access the application from the browser. You may also run it using the emulator.
6. Tap a particular pie. This will show the pie highlighted and standing out from the rest of the chart as shown in the following screenshot:



How it works...

This is taken care of by the default `itemhighlight` interaction on the pie series of the Sencha Touch chart framework. Setting the `margin` value on the `highlightCfg` config will tell the framework how much the margin of the pie should be from the center when it is highlighted.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with stores* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Working with an area chart* recipe
- The *Working with a pie chart* recipe

Working with a 3D pie chart

This recipe shows how to create a pie chart and work with some interesting features offered by the framework.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

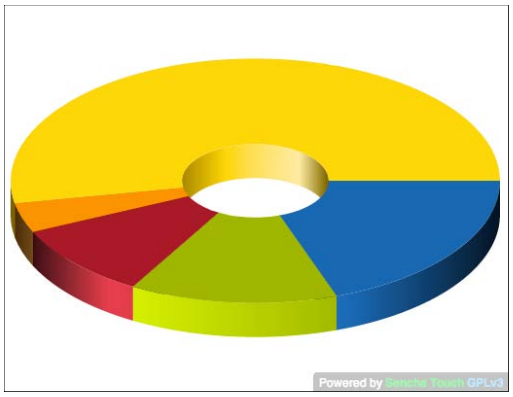
How to do it...

Follow these steps to create a 3D pie chart:

1. Create and open a new file, `ch07_11.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
   var store = Ext.create(...); //defined in Area Chart recipe

      var chart = Ext.create('Ext.chart.PolarChart', {
        store: store,
        innerPadding: 45,
        colors: ["#115fa6", "#94ae0a", "#a61120", "#ff8809",
"#ffd13e"],
      interactions: ['rotatePie3d'],
              series: [{
              type: 'pie3d',
            labelField: 'name',
            field: 'data1',
            donut: 30
        }]
  });
      Ext.Viewport.setLayout('fit');
    Ext.Viewport.add(chart);

    }
});
```

2. Include `ch07_11.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will see the following screen:



How it works...

The preceding code creates a `PolarChart` object with series of type `pie3d`. The `field` property is the record field, which is used to calculate the angle. The `legend` configuration is specified on the chart to show the legend.

The `labelField` config contains the information about how the label for each pie shall be generated. `labelField` is the record field name whose value shall be used as the label text. `donut` is set in the series to create a donut effect and `rotatePie3d` is set as the interaction, which allows the user to rotate the pie with a drag/swipe.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with stores* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Working with an area chart* recipe

Working with a radar chart

A radar chart is a useful visualization technique for comparing different quantitative values for a constrained number of categories and this recipe is going to show us how to create one.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

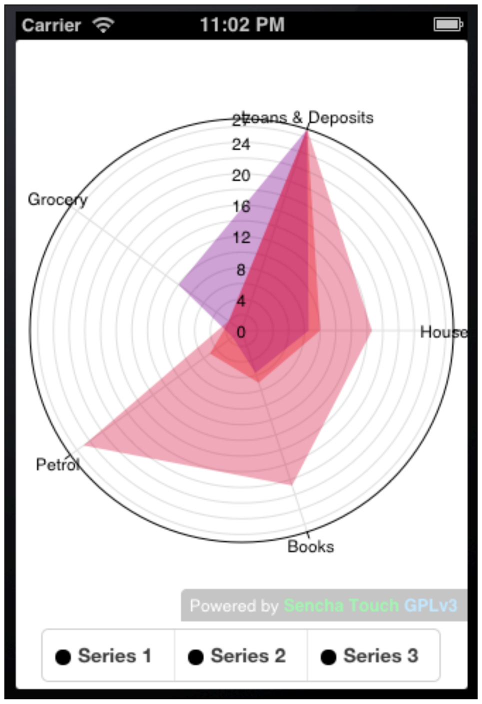
How to do it...

Follow these steps to create a radar chart:

1. Create and open a new file, `ch07_12.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
   var store = Ext.create(...); //defined in Area Chart recipe

      var chart = Ext.create('Ext.chart.PolarChart', {
        store: store,
        axes: [{
            type: 'numeric',
            position: 'radial',
            fields: 'data1',
            style: {
                estStepSize: 10
            },
            grid: true
        }, {
            type: 'category',
            position: 'angular',
            fields: 'name',
            grid: true
          }],
          legend: {
          docked: 'bottom'
        },
        series: [{
            type: 'radar',
            title: 'Series 1',
            xField: 'name',
            yField: 'data1',
            style: {
                'stroke-width': 2,
                fill: 'red',
                opacity: 0.4
            }
        },{
            type: 'radar',
            title: 'Series 2',
            xField: 'name',
            yField: 'data2',
            style: {
                'stroke-width': 2,
                fill: 'purple',
                opacity: 0.4
            }
        },{
            type: 'radar',
            title: 'Series 3',
            xField: 'name',
            yField: 'data4',
            style: {
                'stroke-width': 4,
                fill: 'crimson',
                opacity: 0.4
            }
        }]
  });
      Ext.Viewport.setLayout('fit');
    Ext.Viewport.add(chart);

    }
});
```

2. Include `ch07_12.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will then see the following screen:



How it works...

The preceding code creates a `PolarChart` object with the series of type `radar`. `xField` and `yField` are the record field names that are used to render the radar. `style` is used to style each of the series using the properties defined in it. In the `style` config object we set the `fill` color and the `transparency` level.

There's more...

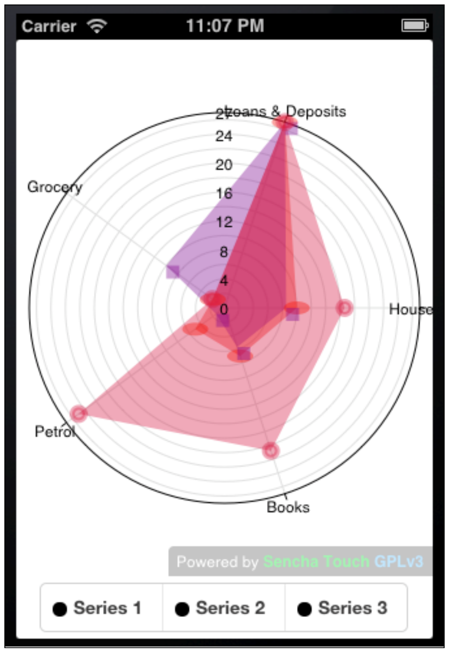
By default, the series does not show markers. In case we want to show markers and use different markers for different series, following is how we will do it:

Using a different marker

The kind of marker to be used is derived from the `type` property of the `marker` object. The following code, in bold, shows using the `ellipse`, `rect`, and `circle` as different markers on different series:

```
series: [{
    type: 'radar',
    title: 'Series 1',
    xField: 'name',
    yField: 'data1',
    marker: {
            type: 'ellipse',
            rx: 10,
            ry: 5,
            fillStyle: 'red'
    },
    ...
        },{
            type: 'radar',
            title: 'Series 2',
            ...,
            marker: {
              type: 'rect',
              width:10,
              height:10,
              fillStyle: 'purple'
            },
            ...
        },{
            type: 'radar',
            title: 'Series 3',
            ...,
            marker: {
              type: 'circle',
              r:5,
              stroke: 'crimson'
            },
            ...
        }]
```

Once set, the following screenshot shows what the new marker will look like:



See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Working with stores* recipe in *Chapter 5, Dealing with Data and Data Sources*
- The *Working with an area chart* recipe

Working with a scatter chart

The scatter chart is useful when trying to display more than two variables in the same visualization. This recipe will show us how to work with a scatter chart.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1, Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

How to do it...

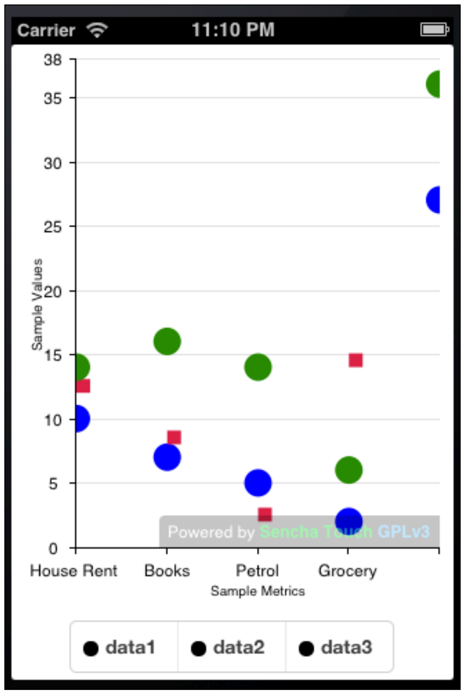
Follow these steps to create a scatter chart:

1. Create and open a new file, `ch07_13.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
   var store = Ext.create(...); //defined in Area Chart recipe

      var chart = Ext.create('Ext.chart.CartesianChart', {
        store: store,
        axes: [{
            type: 'numeric',
            position: 'left',
            fields: ['data1', 'data2', 'data3'],
            title: 'Sample Values',
            grid: true,
            minimum: 0
        }, {
            type: 'category',
            position: 'bottom',
            fields: ['name'],
            title: 'Sample Metrics'
        }],
          legend: {
              docked: 'bottom'
        },
        series: [{
            type: 'scatter',
            xField: 'name',
            yField: 'data1',
            fill: true,
            marker: {
                type: 'circle',
                fillStyle: 'blue',
                radius: 10,
                lineWidth: 0
            }
        }, {
            type: 'scatter',
            marker: {
              type: 'rect',
              width: 10,
              height: 10,
                fillStyle: 'crimson'
            },
            axis: 'left',
            xField: 'name',
            yField: 'data2'
        }, {
            type: 'scatter',
            marker: {
              type: 'circle',
              radius: 10,
                fillStyle: 'green'
            },
            axis: 'left',
            xField: 'name',
            yField: 'data3'
        }],
          interactions: [{
                    type: 'panzoom',
                    zoomOnPanGesture: true
                },
                {
                    type: 'iteminfo',
                    gesture: 'itemtaphold'
                }]
  });
      Ext.Viewport.setLayout('fit');
    Ext.Viewport.add(chart);

    }
});
```

2. Include `ch07_13.js` in the `index.html` file.

3. Deploy and access it from the browser. You may also run it using the emulator. You will then see the following screen:



How it works...

The preceding code creates a `CartesianChart` object with a series of type `scatter`. Each series contains the `xField` and `yField` properties, which are set to record data fields. `marker` contains the markers that need to be used for each `scatter` series.

Also, there are two interactions added—`panzoom` and `iteminfo`. Since there is no handler written for the show event in case of `iteminfo`, `itemtaphold` will show a blank pop up with the title **Item Detail**.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with stores* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Working with an area chart* recipe

Working with a candlestick/OHLC chart

Candlestick or OHLC (Open-High-Low-Close) are typically used to describe price movements of a security, derivative, or currency over time. This is a new chart introduced in Sencha Touch 2, and that we will cover in this recipe.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder, `ch07`, in the `www` folder. We will be using this new folder to keep the code.

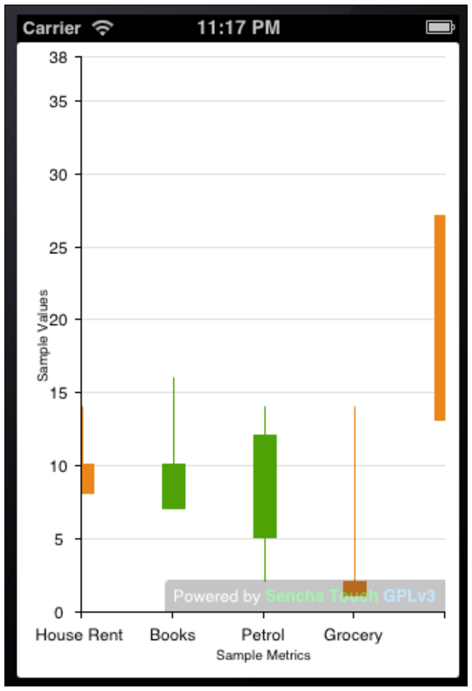
How to do it...

Follow these steps to create a candlestick/OHLC chart:

1. Create and open a new file, `ch07_14.js`, and paste the following code in it:

```
Ext.application({
  name: 'MyApp',

  launch: function() {
   var store = Ext.create(...); //defined in Area Chart recipe

      var chart = Ext.create('Ext.chart.CartesianChart', {
        store: store,
        axes: [{
            type: 'numeric',
            position: 'left',
            fields: ['data1', 'data2', 'data3', 'data4',
                     'data5'],
            title: 'Sample Values',
            grid: true,
            minimum: 0
        }, {
            type: 'category',
            position: 'bottom',
            fields: ['name'],
            title: 'Sample Metrics'
        }],
        series: [{
            type: 'candlestick',
            xField: 'name',
            openField: 'data1',
            highField: 'data2',
            lowField: 'data3',
            closeField: 'data4',
            fill: true,
            style: {
                dropStyle: {
                  fill: 'rgb(237, 123, 43)',
                  stroke: 'rgb(237, 123, 43)'
                },
                raiseStyle: {
                  fill: 'rgb(55, 153, 19)',
                  stroke: 'rgb(55, 153, 19)'
                }
              }
        }]
  });
    Ext.Viewport.setLayout('fit');
    Ext.Viewport.add(chart);

    }
});
```

2. Include `ch07_14.js` in the `index.html` file.
3. Deploy and access it from the browser. You may also run it using the emulator. You will then see the following screen:



How it works...

The preceding code creates a chart object with a series of type `scatter`. Each series contains the `xField` and `yField` properties, which are set to the record data fields. `markerConfig` contains the markers that need to be used for each `scatter` series.

Also, there are two interactions added—`panzoom` and `iteminfo`. Since there is no handler written for the show event in case of `iteminfo`, `taphold` will show a blank pop up with the title **Item Detail**.

`dropStyle` is used to style the candle if the security closed lower than it opened (drop) and `raiseStyle` is used to style the candle if the security closed higher than it opened (raise).

There's more...

Let us see what needs to be done to show the OHLC chart and how we can control the width of the candle bar.

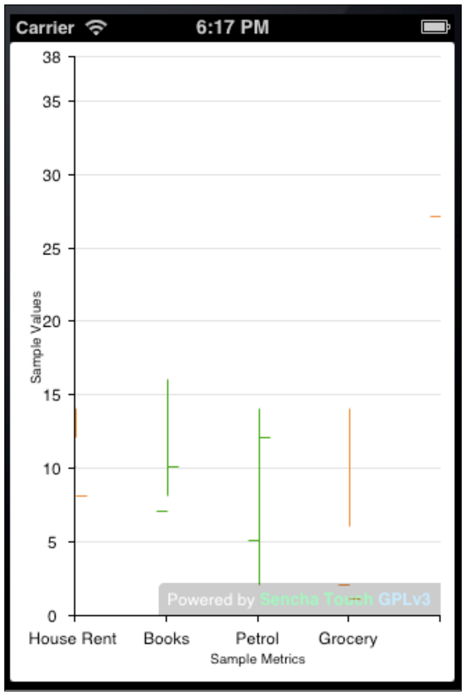
Using OHLC charts

The OHLC chart is used to show the price movement of a financial instrument over time. It is very similar to the candlestick chart but shows a tick on the left-hand side for the opening prices and a tick on the right-hand side for the closing prices. Also, the body is rendered as a line rather than a bar.

To show the OHLC chart, set the `ohlcType` config to `ohlc` in the `style` property of the candlestick series, as shown in the following code snippet:

```
style: {
        ohlcType: 'ohlc',
        dropStyle: {
                fill: 'rgb(237, 123, 43)',
...
```

The following screenshot shows what the OHLC chart will look like:



Changing the bar width

To change the width of the candlestick body bar, we must set the `barWidth` config on the `style` config of the series. It accepts a numeric value in pixels.

See also

- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Working with stores* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Working with an area chart* recipe

8
Taking Your Application Offline

In this chapter, we will cover:

- Detecting offline mode
- Storing your data offline in localstorage
- Storing your data offline using Sencha.io
- Storing your images offline
- Application caching

Introduction

When it comes to building mobile applications, there is a special case that makes the mobile application different from today's desktop applications: **Offline mode** or the **Flight mode**. Though this Offline mode was used on desktop applications for some time, we had intermittent or slow network connections. However, they are now more common to the mobile world. This mode means the mobile is not on the network and does not have access to the WLAN or GPRS data connection.

There are various cases where the user will need to use the application without network connectivity. For example, imagine that you are a field maintenance staff and you work in areas where there is no network coverage. However, you need to carry the list of customers, their orders, and the detail containing the list of products the customer has ordered, the quantity, prices, and so on. As field maintenance staff, you are expected to fulfill the order, collect the payment, and issue a receipt to customers. And in your company, the order is created in a centralized ERP system.

In this case, it would be impossible to manage things electronically if the offline application to enable the field maintenance staff were not there. A typical offline application can help maintenance staff to download the orders for a day on their mobiles. It enables them to update the order status locally on their mobile and create and issue a receipt to the customer after the order is completed. And when they come back to their office, they can sync the updated orders and the receipts and other updates with the centralized system. I am sure there can be many more interesting scenarios where offline applications would be useful. The bottom line is that having an offline capability in our application makes lot of sense, and it is a powerful feature to have in a mobile application.

A typical touch application consists of one or more JavaScript files, one or more CSS files, and work with the data and images. Taking this application offline means all these things need to be available on the local device and should be stored in such a way that the absence of the network does not make any difference to the application. In this chapter, we will see how to take our application completely offline and learn how to model our application for online and offline mode support.

Detecting offline mode

The life of an offline application starts with identifying whether the device/browser is online or offline and, based on that, taking the appropriate action. In this recipe we will see the different ways in which we can identify whether the device or the browser is online or offline, which will help us make decisions in the subsequent recipes. We will start with using Cordova API to detect the mode and then later look at other alternatives.

We have already set up our project with Cordova support as part of the setup in *Chapter 1, Gear Up for the Journey*. You may refer to `http://cordova.apache.org/docs/en/` for more details on its APIs.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1, Gear Up for the Journey*.

Create a new folder, `ch08`, inside the `www` folder. We will be using this new folder to keep the code.

How to do it...

Carry out the following steps for detecting offline mode:

1. Create and open a new file, `ch08_01.js`, and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    function onDeviceReady() {
      checkConnection();
    }
    function checkConnection() {
      var networkState = navigator.connection.type;
      var states = {};
      states[Connection.UNKNOWN] = 'Unknown connection';
      states[Connection.ETHERNET] = 'Ethernet connection';
      states[Connection.WIFI] = 'WiFi connection';
      states[Connection.CELL_2G] = 'Cell 2G connection';
      states[Connection.CELL_3G] = 'Cell 3G connection';
      states[Connection.CELL_4G] = 'Cell 4G connection';
      states[Connection.NONE] = 'No network connection';
      var str = (navigator.onLine ? 'ONLINE' : 'OFFLINE') +
        ' - '+ states[networkState];
      Ext.Msg.alert('INFO', str);
    }
    document.addEventListener("deviceready", onDeviceReady,
      false);
  }
});
```

2. Include the `ch08_01.js` file in the `index.html` file.
3. Deploy and access it from the device of your choice.

How it works...

`navigator.connection.type` gets the connection type from the `navigator` JavaScript object and is used to compare the network status values defined in Cordova's `Connection` object to determine whether the device is online or offline. Additionally, the `Connection` object gives more information about the kind of network available in the online case. `Connection.NONE` indicates the offline mode of the device. `navigator.onLine` allows us to check whether the browser is in online or offline mode. This is different from the device's online or offline mode. Many browsers would say they are online even if there is no network.

At the time of writing this chapter, the `Connection` API supports the following platforms:

- iOS
- Android
- BlackBerry WebWorks (OS 5.0 and higher)

There's more...

The preceding code works well as long as we are using compatible browsers (that support the property on the `navigator` object) and the Cordova APIs. However, using Cordova is not mandatory for creating Sencha Touch-based applications. There is one more technique that we can use to identify the offline mode. Let us see how to use that technique. Also, Sencha's `Device` APIs have added a similar API to check the connection status and related information. We will see what those APIs are and how to use them.

Using aggressive timeout

In *Chapter 5*, *Dealing with Data and Data Sources*, we saw how to use the stores and the proxies to connect to the data sources and load the data. Proxy is configured on a model or a store. To figure out whether we are in online or offline mode, we can use the `timeout` property on the proxy and set a very short timeout. If the connection fails, the `exception` handler will take care of using the offline data for the application, as shown in the following code snippet:

```
proxy: {
  type: 'ajax',
  url : 'orders.json',
  reader: {
    type: 'json',
    rootProperty: 'orders',
    totalProperty: 'totalRecords',
    successProperty: 'success'
  },
  timeout: 2000,
  listeners: {
    exception:function (proxy, response, operation) {
      //we are offline
    }
  }
}
```

Using Sencha's Device API

Every mobile device has various features and data worth using in an application; for example, accessing the device contact list, capturing photos and videos using the device camera, notification, and so on. Since, Sencha is a JavaScript framework, direct access to the platform (Android/iOS/WebOS) APIs is not available. The framework implements different approaches to integrate with device features based on whether you are packaging for Cordova native or simulator. Sencha's Device APIs offer the `Ext.device.Connection` class, which can be used to detect the connectivity and find out the type of connection. The following code shows how to use this class:

```
Ext.require('Ext.device.Connection');
Ext.application({
  name : 'MyApp',
  launch : function() {
    var isOnline = Ext.device.Connection.isOnline();
    var type = Ext.device.Connection.getType();
    alert('Connection status is: ' +
      (isOnline?'ONLINE':'OFFLINE') + ' and type is: ' + type);
  }
});
```

In the preceding code, we had to add `Ext.require` so that the `Connection` class is loaded as it is not packaged with the `sencha-touch-all[-debug].js` files. The static method, `isOnline`, tells whether the device is online or offline by returning `true` or `false`, respectively. `getType` returns a string describing the kind of connection that the device has, for example, Wi-Fi, and ethernet.

If we do a native packaging of the application using the **Sencha Cmd** tool, the `Connection` class also fires an event, named `onlinechange`, whenever the connection status of the device changes. This could be very handy when we want to monitor the connection status change and execute some application logic. The following code snippet shows how to register the handler for the `onlinechange` event:

```
Ext.device.Connection.on('onlinechange', function(isOnline, type,
  eOpts) {
  alert('Connection status is: ' + (isOnline?'ONLINE':'OFFLINE')
     + ' and type is: ' + type);
});
```

The current status of the connection and type information is passed to the handler.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Loading data through AJAX using the Ajax proxy* recipe in *Chapter 5*, *Dealing with Data and Data Sources*

Storing your data offline in localstorage

Any application has to deal with data to provide a rich set of functionalities. Moreover, when the application goes offline, the data that's required to work with also needs to be available locally. In this recipe we will look at how to take our data offline and use it in the application.

We have taken the example of an application that will download the list of orders and their details on the device and uses it to allow the user to look at the list of orders and their details.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch08` folder created inside the `www` folder.

How to do it...

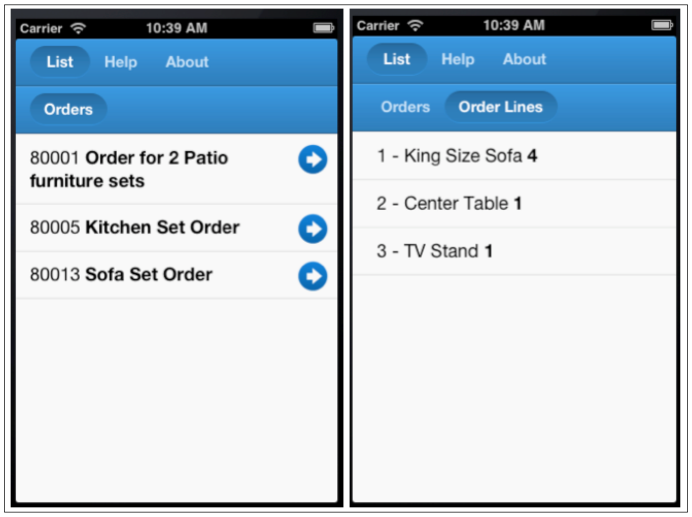
Carry out the following steps:

1. Create and open a new file, `ch08_02.js`, and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {
    var orderList, onlineStore, offlineStore;
    Ext.define('OrderLine', {
      extend: 'Ext.data.Model',
      config: {
        fields: ['id', 'product', 'description',
          'orderedQty', 'uom', 'price', 'lineNo']
      }
    });
    Ext.define('Order', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          'id',
          {name: 'orderNbr',  type: 'int', mapping:
            'documentNo'},
          {name: 'description',   type: 'string'},
          {name: 'dateOrdered', type: 'string'},
          {name: 'customer', type: 'string'},
          {name: 'customerLocation', type: 'string'},
          {name: 'isNewOrder', type: 'boolean',
            defaultValue: true}
        ],
        //association with order lines
        hasMany: {model: 'OrderLine', name: 'orderlines',
          associationKey:'orderlines'},
        proxy: {
          type: 'ajax',
          url : 'ch08/orders.json',
          reader: {
            type: 'json',
            rootProperty: 'orders',
            totalProperty: 'totalRecords',
            successProperty: 'success'
          },
          timeout: 2000,
          listeners: {
            exception:function (proxy, response, operation)
              {
              //fall back to the offline store
              orderList.setStore(offlineStore);
              offlineStore.load();
            }
          }
        }
      }
    });

    onlineStore = Ext.create('Ext.data.Store', {
      model: 'Order'
    });
    offlineStore = Ext.create('Ext.data.Store', {
      model: 'Order',
      autoSync: true,
      proxy: {
        type: 'localstorage',
        id: 'yapps-01'     //unique id
      }
    });
    //populate the offline store with the data read from
      the online store
    onlineStore.addListener('load', function (store,
      records, successful) {
      if (successful) {
        //clear old records
        offlineStore.getProxy().clear();
        //since id is already populated on the
        //records, mark the records dirty otherwise
        //they will not be saved
        for(var i=0; i<records.length; i++)
        records[i].setDirty();
        offlineStore.add(records);
        //save records in localstorage
        offlineStore.sync();
        orderList.setStore(offlineStore);
      }
    });
    orderList = Ext.create('Ext.List', {
      title: 'Orders',
      itemTpl: '<tpl for="."><div>{orderNbr}
        <b>{description}</b></div></tpl>',

      //show order lines when disclose icon is tapped on an
        order entry
      onItemDisclosure: function(){
        var orderTabPnl = Ext.getCmp('ordertab-pnl-id');
        var orderLinesPnl = Ext.getCmp('orderlines-pnl-
          id');
        //we might have already shown the order line tab.
          if so, destroy it
        if (!Ext.isEmpty(orderLinesPnl))
        orderTabPnl.remove(orderLinesPnl);
        //get order lines from the order object, which we
          had loaded from json file
        var ols = [];
        arguments[0].orderlines().each(function(ol) {
          ols.push(ol);
        });
        orderLinesPnl = Ext.create('Ext.List', {
          id: 'orderlines-pnl-id',
          title: 'Order Lines',
          itemTpl: '<tpl for="."><div style="padding-
            left:10px;">{lineNo} - {product}
              <b>{orderedQty}</b></div></tpl>',
          store: Ext.create('Ext.data.Store', {
            model: 'OrderLine',
            data : ols
          })
        });
        //show the Order Lines tab to the user
        orderTabPnl.insert(1, orderLinesPnl);
```

```
          orderTabPnl.setActiveItem(1);
        },
        store: onlineStore
      });
      //Order tab panel with Order and Order Lines tabs
      var orderTab = Ext.create('Ext.TabPanel', {
        id: 'ordertab-pnl-id',
        title: 'List',
        ui        : 'light',
        items: [orderList]
      });

      //Main tab panel
      Ext.create('Ext.TabPanel', {
        id: 'tab-pnl-id',
        fullscreen: true,
        ui        : 'light',
        sortable  : true,
        items: [orderTab,
          {
            title: 'Help',
              html: '<h1 style="font-size:16px;"><b>Help</b>
                </h1><p>This application shows the orders and
                  their line items.</p>',
              styleHtmlContent: true
          },
          {
            title: 'About',
            html : '<h1 style="font-size:16px;"><b>About this
              app!</b></h1><p>Version 0.1</p>',
            styleHtmlContent: true
          }
        ]
      });

      onlineStore.load();
    }
  });
```

2. Include `ch08_02.js` in place of `ch08_01.js` in the `index.html` file.
3. Deploy and access it from the device of your choice.



80001 **Order for 2 Patio furniture sets**
80005 **Kitchen Set Order**
80013 **Sofa Set Order**

1 - King Size Sofa **4**
2 - Center Table **1**
3 - TV Stand **1**

4. Rename `orders.json` to something else so that the proxy throws an error as it cannot find the file on the specified URL.
5. Reload the application. Though the proxy does not find the `JSON` file, it falls back to `offlineStore` and we still see **Orders** populated. Since, we have not stored **Order Lines** in `localStorage`, the **Order Lines** tab will not be loaded with the data.

How it works...

In the preceding code, we defined two models, `Order` and `OrderLine`; the association between them is one-to-many, which is indicated by `hasMany`. Then we created two stores: `onlineStore` and `offlineStore`. `onlineStore` is of type `ajax` and loads the order data from the `orders.json` file. `offlineStore` is bound to the HTML5 `localStorage` instance.

`onlineStore` is bound to `orderList` and we registered a handler for the `load` event on `onlineStore`. The handler function saves all the orders in the local storage and binds `orderList` to `offlineStore`. So, we first download all the orders from the remote system, save them locally, and work with the local data.

To switch to the offline mode, we have used the timeout technique and the exception handler binds `orderList` with `offlineStore` and then loads the data from there.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Loading data through AJAX using the Ajax proxy* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Creating a model* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Relating models using association* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Managing a list of data using List* recipe in *Chapter 6*, *Adding Components*
- The *Working with Tab panels* recipe in *Chapter 6*, *Adding Components*

Storing your data offline using Sencha.io

The previous recipe is useful for any hosted application. Let us say you are using Sencha's cloud service, **Sencha.io**, to store your data and you want to have the data available for offline usage as well. For this purpose, Sencha.io comes with a `syncstorage` proxy that helps us manage the data on the cloud and also offers the offline capability for which it saves the data in `localStorage`. In this recipe, we will see how to set up our application to use Sencha.io APIs and use the `syncstorage` proxy to manage the application data.

> At the time of writing this book, Sencha.io was in beta and the final APIs may differ from what have been used in this chapter.

Getting ready

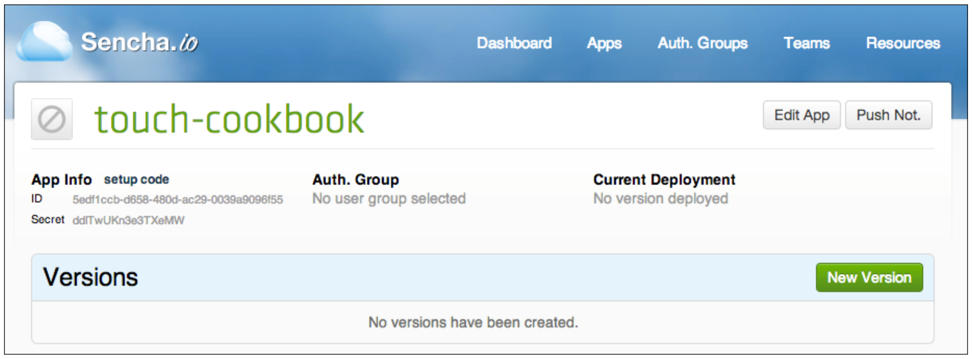
Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch08` folder created inside the `www` folder.

How to do it...

Carry out the following steps to store data offline using Sencho.io:

1. Download the Sencha.io SDK from `http://www.sencha.com/products/io/`.
2. Extract the ZIP files and keep the SDK files under `www/sencha-io-<version>` folder; for example, `www/sencha-io-0.7.15`.
3. Open `https://manage.sencha.io/#!/dashboard` in your browser. Register yourself or log in to the Dashboard.
4. Create a new application and take a note of the values populating **ID** and **Secret**, as shown in the following screenshot:



touch-cookbook

App Info setup code
ID 5edf1ccb-d658-480d-ac29-0039a9096f55
Secret ddlTwUKn3e3TXeMW

Auth. Group
No user group selected

Current Deployment
No version deployed

Versions

No versions have been created.

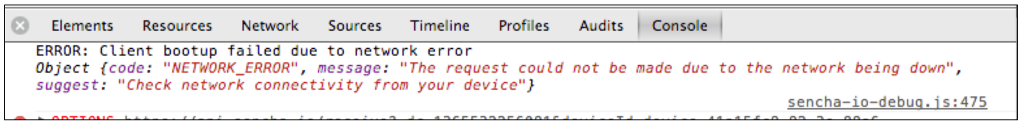
5. Create and open a new file, `ch08_03.js`, and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  io: {
    appId:'5edf1ccb-d658-480d-ac29-0039a9096f55',
    appSecret: 'ddlTwUKn3e3TXeMW'
  },
  launch: function() {
    var orderList, syncStore;
    Ext.define('Order', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          'id',
          {name: 'orderNbr',  type: 'int', mapping:
            'documentNo'},
          {name: 'description',   type: 'string'},
          {name: 'dateOrdered', type: 'string'},
          {name: 'customer', type: 'string'},
          {name: 'customerLocation', type: 'string'},
          {name: 'isNewOrder', type: 'boolean',
            defaultValue: true}
        ]
      }
    });

    syncStore = Ext.create('Ext.data.Store', {
      model: 'Order',
      autoLoad: true,
      autoSync: false, //manual sync
      proxy: {
        type: 'syncstorage',
        owner: 'user',
        access: 'private',
        id: 'ioorders'
     }
    });

    orderList = Ext.create('Ext.List', {
      title: 'Orders',
      itemTpl: '<tpl for="."><div>{orderNbr}
        <b>{description}</b></div></tpl>',
      store: syncStore
    });

    Ext.create('Ext.TabPanel', {
      id: 'ordertab-pnl-id',
      fullscreen: true,
      title: 'List',
      ui        : 'light',
      items: [orderList, {
        docked: 'bottom',
        xtype: 'titlebar',
        items: [{
          text: 'Load Data',
          align:'right',
          handler: function() {
            var rec = Ext.create('Order', {
              orderNbr: 1,
              documentNo: '80001',
              description: 'Order for 2 Patio furniture
                sets',
              customer: 'Ajit Kumar',
              customerLocation: ''
            });
          syncStore.add(rec);
          syncStore.sync();
          }
        }]
      }]
    });
  }
});
```

6. Include `ch08_03.js` in place of `ch08_02.js` in the `index.html` file.
7. Include the following script in `index.html`, before `ch08_03.js`, to add the `Sencha.io` framework file:

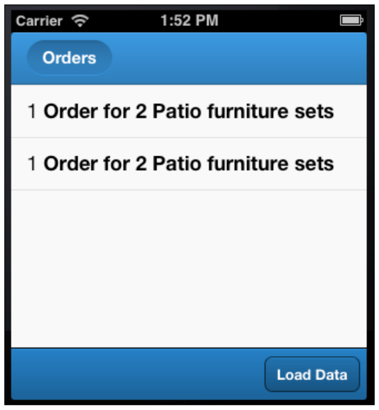
   ```
   <script type="text/javascript" charset="utf-8" src="sencha-
     io-0.7.15/sencha-io-debug.js"></script>
   ```

8. Deploy and access it from the browser/simulator.
9. Click on the **Load Data** button to load the record in the list.
10. Disconnect your computer where the app is hosted and reload the application. This time you will see the following error in developer tool, which shows the Sencha.io API is not able to contact the cloud:



```
ERROR: Client bootup failed due to network error
Object {code: "NETWORK_ERROR", message: "The request could not be made due to the network being down",
suggest: "Check network connectivity from your device"}
                                                                    sencha-io-debug.js:475
```

You will still see the records loaded in the list, as shown in the following screenshot:


Carrier
1:52 PM
Orders
1 Order for 2 Patio furniture sets
1 Order for 2 Patio furniture sets
Load Data

How it works...

In the recipe, we first registered with **Sencha.io** and created an application to get `appID` and `appSecret`. Using `appID` and `appSecret`, we configured `io` on the application, which will be used to validate the access from our application.

In the preceding code, we defined a model, `Order`. Then we created a store named `syncStore`. `syncStore` uses the `syncstorage` type proxy offered by Sencha.io.

`syncStore` is bound to `orderList`, which has got a bottom toolbar with the **Load Data** button. The `handler` function creates a new `Order` model, adds it to `syncStore`, and calls `sync` on the store to save the data.

To switch to offline mode, we did not have to use the timeout technique and the exception handler as the `syncstorage` proxy takes care of this switching automatically for us.

See also

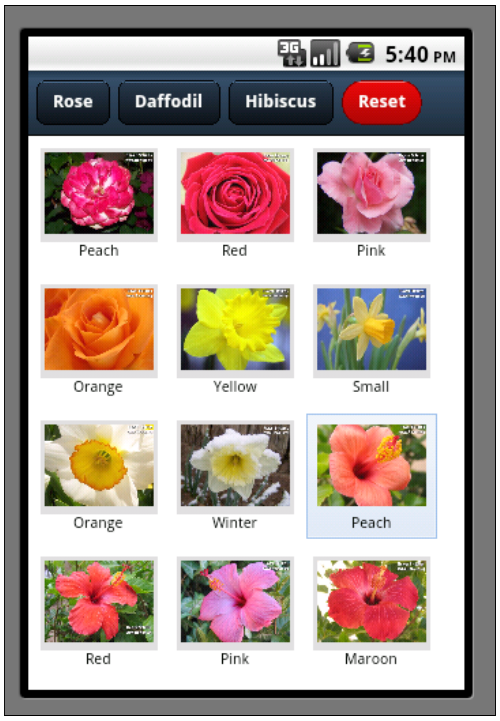
- The *Setting up the Android-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1, Gear Up for the Journey*

- The *Loading data through AJAX using the Ajax proxy* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Creating a model* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Managing a list of data using List* recipe in *Chapter 6*, *Adding Components*
- The *Working with Tab panels* recipe in *Chapter 6*, *Adding Components*

Storing your images offline

In the previous recipe, we talked about storing data offline. Images are used extensively in applications and enhance the overall presentation. Typically, an image is accessed as a URL. These URLs will not be accessible when the device or the browser goes offline. To some extent, this can be managed by using the image-caching feature of the browser and by giving it a long period of validity before it expires. But this may not be honored all the time by the browsers. We need a better mechanism to contain the images that is under the complete control of our application. In this recipe, we will see what it takes to persist the images locally and use them in the application.

For the demonstration, we will enhance the application that we built in *Chapter 4*, *Building Custom Views*, where we used images from a third-party website to show the album of flowers shown in the following screenshot:



Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch08` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. Create and open a new file, `ch08_04.js`, and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {
    Ext.define('Flower', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          'id','album','url','title', 'about'
        ],
        proxy: {
          type: 'ajax',
          url : 'ch08/flowers.json',
          reader: {
            type: 'json',
            rootProperty: 'flowers',
            totalProperty: 'totalRecords',
            successProperty: 'success'
          },
          timeout: 2000,
          listeners: {
            exception:function (proxy, response, operation)
              {
              var dv = Ext.getCmp('dataview-id');
                dv.setStore(offlineStore);
                offlineStore.load();
          }
        }
      }
    },
    setUrl: function() {
      var script = document.createElement("script");
      script.setAttribute("src",
      "http://src.sencha.io/data.setPhotoUrl-" +
         this.getId() +"/" + this.get('url')
    );
    script.setAttribute("type","text/javascript");
    document.body.appendChild(script);
  }
});

setPhotoUrl = function (id, dataUrl) {
  var flower = this.offlineStore.getById(id);
  flower.set('url', dataUrl);
  offlineStore.sync();
};

onlineStore = Ext.create('Ext.data.Store', {
    model: 'Flower'
});

  onlineStore.addListener('load', function (store, records,
    successful) {
    if (successful) {
      offlineStore.getProxy().clear();
      this.each(function (record) {
        var flower = offlineStore.add(record.data)[0];
        flower.setUrl();
      });
      offlineStore.sync();
      var dv = Ext.getCmp('dataview-id');
      dv.setStore(offlineStore);
    }
  });

  offlineStore = Ext.create('Ext.data.Store', {
    model: 'Flower',
    proxy: {
      type: 'localstorage',
      id: 'yapps-02'
    }
  });
```

```
var tpl = new Ext.XTemplate(
  '<tpl for=".">',
  '<div class="thumb-wrap" id="{title}">',
  '<div class="thumb"><img src="{url}"
    title="{title}"></div>',
  '<span>{about}</span></div>',
  '</tpl>',
  '<div class="x-clear"></div>'
);

var filter = function(criteria) {
  var dv = Ext.getCmp('dataview-id');
  var store = dv.getStore();
  store.clearFilter();
  return store.filterBy(function(record, id){
    if (record.get('album') === criteria ||
      Ext.isEmpty(criteria))
      return true;
    else
      turn false;
  });
}

var pnl = Ext.create('Ext.Panel', {
  id:'images-view',
  fullscreen: true,
  scroll: false,
  monitorOrientation: true,
  layout: 'card',
  defaults: {
    border: false
  },
  items: [Ext.create('Ext.DataView', {
    id: 'dataview-id',
    store: onlineStore,
    scroll: 'vertical',
    itemTpl: tpl,
    autoHeight:true,
    singleSelect: true,
    overItemCls:'x-view-over',
    itemSelector:'div.thumb-wrap',
    emptyText: 'No images to display',
    monitorOrientation: true,
    listeners: {
          selectionchange: function(model, recs) {
            if (recs.length > 0) {
              Ext.getCmp('detail-panel').setHtml('<img src="'
                + recs[0].data.url + '" title="' +
                  recs[0].data.title + '">');
              Ext.getCmp('images-view').setActiveItem(1);
              Ext.getCmp('back-button').show();
              Ext.getCmp('rose-button').hide();
              Ext.getCmp('daffodil-button').hide();
              Ext.getCmp('hibiscus-button').hide();
            }
          },
          orientationchange: function(pnl, orientation,
            width, height){
            pnl.refresh();
          }
        }
      }),
      Ext.create('Ext.Panel', {
        id: 'detail-panel',
        styleHtmlContent: true,
        scroll: 'vertical',
        cls: 'htmlcontent'
      }),
      {
        xtype: 'toolbar',
        docked: 'top',
        items: [
          {
            text: 'Rose',
            id: 'rose-button',
            handler: function() {
              filter('rose');
            }
          },
          {
            text: 'Daffodil',
            id: 'daffodil-button',
            handler: function() {
              filter('daffodil');
            }
          },{
            text: 'Hibiscus',
            id: 'hibiscus-button',
          handler: function() {
            filter('hibiscus');
          }
        },{
          text: 'Reset',
          id: 'reset-button',
          ui: 'decline-round',
          handler: function() {
            Ext.getCmp('images-view').setActiveItem(0);
            filter('');
          }
        }, {
          text: 'Back',
          id: 'back-button',
          ui: 'back',
          hidden: true,
          handler: function() {
            Ext.getCmp('images-view').setActiveItem(0);
            this.hide();
            Ext.getCmp('rose-button').show();
            Ext.getCmp('daffodil-button').show();
            Ext.getCmp('hibiscus-button').show();
          }
        }
      ]
    }]
  });

    onlineStore.load();
  }
});
```

2. Create and open a new file, `flowers.json`, in the `ch08` folder and paste the following code into it:

```
{
  "totalRecords": "20",
  "success": "true",
  "flowers": [
    {
      "id": "1",
      "album": "rose",
      "url": "http://images.flowers.vg/
        250x300/rdroses01.jpg",
      "title": "Rose1",
      "about": "Peach"
    },
    {
      "id": "2",
      "album": "rose",
      "url": "http://images.flowers.vg/250x300/roses-
        maroon3.jpg",
      "title": "Rose2",
      "about": "Red"
    },
    {
      "id": "3",
      "album": "rose",
      "url": "http://images.flowers.vg/250x300/roses-dark-
        pink.jpg",
      "title": "Rose3",
      "about": "Pink"
    },
    {
      "id": "4",
      "album": "rose",
      "url": "http://images.flowers.vg/250x300/roses-
        bright-orange.jpg",
      "title": "Rose4",
      "about": "Orange"
    },
    {
      "id": "5",
      "album": "daffodil",
      "url": "http://images.flowers.vg/250x300/
        daffodil.jpg",
      "title": "Daffodil1",
      "about": "Yellow"
    },
    {
      "id": "6",
      "album": "daffodil",
      "url": "http://images.flowers.vg/250x300/daffodil-
        yellow.jpg",
      "title": "Daffodil2",
      "about": "Small"
    },
    {
```

```
    "id": "7",
    "album": "daffodil",
    "url": "http://images.flowers.vg/250x300/daffodil-
      white-orange.jpg",
    "title": "Daffodil2",
    "about": "Orange"
  },
  {
    "id": "8",
    "album": "daffodil",
    "url": "http://images.flowers.vg/250x300/
      winter_flowers_daffodil_white.jpg",
    "title": "Daffodil2",
    "about": "Winter"
  },
  {
    "id": "9",
    "album": "hibiscus",
    "url": "http://images.flowers.vg/250x300/hibiscus-
      peach.jpg",
    "title": "Hibiscus1",
    "about": "Peach"
  },
  {
    "id": "10",
    "album": "hibiscus",
    "url": "http://images.flowers.vg/250x300/
      hibiscusred.jpg",
    "title": "Hibiscus1",
    "about": "Red"
  },
  {
    "id": "11",
    "album": "hibiscus",
    "url": "http://images.flowers.vg/250x300/hibiscus-
      pink-pink.jpg",
    "title": "Hibiscus1",
    "about": "Pink"
  },
  {
    "id": "12",
    "album": "hibiscus",
    "url": "http://images.flowers.vg/250x300/hibiscus-
      red-maroon.jpg",
        "title": "Hibiscus1",
        "about": "Maroon"
      },
      {
        "id": "13",
        "album": "hibiscus",
        "url": "http://images.flowers.vg/250x300/hibiscus-
          pink-pink.jpg",
        "title": "Hibiscus1",
        "about": "Pink"
      },
      {
        "id": "14",
        "album": "hibiscus",
        "url": "http://images.flowers.vg/250x300/hibiscus-
          red-bright.jpg",
        "title": "Hibiscus1",
        "about": "BrightRed"
      }
    ]
  }
```

3. Create and open a new file, `ch08.css`, inside the `ch08` folder and paste the following code inside it:

```
#images-view .x-panel-body{
  background: white;
  font: 11px Arial, Helvetica, sans-serif;
}
#images-view .thumb{
  background: #dddddd;
  padding: 3px;
}
#images-view .thumb img{
  height: 60px;
  width: 80px;
}
#images-view .thumb-wrap{
  float: left;
  margin: 4px;
  margin-right: 0;
  padding: 5px;
}
```

```
#images-view .thumb-wrap span{
  display: block;
  overflow: hidden;
  text-align: center;
}

#images-view .x-view-over{
  border:1px solid #dddddd;
  background: #efefef url(images/row-over.gif) repeat-x
    left top;
  padding: 4px;
}
#images-view .x-item-selected{
  background: #eff5fb url(images/selected.gif) no-repeat
    right bottom;
  border:1px solid #99bbe8;
  padding: 4px;
}
#images-view .x-item-selected .thumb{
  background:transparent;
}
```

4. Include `ch08_04.js` in place of `ch08_03.js` in the `index.html` file.
5. Include `ch08.css` in the `index.html` file.
6. Deploy and access it from the device of your choice.

How it works...

In the preceding code, we defined a model, `Flower`. Then we created two stores, namely, `onlineStore` and `offlineStore`. `onlineStore` is of type `ajax` and loads the flower data from the `flowers.json` file. `offlineStore` is bound to the HTML5 `localStorage` instance.

`onlineStore` is bound to `DataView`, and we registered a handler for the `load` event on `onlineStore`. The handler function saves all the orders into the local storage and binds `DataView` to `offlineStore`. While adding a model to the local storage, we called the `setUrl` method on the model to set the Sencha.io cloud service to get the `dataUrl` method corresponding to an image URL. Another alternative to using Sencha.io is to have our own server-side implementation that can convert an image URL to a data URL. After the image is loaded, `id` and `dataUrl` are passed to the `setPhotoUrl` callback method. The callback method then sets the URL on a model to the `dataUrl` method received from the Sencha.io service and updates the model in the local storage. The `dataUrl` mechanism allows us to persist the image locally without worrying about browser caching, expiry time, and so on.

To switch to offline mode, we have used the timeout technique and the exception handler binds the `DataView` with `offlineStore` and then loads the data from there.

> You may learn more about Sencha.io at:
>
> - `http://www.sencha.com/products/io/`
> - `http://www.sencha.com/learn/how-to-use-src-sencha-io/`

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Loading data through AJAX using the Ajax proxy* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Creating model* recipe in *Chapter 5*, *Dealing with Data and Data Sources*
- The *Designing custom views using DataView* recipe in *Chapter 4*, *Building Custom Views*
- The *Using XTemplate for advanced templating* recipe in *Chapter 4*, *Building Custom Views*
- The *Storing your data offline in localstorage* recipe
- The *Storing your data offline using Sencha.io* recipe

Application caching

So far, we have seen how to detect the online or offline modes and store data and images locally. The last thing to do is to cache the application code so that they are downloaded locally and are available for offline use. In this recipe we will go through the steps to achieve it.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch08` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. Create and open a new file, `touch.manifest`, inside the `ch08` folder, and paste the following code into it:

   ```
   CACHE MANIFEST

   #version 0.1

   ../index.html

   ../touch/resources/css/sencha-touch.css
   ../touch/sencha-touch-all-debug.js
   ch08.css
   ch08_04.js
   ```

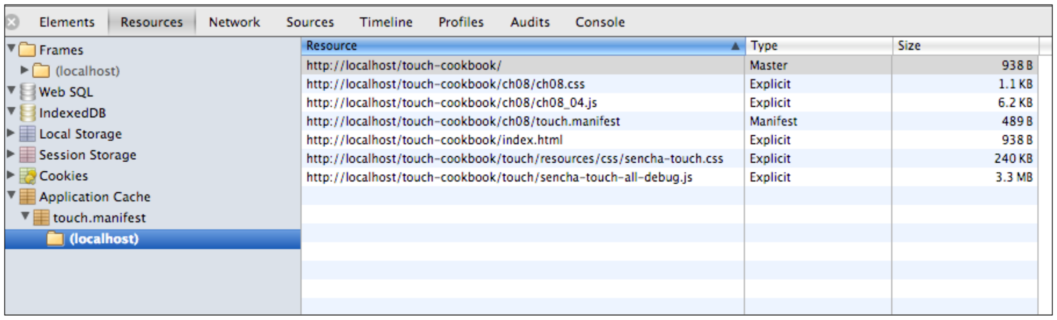
2. Modify the `index.html` file to include the manifest file:

   ```
   <html manifest="ch08/touch.manifest">
   ```

3. Add the following to the `mime.types` file of the Apache web server:

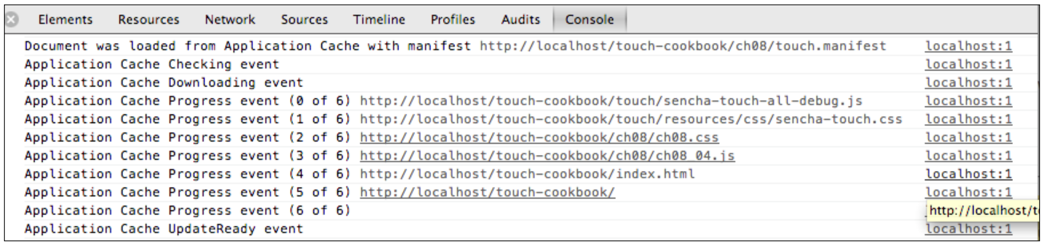
   ```
   text/cache-manifest manifest
   ```

4. Deploy and access it from the device of your choice. If you access the file from a browser, you will see an entry created in **Application Cache**, as shown in the following screenshot:



Resource	Type	Size
http://localhost/touch-cookbook/	Master	938 B
http://localhost/touch-cookbook/ch08/ch08.css	Explicit	1.1 KB
http://localhost/touch-cookbook/ch08/ch08_04.js	Explicit	6.2 KB
http://localhost/touch-cookbook/ch08/touch.manifest	Manifest	489 B
http://localhost/touch-cookbook/index.html	Explicit	938 B
http://localhost/touch-cookbook/touch/resources/css/sencha-touch.css	Explicit	240 KB
http://localhost/touch-cookbook/touch/sencha-touch-all-debug.js	Explicit	3.3 MB

5. Reload the application. If you are accessing the user file from the browser, you will see the following messages showing that the files are being loaded from the cache:



```
Document was loaded from Application Cache with manifest http://localhost/touch-cookbook/ch08/touch.manifest   localhost:1
Application Cache Checking event                                                                               localhost:1
Application Cache Downloading event                                                                            localhost:1
Application Cache Progress event (0 of 6) http://localhost/touch-cookbook/touch/sencha-touch-all-debug.js      localhost:1
Application Cache Progress event (1 of 6) http://localhost/touch-cookbook/touch/resources/css/sencha-touch.css localhost:1
Application Cache Progress event (2 of 6) http://localhost/touch-cookbook/ch08/ch08.css                        localhost:1
Application Cache Progress event (3 of 6) http://localhost/touch-cookbook/ch08/ch08_04.js                      localhost:1
Application Cache Progress event (4 of 6) http://localhost/touch-cookbook/index.html                           localhost:1
Application Cache Progress event (5 of 6) http://localhost/touch-cookbook/                                     localhost:1
Application Cache Progress event (6 of 6)                                                                      http://localhost/t
Application Cache UpdateReady event                                                                            localhost:1
```

6. Disconnect the device from the network or stop the web server.
7. Reload the application. You will still see the complete UI rendered with the data loaded.

How it works...

The steps outlined use Cache Manifest to tell the browser to cache the resources listed in the `touch.manifest` file. To the manifest file, we added a `#version` line, which we update whenever we make any changes to the code. This is added to overcome the problem of resources (JavaScript files) not being reloaded when there are no changes in the manifest file. Afterwards, we added a new MIME type support to our Apache web server by extending the `mime.types` file. You may have to check the specifics related to your web server and configure the MIME type accordingly.

Once the manifest file was created and the support was added to the web server, we added the `manifest` attribute to `<html>` where we specified our manifest file. This way, the browser will load the manifest file and all the resources listed inside it.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up a browser-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

9
Increased Relevance Using Geolocation

In this chapter, we will cover:

- Finding out your location
- Finding out the location using native device APIs
- Auto update of your location
- Tracking direction and speed
- Hooking up Google Maps with your application
- Working with the Google Maps options
- Mapping Geolocation on Google Maps

Introduction

Imagine how good it would be to build an application that can automatically determine the user mobile location and provide local searches such as suggesting places of interest, hotels, the nearest police station, and so on. Imagine how the user would feel if, after determining the location, we display the relevant information and the routes on a map that give clear directions on how to reach a place. Another example could be providing an application that can tell my average speed while I am jogging or an application that can help track a fleet of trucks, provide a route map that is less congested, send an SOS message to a friend with your location detail, and so on.

All of this is feasible and possible with the newly introduced Geolocation specification from the W3 Consortium (`http://dev.w3.org/geo/api/spec-source.html`).

This specification provides us with the required objects, methods, and events to get the location detail and work with it.

In this chapter, we will look at the classes provided by the Sencha Touch framework to work with Geolocation. The classes implement the W3C Geolocation specification. Additionally, we will see how to work with Google Maps and complement it with Geolocation.

Sencha Touch wraps the Google Map JavaScript APIs, outlined in `http://code.google.com/apis/maps/documentation/javascript/`, into a convenient class that we will be making use of in this chapter.

Finding out your location

W3C's Geolocation specification is implemented by the `Ext.util.GeoLocation` class in Sencha Touch. In this recipe, we will look into the class and see how to learn about our device location.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder `ch09` in the `www` folder. We will be using this new folder to keep the code.

How to do it...

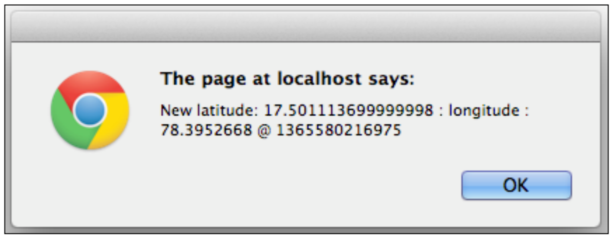
Carry out the following steps to find out your location:

1. Create and open a new file `ch09_01.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',
  launch: function() {
    var geo = Ext.create('Ext.util.GeoLocation', {
      autoUpdate: true,
      listeners: {
        locationupdate: function (geo) {
          alert('New latitude: ' + geo.getLatitude() + ' :
              longitude : ' + geo.getLongitude() + ' @ ' +
                geo.getTimestamp());
```

```
            },
            locationerror: function (   geo,
              bTimeout,
              bPermissionDenied,
              bLocationUnavailable,
            message) {
              if(bTimeout){
                alert('Timeout occurred.');
              }
              if (bPermissionDenied){
                alert('Permission denied.');
              }
              if (bLocationUnavailable) {
                alert('Location unavailable.');
              }
            }
          }
        });

        geo.updateLocation();

      }
    });
```

2. Include `ch09_01.js` in the `index.html` file.
3. Deploy and access it from the device of your choice. You will see a message showing the longitude and latitude of your location as shown in the following screenshot:



The page at localhost says:

New latitude: 17.501113699999998 : longitude : 78.3952668 @ 1365580216975

How it works...

In the preceding code, we created an instance of the `Ext.util.GeoLocation` class with `autoUpdate` set to `false`. This means that the browser will not watch for a change in location. The update is fired manually by calling the `updateLocation` method on the `geo` object. Additionally, on the `geo` object, listeners have been set up for the `locationupdate` and `locationerror` events. The `locationupdate` event is fired when the location is updated. The framework passes the object representing the location information at that instance of time. The `geo` object contains the following fields:

- `latitude`
- `longitude`
- `timestamp`
- `accuracy`
- `altitude`
- `altitudeAccuracy`
- `heading`
- `speed`

Out of the preceding listed properties, `latitude`, `longitude`, `timestamp`, and `accuracy` will be provided. However, other properties can be `null` based on the device on which we are using the API.

In case any errors occur while trying to get the updated location information, the framework fires the `locationerror` event where it indicates the following three types of errors:

- The operation timed-out
- User does not have permission
- Location information is not available

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1, Gear Up for the Journey*

Finding out the location using native device APIs

In some applications, manual updating of the location may not be desirable. For example, if your application is expected to update the location periodically to show the path in which a vehicle is moving. In this recipe, we will see how to configure the `Ext.util.GeoLocation` class to have the location automatically updated and how to control the frequency with which the location update should be attempted.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch09` folder created inside the `www` folder.

How to do it...

Carry out the following steps for finding out the location using native device APIs:

1. Create and open a new file `ch09_02.js` and paste the following code into it:

```
Ext.require('Ext.device.Geolocation');
Ext.application({
  name : 'MyApp',

  launch: function() {
    Ext.device.Geolocation.getCurrentPosition({
      success: function(position) {
        alert('Latitude: ' + position.coords.latitude + '
          Longitude: ' + position.coords.longitude);
      },
      failure: function() {
        console.log('Could not get the location!');
      }
    });
  }
});
```

2. Include `ch09_02.js` in place of `ch09_01.js` in the `index.html` file.
3. Deploy and access it from the device of your choice. You will see a pop up with the latitude and longitude information.

How it works...

Unlike the `Ext.util.Geolocation` class, `Ext.device.Geolocation` uses native device APIs to get the current location detail. What native device API is used is determined based on the device from which the application is accessed. If the application were packaged with Cordova, it would use Cordova APIs whereas if it was packaged using Sencha Cmd, which we will discuss in *Chapter 10*, *Device Integration*, it would work with WebView to get the location information. If the application is accessed from a simulator or a browser, the `Ext.device.Geolocation` class uses the `Ext.util.Geolocation` class.

The `position.coords` object contains the same information as the `geo` class.

There's more...

We saw in the earlier recipe how setting `autoUpdate` to `true` instructs the framework to monitor the location update and fire the `locationupdate` event. Let's see how we can achieve a similar functionality using the `Ext.device.Geolocation` class.

Watch for the location update

`Ext.device.Geolocation` offers the `watchPosition` method, which can be called to get regular location updates at the configured frequency. The following code snippet shows the usage of this method:

```
Ext.device.Geolocation.watchPosition({
  callback: function(position) {
  alert('New Latitude: ' + position.coords.latitude + '
    Longitude: ' + position.coords.longitude);
  },
  failure: function() {
    Ext.Msg.alert('Error', 'Could not get the location!');
  }
});
```

The update frequency is controlled using the `frequency` configuration, which defaults to `10` seconds. The following code snippet shows how you can specify the frequency:

```
Ext.device.Geolocation.watchPosition({
  frequency: 3000,
  callback: function(position) {..
```

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the Blackberry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Auto update of your location

In some applications, manual updating of location may not be desirable. For example, if your application is expected to update the location periodically to show the path in which a vehicle is moving. In this recipe, we will see how to configure the `Ext.util.GeoLocation` class to have the location automatically updated and how to control the frequency with which the location update should be attempted.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch09` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. Set the following properties on the `Ext.util.GeoLocation` class while instantiating:

   ```
   autoUpdate: true,
   timeout: 5000,
   ```

2. Deploy and access it from the device of your choice.

How it works...

Setting `autoUpdate` to `true` no longer requires the application code to call the `updateLocation` method explicitly. The location is updated automatically and the `locationupdate` or `locationerror` event is fired based on whether the update operation was successful or a failure. `True` is the default value for `autoUpdate`.

The `timeout` property allows us to control how frequently the location update will be attempted. It accepts the time in milliseconds. For example, in the preceding code snippet we set the value to `5000` milliseconds (5 seconds). This is a useful property if you want to save your mobile's battery as frequent updates will eat it up. The default value is set to `10` seconds.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the Blackberry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Finding out your location* recipe

Tracking direction and speed

The `Geolocation` object in Sencha Touch provides properties that we can use to figure out the direction and speed at which we are moving. This could be useful in applications where you may want to suggest to the user the nearest petrol pump based on his direction. In this recipe, we will look at the use of related properties.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch09` folder created inside the `www` folder.

How to do it...

Carry out the following steps:

1. In the `locationupdate` event handler, add the following line of code:

   ```
   alert('Heading:' + geo.getHeading() + ': Speed:' +
     geo.getSpeed());
   ```

2. Deploy and access it from the device of your choice.

How it works...

The preceding code uses the two important properties of the `Geolocation` class: `heading` and `speed`. The `heading` property gives the direction of travel of the device. It is specified in nonnegative degrees between 0 and 359. The angular degree is returned with respect to the real North. If the device is stationary, the value of this property is `undefined`.

The `speed` property gives the current ground speed of the device and the value will be in metres per second. If the device is stationary, the value of this property is `0`.

These two properties are optional and may not be available on every device. If these properties are not supported on a device, their value will be `null`. For example, in Android, the values returned are `null`. On such devices, we can derive these values using the `longitude`, `latitude`, and `timestamp` attributes.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Finding out your location* recipe

Hooking up Google Maps with your application

Google provides the Map service and also the APIs to integrate it into our application. Sencha Touch has wrapped it inside a component `Ext.Map`, which provides the complete map-related functionality. It, internally, uses the Google Maps' JavaScript APIs to provide us with a working map component. In this recipe, we will see how to make use of the `Map` class.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1, Gear Up for the Journey*.

Make sure that you have the `ch09` folder created inside the `www` folder.

How to do it...

Carry out the following steps to hook up Google maps with your application:

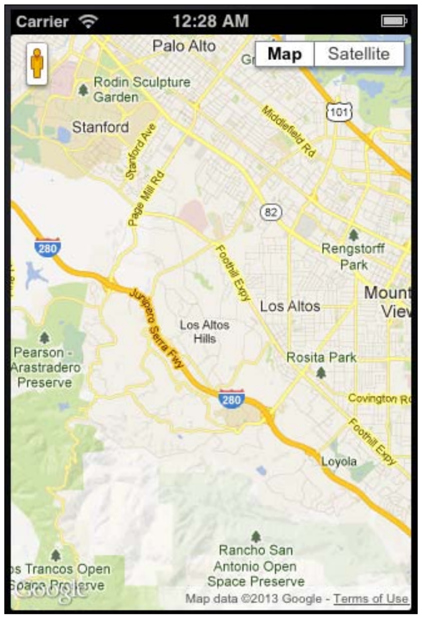
1. Create and open a new file `ch09_03.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',

  launch: function() {


    Ext.Viewport.add({
       xtype: 'map'
    });

  }
});
```

2. Include `ch09_03.js` in place of `ch09_02.js` in the `index.html` file.
3. Add the following to the `index.html` file to include the Google Maps' JavaScript APIs:

```
<script type="text/javascript" src="http://
  maps.google.com/maps/api/js?sensor=true"></script>
```

4. Deploy and access it from the device of your choice. You will see a screen showing Google Maps with its default longitude and latitude set to Palo Alto as shown in the following screenshot:



How it works...

In the preceding code, we created a panel and added a map component to it using the `xtype: 'map'` property. Usage of this `xtype` attribute leads to the instantiation of the `Ext.Map` class, which wraps Google Maps inside it. It initializes the Google Maps class with the following default map options:

- Map center is set to the location of Palo Alto (latitude – 37.381592, longitude – 122.135672)
- Map type is set to ROADMAP
- Zoom level is set to 12

There's more...

What if you want to set the center of the map to your current location? Let's see how we can do this.

Using the current location as the map center

To use the device's current location as the map center, you need to set `useCurrentLocation` configuration on the map instance to `true`.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Working with the Google Maps options

In the previous recipe, we looked at the default map options set by the `Ext.Map` class. In your application, say you are building an application to show the forest, mountains, and rivers around a particular place. In this case, you will have to set the map options according to your application needs. This recipe will show us how to achieve this.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch09` folder created inside the `www` folder.

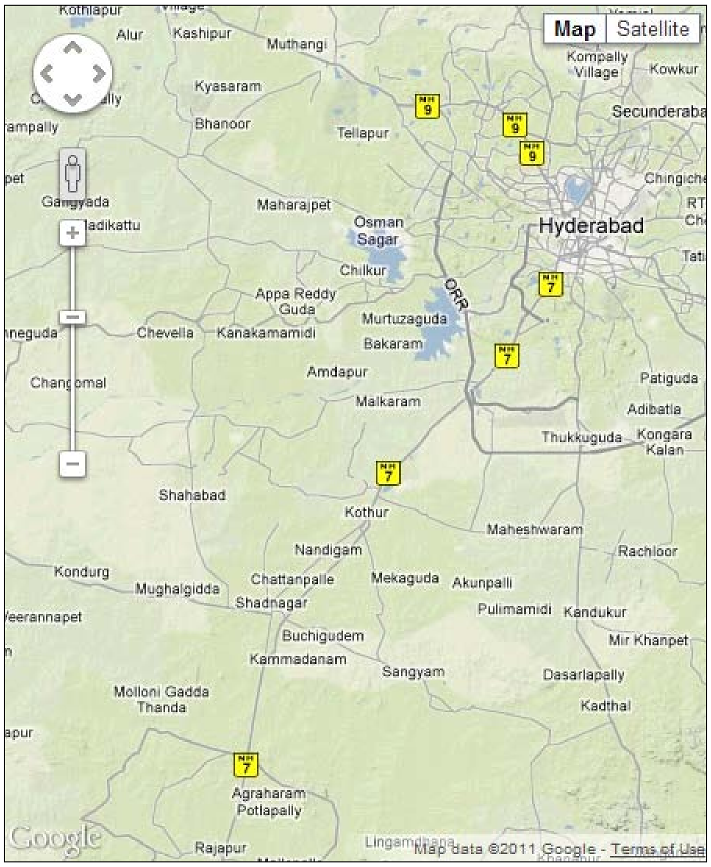
How to do it...

Carry out the following steps for working with the Google Maps options:

1. Edit the `ch09_03.js` file and add the `mapOptions` property as shown in the following code:

```
Ext.setup({
  onReady: function() {

    var pnl = new Ext.Panel({
      fullscreen: true,
      items     : [
        {
          xtype : 'map',
        mapOptions: {
          center: new google.maps.LatLng(17.22, 78.28),
          mapTypeId: google.maps.MapTypeId.TERRAIN,
          zoom: 10
        }
        }
      ]
    });
  }
});
```

2. Deploy and access it from the device of your choice. You will see a screen showing Google Maps with its default longitude and latitude set to `78.28` and `17.22`, respectively, as shown in the following screenshot:



How it works...

In the preceding code, we set the `mapOptions` property on the `Ext.Map` class, which accepts the `mapOptions` configuration that the Google Maps API can take. We specified three properties: `center`, `mapTypeId`, and `zoom`. To the `center` property, we set the latitude and longitude of a location that will be used to center the map. The longitude and latitude specified here are of Hyderabad, India. The `mapTypeId` property is set to `TERRAIN` so that in our application we can show the mountains, forest, and rivers around the center location. Using `zoom` we set the map zoom level to 10.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Hooking up Google Maps with your application* recipe

Mapping Geolocation on Google Maps

So far in this chapter, we have looked at the `Ext.util.GeoLocation` and `Ext.Map` classes of Sencha Touch to see how to get the location and how to display a map. In this recipe, we will put these two pieces together so that the location information read from the `Geolocation` class can be used on the `Map` class in rendering the information on the map. This can then be used, for example, based on the current location to highlight the nearest restaurants on the map.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Make sure that you have the `ch09` folder created inside the `www` folder.

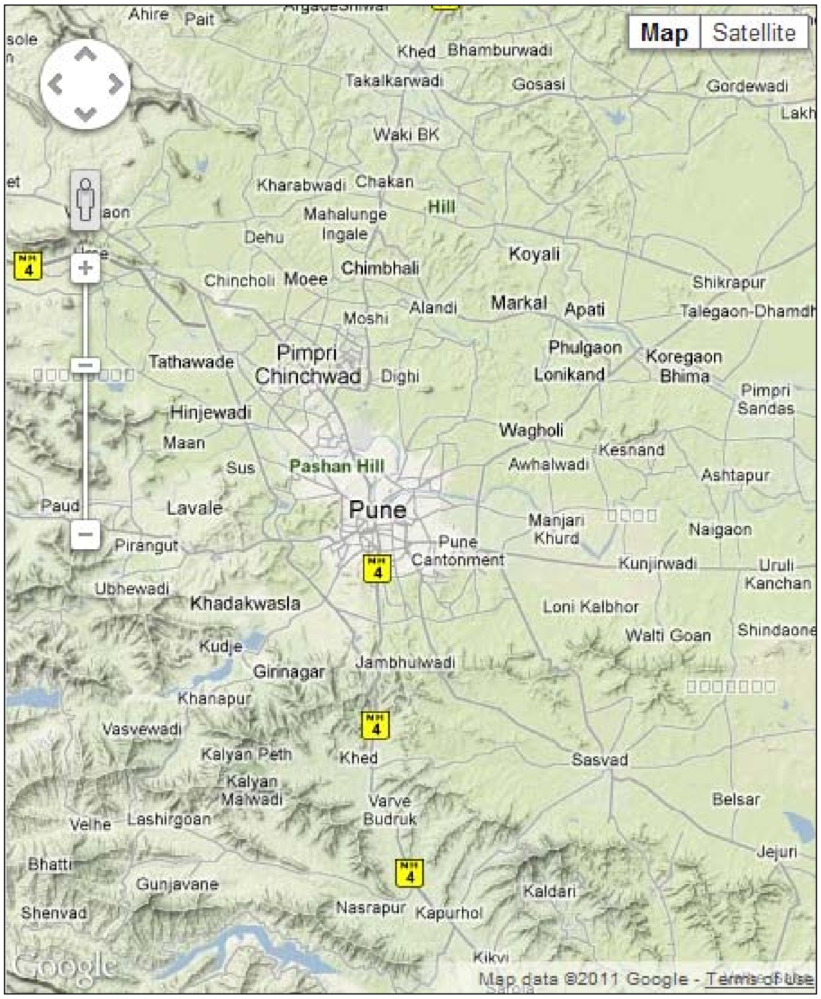
How to do it...

Carry out the following steps:

1. Create and open a new file `ch09_04.js` and paste the following code into it:

```
Ext.application({
  name : 'MyApp',

  launch: function() {

    var geo = Ext.create('Ext.util.GeoLocation', {
      autoUpdate: true,
      listeners: {
        locationupdate: function (geo) {
          var map = Ext.getCmp('google-map-id');
          map.setGeo(geo);
        },
        locationerror: function (   geo,
          bTimeout,
          bPermissionDenied,
          bLocationUnavailable,
          message) {
            if(bTimeout){
                alert('Timeout occurred.');
            }
            if (bPermissionDenied){
              alert('Permission denied.');
            }
            if (bLocationUnavailable) {
              alert('Location unavailable.');
            }
          }
      }
    });

    Ext.Viewport.add({
      xtype: 'map',
      id: 'google-map-id',
      geo: geo,
      mapOptions: {
        mapTypeId: google.maps.MapTypeId.TERRAIN,
        zoom: 10
      }
    });

  }
});
```

2. Include `ch09_04.js` in place of `ch09_03.js` in the `index.html` file.
3. Deploy and access it from the device of your choice. You will see a screen showing the Google map with the location set as per the longitude and latitude values returned by the Geolocation API as shown in the following screenshot:



How it works...

In the preceding code, we created the `Geolocation` instance with `autoUpdate` set to `true` and also a panel with a map. We have given an ID to the map component, `google-map-id`, which we use in the `locationupdate` event listener on the `GeoLocation` object. This then gets the map component and calls the `setGeo` method on it to update the map with the new location information. Though the complete `geo` object is passed to the `update` method, it only uses the `longitude` and `latitude` properties of it. This way, the location information is fetched from `GeoLocation` and is passed on to the `Map` attribute to get them working together.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Finding out your location* recipe
- The *Hooking up Google Maps with your application* recipe

10
Device Integration

In this chapter, we will cover:

- Capturing and managing photos using a camera
- Pulling out those contacts
- Working with orientation
- Managing notifications
- Reading a file
- Handling the home button on Android, iOS, and BlackBerry
- Handling the back button on Android and BlackBerry
- Handling the menu button on Android and BlackBerry
- Handling the search button on Android
- Navigating using the BlackBerry trackpad

Introduction

Sencha Touch offers neat widgets and functionalities to build enterprise applications compatible with different platforms. In spite of all that Sencha Touch offers, there is one important aspect where support is still not very well encapsulated inside the framework and we have to go out of the framework to support them in our application. And this aspect is integration with various device features. For example, using a camera, pulling out the photos from the filesystem and linking the application behavior with the home, back, and search buttons on the device.

In this chapter, we will look at the recipes related to the integration of Sencha Touch with different device features, either using the Sencha APIs or Cordova.

All the device APIs are packaged under `Ext.device`. You may refer to the API documentation to see which APIs are available and learn more about them.

Capturing and managing photos using a camera

This recipe is going to show how you can pull out the photos from your phone or capture a live photo and use it in your application.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1, Gear Up for the Journey*.

Create a new folder `ch10` inside the `www` folder. We will be using this new folder to keep the code.

How to do it...

Follow the ensuing steps:

1. Create and open a new file `ch10_01.js` and paste the following code in it:

```
Ext.application({
  name : 'MyApp',

   //this is a must as it is not part of the standard package
  requires: ['Ext.device.Camera'],

  launch: function() {
    //template to show the photos
    var tpl = new Ext.XTemplate(
            '<tpl for=".">',
               '<div class="thumb"><img src="{url}">
                </div>',
            '</tpl>'
       );

    var data = [];

    var store = Ext.create('Ext.data.Store', {
       fields: ['url'],
       data: data
    });
    var dv = Ext.create('Ext.DataView', {
          store: store,
          scrollable: 'vertical',
          itemTpl: tpl,
          singleSelect: true,
          overItemCls:'x-view-over',
          itemSelector:'div.thumb-wrap',
          emptyText: 'No images to display'
    });


    Ext.create('Ext.Panel', {
       id:'images-view',
        fullscreen: true,
        layout: 'fit',
        items: [dv, {
             xtype: 'toolbar',
             docked: 'bottom',
             items: [{
                text: 'Capture Photo',
                handler: function() {
                   Ext.device.Camera.capture({
                      success: function(image) {
                           Ext.Msg.alert('Url', image);
                           data.push({url: image});
                           store.applyData(data);
                       },
                       failure: function(msg) {
                              Ext.Msg.alert('Error', 'Failed to
                              capture photo: ' + msg);
                       },
                       quality: 75,
                       destination: 'file',
                       source: 'camera'
                   });
                }
             }, {
                text: 'Select Media',
                handler: function() {
                   Ext.device.Camera.capture({
                      success: function(image) {
                           Ext.Msg.alert('Url', image);
                           data.push({url: image});
                           store.applyData(data);
                       },
                              failure: function(msg) {
                                  Ext.Msg.alert('Error', 'Failed to fetch
                                                photo: ' + msg);
                              },
                              quality: 75,
                              destination: 'file',
                              source: 'library'
                          });
                      }
                   }]
              }]
          });

      }
    });
```

2. Include `ch10_01.js` in the `index.html` file.
3. Deploy and access it from the device of your choice. When you tap on **Capture Photo**, it shall open the camera and allow you to take a photo, whereas when you tap on the **Select Media** button, it shall prompt you to select a photo from the library.

How it works...

In the preceding code, we created a data view and added it to a panel. We added a toolbar to the panel with two buttons: **Capture Photo** and **Select Media**. When the user clicks on the buttons, the handler calls the `capture` method of the camera device API offered by the Sencha Touch framework. The `capture` method accepts a config where we have specified two callbacks—`success` and `failure`—that are called to check whether the API carried out the capture operation successfully or not. Other important configs are `source` and `destination`. The `source` config tells you what the source for the photo is. The **Capture Photo** handler passes `camera` as the source to indicate that the user wants to capture a live photo using the camera, whereas the **Select Media** handler passes `library` as the source to indicate that the user will select a photo from the phone's photo library. Another valid value is `album`; it indicates that the user is going to select a photo from a photo album on their phone.

The `destination` config indicates the format in which you would like to get the image transferred to your success callback. Set it to `file` to indicate that you want the file URL to be passed to the `success` callback. Another valid value is `data`, if you want to receive the Base64 encode image (as a data URL) in your `success` callback.

The API offers three different implementations:

- **Sencha** – If your application has been packaged using Sencha Cmd, which we will learn about in the next chapter, this implementation will be used. It achieves functionality by communicating with the WebView and sending different camera-related commands to it.
- **Cordova** – If your application has been packaged with Cordova, this implementation will use the underlying Cordova APIs to provide camera integration. The application's behavior is consistent with the one packaged for Sencha.
- **Simulator** – If you have packaged your application to run in a simulator, this implementation will return a fixed Sencha icon.

Currently, the implementation supports only photos. For videos, you will have to directly interface with Cordova's Camera API where you need to set the media type accordingly.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Pulling out those contacts

Every phone has its contacts list, and we would have to interface with the native device APIs to fetch the contact information. In this recipe, we will see how to integrate with the Cordova API as Sencha does not provide the Cordova version of their Contact API, and you would not be able to get the contacts working if you packaged your application using Cordova.

We will revisit Sencha's Contact API in the next chapter when we talk about packaging.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder `ch10` in the `www` folder. We will be using this new folder to keep the code.

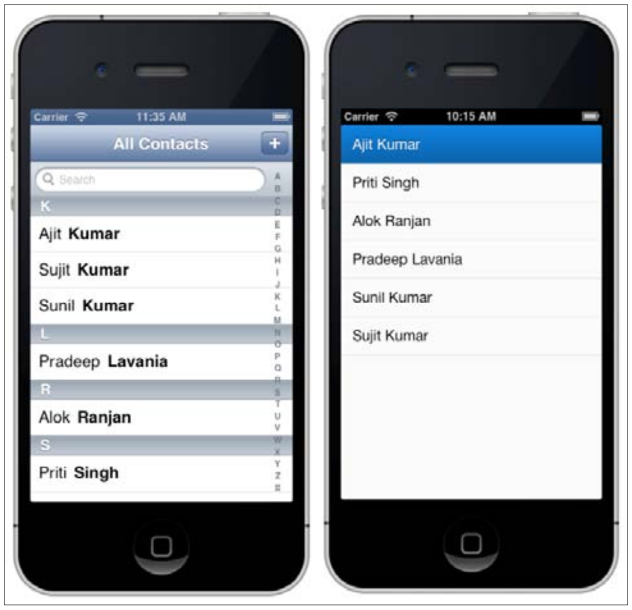
How to do it...

Follow the ensuing steps to integrate with the Cordova API:

1. Create and open a new file `ch10_02.js` and paste the following code in it:

```
Ext.application({
    name: 'MyApp',
    requires: 'Ext.device.Contacts',
    launch: function() {

      document.addEventListener("deviceready", this.onDeviceReady,
false);

    },
    onDeviceReady : function() {
        var options = new ContactFindOptions();
        options.filter="";
        options.multiple = true;

        var fields = ["displayName"];

        navigator.contacts.find(fields, function(contacts) {

          var data = [];
          for (var i=0; i<contacts.length; i++) {
             data.push({name: contacts[i].displayName});
         }

    Ext.Viewport.add({
        xtype: 'list',
        itemTpl: '{name}',
        store: {
            fields: ['name'],
            data: data
        }
        });

        }, function(msg) {
           alert('onError: ' + msg);
        }, options);
    }
});
```

2. Include `ch10_02.js` in place of `ch10_01.js` in the `index.html` file.
3. Deploy and access it from the device of your choice. You will see all the contact names as shown in the following screenshot:



How it works...

Cordova requires us to first watch for the `deviceready` event that will indicate that the Cordova framework has loaded. In the `deviceready` event handler, first we initialized the `ContactFilterOption` where we set a filter to `""` (an empty string) to indicate that no filtering needs to be applied on the contacts, and we set `multiple` to `true` to tell the API that we want to get all the contacts. If this property is not set, the API returns the first contact only. In the `fields` array, we listed only `displayName`. However, if you want you can list out any number of contact fields that you want the API to return. After that, we called the `contacts.find` API to get the contacts. For each contact, we pushed the name on the `data` array that we used to show the list of contacts in a data view.

You may refer to the `Contact` object detail on the Cordova Contact API documentation page to learn about the various fields that it can return.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Working with orientation

If you have looked at car game applications on mobile phones and liked the idea of using the phone as a way to steer your car, you will know that one of the uses of interfacing with the device's orientation API, using which you can get to know about the current orientation and also the device, is that it tells you about the new orientation as soon as it detects the change.

Sencha supports two implementations of orientation APIs, one using HTML 5 APIs as outlined in `http://dev.w3.org/geo/api/spec-source-orientation.html`, the specification of W3C, and the second where it works with the WebView.

In this recipe, we will see how to use the orientation APIs in our application. And since it has not been packaged using Sencha Cmd, it is going to use the HTML 5 version of the implementation.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder `ch10` in the `www` folder. We will be using this new folder to keep the code.

How to do it...

Follow the ensuing steps to use the orientation APIs in our application:

1. Create and open a new file `ch10_03.js` and paste the following code in it:

```
Ext.application({
    name: 'MyApp',
    requires: 'Ext.device.Orientation',

    launch: function() {
```

```
//defined a new class
Ext.define('MyOrientation', {
    extend: 'Ext.Container',
    config: {
        items: [
            {
                xtype: 'titlebar',
                docked: 'top',
                ui: 'light',
                items: [
                    {
                        text: 'Reset',
                        align: 'right'
                    }
                ]
            },
            {
                id: 'information',
                styleHtmlContent: true,
                html: 'no information received'
            },
            {
                id: 'cube',
                html: '<div></div><div></div><div><
                /div><div></div><div></div><div></div>',
                centered: true
            }
        ]
    },
    initialize: function() {
        this.on({
            painted: 'onPainted',
            erased : 'onErased'
        });
        this.on({
            delegate: 'button',
            tap: 'onReset'
        });
        if (!Ext.feature.has.Css3dTransforms) {
            Ext.getCmp('cube').hide();
        }
    },
    onReset: function() {
```

```
                this.originalOrientation = null;
            },
            onPainted: function() {
            Ext.device.Orientation.on('orientationchange',
'onDeviceOrientation', this);
            },
            onErased: function() {
            Ext.device.Orientation.un('orientationchange',
'onDeviceOrientation', this);
                this.originalOrientation = null;
            },
            onDeviceOrientation: function(e) {
                var alpha = Math.round(e.alpha),
                    beta = Math.round(e.beta),
                    gamma = Math.round(e.gamma),
                    origin;

                if (!this.originalOrientation) {
                    this.originalOrientation = {
                        alpha: alpha,
                        beta: beta,
                        gamma: gamma
                    };
                    return;
                }
                else {
                    origin = this.originalOrientation;
                    alpha -= origin.alpha;
                    beta -= origin.beta;
                    gamma -= origin.beta;
                }
                Ext.getCmp('information').setHtml([
                    'This example is best viewed when your phone is
on a flat surface.<br /><br />',
                    '<b>alpha</b>: ' + alpha,
                    '<br /><b>beta</b>: ' + beta,
                    '<br /><b>gamma</b>: ' + gamma
                ].join(''));
                if (!this.originalOrientation) {
                    this.originalOrientation = {
                        alpha: alpha,
                        beta: beta,
                        gamma: gamma
                    };
                    return;
                }
                else {
                    origin = this.originalOrientation;
                    alpha -= origin.alpha;
                    beta -= origin.beta;
                    gamma -= origin.beta;
                }
                if (Ext.feature.has.Css3dTransforms) {
                    Ext.getCmp('cube').element.dom.style.
webkitTransform = 'rotateX('+beta+'deg) rotateY('+alpha+'deg)';
                }
            }
        });
        Ext.Viewport.add(Ext.create('MyOrientation', {}));

    }
});
```

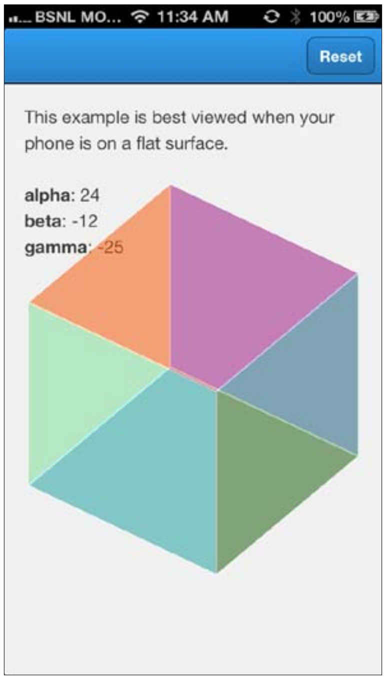
2. Add the following CSS to the `ch10.css` file:

```
#cube {
  -webkit-transform-style: preserve-3d;
  width: 200px;
  height: 200px;
  overflow: visible;
  opacity: .3
}

#cube .x-innerhtml {
  position: absolute;
  width: 100%;
  height: 100%;
  overflow: visible
}
```

```
#cube .x-innerhtml>div {
  width: 100%;
  height: 100%;
  background-color: #000;
  position: absolute;
  border: 1px solid #FFF
}
#cube .x-innerhtml>div:first-child {
  -webkit-transform: rotateX(90deg) translateZ(100px);
  background-color: red
}

#cube .x-innerhtml>div:nth-child(2) {
  -webkit-transform: translateZ(100px);
  background-color: green
}

#cube .x-innerhtml>div:nth-child(3) {
  -webkit-transform: rotateY(90deg) translateZ(100px);
  background-color: blue
}

#cube .x-innerhtml>div:nth-child(4) {
  -webkit-transform: rotateY(180deg) translateZ(100px);
  background-color: orange
}

#cube .x-innerhtml>div:nth-child(5) {
  -webkit-transform: rotateY(-90deg) translateZ(100px);
  background-color: cyan
}

#cube .x-innerhtml>div:nth-child(6) {
  -webkit-transform: rotateX(-90deg) rotate(180deg)
   translateZ(100px)
}
```

3. Include `ch10_03.js` in place of `ch10_02.js` and include `ch10.css` in the `index.html` file.

4. Deploy and access it from the device of your choice. You will see the orientation data and cube rotating based on the mobile's orientation, as shown in the following screenshot:



This example is best viewed when your phone is on a flat surface.

alpha: 24
beta: -12
gamma: -25

How it works...

In the preceding code, we created a container with two items: we created one to show the orientation information and the other to show a cube. The cube is created by styling six divs.

The orientation-related magic starts when we register the handler for the `orientationchange` event as follows:

```
Ext.device.Orientation.on('orientationchange', 'onDeviceOrientation',
this);
```

The handler receives the event object; it has three properties, as follows, that give information about the current orientation of the device:

- `alpha` – The angle by which the device frame was rotated around the z axis
- `beta` – The angle by which the device frame was rotated around the x axis
- `gamma` – The angle by which the device frame was rotated around the y axis

Based on the `alpha`, `beta`, and `gamma` values, we applied CSS 3D transformation to the cube on the following line:

```
Ext.getCmp('cube').element.dom.style.webkitTransform =
'rotateX('+beta+'deg) rotateY('+alpha+'deg)';
```

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Managing notifications

In various examples, we used `Ext.Msg.alert()` to show a notification message to the user. The thing to note is that it is a Sencha Touch component that shows a message in the form of a pop-up dialog. It looks similar on every platform unless you apply platform-specific styles. This recipe will explain how to show notifications with a complete native look and feel and behavior, and also how we can vibrate the phone to notify the user of some application event.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder `ch10` in the `www` folder. We will be using this new folder to keep the code.

How to do it...

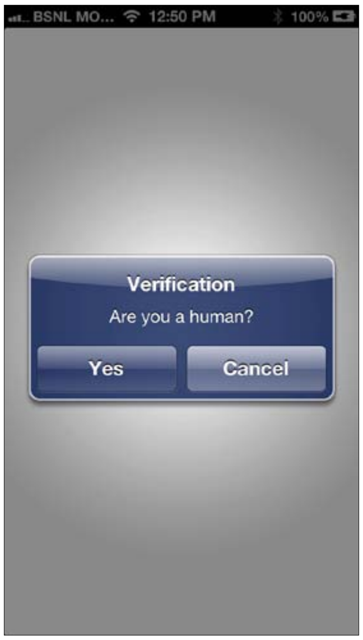
Follow the ensuing steps:

1. Create and open a new file `ch10_04.js` and paste the following code in it:

```
Ext.application({
  name : 'MyApp',
  requires : 'Ext.device.Notification',

  launch : function() {

      Ext.device.Notification.show({
       title : 'Verification',
       message : 'Are you a human?',
       buttons : [Ext.MessageBox.YES,
                    Ext.MessageBox.CANCEL],
       callback : function(button) {
          if (button === "yes") {
             alert('Verified');
          } else {
             alert('Nope');
          }
       }
    });


    Ext.device.Notification.vibrate();
  }
});
```

2. Include `ch10_04.js` in place of `ch10_03.js` in the `index.html` file.

3. Deploy and access it from the device of your choice. You will see a native notification message as shown in the following screenshot:



How it works...

In the preceding code, we have used Sencha's Notification API to show the native notification message. The API provides three different implementations:

- Sencha – This uses the native API to show the notification
- Cordova – This uses Cordova APIs to show the notification
- Simulator – This uses the `Ext.MessageBox` class to show the notification

The `buttons` config accepts an array or one or more button objects/configs where we added the **Yes** and **Cancel** buttons. When a button is tapped, the `callback` function receives the name of the button that was tapped by the user.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Reading a file

On a phone, there are various types of files and folders that we create in storage. In the *Capturing and managing photos using the camera* recipe, we saw how to fetch a photo from a library/album or capture a live photo. The limitation with this API is that it works well with photos but not with other files. Also, it allows us to select a single photo at a time. If you are building an e-mail kind of application where you want to be able to attach one or more different types of documents, the File API is something that can help you implement this functionality. It is based on the HTML 5 File API specification outline described at the following URLs:

- `http://www.w3.org/TR/FileAPI/`
- `http://www.w3.org/TR/file-upload/`

In this recipe, we will see how we can read one or more photos/images from the system using the HTML 5 APIs.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder `ch10` in the `www` folder. You will be using this new folder to keep the code.

How to do it...

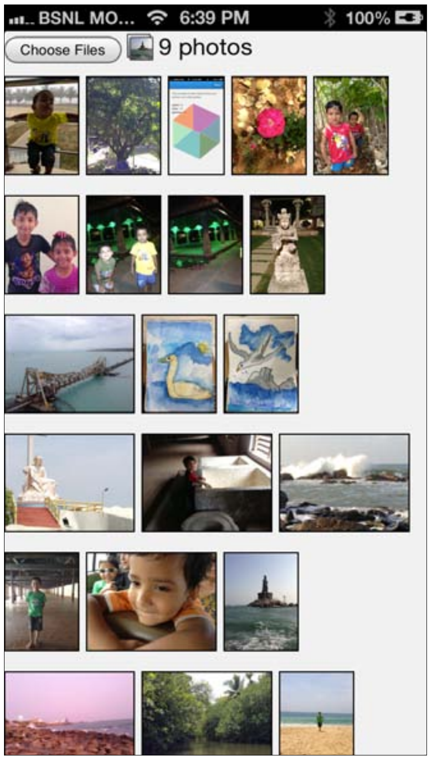
Follow the ensuing steps:

1. Create and open a new file `ch10_05.js` and paste the following code in it:

```
Ext.application({
    name: 'MyApp',
    requires: 'Ext.device.Contacts',

    launch: function() {

       var me = this;

       Ext.Viewport.add(Ext.create('Ext.Container', {
          html: '<input type="file" id="files" name="files[]"
multiple />' +
                '<output id="list"></output>',
          listeners: {
              painted: function() {
                 Ext.get('files').dom.
addEventListener('change', me.handleFileSelect, false);
              }
          }
       }));


    },
handleFileSelect: function(evt) {
        var files = evt.target.files; // FileList object

        // Loop through the FileList and render image files as
thumbnails.
        for (var i = 0, f; f = files[i]; i++) {

            // Only process image files.
            if (!f.type.match('image.*')) {
                continue;
            }

            var reader = new FileReader();

            // capture the file information.
            reader.onload = (function(theFile) {
                             return function(e) {
                             // Render thumbnail
                             var span = document.
createElement('span');
```

```
                                span.innerHTML = ['<img class="thumb"
src="', e.target.result,
                                                  '" title="',
escape(theFile.name), '"/>'].join('');
                                Ext.get('list').dom.
insertBefore(span);
                                };
                                })(f);

                // Read in the image file as a data URL.
                reader.readAsDataURL(f);
            }
        }

});
```

2. Include `ch10_05.js` in place of `ch10_04.js` in the `index.html` file.
3. Deploy and access it from a device of your choice.
4. Tap on the **Choose Files** button to select one or more photos. Once the photos/images are uploaded, you will see the images appended to the container as shown in the following screenshot:



How it works...

In the preceding code, we created a container with an `input` and an `output` element. The `type-"file"` input on the `input` element discloses that you are using the file upload functionality of the `input` element. So, when this element is rendered on the screen, it appears as a button with the label **Choose Files**. Clicking on this button allows the user to upload one or more files. As soon as the user confirms the upload, the `change` event is fired on the input element. We registered the handler for this inside the `painted` event handler as soon as the container was shown to the user.

The `change` event handler iterates through all the files and picks up the image files. We then registered the `onload` event on the `FileReader` object, which is called when the file data is uploaded. The handler adds the file's thumbnail to the container.

After registering the event handler, we called `readAsDataURL` to read the file's content.

> The kind of storage that is accessible to a File API varies from one platform to another. For example, on an iPhone and iPad it only allows us to select photos and videos, whereas on Android it allows us to pick up files from Dropbox and files on the device storage as well. So you may have to review the kind of storage access that you require and whether the File API implementation on that platform allows the implementation or not.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Handling the home button on Android, iOS, and BlackBerry

Imagine if you have a scheduled task in your application that periodically reads the new messages from a remote data source. Now, the battery is a very critical resource on a mobile device and must be managed well by any application. So when the home button is clicked, which puts the application in the background, you may want to shut down this activity to save resources; conversely, when the application becomes active, you may want to resume it. In this recipe, we will see how we can track when the application was sent to the background (that is, when the home button was pressed) and when it becomes active.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder `ch10` in the `www` folder. We will be using this new folder to keep the code.

How to do it...

Follow the ensuing steps:

1. Create and open a new file `ch10_06.js` and paste the following code in it:

```
Ext.application({
  name : 'MyApp',

  launch : function() {
    //Home button
    document.addEventListener("deviceready", function() {
       var counter = 0;
       document.addEventListener("pause", function() {
         counter++;
       }, false);

       document.addEventListener("resume", function() {
          alert('Value is: ' + counter);
       }, false);

    }, false);

  }
});
```

2. Include `ch10_06.js` in place of `ch10_05.js` in the `index.html` file.
3. Deploy and access it from a device of your choice.
4. Tap the home button. This will send the application to the background.
5. Bring the application to the foreground. This is done differently on different platforms. For example, on an iPhone, you need to tap the home button twice to view the running tasks and tap on our task. You will see an alert showing **Value is: 1**.
6. Repeat steps 4 and 5 and you will see the counter incrementing.

How it works...

In the preceding code, we have used Cordova to detect when the application is sent to the background and foreground. It gives us the following two events:

- `pause` – This is fired when the home button is pressed to send the application to the background
- `resume` – This is fired when the application becomes active

Both the events are available on the document. So, we registered the handlers for the `pause` and `resume` events inside the `deviceready` event handler. The `pause` event handler increments the value of `counter`. This will keep track of how many times the application was sent to the background. This is where you will put your code to suppress/stop/unregister any functionality that needs to be active only if the application is active. The `resume` handler is just showing the value of `counter`.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the iOS-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Handling the back button on Android and BlackBerry

Android and BlackBerry devices have a back button that helps us to get back to the previous state of the application. When we are building a Sencha Touch application where we have multiple screens and one is shown after the other, it becomes necessary to link our application state with the device's back button for the sake of usability. In this recipe, we will see how to detect if the user has pressed the back button.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder `ch10` in the `www` folder. You will be using this new folder to keep the code.

How to do it...

Follow the ensuing steps:

1. Create and open a new file `ch10_07.js` and paste the following code in it:

```
Ext.application({
  name : 'MyApp',
  launch : function() {
    //Back button
    document.addEventListener("deviceready", function() {
      var i = 0;
      document.addEventListener("backbutton", function() {
        alert('Back button pressed!');
      }, false);
    }, false);

  }
});
```

2. Include `ch10_07.js` in place of `ch10_06.js` in the `index.html` file.
3. Deploy and access it from a device of your choice.
4. Press the back button.

How it works...

In the preceding code, we have used the `backbutton` event offered by Cordova. This is a platform-neutral way to detect if the back button was pressed on the device. It is the handler where your application-specific logic should go.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1, Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1, Gear Up for the Journey*

Handling the menu button on Android and Blackberry

On Android and BlackBerry devices, we find a menu button; clicking on it brings up the application menu. Say you have a menu created in your Sencha Touch application using **action sheet** and would like to show the action sheet when the user clicks on the menu button; this recipe will show you how to detect whether the menu button was pressed and how to handle it.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder `ch10` in the `www` folder. You will be using this new folder to keep the code.

How to do it...

Follow the ensuing steps:

1. Create and open a new file `ch10_08.js` and paste the following code in it:

```
Ext.application({
  name : 'MyApp',

  launch : function() {

    //Menu button
    document.addEventListener("deviceready", function() {
      var i = 0;
      document.addEventListener("menubutton", function() {
         alert('Menu button pressed!');
      }, false);
    }, false);

  }
});
```

2. Include `ch10_08.js` in place of `ch10_07.js` in the `index.html` file.
3. Deploy and access it from a device of your choice.
4. Press the menu button. You will see the alert message.

How it works...

The preceding code uses the `menubutton` event fired by Cordova. The event is available on the document.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*
- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Handling the search button on Android

While the back, home, and menu buttons are listed as the mandatory buttons that every Android device must support, there is an additional optional button called search that may be present on some devices. For example, HTC Android models have the search button whereas some devices, such as Samsung Galaxy models, do not have it. In case you have a need to support the search button, this recipe will show you how to detect whether the search button was pressed.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder `ch10` in the `www` folder. You will be using this new folder to keep the code.

How to do it...

Follow the ensuing steps:

1. Create and open a new file `ch10_09.js` and paste the following code in it:

```
Ext.application({
  name : 'MyApp',

  launch : function() {

    //Search button
    document.addEventListener("deviceready", function() {
      var i = 0;
      document.addEventListener("searchbutton", function() {
```

```
               alert('Search button pressed!');
             }, false);
           }, false);

         }
       });
```

2. Include `ch10_09.js` in place of `ch10_08.js` in the `index.html` file.
3. Deploy and access it from a device of your choice.
4. Press the search button. You will see the alert message.

How it works...

In the preceding code, we have used the Cordova API; it fires the `searchbutton` event when the search button is pressed.

See also

- The *Setting up the Android-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Navigating using the BlackBerry trackpad

Sencha Touch supports the WebOS 6+ version, and many of you may have a need to support BlackBerry devices in your application. Some of these devices are not touch devices, and it expects the applications to respond to their trackpad events. A trackpad is used to scroll as well as click/tap. But, Sencha Touch does not provide in-built support for a trackpad. In this recipe, we will see how we can implement scrolling on the containers and tap on the list item using a trackpad.

Getting ready

Make sure that you have set up your development environment by following the recipes outlined in *Chapter 1*, *Gear Up for the Journey*.

Create a new folder `ch10` in the `www` folder. You will be using this new folder to keep the code.

How to do it...

Follow the ensuing steps to implement scrolling on the containers and tapping on the list item using a trackpad:

1. Create and open a new file `ch10_10.js` and paste the following code in it:

```
Ext.application({
  name : 'MyApp',

  launch : function() {
    //this is where rest of the code will go
  }
});
```

2. To handle the autoscrolling for a container, add the following code inside the `launch` method:

```
    Ext.define('Wtc.tux.ContainerOverride', {
      override : 'Ext.Container',

      initConfig : function() {
        Ext.apply(this.initialConfig, {});
        this.callParent(arguments);

        //If it is BlackBerry, register the handler for mousemove
to handle scrolling
        if (Ext.os.is.BlackBerry) {
          var task = Ext.create('Ext.util.DelayedTask', function()
{
            TrackPadUtils.handleScroller(this.event, this);
          });

          this.element.on('mousemove', function(event) {
            this.event = event;
            task.delay(30);
            task.setScope(this);
          }, this);
        }
      }

    });
```

3. The preceding code calls the `handlerScroller` method on `TrackPadUtils`. Add the following code to the `launch` method:

```
Ext.define('TrackPadUtils', {
      statics : {
        handleScroller: function(evt, view, el) {
          var element = evt;
          var margin = null;
          var scrollerSize = null;
          var containerSize = null;
          if( view.getScrollable() ) {
            view.scroller = view.getScrollable().getScroller();
            margin = view.element.getHeight()+25;
            scrollerSize = view.scroller.getSize();
            containerSize = view.scroller.getContainerSize();
            if( !view.initialScroll ) {
              view.initialScroll = 0;
            }

            if( evt.pageY >= margin && (( scrollerSize.y-
containerSize.y )>view.initialScroll) ) {
              view.initialScroll = view.initialScroll+8;
              view.scroller.scrollTo(containerSize.x,
                             view.initialScroll,true);
              }else if ( evt.pageY < 70 && view.initialScroll > 0)
{
              view.initialScroll = view.initialScroll - 8;
              view.scroller.scrollTo(containerSize.x,
view.initialScroll,true);
              }
          }
        }
      }
});
```

4. To support the list of item tap and disclosure features using a trackpad, add the following code:

```
Ext.define('Wtc.tuc.bb.List', {
      xtype : 'bblist',
      extend : 'Ext.dataview.List',

      initialize : function() {
        this.callParent();
        if(Ext.os.is.BlackBerry) {
          //list item tap handling
          this.element.on({
            mouseup : this.handleListTap,
            delegate: '.' + Ext.baseCSSPrefix + 'list-item-body',
            scope : this
          });

          //item disclosure icon tap handling
          this.element.on({
            mouseup : this.handleItemDisclosure,
            delegate: '.' + this.getBaseCls() + '-disclosure',
            scope : this
          });
        }
      },
      handleListTap: function(event,target,e) {
        this.onItemTap(this,target,index,e);
      }
    });
```

How it works...

The BlackBerry trackpad fires mouse events, out of which we have handled the following two events:

- `mousemove` – This is handled to implement scrolling
- `mouseup` – This is handled to implement item tap and list disclosure

In the preceding code, we first defined an override class `Wtc.tux.ContainerOverride` to register the `mousemove` event handler for the BlackBerry device. You may want to check specifically for the model number that supports the trackpad, which you can do by looking into the `UserAgent` string. A typical `UserAgent` string for BlackBerry applications is as follows:

```
Mozilla/5.0 (BlackBerry; U; BlackBerry AAAA; en-US)
AppleWebKit/534.11+ (KHTML, like Gecko) Version/X.X.X.X Mobile
Safari/534.11+
```

The letters `AAAA` represent the BlackBerry model number from which the request was initiated, for example, 9800. The letter `X.X.X.X` represents the version of the OS running on the device, for example, 6.0.0.141.

The override class relies on `TrackPadUtils.handleScroller` to handle the `mousemove` event and implement the required scrolling on a container. The `TrackPadUtils` class implements `handleScroller` as a static method. The method detects the direction in which the user is trying to scroll by working with the following variables:

- The container's height
- The height of the `scroller` function associated with the container
- The (x,y) coordinate where the `mousemove` event had occurred
- The vertical margin that you want to consider before deciding to scroll up/down—we have used 25 for bottom scrolling and 70 for upward scrolling

Based on the determined scrolling direction, we scrolled by 8 pixels. Note that this method only scrolls vertically. However, it will be extended to support horizontal scrolling.

To try out this code, you can now create a container and add any number of items to it with scrolling enabled.

We then defined a new class `Wtc.tux.bb.List` where we extended Sencha Touch's list component to add support for item tap and item disclosure using a trackpad. We registered the `mouseup` event handler on the list item, which calls `handleListTap`. The handler, in turn, calls the existing method `onItemTap` that handles the tap behavior on the list. For the disclosure, we registered the existing list method `handleItemDisclosure` as a `mouseup` event handler on the disclosure icon.

See also

- The *Setting up the BlackBerry-based development environment* recipe in *Chapter 1*, *Gear Up for the Journey*

Index