



Community Experience Distilled

# Sencha Touch 2 Mobile JavaScript Framework

Get started with Sencha Touch and build awesome, native-quality mobile web applications

John Earl Clark  
Bryan P. Johnson

[PACKT] open source\*  
PUBLISHING  
community experience distilled

[www.allitebooks.com](http://www.allitebooks.com)

# Sencha Touch 2 Mobile JavaScript Framework

Get started with Sencha Touch and build awesome,  
native-quality mobile web applications

**John Earl Clark**

**Bryan P. Johnson**



BIRMINGHAM - MUMBAI

# Sencha Touch 2 Mobile JavaScript Framework

Copyright © 2013 Packt Publishing

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

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

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

First published: November 2013

Production Reference: 1191113

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

ISBN 978-1-78216-074-8

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Abhishek Pandey ([abhishek.pandey1210@gmail.com](mailto:abhishek.pandey1210@gmail.com))

# Credits

**Authors**

John Earl Clark  
Bryan P. Johnson

**Project Coordinator**

Joel Goveya

**Reviewers**

Paul David Clegg  
Alex Graham  
Juris Vecvanags

**Proofreaders**

Simran Bhogal  
Maria Gould  
Ameesha Green  
Paul Hindle

**Acquisition Editors**

Usha Iyer  
James Jones

**Indexer**

Monica Ajmera Mehta

**Lead Technical Editor**

Sweny M. Sukumaran

**Production Coordinator**

Shantanu Zagade

**Technical Editors**

Vrinda Nitesh Bhosale  
Ritika Singh  
Nikhita K. Gaikwad

**Cover Work**

Shantanu Zagade

**Copy Editors**

Sarang Chari  
Janbal Dharmaraj  
Tanvi Gaitonde  
Alfida Paiva  
Kirti Pai  
Shambhavi Pai

# About the Authors

**John Earl Clark** holds a Master's degree in Human Computer Interaction from Georgia Tech and an undergraduate degree in Music Engineering from Georgia State University. He and his co-author, Bryan P. Johnson, worked together at MindSpring and, later, EarthLink; starting out in Technical Support and Documentation before moving into application development and, finally, the management of a small development team. After leaving EarthLink in 2002, John began working independently as a consultant and a programmer, before starting Twelve Foot Guru, LLC. with Bryan in 2005.

He has been working with Sencha Touch since its first beta releases. He has also worked with Sencha's ExtJS since its early days when it was still known as YUI-Ext. He has also previously written a book with Bryan Johnson called *Sencha Touch Mobile JavaScript Framework*, Packt Publishing.

When he is not buried in code, John spends his time woodworking, playing guitar, and brewing his own beer.

---

I would like to thank my family for all of their love and support. I would also like to thank Bryan for his help, patience, and continued faith in our efforts.

---

**Bryan P. Johnson** is a graduate of the University of Georgia. He went to work for MindSpring Enterprises in late 1995, where he met his co-author John Earl Clark. At MindSpring and later, EarthLink; Bryan served in multiple positions for over seven years, including the Director of System Administration and Director of Internal Application Development. After leaving EarthLink, he took some time off to travel before joining John to start Twelve Foot Guru.

Bryan has worked with Sencha's products since the early days of YUI-Ext and has used Sencha Touch since its first betas.

---

I would like to thank my family for their support, and my co-author John for his patience during the creation of this book.

---

# About the Reviewers

**Paul David Clegg** is a software engineer and part-time photographer from Manchester, United Kingdom. He attended the University of Manchester from 2006-2010, graduating with a B.Sc. in Computer Science and, shortly after, an M.Sc. in Software Engineering. Web-based and mobile technologies have been the main focus throughout his career, although semantic systems and Augmented Reality have also played their part.

While studying at the university, he looked at how Augmented Reality could be used in a location-based service. In 2010, he produced a content management system for AR mobile apps, turning Google SketchUp models into points of interest that could show the location of the user at full scale using Augmented Reality on a mobile device.

He moved on from studying to developing a web-based mobile platform serving dynamic content to mobile apps for iPhone and Android. The platform used popular technologies, such as Sencha Touch, Cordova, and Zend.

After working with various creative agencies around the country, he eventually started his own company in 2012, Gather Digital. The company specializes in scalable digital asset management systems and adaptive web development.

**Alex Graham** is a graduate of Southampton University in History and Media and holds an M.Sc. in IT from De Montfort University. He is a developer and works mainly on the Microsoft Technology Stack. His interest in mobile app development has led him to Sencha Touch, which he has worked with since Version 1, and written about on his blog that can be found at <http://lalexgraham.me.uk>. He lives in Birmingham, UK, with his wife and two children.

---

For Jenny, Sophie, and Elliott. With love.

---

**Juris Vecvanags** started a career in the IT field in the early 90s. During this time, he had the chance to work with a broad range of technologies and share his knowledge with Fortune 500 companies as well as private and government customers.

Before moving to Silicon Valley, he had a well-established web design company in Europe. Currently, he is working as Senior Solutions Engineer at Sencha Inc., helping customers write better apps, both for desktops and emerging mobile platforms.

When away from the office, he speaks at meet-ups across the San Francisco bay area and Chicago. Among the topics he speaks about are Node.js, ExtJS, Sencha Touch.

He is passionate about bleeding edge technologies and everything JavaScript-related.

---

I would like to thank Hyle Campbell for giving me the opportunity to work with Packt Publishing and mentoring this review. Also, a big thanks to my wife Baiba for her endless support while working on this book.

---

# [www.PacktPub.com](http://www.PacktPub.com)

## **Support files, eBooks, discount offers and more**

You might want to visit [www.PacktPub.com](http://www.PacktPub.com) for support files and downloads related to your book.

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

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



<http://PacktLib.PacktPub.com>

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

### **Why Subscribe?**

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

### **Free Access for Packt account holders**

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



# Table of Contents

<b>Preface</b>	<b>1</b>
<b>Chapter 1: Let's Begin with Sencha Touch</b>	<b>7</b>
<b>Frameworks</b>	<b>8</b>
Building from a foundation	10
Building with a plan	11
Building with a community	12
<b>Mobile application frameworks</b>	<b>12</b>
Native application versus web application	12
Web-based mobile frameworks	14
Web frameworks and touch technology	15
<b>Designing applications for mobile devices and touch technology</b>	<b>16</b>
Why touch?	17
<b>Getting started with Sencha Touch</b>	<b>18</b>
The API	18
Examples	19
The Kitchen Sink application	20
Learn	21
Forums	21
<b>Setting up your development environment</b>	<b>21</b>
Setting up web sharing on Mac OS X	22
Installing a web server on Microsoft Windows	22
Download and install the Sencha Touch framework	24
<b>Additional tools for developing with Sencha Touch</b>	<b>25</b>
Safari and Chrome Developer Tools	25
JavaScript Console	26
The Network tab	27
The web inspector	28
The Resources tab	29

*Table of Contents*

---

Other Sencha products	30
Sencha Cmd	30
Sencha Architect	30
Sencha Animator	30
Third-party developer tools	31
Notepad++	31
WebStorm	31
Xcode 5	31
Android Emulator	32
YUI test	32
Jasmine	32
JSLint	32
Summary	33
<b>Chapter 2: Creating a Simple Application</b>	<b>35</b>
<b>Setting up the application</b>	<b>36</b>
Getting started with Sencha Cmd	36
Creating the app.js file	40
Creating the Main.js file	44
Exploring the tab panel	45
Adding a panel	47
<b>Controlling the look with layouts</b>	<b>48</b>
Using a fit layout	50
Using a vbox layout	51
Using an hbox layout	53
<b>Testing and debugging the application</b>	<b>54</b>
Parse errors	54
Case sensitivity	55
Missing files	55
The web inspector console	55
<b>Updating the application for production</b>	<b>56</b>
<b>Putting the application into production</b>	<b>57</b>
Summary	59
<b>Chapter 3: Styling the User Interface</b>	<b>61</b>
<b>Styling components versus themes</b>	<b>61</b>
<b>UI styling for toolbars and buttons</b>	<b>63</b>
Adding the toolbar	63
Styling buttons	65
The tab bar	69
<b>Sencha Touch themes</b>	<b>70</b>
Introducing Sass and Compass	70
Variables in Sass	70
Mixins in Sass	72
Nesting in Sass	73

---

---

*Table of Contents*

Selector inheritance in Sass	76
Compass	77
Sass + Compass = themes	78
<b>Setting up Sass and Compass</b>	<b>78</b>
Installing Ruby on Windows	78
<b>Creating a custom theme</b>	<b>79</b>
Base color	81
Mixins and the UI configuration	81
Adding new icons	83
Variables	84
More Sass resources	85
<b>Default themes and theme switching</b>	<b>86</b>
<b>Images on multiple devices with Sencha.io Src</b>	<b>89</b>
Specifying sizes with Sencha.io Src	91
Sizing by formula	91
Sizing by percentage	91
Changing file types	93
<b>Summary</b>	<b>93</b>
<b>Chapter 4: Components and Configurations</b>	<b>95</b>
<b>The base component class</b>	<b>96</b>
<b>Taking another look at layouts</b>	<b>96</b>
Creating a card layout	97
Creating an hbox layout	98
Creating a vbox layout	100
Creating a fit layout	102
Adding complexity	103
<b>The TabPanel and Carousel components</b>	<b>108</b>
Creating a TabPanel component	108
Creating a Carousel component	110
<b>Creating a FormPanel component</b>	<b>112</b>
Adding a DatePicker component	115
Adding sliders, spinners, and toggles	116
<b>The MessageBox and Sheet components</b>	<b>117</b>
Creating a MessageBox component	118
Creating a Sheet component	121
Creating an ActionSheet component	124
<b>Creating a Map component</b>	<b>126</b>
<b>Creating lists</b>	<b>128</b>
Adding grouped lists	130
Adding nested lists	131

---

*Table of Contents*

<b>Finding more information with the Sencha Docs</b>	<b>135</b>
Finding a component	136
Understanding the component page	137
<b>Summary</b>	<b>138</b>
<b>Chapter 5: Events and Controllers</b>	<b>139</b>
<b>Exploring events</b>	<b>140</b>
Asynchronous versus synchronous actions	140
<b>Adding listeners and handlers</b>	<b>141</b>
<b>Controllers</b>	<b>146</b>
Refs and control	148
Referencing multiple items with ComponentQuery	151
<b>Getting more out of events</b>	<b>157</b>
Custom events	158
Exploring listener options	158
Taking a closer look at scope	160
Removing listeners	161
Using handlers and buttons	162
Exploring common events	162
<b>Additional information on events</b>	<b>163</b>
<b>Summary</b>	<b>164</b>
<b>Chapter 6: Getting the Data In</b>	<b>165</b>
<b>Models</b>	<b>165</b>
The basic model	166
Model validations	167
Model methods	170
Proxies and readers	171
<b>Introducing data formats</b>	<b>173</b>
Arrays	173
XML	174
JSON	176
JSONP	177
<b>Introducing stores</b>	<b>178</b>
A simple store	179
Forms and stores	181
Specialty text fields	183
Mapping fields to the model	184
Clearing the store data	186
Editing with forms	188
Switching handlers	189
<b>Deleting from the data store</b>	<b>192</b>
<b>Summary</b>	<b>193</b>

---

---

*Table of Contents*

<b>Chapter 7: Getting the Data Out</b>	<b>195</b>
<b>Using data stores for display</b>	<b>195</b>
Directly binding a store	196
Sorters and filters	197
Paging a data store	199
Loading changes in a store	201
Data stores and panels	202
<b>XTemplates</b>	<b>208</b>
Manipulating data	210
Looping through data	212
Numbering within the loop	213
Parent data in the loop	214
Conditional display	214
Arithmetic functionality	216
Inline JavaScript	217
XTemplate member functions	217
The isEmpty function	220
Changing a panel's content with XTemplate.overwrite	221
<b>Sencha Touch Charts</b>	<b>222</b>
Installing Sencha Touch Charts	222
A simple pie chart	223
A bar chart	225
<b>Summary</b>	<b>228</b>
<b>Chapter 8: Creating the Flickr Finder Application</b>	<b>229</b>
<b>Generating the basic application</b>	<b>229</b>
<b>Introducing the Model View Controller</b>	<b>231</b>
Splitting up the pieces	233
<b>Building the foundation with Sencha Cmd</b>	<b>234</b>
Installing Sencha Cmd	235
Using the Flickr API	238
<b>Adding to the basic application</b>	<b>239</b>
Generating controllers with Sencha Cmd	241
A brief word about including files	242
Creating the Photo data model	243
<b>Making the SearchPhotos components</b>	<b>244</b>
Creating the SearchPhotos store	244
Creating the SearchPhotos list	246
Creating the Navigation view	248
Creating the SearchPhotoDetails view	249
Creating the SearchPhotos controller	250
Setting up the launch function	251

---

*Table of Contents*

---

Using Ext.util.Geolocation	252
Listening to the list	255
<b>Building the SavedPhotos components</b>	<b>259</b>
Creating the SavedPhotos store	260
Creating the SavedPhoto views	261
Finishing up the Add button in SearchPhotos	263
Updating the SavedPhotos controller	265
<b>Polishing your application</b>	<b>267</b>
Adding application icons and startup screens	267
<b>Improving the application</b>	<b>268</b>
Summary	268
<b>Chapter 9: Advanced Topics</b>	<b>269</b>
<b>Talking to your own server</b>	<b>269</b>
Using your own API	270
REST	272
Designing your API	272
Creating the model and store	273
Making a request	276
Ajax requests in an API	278
<b>Going offline</b>	<b>279</b>
Syncing local and remote data	279
Manifests	282
Setting up your web server	284
Updating your cached application	285
Interface considerations	285
Alerting your users	285
Updating your UI	287
Alternate methods of detecting the offline mode	288
<b>Getting into the marketplace</b>	<b>289</b>
Compiling your application	289
Sencha Cmd	290
PhoneGap	291
Other options	292
Registering for developer accounts	292
Becoming an Apple developer	293
Becoming an Android developer	294
<b>Summary</b>	<b>294</b>
<b>Index</b>	<b>297</b>

---

# Preface

Since its initial launch, Sencha Touch has quickly become the gold standard for developing rich mobile web applications with HTML5. Sencha Touch is the first HTML5 mobile JavaScript framework that allows you to develop mobile web apps that look and feel like native apps on iPhone, Android, BlackBerry, and Windows Phone touch-screen devices. Sencha Touch is the world's first application framework built specifically to leverage HTML5, CSS3, and JavaScript for the highest level of power, flexibility, and optimization. It makes specific use of HTML5 to deliver components such as audio and video as well as a localStorage proxy for saving data offline. Sencha Touch also makes extensive use of CSS3 in its components and themes to provide an incredibly robust styling layer, giving you total control over the look of your application.

Sencha Touch enables you to design for multiple platforms without the need to learn multiple arcane programming languages. Instead you can leverage your existing knowledge of HTML and CSS to quickly create rich web applications for mobile devices in JavaScript. This book will show you how you can use Sencha Touch to efficiently produce attractive, exciting, and easy-to-use web applications that keep your visitors coming back for more.

The Sencha Touch Mobile JavaScript framework teaches you all you need to get started with Sencha Touch and build awesome mobile web applications. Beginning with an overview of Sencha Touch, this book will guide you through creating a complete simple application followed by styling the user interface and will explain the list of Sencha Touch components through comprehensive examples. Next, you will learn about the essential touch and component events, which will help you create rich, dynamic animations. The book follows this up with information about core data packages and dealing with data, and wraps it up with building another simple but powerful Sencha Touch application.

In short, this book has a step-by-step approach and extensive content to turn a beginner to Sencha Touch into an ace quickly and easily.

Exploit Sencha Touch, a cross-platform library aimed at next generation, touch-enabled devices.

## What this book covers

*Chapter 1, Let's Begin with Sencha Touch*, provides an overview of Sencha Touch and a walk-through of the basics of setting up the library for development. We will also talk about programming frameworks and how they can help you develop touch-friendly applications quickly and easily.

*Chapter 2, Creating a Simple Application*, starts out by creating a simple application and using it to discover the basic elements of Sencha Touch. We will also explore some of the more common components such as lists and panels, and we will show you how to find common errors and fix them when they occur.

*Chapter 3, Styling the User Interface*, explores ways to change the look and feel of individual components using CSS styles once we have our simple application. Then we will dive into using Sencha Touch themes to control the look of your entire application using SASS and Compass.

*Chapter 4, Components and Configurations*, examines the individual components for Sencha Touch in greater detail. We will also cover the use of layouts in each component, and how they are used to arrange the different pieces of your application.

*Chapter 5, Events and Controllers*, helps us take a look at the Sencha Touch events system, which allows these components to respond to the users' touch and communicate with each other. We will cover the use of listeners and handlers, and explore ways to monitor and observe events so that we can see what each part of our application is doing.

*Chapter 6, Getting the Data In*, explains the different methods for getting data into our application using forms to gather information from the user, ways to verify the data, and details about how to store it as data is a critical part of any application. We will also talk about the different data formats that are used by Sencha Touch, and how we can manipulate that data using Sencha Touch's models and stores.

*Chapter 7, Getting the Data Out*, will discuss the use of panels and XTemplates to display the data, as after we have data in our application, we need to be able to get it back out to display to the user. We will also take a look at using our data to create colorful charts and graphs using Sencha Touch Charts.

*Chapter 8, Creating the Flickr Finder Application*, creates a more complex application that grabs photos from Flickr, based on our current location, using the information we have learned about Sencha Touch. We will also use this as an opportunity to talk about best practices for structuring your application and its files.

*Chapter 9, Advanced Topics*, explores ways to synchronize your data with a database server by creating your own API. Additionally, we will look at ways to synchronize data between the mobile device and a database server, compiling your application with Phone Gap and NimbleKit. We will also look at ways to get started as an Apple iOS or Google Android developer.

## What you need for this book

To accomplish the tasks in the book, you will need a computer with the following items:

- Sencha Touch 2.1.x
- Sencha Cmd 3.1.x or greater
- A programming editor such as BBEdit, Text Wrangler, UltraEdit, TextMate, WebStorm, Aptana, or Eclipse
- A local web server such as the built-in Apple OSX web server, Microsoft's built-in IIS server, or the downloadable WAMP server and software package

Links for these items are provided in the *Setting up your development environment* section in *Chapter 1, Let's Begin with Sencha Touch*. Other optional but helpful software will be linked in specific sections when needed.

## Who this book is for

This book is ideal if you want to gain practical knowledge in using the Sencha Touch mobile web application framework to make attractive web apps for mobiles. You should have some familiarity with HTML and CSS. If you are a designer, this book will give you the skills you need to implement your ideas and if you are a developer, this book will provide you with creative inspiration through practical examples. It is assumed that you know how to use touch screens, touch events, and mobile devices such as Apple iOS, Google Android, Blackberry, and Windows Phone.

## 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: "The title at the top also lists the `xtype` value for the component right next to it."

A block of code is set as follows:

```
var nestedList = Ext.create('Ext.NestedList', {  
    fullscreen: true,  
    title: 'Minions',  
    displayField: 'text',  
    store: store  
});
```

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

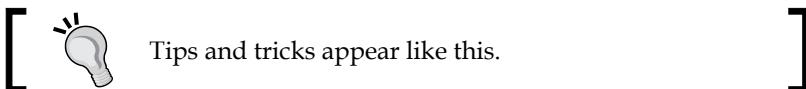
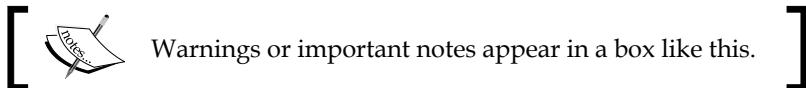
```

```

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

```
C:\Ruby192>ruby -v  
ruby 1.9.2p180 (2011-02-18) [i386-mingw32]
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes appear in the text like this: "There is also a **Select Code** option, which will let you copy the code and paste it into your own applications."



## 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](mailto: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](http://www.packtpub.com/authors).

## Customer support

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

## Downloading the example code

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

## Errata

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

## **Piracy**

Piracy of 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](mailto: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](mailto: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

## Let's Begin with Sencha Touch

With the growing popularity of mobile devices, cell phones, and tablet computers, consumers have quickly moved on to embrace touch-based operating systems and applications. This popularity has given developers a wide variety of platforms to choose from: Apple's iOS (including the iPhone, iPod Touch, and iPad), Google's Android, Windows 7 Mobile, and many more. Unfortunately, this rich variety of platforms brings with it an equally rich variety of programming languages to choose from. Picking any single language often locks you into using a specific platform or device.

Sencha Touch removes this obstacle by providing a framework based on JavaScript, HTML5, and CSS. These standards have gained strong support across most modern browsers and mobile devices. Using a framework based on these standards, you can deploy applications to multiple platforms without having to completely rewrite your code.

This book will help familiarize you with Sencha Touch, from the basic setup to building complex applications. We will also cover some of the basics of frameworks and touch-based applications in general, as well as provide tips on how to set up your development environment and deploy your applications in a number of different ways.

In this chapter, we will cover the following topics:

- Frameworks
- Mobile application frameworks
- Designing applications for Sencha Touch

- Getting started with Sencha Touch
- Setting up your development environment
- Additional tools for developing applications using Sencha Touch

## Frameworks

A framework is a reusable set of code that provides a collection of objects and functions you can use to get a head start on building your application. The main goal of a framework is to keep you from re-inventing the wheel each time you build an application.

A well-written framework also helps by providing some measure of consistency and gently nudging you to follow standard practices. This consistency also makes the framework easier to learn. The keys to reusability and ease of learning are two coding concepts called **objects** and **inheritance**.

Most frameworks such as Sencha Touch are built around an **Object-oriented Programming** style (also called **OOP**). The idea behind OOP is that the code is designed around simple base objects. A base object will have certain properties and functions that it can perform.

For example, let's say we have an object called `wheeledVehicle`. Our `wheeledVehicle` object has a few properties that are listed as follows:

- One or more wheels
- One or more seats
- A steering device

It also has a few functions that are listed as follows:

- `moveForward`
- `moveBackward`
- `moveLeft`
- `moveRight`
- `stop`

This is our base object. Once this base object is created, we can extend it to add more functionalities and properties. This allows us to create more complex objects, such as bicycles, motorcycles, cars, trucks, buses, and more. Each of these complex objects does a lot more than our basic wheeled object, but they also inherit the properties and abilities of the original object.

We can even override the original functions, such as making our `moveForward` function run quicker for the car than our bicycle, if needed. This means we can build lots of different `wheeledVehicles` instances without having to recreate our original work. We can even build more complex objects. For example, once we have a generic car, we can build everything from a Volkswagen to a Ferrari just by adding in the new properties and functions for the specific model.

Sencha Touch is also built upon the **OOP** concept. Let's take an example from Sencha Touch. In Sencha Touch, we have a simple object called `container`.

The `container` object is one of the basic building blocks of Sencha Touch and, as the name implies, it is used to contain other items in the visual areas of the application. Other visual classes, such as panel, toolbar, and form panel, extend the `container` class. The component class has many configurations that control simple things, such as the following:

- `height`
- `width`
- `padding`
- `margin`
- `border`

Configuration options also define more complex behaviors, such as the following:

- `layout`: This determines how items in the container will be positioned
- `listeners`: This determines which events the container should pay attention to and what to do when the container hears the event

The component object has a certain number of methods that control its behavior and configurations. Examples of some simple methods are as follows:

- `show`
- `hide`
- `enable`
- `disable`
- `setHeight`
- `setWidth`

It also supports more complex methods, such as the following:

- `query`: This performs a search for specific items within the container
- `update`: This takes HTML or data and updates the contents of the container

The container has a certain number of properties that you can use and events that you can listen for. For example, you can listen for the following events:

- show
- hide
- initialize
- resize

The basic `container` object is used as a building block in Sencha Touch to create other visual objects, such as panels, tabs, toolbars, form panels, and form fields. These sub objects, or **child** objects, inherit all of the abilities and attributes of the container object (the **parent** object). Each will include the same configuration options for height, width, and so on, and they will know how to do all the things a container can: show, hide, and so on.

Each of these child objects will also have additional, unique configurations and methods of their own. For example, buttons have an additional `text` property that sets their title and buttons are notified when a user taps on them. By extending the original object, the person creating the button only has to write code for these extra configurations and methods.

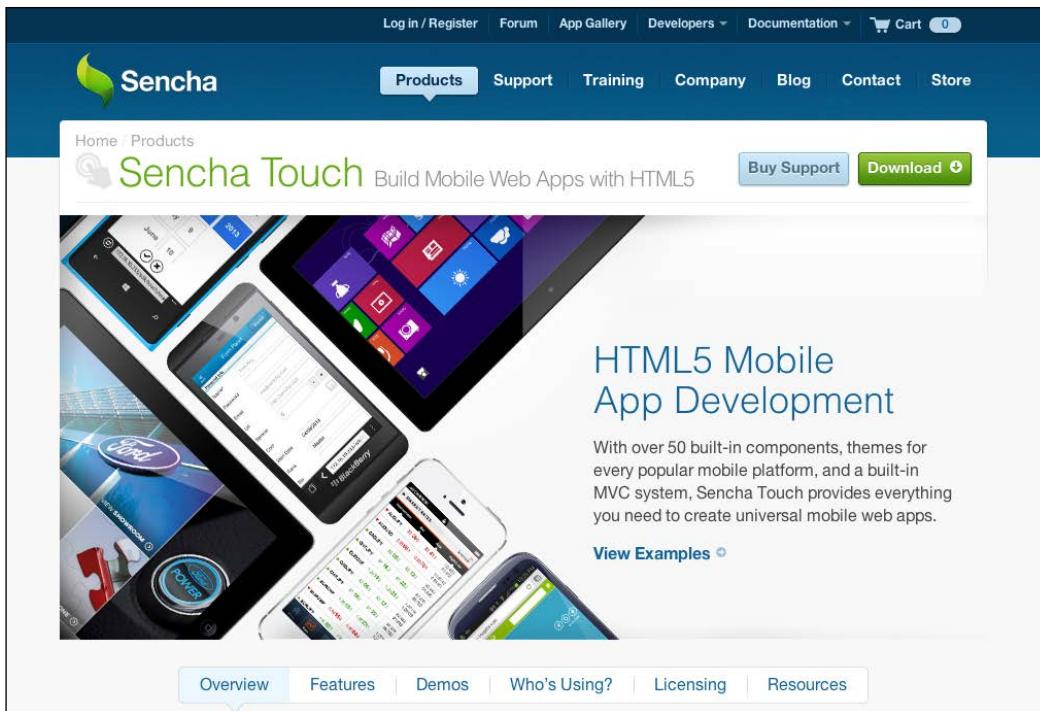
From a coding perspective, objects and inheritance make it so that we can re-use a lot of our work. It also means that when we encounter a new framework such as Sencha Touch, we can use what we learned about the basic code objects to quickly understand the more complex objects.

## Building from a foundation

In addition to providing reusability, frameworks also provide you with a collection of core objects and functions, commonly used to build applications. This keeps you from starting from scratch each time you begin a new application.

These code objects typically handle most of the ways a user will input, manipulate, or view data. They also cover the common tasks that occur behind the scenes in an application, such as managing data, handling sessions, dealing with different file formats, and formatting or converting different kinds of data.

The goal of a framework is to anticipate common tasks and provide the coder with pre-constructed functions to handle those tasks. Once you are familiar with the wide range of objects and functions provided by a framework such as Sencha Touch, you can develop your applications quickly and more efficiently.



## Building with a plan

One of the key things to look for in any framework is the documentation. A framework with no documentation or, worse yet, one with bad documentation is simply an exercise in frustration. Good documentation provides low-level information about every object, property, method, and event in the framework. It should also provide more generalized information, such as examples of how the code is used in various situations.

Providing good documentation and examples are the two places in which Sencha Touch excels as a framework. Extensive information is available on the main Sencha website, <http://docs.sencha.com>, under **Sencha Documentation Resources | Touch**. There is documentation for every version of Sencha Touch since Version 1.1.0. In this book, we use Version 2.2.1, so clicking on the **Touch 2.2.1** link will take you to the relevant documentation. You can also download the documentation as a ZIP file.

A well-designed framework also maintains a set of standards and practices. These can be simple things such as using camel case for variable names (for example, `myVariable`) or more complex practices for commenting on and documenting the code. The key to these standards and practices is consistency.

Consistency allows you to quickly learn the language and understand intuitively where to find the answers to your questions. It's a little like having a plan for a building; you understand quickly how things are laid out and how to get where you need to go.

A framework will also help you understand how to structure your own applications by providing examples for both structure and consistency in coding.

In this regard, Sencha has made every effort to encourage consistency, observe standards, and provide extensive documentation for the Sencha Touch framework. This makes Sencha-Touch a very effective first language for beginners.

## **Building with a community**

Frameworks seldom exist in isolation. Groups of developers tend to collect around specific frameworks and form communities. These communities are fantastic places to ask questions and learn about a new language.

As with all communities, there are a number of unwritten rules and customs. Always take the time to read through the forum before posting a question, just in case the question has already been asked and answered.

Sencha Touch has an active developer community with a forum that can be accessed from the main Sencha website at <http://www.sencha.com/forum> (scroll down on the website to find the Sencha Touch-specific forums).

## **Mobile application frameworks**

Mobile application frameworks need to address different functionalities from a standard framework. Unlike a traditional desktop application, mobile devices deal with touches and swipes instead of mouse clicks. The keyboard is part of the screen, which can make traditional keyboard navigation commands difficult, if not impossible. Also, there are various screen sizes and resolutions to choose from in mobile devices. So, the framework has to adjust itself in accordance with the screen and type of device. Mobile devices do not have as much computing power as desktops or as many resources, so mobile frameworks should consider these limitations. In order to understand these constraints, we can begin by looking at different types of mobile frameworks and how they work.

## **Native application versus web application**

There are two basic types of mobile application framework: one that builds **native applications** and another that builds **web-based applications**, such as Sencha Touch.

A native application is one that is installed directly on the device. It typically has more access to the device's hardware (camera, GPS, positioning hardware, and so on) and other programs on the device, such as the address book and photo album. Updates to a native application typically require each user to download a new copy of the program being updated.

Web-based applications, as the name implies, require a public web server that your users will need to access to use the application. Users will navigate to your application's website using the browser on their mobile device. Since the application runs inside the web browser, it has less access to the local filesystem and hardware, but it also doesn't require the user to walk through a complex download and installation process. Updates to a web-based application can be accomplished by making a single update to the public web server. The program is then updated automatically for anyone who accesses the site.

Web-based applications can also be modified to behave more like native applications or even be compiled by a separate program to become a full native application.

Most mobile browsers allow users to save the application to the desktop of a mobile device. This will create an icon on the mobile device's home screen. From there, the application can be launched and will behave much like a native application. The browser's navigation will be invisible when the application is launched from the home screen icon. Web applications can also use mobile devices' built-in storage capacity, such as using **HTML5 local storage** to store data on a device and making the application work offline without network connectivity.



If you find that you need the full functionality of a native application, you can use the Sencha Cmd command-line utility or an external compiler such as PhoneGap (<http://www.phonegap.com/>) to take your web-based application and compile it into a full native application that you can upload and sell on Apple's App Store or the Google Play store. We will talk more about these options a bit later in the book.

## Web-based mobile frameworks

A web-based mobile framework depends on the web browser running the application. This is a critical piece of information for a couple of reasons.

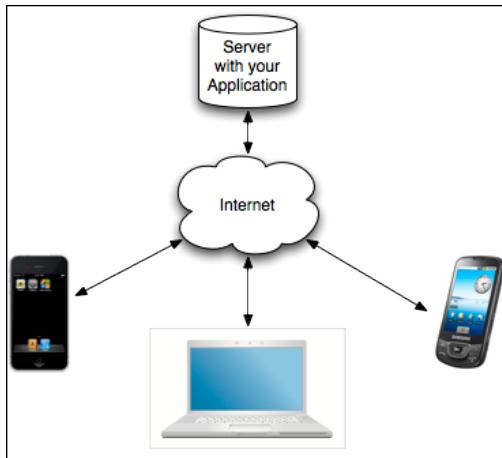
The first consideration for a web-based application is that the web browser has to be consistent across mobile platforms. If you have previously done any website development, you are familiar with the painful issue of browser compatibility. A website can look completely different depending on the browser. A JavaScript that works in one browser may not work in another. People also tend to hold on to older browsers without updating them. Fortunately, these problems are less of an issue with most mobile devices and no problem at all for iOS and Android.

The web browser for both Apple's iOS and Google's Android is based on the **WebKit** engine. WebKit is an open source engine that basically controls how the browser displays pages, handles JavaScript, and implements web standards. What this means for you is that your application should work in the same way on both platforms.

However, mobile devices that do not use WebKit (such as Windows Mobile) will be unable to use your application. The good news is that as more browsers adopt HTML5 standards, this problem may also begin to disappear.

The second consideration for a web-based application is where it lives. A native application gets installed on the user's device. A web-based application needs to be installed on a public server. Users should be able to enter a URL into their web browsers and navigate to your application. If the application exists only on your computer, you are the only one who can use it. This is great for testing, but if you want to have other people using your application, you will need to have it hosted on a public server.

The third consideration is connectivity. If a user cannot connect to the Internet, they won't be able to use your application. However, Sencha Touch can be configured to store your application and all of its data locally. At first glance, this ability seems to negate the problem of connectivity altogether, but it actually causes problems when users connect to your application with more than one device.



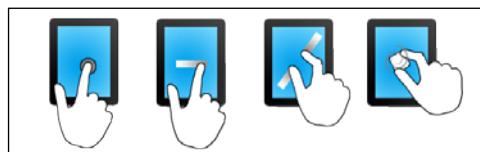
A web-based application resides on the Internet, so it can be accessed from any device that has a browser and a network connection. The same application can simultaneously be used on a cell phone, personal computer, and mobile device. This is the biggest advantage for an information-rich application if its data is stored on a central server. This means that data entered into one device can be accessed on another device.

But, if an application stores data locally, this would not be possible, as data entered into a mobile device is not accessible on a personal computer. If the user is using a personal computer to view the site, the application will create another set of local data.

Fortunately, Sencha Touch can be set up to synchronize data between the server and various other devices. When your application is connected to the Internet, it will synchronize any existing offline data and use the remote server to store any data while online. This makes sure that your data is accessible to you across all of your devices while allowing you to work offline as needed.

## Web frameworks and touch technology

Standard web application frameworks have been designed to work with the mouse-and-keyboard environment, but, mobile frameworks should work with touch technology for both navigation and data entry.



The following are the common touch gestures:

- **Tap:** A single touch on the screen
- **Double tap:** Two quick touches on the screen
- **Tap Hold:** Tapping once on the device and holding the finger down
- **Swipe:** Moving a single finger across the screen from left to right or top to bottom
- **Pinch or spread:** Touching the screen with two fingers and bringing them together in a pinching motion or spreading them apart to reverse the action
- **Rotate:** Placing two fingers on the screen and twisting them clockwise or counterclockwise, typically to rotate an object on the screen

Initially, these interactions were only supported in native applications, but Sencha Touch has made them available in web applications.

## **Designing applications for mobile devices and touch technology**

Mobile applications require some change in thinking. The biggest consideration is that of scale.

If you are used to designing an application on a 21 inch monitor, dealing with a 3.5 inch phone screen can be a painful experience. Phones and mobile devices use a variety of screen resolutions; some examples are as follows:

- **iPhone 5 retina display:** 1136 x 640
- **iPhone 5:** 960 x 640
- **iPhone 4 and iPod Touch 4:** 960 x 640
- **iPhone 4 and iPod Touch 3:** 480 x 320
- **Android 4 Phones:** These support four general sizes:
  - Xlarge screens are at least 960 x 720
  - Large screens are at least 640 x 480
  - Normal screens are at least 470 x 320
  - Small screens are at least 426 x 320
- **HTC Phones:** 800 x 480
- **Samsung Galaxy S3:** 1280 x 720
- **iPad:** 1024 x 768
- **iPad retina:** 2048 x 1536

Additionally, Android tablets come in a wide variety of resolutions and sizes. Designing applications for these various screen sizes can take a bit of extra effort.

When designing a mobile application, it's usually a good idea to mock up the design to get a better idea of scale and where your various application elements will go. There are a number of good layout programs available to help you with this, listed as follows:

- Omni Graffle for the Mac (<http://www.omnigroup.com/products/omnigraffle/>)
- Balsamiq Mockups for Mac, Windows, and Linux (<http://balsamiq.com/>)
- DroidDraw for Mac, Windows, and Linux (<http://www.droiddraw.org/>)
- iMockups for the iPad (<http://www.endloop.ca/imockups/>)

Touch applications also have certain considerations for you to keep in mind. If you are coming from a typical web development background, you might be used to using events such as hover.

Hover is typically used in web applications to alert the user that an action can be performed or to provide tool tips; for example, showing that an image or text can be clicked on by causing its color to change when the user hovers the mouse cursor over it. Since touch applications require the user to be in contact with the screen, there really is no concept of hovering. Objects the user can activate or interact with should be obvious and icons should be clearly labeled.

Unlike mouse-driven applications, touch applications are also typically designed to mimic real-world interactions. For example, turning the pages of a book in a touch application is usually accomplished by swiping your finger across the page horizontally in much the same way you would in the real world. This encourages exploration of the application, but it also means that coders must take special care with any potentially destructive actions, such as deleting an entry.

## Why touch?

Before the advent of touch screens, applications were generally limited to input from external keyboards and the mouse. Neither of these is very desirable on a mobile platform. Even when full internal keyboards are used in non-touch-based devices, they can take up a tremendous amount of space on the device, which in turn limits the available screen size. By contrast, a touch-based keyboard disappears when it isn't needed, leaving a larger screen area available for display.

Slide-out keyboards on mobile devices do not adversely affect the screen size, but they can cramp space and be uncomfortable to use. Additionally, a touch-screen keyboard allows for application-specific keyboards and keys, such as the *.com* key when using a web browser.

Keyboards and mouse devices can also present a mental disconnect for some users. Using a mouse on your desk to control a tiny pointer on a separate screen often leads to a sense that you are not entirely in control of the activity, whereas directly touching an object on the screen and moving it places you at the center of the activity. Because we interact with the physical world by touching and moving objects by hand, a touch-based application often provides a more intuitive **User Interface (UI)**.

Touch technology is also beginning to make big inroads into the desktop computer arena with the advent of Windows 8. As this technology becomes cheaper and more common, the need for touch-based applications will continue to grow.

## Getting started with Sencha Touch

When getting started with any new programming framework, it's a good idea to understand all of the resources available to you. Buying this book is a great start, but there are additional resources that will prove invaluable to you as you explore the Sencha Touch framework.

Fortunately for us, the Sencha website provides a wealth of information to assist you at every stage of your development.

## The API

The Sencha Touch **Application Programming Interface (API)** documentation provides detailed information about every single object class available to you with Sencha Touch. Every class in the API includes detailed documentation for every configuration option, property, method, and event for that particular class. The API often also includes a short example for each class with a live preview and code editor.

The API documentation is available on the Sencha website,  
<http://docs.sencha.com/touch/2.2.1/>.

A copy, as shown in the following screenshot, is also included as part of the Sencha Touch framework which you will download to create your applications:

The screenshot shows the Sencha Touch API documentation for the `Ext.Container` class. The top navigation bar includes tabs for `Configs`, `Properties`, `Methods`, and `Events`. Below the navigation is a search bar labeled "Filter class members" and a "Show" dropdown. The main content area contains a detailed description of `Ext.Container`, listing its abilities, extra functionality, and layout determinants. A section titled "Adding Components to Containers" provides code examples. To the right, a sidebar displays the class's structure, including "ALTERNATE NAMES", "HIERARCHY", "INHERITED MIXINS", "REQUIRES", and "SUBCLASSES".

## Examples

The Sencha website also includes a number of example applications for you to look at. By far the most helpful of these is the Kitchen Sink application. The following screenshot shows how a Kitchen Sink application looks:

The screenshot shows the Sencha Kitchen Sink application interface. On the left is a vertical navigation menu with items: Back, User Int..., Buttons, Forms, DataViews, Lists, Nested List, Icons (which is currently selected), Toolbars, Carousel, Tabs, Bottom Tabs, and Overlays. The main content area has a title "Icons". It displays a toolbar with various icons (e.g., back, forward, search, etc.) and a descriptive text block: "Both toolbars and tabbars have built-in, ready to use icons. Sencha Touch comes with over 300 icons that can optionally be included in your app via Sass and Compass." At the bottom, there is a footer bar with icons for Info, Download, Favorites, Bookmarks, and More.

## The Kitchen Sink application

The Kitchen Sink application provides examples for the following:

- User interface items, such as buttons, forms, toolbars, lists, and more
- Animations for actions, such as flipping pages or sliding in a form
- Touch events, such as tap, swipe, and pinch
- Data handling for JSON, YQL, and AJAX
- Media handling for audio and video
- Themes to change the look of your application

Each example has a **Source** button in the upper-right corner that will display the code for the Kitchen Sink example.

The Kitchen Sink application also provides **Event Recorder** and **Event Simulator**. These will allow you to record, store, and play back any touch events executed by the device's screen.

These simulators demonstrate how to record actions in your own application for playback as a live demonstration or tutorial. They can also be used for easily repeatable testing of functionality.

You can play around with the Kitchen Sink application on any mobile device or on a regular computer using Apple's Safari web browser. The Kitchen Sink application is available on the Sencha website, <http://docs.sencha.com/touch/2.2.1/touch-build/examples/kitchensink/>.

A copy of the Kitchen Sink application is also included as part of the Sencha Touch framework that you will download to create your applications.

## Learn

Sencha also has a section of the site devoted to more detailed discussions of particular aspects of the Sencha Touch framework. The section is appropriately titled **Learn**. It contains a number of tutorials, screencasts, and guides available for you to use. Each subsection is labeled as **Easy**, **Medium**, or **Hard** so that you have some idea about what you are getting into.

The **Learn** section is available on the Sencha Website at  
<http://www.sencha.com/learn/touch/>.

## Forums

The Sencha forums are worth mentioning again. These community discussions provide general knowledge, bug reporting, question-and-answer sessions, examples, contests, and more. The forums are a great place to find solutions from people who use the framework on a daily basis.

## Setting up your development environment

Now that you've familiarized yourself with the available Sencha Touch resources, the next step is to set up your development environment and install the Sencha Touch libraries.

In order to start developing applications using Sencha Touch, it is highly recommended that you have a working web server on which you can host your application. While it's possible to develop Sencha Touch applications by viewing local folders using your web browser, without a web server, you won't be able to test your application using any mobile devices.

## Setting up web sharing on Mac OS X

If you are using Mac OS X, you already have a web server installed. To enable it, launch your system preferences, choose **Sharing**, and enable **Web Sharing**. If you haven't done so already, click on **Create Personal Website Folder** to set up a web folder in your home directory. By default, this folder is named **Sites**, and this is where we will be building our application:



The **Sharing** panel will tell you your web server URL. Remember this for later.

## Installing a web server on Microsoft Windows

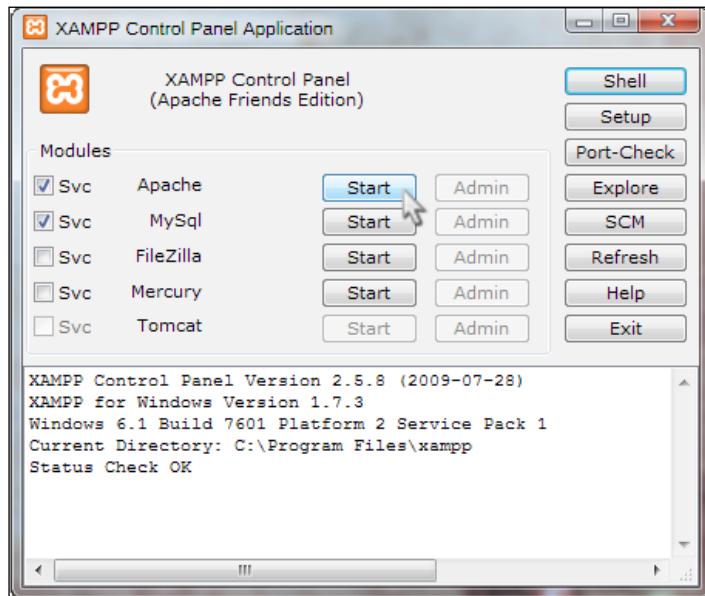
If you're running Microsoft Windows, you may be running Microsoft's **Internet Information Server (IIS)**. You can find out by going into your **Control Panel** and choosing either of the following options:

- **Program Features | Turn Windows features on or off** (in Vista or Windows 7). Detailed instructions are at <http://www.howtogeek.com/howto/windows-vista/how-to-install-iis-on-windows-vista/>.
- **Add/Remove Programs | Add/Remove Windows Components** (in Windows XP). Detailed instructions are at <http://www.webwiz.co.uk/kb/asp-tutorials/installing-iis-winXP-pro.htm>.

If you do not have IIS installed or are unfamiliar with its operation, it is recommended to install the Apache server for use with this book. This will allow us to provide consistent instructions for both Mac and PC in our examples.

One of the easiest ways to install Apache is to download and install the XAMPP software package (<http://www.apachefriends.org/en/xampp-windows.html>). This package includes Apache as well as PHP and MySQL. These additional programs can be helpful as your skills grow, allowing you to create more complex programs and data storage options.

After you've downloaded and run XAMPP, you'll be prompted to run the XAMPP control panel. You can also run the XAMPP control panel from the Windows Start menu. You should click on **Start** on the **Apache** line of the control panel to start your web server. If you receive a notice from your firewall software, you should choose the option which will allow Apache to connect to the Internet.



Inside the folder in which you've installed XAMPP, is a subdirectory called `htdocs`. This is the web folder where we will be setting up Sencha Touch. The full path is usually `C:\xampp\htdocs`. Your web server URL will be `http://localhost/`; you'll want to remember this for the next step.

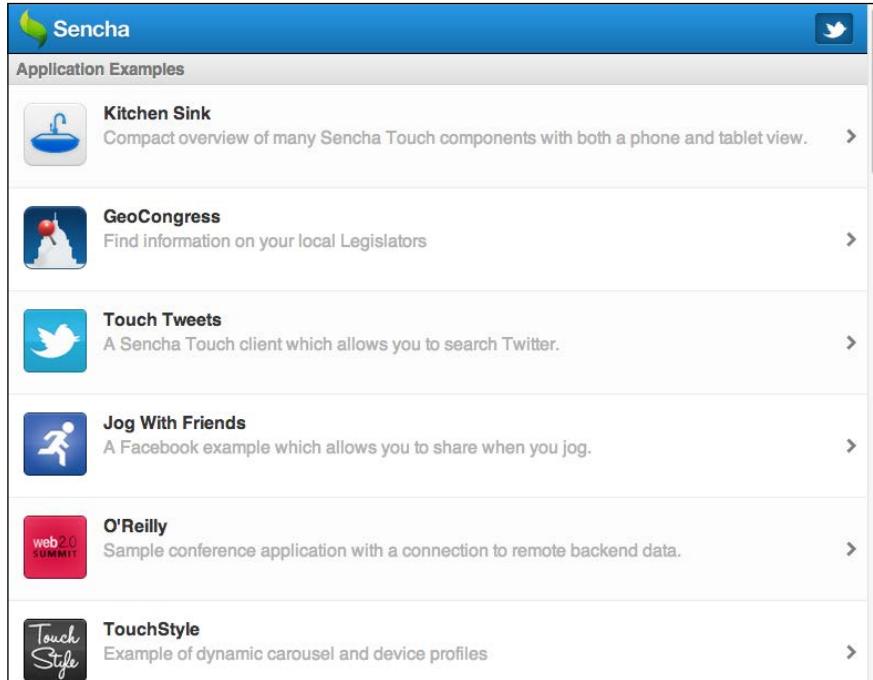
## Download and install the Sencha Touch framework

In your web browser, go to `http://www.sencha.com/products/touch/` and click on the **Download** button. Save the ZIP file to a temporary directory.

[  Note that all examples in this book were written using Sencha Touch Version 2.1.1. ]

Unzipping the file you've downloaded will create a directory called `sencha-touch-version` (in our case, it was `sencha-touch-2.1.1`). Copy this directory to your web folder and rename it, dropping the version number and leaving just `sencha-touch`.

Now, open up your web browser and enter your web URL, adding `sencha-touch/examples` at its end. You should see the following Sencha Touch demo page:



Congratulations! You've successfully installed Sencha Touch.

This demo page contains examples for applications as well as simple examples of components.

## Additional tools for developing with Sencha Touch

In addition to configuring a web server and installing the Sencha Touch libraries, there are some development tools that you may want to take a look at before diving into your first Sencha Touch application. Sencha also has several other products you may find useful to use in your Sencha Touch app, and there are quite a few third-party tools that can help you develop and deploy your app. We're not going to go into a lot of detail about how to set them up and use them, but these tools are definitely worth looking into.

## Safari and Chrome Developer Tools

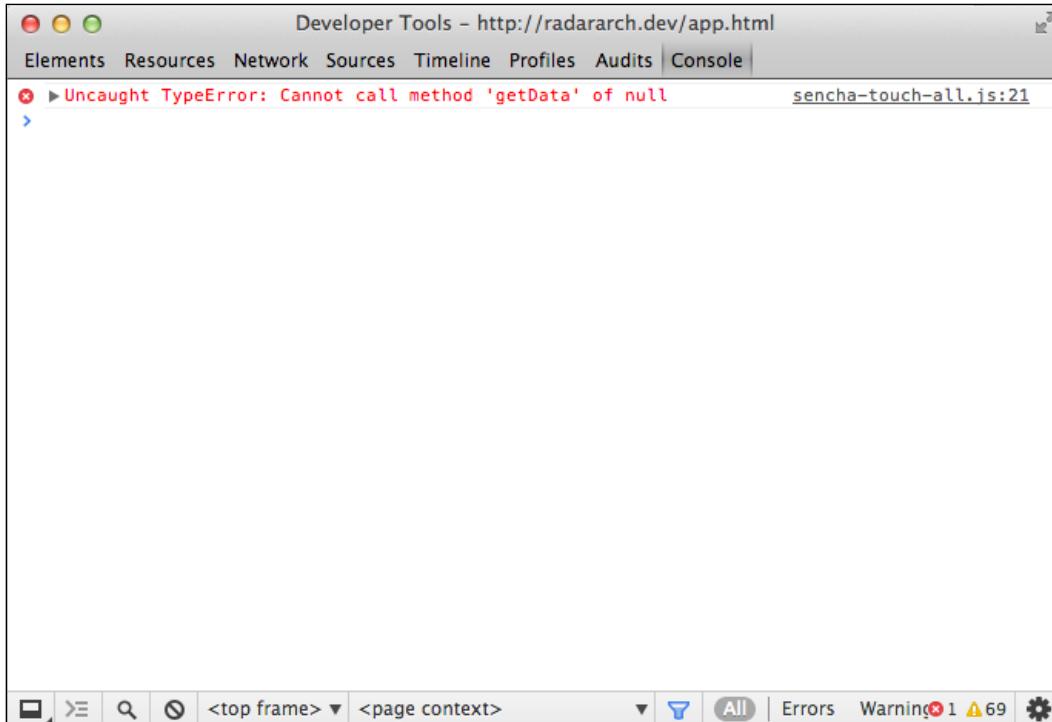
When writing code, it is often very helpful to be able to see what is going on behind the scenes. The most critical tools for working with Sencha Touch are the Safari and Chrome Developer Tools. These tools will help you debug your code in a number of different ways, and we will cover them in a bit more detail as we move further through the book. For now, let's take a quick look at the four basic tools, explained in the following sections.



For Safari users, you can enable the Safari developer menu by going to **Edit | Preferences | Advanced**. Check the **Show Develop** checkbox in the menu bar. Once this menu is enabled, you can see all of the available developer tools. For Chrome users, these tools can be accessed from the **View | Developer | Developer Tools** menu.

## JavaScript Console

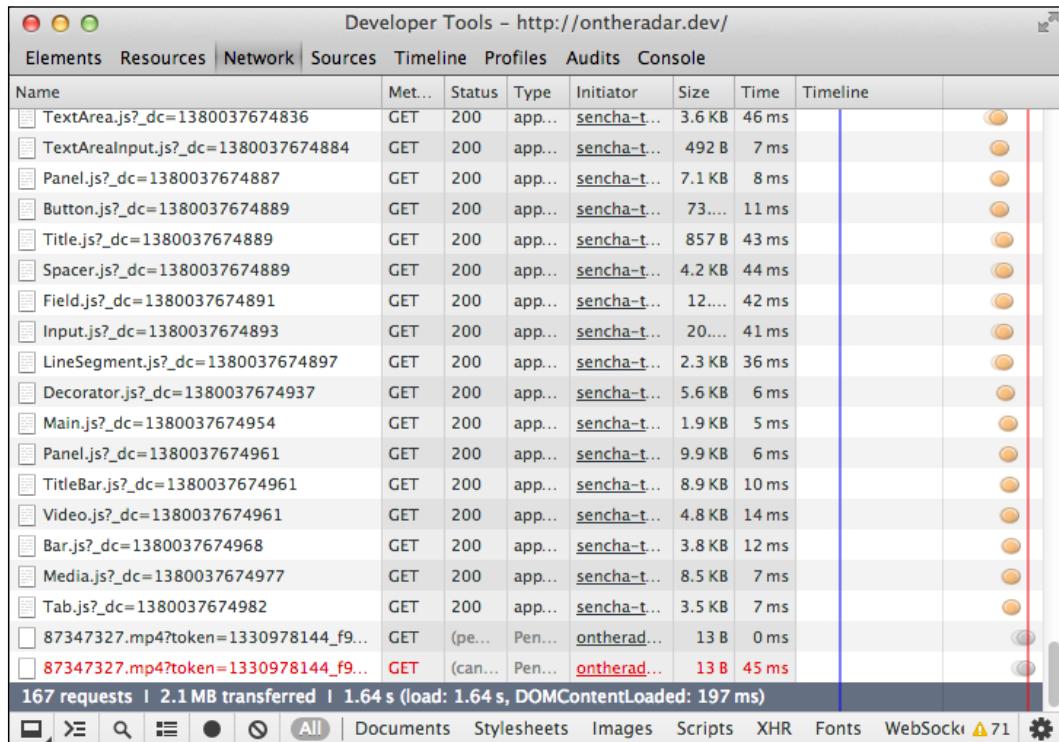
The JavaScript Console displays errors and console logs, which provide you with an indication when things go wrong.



Notice that we get two pieces of information here: the error and the file where the error occurred. You can click on the filename to see the exact line where the error occurred. You should take some time to familiarize yourself with the consoles under either Chrome or Safari. You will likely be spending a lot of time here.

## The Network tab

The second helpful tool is the **Network** tab. This tab shows you all the files that are loaded in the web browser, including an error for any file the browser attempted to load but could not find.



The screenshot shows the Network tab of the Google Chrome Developer Tools. The table lists 167 requests made to the URL `http://ontheradar.dev/`. The columns provide details such as the request name, method (e.g., GET), status code, type (e.g., app...), initiator (e.g., `sencha-t...`), size, time, and a timeline visualization. Two requests are highlighted in red: `87347327.mp4?token=1330978144_f9...` and `87347327.mp4?token=1330978144_f9...`, indicating they failed to load. The timeline shows a vertical blue bar for successful requests and a red bar for errors.

Name	Method	Status	Type	Initiator	Size	Time	Timeline
TextArea.js?_dc=1380037674836	GET	200	app...	<code>sencha-t...</code>	3.6 KB	46 ms	
TextAreaInput.js?_dc=1380037674884	GET	200	app...	<code>sencha-t...</code>	492 B	7 ms	
Panel.js?_dc=1380037674887	GET	200	app...	<code>sencha-t...</code>	7.1 KB	8 ms	
Button.js?_dc=1380037674889	GET	200	app...	<code>sencha-t...</code>	73....	11 ms	
Title.js?_dc=1380037674889	GET	200	app...	<code>sencha-t...</code>	857 B	43 ms	
Spacer.js?_dc=1380037674889	GET	200	app...	<code>sencha-t...</code>	4.2 KB	44 ms	
Field.js?_dc=1380037674891	GET	200	app...	<code>sencha-t...</code>	12....	42 ms	
Input.js?_dc=1380037674893	GET	200	app...	<code>sencha-t...</code>	20....	41 ms	
LineSegment.js?_dc=1380037674897	GET	200	app...	<code>sencha-t...</code>	2.3 KB	36 ms	
Decorator.js?_dc=1380037674937	GET	200	app...	<code>sencha-t...</code>	5.6 KB	6 ms	
Main.js?_dc=1380037674954	GET	200	app...	<code>sencha-t...</code>	1.9 KB	5 ms	
Panel.js?_dc=1380037674961	GET	200	app...	<code>sencha-t...</code>	9.9 KB	6 ms	
TitleBar.js?_dc=1380037674961	GET	200	app...	<code>sencha-t...</code>	8.9 KB	10 ms	
Video.js?_dc=1380037674961	GET	200	app...	<code>sencha-t...</code>	4.8 KB	14 ms	
Bar.js?_dc=1380037674968	GET	200	app...	<code>sencha-t...</code>	3.8 KB	12 ms	
Media.js?_dc=1380037674977	GET	200	app...	<code>sencha-t...</code>	8.5 KB	7 ms	
Tab.js?_dc=1380037674982	GET	200	app...	<code>sencha-t...</code>	3.5 KB	7 ms	
87347327.mp4?token=1330978144_f9...	GET	(pe...	Pen...	<code>ontherad...</code>	13 B	0 ms	
87347327.mp4?token=1330978144_f9...	GET	(can...	Pen...	<code>ontherad...</code>	13 B	45 ms	

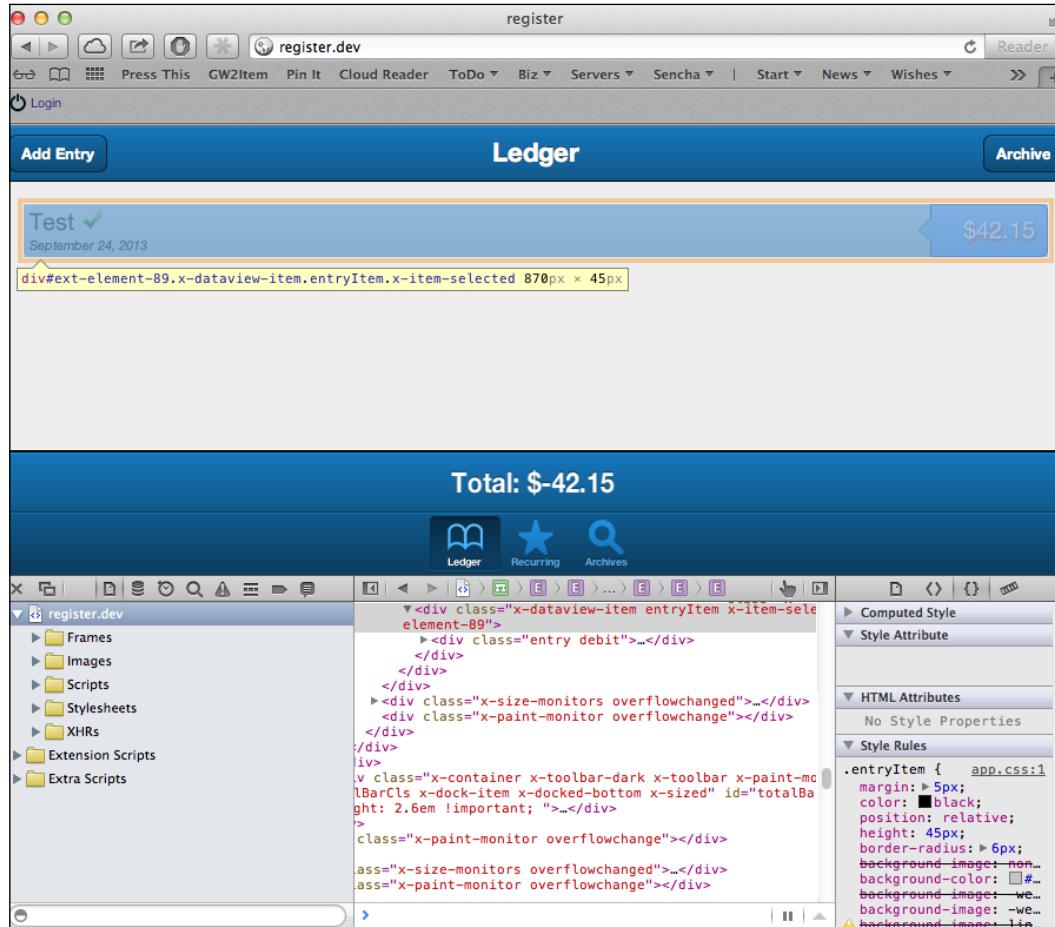
167 requests | 2.1 MB transferred | 1.64 s (load: 1.64 s, DOMContentLoaded: 197 ms)

Documents Stylesheets Images Scripts XHR Fonts WebSockets 71

Missing files are shown in red. Clicking on a file will show more details, including any data that was passed to the file and any data that was returned.

## The web inspector

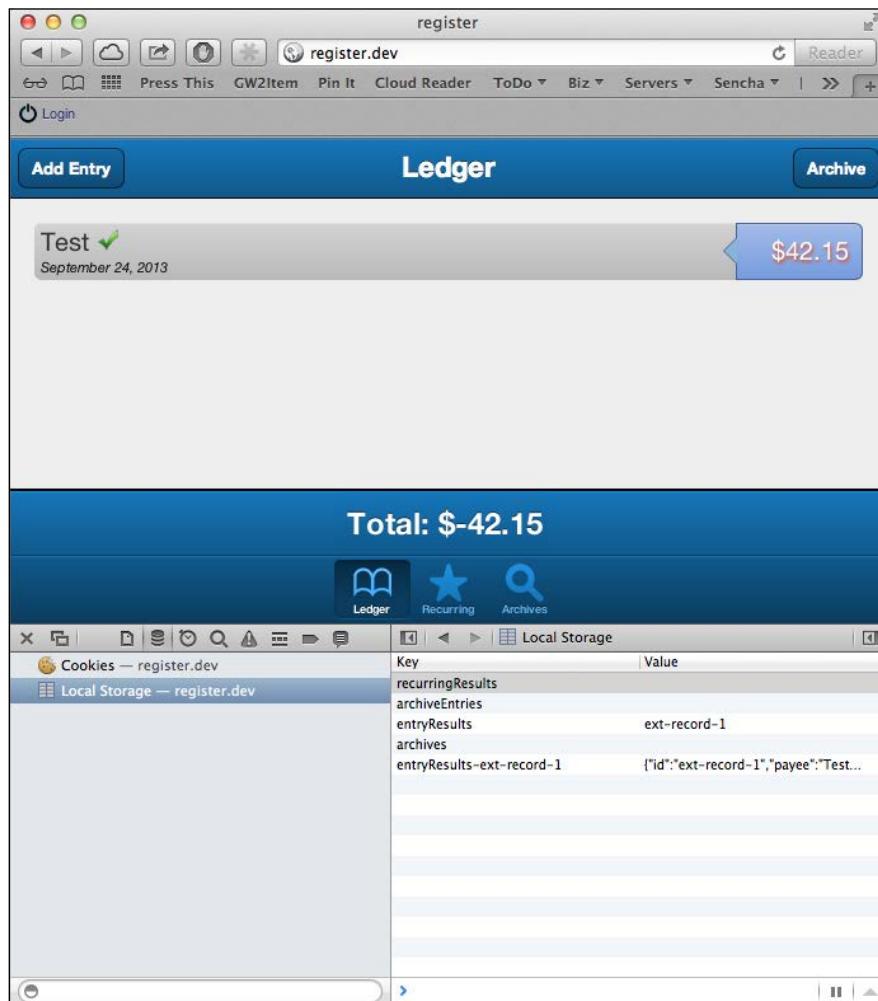
The web inspector allows you to examine the underlying HTML and CSS for any element displayed on the page. The following screenshot shows the web inspector:



The web inspector can be particularly helpful when trying to hunt down display and positioning errors in your application. You can click on the magnifying glass in Chrome or the pointer in Safari to select any element on the page and see the display logic.

## The Resources tab

The **Resources** tab shows information that the browser is storing for our application. This includes information about any data we are storing locally and any cookies we might have created for this application as shown in the following screenshot:



From this tab, you can double-click on an item to edit it or right-click on it to delete it.

We will take a closer look at these tools as we proceed through the book and show you some additional uses and tips.

A full discussion of the Safari Developers Tools can be found at  
<https://developer.apple.com/technologies/safari/developer-tools.html>.

The Chrome Developers Tools are covered at  
<https://developers.google.com/chrome-developer-tools/>.

## **Other Sencha products**

Sencha offers several products that can speed up code development and even expand the capabilities of Sencha Touch.

### **Sencha Cmd**

Sencha Cmd is a command-line tool that allows you to generate the basic Sencha Touch files from the command-line prompt. It also lets you compile applications for the Web and as a binary file that you can sell on the various app stores.

We will be using Sencha Cmd in several instances in this book. You can download it from the following website:

<http://www.sencha.com/products/sencha-cmd/download>

### **Sencha Architect**

Sencha Architect is the **Integrated Development Environment (IDE)** for Sencha Touch and ExtJS applications. Sencha Architect allows you to build your application in a graphical environment, dragging and dropping controls on the screen. You can arrange and manipulate these components in a number of ways and Sencha Architect will write the underlying code for you. You can download it from the following website:

<http://www.sencha.com/products/architect>

### **Sencha Animator**

Sencha Touch comes with a few built-in animations; but, for more intensive animations, a more robust application is needed. With the Sencha Animator desktop application, you can create professional animations that rival Flash-based animations. However, unlike Flash-based animations, Sencha Animator animations run on most mobile browsers, making them perfect for adding extra flair to your Sencha Touch application. You can download Sencha Animator at the following website:

<http://www.sencha.com/products/animator/>

## Third-party developer tools

You can also choose from a variety of developer tools that you may find useful in developing your Sencha Touch apps.

### **NotePad++**

NotePad++ is a code editor and is very useful for writing JavaScript code. It has certain useful features, such as syntax highlighting, syntax folding, multiview and multilanguage environments, and multiple documents. This is a free and open source tool available at <http://notepad-plus-plus.org/features.html>. This is available only for Windows and Linux operating systems.

### **WebStorm**

WebStorm is an IDE (a code editor) for developing web applications in languages such as JavaScript and many more. WebStorm is available for Windows, OS X, and Linux. Webstorm is available as a 30-day free trial with licensing options for commercial, personal, and educational usage. You can find it at the following website:

<http://www.jetbrains.com/webstorm/>

### **Xcode 5**

Xcode 5 is Apple's complete development environment, designed for people writing for any Apple platform—OS X, iPhone, or iPad. As such, it comes with a lot of stuff that is not really necessary for writing Sencha Touch applications. However, one thing that is included with Xcode 5 that can be very handy for Sencha Touch developers is the iOS Simulator. With the iOS Simulator, you can test your application on various iOS devices without having to actually own them.

Most of the uses of Xcode 5 require membership to the Apple Developer program (for things like selling applications on the app store). However, the iOS Simulator can be used by anyone. You can download Xcode 5 from the following website:

<http://developer.apple.com/xcode/>

## Android Emulator

Android Emulator is the Android counterpart to the iOS Simulator that comes with Xcode 5. Android Emulator is part of the free Android SDK download at <http://developer.android.com/guide/developing/devices/emulator.html>. Android Emulator can be configured to mimic many specific Android mobile devices, allowing you to test your application across a broad range of devices.

## YUI test

A common part of any kind of programming is testing. A YUI test, part of Yahoo's YUI JavaScript library, allows you to create and automate unit tests, just as JUnit does for Java. Unit tests set up test cases for specific segments of code. Then, if in the future that code changes, the unit tests can be re-run to determine whether or not the code still succeeds. This is very useful, not only for finding errors in code, but also for ensuring code quality before a release. YUI tests can be found at the following website:

<http://yuilibrary.com/yui/docs/test/>

## Jasmine

Jasmine is a testing framework similar to YUI tests, except it's based on **Behavioral Driven Design (BDD)**. In BDD testing, you start with specifications—stories about what your application should do in certain scenarios—and then write code that fits those specifications. Both YUI tests and Jasmine accomplish the same goals of testing your code, they just do it in different ways. You can download Jasmine at the following website:

<http://pivotal.github.com/jasmine/>

## JSLint

Possibly the most useful JavaScript tool on this list, **JSLint** will examine your code for syntax errors and code quality. Written by Douglas Crockford, one of the fathers of JavaScript, JSZip will examine your code in great detail, which is great for finding errors before you deploy your code. You can find more information at the following website:

<http://www.jslint.com/lint.html>

## Summary

In this chapter, we've covered the fundamentals of web application frameworks and why you should use Sencha Touch. We've walked through setting up a development environment and installing the Sencha Touch libraries. We also took a brief look at what the limitations of mobile devices are and how you can overcome them. We also took a brief look at what the things are that we should keep in mind while developing mobile applications. We've also explored the additional tools that are useful for mobile application development:

- Sencha Touch Learning Center (<http://www.sencha.com/learn/touch/>)
- Apple's iOS Human Interface Guidelines (<http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html>)—an in depth guide to developing user interfaces for iOS devices.
- Android Interface guidelines ([http://developer.android.com/guide/practices/ui\\_guidelines/index.html](http://developer.android.com/guide/practices/ui_guidelines/index.html))

In the next chapter, we'll create our first Sencha Touch application and, in the process, learn the basics of using Sencha Touch development and the MVC framework.



# 2

## Creating a Simple Application

In this chapter, we will go through the basics of creating a simple application in Sencha Touch. We will cover the basic elements that are used in most Sencha Touch applications and take a look at the more common components you might use in your own applications: containers, panels, lists, toolbars, and buttons.

In this chapter, we will cover the following topics:

- Creating a basic application with Sencha Cmd
- Understanding the application's files and folders
- Modifying the application
- Controlling the application's layout
- Testing and debugging the application
- Updating the application for production

Let's learn how to set up a basic Sencha Touch application.

## Setting up the application

Before we get started, you need to be sure that you've set up your development environment properly as per the outline in the previous chapter.

### The root folder

As noted in the previous chapter, you will need to have files and folders for your application located in the correct folder on your local machine in order to allow the web server to locate them.

 On a Mac machine, this will be the `Sites` folder under your home folder if you are using web sharing. If you are using MAMP, the location is `/Applications/MAMP/htdocs`.

On Windows, this will be `C:\xampp\htdocs` (assuming you installed XAMPP as described in the previous chapter).

Through out the rest of this book, we will refer to this folder as the root folder.

In previous versions of Sencha Touch, you had to set up your directory structure manually. In an effort to make this a bit easier and more consistent, Sencha now recommends the use of Sencha Cmd to create the initial application structure.

## Getting started with Sencha Cmd

As mentioned in the previous chapter, Sencha Cmd is a command-line tool that allows you to generate a number of basic Sencha Touch files from the command line.

You will first need to download a copy of Sencha Cmd from:

<http://www.sencha.com/products/sencha-cmd/download>

On Windows or Mac, you can run the installer after it downloads and then follow the prompts for installing Sencha Cmd.

Once you have installed Sencha Cmd, you can open the command-line prompt on your computer in the following manner:

- On Mac OS X, go to Applications/Utilities and launch **Terminal**
- On Windows, go to **Start | Run** and type `cmd`

Once the command line is available, type `sencha` and you should see something similar to this:

```

12ftguru: ~: sencha
Sencha Cmd v3.1.2.342
Sencha Cmd provides several categories of commands and some global switches. In
most cases, the first step is to generate an application based on a Sencha SDK
such as Ext JS or Sencha Touch:

    sencha -sdk /path/to/sdk generate app MyApp /path/to/myapp

Sencha Cmd supports Ext JS 4.1.1a and higher and Sencha Touch 2.1 and higher.

To get help on commands use the help command:

    sencha help generate app

For more information on using Sencha Cmd, consult the guides found here:

http://docs.sencha.com/ext-js/4-2/#!/guide/command
http://docs.sencha.com/ext-js/4-1/#!/guide/command

http://docs.sencha.com/touch/2-2/#!/guide/command
http://docs.sencha.com/touch/2-1/#!/guide/command

Options
* --cwd, -cw - Sets the directory from which commands should execute
* --debug, -d - Sets log level to higher verbosity
* --nologo, -n - Suppress the initial Sencha Cmd version display
* --plain, -pl - enables plain logging output (no highlighting)
* --quiet, -q - Sets log level to warnings and errors only
* --sdk-path, -s - The location of the SDK to use for non-app commands
* --time, -ti - Display the execution time after executing all commands

```

This tells you that the command was successful and provides you with some of the basic help options for Sencha Cmd. In fact, we will be using the first command listed in this Help section to generate our new application:

```
sencha -sdk /path/to/sdk generate app MyApp /path/to/myapp
```

There are seven pieces to this command, so let's take a look at them one by one:

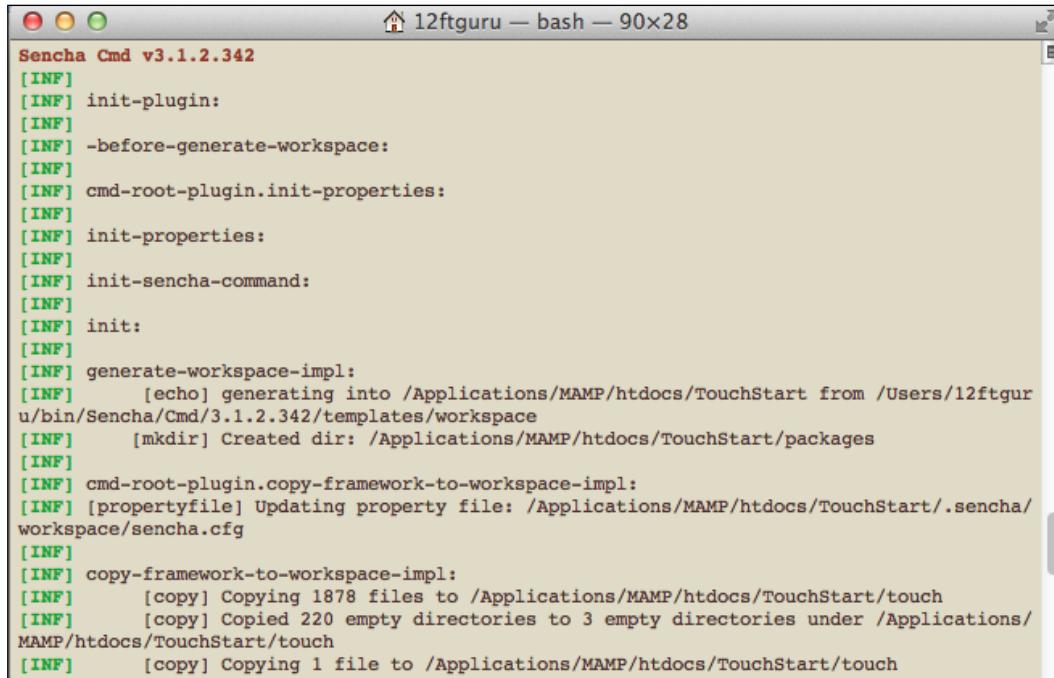
- `sencha`: This tells the command line the name of the application that will process the command; in this case, `Sencha Cmd`, or just `sencha` for short.
- `-sdk`: This tells Sencha Cmd that we will specify the path to our Sencha Touch libraries. We can also just change the directory to the folder where we downloaded these libraries and eliminate the `-sdk` part and the path information that follows.
- `/path/to/sdk`: This will be replaced with the actual path to the Sencha Touch library files we downloaded (not Sencha Cmd, but the actual Sencha Touch library).
- `generate`: This indicates what we are going to do.

- **app:** Since we are generating something, what will we be generating? This part of the command answers this question. In this case, we will be generating an application.
- **MyApp:** This will be the name of your application. It will also be used for the JavaScript namespace, which we will cover a bit later. This is arbitrary, but it must be a single word with no spaces.
- **/path/to/myapp:** This will be the path to your new application. This path should be in a new folder in the root folder we talked about earlier.

For this chapter, we are going to create an application called TouchStart. Your path information will need to reflect your own personal setup, but the command should look something similar to this:

```
sencha -sdk /Users/12ftguru/Downloads/touch-2.2.1 generate app TouchStart  
/Applications/MAMP/htdocs/TouchStart
```

Adjust your paths to match the location of your Sencha Touch libraries and your root folder. Once the command is executed, you will see a number of messages appearing in your terminal in the following manner:



The screenshot shows a terminal window titled "12ftguru — bash — 90x28". The window contains the output of the Sencha Cmd command. The output is color-coded with green for informational messages and blue for command names. The process starts with the Sencha Cmd version (v3.1.2.342), followed by a series of "[INFO]" messages related to plugin initialization and workspace generation. It then creates a directory for the application packages. Finally, it copies files from the template directory to the application's touch directory, including 1878 files and 220 empty directories.

```
Sencha Cmd v3.1.2.342
[INFO]
[INFO] init-plugin:
[INFO]
[INFO] -before-generate-workspace:
[INFO]
[INFO] cmd-root-plugin.init-properties:
[INFO]
[INFO] init-properties:
[INFO]
[INFO] init-sencha-command:
[INFO]
[INFO] init:
[INFO]
[INFO] generate-workspace-impl:
[INFO]     [echo] generating into /Applications/MAMP/htdocs/TouchStart from /Users/12ftguru/bin/Sencha/Cmd/3.1.2.342/templates/workspace
[INFO]     [mkdir] Created dir: /Applications/MAMP/htdocs/TouchStart/packages
[INFO]
[INFO] cmd-root-plugin.copy-framework-to-workspace-impl:
[INFO] [propertyfile] Updating property file: /Applications/MAMP/htdocs/TouchStart/.sencha/workspace/sencha.cfg
[INFO]
[INFO] copy-framework-to-workspace-impl:
[INFO]     [copy] Copying 1878 files to /Applications/MAMP/htdocs/TouchStart/touch
[INFO]     [copy] Copied 220 empty directories to 3 empty directories under /Applications/MAMP/htdocs/TouchStart/touch
[INFO]     [copy] Copying 1 file to /Applications/MAMP/htdocs/TouchStart/touch
```

Sencha Cmd copies the files that it needs and sets up your application. Once the command has been executed, you should have a new folder in your root folder called TouchStart.

Open that folder and you will see the following files and directories:



We will be working almost exclusively with the files in the `app` folder, but it's worth covering a little bit about each of these files and directories:

- `app`: This is where all of our application files reside; we will cover this in detail throughout this chapter.
- `app.js`: This is the JavaScript file that sets up our application and handles the initial launch of the application. We will take a closer look at this in the next section.
- `build.xml`: This is a configuration file for building compiled applications. You may not need to change this file.
- `index.html`: This file is much like the `index.html` file for any website. It is the first file that is loaded by the web browser. However, unlike traditional websites, the `index.html` file of our application only loads our initial JavaScript and doesn't do anything else. You shouldn't need to change this file.
- `packager.json`: This is a configuration file that tells our application how the files are set up and where they are located. For the most part, you may not need to change this file.
- `packages`: The `packages` directory is a placeholder where you can install additional packages for your application. It is largely unused at this juncture.
- `resources`: The `resources` directory contains our CSS files and start up screens and icons. We will learn more about this directory in the next chapter on styling.
- `touch`: This directory contains a copy of the Sencha Touch library files. It should never be modified.

We can also view our new application in a web browser by going to our web directory. This would be `http://localhost/TouchStart` for Windows and MAMP or `http://localhost/~username/TouchStart` for Mac users with web sharing enabled.



[ It is also worth noting that Sencha Cmd itself has a built-in web server that you can use to view your Sencha Touch applications. You can start the Sencha Cmd web server using the following command:

`sencha fs web -port 8000 start -map /path/to/your/appfolder`

You can then open your web browser by going to `http://localhost:8000`.

For more information on using the Sencha Cmd web server, visit <http://docs.sencha.com/cmd/3.1.2/#!/guide/command>.



As we can see from the basic application that has been created, we are looking at the contents of a file called `Main.js` located at `app/view`. We can make changes to this file and see the results when we reload the page.

Before we start tinkering around with the `Main.js` file, we need to take a look at the file that loads everything up for us, namely `app.js`.

## Creating the `app.js` file

The `app.js` file is responsible for setting up our application, and though we don't need to modify it very often, it's a good idea to get a feel of what it does and why.

Open your `app.js` file in your code editor; at the top, you will see a long block of comments (which you should read and familiarize yourself with). Underneath the comments, the code begins with:

```
Ext.Loader.setPath({
    'Ext': 'touch/src'
});
```

This tells the application where our Sencha Touch library files are located.

Next, we define our application with the following code:

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

    requires: [
        'Ext.MessageBox'
    ],

    views: [
        'Main'
    ],

    icon: {
        '57': 'resources/icons/Icon.png',
        '72': 'resources/icons/Icon~ipad.png',
        '114': 'resources/icons/Icon@2x.png',
        '144': 'resources/icons/Icon~ipad@2x.png'
    },
    isIconPrecomposed: true,

    startupImage: {
        '320x460': 'resources/startup/320x460.jpg',
        '640x920': 'resources/startup/640x920.png',
        '768x1004': 'resources/startup/768x1004.png',
        '748x1024': 'resources/startup/748x1024.png',
        '1536x2008': 'resources/startup/1536x2008.png',
        '1496x2048': 'resources/startup/1496x2048.png'
    },
    launch: function() {
        // Destroy the #appLoadingIndicator element
        Ext.fly('appLoadingIndicator').destroy();

        // Initialize the main view
        Ext.Viewport.add(Ext.create('TouchStart.view.Main'));
    }
});
```

Now that is quite a bit of code for one big bite, so let's understand one piece at a time.

The first part, `Ext.Application({...});`, creates a new application for Sencha Touch. Everything listed between the curly braces is a configuration option for this new application. While there are a number of configuration options for an application, most consist of at least the application's name and launch function.

### Namespaces

One of the biggest problems with using someone else's code is the issue of naming. For example, if the framework you are using has an object called `Application`, and if you create your own object called `Application`, the two functions will conflict. Sencha Touch uses the concept of namespaces to keep these conflicts from happening.

In this case, Sencha Touch uses the namespace `Ext`. You will see this namespace being used throughout the code in this book. It is simply a way to eliminate potential conflicts between the frameworks' objects and code and your own objects and code.

Sencha will automatically set up a namespace for your own code as part of the new `Ext.Application` object. In this case, it will be `TouchStart`, which we used to generate our application.

`Ext` is also part of the name of Sencha's web application framework called `ExtJS`. Sencha Touch uses the same namespace convention to allow developers to familiarize with one library and easily understand the other.



When we create a new application, we need to pass it some configuration options. This will tell the application how to look and what to do. These configuration options are contained within curly braces `{}` and separated by commas. The first option is:

```
name: 'TouchStart'
```

This sets the name of our application to whatever is between the quotes. The `name` value should not contain spaces as Sencha also uses this value to create a namespace for your own code objects. In this case, we have called the application `TouchStart`.

After the `name` option, we have a `requires` option:

```
requires: [
    'Ext.MessageBox'
]
```

This is where we list any files that are required as soon as the application launches. Since we actually use the `Ext.Msg.confirm` function towards the bottom of this file, we have to include the `Ext.MessageBox` class here.

Next, we have the `views` section:

```
views: [
    'Main'
]
```

This section serves as a reference to our `Main.js` file in the `app/view` folder. We can also have listings for controllers, stores, or models here, but right now, the `Main.js` view file is the only one we have as part of this skeleton app. We will learn more about controllers, models, stores, and views in later chapters.

The `icon` and `startupImage` sections provide links to the image files used for the application's icon and startup screens. The various sizes listed ensure that the application's images display correctly across multiple devices.

The next option is where things start to get interesting:

```
launch: function() {
    // Destroy the #appLoadingIndicator element
    Ext.fly('appLoadingIndicator').destroy();

    // Initialize the main view
    Ext.Viewport.add(Ext.create('TouchStart.view.Main'));
}
```

The `launch` function executes once the required JavaScript files (as well as any views, models, stores, and controllers that are listed) have been loaded. This function first destroys our loading indicator (since we are done loading files). It then creates our main view and adds it to the viewport. This viewport is where we will add things that we want to display to the user.

In this case, `TouchStart.view.Main` refers to our `Main.js` file in the `app/view` folder. This is how Sencha Touch knows how to find files:

- `TouchStart` is our application
- `view` is the `views` folder
- `Main` is our `Main.js` file

Let's take a closer look at this `Main.js` file and see how it creates all of the visual pieces we currently see in our skeleton application.

## Creating the Main.js file

The `Main.js` file is located in the `app/view` folder. The `view` files are the visual components of our application. Let's open this file and see how a simple tab panel is created.

```
Ext.define('TouchStart.view.Main', {
    extend: 'Ext.tab.Panel',
    xtype: 'main',
    requires: [
        'Ext.TitleBar',
        'Ext.Video'
    ],
    config: {
        tabBarPosition: 'bottom',

        items: [
            ...
        ]
    }
});
```

We've removed the contents of the `items` section from our code example to make this a bit easier to read through.

The first two lines of the preceding code are common to pretty much every component you will create in Sencha Touch.

```
Ext.define('TouchStart.view.Main', {
    extend: 'Ext.tab.Panel'
```

The first line defines the full name of your component in the form:

- App name (namespace)
- Folder name
- Filename (no extension)

Next, we list the component we are extending; in this case, a tab panel. You will see this `define/extend` pattern throughout this book.

You will also notice that the tab panel is called `Ext.tab.Panel`. This lets Sencha Touch know that the component is a native component (`Ext`) located in a folder named `tab` in a file called `Panel.js`. This pattern allows Sencha Touch to load the correct file and extend it using our new configuration options:

```
 xtype: 'main',
 requires: [
    'Ext.TitleBar',
    'Ext.Video'
 ],
 config: {
    tabBarPosition: 'bottom'
```

The first thing we do is set an `xtype` value for our new component. The `xtype` part is a short name that allows us to easily reference and create copies of our component without having to use the full name. You will see some examples of this later on in this book.

Our skeleton application uses a `TitleBar` and a `Video` component, so we need these two files.

Next, we set up a `config` section. This is where we set any custom settings for our new component. In this case, we position our tab bar at the bottom.

Now we want to take a look at the `items` section that we removed from our code example and see what effect this section has on our tab panel.

## Exploring the tab panel

The `Ext.tab.Panel` is designed to do a few things for us automatically. The most important thing is that for every panel we add in the `items` section, a corresponding tab is created for us in the tab panel. By default, only the first panel is actually shown. However, the tab panel also switches these panels for us automatically when the panel's tab is tapped.

If you look back at our app in the browser, you will also see that each tab has a title and an icon. These two `config` options are set as part of the individual items that currently look similar to this:

```
 items: [
  {
    title: 'Welcome',
    iconCls: 'home',
    styleHtmlContent: true,
    scrollable: true,
    items: {
      docked: 'top',
      xtype: 'titlebar',
      title: 'Welcome to Sencha Touch 2'
    },
  ],
```

```
html: [
    "You've just generated a new Sencha Touch 2 project. What
    you're looking at right now is the ",
    "contents of <a target='_blank' href=\"app/view/Main.
    js\">app/view/Main.js</a> - edit that file ",
    "and refresh to change what's rendered here."
].join(""))

},
{
    title: 'Get Started',
    iconCls: 'action',
    items: [
        {
            docked: 'top',
            xtype: 'titlebar',
            title: 'Getting Started'
        },
        {
            xtype: 'video',
            url: 'http://av.vimeo.com/64284/137/87347327.
            mp4?token=1330978144_f9b698fea38cd408d52a2393240c896c',
            posterUrl: 'http://b.vimeocdn.com/
            ts/261/062/261062119_640.jpg'
        }
    ]
}
]
```

Notice that our `items` list is enclosed in brackets and the individual components within the item list are contained in curly braces. This nested component structure is a key part of Sencha Touch, and you will see it in use throughout this book.

The `title` and the `iconCls` properties control how the tabs appear for each item. Our titles are currently set to `Welcome` and `Getting Started`. Our `iconCls` configuration determines the icon that will be used in the tab. In this case, we are using two of the default icons: `home` and `action`.

Our panel is the `Welcome` panel and it has config options that allow us to use styled HTML content and make it scrollable (if the content is larger than the screen size). The text inside the `html` config option is what we see as the content of our first panel.

You will also notice that our panel has items of its own. In this case, there's a `titlebar` that will be docked at the `top` of our panel by `title Welcome` to `Sencha Touch 2`.

Our second `Get Started` panel has two items in it: a titlebar like our first panel and a video component which lists a URL for the video and a separate `posterUrl` for the image that will appear before the user plays the video.

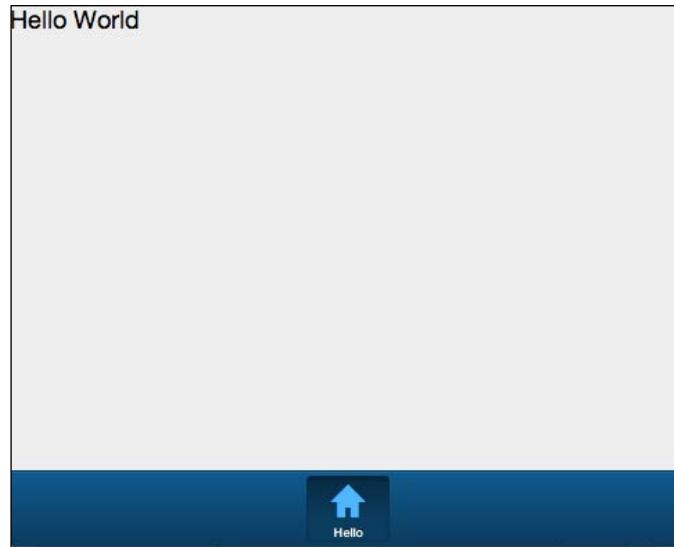
As the text in our first panel notes, we can change the content of this file and see the results when we reload the page. Let's give that a try and see how it works.

## **Adding a panel**

The first thing we want to do is delete everything between the `items` brackets `[ ]` for our tab panel. Next, we will add in a new panel similar to this:

```
items: [
  {
    title: 'Hello',
    iconCls: 'home',
    xtype: 'panel',
    html: 'Hello World'
  }
]
```

If we reload the browser now, we see this:



Since we only have one panel now, we only get one tab. We also got rid of the title bar, so we don't have anything at the top of the page.



Sharp-eyed readers will also notice that we explicitly set an `xtype` value for `panel` in this example. The tab panel automatically assumes that if you don't specify an `xtype` value for one of its items, it's a panel. However, it's a good idea to get in the habit of setting the `xtype` value for any components you use. We'll talk more about `xtypes` in *Chapter 4, Components and Configurations*.

Right now, our panel is very simple and only contains one line of text. In the real world, applications are rarely this simple. We need a way to arrange different elements inside our panel so we can create modern, complex layouts. Fortunately for us, Sencha Touch has a built-in configuration called `layout`, which will help us with this.

## Controlling the look with layouts

Layouts give you a number of options for arranging content inside containers. Sencha Touch offers five basic layouts for containers:

- `fit`: This is a single item layout that automatically expands to take up the whole container
- `hbox`: This arranges items horizontally in the container
- `vbox`: This arranges items vertically in the container
- `card`: This arranges items like a stack of cards, where only the active card is initially visible
- `docked`: This places an item at the top or bottom or to the left or right of the display

In our previous example, we did not declare a layout. In general, you will always want to declare a layout for any container. If you don't, the components inside the container may not size themselves appropriately when they appear.

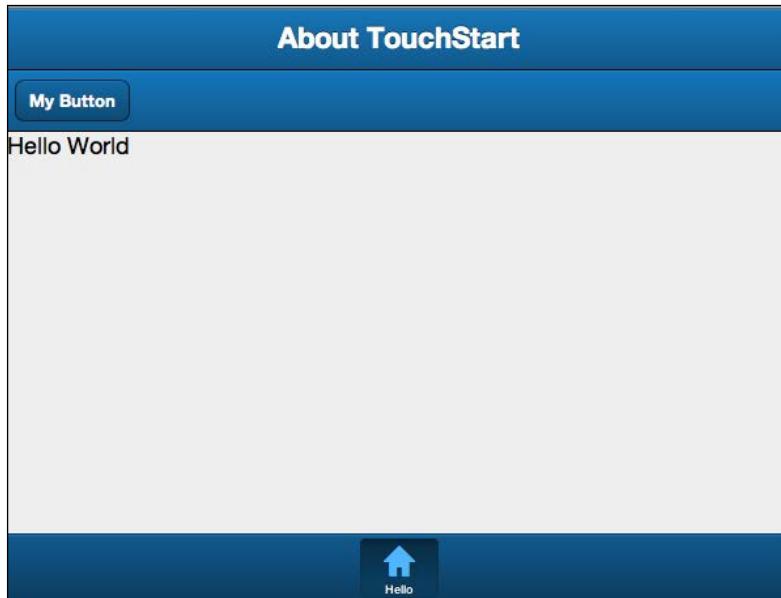
We have seen the last two layouts already. The tab panel uses a `card` layout to switch between the different panels in its `items` list.

The title bars in our original `Main.js` file had a `docked` property as part of their configuration. This configuration docks them to a particular part of the screen. You can even dock multiple items to one of the four sides of a panel.

For example, if we add an `items` section to our current panel with the following:

```
items: [
  {
    xtype: 'titlebar',
    docked: 'top',
    title: 'About TouchStart'
  },
  {
    xtype: 'toolbar',
    docked: 'top',
    items: [
      {
        xtype: 'button',
        text: 'My Button'
      }
    ]
  }
]
```

The two bars will stack one on top of the other in the following manner:

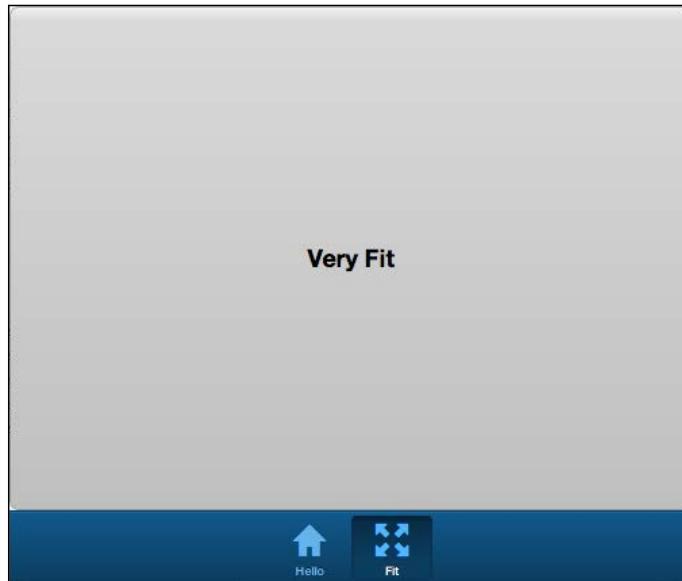


## Using a fit layout

Let's add a second panel to understand what we did earlier. After the closing curly brace from our first panel, add a comma and then the following code:

```
{  
    title: 'Fit',  
    iconCls: 'expand',  
    xtype: 'panel',  
    layout: 'fit',  
    items: [  
        {  
            xtype: 'button',  
            text: 'Very Fit'  
        }  
    ]  
}
```

For this panel, we have added a config option, `layout: 'fit'`, and an `items` section with a single button.



As you can see from the preceding screenshot, this gives us a second tab, which contains our new button. Since the layout is set to fit, the button expands to take up all the available space. While this is useful when you want a component to take up all the available space, it doesn't work very well if you want to nest multiple components.

## Using a vbox layout

The `vbox` layout arranges components in a stack from top to bottom. In this case, multiple components will fill up the available screen space. Let's add another panel to see what this looks like. As before, after the closing curly brace from our last panel, add a comma and then the following code:

```
{  
    title: 'VBox',  
    iconCls: 'info',  
    xtype: 'panel',  
    layout: 'vbox',  
    items: [  
        {  
            xtype: 'container',  
            flex: 2,  
            html: '<div id="hello">Hello World Top</div>',  
            style: 'background:red',  
            border: 1  
        }, {  
            xtype: 'container',  
            flex: 1,  
            html: '<div id="hello">Hello World Bottom</div>',  
            style: 'background:yellow',  
            border: 1  
        }, {  
            xtype: 'container',  
            height: 50,  
            html: '<div id="footer">Footer</div>',  
            style: 'background:green',  
        }  
    ]  
}
```

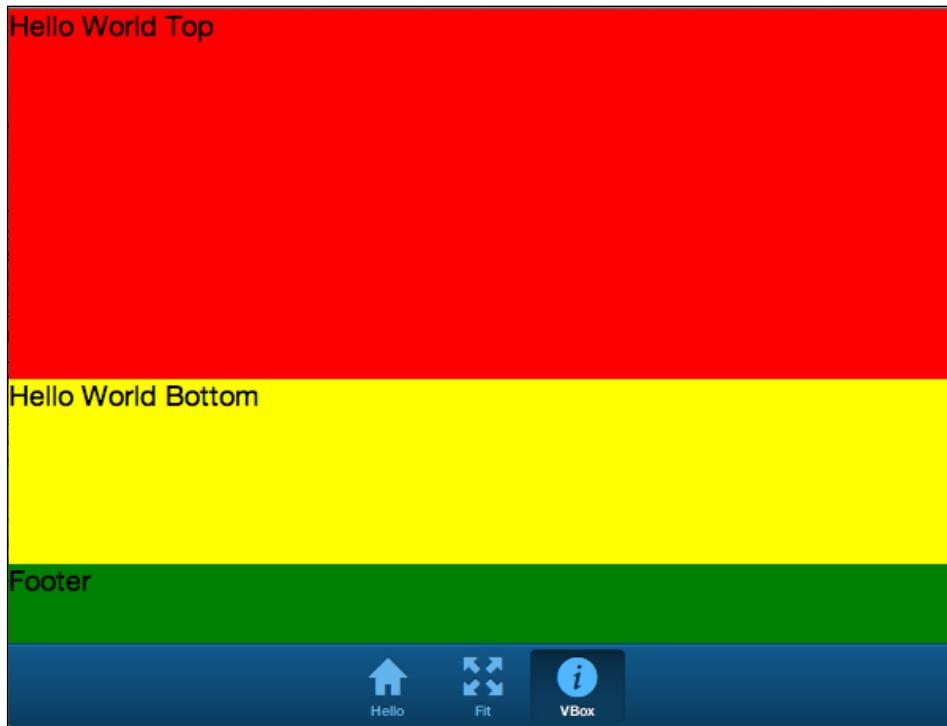
As you can see, this panel has a configuration of `layout: 'vbox'` and a list of three items. The items are a collection of `container` components we want to include inside our panel.

The `container` component is a simplified version of the `panel`, which has no options for elements such as toolbars or title bars.

Our first two containers have a configuration called `flex`. The `flex` configuration is unique to `vbox` and `hbox` layouts (we'll get to `hbox` right after this). The `flex` configuration controls how much space the component will take up proportionally in the overall layout. You may also have noticed that the last container does not have a `flex` configuration. Instead, it has `height: 50`. The `vbox` layout will interpret these values to layout the container as follows:

1. Since we have a component with a height of 50, the `vbox` layout will leave that component's height to 50 pixels.
2. The `vbox` layout will then use the `flex` values of the other two components as a ratio. In this case, 2:1.
3. The end result is a 50-pixel high container at the bottom of the screen. The other two containers will take up the rest of the available space. The top container will also be twice as tall as the middle container.

In order to make these sizes clearer, we have also added a style to each container to color the background and make it stand out a bit. The result is as follows:



This layout will also shrink and expand when the window is resized, making it a very effective layout for various device sizes.

## Using an hbox layout

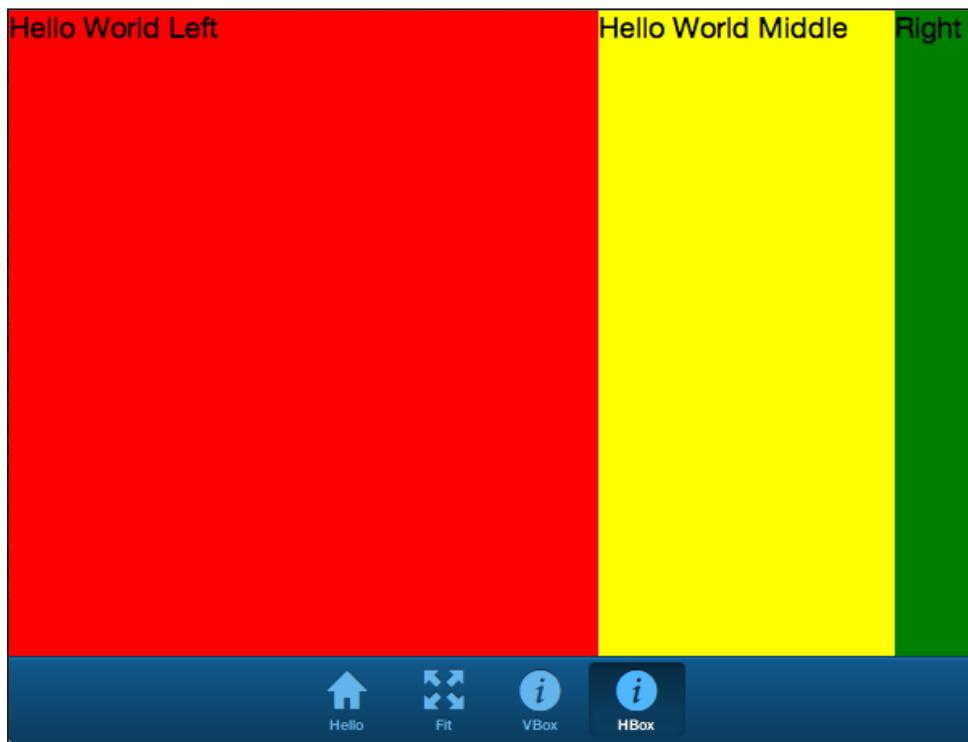
The `hbox` layout works almost in the same way as the `vbox` layout, except that the containers are arranged from left to right in the `hbox` layout.

You can add a panel with an `hbox` layout by copying our previous `vbox` example and pasting it after the last panel in our `items` list (don't forget the comma between the items).

Next we need to modify a few configurations in our new panel:

- Set `title: 'VBox'` to `title: 'HBox'`
- Set `layout: 'vbox'` to `layout: 'hbox'`
- In the last container, set `height: 50` to `width: 50`

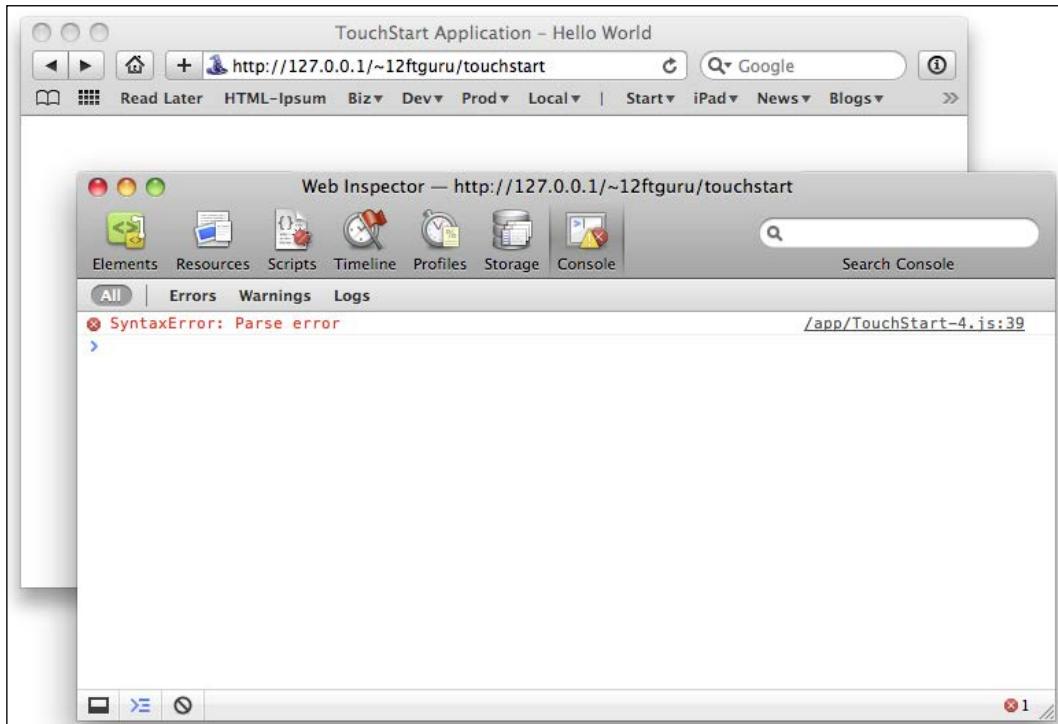
When you reload the page, you should be able to click on the **HBox** tab and see something similar to the following screenshot:



You can nest these basic layouts to arrange your components in any number of ways. We will also cover some ways to style components in *Chapter 3, Styling the User Interface*.

## Testing and debugging the application

The first place to start while testing an application is the error console. In Safari, from the **Develop** menu, select **Show Error Console**. In Chrome, from the **View** menu, choose **Developer** and then **JavaScript Console**.



## Parse errors

The error console in the previous screenshot tells us two very important things. The first is that we have **SyntaxError: Parse error**. This means that somewhere in the code, we did something that the browser didn't understand. Typically, this can be something such as:

- Forgetting to close a parenthesis, bracket, or brace, or adding an extra one
- Not having a comma between the configuration options or adding an extra comma
- Leaving out a semicolon at the end of one of the variable declarations
- Not closing quotes or double quotes (also not escaping quotes where necessary)

The second important bit of information is `/app/TouchStart-4.js: 39`. It tells us that:

- `/app/TouchStart-4.js` is the file where the error occurred
- `39` is the line where the error occurred

Using this information, we should be able to track down the error quickly and fix it.

## Case sensitivity

JavaScript is a case-sensitive language. This means that if you type `xtype: 'Panel'`, you will get the following in the Error Console:

**Attempting to create a component with an xtype that has not been registered: Panel**

This is because Sencha Touch is expecting `panel` and not `Panel`.

## Missing files

Another common problem is missing files. If you don't point your `index.html` file at your `sencha-touch-debug.js` file correctly, you will get two separate errors:

- **Failed to load resource: the server responded with a status of 404 (Not Found)**
- **ReferenceError: Can't find variable: Ext**

The first error is the critical bit of information; the browser could not find one of the files you tried to include. The second error is caused by the missing file and simply complains that the `Ext` variable cannot be found. In this case, it's because the missing file is `sencha-touch-debug.js`, which sets up the `Ext` variable in the first place.

## The web inspector console

Another feature of the Safari web inspector that is incredibly useful for debugging applications is the console. In your `app.js` file, add the following command:

```
console.log('Creating Application');
```

Add it just before this `Ext.Application` line:

```
Ext.Application({
```

You should see the text **Creating Application** in your web inspector's console tab. You can also send variables to the console where you can view their contents:

```
console.log('My viewport: %o', Ext.Viewport);
```

If you place this console log after the line `Ext.Viewport.add(Ext.create('TouchStart.view.Main'));` in `app.js`, the console will display the full viewport and all of its nested child components. This is useful if, for some reason, you have a component that is not displaying properly. Sending an object to the console allows you to see the object as JavaScript sees it.

 For more information about the Chrome Developer Tools, go to <https://developers.google.com/chrome-developer-tools/>.

If you'd like to learn more about using the Safari web inspector for debugging your application, visit Apple's *Debugging your website* page at [http://developer.apple.com/library/safari/#documentation/Applications/Conceptual/Safari\\_Developer\\_Guide/DebuggingYourWebsite/DebuggingYourWebsite.html](http://developer.apple.com/library/safari/#documentation/Applications/Conceptual/Safari_Developer_Guide/DebuggingYourWebsite/DebuggingYourWebsite.html).

## Updating the application for production

When an application is ready for production, there are typically a number of steps to get your code ready and optimized. This process involves compressing the JavaScript to load faster, optimizing images, and getting rid of any parts of the code library that are not actually required by your application. This can be a pretty tedious process, but Sencha Cmd will actually do this for you with a single command.

When you are ready to update your application for production, you can open your command line and move into your code's root directory using the `cd` command:

```
cd /path/to/my/application
```

Once you are in the directory, you can type the following command:

```
sencha app build
```

This command will create a `build` directory with an optimized version of your application inside it. You can test this optimized version for any errors. If you need to make changes to the application, you can make them to the unoptimized code and then run the `build` command again.

Once you are satisfied with the code building, put the application into production.

## Putting the application into production

Now that you've written and tested your application and prepared it for production, we need to figure out where our code is going to live. Since the method for putting an application into production will vary based on your setup, we will be covering this task in very general terms.

The first thing to do is to familiarize yourself with three basic pieces of the puzzle for putting your application into production:

- Web hosting
- File transfer
- Folder structure

While it is fine to develop your application on a local web server, if you want anyone else to see it, you will need a publicly accessible web server with a constant connection to the Internet. There are a number of web hosting providers, such as GoDaddy, HostGator, Blue Host, HostMonster, and RackSpace.

Since our application is pure HTML/JavaScript/CSS, you don't need any fancy add-ons, such as databases or server-side programming languages (PHP or Java), for your web hosting account. Any account that can serve up HTML pages is good enough. The key to this decision should be customer support. Make sure to check the reviews before choosing a provider.

The hosting provider will also supply information on setting up your domain and uploading your files to the web server. Be sure to keep good track of your username and password for future reference.

In order to copy your application to your web hosting account, you'll probably have to familiarize yourself with an **FTP (File Transfer Protocol)** program such as **FileZilla**. As with hosting providers, there is a huge selection of FTP programs. Most of them follow a few basic conventions.

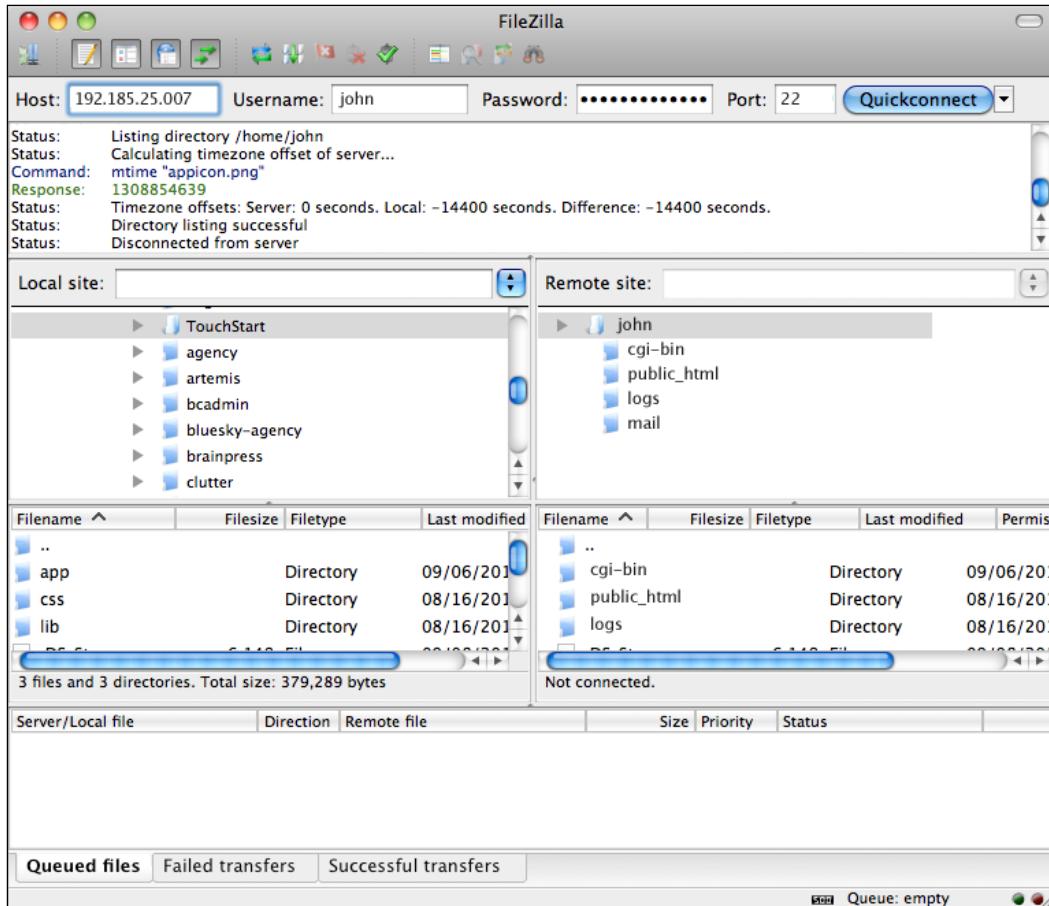
To begin with, you will need to connect to the web server with the FTP program. For this, you will need the following:

- A name or IP address for the web server
- Your web hosting username and password
- A connection port for the web server

## *Creating a Simple Application*

---

Your web hosting provider should provide you with this information when you sign up.



Once you are connected to the server, you will see a list of files on your local machine as well as files on your remote web server. You need to drag the **TouchStart** files on to the remote server to upload them. Your hosting provider will also provide you with the name of a specific folder where these files need to go. The folder is typically called `httpd`, `htdocs`, `html`, or `public_html`.

This brings us to our last consideration for uploading files: the folder path.

The folder path affects how the application locates its files and resources. When you upload the application to the remote web server, it can affect how your folder is seen within the application. If you have any files referenced from an absolute path, such as `http://127.0.0.1/~12ftguru/TouchStart/myfile.js`, then the file will not work when you move things over to the web server.

Even relative URLs can become problematic when you transfer files to the remote server. For example, if you have a file which uses the path `/TouchStart/myFile.js` and you upload the contents of the TouchStart folder instead of uploading the folder itself, the file path will be incorrect.

This is something to keep in mind if you find yourself facing a missing images error or other errors.

Again, your web hosting provider is your best resource for information. Be sure to look for the *Getting Started* document and don't be afraid to seek help from any user forums that your hosting provider may have.

## Summary

In this chapter, we created our first simple application with Sencha Cmd. We learnt some of the basics of Sencha Touch components including configuration and nesting components within one another. We introduced you to the `TabPanel`, `Panel`, and `Container` components. In addition, we explained some basic debugging methodology and prepared our application for production.

In the next chapter, we will create a custom theme for our application by using SASS and the Sencha Touch library's styling tools.



# 3

## Styling the User Interface

Now that we have an understanding of how an application is put together, we are going to take a look at some of the different visual elements you can use to customize your application. In this chapter, we will:

- Take a closer look at toolbars and buttons, using layouts, and additional styles and icons to boost the visual appeal of the user interface
- Expand on our previous work with icons; this includes using the Pictos icon font to display new icons
- Talk about the considerations and shortcuts for working with different devices and screen sizes
- Explore the incredibly powerful Sencha theme engine using Sass and Compass to create complex visual skins using simple CSS-style commands

### Styling components versus themes

Before we get into this chapter, it's important to have a good understanding of the difference between styling an individual component and creating a theme.

Almost every display component in Sencha Touch has the option to set its own style. For example, a panel component can use a style in this way:

```
{  
    xtype: 'panel',  
    style: 'border: none; font: 12px Arial black',  
    html: 'Hello World'  
}
```

The style can also be set as an object using:

```
{  
    xtype: 'panel',  
    style : {  
        'border' : 'none',  
        'font' : '12px Arial black',  
        'border-left': '1px solid black'  
    }  
    html: 'Hello World'  
}
```



You will notice that inside the `style` block, we have quoted both sides of the configuration setting. This is still the correct syntax for JavaScript and a very good habit to get into for using `style` blocks. This is because a number of standard CSS styles use a dash as part of their name. If we do not add quotes to `border-left`, JavaScript will read this as `border minus left` and promptly collapse in a pile of errors.

We can also set a `style` class for a component and use an external CSS file to define the class as follows:

```
{  
    xtype: 'panel',  
    cls: 'myStyle',  
    html: 'Hello World'  
}
```

Your external CSS file could then control the style of the component in the following manner:

```
.myStyle {  
    border: none;  
    font: 12px Arial black;  
}
```

This class-based control of display is considered a best practice as it separates the style logic from the display logic. This means that when you need to change a border color, it can be done in one file instead of hunting through multiple files for individual style settings.

These styling options are very useful for controlling the display of individual components. There are also certain style elements, such as border, padding, and margin, that can be set directly in the components' configuration:

```
{  
    xtype: 'panel',  
    bodyMargin: '10 5 5 5',  
    bodyBorder: '1px solid black',  
    bodyPadding: 5,  
    html: 'Hello World'  
}
```

These configurations can accept either a number to be applied to all sides or a CSS string value, such as `1px solid black` or `10 5 5 5`. The number should be entered without quotes but the CSS string values need to be within quotes.

These kind of small changes can be helpful in styling your application, but what if you need to do something a bit bigger? What if you want to change the color or appearance of the entire application? What if you want to create your own default style for your buttons?

This is where themes and UI styles come into play.

## UI styling for toolbars and buttons

Let's do a quick review of the basic MVC application we created in *Chapter 2, Creating a Simple Application*, and use it to start our exploration of styles with toolbars and buttons.

To begin, we are going to add a few things to the first panel, which has our titlebar, toolbar, and **Hello World** text.

### Adding the toolbar

In `app/views`, you'll find `Main.js`. Go ahead and open that in your editor and take a look at the first panel in our items list:

```
items: [  
    {  
        title: 'Hello',  
        iconCls: 'home',  
        xtype: 'panel',  
        html: 'Hello World',  
        items: [  
            {
```

```
{  
    xtype: 'titlebar',  
    docked: 'top',  
    title: 'About TouchStart'  
}  
]  
}...
```

We're going to add a second toolbar on top of the existing one. Locate the `items` section, and after the curly braces for our first toolbar, add the second toolbar in the following manner:

```
{  
  
    xtype: 'titlebar',  
    docked: 'top',  
    title: 'About TouchStart'  
, {  
    docked: 'top',  
    xtype: 'toolbar',  
    items: [  
        {text: 'My Button'}  
    ]}  
}
```

Don't forget to add a comma between the two toolbars.

#### Extra or missing commas

While working in Sencha Touch, one of the most common causes of parse errors is an extra or missing comma. When you are moving the code around, always make sure you have accounted for any stray or missing commas. Fortunately for us, the Safari Error Console will usually give us a pretty good idea about the line number to look at for these types of parse errors. A more detailed list of common errors can be found at:

<http://javascript.about.com/od/reference/a/error.htm>

Now when you take a look at the first tab, you should see our new toolbar with our button to the left. Since the toolbars both have the same background, they are a bit difficult to differentiate. So, we are going to change the appearance of the bottom bar using the `ui` configuration option:

```
{  
    docked: 'top',  
    xtype: 'toolbar',  
    ui: 'light',  
    items: [  
]
```

```
{text: 'My Button'  
}  
]  
}
```

The `ui` configuration is the shorthand for a particular set of styles in Sencha Touch. There are several `ui` styles included with Sencha Touch, and later on in the chapter, we will show you how to make your own.



## Styling buttons

Buttons can also use the `ui` configuration setting, for which they offer several different options:

- `normal`: This is the default button
- `back`: This is a button with the left side narrowed to a point
- `round`: This is a more drastically rounded button
- `small`: This is a smaller button
- `action`: This is a brighter version of the default button (the color varies according to the active color of the theme, which we will see later)
- `forward`: This is a button with the right side narrowed to a point

Buttons also have some color options built into the `ui` option. These color options are `confirm` and `decline`. These options are combined with the previous shape options using a hyphen; for example, `confirm-small` or `decline-round`.

Let's add some new buttons and see how this looks on our screen. Locate the `items` list with our button in the second toolbar:

```
items: [
  {text: 'My Button'}
]
```

Replace that old `items` list with the following new `items` list:

```
items: [
  {
    text: 'Back',
    ui: 'back'
  }, {
    text: 'Round',
    ui: 'round'
  }, {
    text: 'Small',
    ui: 'small'
  }, {
    text: 'Normal',
    ui: 'normal'
  }, {
    text: 'Action',
    ui: 'action'
  }, {
    text: 'Forward',
    ui: 'forward'
  }
]
```

This will produce a series of buttons across the top of our toolbar. As you may notice, all of our buttons are aligned to the left. You can move buttons to the right by adding a `spacer` `xtype` in front of the buttons you want pushed to the right. Try this by adding the following between our `Forward` and `Action` buttons:

```
{ xtype: 'spacer' },
```

This will make the `Forward` button move over to the right-hand side of the toolbar:



Since buttons can actually be used anywhere, we can add some to our title bar and use the `align` property to control where they appear. Modify the `titlebar` for our first panel and add an `items` section, as shown in the following code:

```
{
    xtype: 'titlebar',
    docked: 'top',
    title: 'About TouchStart',
    items: [
        {
            xtype: 'button',
            text: 'Left',
            align: 'left'
        },
        {
            xtype: 'button',
            text: 'Right',
            align: 'right'
        }
    ]
}
```

Now we should have two buttons in our title bar, one on either side of the title:



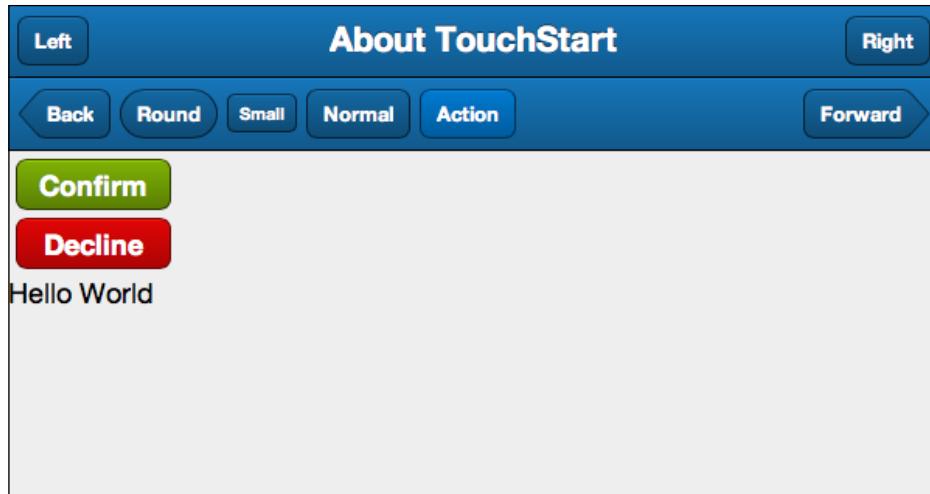
Let's also add some buttons to the panel container so we can see what the ui options `confirm` and `decline` look like.

Locate the end of the `items` section of our `HelloPanel` container and add the following after the second toolbar:

```
{  
    xtype: 'button',  
    text: 'Confirm',  
    ui: 'confirm',  
    width: 100  
, {  
    xtype: 'button',  
    text: 'Decline',  
    ui: 'decline',  
    width: 100  
}
```

There are two things you may notice that differentiate our panel buttons from our toolbar buttons. The first is that we declare `xtype: 'button'` in our panel but we don't in our toolbar. This is because the toolbar assumes it will contain buttons and `xtype` only has to be declared if you use something other than a button. The panel does not set a default `xtype` attribute, so every item in the panel must declare one.

The second difference is that we declare `width` for the buttons. If we don't declare `width` when we use a button in a panel, it will expand to the full width of the panel. On the toolbar, the button auto-sizes itself to fit the text.



You will also see that our two buttons in the panel are mashed together. You can separate them out by adding `margin: 5` to each of the button configuration sections.

These simple styling options can help make your application easier to navigate and provide the user with visual clues for important or potentially destructive actions.

## The tab bar

The tab bar at the bottom also understands the `ui` configuration option. In this case, the available options are `light` and `dark`. The tab bar also changes the icon appearance based on the `ui` option; a `light` toolbar will have dark icons and a `dark` toolbar will have light icons.

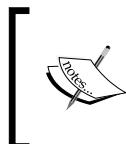
These icons are actually part of a special font called **Pictos**. Sencha Touch started using the Pictos font in Version 2.2 instead of images icons because of compatibility issues on some mobile devices.

[ The icon mask from previous versions of Sencha Touch is available but has been discontinued as of Version 2.2. You can see some of the icons available in the documentation for the Ext.Button component: <http://docs.sencha.com/touch/2.2.0/#!/api/Ext.Button> If you're curious about the Pictos font, you can learn more about it at <http://pictos.cc/> ]

## Sencha Touch themes

Sometimes you want to alter the looks of more than just a single panel or button. Themes in Sencha Touch are a powerful way to quickly change the overall look and feel of your application. We will cover the theming process a bit later in this chapter, but we do need to lay a bit of groundwork before we can get started. There is a lot of conceptual information to cover, but the flexibility you gain will be well worth the effort.

The first thing we need to cover is a basic overview of the tools used by Sencha Touch that make theming your application possible: Sass and Compass.



If you are already familiar with Sass and Compass, you will be more comfortable installing first and then covering the concepts. You can skip ahead to the *Setting up Sass and Compass* section.

## Introducing Sass and Compass

**Syntactically Awesome Stylesheets (Sass)** is used to extend standard CSS to allow variables, nesting, mixins, built-in functions, and selector inheritance. This means that all of your regular CSS declarations will work just fine, but you also get some extra goodies.

### Variables in Sass

Variables allow you to define specific values and then use them throughout the stylesheet. The variable names are arbitrary and start with \$. For example, we can use Sass to define the following:

```
$blue: #4D74C1;  
$red: #800000;  
$baseMargin: 10px;  
$basePadding: 5px;
```

We can then use the following variables as part of our standard CSS declarations in the Sass file:

```
.box1 {  
    border: 1px solid $blue;  
    padding: $basePadding;  
    margin: $baseMargin;  
}
```

We can also use basic math functions as follows:

```
.box2 {  
  border: 1px solid $blue;  
  padding: $basePadding * 2;  
  margin: $baseMargin / 2;  
}
```

This creates a box with twice the padding and half the margin of the first box. This is great for creating flexible, scalable layouts. By changing your base values, you can quickly scale your application to deal with multiple devices that have multiple resolutions and screen sizes.

Additionally, when you decide you want to change the shade of blue you are using, you only have to change it in one place. Sass also has a number of built-in functions for adjusting colors, such as:

- `darken`: This makes the color darker by percentage
- `lighten`: This makes the color lighter by percentage
- `complement`: This returns the complementary color
- `invert`: This returns the inverted color
- `saturate`: This saturates the color by a numerical value
- `desaturate`: This desaturates the color by a numerical amount

These functions allow you to perform operations, such as:

```
.pullQuote {  
  border: 1px solid blue;  
  color: darken($blue, 15%);  
}
```

There are also functions for numbers, lists, strings, and basic if-then statements. These functions help make your stylesheets as flexible as your programming code.



#### Sass functions

The full list of Sass functions can be found at <http://sass-lang.com/docs/yardoc/Sass/Script/Functions.html>.

## Mixins in Sass

**Mixins** are a variation of the standard Sass variables. Avoid simply declaring a single one-to-one variable such as the following:

```
$margin: 10px;
```

Instead, you can use a mixin to declare an entire CSS class as a variable:

```
@mixin baseDiv {  
  border: 1px solid #f00;  
  color: #333;  
  width: 200px;  
}
```

You can then take that mixin and use it in the Sass file:

```
#specificDiv {  
  padding: 10px;  
  margin: 10px;  
  float: right;  
  @include baseDiv;  
}
```

This gives you all of the attributes of the `baseDiv` mixin component plus the specific styles you declared in the `#specificDiv` class.

You can also set your mixin to use arguments to make it even more flexible.  
Let's look at an alternative version of what we had seen previously:

```
@mixin baseDiv($width, $margin, $float) {  
  border: 1px solid #f00;  
  color: #333;  
  width: $width;  
  margin: $margin;  
  float: $float;  
}
```

This means we can set values for `width`, `margin`, and `float` as part of our Sass code as follows:

```
#divLeftSmall {  
  @include baseDiv(100px, 10px, left);  
}  
#divLeftBig{  
  @include baseDiv(300px, 10px, left);  
}  
#divRightBig {  
  @include baseDiv(300px, 10px, right);  
}  
#divRightAlert {  
  @include baseDiv(100px, 10px, right);  
  color: #F00;  
  font-weight: bold;  
}
```

This gives us four `div` tags with slightly different properties. All of them share the same base properties as the mixin `baseDiv` class, but they have different values for `width` and `float`. We can also override the values for the mixin `baseDiv` by adding them after we include the mixin as seen in our `#divRightAlert` example.

## Nesting in Sass

Sass also allows nesting of CSS declarations. This not only lets you write styles that more closely mirror the structure of your HTML, but also makes for cleaner, more easily maintainable code.

In HTML, we often nest elements within one another to give the document a structure. A common example of this would be an unordered list that contains several list items, such as the following:

```
<ul>  
  <li>Main List Item 1</li>  
  <li>Main List Item 2</li>  
</ul>
```

Normally, to style this list via CSS, you would write rules for the `ul` elements separately from the rules for the `li` elements. The two rules might not even be near one another in your CSS files, making debugging or modifying the styles more difficult.

In Sass, we can write the following:

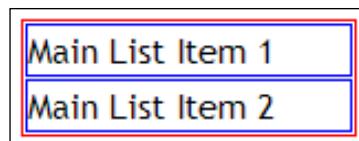
```
ul {  
    width: 150px;  
    border: 1px solid red;  
  
    li {  
        margin: 1px;  
        border: 1px solid blue;  
    }  
  
}
```

See how we nest the style declarations for our `li` element inside the style declaration for `ul`? Not only does nesting match the structure of the HTML document, but it also lets you know that it can be found inside the `ul` element when you want to update the `li` element.

When you compile this with Sass, the resulting CSS has separate rules for the `ul` and `li` elements:

```
ul {  
    width: 150px;  
    border: 1px solid red;  
}  
ul li {  
    margin: 1px;  
    border: 1px solid blue;  
}
```

If you were to view this list in your browser, you would see a list with a red border around it and blue borders around each of the individual list items.



It's also possible to reference the item one level up in the nesting using the ampersand (&) character. This is useful while adding things like hover states to nested elements, or more generally, grouping together the exceptions to your rules.

Suppose we want to change the background color when we hover over one of our `li` elements. We could add `&:hover` inside the `li` style declaration:

```
ul {  
  width: 150px;  
  border: 1px solid red;  
  
  li {  
    margin: 1px;  
    border: 1px solid blue;  
  
    &:hover {  
      background-color: #B3C6FF;  
    }  
  }  
}
```

The `&:hover` gets translated into `li:hover` by the Sass compiler:

```
ul li:hover {  
  background-color: #B3C6FF;  
}
```

The `&` special character doesn't have to be used at the beginning of a rule. Say your designer has the elements `li`, which use a bigger border when they're located in a special `#sidebar` component. You could write a separate rule after your `ul/li` rules or you could add the exception inside the `li` ruleset using the special `&` character:

```
ul {  
  li {  
    margin: 1px;  
    border: 1px solid blue;  
  
    &:hover {  
      background-color: #B3C6FF;  
    }  
    div#sidebar& {  
      border-width: 3px;  
    }  
  }  
}
```

The preceding code will be translated to the following rule:

```
div#sidebar ul li { border-width: 3px; }
```

You can also nest CSS namespaces. In CSS, if properties all start with the same prefix, such as `font-`, then you can nest them as well:

```
li {
  font: {
    family: Verdana;
    size: 18px;
    weight: bold;
  }
}
```

Be sure to remember to put the colon after the namespace. When compiled, this will become the following:

```
li {
  font-family: Verdana;
  font-size: 18px;
  font-weight: bold;
}
```

This works for any namespace CSS property, such as `border-` or `background-`.

## Selector inheritance in Sass

Selector inheritance in Sass is analogous to object inheritance in JavaScript. In the same way, a `panel` component extends the `container` object, meaning that a `panel` has all the properties and functions of a `container`, and then some. Sass lets you have objects that inherit the styles of other objects.

Say we want to create some message box elements for our application, one for informational messages and one for errors. First, we need to define a generic box:

```
.messageBox {
  margin: 10px;
  width: 150px;
  border: 1px solid;
  font: {
    size: 24px;
    weight: bold;
  }
}
```

Now, in any class where we want to include the `.messageBox` styles, we just use the `@extend` directive `@extend .messageBox;` on a line by itself:

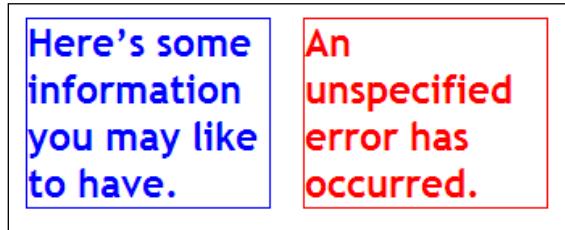
```
.errorBox {
  @extend .messageBox;
  border-color: red;
  color: red;
}

.infoBox {
  @extend .messageBox;
  border-color: blue;
  color: blue;
}
```

Then, in HTML, we would just use the `.errorBox` and `.infoBox` classes:

```
<div class="infoBox">Here's some information you may like to have.</div>
<div class="errorBox">An unspecified error has occurred.</div>
```

Put it all together and you will see the left box with a blue border and blue text and the right box with a red border and red text:



## Compass

Just as Sencha Touch is a framework built on the lower-level languages of JavaScript, CSS, and HTML, Compass is a framework built on Sass and CSS. Compass provides a suite of reusable components for styling your application. These include:

- **CSS Resets:** These enforce a uniform appearance for most HTML across all of the major web browsers.
- **Mixins:** These allow you to declare complex programmatic functions for your CSS.
- **Layouts and Grids:** These enforce width and height standards to assist in keeping your layout consistent across all pages.

- **Image Spriting:** This allows you to automatically generate a single image from multiple smaller images (this is faster for the browser to download). The CSS will automatically show just the portion of the image you need, hiding the rest.
- **Text Replacement:** This allows you to automatically swap specific text pieces within your document.
- **Typography:** This provides advanced options for using fonts within your web pages.

Compass also incorporates into its components the latest in CSS best practices, meaning that your stylesheet will be leaner and more efficient.

## Sass + Compass = themes

Sencha Touch themes take Sass and Compass one step further by providing variables and mixins whose functionalities are specific to Sencha Touch. The JavaScript portion of Sencha Touch generates lots of very complex HTML in order to display various components such as toolbars and panels. Rather than learning all of the intricate classes and HTML tricks used by Sencha Touch, you can simply use the appropriate mixins to change the appearance of your application.

## Setting up Sass and Compass

If you decide that you would like to create your own Sencha Touch theme, you won't have to install either Sass or Compass as they come packaged with Sencha Cmd.

However, Windows users will first need to install Ruby. Ruby is used to compile the Sass/Compass files into a working theme. Linux and OS X users should already have Ruby installed on their computers.

## Installing Ruby on Windows

Windows users should download the Ruby installer from <http://rubyinstaller.org/>.



We recommend downloading Version 1.9.2 as Sencha Cmd can have problems with newer versions of Ruby.

Run the installer and follow the onscreen instructions to install Ruby. Be sure to check the box that says **Add Ruby executables to your PATH**. This will save you a lot of typing on the command line later on.

Once the installation is complete, open up the command line in Windows by going to **Start | Run**, typing `cmd`, and pressing *Enter*. This should bring up the command line.

Now, try typing `ruby -v`. You should see something such as the following:

```
C:\Ruby192>ruby -v  
ruby 1.9.2p180 (2011-02-18) [i386-mingw32]
```

This means that Ruby is correctly installed.

## Creating a custom theme

The next thing we need to do is create our own theme SCSS file. Locate the `app.scss` file in `TouchStart/resources/sass` and make a copy of the file. Rename the new copy of the file as `myTheme.scss`.

Once you have renamed the file, you will need to compile the theme into an actual CSS file that our application can read. To do this, we need to return to the command line and move into our `TouchStart/resources/sass` directory:

```
cd /path/to/TouchStart/resources/sass
```

Once you are in the directory, you can enter the following command:

```
compass compile
```

This will compile our new theme and create a new file under `resources/css` called `myTheme.css`.



Using `compass compile` will compile any `.scss` files in the directory. You will need to run this command each time you make changes to the `.scss` file. However, you can also use the command `compass watch` to monitor the current folder for any changes and compile them automatically.

Now that we have our new CSS theme file, we need to tell the application to load it. In previous versions of Sencha Touch, the CSS files were loaded from the `index.html` file. However, with applications generated by Sencha Cmd, the CSS files are actually loaded from within the `app.json` file located in our main `TouchStart` directory.

Open `app.json` and look for the section where it says:

```
"css": [  
  {  
    "path": "resources/css/app.css",  
    "update": "delta"  
  }  
]
```

Change this section to:

```
"css": [  
  {  
    "path": "resources/css/myTheme.css",  
    "update": "delta"  
  }  
]
```

#### **SCSS and CSS**

 Notice that we are currently including a stylesheet from the `css` folder called `sencha-touch.css` and we have a matching file in the `scss` folder called `sencha-touch.scss`. When the SCSS files are compiled, they create a new file in your `css` folder. This new file will have the suffix `.css` instead of `.scss`.  
`.scss` is the file extension for Sass files.

If you reload the application in your web browser, you won't see any changes since we have simply duplicated the file for our theme. Let's take a look at how we can change that. Open your `myTheme.scss` file. You should see the following:

```
@import 'sencha-touch/default';  
@import 'sencha-touch/default/all';
```

This code grabs all of the default Sencha Touch theme information. When we run `compass compile` or `compass watch`, it gets compiled and compressed into a CSS file that our application can read.

The best part is that we can now change the entire color scheme of the application with a single line of code.

## Base color

One of the key variables in the Sencha Touch theme is `$base_color`. This color and its variations are used throughout the entire theme. To see what we mean, let's change the color of our theme to a nice forest green by adding the following to the top of our `myTheme.scss` file (on top of all the other text):

```
$base_color: #546346;
```

Next, we need to recompile the Sass files to create our `myTheme.css` file. If you are running `compass watch`, this will happen automatically when you save the Sass file. If not, you will need to run `compass compile` as before to update the CSS (remember you need to run this from inside the `resources/sass` directory).

### **compass compile versus compass watch**

Compass uses the `compile` command to create the new stylesheet based on your SCSS file. However, you can also set up Compass to watch a particular file for changes and automatically compile files when anything new is added. This command is entered on the command line as the following:

#### **compass watch filename**



This command will remain active as long as your terminal is open. Once you close the terminal window, you will need to run the command again in order to make Compass watch out for changes.

Reload the page in Safari and you should see a new forest green look for our application.

Note that this one line of code has created variations for both our dark and light toolbars. Changing the base color has also changed the icons for our tab bar at the bottom.

This is all pretty cool, but what if we want to tweak individual parts of the theme? Sencha Touch themes provide exactly what we need using mixins and the `ui` configuration option.

## Mixins and the UI configuration

As we have noted previously, the Sencha theme system is a set of predefined mixins and variables that get compiled to create a CSS stylesheet. Each component has its own mixins and variables for controlling styles. This means you can override these variables or use the mixins to customize your own theme.

You can also use mixins to create additional options for the `ui` configuration option (beyond the simple `light` and `dark` values that we have seen previously). For example, we can modify the color of our toolbar by adding a new mixin to our `myTheme.sass` file.

In our `myTheme.sass` file, locate the line that says the following:

```
@import 'sencha-touch/default/all';
```

After this line, add the following line:

```
@include sencha-toolbar-ui('subnav', #625546, 'matte');
```

This code tells Sass to create a new `ui` option for the toolbar. Our new option will be called `subnav`, and it will have a base color of `#625546`. The last option sets the style for the gradient. The available styles are:

- `flat`: No gradient
- `matte`: A subtle gradient
- `bevel`: A medium gradient
- `glossy`: A glassy style gradient
- `recessed`: A reversed gradient

You can find additional information about these variables (and any available mixins) at the top of each component in the Sencha Touch documentation at <http://docs.sencha.com/touch/2.2.0/>.

The screenshot shows the Sencha Touch Ext.Button component documentation. At the top, there's a gear icon and the text "Ext.Button" in green, followed by " xtype: button". Below this is a navigation bar with tabs: "Configs 57", "Properties 0", "Methods 155", "Events 27", "CSS Vars 7", and "CSS Mixins 1". The main content area has a title "Simple Button" and a description: "Here is a Button in its simplest form:". To the right of the content, a sidebar lists several CSS variables: \$button-gradient, \$button-height, \$button-radius, \$button-stroke-weight, \$include-button-highlights, \$include-button-uis, and \$toolbar-icon-size. At the bottom of the page are links for "Code Editor" and "Live Preview".

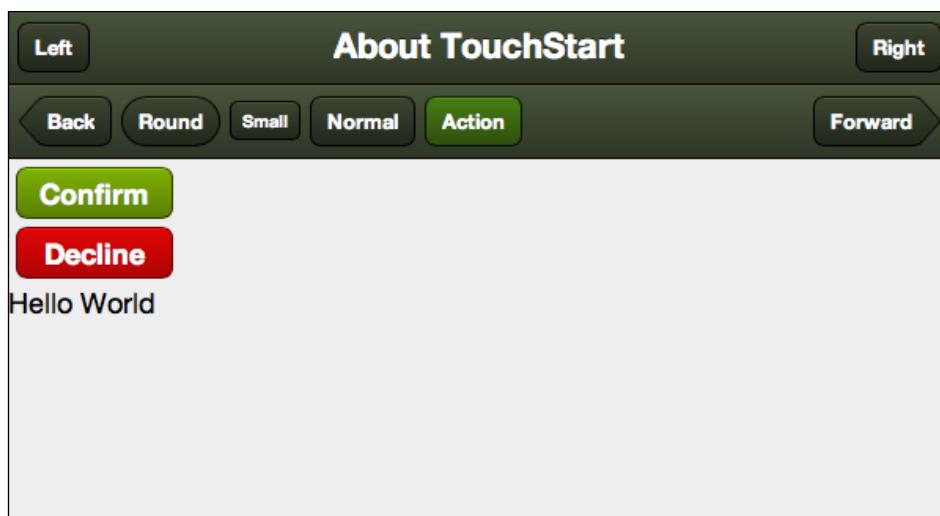
Once you have saved the file, you will need to recompile the stylesheet using the `compass compile` command on the command line.

We also need to change the `ui` configuration option in our JavaScript file. Locate our `Main.js` file in the `app/view` folder and open it. Find the second toolbar in our application, just on top of where we added the buttons. It should look as follows:

```
dock: 'top',
 xtype: 'toolbar',
 ui: 'light'
```

You will need to change `ui: 'light'` to `ui: 'subnav'` and save the file.

You can then reload the page to see your changes.



You will also notice that the buttons within the toolbar also adjust their colors to match the new toolbar's `ui` configuration.

## Adding new icons

As we mentioned earlier in the chapter, previous versions of Sencha Touch used icon masks for creating icons in your application. This caused some issues with browser compatibility, so the new icons are actually generated from the Pictos icon font. By default, 26 of these icons are included, but you can add more of them using the `icon` mixin.



A list of default icons in Sencha Touch is available at <http://docs.sencha.com/touch/2.2.0/#!/api/Ext.Button>.  
A full list of Pictos icons is available at <http://pictos.cc/font/>.

In your `myTheme.sass` file, locate the line that says:

```
@import 'sencha-touch/default/all';
```

After this line, add the following line:

```
@include icon('camera', 'v');
```

The `icon` mixin takes two arguments: the name you want to refer to the icon with (which is arbitrary) and the corresponding letter of the icon in the Pictos font. This second argument can be looked up on the Pictos website referred in the preceding tip.

Once the stylesheet is recompiled, we can change the `iconCls` value in our panel to use the new image.

In the `app/Main.js` file, locate `iconCls` for our `HBox` panel that currently says:

```
iconCls: 'info',
```

Replace the line with the following:

```
iconCls: 'camera',
```

Save your changes and reload the page to see your new icon. Don't forget to recompile the Sass file using `compass compile` on the command line.

## Variables

Variables are also available for most components, and they are used to control specific color, size, and appearance options. Unlike mixins, variables target a single setting for a component. For example, the `button` component includes variables for the following:

- `$button-gradient`: The default gradient for all buttons
- `$button-height`: The default height for all buttons
- `$button-radius`: The default border radius for all buttons
- `$button-stroke-weight`: The default border thickness for all buttons

As mentioned previously, you can find a listing for each of these variables (and any available mixins) at the top of each component in the Sencha Touch documentation at <http://docs.sencha.com/touch/2.2.0/>.

For example, if we add `$button-height: 2em;` to our `myTheme.scss` file, then we can recompile and see that buttons in our toolbar are now larger than they were before.



You will also notice that our `small` button did not change in size. This is because its UI configuration (`small`) has already been defined separately and includes a specific height. If you wanted to change the size of this button, you would need to remove the `ui` configuration for it in the `Main.js` file.

## More Sass resources

Using the mixins and variables included in the Sencha Touch theme, you can change almost any aspect of your interface to look exactly the way you want it to. There are a number of online resources that will help you dig deeper into all the possibilities with Sass and Compass.

### Additional resources



A full list of the Sencha Touch theme mixins and variables is available at <http://dev.sencha.com/deploy/touch/docs/theme/>.

Learn more about Sass at <http://sass-lang.com/>.

The Compass home page has examples of sites using Compass, tutorials, help, and more; it is available at <http://compass-style.org/>.

## Default themes and theme switching

With the introduction of Sencha Touch 2.2, there is now support for Blackberry 10 and Windows Phone platforms. To help style your application for these platforms, Sencha Touch 2.2 includes default themes for both of them. Let's take a look at how this works by creating a few new theme files.

Start by making two copies of our original `resources/sass/app.scss` file and rename them to `windows.scss` and `blackberry.scss`.

In both files, locate the following lines:

```
@import 'sencha-touch/default';
@import 'sencha-touch/default/all';
```

In `windows.scss`, change the lines to:

```
@import 'sencha-touch/windows';
@import 'sencha-touch/windows/all';
```

In `blackberry.scss`, change the lines to:

```
@import 'sencha-touch/bb10';
@import 'sencha-touch/bb10/all';
```

Next, you will need to run `compass compile` to create the new CSS files.

Now we can use our `app.json` file to switch these themes based on the platform our application is running on. Open up the `app.json` file and look for our `css` section again. It should look like this:

```
"css": [  
  {  
    "path": "resources/css/myTheme.css",  
    "update": "delta"  
  }  
]
```

Let's change that to look like this:

```
"css": [  
  {  
    "path": "resources/css/myTheme.css",  
    "platform": ["chrome", "safari", "ios", "android", "firefox"],  
    "theme": "Default",  
    "update": "delta"  
  },  
  {  
    "path": "resources/css/windows.css",  
    "platform": ["ie10"],  
    "theme": "Windows",  
    "update": "delta"  
  },  
  {  
    "path": "resources/css/blackberry.css",  
    "platform": ["blackberry"],  
    "theme": "Blackberry",  
    "update": "delta"  
  }  
]
```

Since most of us are not rolling in money, we probably don't have one of every kind of device to test with. However, we can add an argument at the end of our application URL to test each of our themes. For example:

`http://myapplication.com?platform=ie10`

## Styling the User Interface

This will be handled automatically in the application, but we can test our application by adding this argument to the URL. We should now have three different themes based on the platform.



We can make these kinds of conditional themes based on more than just these three options. The available platforms are:

- Phone, tablet, and desktop
- iOS, Android, and Blackberry
- Safari, Chrome, IE 10, and Firefox

This means we can change styles based on any of the platforms mentioned in the preceding list. Just generate new Sass/CSS stylesheets and include the appropriate configuration lines in `app.json` as we have in the previous examples.

These types of conditional style tweaks will help keep your application readable and usable across multiple devices.

## **Images on multiple devices with Sencha.io Src**

If your application uses images, you probably need something a bit more robust than conditional styles, such as those used in the previous section. Creating individual image sets for each device would be a nightmare. Fortunately, the folks at Sencha have an answer to this problem: a web-based service called `Sencha.io Src`.

`Sencha.io Src` is a separate service from Sencha and can be used in any web-based application. The service works by taking an original image and resizing it on the fly to fit the current device and screen size. These images are also cached by the service and optimized for quick, repeatable delivery. To use the `Sencha.io Src` service, the only thing you need to change is the URL for your image.

For example, a basic HTML image tag looks like this:

```

```

The same image tag, using the `Sencha.io Src` service, would look like this:

```

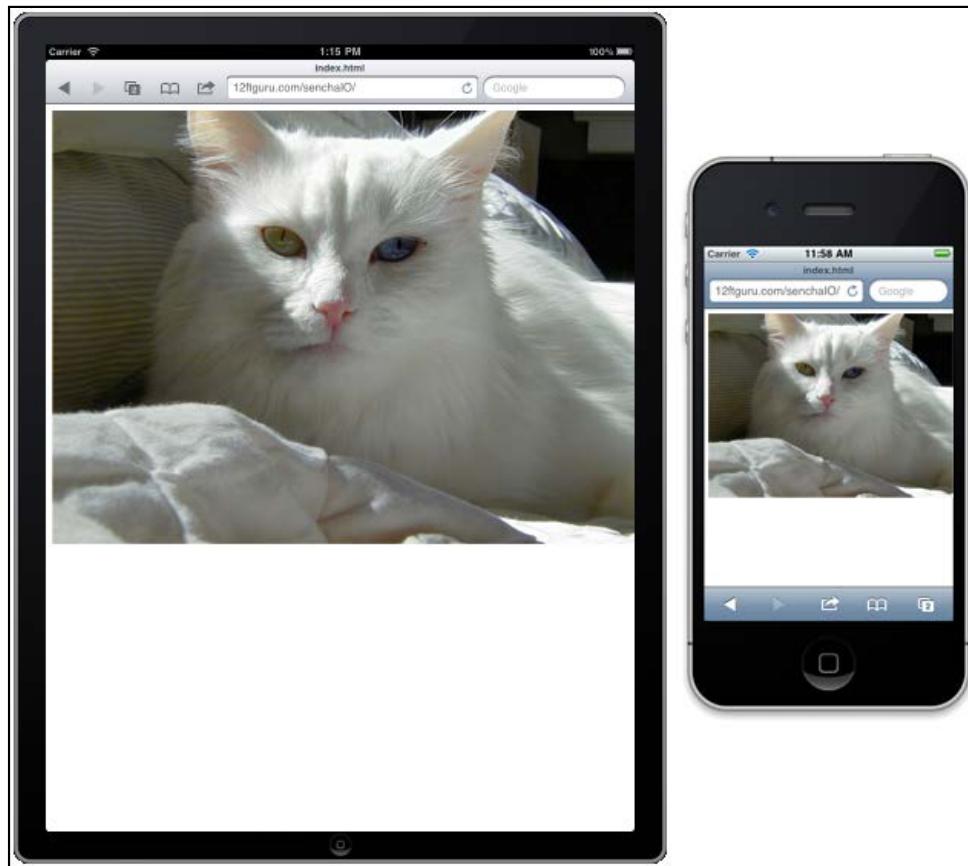
```

This passes the actual URL of your image to the system for processing.

**Image URLs in Sencha.io Src**



As you can see in the example, we are using a full image URL (with `http://www.mydomain.com/`) instead of a shorter relative URL (such as `/images/my-big-image.jpg`). Since the Sencha.io Src service needs to be able to directly get to the file from the main Sencha.io server, a relative URL will not work. The image file needs to be on a publicly available web server in order to work correctly.



By using this service, our large image will be scaled to fit the full width of our device's screen no matter what the size of the device we use. Sencha.io Src also keeps the image proportions correct without any squishing or stretching.

## Specifying sizes with Sencha.io Src

We don't always use fullscreen images in our applications. We often use them for things such as icons and accents within the application. Sencha.io Src also lets us specify a particular height and/or width for an image:

```

```

In this case, we have set the width of our image to be resized to 320 pixels and the height to 200 pixels. We can also constrain just the width; the height will automatically be set to the correct proportion:

```

```



It is important to note that Sencha.io Src will only shrink images; it will not enlarge them. If you enter a value larger than the dimensions of the actual image, it will simply display at the full image size. Your full-size image should always be the largest size you will need for the display.

## Sizing by formula

We can also use formulas to make changes based on the screen size of the device. For example, we can use the following code to make our photo 20 pixels narrower than the full width of the screen:

```

```

This is useful if you want to leave a small border around the image.

## Sizing by percentage

We can also use percentage widths to set our image sizes:

```

```

## Styling the User Interface

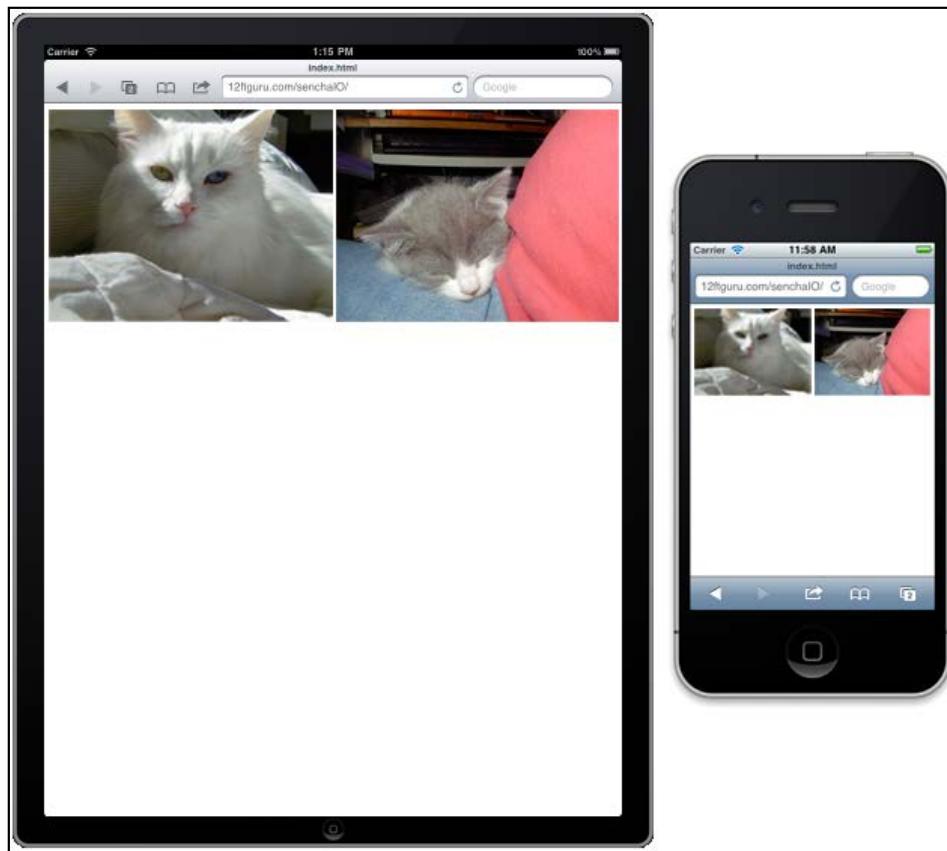
The `x50` part of our URL sets the image size to 50 percent of the screen width.

We can even combine these two elements to create a scalable image gallery:

```


```

By using the formula `-20x50-5`, we take our original image, remove 20 pixels for our margin, shrink it to 50 percent, and then remove an additional five pixels to allow for space between our two images.



## Changing file types

Sencha.io Src offers some additional options you may find useful. Firstly, it lets you change the file type for your image on the fly. For example, the following code will turn your JPG file into a PNG:

```

```

This can be useful while offering your applications' users multiple image downloading options.

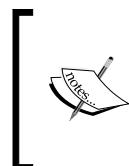
This option can also be combined with the resizing options:

```

```

This would convert the file to PNG format and scale it to 50 percent.

By using the functions available in Sencha.io Src, you can automatically size images for your application and provide a consistent look and feel across multiple devices.



Sencha.io is a free service. For a full list of all the functions you can use with Sencha.io Src, go to:

<http://www.sencha.com/learn/how-to-use-src-sencha-io/>

## Summary

In this chapter, we learnt how to style toolbars using the ui configuration option. We also talked about how Sencha Touch uses Sass and Compass to create a robust theme system. We included installation instructions for Sass and Compass and explained mixins, variables, nesting, and selector inheritance. Finally, we touched upon designing interfaces for multiple devices and handling automatic image resizing using Sencha.io Src.

In the next chapter, we will dive right back into the Sencha Touch framework. We'll review a bit of what we have previously learned about the components' hierarchies. Then, we will cover some of the more specialized components that are available. Finally, we'll give you some tips on finding the information you need in the Sencha Touch API documentation.



# 4

## Components and Configurations

In this chapter, we are going to take a deeper look at the individual components available in Sencha Touch. We will examine the layout configuration options and how they affect each of the components.

Throughout the chapter, we will use the simple base components as a starting point for learning about the more complex components. We'll also talk a bit about how to access our components after they have been created.

Finally, we will wrap up with a look at how to use the Sencha Touch API documentation to find detailed information on configurations, properties, methods, and events for each component.

This chapter will cover the following topics:

- The base component class
- Layouts revisited
- The TabPanel and Carousel components
- The FormPanel components
- MessageBox and Sheet
- The Map component
- The List and NestedList components
- Where to find more information on components

## The base component class

When we talk about components in Sencha Touch, we are generally talking about buttons, panels, sliders, toolbars, form fields, and other tangible items that we can see on the screen. However, all of these components inherit their configuration options, methods, properties, and events from a single base component with the startlingly original name of `Component`. This can obviously lead to a bit of confusion, so we will refer to this as `Ext.Component` for the rest of this book.

One of the most important things to understand is that you don't always use `Ext.Component` directly. It is more often used as a building block for all of the other components in Sencha Touch. However, it is important to be familiar with the base component class, because anything that it can do, all the other components can do. Learning this one class can give you a huge head start on everything else. Some of the most useful configuration options of `Ext.Component` are as follows:

- border
- cls
- disabled
- height/width
- hidden
- html
- margin
- padding
- scroll
- style
- ui

As the other components, which we will cover later in this chapter, inherit from the base component class, they will all have these same configuration options. One of the most critical of these configurations is `layout`.

## Taking another look at layouts

When you start creating your own applications, you will need a firm understanding of how the different layouts affect what you see on the screen. To this end, we are going to start out with a demonstration application that shows how the different layouts work.

For the purposes of this demo application, we will create the different components, one at a time, as individual variables. This is done for the sake of readability and should not be considered the best programming style. Remember that any items created in the following way will take up memory, even if the user never views the component:



```
var myPanel = Ext.create('Ext.Panel', { ... }
```

It is always a much better practice to create your components, using the `xtype` attributes, within your main container, as shown in the following code snippet:

```
items: [{ xtype: 'panel', ... }
```

This allows Sencha Touch to render the components as they are needed, instead of rendering them all at once when the page loads.

## Creating a card layout

To begin with, we will create a simple app with a container that is configured to use the `card` layout:

```
var myApp = Ext.create('Ext.Application', {
    name:'TouchStart',
    launch:function () {
        var mainPanel = Ext.create('Ext.Container', {
            fullscreen:true,
            layout:'card',
            cardSwitchAnimation:'slide',
            items:[hboxTest]
        });
        Ext.Viewport.add(mainPanel);
    }
});
```

This sets up a single container called `mainPanel` with a `card` layout. This `mainPanel` container is where we will be adding the rest of our layout example containers in this section.

The `card` layout arranges its items so that it looks similar to a stack of cards. Only one of these cards is active and displayed at a time. The `card` layout keeps any additional cards in the background and only creates them when the panel receives the  `setActiveItem()` command.

Each item in the list can be activated by using  `setActiveItem(n)`, where `n` is the item number. This can be a bit confusing, as the numbering of the items is zero-based, meaning that you start counting at 0 and not 1. For example, if you want to activate the fourth item in the list, you would use:

```
mainPanel.setActiveItem(3);
```

In this case, we are starting out with only a single card/item called  `hboxTest`. We need to add this container to make our program run.

## Creating an `hbox` layout

In the code in the preceding section, on top of the `var mainPanel = Ext.create('Ext.Container', {` line, add the following code:

```
var hboxTest = Ext.create('Ext.Container', {
    layout:{
        type:'hbox',
        align:'stretch'
    },
    items:[
        {
            xtype:'container',
            flex:1,
            html:'My flex is 1',
            margin:5,
            style:'background-color: #7FADCF'
        },
        {
            xtype:'container',
            flex:2,
            html:'My flex is 2',
            margin:5,
            style:'background-color: #7FADCF'
        },
        {
            xtype:'container',
            width:80,
            html:'My width is 80',
            margin:5,
            style:'background-color: #7FADCF'
        }
    ]
});
```

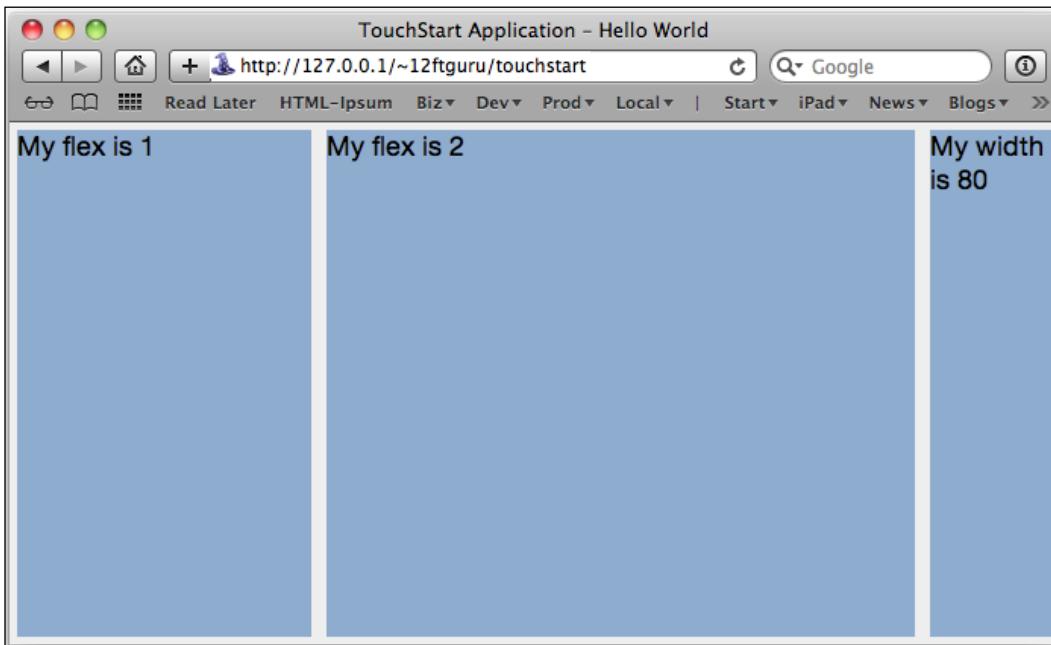
This gives us a container with an hbox layout and three child items.

**Child and parent**



In Sencha Touch, we often find ourselves dealing with very large arrays of items, nested in containers that are in turn nested in other containers. It is often helpful to refer to a container as a parent to any items it contains. These items are then referred to as the children of the container.

The hbox layout stacks its items horizontally and uses the `width` and `flex` values to determine how much horizontal space each of its child items will take up. The `align: 'stretch'` configuration causes the items to stretch in order to fill all of the available vertical space.



You should try adjusting the `flex` and `width` values to see how they affect the size of the child containers. You can also change the available configuration options for `align` (center, end, start, and stretch), to see the different options available. Once you are finished, let's move on and add some more items to our card layout.

## Creating a vbox layout

On top of our `var hboxTest = Ext.create('Ext.Container', {` line, add the following code:

```
var vboxTest = Ext.create('Ext.Container', {
    layout:{
        type:'vbox',
        align:'stretch'
    },
    items:[
        {
            xtype:'container',
            flex:1,
            html:'My flex is 1',
            margin:5,
            style:'background-color: #7FADCF'
        },
        {
            xtype:'container',
            flex:2,
            html:'My flex is 2',
            margin:5,
            style:'background-color: #7FADCF'
        },
        {
            xtype:'container',
            height:80,
            html:'My height is 80',
            margin:5,
            style:'background-color: #7FADCF'
        }
    ]
});
```

This code is virtually identical to our previous `hbox` code, a container with three child containers. However, this parent container uses `layout: vbox`, and the third child container in the `items` list uses `height` instead of `width`. This is because the `vbox` layout stacks its items vertically and uses the values for `height` and `flex` to determine how much space the child items will take up. In this layout, the `align: 'stretch'` configuration causes the items to stretch to fill the horizontal space.

Now that we have our `vbox` container, we need to add it to the items in our main `layoutContainer`. Change the `items` list in `layoutContainer` to say the following:

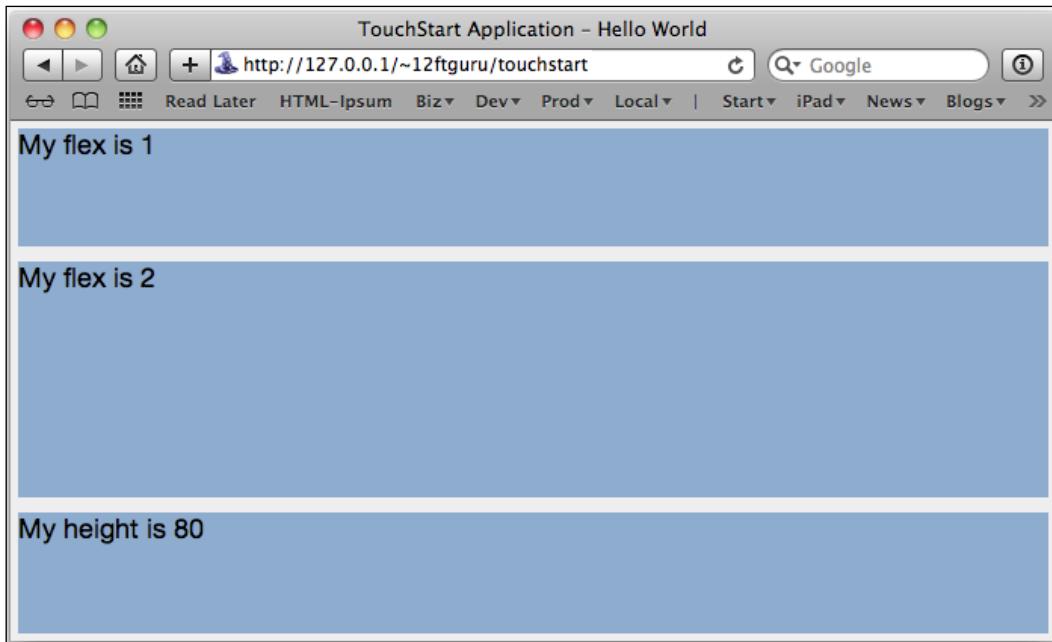
```
items: [hboxTest, vboxTest]
```

If we run the code now, it's going to look exactly the same as before. This is because our card layout on `layoutContainer` can only have one active item. You can set `layoutContainer` to show our new `vbox` by adding the following configuration to our `layoutContainer`:

```
activeItem: 1,
```

Remember that our items are numbered starting with zero, so item 1 is the second item in our list: `items: [hboxTest, vboxTest]`.

You should now be able to see the `vbox` layout for our application:



As with `hbox`, you should take a moment to adjust the `flex` and `width` values and see how they affect the size of the containers. You can also change the available configuration options for `align` (`center`, `end`, `start`, and `stretch`) in order to see the different options available. Once you are finished, let's move on and add some more items to our card layout.

## Creating a fit layout

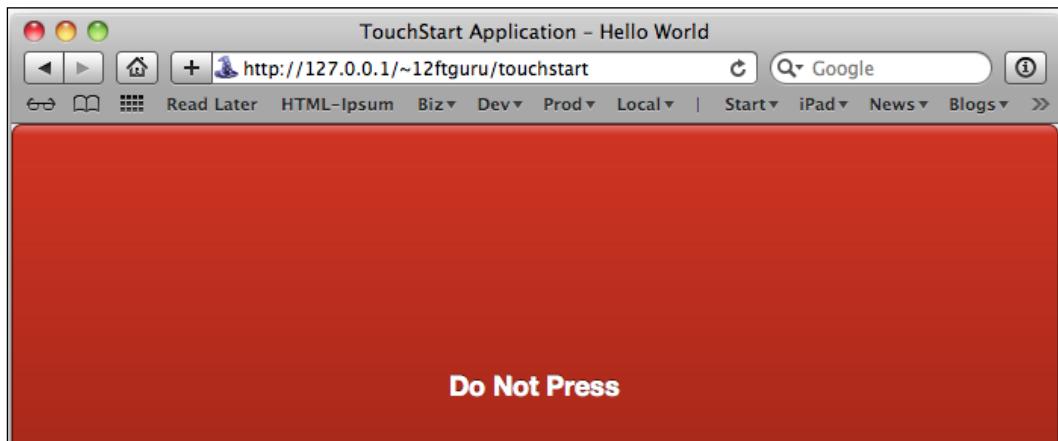
The `fit` layout is the most basic layout, and it simply makes any child items fill up the parent container. While this seems pretty basic, it can also have some unintended consequences, as we will see in our example.

On top of our preceding `var vboxTest = Ext.create('Ext.Container', {` line, add the following code:

```
var fitTest = Ext.create('Ext.Container', {
    layout:'fit',
    items:[
        {
            xtype:'button',
            ui:'decline',
            text:'Do Not Press'
        }
    ]
});
```

This is a single container with a `fit` layout and a button. Now, all we need to do is set the `activeItem` configuration on our main `layoutContainer` component by changing `activeItem: 1` to `activeItem: 2`.

If you reload the page now, you will see what we mean by unintended consequences:



As you can see, our button has expanded to fill the entire screen. We can change this by declaring a specific height and width for the button (and any other items we place in this container). However, `fit` layouts tend to work best for a single item that is intended to take up the entire container. This makes them a pretty good layout for child containers, where the parent container controls the overall size and position.

Let's see how this might work.

## Adding complexity

For this example, we are going to create a nested container and add it to our card stack. We will also add some buttons to make switching the card stack easier.

Our two new containers are variations on what we already have in our current application. The first is a copy of our hbox layout with a few minor changes:

```
var complexTest = Ext.create('Ext.Container', {
    layout: {
        type: 'vbox',
        align: 'stretch'
    },
    style: 'background-color: #FFFFFF',
    items: [
        {
            xtype: 'container',
            flex: 1,
            html: 'My flex is 1',
            margin: 5,
            style: 'background-color: #7FADCF'
        },
        hboxTest2,
        {
            xtype: 'container',
            height: 80,
            html: 'My height is 80',
            margin: 5,
            style: 'background-color: #7FADCF'
        }
    ]
});
```

You can copy and paste our old vboxTest code and change the first line to say complexTest instead of vboxTest. You will also need to remove the second container in our items list (parentheses and all) and replace it with hboxTest2. This is where we will nest another container with its own layout.

Now, we need to define hboxTest2 by copying our previous hboxTest code, and make a few minor changes. You will need to paste this new code up on top of where you placed the complexTest code; otherwise, you will get errors when we try to use hboxTest2 before we actually define it:

```
var hboxTest2 = Ext.create('Ext.Container', {
    layout: {
        type: 'hbox',
        align: 'stretch'
    },
    flex: 2,
    style: 'background-color: #FFFFFF',
    items: [
        {
            xtype: 'container',
            flex: 1,
            html: 'My flex is 1',
            margin: 5,
            style: 'background-color: #7FADCF'
        },
        {
            xtype: 'container',
            flex: 2,
            html: 'My flex is 2',
            margin: 5,
            style: 'background-color: #7FADCF'
        },
        {
            xtype: 'container',
            width: 80,
            html: 'My width is 80',
            margin: 5,
            style: 'background-color: #7FADCF'
        }
    ]
});
```

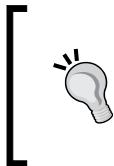
After you paste in the code, you will need to change the variable name to hboxTest2, and we will need to add a `flex` configuration to the main parent container. As this container is nested within our `vbox` container, the `flex` configuration is needed to define how much space `hboxTest2` will occupy.

Before we take a look at this new complex layout, let's make our lives a bit easier by adding some buttons to switch between our various layout cards.

Locate `mainPanel`, and underneath it, where we define the list of `items`, add the following code at the top of the list of `items`:

```
{
    xtype:'toolbar',
    docked:'top',
    defaults:{
        xtype:'button'
    },
    items:[
        {
            text:'hbox',
            handler:function () {
                mainPanel.setActiveItem(0);
            }
        },
        text:'vbox',
        handler:function () {
            mainPanel.setActiveItem(1);
        }
    ],
    {
        text:'fit',
        handler:function () {
            mainPanel.setActiveItem(2);
        }
    },
    {
        text:'complex',
        handler:function () {
            mainPanel.setActiveItem(3);
        }
    }
]
```

This code adds a toolbar to the top of our `mainPanel`, with a button for each of our layout cards.



In previous versions of Sencha Touch, the `toolbar` item was defined separately from the rest of the items and used a configuration called `dock` to control its position. In current versions, the `toolbar` component is defined inline with other items, and the placement of the toolbar is controlled by the `docked` configuration.

## *Components and Configurations*

---

Each button has a text configuration, which serves as the button's title, and a handler configuration. The handler configuration defines what happens when the button is tapped. For each of our buttons, we use the `mainPanel` variable we set earlier in the code:

```
var mainPanel = Ext.create('Ext.Container', { ... }
```

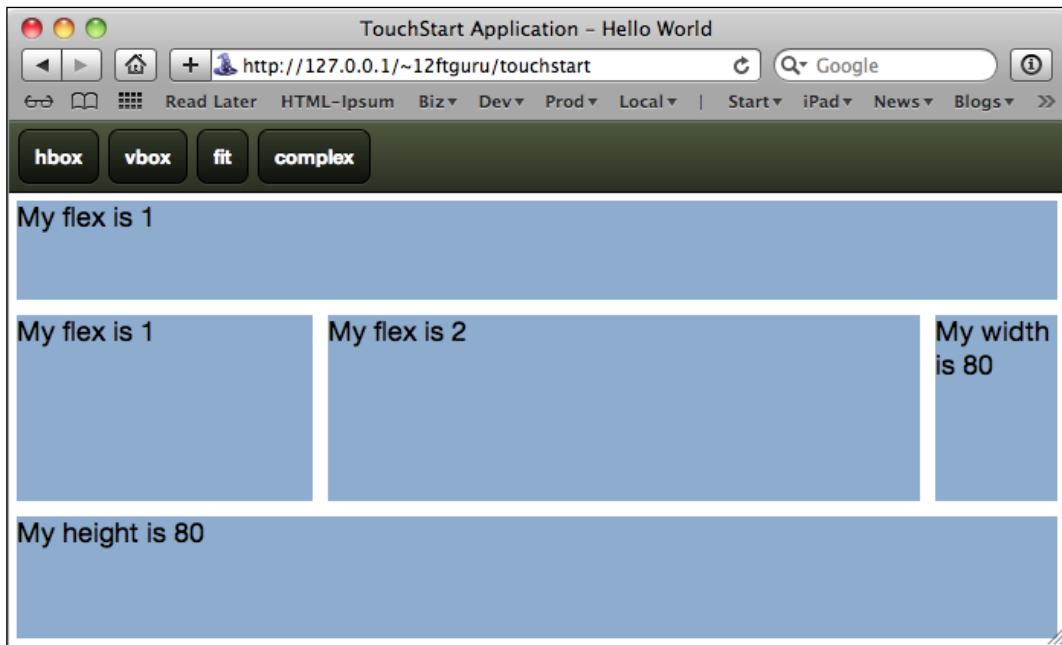
This lets us use any of the methods available to the container and its card layout. In the code for each button, we set the active item (which tab is visible) by using the following line of code:

```
mainPanel.setActiveItem(x);
```

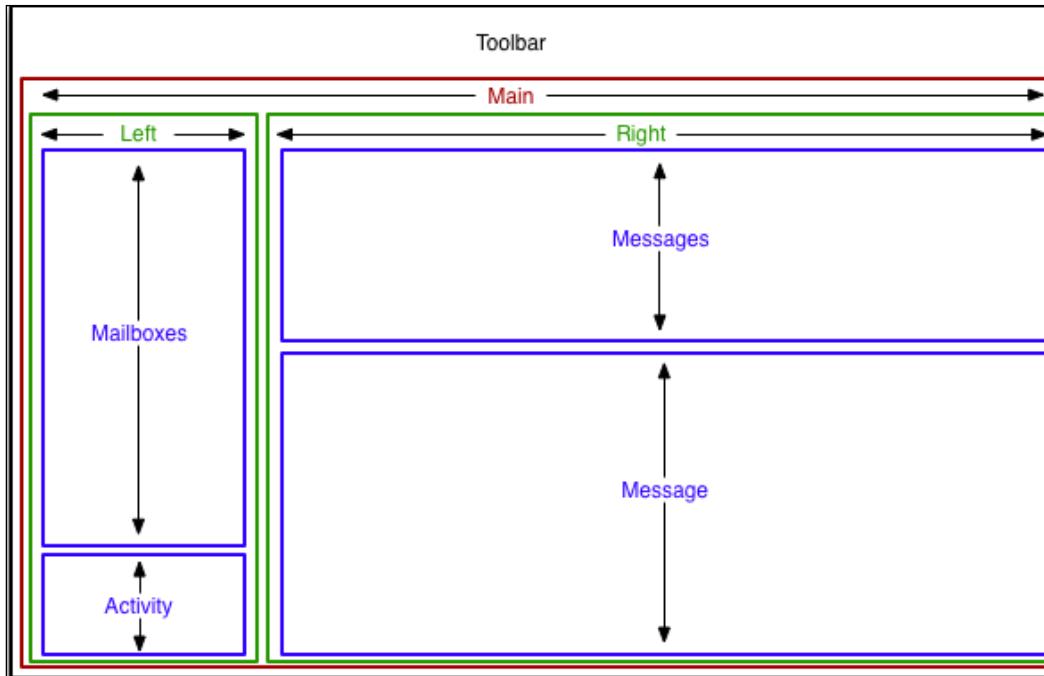
The `x` value, in this case, would be replaced by the index of the item we want to activate (remember that these go in order, starting with 0 and not 1).

Notice that we also leave the initial configuration option for `activeItem` in our `mainPanel` component. This will control which item is displayed when our application starts.

If you refresh the page, you should be able to click on the buttons and see each of our layouts, including the new complex layout.



As you can see from this example, our `vbox` layout splits the window into three rows. The `hbox` layout, in the second row, splits it into three columns. Using these types of nested layouts makes it pretty easy to create traditional layouts, such as those used in e-mails or social networking applications.



In this example, we have a typical layout for an e-mail application. This layout can be conceptually broken down into the following pieces:

- An application container with a **Toolbar** menu and a single container called **Main** with a `fit` layout.
- The **Main** container will have an `hbox` layout and two child containers called **Left** and **Right**.
- The **Left** container will have a `flex` value of 1 and a `vbox` layout. It will have two child containers called **Mailboxes** (with a `flex` of 3) and **Activity** (with a `flex` of 1).
- The **Right** container will have a `flex` of 3 and a `vbox` layout. It will also have two child containers called **Messages** (with a `flex` of 1) and **Message** (with a `flex` of 2).

Building container layouts such as these is a good practice. To see the example code for this container layout, take a look at the `TouchStart2b.js` file in the code bundle. It's also a good idea to create some base layouts such as these to use as templates for getting a jumpstart on building your future applications.

Now that we have a better understanding of layouts, let's take a look at some of the components we can use inside the layouts.

## The TabPanel and Carousel components

In our last application, we used buttons and a card layout to create an application that switched between different child items. While it is often desirable for your application to do this programmatically (with your own buttons and code), you can also choose to have Sencha Touch set this up automatically, using a `TabPanel` or `Carousel`.

### Creating a TabPanel component

A `TabPanel` component is useful when you have a number of views the user needs to switch between, such as contacts, tasks, and settings. The `TabPanel` component auto-generates the navigation for the layout, which makes it very useful as the main container for an application.

One of our early example applications in *Chapter 2, Creating a Simple Application*, used a simple `TabPanel` to form the basis of our application. The following is a similar code example:

```
Ext.application({
    name:'TouchStart',
    launch:function () {
        var myTabPanel = Ext.create('Ext.tab.Panel', {
            fullscreen:true,
            tabBarPosition:'bottom',
            items:[
                {
                    xtype:'container',
                    title:'Item 1',
                    fullscreen:false,
```

```
        html:'TouchStart container 1',
        iconCls:'info'
    },
{
    xtype:'container',
    html:'TouchStart container 2',
    iconCls:'home',
    title:'Item 2'
},
{
    xtype:'container',
    html:'TouchStart container 3',
    iconCls:'favorites',
    title:'Item 3'
}
])
});
Ext.Viewport.add(myTabPanel);
}
);
});
```

In this code, `Ext.tab.Panel` automatically generates a card layout; you don't have to declare a layout. You will probably want to declare a `tabBarPosition` value for the component. This is where your tabs will automatically appear; the default is the top of the screen.

This will generate a large square button for each child item in the `items` list. The button will also use the `iconCls` value to assign an icon to the button. The `title` configuration is used to name the button.

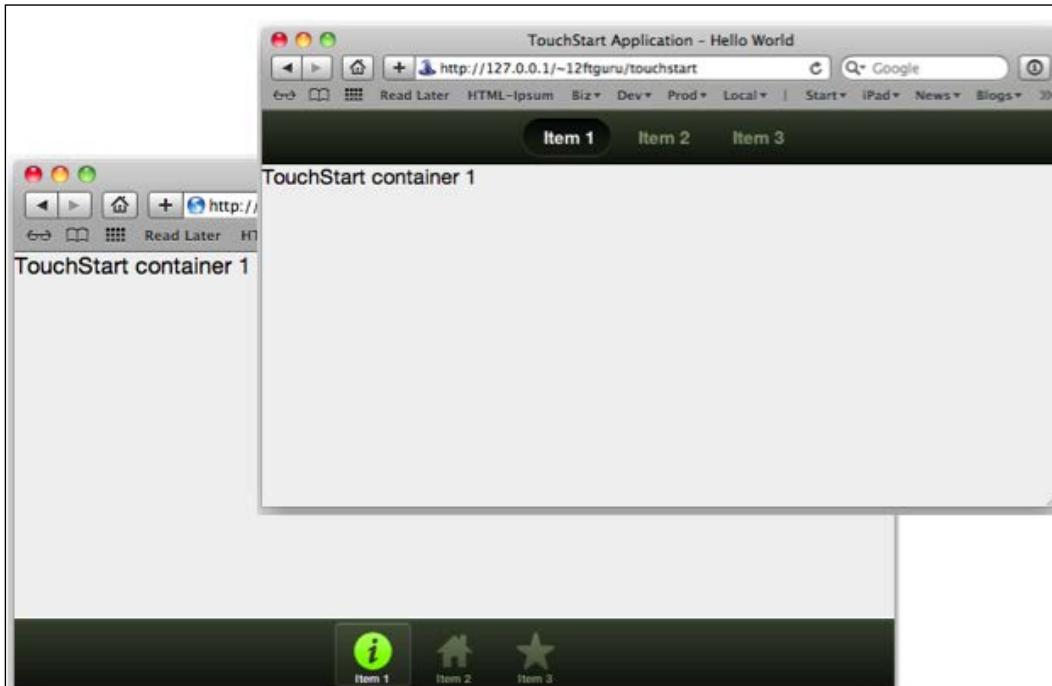


See the previous chapter for more information of the available icons and styling for the tab panel. It should also be noted that these icons are only used when the `tabBarPosition` value is set to `bottom`.

## *Components and Configurations*

---

If you set the `tabBarPosition` value to the top (or leave it blank), it makes the buttons small and round. It also eliminates the icons, even if you declare a value for `iconCls`, in your child items.



## **Creating a Carousel component**

The `Carousel` component is similar to `tabpanel`, but the navigation it generates is more appropriate for things such as slide shows. It probably would not work as well as a main interface for your application, but it does work well as a way to display multiple items in a single swipeable container.

Similar to `tabpanel`, `Carousel` gathers its child items and automatically arranges them in a card layout. In fact, we can actually make just some simple modifications to our previous code to make it into a `Carousel` component:

```
Ext.application({
    name:'TouchStart',
    launch:function () {
        var myCarousel = Ext.create('Ext.carousel.Carousel', {
            fullscreen:true,
            direction:'horizontal',
            items:[
                {
                    title:'Item 1',
                    html:'Content for Item 1'
                },
                {
                    title:'Item 2',
                    html:'Content for Item 2'
                }
            ]
        });
    }
});
```

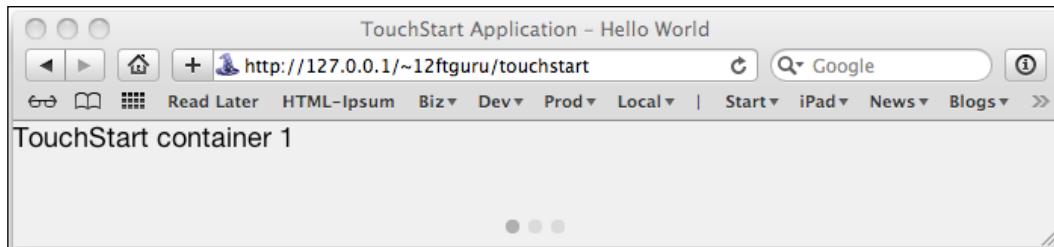
```

        {
            html:'TouchStart container 1'
        },
        {
            html:'TouchStart container 2'
        },
        {
            html:'TouchStart container 3'
        }
    ]
});
Ext.Viewport.add(myCarousel);
}
);
}
);

```

The first thing we did was use `Ext.create` to make a new `Ext.carousel.Carousel` class instead of a new `Ext.tab.Panel` class. We also added a configuration for `direction`, which can be either `horizontal` (scrolling from left to right) or `vertical` (scrolling up or down).

We removed the docked toolbar, because, as we will see, `Carousel` doesn't use one. We also removed the icon's class and the title from each of our child items for the same reason. Finally, we removed the `xtype` configuration, as the `Carousel` component automatically creates an `Ext.Container` class for each of its items.



Unlike `tabpanel`, `carousel` has no buttons, only a series of dots at the bottom, with one dot for each child item. While it is possible to navigate using the dots, the `carousel` component automatically sets itself up to respond to a swipe on a touch screen. You can duplicate this gesture in the browser by clicking and holding the pointer with the mouse, while moving it horizontally. If you declare a `direction: vertical` configuration in your `carousel`, you can swipe vertically to move between the child items.

Similar to the card layout in our example at the beginning of the chapter, both the `tabpanel` and `carousel` components understand the `activeItem` configuration.

This lets you set which item appears when the application first loads. Additionally, they all understand the `setActiveItem()` method that allows you to change the selected child item after the application loads.

The `Carousel` component also has methods for `next()` and `previous()`, which allow you to step through the items in order.

It should also be noted that as `tabpanel` and `carousel` both inherit from `Ext.Container`, they also understand any methods and configurations that containers understand.

Along with containers, `tabpanel` and `carousel` will serve as the main starting point for most of your applications. However, there is another type of container you will likely want to use at some point: the `FormPanel` component.

## Creating a FormPanel component

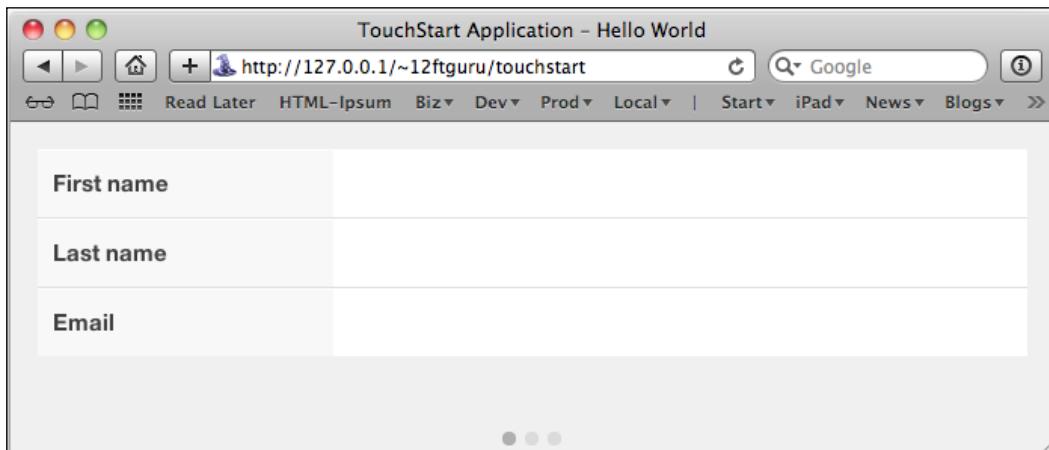
The `FormPanel` component is a very specialized version of the `Ext.Container` component, and as the name implies, it is designed to handle form elements. Unlike panels and containers, you don't need to specify the layout for `formpanel`. It automatically uses its own special form layout.

A basic example of creating a `formpanel` component is as follows:

```
var form = Ext.create('Ext.form.FormPanel', {
    items: [
        {
            xtype: 'textfield',
            name : 'first',
            label: 'First name'
        },
        {
            xtype: 'textfield',
            name : 'last',
            label: 'Last name'
        },
        {
            xtype: 'emailfield',
            name : 'email',
            label: 'Email'
        }
    ]
});
```

For this example, we just create the panel and add items for each field in the form. Our `xtype` tells the form what type of field to create. We can add this to our `carousel` and replace our first container, as follows:

```
Ext.application({
    name:'TouchStart',
    launch:function () {
        var myCarousel = Ext.create('Ext.carousel.Carousel', {
            fullscreen:true,
            direction:'horizontal',
            items:[
                {
                    xtype:'form',
                    html:'TouchStart container 2'
                },
                {
                    xtype:'form',
                    html:'TouchStart container 3'
                }
            ]
        });
        Ext.Viewport.add(myCarousel);
    }
});
```



Anyone who has worked with forms in HTML should be familiar with all of the standard field types, so the following `xtype` attribute names will make sense to anyone who is used to standard HTML forms:

- `checkboxfield`
- `fieldset`
- `hiddenfield`
- `passwordfield`

- `radiofield`
- `selectfield`
- `textfield`
- `textareafield`

These field types all match their HTML cousins for the most part. Sencha Touch also offers a few specialized text fields that can assist with validating the user's input:

- `emailfield`: This accepts only a valid e-mail address, and on iOS devices, it will pull up an alternate e-mail address and URL-friendly keyboard
- `numberfield`: This accepts only numbers
- `urlfield`: This accepts only a valid web URL, and also brings up the special keyboard

These special fields will only allow the submit operation if the input is valid.

All of these basic form fields inherit properties from the main container class, so they have all of the standard `height`, `width`, `cls`, `style`, and other container configuration options.

They also have a few field-specific options:

- `label`: This is a text label to use with the field
- `labelAlign`: This is where the label appears; this can be top or left, and defaults to left
- `labelWidth`: This tells us how wide the label should be
- `name`: This corresponds to the HTML name attribute, which is how the value of the field will be submitted
- `maxLength`: This tells us how many characters can be used in the field
- `required`: This tells us if the field is required in order for the form to be submitted

#### **Form field placement**

While `FormPanel` is typically the container you will use while displaying form elements, it has the advantage of understanding the `submit()` method that will post the form values via an AJAX request or POST.

If you include a form field in something that is not a `FormPanel` component, you will need to get and set the values for the field using your own custom JavaScript method.



In addition to the standard HTML fields, there are a few specialty fields available in Sencha Touch. These include the `DatePicker`, `slider`, `spinner`, and `toggle` fields.

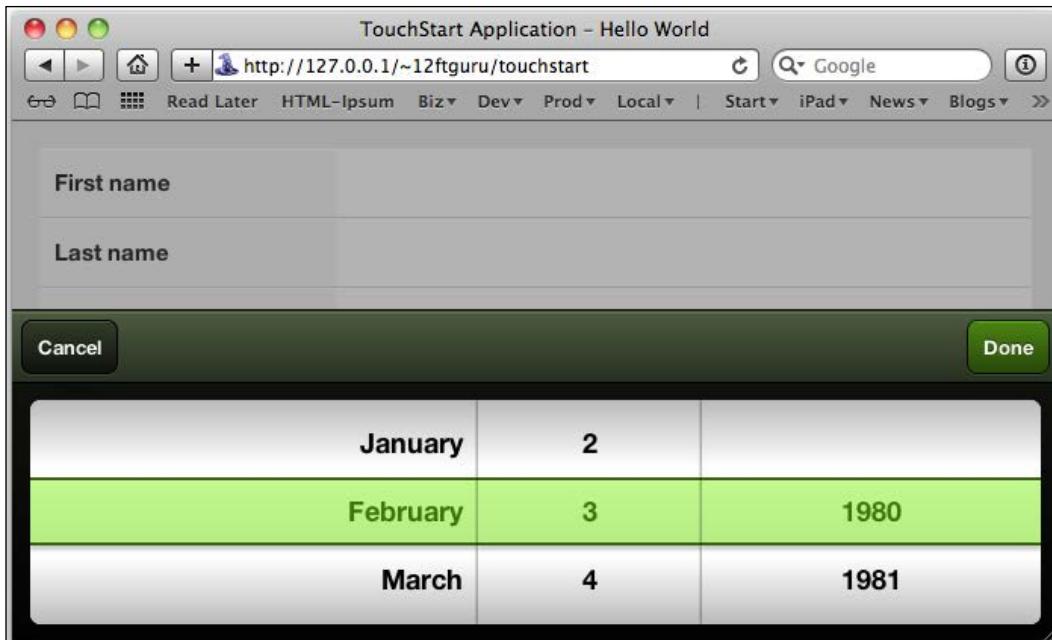
## Adding a DatePicker component

The `datepickerfield` component (is this correct?) places a clickable field in the form with a small triangle on the far-right side.

You can add a date picker to our form by adding the following code after the `emailfield` item:

```
{
    xtype: 'datepickerfield',
    name : 'date',
    label: 'Date'
}
```

When the user clicks on the field, a `DatePicker` component will appear, allowing the user to select a date by rotating the month, day, and year wheels, or by swiping up or down.



The datepickerfield also has the option configs for the following:

- `yearFrom`: The start year for the date picker.
- `yearTo`: The end year for the date picker.
- `slotOrder`: This uses an array of strings to set the slot order. It defaults to `['month', 'day', 'year']`.

## Adding sliders, spinners, and toggles

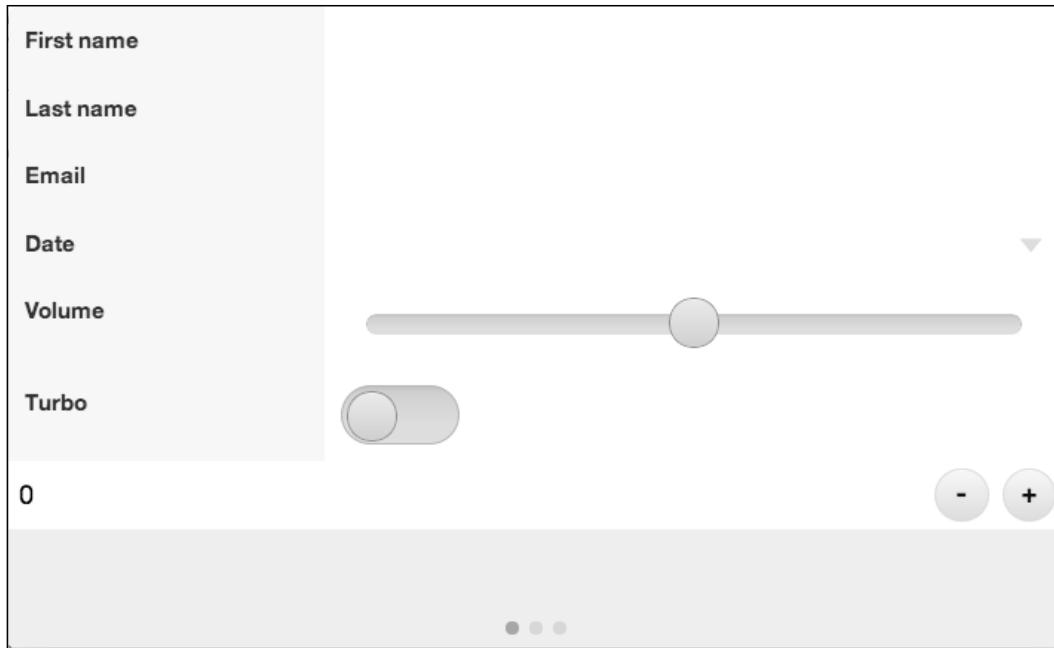
Sliders allow for the selection of a single value from a specified numerical range. The sliderfield value displays a bar with an indicator that can be slid horizontally to select a value. This can be useful for setting volume, color values, and other ranged options.

Like the slider, a spinner allows for the selection of a single value from a specified numerical range. The spinnerfield value displays a form field with a numerical value with + and - buttons on either side of the field.

A toggle allows for a simple selection between one and zero (on and off) and displays a toggle-style button on the form.

Add the following new components to the end of our list of items:

```
{
  xtype: 'sliderfield',
  label: 'Volume',
  value: 5,
  minValue: 0,
  maxValue: 10
},
{
  xtype: 'togglefield',
  name : 'turbo',
  label: 'Turbo'
},
{
  xtype: 'spinnerfield',
  minValue: 0,
  maxValue: 100,
  incrementValue: 2,
  cycle: true
}
```



Our `sliderfield` and `spinnerfield` have configuration options for `minValue` and `maxValue`. We also added an `incrementValue` attribute to `spinnerfield` that will cause it to move in increments of 2 whenever the + or - button is clicked.



We will cover sending and receiving data with forms later in *Chapter 6, Getting the Data In.*



## The MessageBox and Sheet components

At some point, your application will probably need to give feedback to the user, ask the user a question, or alert the user to an event. This is where the `MessageBox` and `Sheet` components come into play.

## Creating a MessageBox component

The MessageBox component creates a window on the page, and can be used to display alerts, gather information, or present options to the user. MessageBox can be called in three different ways:

- Ext.Msg.alert takes a title, some message text, and an optional callback function to call when the **OK** button on the alert is clicked.
- Ext.Msg.prompt takes a title, some message text, and a callback function to call when the **OK** button is pressed. The prompt command creates a text field and adds it to the window automatically. The function, in this case, is passed the text of the field for processing.
- Ext.Msg.confirm takes a title, some message text, and a callback function to call when either one of the buttons is pressed.

### The callback function

A callback function is a function that gets called automatically in response to a particular action taken by the user or the code. This is basically the code's way of saying "When you are finished with this, call me back and tell me what you did". This callback allows the programmer to make additional decisions based on what happened in the function.

Let's try a few examples, starting with a simple message box:

```
Ext.application({  
    name:'TouchStart',  
    launch:function () {  
        var main = Ext.create('Ext.Container', {  
            fullscreen:true,  
            items:[  
                {  
                    docked:'top',  
                    xtype:'toolbar',  
                    ui:'light',  
                    items:[  
                        {  
                            text:'Panic',  
                            handler:function () {  
                                Ext.Msg.alert('Don\'t Panic!', 'Keep  
Calm. Carry On.');                            }  
                        ]  
                    ]  
                }  
            ]  
        }  
    }  
}
```

```

        ]
    });

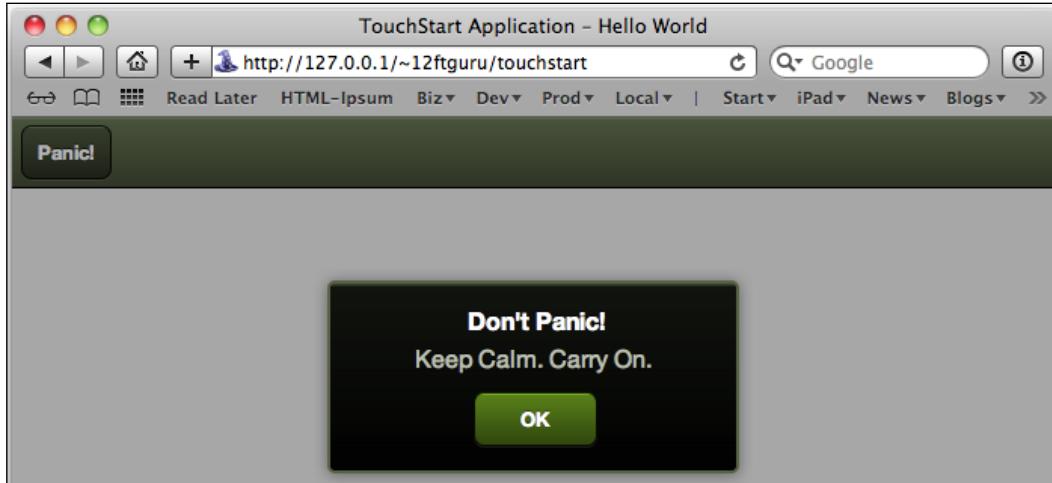
Ext.Viewport.add(main);
}
});

```

This code sets up a simple panel with a toolbar and a single button. The button has a handler that uses `Ext.Msg.alert()` to show our message box.

 **Escaping quotes**

In our previous example, we use the string `Don\'t Panic` as the title for our message box. The `\` tells JavaScript that our second single quote is part of the string and not the end of the string. You can see in the example that the `\` disappears in our message box.



Now, let's add a second button to our items in our `toolbar` component for a `Ext.Msg.prompt` style message box:

```

{
    text:'Greetings',
    handler:function () {
        Ext.Msg.prompt('Greetings!', 'What is your name?', function
(btn, text) {
            Ext.Msg.alert('Howdy', 'Pleased to meet you ' + text);
        });
    }
}

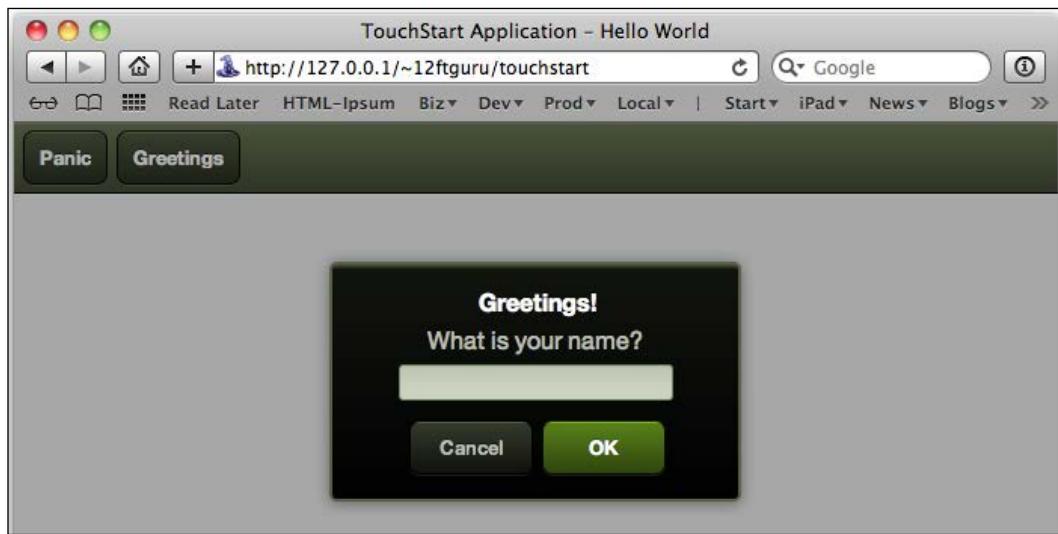
```

## *Components and Configurations*

---

This message box is a bit more complex. We create our Ext.Msg.prompt class with a title, a message, and a function. The prompt will create our text field automatically, but we need to use the function to determine what to do with the text that the user types in the field.

The function is passed a value for the button and a value for the text. Our function grabs the text and creates a new alert to respond, along with the name the user typed into the field.



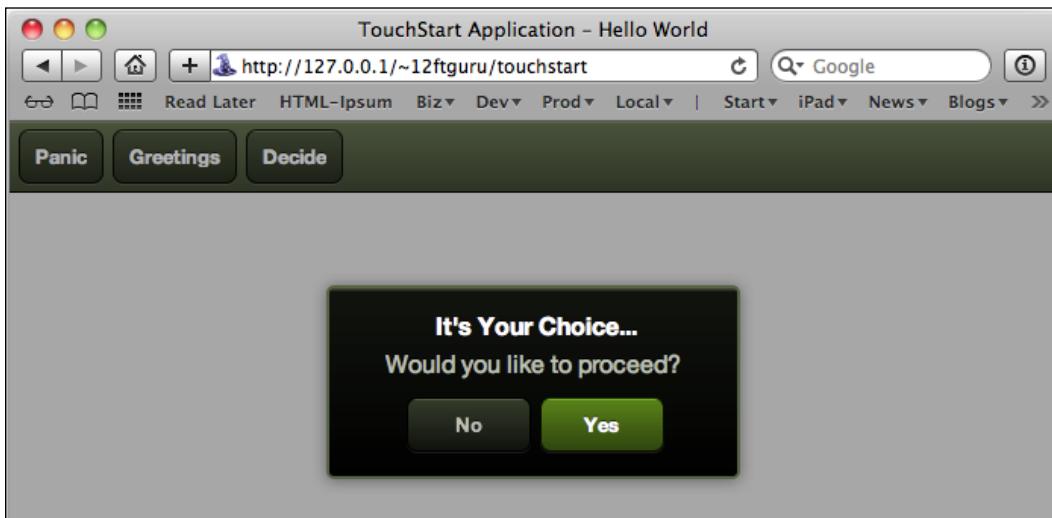
The Ext.Msg.confirm class of MessageBox is used for decisions the user needs to make, or confirmation of a particular action the system is going to take.

Let's add the following component to our list of items in the toolbar component:

```
{  
    text: 'Decide',  
    handler: function() {  
        Ext.Msg.confirm('It\'s Your Choice...', 'Would you like to  
proceed?', function(btn) {  
            Ext.Msg.alert('So be it!', 'You chose '+btn);  
        });  
    }  
}
```

Similar to the prompt function of the Ext.Msg component, the confirm version takes a title, a message, and a callback function. The callback function is passed the button that the user pressed (as the value btn), which can then be used to determine what steps the system should take next.

In this case, we just pop up an alert to display the choice the user has made. You can also use an `if...then` statement to take different actions depending on which button is clicked.



## Creating a Sheet component

The Sheet component is similar to the `Ext.Msg` component, in that it is typically used to pop up new information or options on the screen. It also presents this new information by appearing over the top of the existing screen. As with `MessageBox`, no further actions can be taken until `Sheet` is closed or responded to in some fashion.

Let's add another button to our list of items in the `items` section of our toolbar component. This button will pop up a new Sheet component:

```
{
    text:'Sheet',
    handler:function () {
        var mySheet = Ext.create('Ext.Sheet', {
            height:250,
            layout:'vbox',
            stretchX:true,
            enter:'top',
            exit:'top',
            items:[
                {
                    xtype:'container',
                    layout:'fit',

```

```
        flex:1,
        padding:10,
        style:'color: #FFFFFF',
        html:'A sheet is also a panel. It can do anything
the panel does.'
    },
{
    xtype:'button',
    height:20,
    text:'Close Me',
    handler:function () {
        this.up('sheet').hide();
    }
},
],
listeners:{
    hide:function () {
        this.destroy();
    }
}
);
}
}
Ext.Viewport.add(mySheet);
mySheet.show();
```

There are a lot of new things here, but some should seem familiar. Our button starts with the `text` value for the button to be displayed and then creates a `handler` value that tells the button what to do when tapped.

We then create a new `Ext.Sheet` class. As `Sheet` inherits from the `panel`, we have familiar configuration options, such as `height` and `layout`, but we also have a few new options. The `stretchX` and `stretchY` configurations will cause the `Sheet` component to expand to the full width (`stretchX`) or height (`stretchY`) of the screen.

The values for `enter` and `exit` control how the `Sheet` component will slide into place on the screen. You can use `top`, `bottom`, `left`, and `right`.

Our sheet uses a `vbox` layout with two items, a `container` object for our text and a `button` object to hide the `Sheet` component when the user has finished reading it. The `button` component itself contains an interesting bit of code:

```
this.up('sheet').hide();
```

When we refer to the `this` keyword, we are referring to the button object, as the function occurs inside the button itself. However, we really need to get to the `Sheet` that the button is contained by, in order to close it when the button is clicked. In order to do this, we use a clever little method called `up`.

The `up` method will basically crawl upwards through the structure of the code, looking for the requested item. In this case, we are searching by `xtype` and we have requested the first sheet encountered by the search. We can then hide the sheet with the `hide()` method.

### **Ext.ComponentQuery**

When you want to get one component, and you've given it an ID, you can use `Ext.get`(`Cmp`), as we discussed earlier. If, instead, you want to get multiple components, or one component based on where it is in relation to another component, you can use `query()`, `up()`, and `down()`. To hide a toolbar that's inside a panel, you can use the following code:

```
panel.down('toolbar').hide();
```

Additionally, to get all toolbars in your application, you could use the following command:

```
var toolbars = Ext.ComponentQuery.query('toolbar');
```



Once we hide the `Sheet` component, we still have a bit of a problem. The `Sheet` component is now hidden, but it still exists in the page. If we go back and click on the button again, without destroying the `Sheet`, we will just keep creating more and more new sheets. That means more and more memory, which also means an eventual death spiral for your application.

What we need to do is make sure we clean up after ourselves, so that the sheets don't pile up. This brings us to the last part of our code and the `listeners` configuration at the end:

```
listeners: {
  hide: {
    fn: function(){ this.destroy(); }
  }
}
```

A listener listens for a particular event, in this case, the `hide` event. When the `hide` event occurs, the listener then runs the additional code listed in the `fn` configuration. In this case, we destroy the `Sheet` component using `this.destroy()`.

In the next chapter, we will cover listeners and events in detail.

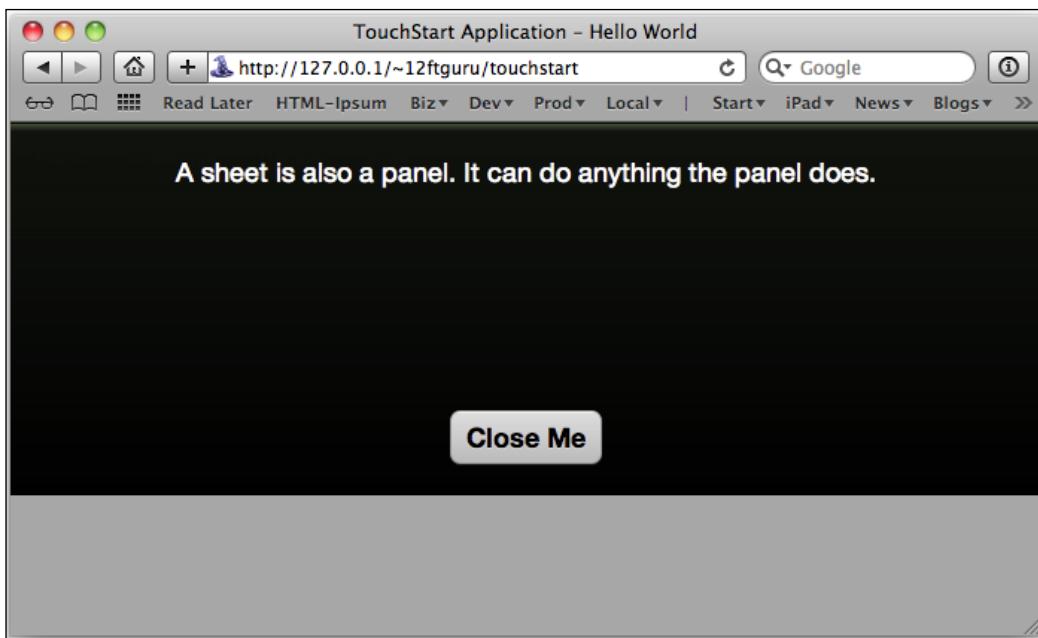
#### A word about the `this` variable

When we use the variable `this` in our programs, it always refers to the current item. In the preceding case, we used `this` in two separate places, and it referred to two separate objects. In our initial usage, we were inside the configuration options for the button, and so `this` referred to the button. When we later used `this` as part of our listener, we were inside the configuration for the sheet, and this referred to the sheet.

If you find yourself getting confused, it can be very helpful to use `console.log(this)`; to make sure you are addressing the correct component.



You should now be able to click on the **Sheet** button and view our new sheet.



## Creating an ActionSheet component

ActionSheet is a variation of the standard sheet designed to display a series of buttons. This is a good option when you only need a quick decision from the user, with obvious choices that don't require a lot of explanation. For example, a delete confirmation screen would be a good use for an action sheet.

Let's add a new button to our layout that will pull up an `ActionSheet` component for a delete confirmation:

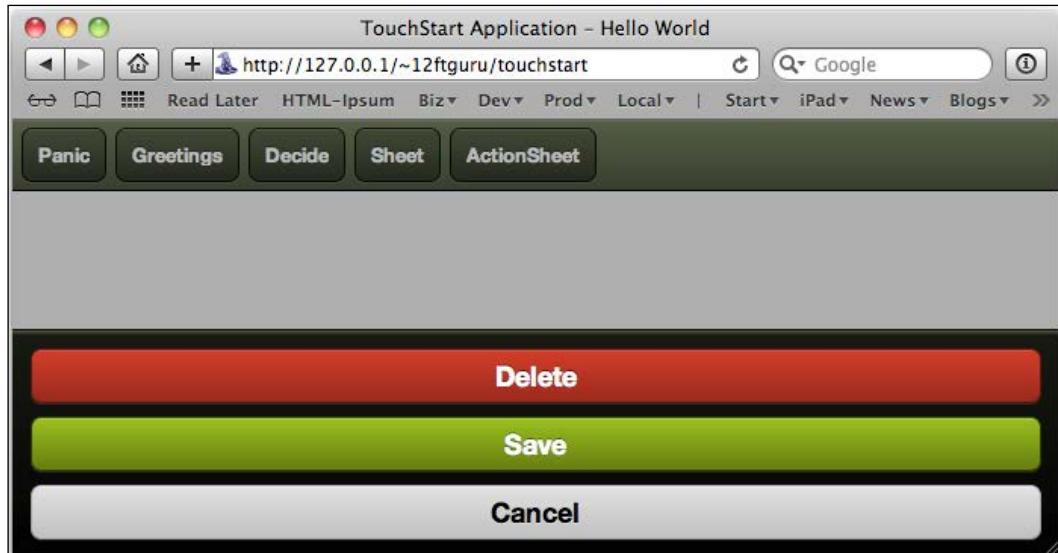
```
{
    text: 'ActionSheet',
    handler: function() {
        var actionSheet = Ext.create('Ext.ActionSheet', {
            items: [
                {
                    text: 'Delete',
                    ui : 'decline'
                },
                {
                    text: 'Save',
                    ui : 'confirm'
                },
                {
                    text: 'Cancel',
                    handler: function() {
                        this.up('actionsheet').hide();
                    }
                }
            ],
            listeners: {
                hide: {
                    fn: function(){ this.destroy(); }
                }
            }
        });
        Ext.Viewport.add(actionSheet);
        actionSheet.show();
    }
}
```

The `ActionSheet` object is created in much the same fashion as our previous sheet example. However, the action sheet assumes that all of its items are buttons, unless you specify a different `xtype` value.

Our example has three simple buttons: **Delete**, **Save**, and **Cancel**. The **Cancel** button will hide the `ActionSheet` component, and the other two buttons are just for show.

As with our previous example, we also want to destroy the `ActionSheet` component when we hide it. This prevents copies of the `ActionSheet` component from stacking up in the background and creating problems.

Clicking on the **ActionSheet** button in our application should now display the action sheet we created:



## Creating a Map component

The Map component is a very specialized container designed to work with the Google Maps API. The container can be used to display most of the information that Google Maps display.

We are going to create a very basic example of the Map container for this section, but we will come back to it in *Chapter 9, Advanced Topics*, and cover some of the more advanced tricks you can use.

For this example, let's create a new JavaScript file:

```
Ext.application({
    name: 'TouchStart',
    launch: function() {
        var map = Ext.create('Ext.Container', {
            fullscreen: true,
            layout: 'fit',
            items: [
                {
                    xtype: 'map',
                    useCurrentLocation: true
                }
            ]
    }
})
```

```

        });
        this.viewport = map;
    }
});

```

For this example, we are just creating a single Container component with one item. The item is a map and has the configuration `useCurrentLocation: true`. This means that the browser will attempt to use our current location as the center of the map's display. The user is always warned when this happens, and is given an option to decline.

Before we can see how this works, we need to make one change to our standard `index.html` file. Underneath the line where we include our other JavaScript files, we need to include a new file from Google:

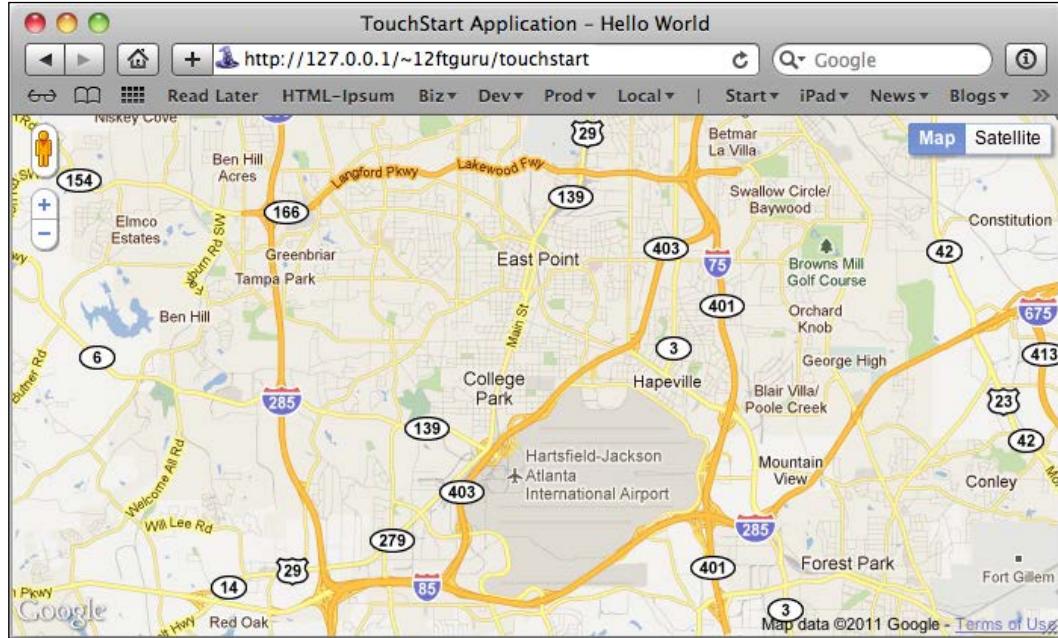
```

<!-- Google Maps API -->
<script type="text/javascript" src="http://maps.google.com/maps/api/
js?sensor=true"></script>

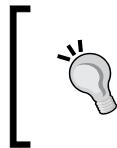
```

This will include all of the functions needed for us to use the Google Maps API.

If you reload the page, you will be asked if you want to allow your current location to be used by the application. Once you accept, you should see a new map with your current location at the center.



You can also use the `map` property and the `mapOptions` configuration option to access the rest of the Google Maps functionality. We will explore some of these options and go into much greater detail in *Chapter 9, Advanced Topics*.



#### Google Maps API documentation

The full Google Maps API documentation can be found at <http://code.google.com/apis/maps/documentation/v3/reference.html>.



## Creating lists

Sencha Touch offers few different kinds of `list` components. Each of these `list` components consists of three basic parts:

- **List panel:** This is in charge of gathering the other items as part of its configuration options
- **XTemplate:** This determines how each line in the list is displayed
- **Data store:** This contains all of the data that will be used in the list



It should also be noted that a store can (and typically will) have a model associated with it to define the data records for the store. However, it is also possible to simply define the fields as part of the store, which we will do in this example. We will cover models and stores later in the chapters related to data in this book.



In one of our first examples, we created a list object similar to this one:

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

        var myDudeList = Ext.create('Ext.Container', {
            fullscreen: true,
            layout: 'fit',
            items: [
                {
                    xtype: 'list',
```

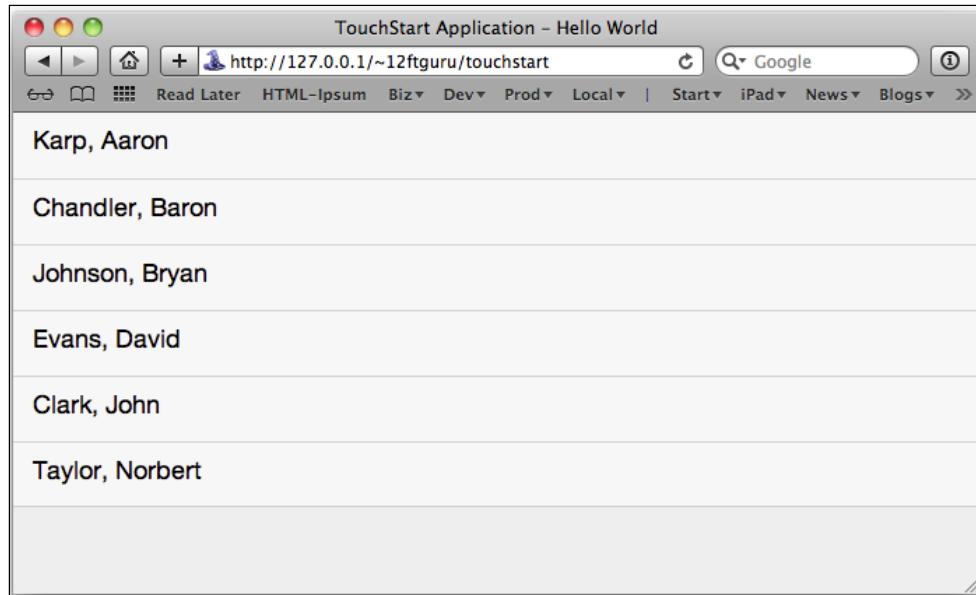
```
itemTpl: '{last}, {first}',  
store: Ext.create('Ext.data.Store', {  
    fields: [  
        {name: 'first', type: 'string'},  
        {name: 'last', type: 'string'}  
    ],  
    data: [  
        {first: 'Aaron', last: 'Karp'},  
        {first: 'Baron', last: 'Chandler'},  
        {first: 'Bryan', last: 'Johnson'},  
        {first: 'David', last: 'Evans'},  
        {first: 'John', last: 'Clark'},  
        {first: 'Norbert', last: 'Taylor'}  
    ]  
})  
});  
});  
Ext.Viewport.add(myDudeList);  
}  
});
```

We start by creating our application as before. We then created a single container with a list item. The list item requires a data store, and the data store requires a set of fields or a data model. In this case, we will use a set of fields for simplicity.

```
fields: [  
    {name: 'first', type: 'string'},  
    {name: 'last', type: 'string'}  
]
```

This code gives us two potential values for each data record we will be using in the list: `first` and `last`. It also tells us the `type` for each value; in this case, both are strings. This lets the data store know how to handle sorting the data and lets the XTemplate understand how the data can be used.

In this example, we have set `itemTpl: '{last}, {first}'`. This `itemTpl` value serves as a template or XTemplate in Sencha Touch terms. The XTemplate takes the data from each record in the store and tells the list to display each data record as: the last name, followed by a comma, and then the first name. We will cover the XTemplates in greater detail in *Chapter 7, Getting the Data Out*.



Note that, right now, our list is not sorted alphabetically. We need to add a sorter to the store underneath the configuration option for our model:

```
sorters: 'last'
```

This will sort our list by the value `last` (the person's last name).

## Adding grouped lists

Grouped lists are also common to a number of applications. Typically, grouping is used for lists of people or other alphabetical lists of items. Address books or long lists of alphabetical data are great places for grouped lists. A grouped list places an `indexBar` component on the screen, allowing the user to jump to a specific point in the list.

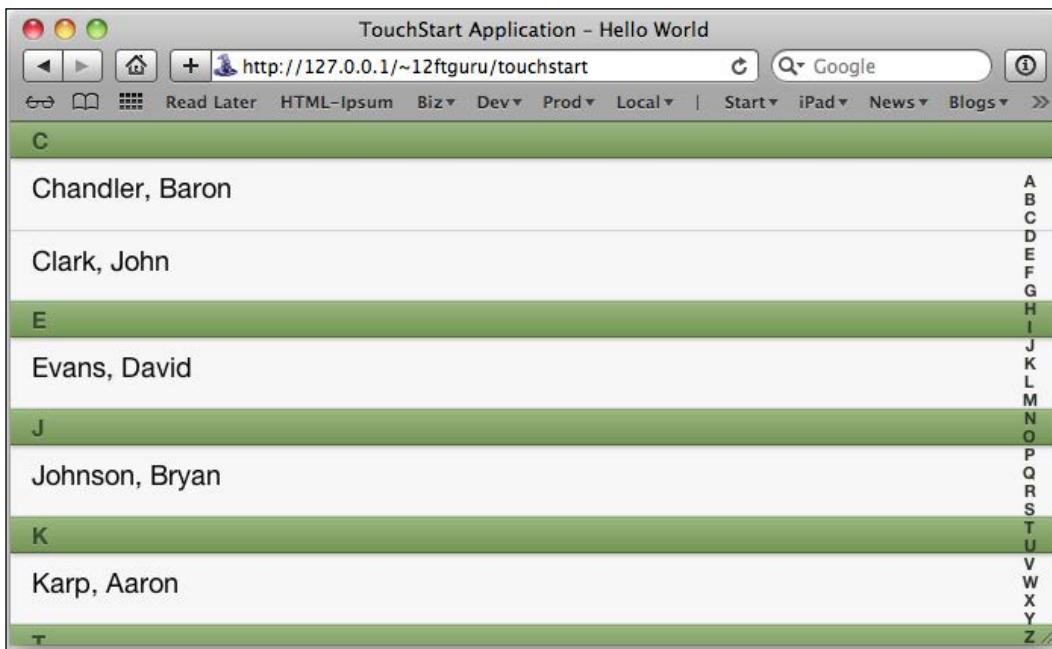
To group our current list, we need to add two configuration settings to our `list` component. Add the following code beneath where we declare `xtype: 'list'`:

```
grouped: true,  
indexBar: true,
```

We also need to add a function to our store that will get the string used to display our alphabetical `indexBar`. Add the following code in place of the `sorters` configuration in the `store` component:

```
grouper: {
  groupFn : function(record) {
    return record.get('last').substr(0, 1);
  },
  sortProperty: 'last'
}
```

This code uses `record.get('last').substr(0, 1)` to get the first letter of the last name of our contact. This lets the list know where to scroll to when one of the letters on the `indexBar` component is clicked.



## Adding nested lists

The `NestedList` component automates the layout and navigation of a nested data set. This can be very useful for situations where you have a list of items and details for each item in the list. For example, let's assume we have a list of offices where each office has a set of departments, and each department is made up of a number of people.

We can initially represent this on screen as a list of offices. Clicking on an office takes you to a list of departments within that office. Clicking on a department takes you to a list of people in that department.

The first thing we need is a set of data to use with this list:

```
var data = {
    text:'Offices',
    items:[
        {
            text:'Atlanta Office',
            items:[
                {
                    text:'Marketing',
                    items:[
                        {
                            text:'David Smith',
                            leaf:true
                        },
                        {
                            text:'Alex Wallace',
                            leaf:true
                        }
                    ]
                },
                {
                    text:'Sales',
                    items:[
                        {
                            text:'Jane West',
                            leaf:true
                        },
                        {
                            text:'Mike White',
                            leaf:true
                        }
                    ]
                }
            ]
        },
        {
            text:'Athens Office',
            items:[
                {

```

```
        text:'IT',
        items:[
            {
                text:'Baron Chandler'
                leaf:true
            },
            {
                text:'Aaron Karp',
                leaf:true
            }
        ]
    },
{
    text:'Executive',
    items:[
        {
            text:'Bryan Johnson',
            leaf:true
        },
        {
            text:'John Clark',
            leaf:true
        }
    ]
}
]
}
};
```

This is a rather large and ugly-looking array of data, but it can be broken down into a few simple pieces:

- We have one main item called Offices
  - Offices has a list of two items, Atlanta Office and Athens Office
  - The two items each have two departments
  - Each department has two people

Each of the people in this list has a special attribute called `leaf`. The `leaf` attribute tells our program that it has reached the end of the nested data. Additionally, every item in our list has an attribute called `text`. This `text` attribute is part of the `fields` list in our `store`.

## *Components and Configurations*

---

We can then create our store and add our data to it:

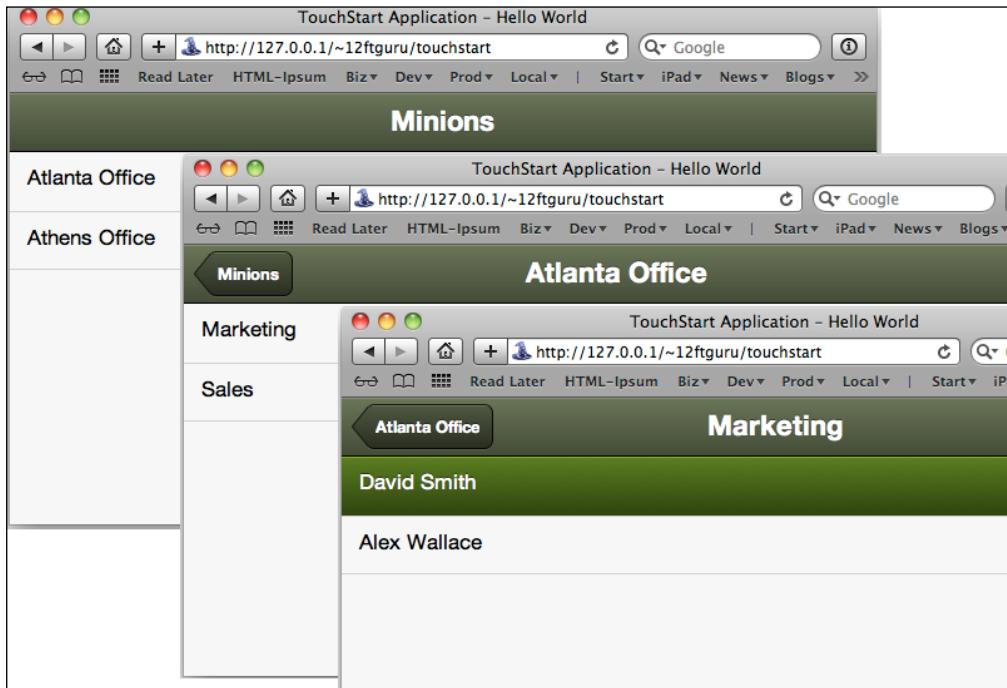
```
var store = Ext.create('Ext.data.TreeStore', {  
    root: data,  
    fields: [{name: 'text', type: 'string'}],  
    defaultRootProperty: 'items',  
    autoLoad: true  
});
```

For a `NestedList` component, we need to use a `TreeStore` class and set the `root` configuration to point to the variable `data` array we defined earlier. This will tell the store where to begin looking in the first set of items in our data.

Finally, we need to create our `NestedList`:

```
var nestedList = Ext.create('Ext.NestedList', {  
    fullscreen: true,  
    title: 'Minions',  
    displayField: 'text',  
    store: store  
});
```

We set the `NestedList` component to `fullscreen`, and also set the `title` value, tell it what field to display, and finally, we point it to our store so it can grab the data we created.



If you click on the nested list, you will notice that the click actions have been added automatically. This is the same for the upper navigation and titles.

The `NestedList` component provides a great starting point for displaying hierarchical data quickly and efficiently on a small screen.

## Finding more information with the Sencha Docs

We have covered quite a bit of information in this chapter, but it's only a fraction of the information that is available to you in the Sencha Touch API documentation.

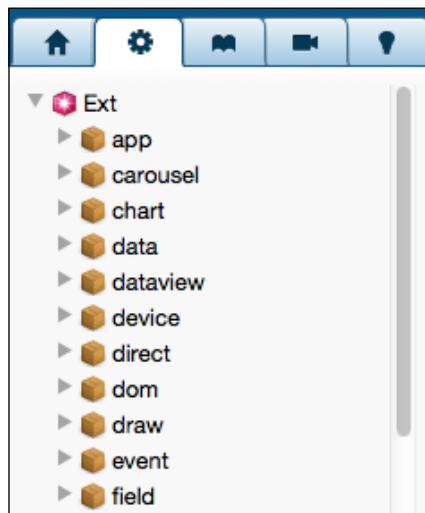
The screenshot shows the Sencha Touch 2.2.0 API Documentation. The left sidebar lists packages: Ext, App, carousel, chart, data, dataview, device, direct, dom, draw, event, field, form, layout, mixin, navigation, picker, plugin, scroll, tab, util, ActionSheet, Ajax, Anim, anims, Array, Audio. Below this are buttons for 'By Package' (selected), 'By Inheritance', and 'Show private classes'. The main content area has tabs for 'Base' and 'View'. The 'Base' tab is active, showing sections for Class System (Ext 38, Ext.Base 9, Ext.Class 9, Ext.ClassManager 3, Ext.Loader 7), Application Architecture (Ext.app.Application 24, Ext.app.Controller 40, Ext.app.Profile 2, Ext.data.ModelManager), DOM & Browser (Ext.dom.CompositeElement 38, Ext.dom.CompositeElementLite 4, Ext.dom.Element 14, Ext.DomHelper, Ext.DomQuery 1, Ext.Ajax 3), and Support (Ext.browser, Ext.feature, Ext.os, Ext.Version, Ext.Logger 3). The 'View' tab shows sections for Containers & Panels (Ext.Container 100), Draw (Ext.draw.Animator), and Styles (Global CSS 5). At the bottom, there are links for 'docs.sencha.com/touch/2.2.0/#/api/Ext.dom.CompositeElementLite' and 'Touch 2.2.0 Docs - Generated on Thu 23 May 2013 07:50:30 by JSDuck 4.10.1, Terms of Use'.

At first, the API can seem a bit overwhelming, but if you understand the organization, you can quickly find the information you need. Here are a couple of tips to get you started.

## Finding a component

The left-hand side of the API contains five tabs, which are as follows:

- The home screen, which contains general marketing information for Sencha Touch
- The API Documentation with a list of every available component
- The **Guides** section, which contains more detailed articles about the various components and their uses
- The **Videos** section, which has a number of video presentations covering topics such as layout and MVC in greater detail
- The **Examples** section, which contains numerous examples of many of the Sencha Touch components and their functionalities



If you click on the API tab, you can browse a list of components. You can also use the search field in the upper-right corner of the documentation page to find components quickly.

As you click on the items in the API list, tabs will open the main part of the screen with detailed information about the component.

## Understanding the component page

The information at the top of the individual component page provides a huge jump-start in understanding how the component works.

The screenshot shows a web browser window displaying the Sencha Touch 2.2.0 documentation. The title bar says "Touch 2.2.0 Sencha Docs". The navigation bar includes links for Home, Settings, Bookmarks, Help, and Sign in / Register. There are tabs for TreeStore, NestedList, Container, and Panel, with Panel currently selected. On the left, there's a sidebar with a tree view of the Ext API, showing categories like app, carousel, chart, data, dataview, device, direct, dom, draw, event, field, form, layout, mixin, navigation, picker, plugin, scroll, tab, and util. The "Panel" item is highlighted with a blue selection bar. The main content area is titled "Ext.tab.Panel" with the xtype: tabpanel. It contains a brief description of Tab Panels, code editor and live preview sections, and a mobile device screenshot showing a "Home Screen" tab. At the bottom, there are "Code Editor" and "Live Preview" buttons, and a "Select Code" link.

A quick scan of the component hierarchy, on the right, will tell you which other items the component inherits from. If you understand the base components, such as the container and panel, you can quickly use that knowledge to guide you through using the new component.

The title at the top also lists the `xtype` value for the component right next to it.

Underneath the title, there are a series of menus that include:

- **Config:** The initial options that are used when the component is created
- **Properties:** The information you can gather from the component after it is created
- **Methods:** The things the component knows how to do, once it's created
- **Events:** The things the component pays attention to, once it's created

- **CSS Vars:** A list of any available CSS variables you can use in styling the component (only on certain components)
- **CSS Mixins:** A list of any available mixins for the component (only on certain components)

There is also a textbox for filtering the class members, a menu for controlling what types of class members appear in the list, and a button for expanding all of the items on the page.

Most of the common components include examples at the top of the page. When viewed in a WebKit browser (Safari or Chrome), these examples include a **Live Preview / Code Editor** option that can be toggled back and forth. This will display either the component, as the user would see it, or the actual code to create the component.

As the name implies, the **Code Editor** option can actually be edited to test different configuration options. There is also a **Select Code** option, which will let you copy the code and paste it into your own applications.

These bits of information should provide you with a starting point for learning any component in the API.

## Summary

In this chapter, we started with a look at the base component called `Ext.Component`. We also looked at how components are created. We then explored the layout for containers in more detail, showing how it affects the child items inside the container.

The chapter also described a number of the more common and useful components in Sencha Touch, including: Containers, Panels, TabPanel, Carousel, FormPanel, FormItem, MessageBox, Sheet, List, and NestedList. We ended the chapter with a bit of advice on using the Sencha Touch API.

In the next chapter, we will cover the use of events in Sencha Touch.

# 5

## Events and Controllers

In the previous chapter, we took a closer look at the components available in Sencha Touch. However, simply creating components isn't enough to build an application. The components still need to communicate with each other in order to make our application do anything truly useful. This is where events and controllers come into play.

In this chapter, we will examine events and controllers in Sencha Touch: what they are, why we need them, and how they work. We will discuss how to use listeners and handlers to make your application react to the user's touch as well as to events happening behind the scenes. We will also cover some helpful concepts such as observable capture and event delegation. We will finish up with a walkthrough of the touch-specific events and a look at how you can get more information from the Sencha Touch API.

The chapter will cover the following points:

- Events
- Listeners and handlers
- Controllers
- Listener options
- Scope
- Removing events
- Handlers and buttons
- Common events
- Additional information on events

## Exploring events

As programmers, we tend to think of code as an orderly sequence of instructions, executing one line after another. It's easy to lose sight of the fact that our code really spends a lot of time sitting and waiting for the user to do something. It's waiting for the user to click on a button, open a window, or select from a list. The code is waiting for an event to occur.

Typically, an event occurs right before or immediately after a component performs a specific task. When the task is performed, the event is broadcasted to the rest of the system, where it can trigger a specific code or can be used by other components to trigger new actions.

For example, a button in Sencha Touch will trigger an event whenever it is tapped. This tap can execute code inside the button that creates a new dialog box, or a panel component can "listen" to what the button is doing and change its color when it "hears" the button trigger a `tap` event.

Given that most applications are intended for human interaction, it's safe to say that a lot of the functionality of your programs will come from responding to events. From a user's perspective, the events are what make the program actually "do" something. The program is responding to the user's request.

In addition to responding to requests, events also have an important role to play in making sure that things happen in the correct order.

## Asynchronous versus synchronous actions

Albert Einstein once remarked:

*"The only reason for time is so that everything doesn't happen at once."*

While this might seem like an offhand comment, it actually has a great deal of relevance when it comes to writing code.

As we write our code in Sencha Touch, we are directing the web browser to create and destroy components on the user's screen. The obvious limitation of this process is that we can neither manipulate a component before it gets created nor after it's destroyed.

This seems pretty straightforward at first glance. You would never write a line of code that tries to talk to a component on the line before you actually create the component, so what's the problem?

The problem has to do with asynchronous actions within the code. While most of our code will execute sequentially or in a synchronous fashion, there are a number of cases where we will need to send out a request and get back a response before we can proceed. This is especially true with web-based applications.

For example, let's say we have a line of code that builds a map using a request from Google Maps. We will need to wait until we have received a response from Google and render our map before we can start working on it. However, we don't want the rest of our application to freeze while we wait on the response. So we make an asynchronous request, one that happens in the background, while the rest of our application goes about its business.

Such asynchronous requests are called Ajax requests. **Ajax** stands for **A**synchronous **J**ava**S**cript and **X**ML. If we configure one of our buttons to send out an AJAX request, the user can still do other things while the application is waiting for a response.

On the interface side of things, you will probably want to let the user know that we made the request and are currently waiting for a response. In most cases, this means displaying a loading message or an animated graphic.

Using events in Sencha Touch, we can show the loading graphic by tying into the `beforerequest` event in the Ajax component. Since we need to know when to make the loading message disappear, our component will wait for the `requestcomplete` event from our Ajax request. Once that event fires, we can execute some code to tell the loading message to go away. We can also use the `requestexception` event to inform the user whether errors occurred during the request.

Using this type of event-driven design allows you to respond quickly to the user's actions, without making them wait for some of the more time-consuming requests your code needs to perform. You can also use the events to inform the user about the errors. The key to events is getting your other components to "listen" for the event, and then telling them how to handle the information they receive.

## Adding listeners and handlers

Every component in Sencha Touch has a long list of events that it generates. Given the number of components you will likely have in your application, you can expect a lot of chatter.

Imagine a party with 100 people, all having lots of different conversations. Now imagine trying to pick out all of the useful information from each conversation. It's impossible. You have to focus on a specific conversation in order to gather anything useful.

In the same way, components also have to be told what to listen for, or else our unfortunate partygoer would quickly be overwhelmed. Fortunately for us, there's a configuration for that.

The `listeners` configuration tells the component what events it needs to pay attention to. Listeners can be added like any other configuration option in Sencha Touch. For example, the configuration option on a panel might look like the following:

```
listeners: {
    singletap: {
        element: 'element',
        fn: function(){ Ext.Msg.alert('Single Tap'); }
    }
}
```

This configuration option tells the panel to listen for the `singletap` event when the user taps once on the element (on the inside) of the panel. When the `singletap` event occurs, we execute the function listed in the `fn` configuration option (this is typically referred to as a handler). In this case, we pop up a message box with the message alert `Single Tap`.

Note that the items in our `listeners` configuration are always part of an object (curly braces on either side), even if there is only one event we are listening for. If we were to add a second event, it would look like the following:

```
listeners: {
    singletap: {
        element: 'element',
        fn: function(){ Ext.Msg.alert('Single Tap'); }
    },
    hide: {
        fn: function(){ this.destroy(); }
    }
}
```



If the event has no other properties, you can also shorten the event declaration like this: `hide: function(){ this.destroy(); }`

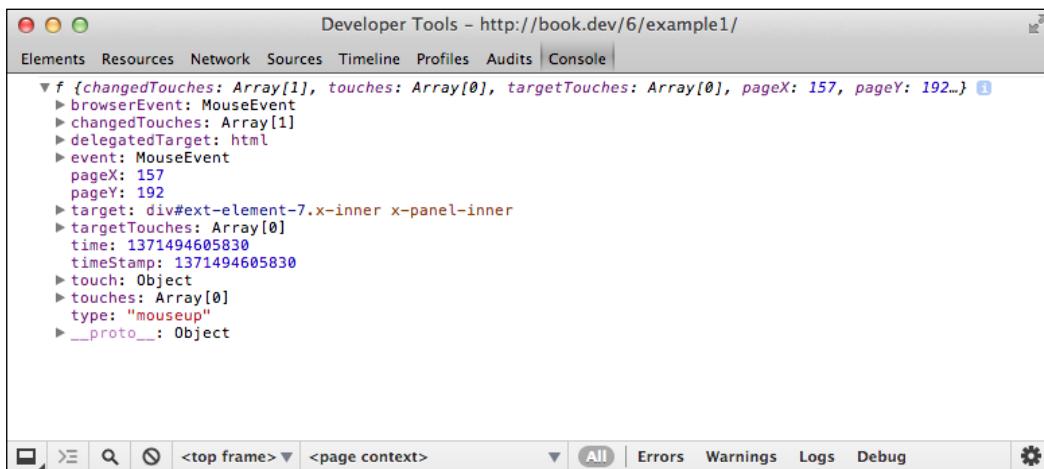
We can also get information back from the listener and use it in our handler functions. For example, the `singletap` event sends back the event object, the DOM element that was clicked, and the `listener` object itself, if we have the following listener on a panel:

```

listeners: {
    singletap: {
        element: 'element',
        fn: function(event, div, listener) {
            console.log(event, div, listener);
        }
    }
}

```

When the user taps inside the panel, we will get a view on our console, something similar to this:



### Arguments for events

You will notice that certain values are passed to our event by default. These default values can be found in the Sencha Touch API event documentation for each component at <http://docs.sencha.com/touch/2.2.1/>.

Each event will have its own default values. Select a component from the Sencha API documentation, and then click on Events at the top of the page to see a list of all events of the component. The description of each event will include its default arguments.



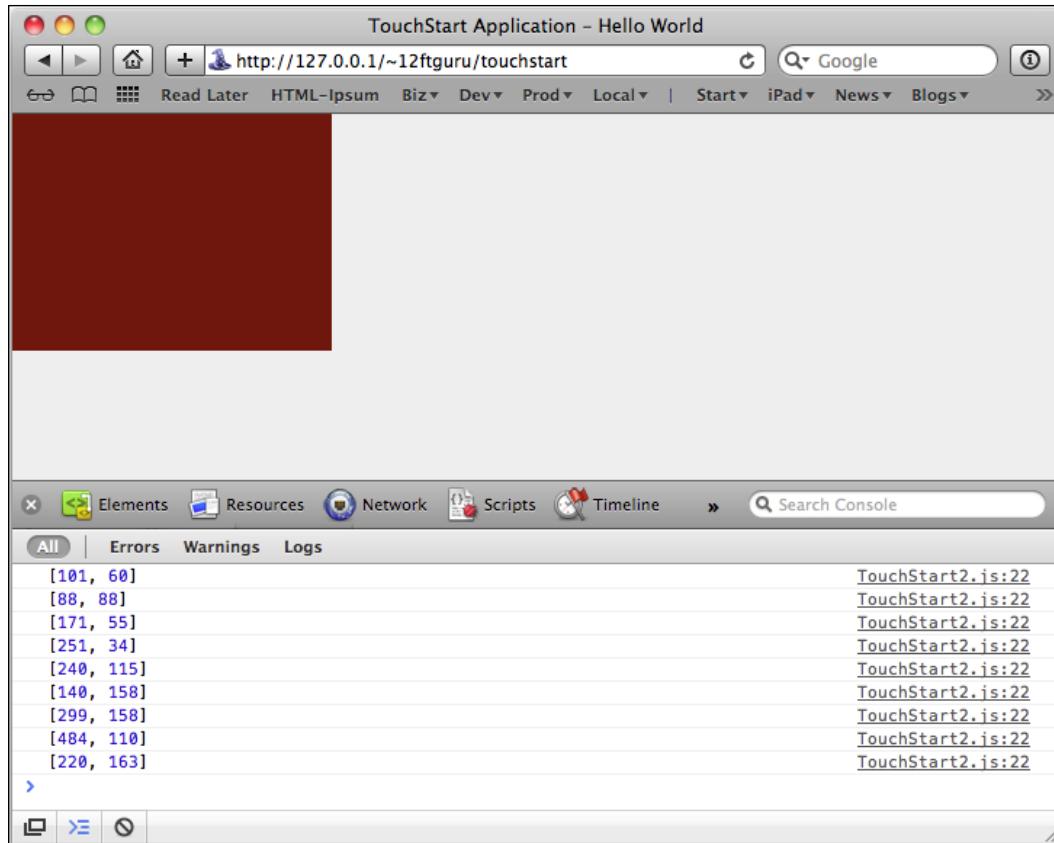
As you can see from the console, our event object contains a Unix `timeStamp` for when the tap occurred, the `pageX` and `pageY` coordinates of the tap itself, as well as the entire content of the `div` tag that was tapped. You may have also noticed that our `tap` event is referred to as a `mouseup` event in our debug output. In Sencha Touch, the `singletap` and `mouseup` events are aliases of one another. This preserves compatibility between the desktop browser's traditional `mouseup` event and the mobile browser's `singletap` event.

We can use all of this information inside our function.

For this example, we will create a simple panel with a red container. Our `singletap` listener will change the size of the red box to match where we tap on the screen, as shown in the following code snippet:

```
Ext.application({
    name: 'TouchStart',
    launch: function() {
        var eventPanel = Ext.create('Ext.Panel', {
            fullscreen: true,
            layout: 'auto',
            items: [{
                xtype: 'container',
                width: 40,
                height: 40,
                id: 'tapTarget',
                style: 'background-color: #800000;'
            }],
            listeners: {
                singletap: {
                    element: 'element',
                    fn: function(event, div, listener) {
                        var cmp = Ext.getCmp('tapTarget');
                        cmp.setWidth(event.pageX);
                        cmp.setHeight(event.pageY);
                        console.log(event.pageX, event.pageY);
                    }
                }
            }
        });
        Ext.Viewport.add(eventPanel);
    }
});
```

If we run this code with the console open, we can see that the x and y coordinates of the location where we tap will appear in the console. Our box also grows or shrinks to match these values.



As you can see in the preceding code, we listen for the `tap` event. We then grab the container component using `Ext.getCmp('tapTarget')`, and change the size of the red box based on the value we got back from the `tap` event:

```

singletap: {
    element: 'element',
    fn: function(event, div, listener) {
        var cmp = Ext.getCmp('tapTarget');
        cmp.setWidth(event.pageX);
        cmp.setHeight(event.pageY);
    }
}

```

This is a very simple example of using events in Sencha Touch. However, most of our applications will typically do more than one simple thing. We can also use IDs and grab them with `Ext.getCmp()`. In a large application it gets very easy to accidentally create components with the same ID, or create a component with an ID that is already used by Sencha Touch. This will typically lead to the spiraling death of your application and much hair pulling.

 As the best practice, it is a good idea to avoid IDs for addressing components. In the next few sections, we will begin showing you more reliable ways to address our various components.

If we are going to build anything more complex than this kind of "one-trick pony", we probably want to start thinking about splitting our events and actions out into a proper controller, and find a better way to address our different components.

## Controllers

In *Chapter 3, Styling the User Interface*, we talked a bit about the **Model View Controller (MVC)** architecture. This architecture splits our files out into datafiles (`Models` and `Stores`), interface files (`Views`), and files that handle functionality (`Controllers`). In this section, we will focus on the controller part of the MVC.

At its most basic level, a controller assigns listeners and actions within the application. Unlike our previous example, where the single component is responsible for handling an event, the controller will handle the events for every component in our application.

This division of labor offers a few different advantages when creating an application; they are as follows:

- The code is easier to navigate when we know that our functions are all in the controller and separate from the display logic.
- The controller offers an easier communications layer between the various display components in our application.
- Controllers can be divided into separate files based on the functionality. For example, we can have a user controller that handles events and listeners for our user data and a separate company controller that handles events and listeners for our company data. This means that if a form for saving new users is not working correctly, we know which file to look at to try and find out the problem.

Let's take a look at an example to see what we are talking about. We will start with the basic starter application we generate from Sencha Cmd using the following command line:

```
sencha generate app TouchStart /Path/to/Save/Application
```



The path will vary based on your setup, but this will give us the basic application we will add our controller to.



For a review of the basics of Sencha Cmd and MVC, see *Chapter 3, Styling the User Interface*.



If we look in the `app/controller` folder of our newly generated application, we will see that it is empty. Let's start by creating a `Main.js` file here. Inside the new file, we will add:

```
Ext.define('TouchStart.controller.Main', {  
    extend: 'Ext.app.Controller',  
  
}) ;
```

This extends the base `Ext.app.Controller` component, but doesn't really do anything else. Our controller needs to understand a few basic things in order to function correctly; they are as follows:

- What pieces of the application does the controller control?
- What component events should it be listening to?
- What should it do when one of those events is fired?

The first part of this puzzle is handled by the references (`refs`).

## Refs and control

The `refs` section uses the `ComponentQuery` syntax to create internal references to the components in your application. The `ComponentQuery` syntax allows us to find components based on ID, xtype, and any other configuration option.

For example, we have a `Main.js` file in our `app/view` directory (it was automatically generated by Sencha Cmd). The view component has an `xtype` value of `main`. We can add this view file to our controller as follows:

```
Ext.define('TouchStart.controller.Main', {
    extend: 'Ext.app.Controller',
    views: ['TouchStart.views.Main'],
    config: {
        refs: {
            mainView: 'main'
        }
    }
});
```

This tells our controller that it has control of the `TouchStart.views.Main` view file and that we will be referencing that particular component with a shorthand `m`(this is an utterly arbitrary name of our choice). By creating this reference, we automatically create a getter function for the component. This means that when we need to refer to this component elsewhere in the controller, for example if we need to add a new tab to our tab panel, we can just grab the component using `this.getMainView()`.



This is another spot where capitalization can attack without warning. You will notice that despite the fact that we named our reference with a lowercase `m`, the get function uses an uppercase `M`. If we had named our reference `mainPanel`, the get function would be `this.getMainPanel()`. The first letter will always be in uppercase.

Let's add some elements to our basic application to see exactly how this works. First we need to add a button to the `Main.js` view file. Inside our first panel (the one with the title), modify the items section as follows to add a button:

```
items: [{  
    docked: 'top',  
    xtype: 'titlebar',  
    title: 'Welcome to Sencha Touch 2',  
    items: [  
        {  
            text: 'Add Tab',  
            action: 'addtab'  
        }  
    ]  
}]
```

Notice that we don't add a handler here this time, but we do have an action of `addtab`, which we will use to reference the button inside our controller:



Back in our `Main.js` file located at `app/controller/`, we will add a `refs` and `control` section as follows:

```
Ext.define('TouchStart.controller.Main', {  
    extend: 'Ext.app.Controller',  
    config: {  
        views: ['TouchStart.view.Main'],  
        refs: {  
            m: 'main',  
            addBtn: 'button[action=addtab]'  
        },  
    },
```

```
control: {
  addBtn: {
    tap: 'addNewTab'
  }
}
});
```

We now have a new reference for our button:

```
addBtn: 'button[action=addtab]'
```



It should be noted that our action configuration on the button is totally arbitrary. We could call it `myPurposeInLife: 'addtab'` and it would make no difference to the component itself. We would have simply referenced the button as `addBtn: 'button [myPurposeInLife = addtab]'` in that case. The term `action` is typically used by convention but it is not a default configuration option for the button. It is simply a value that we will use to find the button later on in our controller using `ComponentQuery`.

Now that we have our reference, we can use `addBtn` when we set our control. This control section is where we set a listener for this particular button:

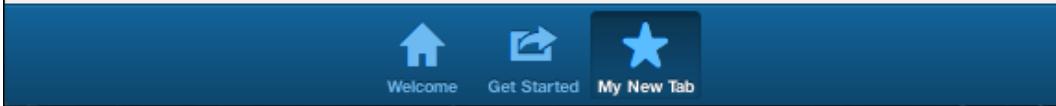
```
control: {
  addBtn: {
    tap: 'addNewTab'
  }
}
```

This control section says that we want our controller to listen to the `tap` event on the `addBtn` button and fire the `addNewTab` function when the user taps the button. Next, we need to add this `addNewTab` function at the bottom of our controller just after the `config` section (don't forget to put a comma between the end of the `config` section and the new function), as shown in the following code snippet:

```
addNewTab: function() {
  this.getMainView().add({
    title: 'My New Tab',
    iconCls: 'star',
    html: 'Some words of wisdom...'
  });
}
```

This function uses our `this.getMainView()` function to grab our main tab panel and add a new tab to it. When we click on the button now, we should get a new tab with a star icon and our HTML text:

Some words of wisdom...



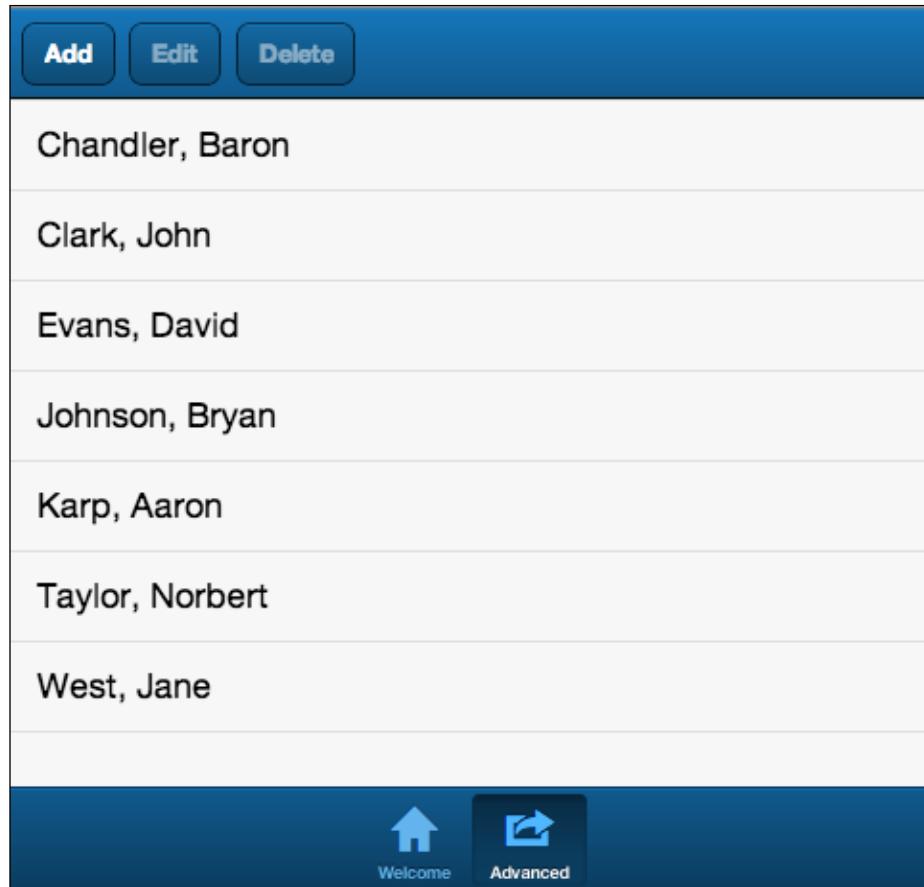
Each controller file can contain any number of views, references, and functions. However, it's often best to split your controllers into separate files based on the kind of data they handle (a controller for users, one for companies, another for messages, and so on). This code organization is entirely up to the coder, but it helps a great deal when trying to hunt down problems.

## **Referencing multiple items with ComponentQuery**

As we have seen in our previous examples, the `refs` section gives us shorthand reference names for our components and the `control` section lets us assign listeners and functions to our components. While we can use the control section to assign a single function to multiple components, the items we include in our `refs` section can only be singular. We cannot create a single reference for multiple components in our `refs` section.

However, we can get around this by using `Ext.ComponentQuery`.

To demonstrate this, let's take a look at a real-world example: a list of items, with buttons for adding, editing, and deleting. The **Add** button should always be enabled but the **Edit** and **Delete** buttons should only be active when something in the list is selected.



We will create our list in a separate file called `PersonList.js` in the `view` folder, as shown in the following code snippet:

```
Ext.define('TouchStart.view.PersonList', {  
    extend: 'Ext.dataview.List',  
    xtype: 'personlist',  
    config: {
```

```

itemTpl: '{last}, {first}',
store: Ext.create('Ext.data.Store', {
    sorters: 'last',
    autoLoad: true,
    fields: [
        {name: 'first', type: 'string'},
        {name: 'last', type: 'string'}
    ],
    data: [
        {first: 'Aaron', last: 'Karp'},
        {first: 'Baron', last: 'Chandler'},
        {first: 'Bryan', last: 'Johnson'},
        {first: 'David', last: 'Evans'},
        {first: 'John', last: 'Clark'},
        {first: 'Norbert', last: 'Taylor'},
        {first: 'Jane', last: 'West'}
    ]
})
});
}
);

```

This is similar to the list we created in *Chapter 5, Events and Controllers*, except we have made it into a separate view component by using `Ext.define` and extending the `Ext.dataview.List` object. We could have simply made it part of our `Main.js` view file, but splitting it out allows us to define a custom xtype of `personlist`, which will make it easier to reference later on in our controller.



For the sake of simplicity, we have left `store` as part of our view instead of splitting it out into its own separate file in the `store` directory. We will talk about how that gets done in *Chapter 7, Getting the Data Out*, and *Chapter 8, Creating the Flickr Finder Application*, where we cover stores and models.

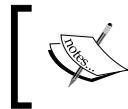
Now that we have our `personlist` view, we need to add it into our `Main.js` view file. Let's replace the second panel in the `Main.js` file (the one with the video link in it). The new panel will look like this:

```
{  
    title: 'Advanced',  
    iconCls: 'action',  
    layout: 'fit',  
    items: [{  
        docked: 'top',  
        xtype: 'toolbar',  
        items: [  
            {  
                text: 'Add',  
                action: 'additem'  
            },  
            {  
                text: 'Edit',  
                action: 'edititem',  
                enableOnSelection: true,  
                disabled: true  
            },  
            {  
                text: 'Delete',  
                action: 'deleteitem',  
                enableOnSelection: true,  
                disabled: true  
            }  
        ]  
    },  
    { xtype: 'personlist'  
    ]  
}
```

This code creates a new panel with a `fit` layout and two items inside it. The first item is a toolbar docked at the top of the panel. The second item (down at the very bottom) is our `personlist` component.

The toolbar has its own items, which consist of three buttons with the text: `Add`, `Edit`, and `Delete`. Each of these buttons has its own individual `action` configuration, and the `Edit` and `Delete` buttons have an extra configuration:

```
enableOnSelection: true
```



Note that like `action`, the `enableOnSelection` configuration is an arbitrary value and not a default configuration for a button component.

The individual `action` configurations will let us assign functions to each of those buttons. The shared `enableOnSelection` configuration will allow us to grab both the `Edit` and `Delete` buttons with a single reference. Let's head back to our `Main.js` controller to see how this works.

The first thing we want to do is let the `Main.js` controller know that it is responsible for our new `personlist` view. We do this by adding it to the list of views in our controller, as shown in the following code snippet:

```
views: ['TouchStart.view.Main', 'TouchStart.view.PersonList']
```

Next, we need to create our references in the `refs` section, as shown in the following code snippet:

```
refs: {
  mainView: 'main',
  addBtn: 'button[action=addtab]',
  addItem: 'button[action=additem]',
  editItem: 'button[action=edititem]',
  deleteItem: 'button[action=deleteitem]',
  personList: 'personlist'
}
```

We will then modify our `control` section to look as follows:

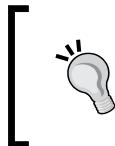
```
control: {
  addBtn: {
    tap: 'addNewTab'
  },
  personList: {
    select: 'enableItemButtons'
  },
  addItem: {
    tap: 'tempFunction'
  },
  editItem: {
    tap: 'tempFunction'
  },
  deleteItem: {
    tap: 'tempFunction'
  }
}
```

Here we have set our `personList` component to listen for the `select` event and fire the `enableItemButtons` function when the event occurs. We have also assigned a single `tempFunction` function to the `tap` event of our three buttons.

Our `tempFunction` is added after the existing `addNewTab` function and looks as follows:

```
tempFunction:function () {  
    console.log(arguments);  
}
```

This is just a placeholder function for demonstration purposes (we will cover adding, editing, and deleting actions in more detail in *Chapter 7, Getting the Data Out*, and *Chapter 8, Creating the Flickr Finder Application*). For now, this temporary function just logs the arguments that are sent to it.



The `arguments` variable is a special variable in JavaScript and it contains many variables that are passed to the function. It's great for use with console logs where you might not be clear on which variables your function is receiving, their order, or their format.

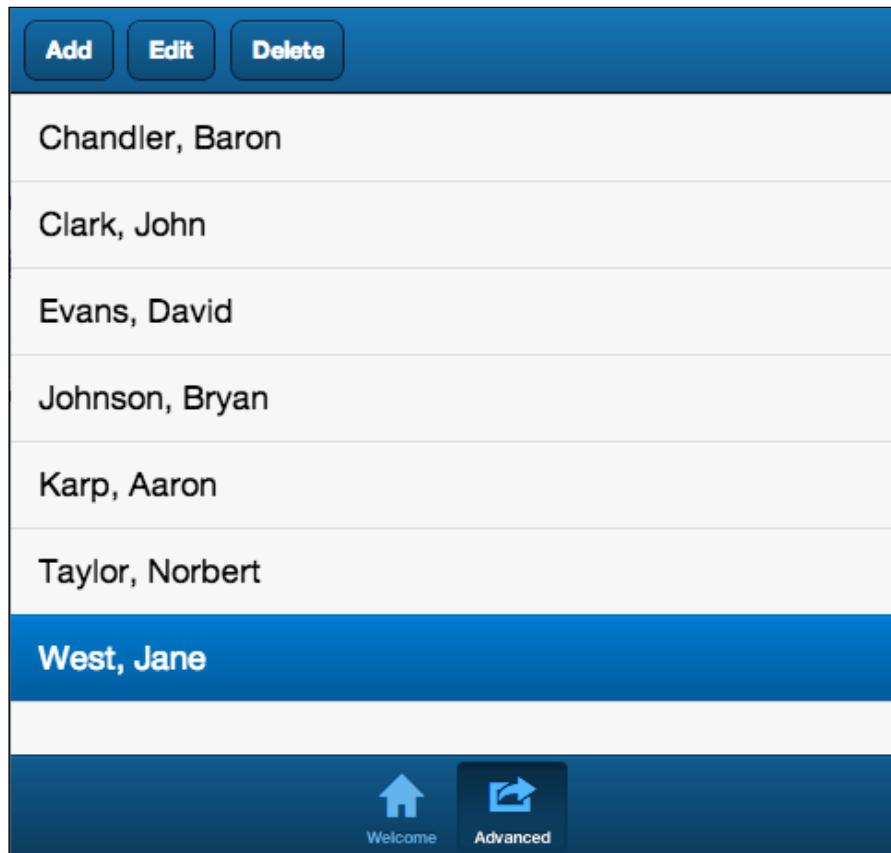
The second function will handle our list selection:

```
enableItemButtons:function () {  
    var disabledItemButtons = Ext.ComponentQuery.query('button[enableOnSelection]');  
    Ext.each(disabledItemButtons, function(button) {  
        button.enable();  
    });  
}
```

As we noted previously, we cannot simply create a `refs` listing for our two disabled buttons. If we had tried `myButtons: 'button[enableOnSelection]'` in our `refs` section, we would only get back the first one.

However, we can use the exact same selector with `Ext.ComponentQuery.query('button[enableOnSelection]')`; and get back both of our buttons as an array of button objects. We can then use `Ext.each` to step through each of the buttons in turn and run a function on them.

In this case, we just run `button.enable();` on each button. Now when an item is selected in the list, both of our buttons will be enabled.



By using `Ext.ComponentQuery`, a single event can easily affect multiple components based on their attributes.

## Getting more out of events

Now that we have seen how events and controllers fit together, we need to take a look at the other uses and options available to us with events.

## Custom events

While Sencha Touch components respond to a large number of events, it can sometimes be helpful to fire custom events within your application.

For example, you could fire a custom event called `vikinginvasion`, which might trigger additional actions within your application. For this example, we will assume we have a component called `cmp`. We can fire events with this component simply by calling:

```
cmp.fireEvent('vikinginvasion');
```

You can then add a listener to any component in the `control` section of your controller for `vikinginvasion`, along with a function to handle the event. If we want to add the listener for our custom event to a component called `trebuchet`, it might look as follows:

```
control: {
    trebuchet: {
        vikinginvasion: 'fireAtWill'
    }
}
```

You can also check a component to see if it has a specific listener, using the `hasListener()` method:

```
if(this.getTrebuchet.hasListener('vikinginvasion')) {
    console.log('Component is alert for invasion');
} else {
    console.log('Component is asleep at its post');
}
```

There are also a number of helpful options you can use to control how listeners will check for events.

## Exploring listener options

For the most part, listeners can simply be configured with the event name, handler, and scope, but sometimes you need a bit more control. Sencha Touch provides a number of helpful options to modify how the listener works; they are:

- `delay`: This will delay the handler from acting after the event is fired. It is given in milliseconds.
- `single`: This provides a one-shot handler that executes after the next event fires and then removes itself.

- **buffer**: This causes the handler to be scheduled to run as part of an `Ext.util.DelayedTask` component. This means that if an event is fired, we wait a certain amount of time before we execute the handler. If the same event fires again within our delay time, we reset the timer before executing the handler (only once). This can be useful for monitoring the change event in a textfield – wait 300 ms after the user's last change before firing the function for the event.
- **element**: This allows us to specify a specific element within the component. For example, we can specify a body within a panel for a `tap` event. This would ignore taps to the docked items and only listen for a tap on the body of the panel.
- **target**: This will limit the listener to the events coming from the target and it will ignore the same event coming from any of its children.

Using the different listener options, the code would look something like the following:

```
this.getTrebuchet.on('vikinginvasion', this.handleInvasion, this, {  
    single: true,  
    delay: 100  
});
```

This example would add a listener for `vikinginvasion` and execute a function called `handleInvasion` in this scope. The handler would only execute once, after a 100-millisecond delay. It would then remove itself from the component.

If you are inside a controller, you can accomplish the same thing in the `control` section like this:

```
control:{  
    Trebuchet:{  
        vikinginvasion: {  
            fn: this.handleInvasion,  
            single: true,  
            delay: 100  
        }  
    }  
}
```

Since we are setting options on the event listener for `vikinginvasion`, it becomes its own configuration object. In turn, our `handleInvasion` function becomes a configuration option called `fn`.

This basic list of configuration options gives you quite a bit of flexibility when adding listeners. However, there is one additional configuration option available in listeners that will require a bit more explanation. It's called `scope`.

## Taking a closer look at scope

Within your handler function is a special variable called `this`. Usually, `this` refers to the component that fired the event, in which case, the scope would typically be set to `scope: this`. However, it's possible to specify a different value for `scope` in your listener configuration:

```
Ext.application({
    name: 'TouchStart',
    launch: function() {
        var btn = Ext.create('Ext.Button', {
            xtype: 'button',
            centered: true,
            text: 'Click me'
        });
        var Mainpanel = Ext.create('Ext.Panel', {
            html: 'Panel HTML'
        });
        btn.on({
            painted: {
                fn: function() {
                    console.log('This should show our button %o', this)
                }
            },
            tap: {
                scope: Mainpanel,
                fn: function() {
                    console.log('This should show our main panel %o', this)
                }
            }
        });
        Ext.Viewport.add(btn);
        Ext.Viewport.add(Mainpanel);
    }
});
```

Here we create a button called `btn` and a panel called `Mainpanel`. We then attach two listeners. The first one is on the button's `painted` event. This event fires as soon as the button is "painted" (appears) on screen. In this case, the function's scope is the button as we would expect by default.

The second is on the `tap` event for the `button`. The `tap` event has a scope of `Mainpanel`. This means that even though the listener is attached to the button, the function treats `this` as the `Mainpanel` component and not the button.

While `scope` may be a hard concept to grasp, it is a very useful part of the listener configuration.

## Removing listeners

Normally, listeners are removed automatically when a component is destroyed. However, sometimes you will want to remove the listener before the component is destroyed. To do so, you'll need a reference to the handler function you created the listener with.

So far, we've been using anonymous functions to create our listeners, but if we're going to remove the listener, we have to do it a bit differently:

```
var myPanel = Ext.create('Ext.Panel', { ... });

var myHandler = function() {
    console.log('myHandler called.');
};

myPanel.on('click', myHandler);
```

This can be a good practice as it allows you to define the handler functions once and re-use them wherever you need them. It also allows you to remove the handler later:

```
myPanel.removeListener('click', myHandler);
```



In Sencha parlance, `on()` is an alias for `addListener()` and `un()` is an alias for `removeListener()`, meaning that they do the exact same thing. Feel free to use whichever you prefer, when dealing with events.

It should also be noted that listeners that are added as part of the `control` section of a controller are never removed.

## Using handlers and buttons

As you might have noticed from some of our previous code, buttons have a default configuration called `handler`. This is because the purpose of a button is generally to be clicked or tapped. The `handler` configuration is just useful shorthand for adding the `tap` listener. As such, the following two pieces of code do exactly the same thing:

```
var button = Ext.create('Ext.Button', {
    text: 'press me',
    handler: function() {
        this.setText('Pressed');
    }
})
var button = Ext.create('Ext.Button', {
    text: 'press me',
    listener: {
        tap: {
            fn: function() {
                this.setText('Pressed');
            }
        }
    }
});
```

Next we will take a look at some common events.

## Exploring common events

Let's take a look at our old friend `Ext.Component` and see some of the common events available to us. Remember, since most of our components will inherit from `Ext.Component`, these events will be common across most of the components we use. The first of these events revolves around the creation of the component.

When the web browser executes your Sencha Touch code, it writes the components into the web page as a series of `div`, `span`, and other standard HTML tags. These elements are also linked to the code within Sencha Touch that standardizes the look and functionality of the component for all supported web browsers. This process is commonly referred to as rendering the component. The event that governs this rendering in Sencha Touch is called `painted`.

Some other common events include:

- `show`: This is fired when the `show` method is used on the component
- `hide`: This is fired when the `hide` method is used on the component

- `destroy`: This is fired when the component is destroyed
- `disabledchange`: This is fired when the `disabled` configuration is changed by `setDisabled`
- `widthchange`: This is fired when `setWidth` is called on the component
- `heightchange`: This is fired when `setHeight` is called on the component

These events provide a way to base the actions of your code on what is being done by, or done to, your components.



Every event whose name ends with `changed` is fired as a result of a config option that has changed; for example, `setWidth`, `setHeight`, and `setTop`. Although listening to these events is like listening to any other event, it is useful to know this convention.

Each component will also have some specific events associated with it. For a list of these events, please consult the documentation available at <http://docs.sencha.com/touch/2.2.1>. Select a component from the list on the left-hand side and click on the **Events** button at the top of the page.

## Additional information on events

The best place to get information about events is the Sencha Docs at <http://docs.sencha.com/touch/2.2.1>. Select a component in the list on the left, and look for the **Events** button at the top. You can click on **Events** to go to the beginning of the section or hover your mouse pointer to see the full list of events and select a specific event from that list.

Clicking on the down arrow next to the event will display a list of parameters for the event and any available examples on how the event can be used.

Another good place to find out about touch-specific events is the Kitchen Sink example application (<http://dev.sencha.com/deploy/touch/examples/kitchensink/>). Inside the application is a **Touch Events** section. This section allows you to tap or click on the screen to see which events are generated from the different taps and gestures.

The WebKit team at Sencha Touch has also created an event recorder for Android. You can find more information at <http://www.sencha.com/blog/event-recorder-for-android-web-applications/>.

## Summary

In this chapter, we have covered a basic overview of events, and how to use listeners and handlers to get your program to respond to these events. We took a deeper look at controllers and how they use references and the control section to attach listeners to components. We covered `Ext.ComponentQuery()` for grabbing components inside the event handlers. We talked about custom events, handlers in buttons, and listed some of the common events.

In the next chapter, we will cover how to get and store data in Sencha Touch, using JSON, data stores, models, and forms.

# 6

## Getting the Data In

One of the key aspects of any application is the handling of data – getting data into the application, so that you can manipulate and store it, and then get it out again for display. We will spend the next two chapters covering data handling in Sencha Touch. This first chapter on data will focus on getting data into your application.

We will start with a discussion of the data models that are used to describe your data. We will then discuss the readers that gather the data and the stores used to hold the data for use in our application. Once we have a grasp on where the data goes, we will cover how to use forms to get it there. We will look at how to validate your data and provide you with some examples of form submission. We will finish up with a look at getting the data back into a form for editing. This will serve as our starting point for the next chapter on data, which will cover getting data back for display.

This chapter covers the following topics:

- Data models
- Data formats
- Data stores
- Using forms and data stores

## Models

The first step in working with data in a Sencha Touch application is to create a model of the data. If you are used to database-driven applications, it's helpful to think of the model as being a database schema; it's a construct that defines the data we are going to store, including the data type, validations, and structure. This provides the rest of our application with a common map for understanding the data being passed back and forth.

In Sencha Touch 2, the model can also be used to hold the information for a single data record. This means that we can create, read, update, and delete the single record using functions that are already built in to the Sencha Touch `Ext.data.Model` component.

## The basic model

At its most basic, the model describes the data fields using `Ext.define()` as follows:

```
Ext.define('User', {
    extend: 'Ext.data.Model',
    config: {
        fields: [
            {name: 'firstname', type: 'string'},
            {name: 'lastname', type: 'string'},
            {name: 'username', type: 'string'},
            {name: 'age', type: 'int'},
            {name: 'email', type: 'string'},
            {name: 'active', type: 'boolean', defaultValue: true},
        ]
    }
})
```

The first line declares that we have named our new model `User` and that we are extending the default `Ext.data.Model`. We set up the model's configuration options inside the `config` section.



The model setup has changed a bit in Version 2. We now use `Ext.define` and `extend` instead of creating things through the old Model Manager. We also wrap the model's options in a `config` section. Outside the `extend` setting, the rest of your model options should be wrapped in this `config` section.

Inside the `config` section, we describe our data fields as an array of `fields` including `name`, `type`, and optional `defaultValue` fields. The `name` field is simply how we want to refer to the data in our code. The valid values for `types` are:

- `auto`: This is a default value that just accepts the raw data without conversion
- `string`: This converts the data into a string
- `int`: This converts the data into an integer
- `float`: This converts the data into a floating point integer
- `boolean`: This converts the data into a true or false Boolean value
- `date`: This converts the data into a JavaScript `Date` object

The `defaultValue` field can be used to set a standard value to be used if no data is received for that field. In our example, we set the value of `active` to `true`. We can use this when creating a new user instance with `Ext.create()`:

```
var newUser = Ext.create('User', {
    firstname: 'Nigel',
    lastname: 'Tufnel',
    username: 'goes211',
    age: 39,
    email: 'nigel@spinaltap.com'
});
```

Notice that we did not provide a value for `active` in our new user instance, so it just uses our `defaultValue` field from the model definition. This can also help when the user forgets to enter a value. We can also double-check the information our user enters by using validations.

## Model validations

Model validations ensure that we are getting the data we think we are getting. These validations serve two functions. The first is to provide the guidelines for how data is entered. For example, we would typically want a username to consist only of letters and numbers; the validation can enforce this constraint and inform the user when they use the wrong character.

The second is security; malicious users can also use the form field to send information that might potentially be harmful to our database. For example, sending `DELETE * FROM users;` as your username can cause problems if the database is not properly secured. It is always a good idea to validate data, just in case.

We can declare validations as part of our data model in much the same way that we declare our fields. For example, we can add the following code to our `User` model:

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

```
validations: [
  {type: 'presence', field: 'age'},
  {type: 'exclusion', field: 'username', list: ['Admin', 'Root']},
  {type: 'length', field: 'username', min: 3},
  {type: 'format', field: 'username', matcher: /([a-z]+)[0-9]{2,3}/}
]
}
```

In our example, we have added four validations. The first one tests the presence of an age value. If there is no value for age, we get an error. The second validator, exclusion, tests for things we don't want to see as a value for this field. In this case, we have a list of two items for the username that we don't want to see: Admin and Root. The third validator tests to make sure that our value for the username is at least three characters long. The final validator checks the format of our username using a regular expression.

### Regular expressions

**Regular expressions**, also called **regexes** or **regexp**s, are an extremely powerful tool for matching the structure of a string. You can use RegEx to search for particular characters, words, or patterns within a string. A discussion of regular expressions would require its own book, but there are a number of good resources available online.



Good tutorials are available at:

<http://www.zytrax.com/tech/web/regex.htm>.

A searchable database of regular expressions can be found at:

<http://regexlib.com>.

A wonderful regular expression tester is also available at:

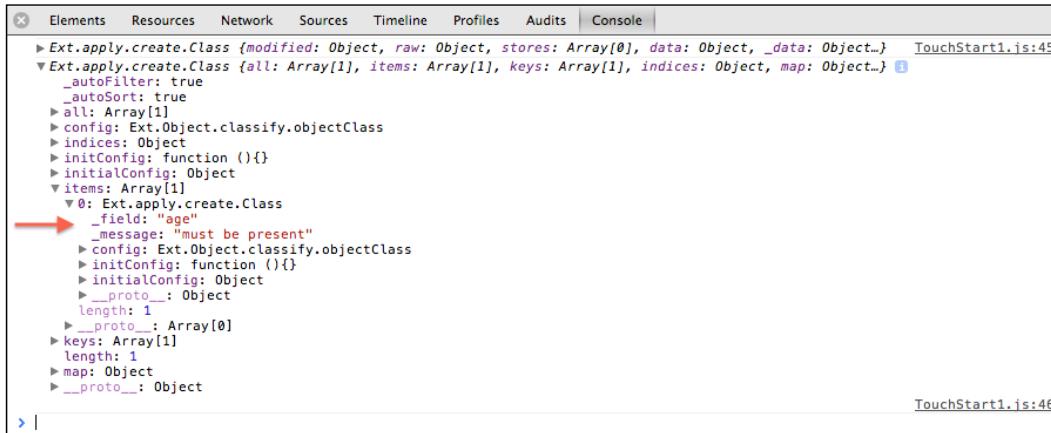
<http://www.rexv.org/>.

We can test our validations by using the validate method on our new User instance:

```
var newUser = Ext.create('User', {
  firstname: 'Nigel',
  lastname: 'Tufnel',
  username: 'goes211',
  email: 'nigel@spinaltap.com'
});

var errors = newUser.validate();
console.log(errors);
```

Note that we intentionally dropped the `age` field this time, to give us an error. If we take a look at our console, we can see the `Ext.data.Errors` object that we get back, as shown in the following screenshot:



```

Elements Resources Network Sources Timeline Profiles Audits Console
▶ Ext.apply.create.Class {modified: Object, raw: Object, stores: Array[0], data: Object, _data: Object...} TouchStart1.js:45
  ▶ Ext.apply.create.Class {all: Array[1], items: Array[1], keys: Array[1], indices: Object, map: Object...} ⓘ
    _autoFilter: true
    _autoSort: true
    all: Array[1]
    config: Ext.Object.classify.objectClass
    indices: Object
    initConfig: function(){}
    initialConfig: Object
    items: Array[1]
      ▶ 0: Ext.apply.create.Class
        _field: "age"
        _message: "must be present"
        config: Ext.Object.classify.objectClass
        initConfig: function(){}
        initialConfig: Object
        __proto__: Object
      length: 1
      __proto__: Array[0]
    keys: Array[1]
    length: 1
    map: Object
    __proto__: Object
  > | TouchStart1.js:46
  
```

This is the console output for our `errors` object. The `errors` object includes a method called `isValid()`, which will return a `true` or `false` value. We can use this method to test for errors and return a message to the user, using something as follows:

```

if(!errors.isValid()) {
  alert("The field: "+errors.items[0].getField()+" returned an
error: "+errors.items[0].getMessage());
}
  
```

Here, we test `errors` to see if it is valid and if not, display the information from the first error. We then use `getField()` and `getMessage()` to display in our alert to the user. This detailed error information is included in the `items` list of the `errors` object. In practical usage there could be more than one error, so we would need to loop through the `items` list to grab all of the errors.

We can also change the default error message by setting additional configuration options on the validations for:

- `exclusionMessage`: This is used when we get an excluded value for a field
- `formatMessage`: This is used when we get an improperly formatted value for a field
- `inclusionMessage`: This is used when we do not get an included value for a field

- `lengthMessage`: This is used when we get a value for a field that does not meet our required length
- `presenceMessage`: This is used when we do not reserve a required value for a field

Customizing these errors will help the user understand exactly what went wrong and what needs to be done to correct the problem.

## Model methods

Our models can also contain methods that can be called on any instance of our model. For example, we can add a method called `deactivate` to our model by adding the following to our `User` model after the `fields` list:

```
deactivate: function() {  
    if(this.get('active')) {  
        this.set('active', false);  
    }  
}
```

This function tests to see if our current value of `active` is `true`. If it is, we set it to `false`. Once we create `newUser` as we did previously, we can call the function as follows:

```
newUser.deactivate();
```

These model methods provide a great way to implement common functions in your model.

### CRUD

While model methods might look like a good place for adding functions to save our model, you really don't need to. These types of functions—Create, Read, Update, and Destroy—are often referred to by the unattractive acronym **CRUD**, and they are handled automatically by Sencha Touch. We will go over these functions a bit later in this chapter.



Now that we have our model's fields, validations, and functions defined, we need a way to pass data to and from the model for the storing and retrieving of our users. This is where the proxy and reader come in.

## Proxies and readers

In the model, the proxy and reader form a partnership to store and retrieve data to be used by the model. The proxy tells a model where its data will be stored, and the reader tells the model what format is being used to store the data.

There are two main types of proxies: local and remote. A local proxy stores its data locally on the device with one of the two proxy types:

- `LocalStorageProxy`: This saves the data to the local storage via the browser. This data is persistent across sessions, unless deleted by the user.
- `MemoryProxy`: This holds the data in the local memory. When the page is refreshed, the data is deleted.

The remote proxy has two basic types:

- `AjaxProxy`: This sends requests to a server within the current domain.
- `JsonP`: This sends requests to a server on a different domain (this was called a scripttag proxy in the previous versions).

Additionally, there are a few specialized proxies, which include:

- `Direct`: This is a proprietary Sencha technology which, like Ajax, allows asynchronous communication with a remote server. However, unlike Ajax, Direct does not need to keep a socket open to the remote server waiting for a response. This makes it ideal for any process that may require a long response delay from the server. For more information on Direct, go to:  
<http://docs.sencha.com/touch/2.2.0/#!/api/Ext.direct.Manager>.
- `Rest`: The Rest proxy takes the basic proxy functions (Create, Read, Edit, and Delete) and maps these to HTTP request types (POST, GET, PUT, and DELETE, respectively). This type of communication is very common in commercial APIs. For more information on the rest of the proxies, go to:  
<http://docs.sencha.com/touch/2.2.0/#!/api/Ext.data.proxy.Rest>.

For more information on the REST protocol itself, visit:

<http://net.tutsplus.com/tutorials/other/a-beginners-introduction-to-http-and-rest/>.

- `Sql`: This proxy lets you store data in a local SQL database. This should not be confused with an actual SQL server. The Sencha Touch SQL proxy outputs the model data into an HTML5 local database using WebSQL.

For this chapter and the next, we will be dealing mostly with local proxies. We will cover remote proxies and synchronizing data in *Chapter 9, Advanced Topics*.

The proxy can be declared as part of the model, shown as follows:

```
proxy: {  
  type: 'localStorage'  
  id: 'userProxy'  
}
```

All proxies require a type (local storage, session storage, and so on); however, some proxies will require additional information such as the unique ID required by the `localStorage` proxy.

We can also add a reader to this proxy configuration. The reader's job is to tell our proxy what format to use for sending and receiving data. The reader understands the following formats:

- `array`: A simple JavaScript array
- `xml`: An Extensible Markup Language format
- `json`: A JavaScript Object Notation format

The reader gets declared as part of the proxy:

```
proxy: {  
  type: 'localStorage',  
  id: 'userProxy',  
  reader: {  
    type: 'json'  
  }  
}
```

#### Declaring proxies and readers

The proxies and readers can be declared as part of the data store as well as the model. If different proxies are declared for a store and a model, then calling `store.sync()` will use the store's proxy, while calling `model.save()` will use the model's proxy. Using separate proxies on both models and the store is typically only needed in complex situations. It can also be confusing, so it's best to only define proxies in the model unless you're sure of what you are doing.



## Introducing data formats

Before we move on to data stores, we need to take a brief look at data formats. The three currently supported by Sencha Touch are Array, XML, and JSON. For each example, we will take a look at how the data would appear for a simple contact model with three fields: an ID, a name, and an e-mail ID.

### Arrays

An `ArrayStore` data format uses a standard JavaScript array, which would look something like this for our contact example:

```
[  
    [1, 'David', 'david@gmail.com'],  
    [2, 'Nancy', 'nancy@skynet.com'],  
    [3, 'Henry', 'henry8@yahoo.com']  
]
```

One of the first things we notice about this type of array is that there are no field names included as part of a JavaScript array. This means if we want to refer to the fields by the name in our template, we have to set up our model to understand where these fields should be mapped by using the `mapping` configuration option:

```
Ext.define('Contact', {  
    extend: 'Ext.data.Model',  
    config: {  
        fields: [  
            'id',  
            {name: 'name', mapping: 1},  
            {name: 'email', mapping: 2}  
        ],  
        proxy: {  
            type: 'memory',  
            reader: {  
                type: 'array'  
            }  
        }  
    }  
});
```

This sets up our `id` field as index 0 of our data, which is the default value. We then use the `mapping` configuration to set `name` and `email` as index 1 and 2 respectively, of the items in our data array. We can then set the template values for the display component using the configuration:

```
itemTpl: '{name}: {email}'
```

While arrays are typically used for simple data sets, a larger or nested data set can become very unwieldy using the simple JavaScript array structure. This is where our other formats come in.

## XML

**Extensible Markup Language (XML)** should be a familiar looking format to anyone who has worked with HTML web pages in the past. XML consists of data nested within a series of tags that identify the name of each part of the dataset. If we put our previous example into an XML format, it would look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<contact>
  <id>1</id>
  <name>David</name>
  <email>david@gmail.com</email>
</contact>
<contact>
  <id>2</id>
  <name>Nancy</name>
  <email>nancy@skynet.com</email>
</contact>
<contact>
  <id>3</id>
  <name>Henry</name>
  <email>henry8@yahoo.com</email>
</contact>
```

Notice that XML always begins with a version and encoding line. If this line is not set, the browser will not interpret the XML correctly and the request will fail.

We also include tags for defining the individual contacts. One advantage of this is that we can now nest data, as shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<total>25</total>
<success>true</success>
<contacts>
  <contact>
    <id>1</id>
    <name>David</name>
    <email>david@gmail.com</email>
  </contact>
  <contact>
    <id>2</id>
    <name>Nancy</name>
    <email>nancy@skynet.com</email>
  </contact>
  <contact>
    <id>3</id>
    <name>Henry</name>
    <email>henry8@yahoo.com</email>
  </contact>
</contacts>
```

In this nested example, we have each individual `contact` tag nested inside a `contacts` tag. We also have tags for our `total` and `success` values.

As we have a nested data structure, we will also need to let the reader know where to look for the pieces we need.

```
reader: {
  type: 'xml',
  root: 'contacts',
  totalProperty : 'total',
  successProperty: 'success'
}
```

The `root` property tells the reader where to start looking for our individual contacts. We also set a value outside of our `contacts` list for `totalProperty`. This tells the store that there are a total of 25 contacts, even though the store only receives the first three. The `totalProperty` property is used for paging through the data (that is, showing three of 25).

The other property outside of our `contacts` list is `successProperty`. This tells the store where to check whether the request was successful.

The only disadvantage of XML is that it's not a native JavaScript format, so it adds a little bit of overhead when it's parsed by the system. Typically, this is only noticeable in very large or deeply nested arrays, but it can be an issue for some applications.

Fortunately for us, we can also use JSON.

## JSON

**JavaScript Object Notation (JSON)** has all of the advantages of XML, but as a native JavaScript construct, it has less overhead associated with parsing. If we look at our data set as JSON, we would see the following:

```
[  
  {  
    "id": 1,  
    "name": "David",  
    "email": "david@gmail.com"  
  },  
  {  
    "id": 2,  
    "name": "Nancy",  
    "email": "nancy@skynet.com"  
  },  
  {  
    "id": 3,  
    "name": "Henry",  
    "email": "henry8@yahoo.com"  
  }  
]
```

We can also nest JSON in much the same way we do with XML:

```
{  
  "total": 25,  
  "success": true,  
  "contacts": [  
    {  
      "id": 1,  
      "name": "David",  
      "email": "david@gmail.com"  
    },  
    {  
    }
```

```

    "id": 2,
    "name": "Nancy",
    "email": "nancy@skynet.com"
},
{
    "id": 3,
    "name": "Henry",
    "email": "henry8@yahoo.com"
}
]
}

```

The reader would then be set up just as our XML reader, but with the type listed as JSON:

```

reader: {
    type: 'json',
    root: 'contacts',
    totalProperty : 'total',
    successProperty: 'success'
}

```

As before, we set properties for both `totalProperty` and `successProperty`. We also provide the reader with a place to start looking for our `contacts` list.



It should also be noted that the default values for `totalProperty` and `successProperty` are `total` and `success` respectively. If you are using `total` and `success` in your JSON return values, you don't really need to set these configuration options on `reader`.

## JSONP

JSON also has an alternate format called JSONP, or JSON with padding. This format is used when you need to retrieve data from a remote server. We need this option because most browsers follow a strict same-origin policy when handling JavaScript requests.

The same-origin policy means that a web browser will permit JavaScript to run as long as the JavaScript is running on the same server as the web page. This will prevent a number of potential JavaScript security issues.

However, there are times when you will have a legitimate reason for making a request from a remote server, for example querying an API from a web service such as Flickr. Because your app isn't likely to be running on `flickr.com`, you'll need to use JSONP, which simply tells the remote server to encapsulate the JSON response in a function call.

Luckily, Sencha Touch handles all of that for us. When you set up your proxy and reader, set the proxy type to `jsonp`, and set your reader up like you would a regular JSON reader. This tells Sencha Touch to use `Ext.data.proxy.JsonP` to do the cross-domain request, and Sencha Touch takes care of the rest.



If you'd like to see JSONP and `Ext.data.proxy.JsonP` in action, we use both to build the **Flickr Finder** application in *Chapter 8, Creating the Flickr Finder Application*.



While we have a number of formats to choose from, we will be using the JSON format for all of our examples moving forward in this chapter.

## Introducing stores

Stores, as the name implies, are used to store data. As we have seen in previous chapters, the list components require a store in order to display data, but we can also use a store to grab information from forms and hold it for use anywhere in our application.

The store, in combination with the model and proxy, works in much the same way as a traditional database. The model provides the structure for our data (say a schema in a traditional database), and the proxy provides the communication layer to get the data in and out of the store. The store itself holds the data and provides a powerful component interface for sorting, filtering, saving, and editing data.

The store can also be bound to a number of components, such as lists, nested lists, select fields, and panels, to provide data for display.

We will cover display, sorting, and filtering in *Chapter 7, Getting the Data Out*, but for now, we are going to look at saving and editing data with the store.

## A simple store

As this chapter is concerned with getting data into the store, we are going to start out with a very simple local store for our example:

```
var contactStore = Ext.create('Ext.data.Store', {
    model: 'Contact',
    autoLoad: true
});
```

This example tells the store which model to use, which in turn defines both the fields the store knows about and also the proxy the store should use, since the store will adopt both the field list and the proxy from its model. We also set the store to `autoLoad`, which means that it will load the data as soon as the store is created.

 If you declare a proxy in the store configuration, that proxy will be used instead of the model's proxy. This is useful in some situations where you want to store information about the collection of records such as a group of admin users. In that case, the model would be used to store the user details, but the store would be used to collect several users of a specific type (admin users) together.

We also need to be sure our model is set up correctly in order to use this store. Since we don't have a proxy listed as part of the store, we need to be sure the model has one if we want to save our data:

```
Ext.define('Contact', {
    extend: 'Ext.data.Model',
    config: {
        fields: [
            {name: 'id', type:'int'},
            {name: 'name', type: 'string'},
            {name: 'email', type: 'string'}
        ],
        proxy: {
            type: 'localstorage',
            id: 'myContacts',
            reader: {
                type: 'json'
            }
        }
    }
});
```

This is our simple model with three items: an ID, a name, and an e-mail address. We would then create a new contact as we did before:

```
var newContact = Ext.create('Contact', {  
    name: 'David',  
    email: 'david@msn.com'  
});
```

Notice that we don't set the ID this time. We want the store to set that for us (similar to the way autoincrement works in a typical database). We can then add this new contact to the store and save it:

```
var addedUser = contactStore.add(newContact);  
contactStore.sync();
```

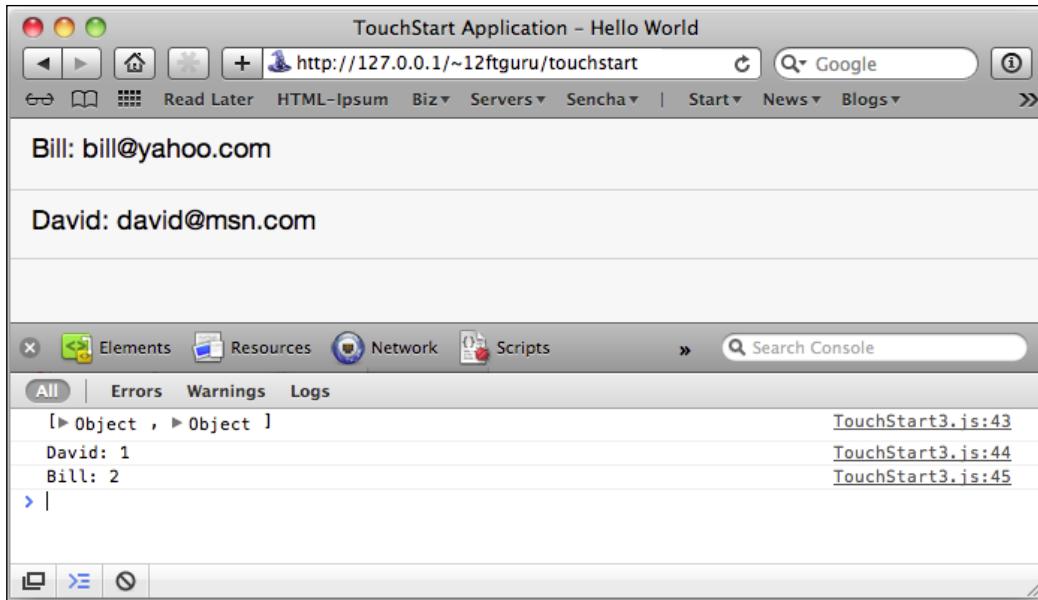
The first line adds the user to the store and the second line saves the contents of the store. By splitting the add and sync functionalities, you can add multiple users to the store and then perform a single save, as shown in the following code:

```
var newContact1 = Ext.create('Contact', {  
    name: 'David',  
    email: 'david@msn.com'  
});  
  
var newContact2 = Ext.create('Contact', {  
    name: 'Bill',  
    email: 'bill@yahoo.com'  
});  
  
var addedContacts = contactStore.add(newContact1, newContact2);  
contactStore.sync();
```

In both the cases, when we add contacts to the store, we set up a return variable to grab the return value of the add method. This method returns an array of contacts that will now have a unique ID as part of each contact object. We can take a look at these values by adding a couple of console logs after our sync:

```
console.log(addedContacts);  
console.log(addedContacts[0].data.name + ': ' + addedContacts[0].data.id);  
console.log(addedContacts[1].data.name + ': ' + addedContacts[1].data.id);
```

This will show that two contact objects in an array are returned. It also shows how to get at the data we need from those objects by using the index number of the specific contact in the array. We can then drill down into the data for a name and the new ID that was assigned when we synced.



Now that we have a general idea of how to get data into a store, let's take a look at how to do it with a form.

## Forms and stores

For this example, we are going to use the same store and model as our previous example, but we will add a list and a form so that we can add new contacts and see what we have added. Let's start with the list:

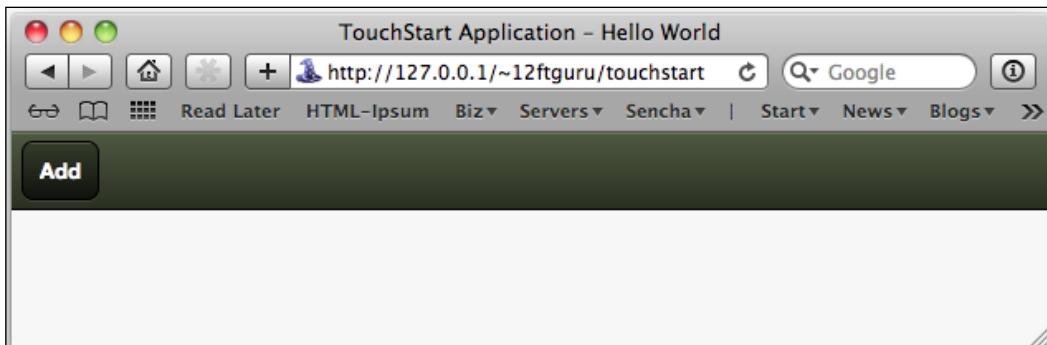
```
this.viewport = Ext.create('Ext.Panel', {
    fullscreen: true,
    layout: 'fit',
    items: [
        {
            xtype: 'toolbar',
            docked: 'top',
            items: [{
                text: 'Add',
                handler: function() {
                    Ext.Viewport.add(addNewContact);
                    addNewContact.show()
                }
            }]
        },
    ]
},
```

## *Getting the Data In*

---

```
{  
    xtype: 'list',  
    itemTpl: '{name}: {email}',  
    store: contactStore  
}  
});
```

You will get something similar to what is shown in the following screenshot:



Most of the code here is pretty similar to the previous examples. We have a single panel with a `list` component. Our list has a template `itemTpl` that uses the same field names as our `contact` model and arranges the manner in which they will be displayed. We have also added a docked toolbar with our new **Add** button.



The toolbar component has also changed from the previous versions of Sencha Touch. In Version 2, `toolbar` is part of the `items` list and not a separate `dockedItem`, as in the past versions. Additionally, the position of the toolbar was previously set by the `dock` configuration option. This was changed to `docked` in Sencha Touch 2. It should also be noted that if you try using the older `dockedItem` and `dock` configurations, you will not get any errors. You will also not get a toolbar. This can lead to a great deal of hair pulling and coarse language.

The button has a very simple function that will add an `Ext.Sheet` called `addNewContact` to our `viewport` and then show the sheet. Now we need to actually create the sheet:

```
var addNewContact = Ext.create('Ext.Sheet', {  
    height: 250,  
    layout: 'fit',  
    stretchX: true,
```

```

    enter: 'top',
    exit: 'top',
    items: [...]
});

```

This gives us our new sheet that will appear when we click the **Add** button. Now, we need to add our form fields to the `items` section of the sheet we just created:

```

{
  xtype: 'formpanel',
  padding: 10,
  items: [
    {
      xtype: 'textfield',
      name : 'name',
      label: 'Full Name'
    },
    {
      xtype: 'emailfield',
      name : 'email',
      label: 'Email Address'
    }
  ]
}

```

We start by creating our `formpanel` component and then adding `textfield` and `emailfield` to the `items` list of `formpanel`.

## Specialty text fields

Sencha Touch uses specialty text fields such as `emailfield`, `urlfield`, and `numberfield`, to control which keyboard is used by the mobile device, as in the following iPhone examples:



The types of keyboards shown in the preceding figure are explained as follows:

- **The URL Keyboard** replaces the traditional Space bar with keys for dot (.), slash (/), and .com
- **The Email Keyboard** shortens the Space bar and makes room for @ and dot (.)
- **The Number Keyboard** initially presents the numeric keyboard instead of the standard QWERTY keyboard

These specialty fields do not automatically validate the data the user enters. Those kinds of validations are handled through model validations.



#### Specialty keyboards

Android and iOS have slightly different special keyboards, so you may find some variation between the two. It is usually helpful to run your application through both the Android and iOS simulators to ensure that the correct keyboard type is being used.

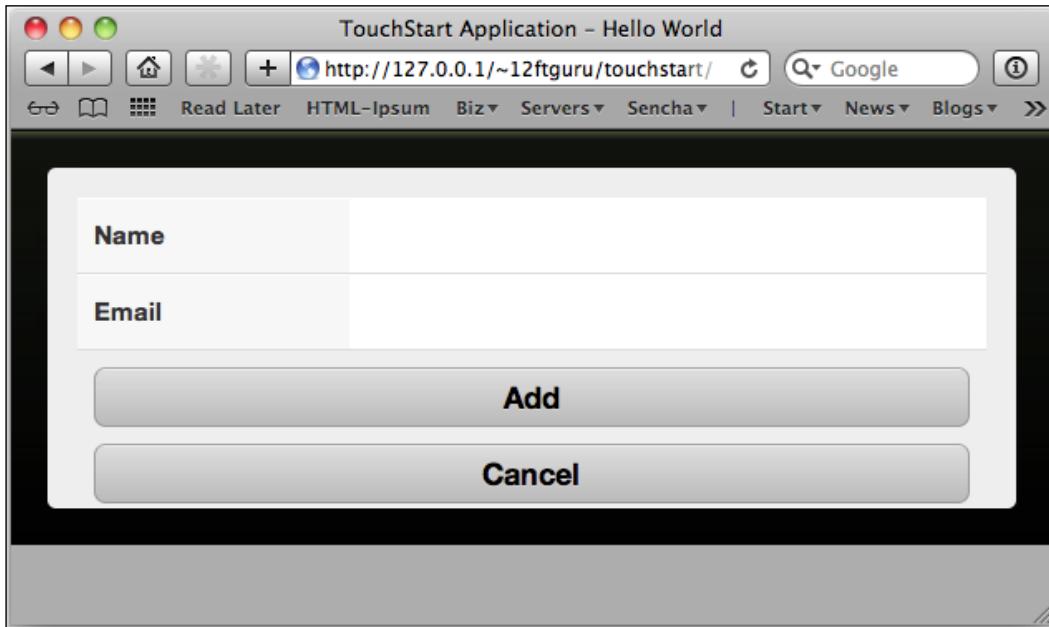
## Mapping fields to the model

You will also notice that the name of each field in our form matches the name used by our contact model; this will allow us to easily create our contacts and add them to the store. However, before we get there, we need to add two buttons (**Save** and **Cancel**) to tell the form what to do.

After the `emailfield` object in our form, we need to add the following:

```
{  
    xtype: 'button',  
    height: 20,  
    text: 'Save',  
    margin: 10,  
    handler: function() {  
        this.up('sheet').hide();  
    }  
}, {  
    xtype: 'button',  
    height: 20,  
    margin: 10,  
    text: 'Cancel',  
    handler: function() {  
        this.up('sheet').hide();  
    }  
}
```

The gives us two buttons at the bottom of our form. Right now, both our **Save** and **Cancel** buttons do the same thing: they call a function to hide the sheet that holds our form. This is a good starting point, but we need a bit more to get our **Save** button to save our data.



Since we were good little coders and named our fields to match our model, we can just use the following code in our button handler to add our form to our store:

```
handler: function() {
    var form = this.up('formpanel');
    var record = Ext.create('Contact', form.getValues());
    contactStore.add(record);
    contactStore.sync();
    form.reset();
    this.up('sheet').hide();
}
```

The first line uses the `up` method to grab the form that surrounds the button. The second line uses `form.getValues()` and pipes the output directly into a new `Contact` model, using the `create()` method from our previous examples. We can then add the new contact to the store and sync, as we did before.

### *Getting the Data In*

---

The last bit of cleanup we need to do is to clear all of the form values by using `form.reset()` and then hide the sheet, as before. If we don't reset the fields, the data would still be there the next time we showed the form.

The list connected to the store will refresh when we sync the store, and our new contact will appear.

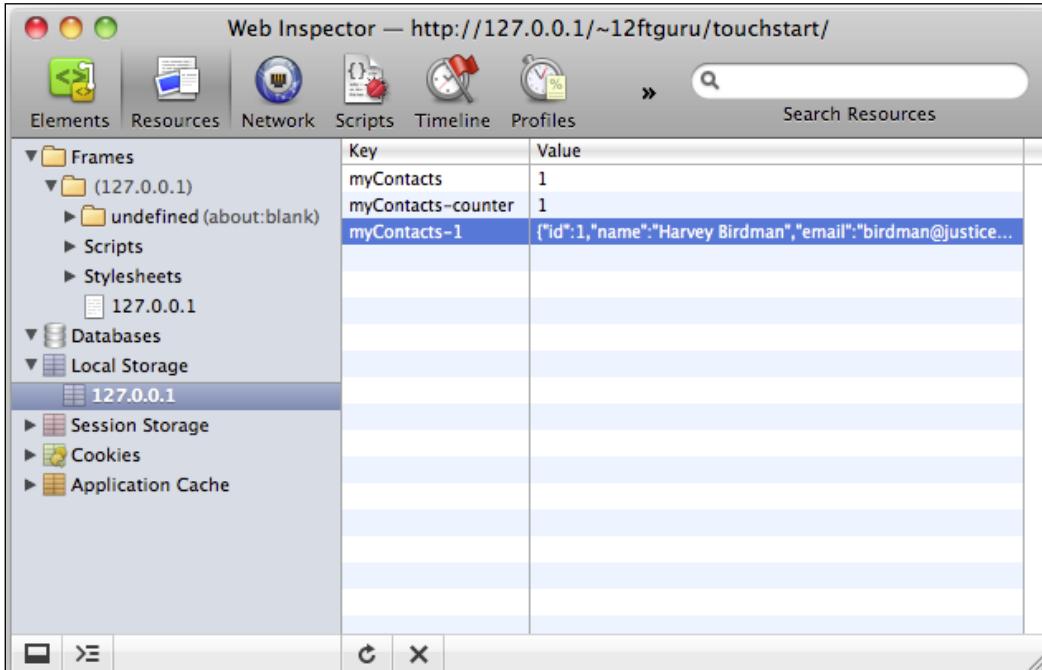


Since this store uses local storage for holding the data, our list will stay in place even after we quit the Safari browser. This can be a bit of a pain when you are testing an application, so let's take a look at how to clear out the store.

## **Clearing the store data**

Local and session storage saves information on our local machine. As we plan on doing lots of testing while coding, it's a good idea to know how to clear out this kind of data without removing other data that you might still need. To clear out the data for your local or session store, use the following steps:

1. Open up **Web Inspector** from the **Develop** menu and select the **Resources** tab.



2. In the **Local Storage** or **Session Storage** section (depending on the method you use), you should see your application's database. Once you select the database, you can delete specific records or empty out the database completely. Just select the records on the right-hand side of the screen, and click on the X at the bottom to delete the record.
  3. You can also reset the value for the counter by double-clicking on it and then changing the number. Be careful not to create multiple records with the same number. This will cause big problems.
  4. Once you are finished in the **Resources** section, let's move on to editing data with our forms.

## Editing with forms

Now that we have taken a look at the basics of getting data into a store, let's look at how to edit that data, using a few modifications to our current form.

The first thing we want to add is an `itemsingletap` listener on our list. This will let us tap an item in the list and bring up the form with the selected entry included in the fields for us to edit. The listener looks like the following:

```
listeners: {
  itemsingletap: {
    fn: function(list, index, target, record) {
      addNewContact.down('form').setRecord(record);
      Ext.Viewport.add(addNewContact);
      addNewContact.show();
    }
  }
}
```

Our `itemsingletap` listener will automatically get back a copy of the `list`, the `index` attributes of the item, the `target` element, and the `record` behind the item that got tapped. We can then grab the form inside our sheet and set the record on it.

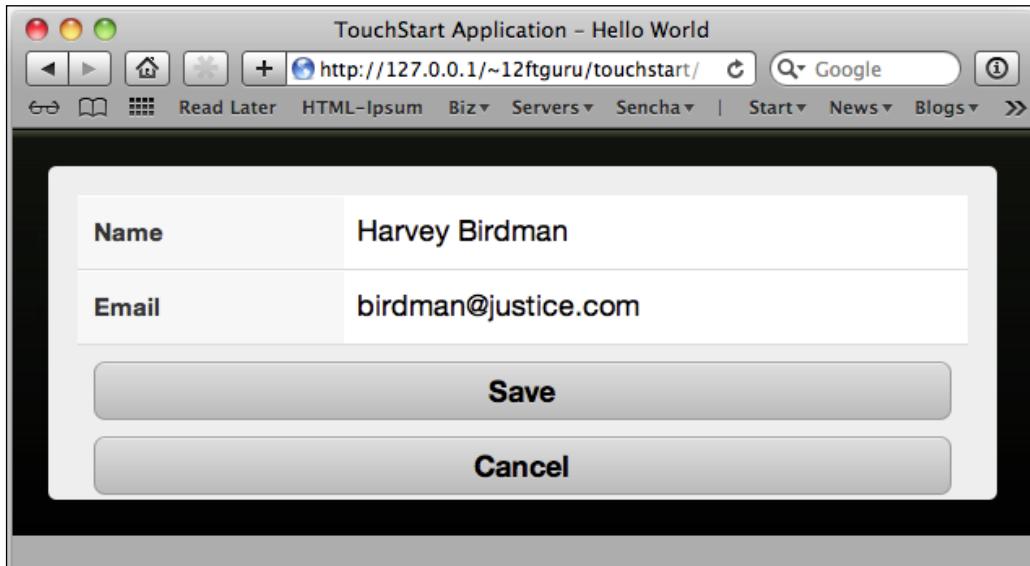
It is often useful to chain functions together in this fashion, especially if the piece you need has to be used only once. For example, we could have done:

```
var form = addNewContact.down('form');
form.setRecord(record);
```

This would also let us use that `form` variable in a number of places within the function. Since we only need it to set the record, we can combine both of these lines as a single line:

```
addNewContact.down('form').setRecord(record);
```

This loads the data into our form in the following manner:



There's still one more problem to be dealt with: our **Save** button is hardcoded to add a new record to the store. If we tap **Save** right now, we will just end up with multiple copies of the same contact. We need to make a change to our form to let us switch what the **Save** button does, depending on whether we are editing or creating a new contact.

## Switching handlers

In order to change the handler, the button fires to save our contact; we need to separate the bulk of code from the button itself. To begin, locate the handler for our **Save** button and copy the current function to your clipboard. Next, we want to replace that function with the name of an external function:

```
handler: addContact
```

We are also going to add an additional config option to our button in the following manner:

```
action: 'saveContact'
```

This will make it easier to grab our button with a component query later on.



The action config option is a totally arbitrary name. You are not restricted to just the options defined by Sencha. You can define any additional options you like for the components and reference them in your handlers and controllers just like any other config option.

Now, we have to create the new `addContact` function for this handler to use. In our JavaScript file, right before where we create our `addNewContact` sheet, add a new function called `addContact` and paste in the code from our old `handler` function. It should look as follows:

```
var addContact = function() {
    var form = this.up('formpanel');
    var record = Ext.create('Contact', form.getValues());
    contactStore.add(record);
    contactStore.sync();
    form.reset();
    this.up('sheet').hide();
};
```

This is the same old form-saving function we used on our button before, and it will work just fine for adding new contacts. Now, we need to create a similar function to update our contacts when we click on them in the list.

On top of our `addContact` function, add the following code:

```
var updateContact = function() {
    var form = this.up('formpanel');
    var rec = form.getRecord();
    var values = form.getValues();
    rec.set(values);
    contactStore.sync();
    form.reset();
    this.up('sheet').hide();
};
```

This does almost the same thing as our other function. However, instead of grabbing the form fields and creating a new record, we grab the record from the form itself using `form.getRecord()`. This record is the one we need to update with our new information.

We then grab the current values of the form using `form.getValues()`.

Our `rec` variable is now set to the old record from the data store. We can then pass that record to our new data using `rec.set(values)`, which will overwrite the old information in the store record with our current form values. The ID will stay the same as we do not pass a new value for that.

After we update the record, we just perform the following actions we did earlier:

- `sync`
- `reset`
- `hide`

Now that the code for our two functions is in place, we need to switch the handler for our **Save** button based on whether the user clicked on the **Add** button at the top of our list or selected an item in the list.

Let's start with the **Add** button. Locate the handler for our **Add** button at the top of our `list` object. We need to add some code to this button that will change the handler on the **Save** button:

```
handler: function() {  
    var button = addNewContact.down('button[action=saveContact]');  
    button.setHandler(addContact);  
    button.setText('Add');  
    Ext.Viewport.add(addNewContact);  
    addNewContact.show();  
}
```

As our `addNewContact` sheet is already defined as a variable elsewhere in the code, we can grab `button` using the `down()` method and make a few changes. The first is to update the handler to see our new `addContact` function, and the second is to change the text of the button to `Create`. We can then add our `addNewContact` sheet to the viewport and call `addNewContact.show()`, as before.

Our **Add** button is now set to show the form and change the text and handler for the button.

Now, we need to do something similar to the `itemssingletap` hander on our list:

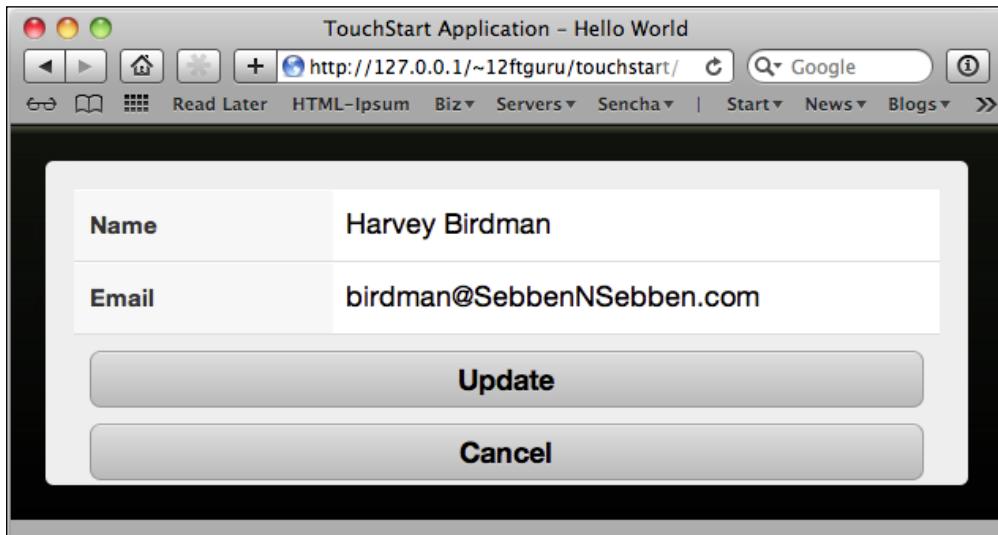
```
itemssingletap: {  
    fn: function(list, index, target, record){  
        addNewContact.down('formpanel').setRecord(record);  
        var button = addNewContact.down('button[action=saveContact]');  
        button.setHandler(updateContact);  
        button.setText('Update');
```

### *Getting the Data In*

---

```
Ext.Viewport.add(addNewContact);
addNewContact.show();
}
}
```

Here, we still take the record and load it into the form, but we grab button with the action value of saveContact and make changes to the handler and text as well. The changes point the **Save** button to our updateContact function and change the text to **Update**.



## Deleting from the data store

If you remember earlier, when we talked about CRUD functions, you can see that we successfully covered Create, Read, and Update. These are all handled automatically by the store with very little code required. What about Delete?

As it turns out, Delete is just as simple as our other store methods. We can use either of the two methods: the first is `remove()` – it takes a record as its argument – and the second is `removeAt`, which takes an index to determine the record to remove. We could implement either of these as part of our edit form by adding a new button at the bottom of the form as follows:

```
{
    xtype: 'button',
    height: 20,
    margin: 10,
    text: 'Delete',
```

```
ui: 'decline',
handler: function() {
    var form = this.up('formpanel');
    contactStore.remove(form.getRecord());
    contactStore.sync();
    form.reset();
    this.up('sheet').hide();
}
```

Using `remove` requires the store record, so we grab the record from our form panel:

```
contactStore.remove(form.getRecord());
```

That takes care of all of our basic Create, Read, Edit, and Delete functions. As long as you remember to set up your model and match your field names, the store will handle most of the basics automatically.



#### Further Information

Sencha has a number of good tutorials on using forms, models, and stores at <http://docs.sencha.com/touch/2.2.1/#!/guide>.

## Summary

In this chapter, we covered the data model that forms the basic structure for all of our data in Sencha Touch. We looked at the proxy and reader that handle communications between the data store and our other components. We also discussed the data store that holds all of our data in Sencha Touch. Finally, we took a look at how you can use forms to get data in and out of the stores as well as at how to delete the data when it is no longer needed.

In the next chapter, we will take a look at all of the other things we can do with data once we get it out of the store.



# 7

## Getting the Data Out

In the previous chapter, we looked at how you can get data into a Sencha Touch data store. Once we have the data, the next step is figuring out how to get it out of the store and use it in our application. Fortunately for us, Sencha Touch has a number of built-in ways to help us accomplish this task. Here we will explore how to use individual data records as well as the complete contents of the data store to display information in our application.

In this chapter, we will look at:

- Using data stores for display
- Binding, sorting, filtering, paging, and loading data stores
- Working with XTemplates
- Looping through data in an XTemplate
- Conditional display and inline functions in XTemplates
- Inline JavaScript and member functions in XTemplates
- Using Sencha Touch Charts to display store data

### Using data stores for display

Being able to store data in your application is only half the battle. You need to be able to easily get the data back out and present it in a meaningful way to the user. Lists, panels and other data-capable components in Sencha Touch offer three configuration options to help you accomplish this task: `store`, `data`, and `tpl`.

## Directly binding a store

Data views, lists, nested lists, form select fields, and index bars are all designed to display multiple data records. Each of these components can be configured with a data store from where to pull these records. We introduced this practice earlier in the book:

```
Ext.application({
    name: 'TouchStart',
    launch: function () {
        Ext.define('Contact', {
            extend: 'Ext.data.Model',
            config: {
                fields: [
                    {name: 'id'},
                    {name: 'first', type: 'string'},
                    {name: 'last', type: 'string'},
                    {name: 'email', type: 'string'}
                ],
                proxy: {
                    type: 'localstorage',
                    id: 'myContacts',
                    reader: {
                        type: 'json'
                    }
                }
            }
        });
    }

    var main = Ext.create('Ext.Panel', {
        fullscreen: true,
        layout: 'fit',
        items: [
            {
                xtype: 'list',
                itemTpl: '{last}, {first}',
                store: Ext.create('Ext.data.Store', {
                    model: 'Contact',
                    autoLoad: true
                })
            }
        ]
    });

    Ext.Viewport.add(main);
}

});
```

The store configuration takes the `model` and `autoLoad` properties as part of its setup. This will grab all of the store's data (using the proxy on the `model` parameter) and pull it into the list for display. We are pretty familiar with this now, but what if we only want some of the data, or if we need the data in a specific order?

As it turns out, Sencha Touch stores can be sorted and filtered both when they are first created and later if we need to change the filtering or sorting in response to the user.

## Sorters and filters

Sorters and filters can be used in a number of ways. The first way is to set up a default configuration on the store as part of its creation.

```
var myStore = Ext.create('Ext.data.Store', {
    model: 'Contact',
    sorters: [
        {
            property: 'lastLogin',
            direction: 'DESC'
        },
        {
            property: 'first',
            direction: 'ASC'
        }
    ],
    filters: [
        {
            property: 'admin',
            value: true
        }
    ]
});
```

Our `sorters` component is set as an array of property and direction values. These are executed in order, so our example sorts first by `lastLogin` (most recent first). Within `lastLogin`, we sort by name (alphabetically ascending).

Our filters are listed as `property` and `value` pairs. In our example, we want the store to show us only `admin`. The store might actually contain non-admins as well, but here we are requesting that those be filtered out initially.

Sorters and filters can also be modified after the initial load-in by using one of the following methods:

- `clearFilter`: This method clears all filters on the store, giving you the full content of the store.
- `filter`: This method takes a filter object, just like the one in our earlier configuration example, and uses it to limit the data as requested.
- `filterBy`: This method allows you to declare a function that is run on each item in the store. If your function returns `true`, the item is included. If it returns `false`, then the item is filtered out.
- `sort`: This method takes a `sort` object just like the ones in our configuration example and uses it to sort the data as requested.

If we use our earlier store example, changing the `sort` order would look like this:

```
myStore.sort( {  
    property : 'last',  
    direction: 'ASC'  
});
```

Filtering has to take into account any previous filters on the store. In our current store example, we are set to filter out anyone without an `admin` value of `true`. If we try the following code, we will not get back anything in the list because we have effectively told the store to filter by both the new (`admin = false`) and previous (`admin = true`) filters:

```
myStore.filter( {  
    property : 'admin',  
    value: false  
});
```

As `admin` is a Boolean value, we get back nothing. We have to clear out the old filter first:

```
myStore.clearFilter();  
myStore.filter( {  
    property : 'admin',  
    value: false  
});
```

This example will clear the old '`admin`' filter from the store and return a list of everyone who is not an admin.

Sorting and filtering provides a powerful tool for manipulating data inside the data store. However, there are a few other situations we should also take a look at. What do you do when you have too much data, and what do you do when you need to reload the data store?

## Paging a data store

In some cases, you will end up with more data than your application can comfortably manage in a single bite. For example, if you have an application with 300 contacts, the initial load-in time might be more than you really want. One way to handle this is by paging the data store. Paging allows us to grab the data in chunks and send the next or previous chunk of data as the user needs it.

We can set up paging using the pageSize configuration:

```
var myStore = Ext.create('Ext.data.Store', {
    model: 'Contact',
    pageSize: 40,
    proxy: {
        type: 'localstorage',
        id: 'myContacts',
        reader: {
            type: 'json'
        }
    },
    autoLoad: true
});
```

We can then move through the data using the paging functions:

```
myStore.nextPage();
myStore.previousPage();
myStore.loadPage(5);
```

This code moves forward one page, moves back one page, and then jumps to page five.



It should be noted that page indexes are 1 based (that is, numbered 1, 2, 3, and so on) rather than 0 based (that is, numbered 0, 1, 2, 3, and so on, like arrays).



If we jump to page five and it doesn't exist, things will probably go poorly for our application (that is, it will go kaboom!), which means that we need a good way to figure out how many pages we actually have. For this we need to know the total number of records in our data store.

We could try using the `getCount()` method for the data store, but this only returns the number of currently cached records in the store. Since we are paging through the data, it means that we are not loading everything available. If we have a maximum page count of 40, but there are 60 records in our database, the `getCount()` method will return 40 for the first page and 20 when we load the second page.

Also, if you filter your store's data, the number returned by `getCount()` will be the number of records that matched the filter, not the total number of records in the store. We need to set up our store's reader to get an actual total back from our system. We also need to tell the store where these records will be when the data comes back to it.

We can set a configuration on the reader for `totalProperty` and `rootProperty`, such as the following:

```
var myStore = new Ext.data.Store({
    model: 'Contact',
    pageSize: 40,
    proxy: {
        type: 'localstorage',
        id: 'myContacts',
        reader: {
            type: 'json',
            totalProperty: 'totalContacts',
            rootProperty: 'contacts'
        }
    },
    autoLoad: true
});
```

This tells our reader to look for two extra properties, called `totalContacts` and `rootProperty`, in the data that it collects. The data that we pull into the store will also have to be set up to include this new property as part of the data string. How this is done will be determined largely by how your data is created and stored, but in a JSON data array, the format will look something like the following:

```
{
    "totalContacts": 300,
    "contacts": [...]
}
```

The `totalContacts` property tells us how many contacts we have, and the `rootProperty` tells the reader where to start looking for those contacts.

Once our data is set up in this fashion, we can grab the total contacts, as follows:

```
var total = myStore.getProxy().getReader().getTotalCount()
```

We can then divide by `myStore.getPageSize()`, to determine the total number of pages in our data. We can also grab the current page with `myStore.currentPage`. These two pieces of information will allow us to display the user's current location in the pages (for instance, **Page: 5 of 8**). Now, we need to account for what happens when the data behind our store changes.

## Loading changes in a store

When we use a data store to pull information from an external source, such as a file, a website, or a database, there is always the chance that the data will change at the external source. This will leave us with stale data in the store.

Fortunately there is an easy way to deal with this using the `load()` function on the store. The `load()` function works as follows:

```
myStore.load({
  scope: this,
  callback: function(records, operation, success) {
    console.log(records);
  }
});
```

The `scope` and `callback` functions are both optional. However, `callback` offers us an opportunity to do a number of interesting things, such as compare our old and new records or alert the user visually once the new records are loaded.



You can also set up a listener on the store for the `load` event. This will make the store use this callback at any point in time when the basic `store.load()` function is called. Additionally, there is an event called `beforeLoad` that, as the name implies, fires off each time before the store is loaded. If the `beforeLoad` event returns `false`, the `load` event is not fired.

Another consideration while loading data stores is whether to automatically load (`autoLoad`) the store as part of its creation or load it later. A good rule of thumb is to only autoload the data stores that you know will be displayed initially. Any subsequent stores can be set to load when the component that they are bound to is shown.

For example, let's say we have a list of system users that will only be accessed occasionally within the program. We can add a listener to the component list itself in the following manner:

```
listeners: {
  show: {
    fn: function(){ this.getStore().load(); }
  }
}
```

This code will load the store only if the `list` component is actually shown. Loading stores in this fashion saves us time while launching our application, and also saves memory. However, it should be noted that the code will also load the store every time this component is shown. This is desirable if you expect the data behind the store to be updated frequently; otherwise, it is better to load the store manually.

We can also save time and memory by using the store to feed multiple components, such as a list of data and a details panel. As with the preceding example, there are some caveats to this tactic. If a filter, sort, or data load is applied to the store by one component, it will also affect any of the other components that this store is bound to.

## Data stores and panels

Unlike lists, where a number of records can be displayed, a panel typically displays a single record. However, we can still grab this information from our data store in the same way that we do for a list.

Let's start with a variation of our contacts example from the beginning of the chapter; we will build a list of names using `first` and `last`, and then add a details panel that shows the full name, e-mail address, and phone number for the selected name.

We start with our model and store first:

```
Ext.define('Contact', {
  extend:'Ext.data.Model',
  config:{},
  fields:[
    {name:'first', type:'string'},
    {name:'last', type:'string'},
    {name:'address', type:'string'},
    {name:'city', type:'string'},
    {name:'state', type:'string'},
    {name:'zip', type:'int'},
    {name:'email', type:'string'},
    {name:'birthday', type:'date'}
```

```
],
proxy:{  
    type:'ajax',  
    url:'api/contacts.json',  
    reader:{  
        type:'json',  
        rootProperty:'children'  
    }
}
});
var contactStore = Ext.create('Ext.data.Store', {  
    model:'Contact',  
    autoLoad:true  
});
```

This gives us our `first` and `last` values, which we will use for our initial list, and the `email`, `birthday`, and `address` information, which we will use for the details.

Sharp-eyed readers will also notice that we have changed our model to use Ajax as the proxy for an `api/contacts.json` URL (remember, our store will automatically use this proxy as well). This means that when the store loads, it will look for a local file in the `api` folder called `contacts.json`. This file is available as part of the downloadable code files for this book, and it contains some test data that we have thrown together. If you want to create your own file instead of downloading it, the format of the file looks like the following:

```
{
    "children": [
        {
            "first": "Ila",
            "last": "Noel",
            "email": "ante.ipsum@Sedmalesuada.ca",
            "address": "754-6686 Elit, Rd.",
            "city": "Hunstanton",
            "state": "NY",
            "zip": 34897,
            "birthday": "Tue, 16 Oct 1979 04:27:45 -0700"
        },
        ...
    ]
}
```

## Getting the Data Out

By setting this store to look at a local text file, we can add data quickly for testing by adding new children to the text file.

### **Test data is your friend**



Whenever you put together an application and test it, you will probably need some data in order to make sure things are working correctly. It's often very tedious to enter this information into a text file manually or to enter it in data forms over and over again. Fortunately, <http://www.generatedata.com/> is a website that generates random data in a number of formats. Just provide the field names and types, and then tell the website how many records you need. Click on the button, and you get back random data, ready for testing. Best of all, it's free!

The screenshot shows the homepage of generatedata.com. At the top, there is a navigation bar with links for 'Generate', 'About', 'News', and 'Donate'. A language selector shows 'English'. Below the navigation, there is a search bar labeled 'Your data set name here...' with a 'SAVE' button and icons for calendar, trash, and search. A section titled 'COUNTRY-SPECIFIC DATA' allows selecting 'All countries'. The main area is titled 'DATA SET' and contains two tables for defining columns. The first table has four rows, each with 'Order', 'Column Title', 'Data Type' (dropdown), and 'Options' (checkbox). The second table is identical. Below these is a 'Add 1 Row(s)' button. Under 'EXPORT TYPES', there are tabs for CSV, Excel, HTML, JSON, Programming Language, SQL, and XML. The 'HTML' tab is selected, with options for 'Data format' (radio buttons for <table>, <ul>, <dl>) and 'Use custom HTML format' (checkbox). At the bottom, there is a 'Generate' button and options for generating 100 rows, choosing the output location ('Generate in-page', 'New window/tab', 'Prompt to download'), and a 'Generate' button.

Our `list` component basically stays the same as before. As `list` uses the template `itemTpl: '{last}, {first}'` for each item in the list, it simply ignores the values for `address`, `city`, `state`, `zip`, `email`, and `birthday`. However, as these values are still part of the overall data record, we can still grab them and use them in our panel to display details.

Before we can add our details panel, we need to create our `main` panel and set it to use a `card` layout. This will let us switch between the list and the details with a single tap:

```
var main = Ext.create('Ext.Panel', {
    fullscreen:true,
    layout:'card',
    activeItem:0,
    items:[
        {
            xtype:'list',
            itemTpl:'{last}, {first}',
            store:contactStore
        }
    ]
});
```

We have set the `main` panel to use a `card` layout, with the `activeItem` component as 0. In this case, item 0 is our `list`, which is built into the `main` panel.

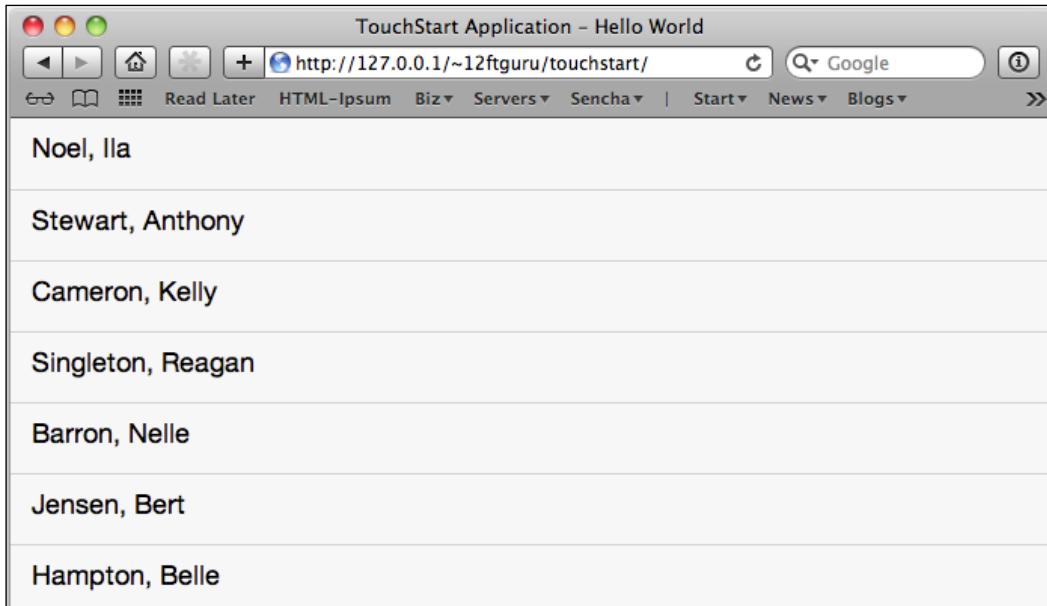
Make sure that all the components are wrapped inside an application launch function like our examples from the previous chapters:

```
Ext.application({
    name:'TouchStart',
    launch:function () {
        //components go here
        Ext.Viewport.add(main);
    }
});
```

At the bottom, within the `launch` function, we add the `main` panel to the `Viewport`.

## Getting the Data Out

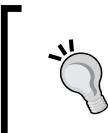
Once you have your data and the main panel set up, load the page to make sure that things are working correctly with what we have so far.



Now, we need to add in our `detailsPanel` component. We will start simple with this first part and add a new panel item after our list:

```
var detailsPanel = Ext.create('Ext.Panel', {
    tpl: '{first} {last}<br>'+
        '{address}<br>'+
        '{city}, {state} {zip}<br>'+
        '{email}<br>{birthday}',
    items: [
        {
            xtype: 'toolbar',
            docked: 'top',
            items: [
                {
                    text: 'Back',
                    ui: 'back',
                    handler: function () {
                        main.setActiveItem(0);
                    }
                }
            ]
        }
    ]
});
```

The first thing we add is a simple template. We include some HTML line breaks to lay out the data in a better way. We also split the template into multiple lines for readability and used the + operator to concatenate the strings together. Then we add a Back button that will bring us back to our main list.

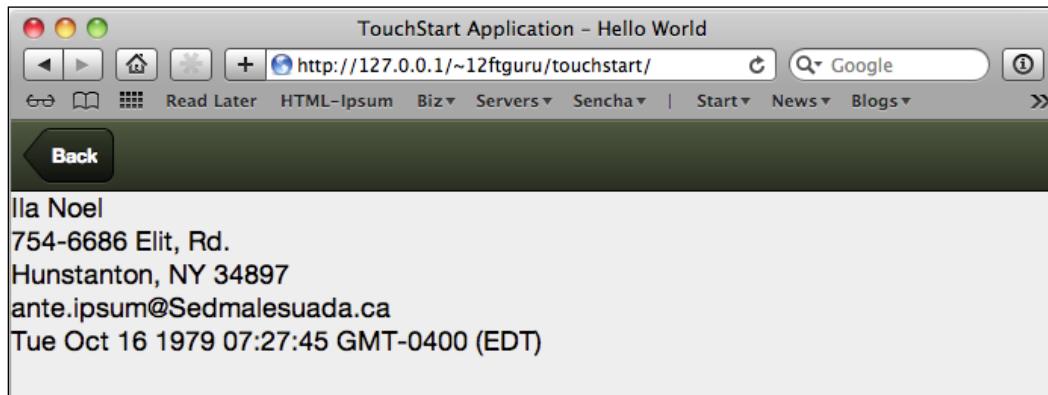


Since we already defined `main` as a variable in our code, we can use it inside our `handler` function. Since the `main` panel is also the first panel inside our `viewport`, we can grab it like this: `console.log(this.up('viewport').down('panel'));`

Once our `detailsPanel` is defined, we need to add a `listeners` section to our list to load the data into the panel:

```
listeners: {
  itemtap: {
    fn: function (list, index, target, record) {
      detailsPanel.setRecord(record);
      main.setActiveItem(1);
    }
  }
}
```

The good thing about this is that we don't really need to load anything new. The list already has access to all of the extra data that is present in the data store. We also receive the record as part of the `itemTap` event. We can take this record and set it on the panel using the `setRecord()` function. Finally, we set the active item to our `detailsPanel` component. When we tap on an item in the list, the result looks as follows:



The `detailsPanel` component includes not only the first and last name from our list, but the address, e-mail, and birthday data as well. All of this data comes from the same data store; we simply use the templates to choose which data to display.

Speaking of templates, ours looks a little dull, and the `birthday` value is a bit more specific than we really need. There must be something we can do to dress this up a bit.

## XTemplates

As we have seen from a number of previous examples, **XTemplate** is a structure that contains HTML layout information and placeholders for our data.

So far we have only created very basic templates for our list and panel using the data values and a bit of HTML. In our first example, we learned how to use the `+` operator to allow us to take a very long string and break it up into smaller strings for readability. Another way to do this is to set these templates up as separate components:

```
var myTemplate = new Ext.XTemplate(
    '{first} {last}<br>',
    '{address}<br>',
    '{city}, {state} {zip}<br>',
    '{email}<br>',
    '{birthday}'
);
```

This will create a template that looks exactly like what we had before. This is the way most examples on the Sencha Touch website are written, so it is good to be aware of both methods.

Once we have our component template, we can then add it to our panel along with `tpl: myTemplate`.

The following two methods provide better readability for your code while working with complex templates:

```
tpl: new Ext.XTemplate(
    '<div style="padding:10px;"><b>{first} {last}</b><br>',
    '{address}<br>',
    '{city}, {state} {zip}<br>',
    '<a href="mailto:{email}">{email}</a><br>',
    '{birthday}</div>'
);
```

or:

```
tpl: '<div style="padding:10px;"><b>{first} {last}</b><br>'+
  '{address}<br>'+
  '{city}, {state} {zip}<br>'+
  '<a href="mailto:{email}">{email}</a><br>'+
  '{birthday}</div>'
```

Both methods provide the same result.

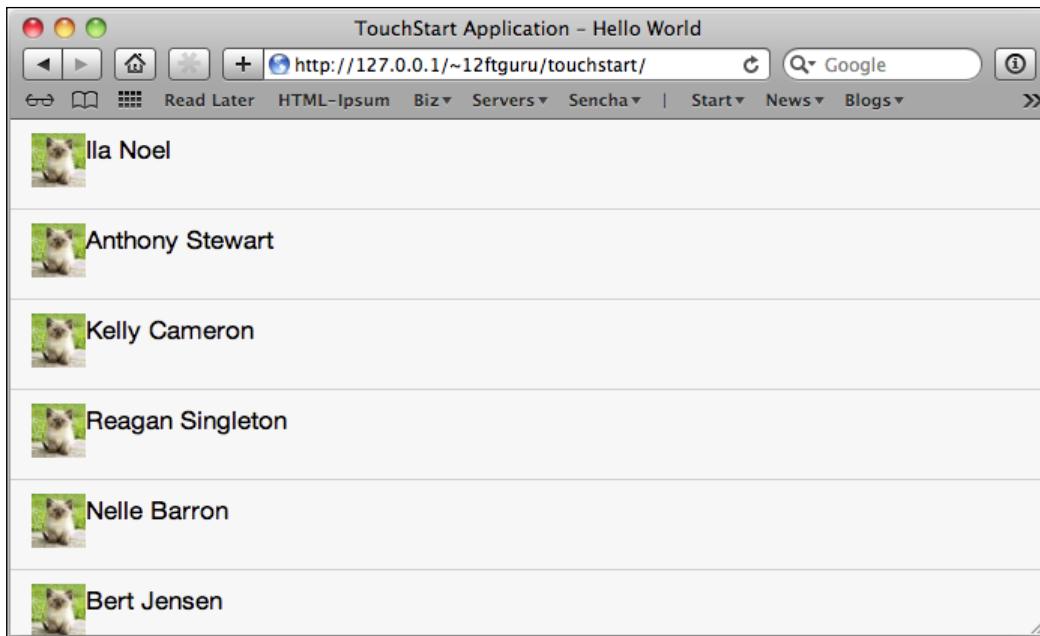


We can also use the same types of XTemplates with our main list to give it a bit more style. For example, adding the following code as the `itemTpl` component for our list will place an adorable kitten picture next to each name in the list:

```
var listTemplate = new Ext.XTemplate(
  '<div class="contact-wrap" id="{first}-{last}">',
  '<div class="thumb" style="float: left;"></div>',
  '<span class="contact-name">{first} {last}</span></div>'
);
```

## Getting the Data Out

For this example, we just added an HTML component to lay out each line of data and then used a random image-generation service (in this case, [placekitten.com](http://placekitten.com)) to place any 36x36 kitten picture, which will line up next to our names on the left (you can also use this to display the contact's picture).



At this point, we are still playing with basic HTML; however, XTemplates are much more powerful than that.

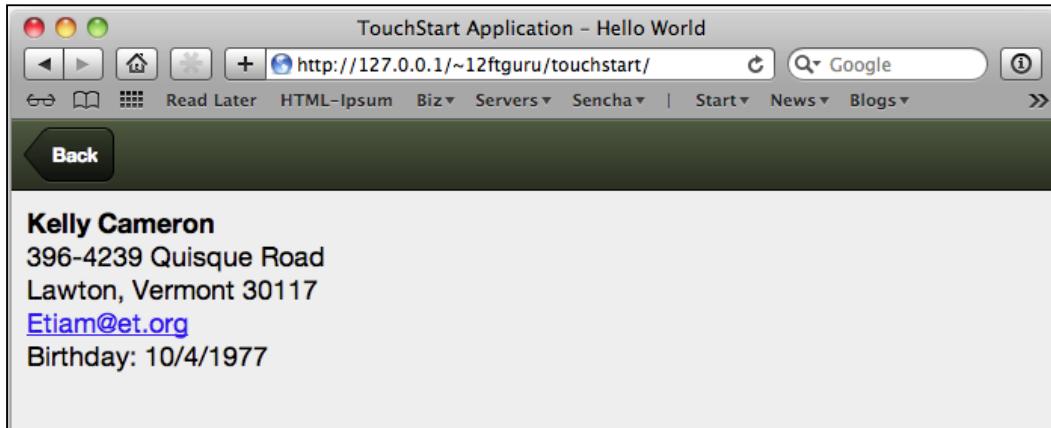
## **Manipulating data**

XTemplates also allow us to directly manipulate the data within the template in a number of ways. The first thing we can do is clean up that ugly birthday value!

Since the `birthday` value is listed in our model as being a `date` object, we can treat it like one in the template. We can replace the current birthday line of our template with the following:

```
'Birthday: {birthday:date("n/j/Y")}</div>'
```

This will use our value of `birthday`, and the `format` function `date.format` uses the string "`n/j/Y`" to convert `birthday` into a more readable format. These format strings can be found on the date page of the Sencha Touch API.



Sencha Touch includes a number of formatting functions that can be used in this fashion. Some of the functions include:

- `date`: This function formats a `date` object using the specified formatting string (the format strings can be found on the date page of the Sencha Touch API).
- `ellipsis`: This function truncates the string to a specified length and adds `...` to the end (note that `...` is considered to be part of the total length).
- `htmlEncode` and `htmlDecode`: These functions convert HTML characters (`&`, `<`, `>`, and `'`) to and from HTML.
- `leftPad`: This function pads the left side of the string with a specified character (good for padding numbers with leading zeros).
- `toggle`: This utility function switches between two alternating values.
- `trim`: This function removes any white space from the beginning and end of the string. It leaves spaces with the string intact.

The basic functions can be used inside the HTML of our XTemplate to format our data. However, XTemplate has a few additional tricks up its sleeve.

## Looping through data

In a list view, the XTemplate for the `itemTpl` component is automatically applied to each item in the list. However, you can also loop through your data manually using the following syntax:

```
'<tpl for=".">' ,  
'{name}</br>' ,  
'</tpl>'
```

When you use the `<tpl>` tag, it tells XTemplate that we are exiting the realm of HTML and making some decisions within the template. In this case, `<tpl for=". ">` tells the code to start a loop and use the root node of our data. The closing `</tpl>` tag tells the loop to stop.

Since we can have complex nested data with both XML and JSON, it can also be helpful to loop the data in other places besides the root node. For example, let's say we have an array of states and each state contains an array of cities. We can loop through this data as follows:

```
'<tpl for=".">' ,  
'{name}</br>' ,  
  
'<tpl for="cities">' ,  
'{name}</br>' ,  
'</tpl>'  
  
'</tpl>'
```

Our first `<tpl>` tag begins looping through our states, printing the name. After the name is printed, it looks for a child array within the individual state called `cities`. This time, when we use the variable `{name}`, it's inside our child loop so it prints the name of each city in the state before moving onto the next state in the loop.



Notice that when we use a field name inside our `<tpl>` tags, we do not use the curly braces like this: `{cities}`. Since we are outside the HTML section of our template, Sencha Touch assumes "cities" is a variable.

We can even access an array nested in each city, for example, postal codes, by adding another loop:

```
'<tpl for=".">>',
'{name}</br>',

'<tpl for="cities">>,
'{name}</br>',

'<tpl for="cities.postal">>,
'{code}</br>',
'</tpl>

'</tpl>

'</tpl>'
```

In this case, we have used `<tpl for="cities.postal">` to indicate that we will loop through the postal codes data array within the cities data array. Our other array loops execute as before.

## Numbering within the loop

When you are working inside a loop, it's often helpful to be able to count the cycles in the loop. You can do this by using `{#}` in your XTemplate:

```
'<tpl for=".">>',
'{#} {name}</br>',

'</tpl>'
```

This will print the current loop number next to each name in the loop. This will work in a similar fashion for nested data:

```
'<tpl for=".">>',
'{#} {name}</br>,

'<tpl for="cities">>,
'{#} {name}</br>',

'</tpl>

'</tpl>'
```

The first `{#}` will display where we are in the main loop, and the second `{#}` will display where we are in the `cities` loop.

## Parent data in the loop

In cases where we have nested data, it can also be helpful to be able to get to the parent properties from within the child loop. You can do this by using the `parent` object. Using our nested example with states, cities, and countries, this would look as follows:

```
'<tpl for=".">',
'{name}</br>',

'<tpl for="cities">',
'{parent.name} - {name}</br>',

'<tpl for="cities.postal">',
'{parent.name} - {code}</br>',
'</tpl>

'</tpl>

'</tpl>'
```

When inside our `cities` loop, `{parent.name}` would display the state name for that city. When we are inside our `cities.postal` loop, `{parent.name}` would display the city name associated with that postal code. Using this `{parent.fieldname}` syntax, we can access any of the parent's values from within the current child item.

## Conditional display

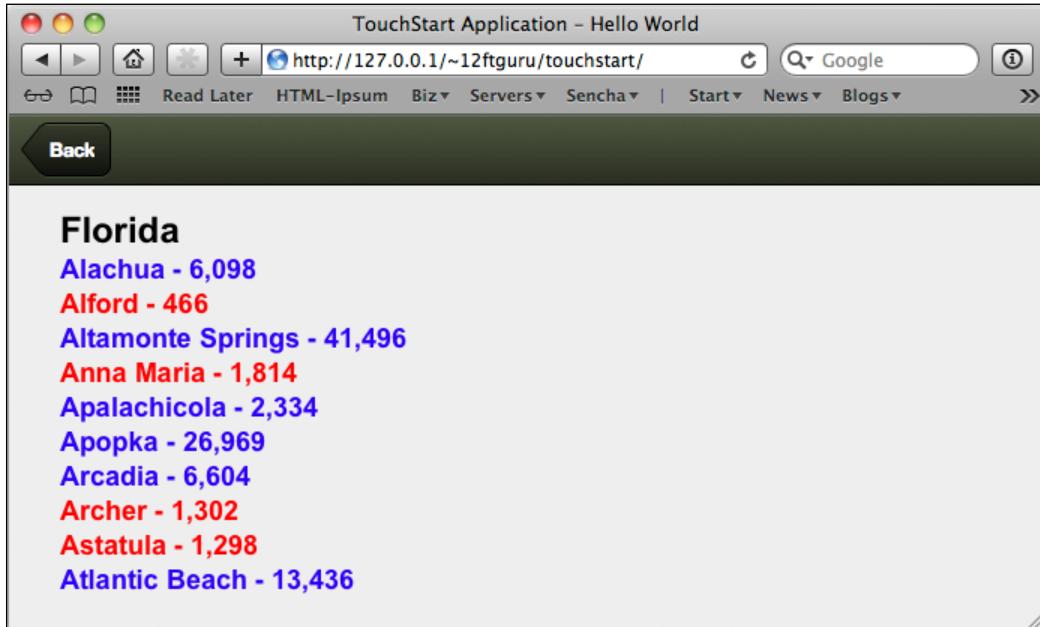
In addition to looping, XTemplates offer some limited conditional logic for use in your template. With the 2.x Version of Sencha Touch, we now have access to the full `if...then...else...` functionality. For example, we can use the `if` statement in our states and cities to only display cities with a population above 2,000:

```
'<tpl for=".">',
'{name}</br>',

'<tpl for="cities">',
'<tpl if="population > 2000">,
'{name}</br>',
'</tpl>',
'</tpl>',
'</tpl>'
```

If we want to color code our cities based on more than one population target, then we can use if...then...else like this:

```
'<tpl for=".">' ,
'  '{name}</br>' ,
'<tpl for="cities">' ,
'  '<tpl if="population > 2000"> ,
'    '<div class="blue">{name}</div>' ,
'  '<tpl elseif="population > 1000"> ,
'    '<div class="red">{name}</div>' ,
'  '<tpl else> ,
'    '<div>{name}</div>' ,
'  '</tpl> ,
'</tpl> ,
'</tpl>'
```



Now, you probably are already asking yourself why we are using `&gt;` and `&lt;` instead of `>` and `<`. The reason is that anything in our conditional statement needs to be HTML encoded for XTemplate to parse it correctly. This can be a bit confusing at first, but the key things to remember are as follows:

- Use `&gt;` instead of `>`.
- Use `&lt;` instead of `<`.
- Use equals as normal `==`. However, if you are comparing a string value, you have to escape the single quotes such as this: `'<tpl if="state == \'PA\'">'`.
- You will need to encode `"` if it is part of your conditional statement. So if you are searching for the word `"spam"` including the quotes, you would have to encode it as `&quot;spam&quot;`.

## Arithmetic functionality

In addition to conditional logic, XTemplates also support basic math functionality for the following:

- Addition `(+)`
- Subtraction `(-)`
- Multiplication `(*)`
- Division `(/)`
- Modulus – the remainder of one number divided by another `(%)`

For example:

```
'<tpl for=".">' ,
'{name}</br>',
'<tpl for="cities">' ,
'{name}</br>',
'Population: {population}</br>',
'Projected Population for next year: {population * 1.15}</br>',
'</tpl>',
'</tpl>'
```

This gives us our initial population value followed by a projected population of 1.15 times the current population. The math functions are included within the curly braces around our variable.

## Inline JavaScript

We can also execute arbitrary inline code as part of our XTemplate by placing the code within a combination of brackets and curly braces: { [...] }. There are also a few special attributes we can access within this code:

- `values`: This attribute holds the values in the current scope
- `parent`: This attribute holds the values of the current parent object
- `xindex`: This attribute holds the current index of the loop you are on
- `xcount`: This attribute holds the total number of items in the current loop

Let's take an example to elaborate on these attributes. We can make sure that our state and city names are in uppercase and the colors alternate on our list of cities by using the following XTemplate:

```
'<tpl for=".">' ,
  '{ [values.name.toUpperCase()] }</br>' ,
  '<tpl for="cities">' ,
  '<div class="{ [xindex % 2 === 0 ? "even" : "odd"] }>' ,
    '{ [values.name.toUpperCase()] }</br>' ,
  '</div>' ,
  '</tpl>' ,
'</tpl>'
```

In this case, we use `{ [values.name.toUpperCase()] }` to force the name of the state and the city to be in uppercase. We also use `{ [xindex % 2 === 0 ? "even" : "odd"] }` to alternate our row colors based on the remainder of the current count divided by 2 (the modulus operator).

Even with the ability to write inline JavaScript, there are a number of cases where you might require something a bit more robust. This is where the XTemplate member functions come into play.

## XTemplate member functions

An XTemplate member function allows you to attach a JavaScript function to your XTemplate and then execute it inside the template by calling `this.function_name`.

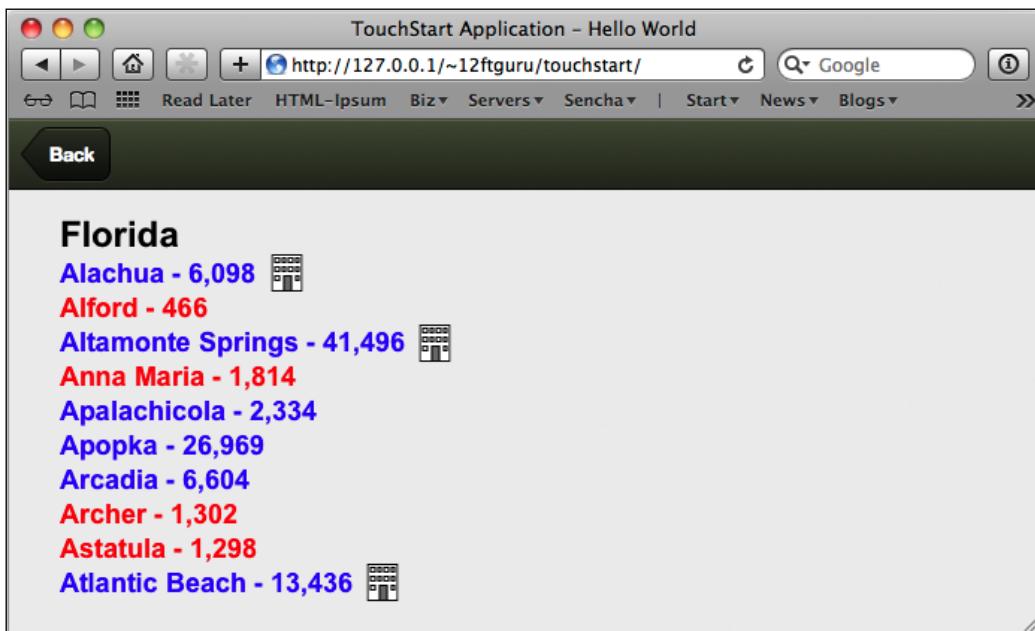
The functions are added to the end of the template, and a template can include multiple member functions. These member functions are wrapped in a set of curly braces in a fashion similar to that for listeners:

```
{  
    myTemplateFunction: function(myVariable) {  
        ...  
    },  
    myOtherTemplateFunction: function() {  
        ...  
    }  
}
```

We can use these member functions to return additional data to our template. Let's use our earlier states and cities example, and see how we can place a special icon next to larger cities based on multiple amounts of our data.

```
'<tpl for=".">' ,  
    '{name}'</br>' ,  
    '<tpl for="cities">' ,  
        '<div>{name} <tpl if="this.isLargeCity(values)"></tpl></div>' ,  
        '</tpl>' ,  
        '</tpl>' ,  
    '</tpl>' ,  
    {  
        isLargeCity: function(values) {  
            if(values.population >= 5000 && values.hasAirport && values.  
hasHospital) {  
                return true;  
            } else {  
                return false;  
            }  
        }  
    }  
}
```

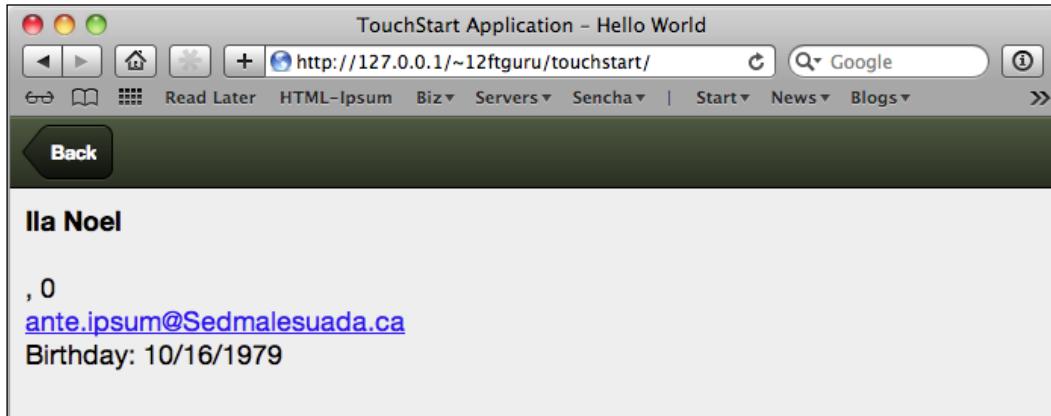
For this example, we have created a member function called `isLargeCity`, within which we pass our data. Since our function can execute any JavaScript we desire, we can use the result to control the template. We can then call the function inside our template with `{ [this.isLargeCity(values)] }` that will print our `bigCity.png` image based on the values in the data record.



We can also use our member functions to help us test for the presence or absence of data. This comes in very handy for controlling our template. For example, let's start with a contacts template that contains a name, address, and e-mail, similar to the following:

```
var myTemplate = new Ext.XTemplate(
    '<div style="padding:10px;"><b>{first} {last}</b><br>',
    '{address}<br>',
    '{city}, {state} {zip}<br>',
    '<a href="mailto:{email}">{email}</a><br>',
    'Birthday: {birthday:date("n/j/Y")}</div>'
);
```

If we have no data for address, city, and state, we will end up with some empty lines and a stray comma. Since our zip variable is an integer according to our model, it will show up as **0** if we don't have a value stored for it.



We need a way to check and see if we have data for these items before we print them onto the screen.

## The isEmpty function

As it turns out, native JavaScript is very problematic when it comes to detecting an empty value. Depending on the function, JavaScript might return the following:

- Null
- Undefined
- An empty array
- An empty string

For most of us, these are pretty much the same thing; we didn't get back anything. However, to JavaScript, these return values are very different. If we try to test for data with `if(myVar == '')` and we get back null, undefined, or an empty array, JavaScript will return `false`.

Fortunately, Sencha Touch has a handy little function called `isEmpty()`. This function will test for null, undefined, empty arrays, and empty strings, all in one function. However, Sencha Touch does not have an opposite function for `hasData`, which is what we really want to test for. Thanks to template member functions, we can write our own.

```
var myTemplate = new Ext.XTemplate(  
    '<div style="padding:10px;"><b>{first} {last}</b><br>',
```

```
'<tpl if="!Ext.isEmpty(address)">,
    '{address}<br>',
    '{city}, {state} {zip}<br>',
'</tpl>',
'<a href="mailto:{email}">{email}</a><br>',
'Birthday: {birthday:date("n/j/Y")}</div>'
```

We don't even need a member function for this data check. We can add `<tpl if="!Ext.isEmpty(address)">` to our template and check for the address in line with our template. The `Ext.isEmpty` function class takes the address data and determines whether it is empty or contains data, returning `true` or `false` respectively. If address is not empty, we print the address and if it is empty, we do nothing.

## Changing a panel's content with XTemplate. overwrite

In our previous examples, we have declared XTemplate as part of our panel or list, using `tpl` or `itemtpl`. However, it can also be helpful to overwrite a template programmatically after the list or panel is displayed. You can do this by declaring a new template and then using the panel's or list's `overwrite` command to combine the template and data, and overwrite the content area of your panel or list.

```
var myTemplate = new Ext.XTemplate(
    '<tpl for=".">' ,
        '{name}</br>',
        '<tpl for="cities">',
            '- {name}<br>',
        '</tpl>',
        '</tpl>',
    '</tpl>'
);

myTemplate.overwrite(panel.body, data);
```

Our `overwrite` function takes an element (`Ext` or `HTML`) as the first argument. So instead of just using `panel`, we need to use the `body` element of the panel as `panel.body`. We can then supply a record from a data store or an array of values as our second argument for the new template to use.

While XTemplates are extremely powerful for displaying our data, they are still very text heavy. What if we want to display data in a more colorful way? Let's take a look at Sencha Touch Charts to learn how we can do this.

## Sencha Touch Charts

So far, we have only looked at data stores and records as a way to display text data, but with the release of Sencha Touch Charts, we are now able to display complex graphical data as part of our applications.



These new components use data stores to display a wide range of chart and graph types, including the following:

- Pie Chart
- Bar
- Line
- Scatter
- Candlestick
- OHLC (Open High Low Close)
- Bubble

While a full exploration of the chart components would be worthy of a book by itself, we will provide an overview of how these components interact with the data store and, hopefully, boost your curiosity.

## Installing Sencha Touch Charts

As of Version 2.1, Sencha Touch Charts are built in to Sencha Touch and no longer require a separate download. At the time of writing, the Charts package license was available as part of the open source GPLv3 license or as a part of Sencha Complete or Sencha Touch Bundle.

## A simple pie chart

Let's start with a simple JavaScript file for our charts example, beginning with a data store:

```
Ext.application({
    name: 'TouchStart',
    launch: function() {
        var mystore = Ext.create('Ext.data.JsonStore', {
            fields: ['month', 'sales'],
            data: [
                {'month': 'June', 'sales': 500},
                {'month': 'July', 'sales': 350},
                {'month': 'August', 'sales': 200},
                {'month': 'September', 'sales': 770},
                {'month': 'October', 'sales': 170}
            ]
        });
    }
});
```

Our store declares two field types, `month` and `sales`, and our data array holds five sets of `month` and `sales` values. This will feed into a polar chart, in this case, a pie chart. After the store definition, we add the following:

```
Ext.create('Ext.chart.PolarChart', {
    background: 'white',
    store: mystore,
    fullscreen: true,
    innerPadding: 35,
    interactions: ['rotate'],
    colors: ["#115fa6", "#94ae0a", "#a61120", "#ff8809", "#ffd13e"],
    legend: {
        position: 'right',
        width: 125,
        margin: 10
    },
    series: [
        {
            type: 'pie',
            xField: 'sales',
            labelField: 'month',
            donut: 25,
            style: {
                miterLimit: 10,
```

```
        lineCap: 'miter',
        lineWidth: 2
    }
}
]
}) ;
```

Much like our other panel components, an `Ext.chart.PolarChart` class takes standard configurations such as `height`, `width`, and `fullscreen`. It also has a few special configurations, such as `innerPadding`, which is the padding between the axis and the series, and `background`, which is the background color behind the chart. The chart component also takes a `store` configuration option that we will set to our previously created `mystore` component.

The `interactions` section allows us to specify some visual tools that allow the user to interact with the chart. Each type of chart has its own set of interactions. The current interactions include:

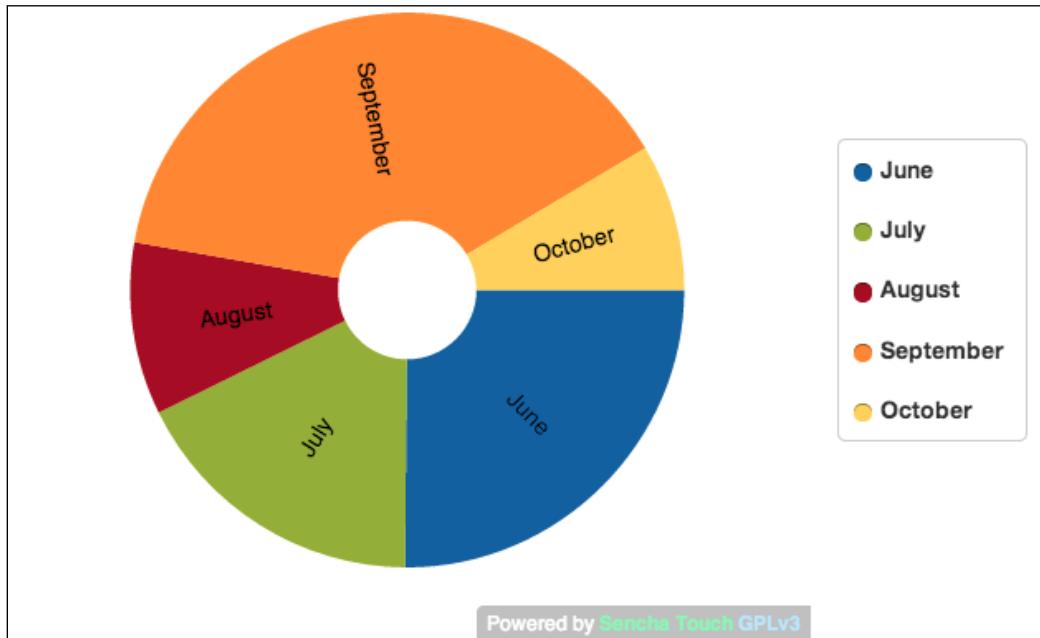
- `panzoom`: This interaction allows us to pan and zoom across the axes
- `itemhighlight`: This interaction allows us to highlight series data points
- `iteminfo`: This interaction allows us to display the details of a data point in a pop-up panel
- `rotate`: This interaction allows the rotation of the pie chart and radar series

Next we have a configuration for our chart's legend. This provides a color-coded reference for all our chart values. We can use a position configuration to designate how the legend should appear in portrait and landscape modes.

The final piece is our `series` configuration. In our example, we have set:

- The **type** of chart we will see
- Which **xfield** the chart will use to determine the size of the pie slices
- The value of **labelField** to use for the slices
- The **size** of the donut hole at the center of our pie chart
- The overall **style** of the chart

When we load it all up, our chart looks as follows:



If you click on any of the months on the legend, you can turn them on or off in the chart. This functionality happens automatically without any additional code. Our interaction setting also lets us click-and-drag to rotate the chart.

This kind of pie chart works well for very simple single-series data, but what happens if we have data for several years? Let's see how a bar chart might work to display this kind of data.

## A bar chart

For our bar chart, let's replace our chart data store with the following:

```
var mystore = Ext.create('Ext.data.JsonStore', {
    fields: ['month', '2008', '2009', '2010'],
    data: [
        {'month': 'June', '2008': 500, '2009': 400, '2010': 570},
        {'month': 'July', '2008': 350, '2009': 430, '2010': 270},
        {'month': 'August', '2008': 200, '2009': 300, '2010': 320},
        {'month': 'September', '2008': 770, '2009': 390, '2010': 670},
        {'month': 'October', '2008': 170, '2009': 220, '2010': 360}
    ]
});
```

This data set has multiple series of data that we need to display (five months, with three years for each month). An effective bar chart will need to display a row for each month and separate bars within the month for each of the years.

Next, we need to change our `PolarChart` to `CartesianChart` like this:

```
Ext.create("Ext.chart.CartesianChart", {  
    fullscreen: true,  
    background: 'white',  
    flipXY: true,  
    store: mystore,  
    interactions: ['panzoom'],  
    legend: {  
        position: 'right',  
        width: 80,  
        margin: 10  
    },  
    axes: [  
        {  
            type: 'numeric',  
            position: 'bottom',  
            grid: true,  
            minimum: 0  
        },  
        {  
            type: 'category',  
            position: 'left'  
        }  
    ],  
    series: [  
        {  
            type: 'bar',  
            xField: 'month',  
            yField: ['2008', '2009', '2010'],  
            axis: 'bottom',  
            highlight: true,  
            showInLegend: true,  
            style: {  
                stroke: 'rgb(40,40,40)',  
                maxBarWidth: 30  
            },  
            subStyle: {  
                fill: ["#115fa6", "#94ae0a", "#a61120"]  
            }  
        ]  
    );
```

Like our pie chart, the bar chart component takes configuration options for background, fullscreen, a data store, and the panzoom interaction. This option us allows to pan and zoom across our axes.

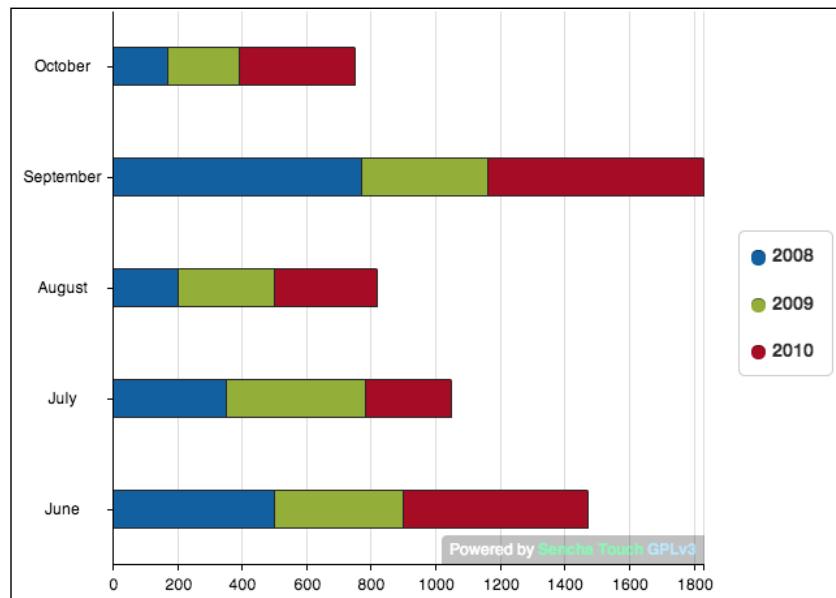
We then have our legend as before, followed by a new configuration option called axes. Since a bar chart operates along the x and y axes, we need to specify what type of data is being fed into each axis (in this case, the bottom and left axes).

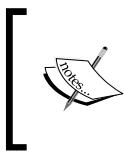
First up is our sales data for each year. The data is numeric, positioned at the bottom and given the title of sales. We also specify what our minimum value should be (this is the number that will appear on the far-left corner of our bar chart and will usually be 0).

The next axis is our category data (which will also be used for our legend). In this case, our position is left, and our title is Month of the Year. With this we complete our axes configuration.

Finally, we have our series configuration, which sets this up as a bar graph. Unlike our previous pie chart example, which only tracked sales data for a single point, the bar chart tracks sales data for two separate points (month and year), so we need to assign our xField and yField variables and declare an axis location. This location should match the axis where you are displaying numerical data (in our case, the data is on the y axis, which is at the bottom). We close out by using showInLegend to display our legend.

The final chart should look as follows:





Charts are an incredibly robust way to use stores to display data. We don't really have time to go through them all here, but you can explore all of the capabilities of Sencha Touch Charts at [http://docs.sencha.com/touch/2.2.0/#!/guide/drawing\\_and\\_chaining](http://docs.sencha.com/touch/2.2.0/#!/guide/drawing_and_chaining).

## Summary

In this chapter, we have explored the way data stores can be used to display both simple and complex data. We talked about binding, sorting, paging, and loading data stores. We then walked through using data stores with both lists and panels.

We also covered how to lay out your application by using XTemplates to control how the data from stores and records will appear. We explored how to manipulate and loop through our data inside an XTemplate, as well as how to use conditional logic, arithmetic, and inline JavaScript. We finished up our conversation on XTemplates by discussing member functions and some of their uses. We concluded our chapter with a look at using the Sencha Touch Charts package to display our store data graphically.

In our next chapter, we will explore how to put all of the information from our previous chapters together into a full-scale application.

# 8

## Creating the Flickr Finder Application

So far, we have looked at Sencha Touch components individually or in small, simple applications. In this chapter, we are going to create a well-structured and more detailed application using Sencha Touch. We will attempt to leverage all of the skills from our previous chapters to create an application that will let us search for photos taken near our location. This chapter will include:

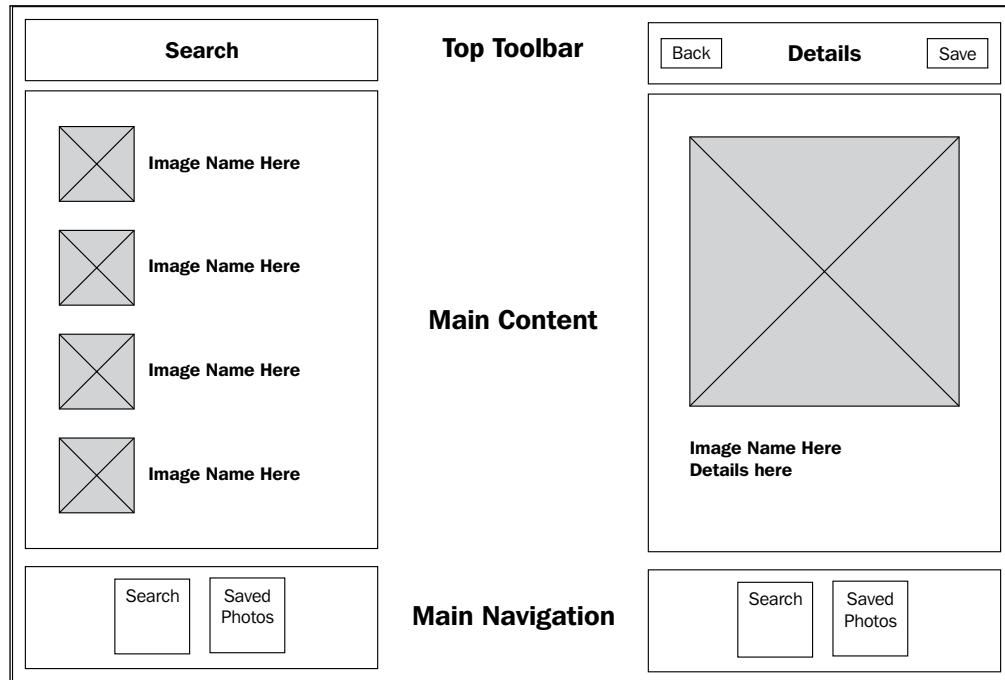
- An introduction to the **Model View Controller (MVC)** design pattern
- Setting up a more robust folder structure
- Setting up the main application files
- Using the Flickr API
- Registering components
- Setting up the `SearchPhotos` component
- Setting up the `SavedPhotos` component
- Adding the finishing touches to publish the application

### Generating the basic application

The basic idea for this application will be to use the Flickr API to discover photos taken near our location. We will also add the ability to save interesting photos we might want to look at later.

When you are first creating an application, it's always a good idea to sketch out the interface. This gives you a good idea of the pieces you will need to build and also allows you to navigate through the various screens the way a user would. It doesn't need to be pretty; it just needs to give you a basic idea of all the pieces involved in creating the application.

Aim for something very basic, such as this:

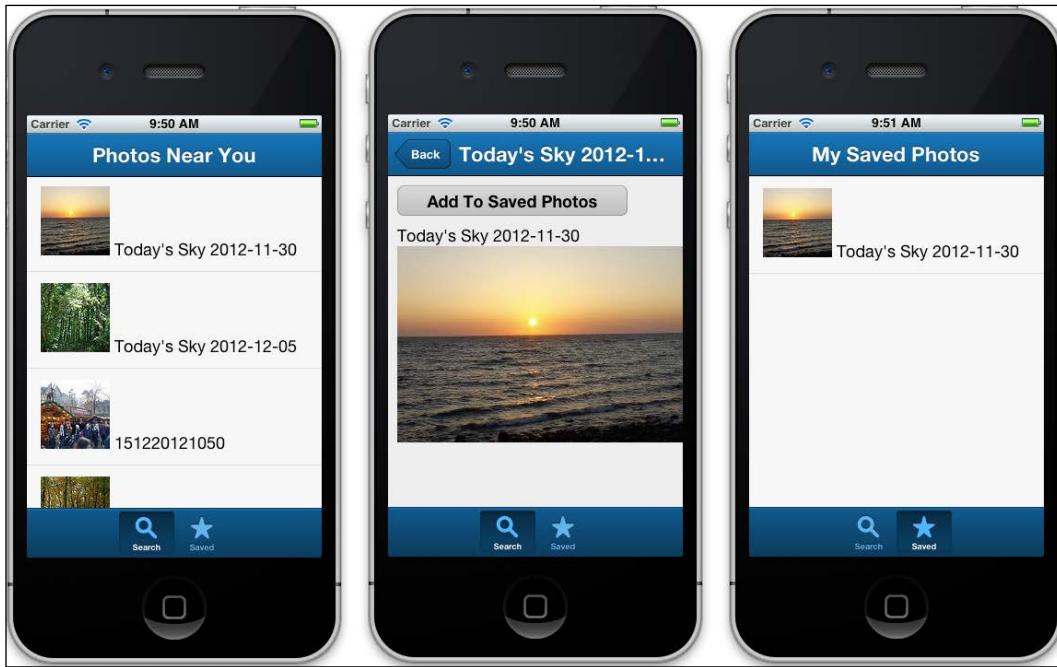


Next, you will want to tap your way through the paper interface, just as you would with a real application and think about where each tap will take the user, what might be missing, and what might be confusing for the user.

Our basic application needs to be able to display a list of photos as well as a close-up of a single photo. When we tap on a photo in the list, we will need to show the larger close-up photo. We will also need a way to get back to our list when we are done with the photo.

When we see a photo we like, we need to be able to save it, which means that we will need a button to save the photo, as well as a separate list of saved photos, and a close-up single view for the saved photo.

Once we are happy with the drawings, we can start putting together the code to make our paper mock-up into something like this:



## Introducing the Model View Controller

Before we get started with building our application, we should spend some time talking about structure and organization. While this might seem like a boring detour into application philosophy, it's actually one of the most critical considerations for your application.

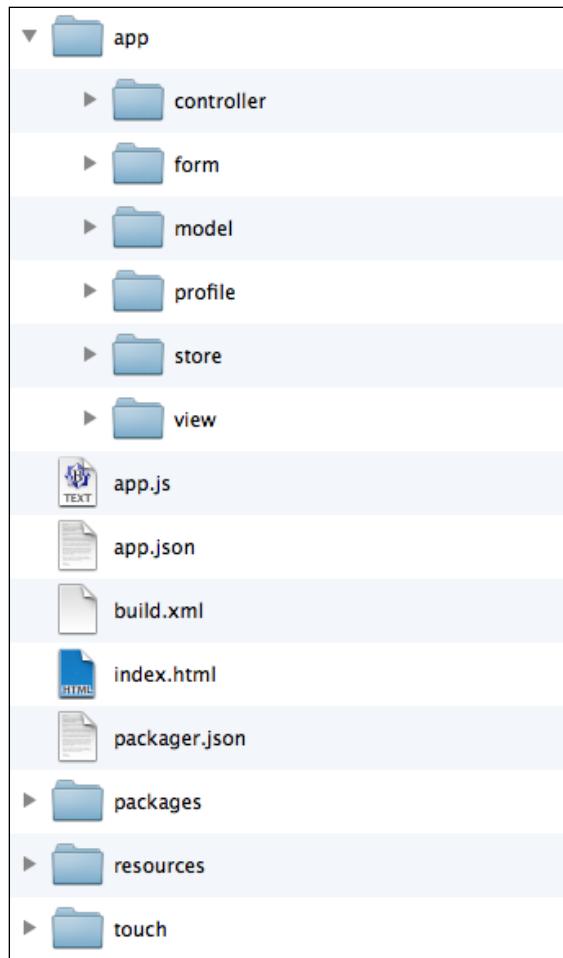
First, consider the monolithic application, with everything in one enormous file. It seems crazy, but you will encounter hundreds of applications that have been coded in just this fashion. Attempting to debug something such as this is a nightmare. Imagine finding the missing closing curly brace inside of a component array 750 lines long. Yuck!

The question then becomes one of how to break up the files in a logical fashion.

As discussed earlier in this book, the Model View Controller, or MVC, architecture organizes the application files based on the functionality of the code:

- Models describe your data
- Views control how the data will be displayed
- Controllers handle the user interactions by taking input from the user and telling the views and models how to respond based on the user's input

Sencha Touch also uses stores, which describe the storage and transmission of the data between components. When we split these pieces of the application apart, it means each part of your application will have separate files for each of these parts. Let's take a look at how this is structured:



This is a basic application skeleton output from Sencha Cmd, which we will be using for our FlickrFindr project. In the root directory, we have folders for:

- **app**: This folder contains our main application files
- **packages**: This folder is used for any external libraries we might need
- **resources**: This folder contains our CSS, SASS, icons, and various image files
- **touch**: This folder contains a copy of the Sencha Touch library

Inside our **app** directory, we have separate files for:

- **controller**: Our controllers will contain the functionality of our application.
- **form**: Our forms will control the appearance of any forms we use.
- **model**: Our models will describe the data we are using.
- **profile**: Our profiles will contain display information for different types of devices. This is not covered in this book, but a detailed explanation can be found on the Sencha website at <http://docs.sencha.com/touch/2.2.1/#!/guide/profiles>.
- **store**: Our stores determine how the application's data will be stored.
- **view**: Our views control the appearance of the application.

By splitting the files out this way, it is much easier to re-use code across applications. For example, let's say you build an application that has a model, store, controller, and views for a user. If you want to create another application that needs to deal with users, you can simply copy over the individual files for the model, store, views, and controller into your new application. If all the files are copied over, then the user code should work just as it did in the earlier application.

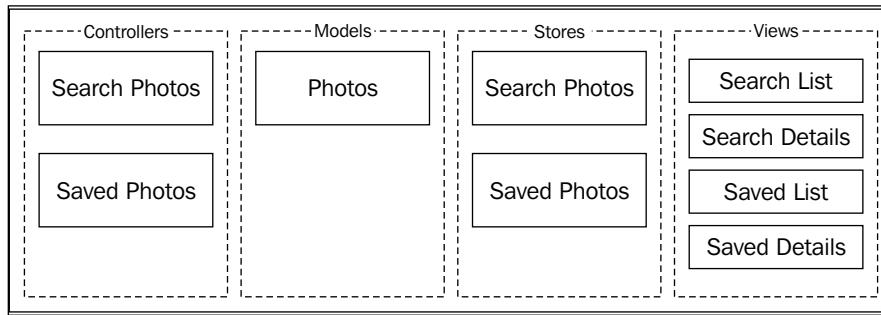
If we build a monolithic application, you would have to hunt through the code, grab out bits and pieces, and reassemble them in the new application. This would be a slow and painful process. By separating our components through functionality, it's much easier to re-use code between projects.

## Splitting up the pieces

The next thing we need to consider is how our application gets split into our separate MVC pieces. For example, if your application tracks people and what cars they own, you would likely have a model and controller for the people and a separate model and controller for the cars. You would also likely have multiple views for both cars and people, such as add, edit, list, and details.

In our application, we will be dealing with two different types of data. The first is search data for our photos and the second is our saved photos.

If we break this down into models, stores, views, and controllers, we get something like the following:



Our controllers are separated through functionality for **Saved Photos** and **Search Photos**.

Since they are dealing with the same type of data, each of our controllers can use the same **Photos** model, but they will need different stores (**Search Photos** and **Saved Photos**), since they're each using different actual data sets.

For views, our search needs a list view for **Search List** and a **Search Details** view. The saved photos will also need a view for the **Saved List** list and a view for editing/adding **Saved Details**.

Now that we have a clear map of the files we need, it's time to start building our application.

## Building the foundation with Sencha Cmd

In Version 1 of Sencha Touch, the setup process for an application was very manual and time consuming. However, the introduction of Sencha Cmd allows us to generate most of the core application files with a single command.

Sencha Cmd is a set of command-line tools that perform a number of basic tasks in Sencha Touch, such as:

- Generating an application skeleton that you can use as the basis for your application
- Generating controllers, forms, and models
- Building your application to "minify" and compress JavaScript and images for production applications
- Building your application as a standalone binary that you can sell in the App Store

Sencha Cmd has a number of additional uses and configuration options. More information on these can be found at <http://docs.sencha.com/touch/2.2.1/#!/guide/command>.

For this project, we will mainly be using the `generate` command to build our basic application, controllers, and models. However, first we need to get everything installed.

## Installing Sencha Cmd

Sencha Cmd is a separate download from our Sencha Touch code and can be found at: <http://www.sencha.com/products/sencha-cmd/download>. This download is available for Windows, OS X, and Linux (32 bit and 64 bit). When you unzip the downloaded file, you can double-click on it to install Sencha Cmd.



For this book, we are using Sencha Cmd Version 3 (at least Version 3.1.2 or greater is required). Detailed installation instructions can be found at <http://docs.sencha.com/touch/2.2.1/#!/guide/command>.

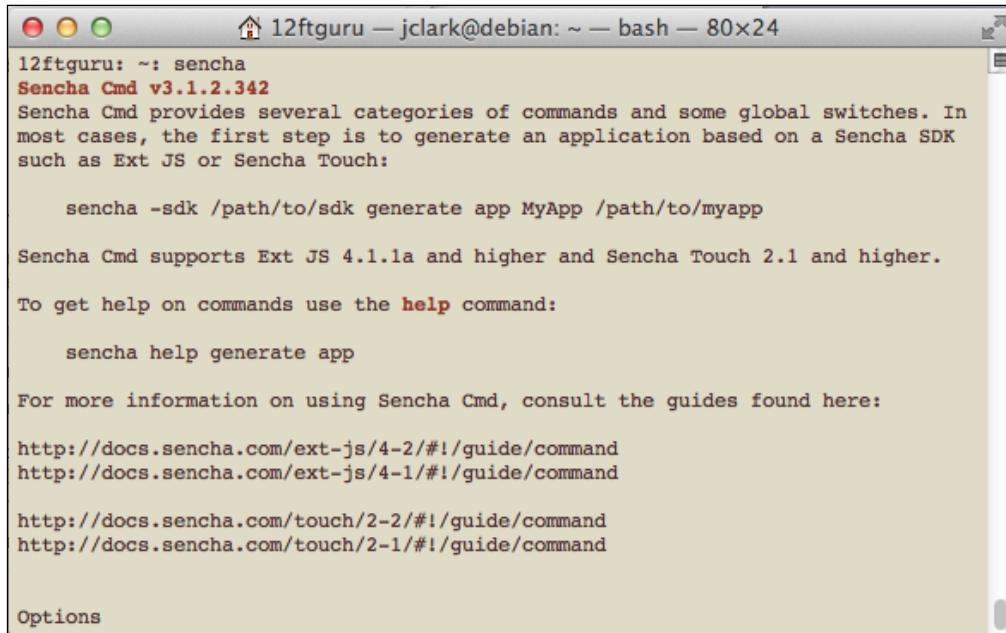
Once you have installed Sencha Cmd, you can open up the command line on your computer as follows:

- On Mac OS X, go to **Applications** and launch **Terminal**
- On Windows, go to **Start | Run** and type `cmd`

## *Creating the Flickr Finder Application*

---

Once your terminal is open, type `sencha`. You should see something like the following screenshot in your terminal:



A screenshot of a terminal window titled "12ftguru — jclark@debian: ~ — bash — 80x24". The window displays the help output for Sencha Cmd version 3.1.2.342. The text includes information about generating applications based on Sencha SDKs like Ext JS or Sencha Touch, command-line options for generating apps, URLs for Sencha Cmd guides, and a link to the Sencha Touch guide. At the bottom left of the terminal window, the word "Options" is visible.

```
12ftguru: ~: sencha
Sencha Cmd v3.1.2.342
Sencha Cmd provides several categories of commands and some global switches. In
most cases, the first step is to generate an application based on a Sencha SDK
such as Ext JS or Sencha Touch:

    sencha -sdk /path/to/sdk generate app MyApp /path/to/myapp

Sencha Cmd supports Ext JS 4.1.1a and higher and Sencha Touch 2.1 and higher.

To get help on commands use the help command:

    sencha help generate app

For more information on using Sencha Cmd, consult the guides found here:

    http://docs.sencha.com/ext-js/4-2/#!/guide/command
    http://docs.sencha.com/ext-js/4-1/#!/guide/command

    http://docs.sencha.com/touch/2-2/#!/guide/command
    http://docs.sencha.com/touch/2-1/#!/guide/command

Options
```

Now that we have Sencha Cmd installed, it's time to generate the skeleton for our application.

First, you will need to change to the directory where your Sencha Touch files are installed (not the Sencha Cmd files we just downloaded, but your original Sencha Touch 2.1 files):

```
cd /path/to/Sencha-touch-directory/
```

Sencha Cmd will use the files from this Sencha Touch directory to generate our application. From here, we use the `generate` command as follows:

```
sencha generate app FlickrFindr /path/to/www/flickrfindr
```

You will, of course, need to adjust the preceding path to match your own personal development environment. This will create our basic application with the directory structure we showed in the the *Introducing the Model View Controller* section.

In addition to the folders we covered earlier, you will also see a number of files that are already created. You can consult the Sencha Cmd documentation for a full listing of what all the files do, but for now we only want to look at the file called `app.js`.

The `app.js` file loads when our application launches and takes care of our basic setup. If you look beneath the comments at the top of the file, you should see something like this:

```
Ext.Loader.setPath({  
    'Ext': 'touch/src'  
});
```

This sets the path to our copy of the Sencha Touch Framework, which Sencha Cmd copied into a `touch` directory as part of our `generate app` command.

The few lines that follow set our namespace (`name`) for the application and handle our required files:

```
Ext.application({  
    name: 'FlickrFinder',  
    requires: [  
        'Ext.MessageBox'  
    ],  
    views: [  
        'Main'  
    ],
```

The `requires` section lists any internal or external libraries that we need for our application. The `Ext.MessageBox` component is included by default. We also have a `views` section where we can list any of the views our application requires. Separate sections can also be added for models, stores, and controllers. We will get to those a bit later.

The next few sections concern themselves with our application icon and startup screen (`startupImage`). These images can be modified by either replacing the existing image or changing the URL to point to a new image file.

The last part of our `app.js` file is the `launch` function. This gets called after all our components are loaded:

```
launch: function() {  
    // Destroy the #appLoadingIndicator element  
    Ext.fly('appLoadingIndicator').destroy();  
  
    // Initialize the main view  
    Ext.Viewport.add(Ext.create('FlickrFindr.view.Main'));  
}
```

This function removes the loading indicator and displays our main window. The `FlickrFindr.view.Main` file is in our `views` folder; we will be modifying that shortly.



Notice that the name of the file in our example is `FlickrFindr.view.Main`, which tells the application that this file is called `Main.js` and that it is located in our `app/view` folder.

If we had a significant number of views, we could split them up into directories inside the `views` folder. For example, we could have `search` and `saved` folders for our application. In that case, the `Details.js` view for our `search` folder would be `FlickrFindr.view.search.Details`, and the `Details.js` view for our `saved` folder would be `FlickrFindr.view.saved.Details`.

We will come back to this file a bit later, but for now, just familiarize yourself with the contents.

Now that we have an idea of how our application needs to be laid out, we have one last task to perform before we really get started. We need to get an API key from Flickr.

## Using the Flickr API

The majority of popular web applications have made an **API (Application Programming Interface)** available for use in other applications. This API works in pretty much the same way as our Sencha Touch Framework. The API provides a list of methods that can be used to read from, and even write data to, the remote server.

These APIs typically require a key in order to use them. This allows the service to keep track of who is using the service and curtail any abuses of the system. API keys are generally free and easy to acquire.

Go to the Flickr API site <http://www.flickr.com/services/api/> and look for the phrase **API Keys**. Follow the link and apply for an API key using the form provided. When you receive your API key, it will be a 32-character-long string composed of numbers and lowercase letters.

Each time you send a request to the Flickr API server, you will need to transmit this key as well. We will get to that part a bit later.

The Flickr API covers a little over 250 methods. Some of these require you to be logged in with a Flickr account, but the others only require an API key.

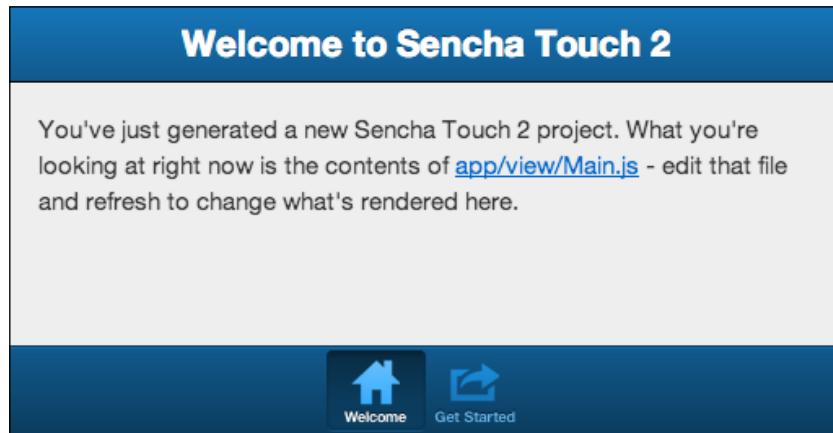
For our purposes, we will be using a single API method called `flickr.photos.search`, which requires no login. This method looks for photos based on some criteria. We will be using the current latitude and longitude of the device to get back photos within a specified distance from our current location.

Our search results come back to us as a big bundle of JSON format that we will need to decode for display.

Once we have the API key, we can begin setting up our models, stores, views, and controllers.

## Adding to the basic application

Let's start by taking a look at our raw generated application. Currently, if you load the application into your web browser, you should see something like this:



This view we are looking at comes from the `app/view/Main.js` file, which is a tab panel with two subpanels. We are going to replace this code with a much simpler tab panel:

```
Ext.define('FlickrFinder.view.Main', {
    extend: 'Ext.tab.Panel',
    xtype: 'main',
    requires: [
        'FlickrFinder.view.SearchPanel',
        'FlickrFinder.view.SavedPanel'
    ],
    config: {
        tabBarPosition: 'bottom',
        items: [
            { xtype: 'searchpanel' },
            { xtype: 'savedpanel' }
        ]
    }
});
```

Like the code we are replacing, this component extends a tab panel and sets an xtype of main.

In this context, xtype can seem a little confusing since Ext.tab.Panel already has xtype oftabpanel. However, since we are extending the tab panel, we are actually creating a new component, and this means that we can set xtype for the component. xtype is completely arbitrary, but it must be unique across all components (including Sencha's own components). The use of main as xtype is a general convention for the starting container of your application. By setting a custom xtype, we can easily find the container from the controllers that we will create later on.

In our requires section, we have two new views listed, one for FlickrFinder.view.SearchPanel and the other for FlickrFinder.view.SavedPanel. Further down in this code block, you will see two xtype values listed in the items section, searchpanel and savedpanel, which correspond to these two required files. We need to create these files next.

In your text editor, you need to create two new panel files in your app/view folder. The first one is called SearchPanel.js, and the code will look like this:

```
Ext.define('FlickrFinder.view.SearchPanel', {
    extend: 'Ext.Panel',
    xtype: 'searchpanel',
    config: {
        title: 'Search',
        iconCls: 'search',
        html: 'Search Panel'

    }
});
```

As with our Main.js file, we start out by setting the filename within the namespace to FlickrFinder.view.SearchPanel. However, this time we extend the Ext.Panel component instead of the tab panel.

We also set xtype to searchpanel, which should match the value we had in the items section of app/view/Main.js.

Lastly, we set up our config section with title, iconCls, and html.

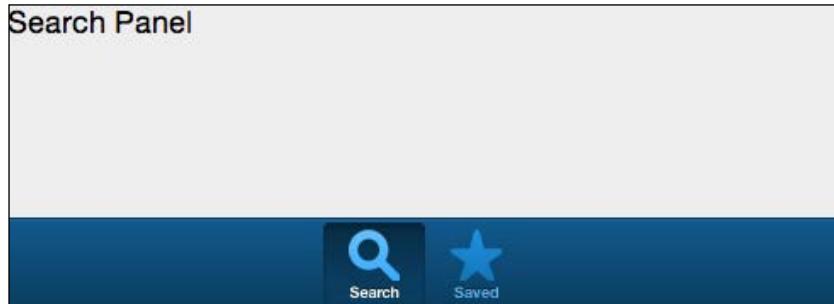
Next, we create a second panel in app/view called SavedPanel.js. This panel's code looks almost the same as our previous panel:

```
Ext.define('FlickrFinder.view.SavedPanel', {
    extend: 'Ext.Panel',
    xtype: 'savedpanel',
    config: {
```

```

        title: 'Saved',
        iconCls: 'favorites',
        html: 'Saved Panel'
    }
});
```

As you can see, we have just replaced the word `search` with the word `saved`. Once this second panel has been saved, you can reload the page to see this:



We now have our tab panel with two base panels, and we can switch back and forth between the two.

Right now, our application doesn't do a lot, but we can fix that. Let's start by adding a pair of controllers to our application.

## Generating controllers with Sencha Cmd

Much like our starter application, we can also generate controllers with Sencha Cmd. To do this, you will need to switch back to your terminal program and from there, change into your application directory (not the Sencha directory like we did when we generated the application). Changing into this directory lets Sencha Cmd know where to create the controller file:

```

cd /path/to/www/myapp
sencha generate controller SearchPhotos
sencha generate controller SavedPhotos
```

This will create two starter files for our controllers, and adds a reference to those files in `app.js`. If you open up `app.js` after running those commands, you should now see a `controllers` section like this:

```

controllers: [
    'SearchPhotos',
    'SavedPhotos'
]
```

If you open up one of the new controller files, you should see something like this:

```
Ext.define('FlickrFinder.controller.SearchPhotos', {
    extend: 'Ext.app.Controller',
    config: {
        // refs: {
        //     TODO: add refs here
        // },
        // control: {
        //     TODO: add event handlers here
        // }
    },
    // Called when the Application is launched, remove if not needed
    launch: function(app) {

    }
});
```

By creating these two empty controllers first, we can then add any models, views, and stores into our two controller files and leave `app.js` as it is (as it already includes our controllers).



In some older versions of Sencha Cmd, using the `generate controller` command would create the files but not add the `controllers` section to `app.js`. It's a good idea to check and make sure that the `controllers` section gets added or your files will fail to load. This will lead to errors and a lot of hair-pulling.

## A brief word about including files

When you split code out into separate files, the framework needs to have a basic understanding of what files need to be included in order to make the application work.

Both `app.js` and our `controller` files can include sections for models, views, and stores. These sections can be specified in either set of files, but the best practice is to include `controllers` in `app.js` and let the individual controllers include the models, stores, and views.

Other components can contain a required section for including files. For example, in our `main.js` view, we require the two panel views (`savedPanel` and `searchPanel`) that are used in the `items` section of our `main.js` file. The requires section is used for any dependencies that are used directly in the component.

We will see some examples of this when we create our models and stores.

## Creating the Photo data model

Both our search and saved photos will be dealing with the same set of information. This means we can create a single shared model called Photo.

Our photo data will be constrained in part by the data we can get back from the Flickr API. However, we also want to display the images as part of our search results. This means that we need to look at the Flickr API and see what is required to display an image from Flickr in our application.

If we take a look at <http://www.flickr.com/services/api/misc.urls.html>, we see that **Photo Source URLs** in Flickr has the following structure:

`http://farm{farm-id}.static.flickr.com/{server-id}/{id}_{secret}.jpg`

This means that, in order to display each photo, we need the following:

- `farm-id`: This variable indicates the group of servers the image is on
- `server-id`: This variable indicates the specific server the image is on
- `id`: This variable indicates a unique ID for the image
- `secret`: This variable indicates the code used by the Flickr API to route requests

These are all the variables that we get back as part of our `flickr.photos.search` request. We also get back the title for the photo, which we can use as part of our display. Now, we will use this information to create our model.

In the model directory, create a new file called `Photo.js`:

```
Ext.define('FlickrFindr.model.Photo', {
    extend: 'Ext.data.Model',
    config: {
        fields: [
```

```
{ name: 'id', type: 'int' },
{ name: 'owner', type: 'string' },
{ name: 'secret', type: 'string' },
{ name: 'server', type: 'int' },
{ name: 'farm', type: 'int' },
{ name: 'title', type: 'string' }
]
}
});
```

We start by defining our new model and extending `Ext.data.Model`. Next, we supply our field definitions in a series of `name type` values. If you leave the type blank, Sencha Touch will try to figure things out on its own, but it's a good idea to specify the type whenever possible.

Now that we have a shared `Photo` model defined, we need to set up our individual components starting with our `SearchPhotos` components.

## Making the SearchPhotos components

For searching photos, we will need a store and two views (list and details). When our application launches, the `SearchPhotos.js` controller will determine the user's current location. The controller will then load the store based on that location and display the photos in our list component. When the user taps on an item in the list, the controller will grab our details view and use it to display more information about the photo.

Let's start by creating our data store.

## Creating the SearchPhotos store

The data store is in charge of contacting the Flickr API and getting the photos for our list. We will also need to include some basic information for paging and a reference to our shared model file.

Create a new file in `app/store` called `SearchPhotos.js` and add the following code:

```
Ext.define('FlickrFindr.store.SearchPhotosStore', {
    extend: 'Ext.data.Store',
    requires: 'FlickrFindr.model.Photo',
    config: {
        model: 'FlickrFindr.model.Photo',
        autoLoad: false,
        pageSize: 25,
        proxy: {
            type: 'jsonp',
```

```

        url: 'http://ycpi.api.flickr.com/services/rest/',
        callbackKey: 'jsonpcallback',
        limitParam: 'per_page',
        reader: {
            type: 'json',
            root: 'photos.photo',
            totalProperty: 'photos.total'
        }
    }
);

```

Here, we define the `FlickrFindr.store.SearchPhotos` store and extend the standard `Ext.data.Store` store. Since we are using our previously created `Photo` model, we also need to add it to our `requires` section. We will be using a `jsonp` proxy for this store.

If you remember from *Chapter 6, Getting the Data In*, this proxy type is used for handling requests to a separate server, much like JSONP. These cross-site requests require a callback function in order to process the data returned by the server. However, unlike JSONP, the `jsonp` proxy will handle the callback functionality for us almost automatically.

We say almost because Flickr's API expects to receive the callback variable as:

```
jsonpcallback = a_really_long_callback_function_name
```

By default, the store sends this variable as:

```
callback = a_really_long_callback_function_name
```

Fortunately, we can change this by setting the following configuration option:

```
callbackParam: 'jsonpcallback'
```

The next section in the previous code snippet sets the URL for contacting the Flickr API, that is, `url: 'http://api.flickr.com/services/rest/'`. This URL is the same for any requests to the Flickr API.

There are a number of additional parameters we will need to send to the Flickr API to get what we need, but we will handle that later on in the controller.

Once we get our data back, we pass it to the reader:

```

reader: {
    type: 'json',
    root: 'photos.photo' ,
    totalProperty: 'photos.total'
}

```

Since our response from Flickr's API will be in JSON, we need to set `type: 'json'` in our `reader` function. We also need to tell the `reader` function where to start looking for photos in the `json` array that gets returned from Flickr. In this case, `root: 'photos.photo'` is the correct value. The last thing we need is `totalProperty` that tells the reader where to get the total number of photos we get back from Flickr. We will use this value for paging.

Now that we have our data model and store set up, we need two views: the `SearchPhotoList` view and the `SearchPhotoDetails` view.

## Creating the SearchPhotos list

We need two views for the `SearchPhotos` part of our applications: a list component and a panel for the details. We will start out by creating our list component.

Create a `SearchPhotoList.js` file in our `views` folder. This will be the first of our two views for `SearchPhotos`. Each view represents a single Sencha Touch display component. In this case, we will be using an `Ext.dataview.List` class for display and an `XTemplate` to control the layout of the list.

At the top of the file, our `XTemplate` looks as follows:

```
var SearchResultTpl = new Ext.XTemplate(
    '<div class="searchresult">',
    '',
    ' {title}</div>',
    {
        getPhotoURL: function(size, values) { /* Form a URL based on
            Flickr's URL specification: http://www.flickr.com/services/api/misc.
            urls.html */
            size = size || 's';
            var url = 'http://farm' + values.farm + '.static.flickr.com/' +
            + values.server + '/' + values.id + '_' + values.secret + '_' + size +
            '.jpg';
            return url;
        }
    });
});
```

The first part of our `XTemplate` supplies the HTML we are going to populate with our date. We start by declaring a `div` tag with the class `searchresult`. This gives us a class we can use later on to specify which photo result is being tapped.

Next, we have an image tag, which needs to include a Flickr image URL for the photo we want in the list. We could assemble this string as part of the HTML of our XTemplate, but we are going to take the opportunity to add some flexibility by making this into a function on our XTemplate.

Flickr offers us a number of sizing options when using photos in this way. We can pass any of the following options along as part of our Flickr image URL:

- `s`: This refers to a small square, 75 x 75
- `t`: This refers to a thumbnail, 100 on the longest side
- `m`: This refers to a small image, 240 on the longest side
- `-`: This refers to a medium image, 500 on the longest side
- `z`: This refers to a larger image, 640 on the longest side
- `b`: This refers to a large image, 1024 on the longest side
- `o`: This refers to the original image, either JPG, GIF, or PNG, depending on the source format

We want to set our function to take one of these options along with our template values and create the Flickr image URL. Our function first looks to see if we were passed a value for size, and if not, we set it to `s`, by default using `size = size || 's';`.

Next, we assemble the URL using our XTemplate values and the respective sizes. Finally, we return the URL for use in our XTemplate HTML. This will let us create a thumbnail for each of our images.

Now, we need to define our list and pass it the XTemplate and store we created previously. After the definition for our XTemplate, add the following code:

```
Ext.define('FlickrFindr.view.SearchPhotoList', {
    extend: 'Ext.dataview.List',
    alias: 'widget.searchphotolist',
    requires: [
        'FlickrFindr.store.SearchPhotosStore'
    ],
    config: {
        store: 'SearchPhotosStore',
        itemTpl: SearchResultTpl
    }
});
```

Here we define and extend, just as we have done with our other components.

Since we are using `SearchPhotosStore` to feed our list, we also need to have it in our `requires` section. Down in the `config` section, we have the basic `store` and `itemTpl` configurations for our `list` component.

## Creating the Navigation view

Now that we have `SearchPhotoList`, we need to add it to the `SearchPhotos` panel, but before we do that, we need to talk a bit about our functionality.

The idea of a list-and-details view is very common among applications: This comprises a list of items with limited information, which the user can select to view a more detailed page on the item. The details page typically includes a button or link back to the main list. In fact, this functionality is so common that Sencha has a built-in component for handling it, called a **Navigation view**.

The Navigation view functions much like a card layout, where only one of the items inside the container is visible at a time. However, the Navigation layout has two special functions:

- **push**: This function adds a new component to the Navigation view and shows the new component with an animated transition
- **pop**: This function removes a component from the Navigation view and shows the previous component with an animated transition

The Navigation view also adds a **Back** button each time you push a new container onto it. This allows you to deeply nest data and have the layout itself control your navigation. We will use this type of view to control the flow between our list and details views, which means that we need to make a change to our `SearchPanel.js` file.

First, locate the line that says:

```
extend: 'Ext.Panel',
```

Then change it to:

```
extend: 'Ext.navigation.View',
```

This single change turns our exiting panel into a Navigation view.

Next, we need to remove our HTML from the old panel and add in our `ShowPhoto` list. By adding it directly to the Navigation view, we will make it the first thing we see when the panel loads. To do this, change the line:

```
html: 'Search Panel'
```

To:

```
items: {
    xtype: 'searchphotolist',
    title: 'Photos Near You'
}
```

Later on in the controller, we will see how the details panel is added to the navigation view, but first we need to create the details view itself.

## Creating the SearchPhotoDetails view

As we said before, when the user clicks on a photo in the list, we want to be able to show a larger version of the photo and offer the user the opportunity to add this to a saved list of photos.

We will start out with a very basic panel for this:

```
Ext.define('FlickrFinder.view.SearchPhotoDetails', {
    extend: 'Ext.Panel',
    xtype: 'searchphotodetails',
    config: {
        tpl: '<div class="photoDetails"><h1>{title}</h1></div>',
        padding: 10,
        scrollable: {
            direction: 'vertical',
            directionLock: true
        },
        items: [
            {
                xtype: 'button',
                action: 'savephoto',
                text: 'Add To Saved Photos',
                width: 250,
                margin: '0 0 10 0'
            }
        ]
    }
});
```

Now that we have our two views, we need to edit our controller to make the views actually work.

## Creating the SearchPhotos controller

Our SearchPhotos controller needs to accomplish the following:

- Obtain the user's location
- Load photos from Flickr that are close to that location
- Allow the user to page through the photos
- Allow the user to select a photo from the list and view a larger version

The first two pieces of this will be accomplished inside the controller's `launch` function so that they happen automatically when the application starts. The others will be handled by separate functions that we attach to the events from our components.

Before we get to that, we want to include our store and view files and set up a few references to make it easier to address the components inside our controller.

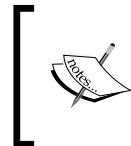
Open up our `SearchPhotos.js` file in the `controller` folder. If you followed along with us and generated the controller files from Sencha Cmd, you will see placeholders for `refs` and `controls`. On top of these placeholders and inside the `config` section, we need to add our views and our store:

```
Ext.define('FlickrFindr.controller.SearchPhotos', {
    extend: 'Ext.app.Controller',
    config: {
        views: [
            'FlickrFindr.view.SearchPhotoList',
            'FlickrFindr.view.SearchPhotoDetails',
            'FlickrFindr.view.SearchPanel',
            'FlickrFindr.view.Main'
        ],
        stores: [
            'FlickrFindr.store.SearchPhotosStore'
        ]
    }
})
```

The `views` and `stores` sections list the pieces of our application that this controller will be communicating with. Next, we will add some references to make it easier to access these pieces from inside the controller. In the `refs` section (after our `stores` section), add the following:

```
refs: {
    SearchPhotoList: 'searchphotolist',
    Main: 'main',
    SearchPanel: 'searchpanel'
}
```

The `refs` section consists of sets of name: target pairs. The name is arbitrary and the target can be `xtype` or a valid `Ext.ComponentQuery()` string.

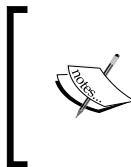


`Ext.ComponentQuery` lets you search for a specific component using a language similar to a standard CSS selector. For more details, see <http://docs.sencha.com/touch/2.2.0/#!/api/Ext.ComponentQuery>.



These `refs` will allow us to grab a component from anywhere inside our controller using `this.getName()`. For example, if we need to add a component to our `searchpanel` from inside our controller, we can simply use the following code:

```
var panel = this.getSearchPanel();
panel.add(myComponent);
```



It is important to note that the names will always be uppercased when using `this.getName()`. So if we had set up our reference to be `searchPhotoList: 'searchphotolist'` (with a lowercase "s"), we would still need to use `this.getSearchPhotoList()` (with a capital "S") to return the component.



Now that we have our references, we are going to skip the `controls` section right now and set up our `launch` function.

## Setting up the launch function

Our `launch` function is where we find the user's location and contact the Flickr API to get back our photos. The first thing we need to do is set up some default values—just in case the user declines to share their location, or we can determine their location.

```
launch: function() {
    var dt = Ext.Date.add(new Date(), Ext.Date.YEAR, -1);

    // Set some defaults.
    var easyparms = {
        "min_upload_date": Ext.Date.format(dt, "Y-m-d H:i:s"),
        "lat": 40.759017,
        "lon": -73.984059,
        "accuracy": 16,
        "radius": 10,
        "radius_units": "km",
        "method": "flickr.photos.search",
        "api_key": Your_API_Key_Goes_Here,
        "format": "json"
    };
}
```

The first part of the preceding code snippet creates a new date by using the current date and subtracting one year from it. This gives us the date exactly one year ago from this date. We will use this date to get back only photos that have been posted in the last year.

Our easyparms variable is a set of default parameters that we will send to Flickr's API if we cannot get a valid user location. These include our minimum upload date, latitude, and longitude (our defaults are for New York, in the middle of Times Square). We also include values for accuracy, radius, and radius units, to define how wide our search should be.

Lastly, we have a method (which is the API method we will use), your Flickr API key, and a format for the returned data (in this case, JSON). As previously noted in the *Using the Flickr API* section, you will need to obtain your own API key and use it here.

Now that we have some defaults, let's see if we can grab the user's location with Ext.util.Geolocation.

## Using Ext.util.Geolocation

The Ext.util.Geolocation component allows us to use the web browser to retrieve the user's location. This class is based on the Geolocation API specification built into most modern browsers. When this component is called, the user is prompted and asked if they are willing to share their location with the application. If they confirm, the component will return a latitude and longitude for the user's current location.

After our default easyparms definition, add the following code to access the Geolocation component:

```
var me = this;
var geo = Ext.create('Ext.util.Geolocation', {
    autoUpdate: false,
    timeout: 10000,
    // 10 second timeout
    listeners: {
        locationupdate: function(geo) {
            // Use our coordinates.
            easyparms = {
                "min_upload_date": Ext.Date.format(dt, "Y-m-d H:i:s"),
                "lat": geo.getLatitude(),
                "lon": geo.getLongitude(),
                "accuracy": 16,
```

```
"radius": 10,
"radius_units": "km",
"method": "flickr.photos.search",
"api_key": me.getApplication().api_key,
"format": "json"
};

var store = me.getSearchPhotoList().getStore();
store.getProxy().setExtraParams(easyparams);
store.load();
},
locationerror: function(geo, bTimeout, bPermissionDenied,
bLocationUnavailable, message) {
Ext.Msg.alert('Unable to set location.');
var store = me.getSearchPhotoList().getStore();
store.getProxy().setExtraParams(easyparams);
store.load();
}
}
);
};

geo.updateLocation();
```

This one is a bit long, so let's take it one piece at a time.

We begin by creating a new instance of the `Ext.util.Geolocation` component. We set `autoUpdate` to `false`, which keeps the component from trying to continually update our location. The idea is that the component will fire only once the application opens (this also keeps us from hammering the user's battery life). Next, we set a `timeout` value of 10000 milliseconds (10 seconds). This means that once the user confirms we are allowed to access their location, the component will spend 10 seconds attempting to get the location information before timing out and reporting an error.

That's really it for the configuration of `Ext.util.Geolocation`, but now we need to set up listeners for dealing with the data that comes back from the component.

We have two basic possibilities for event feedback from `Ext.util.Geolocation`:

- `locationupdate`: We get this if we got back a valid location for the user
- `locationerror`: We get this if something happened and we are unable to get a valid location for the user

Both of these events return our `Geolocation` object with some new data attached to it. In the case of `locationUpdate`, we get back:

- `accuracy`: This gives the last retrieved accuracy level of the latitude and longitude coordinates
- `altitude`: This gives the last retrieved height of the position specified in meters above the ellipsoid
- `altitudeAccuracy`: This gives the last retrieved accuracy level of the altitude coordinate, specified in meters
- `heading`: This gives the last retrieved direction of travel of the hosting device, specified in non-negative degrees between 0 and 359, counting clockwise relative to the true north (this reports `NAN` if the speed is zero)
- `latitude`: This gives the last retrieved geographical coordinate specified in degrees
- `longitude`: This gives the last retrieved geographical coordinate specified in degrees
- `speed`: This gives the last retrieved current ground speed of the device specified in meters per second

For this application, we are only interested in the latitude and longitude that we will pass in to our `easyparams` object. From the previous example code:

```
easyparams = {  
    "min_upload_date": Ext.Date.format(dt, "Y-m-d H:i:s"),  
    "lat": geo.getLatitude(),  
    "lon": geo.getLongitude(),  
    "accuracy": 16,  
    "radius": 10,  
    "radius_units": "km",  
    "method": "flickr.photos.search",  
    "api_key": me.getApplication().api_key,  
    "format": "json"  
};
```

This is the same as our default object, but with accurate coordinates for our location. We can then grab our `data store`, add the `easyparams` object, so it gets sent with our request, and call the store's `load` function to retrieve the photos.

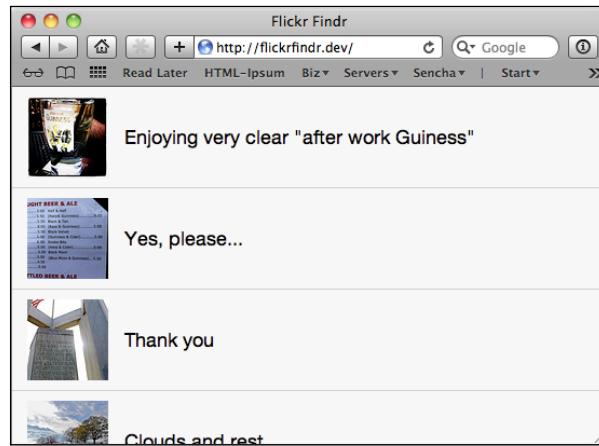
```
var store = me.getSearchPhotoList().getStore();  
store.getProxy().setExtraParams(easyparams);  
store.load();
```

This will cause the photos to appear in our `SearchPhotoList` component.

The `locationError` listener will fire if we can't get a location for the user. It simply alerts the user that we could not get a location and then loads our default set of `easyparams`, with the New York location.

The last thing we do in our `launch` function is fire the `updateLocation` function on our `Geolocation` object using `geo.updateLocation();`

At this point, you should be able to launch the application and see a list of photos near your location.



Now that we have our basic list working, we can swipe up or down to scroll. However, we need to add a bit more functionality before we finish up with our controller.

As you may have noticed, we can scroll but can't view any details yet. Also, we only loaded the first 25 photos from our search results. We need to have a way to tell the list that we want to tap an item to view the details and swipe an item to page through our list of photos. However, as it turns out, we don't want to tell the list anything. We actually want to listen to it.

## Listening to the list

Our list has a number of helpful events that it sends out in response to the user's interactions. The ones we are most concerned with are `itemswipe` and `itemtap`. We need to listen for those events in our controller and write functions to execute when those events occur. Let's start with our `itemtap` event.

In order to listen to an event, we need to add it into the `controls` section of controller as shown in the following code snippet:

```
SearchPhotoList: {
    itemtap: 'showSearchPhotoDetails'
}
```

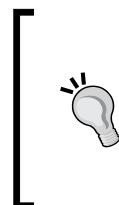
Since we previously made a reference to `SearchPhotoList: 'searchphotolist'` in our controller, we can use the shorthand `SearchPhotoList` to indicate that we are adding listeners to our list.

Here, we have specified that when our list fires the `itemtap` event, we want to execute a function called `showSearchPhotoDetails`. Next, we need to add that function to our controller.

Add a comma after the `launch` function in controller, and then add the following:

```
showSearchPhotoDetails: function(list, index, target, record) {
    var panel = Ext.create('FlickrFindr.view.SearchPhotoDetails', {
        title: record.get('title'),
        record: record
    });
    this.getSearchPanel().push(panel);
}
```

This function creates a new instance of our `SearchPhotoDetails` panel and sets its `title` and `record` based on the list item that was tapped (the record is passed along as part of the item tap event).



The Sencha Touch Docs available at <http://docs.sencha.com/touch/2.2.0/> will show a list of any values passed for a given event. Locate your component and then select an event from the **Events** list. A list of values passed by the event will be on the right-hand side of the event name. Clicking on the blue disclosure triangle will provide details on those event values.

By setting the `record`, we are also setting the data that will be used by our details template for display.

Lastly, we push the new panel onto our `SearchPanel` navigation view component. Remember, since we added `ref` for this in controller, we can retrieve it using `this.getSearchPanel()`. Once we push the new panel onto our navigation view, the list will be hidden and the new panel shows up with our **Back** button. Give it a try.



If you click on the **Back** button, the details panel is automatically removed from the stack, and the list will show up again.

Next, we need to handle the `itemswipe` function, so that it will load the next or previous page of items. In this case, we also need to do a bit of math to make sure that we don't try to page further than the beginning or end of our list. We will also need to do a bit of exploring to get the information we need from the event.

First, let's add our listener to `controller` by modifying the `controls` section to look like this:

```
control: {
  SearchPhotoList: {
    itemtap: 'showSearchPhotoDetails',
    itemswipe: 'pageChange'
  }
}
```

Next, we need to add our `pageChange` function after our previous `showSearchPhotoDetails` function. We will use this function to figure out the direction in which the user has swiped, so we know if we should go forward or backward in our paging.

From the Sencha Docs, we can see that the `itemswipe` event returns the following:

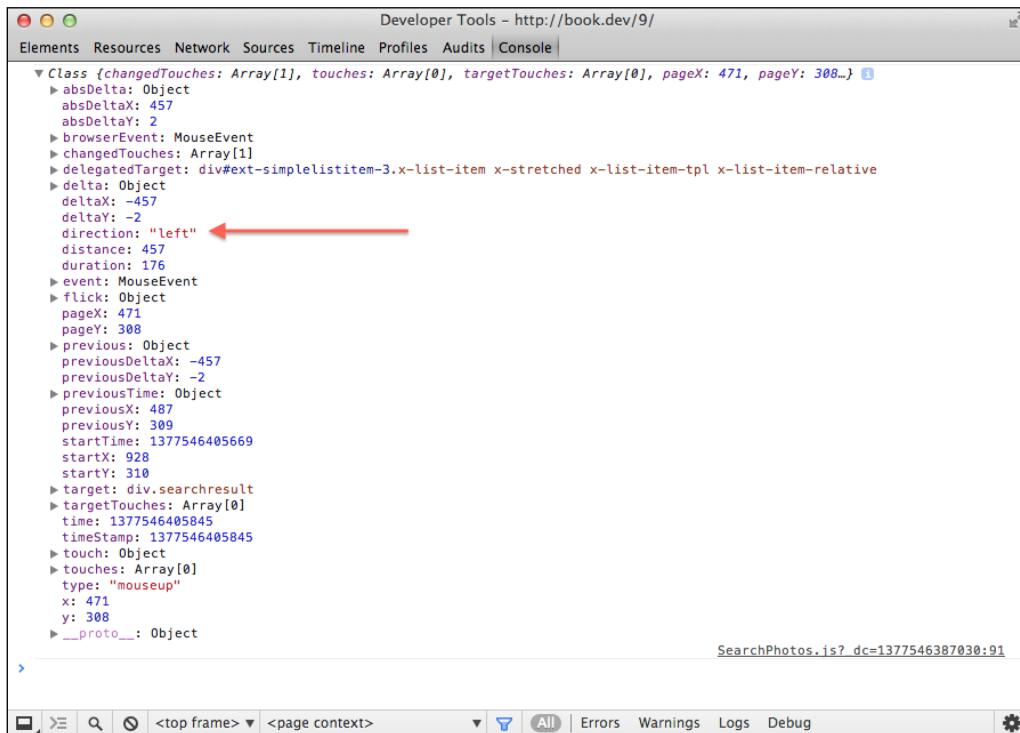
- `this`: This is our list component
- `index`: This is the index of the item swiped
- `target`: This is the element or `DataItem` swiped
- `record`: This is the record associated with the item
- `e`: This is the event object
- `eOpts`: This is the options object passed to `Ext.util.Observable.addListener`

The direction that the user is swiping is buried inside the event object `e`.

We can find the values we need by outputting the event object to the console log. So to begin with, let's make our `pageChange` function look like this:

```
pageChange: function(list, index, target, record, e, eOpts) {  
    console.log(e);  
}
```

If we reload our application and swipe one of the items in our list, we should see a listing for `e` in our console. Click on the disclosure triangle next to the listing to see all the details as follows:



From the details in the preceding screenshot, we can see that we get a lot of information back from the event, but the piece we need is just the `direction`. This means we can test for `e.direction` in our function to see which way we need to page the list of photos.

```
pageChange: function(list, index, target, record, e, eOpts) {
    console.log(e);
    var store = this.getSearchPhotoList().getStore();
    if(e.direction == 'right') {
        if(store.currentPage != 1) {
            store.previousPage();
        }
    } else {
        var total = store.getTotalCount();
        var page = store.getPageSize();
        if(store.currentPage <= Math.floor(total/page)) {
            store.nextPage()
        }
    }
}
```

First, we get our store by grabbing the list and calling `getStore()`. Next, we test whether our swipe direction is to the left or not. If the swipe goes to the left, we are paging back. If our current page is 1, we don't want to go backwards. If our page is greater than 1, we page back using `store.previousPage()`.

If we have swiped to right, we need to make sure we are not at the last page before we try to go to the next page. We do this by grabbing the total number of photos and `pageSize` from `store`. By dividing the total number of photos by the page count and rounding it off (`Math.floor`), we can get the number of the last page. We can then compare that to `currentPage` and decide whether we need to move forward to the next page. You should now be able to swipe back and forth across an item to navigate the pages of the list.

Now that we can view our photos in full size, let's set up a `savedphoto` component that will allow us to save a link to any photos we like.

## Building the SavedPhotos components

Our `SavedPhotos` components will need to store the information for a single photo from our search results. We will also need a list view for our saved photos and a details view, just like our previous `SearchPhotosList` and `SearchPhotoDetails` models.

## Creating the SavedPhotos store

Since our `SavedPhotos` and `SearchPhotos` components are storing the same type of data, we don't need to create a new model. We can just use our `Photo.js` model. However, we do need a separate data store, one that will store our `Photo` model locally.

Let's create a new file called `SavedPhotosStore.js` in our `app/store` folder and add the following code:

```
Ext.define('FlickrFindr.store.SavedPhotosStore', {
    extend: 'Ext.data.Store',
    requires: 'FlickrFindr.model.Photo',
    config: {
        model: 'FlickrFindr.model.Photo',
        autoLoad: true,
        pageSize: 25,
        storeId: 'SavedPhotosStore',
        proxy: {
            type: 'localstorage',
            id: 'flickr-saved'
        }
    }
});
```

Here, we just create our `FlickrFindr.store.SavedPhotosStore` class and extend `Ext.data.Store`. We also reuse our `FlickrFindr.model.Photo` model and set it as part of our required files. We also want this store to load up when the application launches (`autoLoad: true`) and we want it to set the page size to 25. Since it is grabbing local data, this should not present a huge load for the application.

For this store, we are going to include a `storeId`, so we can grab the store later on in our controller. We set our proxy to store the data locally and assign the proxy an `id` component, `flickr-saved`, that will be used to store our data.

Once we are finished with the `SavedPhotosStore.js` file, we will need to add it to our `SavedPhotos.js` controller. Open the `controller` file and add the following in the `config` section:

```
stores: [
    'FlickrFindr.store.SavedPhotosStore'
]
```

This will make sure our store gets loaded. Next, we need to set up our two views for list and details.

## Creating the SavedPhoto views

For the `SavedPhoto` views, we need a list and a details view. These views will be very close to what we already have for our `SearchPhotosList` and `SearchPhotoDetails` models. In fact, we can start by making copies of those two files and tweaking our layouts a bit.

In the `views` folder, make a copy of `SearchPhotoList.js`, and rename it as `SavedPhotoList.js`. You will also need to replace all the occurrences of `SearchPhoto` and `searchphoto`, with `SavedPhoto` and `savedphoto`, respectively (remember that JavaScript is case-sensitive). Your code should look as follows:

```
var SavedResultTpl = new Ext.XTemplate(
    '<div class="savedresult">',
    '',
    ' {title}</div>',
    {
        getPhotoURL: function(size, values) {
            size = size || 's';
            var url = 'http://farm' + values.farm + '.static.flickr.com/' +
            values.server + '/' + values.id + '_' + values.secret + '_' + size +
            '.jpg';
            return url;
        }
    });
});

Ext.define('FlickrFindr.view.SavedPhotoList', {
    extend: 'Ext.dataview.List',
    alias: 'widget.savedphotolist',
    requires: [
        'FlickrFindr.store.SavedPhotosStore'
    ],
    config: {
        store: 'SavedPhotosStore',
        itemTpl: SavedResultTpl
    }
});
```

You will notice that we have created a duplicate of the original  `SearchResultTpl` template in this file. If we wanted, we could simply reuse our `FlickrFindr.view.SearchResultTpl` class from the `SearchPhotos.js` file. It is perfectly fine to reuse the template, but this allows us the option of changing the look of saved photos in a list if we choose to.

Other than that, the file is largely the same as our `SearchPhotosList.js` file.



While it might seem a bit redundant to have two files that are so similar, it should be noted that they both read from different data stores, and that they need to be addressed differently by the controllers. It also gives us the opportunity to tweak the look of our different views later on.

For our `SavedPhotoDetails` view, we will take a similar approach. Copy the `SearchPhotoDetails.js` file to your `views` folder and rename it as `SavedPhotoDetails.js`. This file will display a single saved photo. However, unlike the details for our search photos, this saved photo details panel will get a **Remove** button instead of a **Save** button.

You will need to modify the file to change the **Save** button as follows:

```
Ext.define('FlickrFindr.view.SavedPhotoDetails', {
    extend: 'Ext.Panel',
    xtype: 'savedphotodetails',
    config: {
        tpl: '<div class="photoDetails"><h1>{title}</h1></div>',
        padding: 10,
        scrollable: {
            direction: 'vertical',
            directionLock: true
        },
        items: [
            {
                xtype: 'button',
                action: 'removephoto',
                text: 'Remove From Saved Photos',
                width: 250,
                margin: '0 0 10 0'
            }
        ]
    }
});
```

This is much the same as the `SearchPhotoDetails` file we created earlier; we have switched the names and changed our **Add** button to a **Remove** button. We will add the functionality for these buttons into our controllers in just a moment.

First, as we did with `SearchPhotosList`, we need to add `SavedPhotosList` to our `SavedPanel.js` file, and change it to extend `Ext.navigation.View` instead of `Ext.Panel`.

Open up `SavedPanel.js` and modify the code to make it look like this:

```
Ext.define('FlickrFindr.view.SavedPanel', {
    extend: 'Ext.navigation.View',
    xtype: 'savedpanel',
    config: {
        title: 'Saved',
        iconCls: 'favorites',
        items: [
            {
                xtype: 'searchphotolist',
                title: 'My Saved Photos'
            }
        ]
    });
});
```

Once we have the two views, we will need to add them into our `SavedPhotos.js` controller. Open the `app/controller/SavedPhotos.js` file and add the following code inside the `config` section:

```
views: [
    'FlickrFindr.view.SavedPhotoList',
    'FlickrFindr.view.SavedPhotoDetails'
]
```

Now we can start hooking up the rest of the controllers. We will start by taking a trip back to our `SearchPhotos.js` controller to hook up the **Add** button.

## Finishing up the Add button in SearchPhotos

Open up `SearchPhotos.js` in the `controller` folder; let's add a control for our **Save** button. In the `control` section, underneath our `SearchPhotoList` control, we add the control for the button as follows:

```
'button[action=savephoto]': {
    tap: 'savePhoto'
}
```

Next, we need to add our `savePhoto` function after our previous function definitions:

```
savePhoto: function(btn) {
    var rec = btn.up('searchphotodetails').getRecord();
    var store = Ext.data.StoreManager.lookup('SavedPhotosStore');
    rec.save({
        callback: function() {
            store.load();
            this.getMain().setActiveItem(1);
        }
    }, this);
}
```

We need two pieces to make this function work: the record from our details panel and the Saved Photos store, so we can load `SavedPhotoList` once the record is saved.

We grabbed the record by using the `up` function on our button to look for our `searchphotodetails` panel and then used `getRecord()`. We used `StoreManager` to lookup the store by its unique ID.

Next, we used the model's `save()` function to save the model using the model's proxy (not the store's proxy). We then used the `callback` function to load the store and switch our view after the model had been saved successfully.



You will notice that we also have an option `this` set on the end of the `save` function. As part of the `save` function, we can set the scope of our `callback` function, which you may remember from earlier in the book. By setting the scope to `this`, when we reference `this` inside our function (`this.getMain()`), we are talking about the controller and not the function itself.

Now that you have our function set up, you should be able to reload the application and save photos. We still need to be able to access the details for Saved Photos and remove the ones we no longer want.

## Updating the SavedPhotos controller

Inside our `SavedPhotos` controller, we need to add some refs and controls just like we did in the `SearchPhotos` controller.

Open the `SavedPhotos.js` file and modify the `refs` and `controls` sections like this:

```
refs: {
    SavedPhotoList: 'savedphotolist',
    SavedPanel: 'savedpanel'
},
control: {
    SavedPhotoList: {
        itemtap: 'showSavedPhotoDetails',
        itemswipe: 'pageChange'
    },
    'button[action=removephoto]': {
        tap: 'removePhoto'
    }
}
```

This gives us `refs` for our list and panel (we don't need one for `main`), and `controls` for three functions that will work in pretty much the same way as our `SearchPhotos` functions.

Let's start with the `showSavedPhotoDetails` function and add the following after the `config` section:

```
showSavedPhotoDetails: function(list, index, target, record) {
    var panel = Ext.create('FlickrFindr.view.SavedPhotoDetails', {
        title: record.get('title'),
        record: record
    });
    this.getSavedPanel().push(panel);
}
```

Much like our previous `showSearchPhotosDetails` function, this creates a new copy of our `SavedPhotoDetails` view, assigns a title and record, and then pushes it onto our `SavedPanel`.

Next, we have our `pageChange` function. You can copy and paste this one from our `SearchPhotos.js` controller:

```
pageChange: function(list, index, target, record, e, eOpts) {
    console.log(e);
    var store = this.getSavedPhotoList().getStore();
    if(e.direction == 'right') {
        if(store.currentPage != 1) {
            store.previousPage();
        }
    } else {
        var total = store.getTotalCount();
        var page = store.getPageSize();
        if(store.currentPage <= Math.floor(total/page)) {
            store.nextPage();
        }
    }
}
```

The only line we need to change in the preceding code snippet is the third line, where we get `store` for our `SavedPhotoList`. Other than that, this function accomplishes the same outcome as it did in our other controller; it detects the swipe from the user and pages back and forth through the results.

The last piece we need is our `removePhoto` function. This one will be a bit different. When we remove a photo from our list of saved photos, we need to pop the details view off our `SavedPanel` navigation view instead of changing the view:

```
removePhoto: function(btn) {
    var rec = btn.up('savedphotodetails').getRecord();
    var store = Ext.data.StoreManager.lookup('SavedPhotosStore');
    rec.erase({
        callback: function() {
            store.load();
            this.getSavedPanel().pop();
        }
    }, this);
}
```

For this function, we used the `erase()` method to remove the record from our local storage. We then loaded the store as before and used the `pop()` function to remove our details view. When this view is removed, our `SavedPanel` navigation view will automatically switch back to `SavedPhotosList`.

## Polishing your application

Now that we've finished our application, we will want to add some finishing touches to really make our application shine and add a level of professionalism to the completed product. The good news is that all of these are easily and quickly implemented.

## Adding application icons and startup screens

As mentioned back in *Chapter 1, Let's Begin with Sencha Touch*, users can navigate to your web application, and then choose to save it to the desktop of their mobile device.



In our current application, when someone installs it in this fashion, the default Sencha icon is displayed. However, you can modify the default icon that is displayed on the home screen.

The references folder contains all of the icons that your application will use for various devices. It also contains a startup folder with images for the startup screens used by the application on various devices.

Any of these images can be edited to customize your application's appearance. Just make sure you save them in the same format, the same size, and with the same name.

## Improving the application

There's still plenty of room for improvement in our application, but we will leave this as extra credit for the reader. Some things you might want to try are as follows:

- Allow the user to rename photos when they are saved
- Add an expert search, where you can set your location manually or widen the search radius
- Change the theme and make XTemplates more appealing
- Add the ability to save locations as well as photos

Try using the MVC organization techniques we have covered in this chapter to expand the application and sharpen your skills.

## Summary

In this chapter, we gave you an introduction to the MVC design pattern. We talked about setting up a more robust folder structure and created your main application files. We started our application with an overview of Flickr API and explored how to register our various model, view, and controller components. We then set up our components for the `SearchPhotos` and `SavedPhotos` components. We wrapped up the chapter with some hints for putting the finishing touches on your application and talked about a few extra pieces you might want to add to the application.

In the next chapter, we will cover a few advanced topics such as building your own API's, creating offline applications using a manifest system, and compiling applications with a program such as PhoneGap.

# 9

## Advanced Topics

In this chapter, we will explore a few high-level topics designed to point you in the right direction when building Sencha Touch applications, such as the following:

- Talking to your own server
- Going offline
- Compiling your application
- Getting into the marketplace

### Talking to your own server

Up to this point, we have used local storage as the way to create a database directly on the device that is running our program. While this is very useful, it can also be limiting in a few ways listed as follows:

- If any data is stored in the device, you cannot view it from another device
- If the device is stolen/broken/lost or otherwise unavailable, you also lose its data
- Options for sharing are limited to transferring copies of the data
- Collaborative editing of the data is not possible

Each of these concerns can be addressed by storing the data in an external database, such as MySQL, PostgreSQL, or Oracle. These databases can run on the same server as our application and handle multiple connections from different devices. Since all of the devices contact a single central database, sharing data across devices becomes much easier to accomplish.

Unfortunately, the Sencha Touch framework doesn't communicate directly with these types of external databases. In order to use a Sencha Touch application along with an external database, we need to use a third-party API or create our own. On the plus side, this means we can use any database we want on which to store our data. However, it also means that we will need to write a bit more code in order to connect Sencha with an external database.

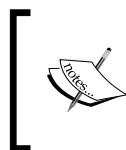
## Using your own API

In the previous chapters, we have learned about using external APIs to work with data from services such as Flickr and Google. External APIs make it possible to grab data stored in the databases for these various services, but what about when you need to get data in and out of your own database server?

As it turns out, the best way to do this using Sencha Touch is to create your very own API. In order to do this, we need to step back and talk a little bit more about what an API is and what it does.

At its most basic level, an API serves as a translator between the storage and interface parts of the application. The frontend makes a request to the API for data (say, a list of contacts) and the API pulls information from the database. The API then translates that data into JSON or XML and sends it back to the frontend for display.

While this might seem an unnecessary separation for an application, it actually has a number of benefits. Firstly, it allows the backend and the frontend to be written in different programming languages. This is important to us because JavaScript, while being a wonderful language for creating interfaces, is not a great tool for talking to more robust database systems, such as MySQL, PostgreSQL, Microsoft SQL Server, and Oracle. The code for an API can be created in a database-friendly language, such as PHP, RUBY, or PERL.



We will be using PHP for our examples, but the choice of API language is entirely up to you. We are also going to be very general when covering the PHP side of things. Our goal is to communicate the concept rather than provide specific PHP code.



The second benefit is that multiple applications can use the API to access the data. This makes it much easier to share data between users and also makes it possible to provide the same data set to completely different applications (as the Flickr API does). We don't even have to care about which programming language has been used to build the frontend as the API handles the translation.

Let's re-examine our FlickrFindr store to explore how this works:

```
Ext.define('FlickrFindr.store.SearchPhotosStore', {
    extend: 'Ext.data.Store',
    requires: 'FlickrFindr.model.Photo',
    config: {
        model: 'FlickrFindr.model.Photo',
        autoLoad: false,
        pageSize: 25,
        proxy: {
            type: 'jsonp',
            url: 'http://api.flickr.com/services/rest/',
            callbackKey: 'jsoncallback',
            limitParam: 'per_page',
            reader: {
                type: 'json',
                root: 'photos.photo',
                totalProperty: 'photos.total'
            }
        }
    }
});
```

We directed this store to a particular URL (<http://api.flickr.com/services/rest/>) and now, in the listener portion of our controller, we also send our location, radius, and accuracy settings:

```
listeners: {
    locationupdate: function(geo) {
        // Use our coordinates.
        easyparms = {
            "min_upload_date": Ext.Date.format(dt, "Y-m-d H:i:s"),
            "lat": geo.getLatitude(),
            "lon": geo.getLongitude(),
            "accuracy": 16,
            "radius": 10,
            "radius_units": "km",
            "method": "flickr.photos.search",
            "api_key": me.getApplication().api_key,
            "format": "json"
        };
        var store = me.getSearchPhotoList().getStore();
        store.getProxy().setExtraParams(easyparms);
        store.load();
    },
}
```

Each of these parameters is sent along as a set of `POST` variables to the Flickr API URL. Flickr then performs the function `flickr.photos.search` using the variables we've supplied in the previous code. The API then assembles the results in the JSON format and passes them back to us. This is what is referred to as a REST request.

## REST

REST stands for **Representational State Transfer**, which is an overly complicated way to say that we want to use the standard methods already built into HTTP in order to communicate. These methods allow HTTP to transmit data via `POST`, `PUT`, `DELETE` and `GET`.

The Sencha Touch Version 2.1 proxy `Ext.data.proxy.Rest` is a strict REST implementation that uses these four separate methods to handle CRUD functions:

- `POST` handles the creation of new records
- `GET` handles the reading of records
- `PUT` handles the updating of existing records
- `DELETE` handles the deletion of records



The `Ext.data.proxy.Ajax` proxy works similar to `Ext.data.proxy.Rest`, but uses only `POST` and `GET`. If the API you're using requires stricter REST compliance, be sure to use the REST proxy instead.

If you have worked with forms on the Web, you are likely to be familiar with `GET` and `POST`. Both are ways to pass extra variables to a web page for processing. For example, `GET` uses a URL to pass its variables, such as the following:

```
http://www.my-application.com/users.php?userID=5&access=admin
```

This sends `userID=5` and `access=admin` to the web page for processing.

`POST`, `PUT`, and `DELETE` variables are sent as part of the HTTP request and do not appear in the URL. However, they transmit the same kind of data as do key-value pairs.

## Designing your API

It's a good idea before you start coding to think about how you would like your API to work. APIs can get complex rather quickly, and spending some time figuring out what yours will and won't do can help you greatly as you build your application.

Different programmers have different philosophies on how to build APIs, so what we present here is just one possible approach.

Sencha Touch's models and proxies come with several methods, specifically the **CRUD** functions (**Create**, **Read**, **Update**, and **Delete**), which map quite well to API calls. This makes them a good place to start. First, make a list of which models you think you will need. For each model, you will need the Create, Read, Update, and Delete functions.

Then, you should take a careful look at the models to see which ones may need additional API methods. A good example is a `user` model. You will definitely need the basic CRUD methods, but probably also an authentication method to log the user in, and perhaps an additional method for checking permissions.

You may find as you go on that you need to add additional API methods to specific models, but the standard CRUD functions should give you a good start when designing your API.

## Creating the model and store

For this example, we will use a variation of the `Bookmarks` model and the store from our `FlickrFindr` application in the previous chapter.

Since our `Bookmarks` component would now be pulled from a database, we need some extra options in the model. Instead of using the `SearchResults` model as we've done before, we will use a new model, such as the following one:

```
Ext.define('FlickrFindr.model.Bookmark', {
    extend: 'Ext.data.Model',
    fields: [
        {
            name: 'id',
            type: 'int'
        },
        {
            name: 'owner',
            type: 'string'
        },
        {
            name: 'secret',
            type: 'string'
        },
        {
            name: 'server',
            type: 'int'
        }
    ]
});
```

```
        },
        {
            name: 'farm',
            type: 'int'
        },
        {
            name: 'title',
            type: 'string'
        }
    ],
    proxy: {
        type: 'rest',
        url : '/api/bookmarks.php'
    }
});
```

Here, we have added a `rest` proxy and `url` values to our model. This will allow us to save, edit, and delete directly from the model.

For example, to save a new bookmark, we can call the following code in Sencha Touch:

```
var bookmark = Ext.create('FlickrFindr.model.Bookmark', {id: 6162315674, owner: 15638, secret:'d94d1629f4', server:6161, farm:7, title:'Night Sky'});
bookmark.save();
```

This code will perform an HTTP POST request to `/api/bookmarks.php` using all of our bookmark variables as key-value pairs.

Similarly, we can take an existing bookmark, change some of its information, and then call `bookmark.save()`. If we do this on an existing bookmark, the model will send the variables as part of a PUT request to `/api/bookmarks.php`.

As you might expect, calling `bookmark.destroy()` will send our variables to `/api/bookmarks.php` as part of a DELETE request.

We also have to modify our Saved Photos store in a similar fashion:

```
Ext.define('FlickrFindr.store.SavedPhotosStore', {
    extend: 'Ext.data.Store',
    requires: 'FlickrFindr.model.Bookmark',
    config: {
        model: 'FlickrFindr.model.Bookmark',
        storeID: 'BookmarkStore',
        autoload: true,
        proxy: {
```

```
        type: 'rest',
        url: '/api/bookmarks.php',
        reader: {
            type: 'json',
            root: 'children'
        }
    }
}) ;
```

Compared to the store discussed earlier in this chapter, the big difference with this one is the proxy configuration. We are using the same `/api/bookmarks.php` file to process our requests. In this case, the store will use the `GET` request method when contacting the `/api/bookmarks.php` file.

Our reader has a `root` property called `children`. This denotes that the data received should look something like the following:

```
{
    "total": 2,
    "children": [
        {
            "id": "6162315674",
            "owner": "Noel",
            "secret": "d94d1629f4",
            "server": "6161",
            "farm": 7,
            "title": "Night Sky"
        },
        {
            "id": "6162337597",
            "owner": "Noel",
            "secret": "f496834m347",
            "server": "6161",
            "farm": 7,
            "title": "Ring of Fire"
        }
    ]
}
```

Our store will begin looking for records inside the `children` array and will use the default variable `total` to get the total number of records.

## Making a request

Once our model and store understand how to make these requests, our PHP-based API file has to decide what to do with them. This means that we have to set our `bookmarks.php` file to process the requests. At a very high level, this means executing something similar to the following code:

```
<?PHP  
$action = $_SERVER['REQUEST_METHOD'];  
  
if($action == 'GET') {  
    // read - return a list of bookmarks as JSON  
} else if($action == 'POST') {  
    // add a new user  
} else if($action == 'PUT') {  
    // save the edit of an existing user  
} else if($action == 'DELETE') {  
    // delete an existing user  
}  
?  
?>
```

The `<?PHP` and `?>` tags simply denote the beginning and end of PHP code.

The `$action = $_SERVER['REQUEST_METHOD'];` line grabs the `request` method and we then base our code decisions (add, edit, read, or delete) on that result.

 We don't want to get too far into code-specific examples as code will vary greatly depending on the language and database you want to use for your API. You will need to consult a guide for your specific API programming language in order to learn how to interact appropriately with your chosen database.

One thing to note when performing `add`, `edit`, and `delete` functions is that the data that comes to your functions will arrive as an array of records, such as the following:

```
{"records": [{"id":6162315674, "owner": "46992422@N08", "secret": "d94d1629f4", "server": 6161, "farm": 7, "title": "foo"}]}
```

This indicates that for any `add`, `edit`, and `delete` options, you will need to loop through the values for each record and make database changes for each one. While you could conceivably access the records directly using `records[0].id`, looping through the values allows you to take advantage of the data store's ability to sync multiple changes at once.

When your API returns the results of the operation, Sencha Touch expects you to return the full record (or records) that was sent to the API in the first place. For example, if you create a new record, the API should, after a successful save, return that record as part of the results. If you modify several records and save them, the API should return all the modified records if they've been saved correctly. The reason for this is that it's possible that your API will make additional changes to the records, which should be reflected in your JavaScript code. Returning the full records ensures that your JavaScript application stays up-to-date with any changes made by your API.

For example, we can add a number of bookmarks to the store instead of creating them directly using the model as we did earlier in our code for bookmarks. When we call the `sync()` function in the store, it will send the data to our API as an array of bookmarks:

```
{"records": [
  {"id":6162315674,
   "owner": "46992422@N08",
   "secret": " your_secret_here ",
   "server":6161,
   "farm":7,
   "title": "foo"},
  {"id": "6162337597",
   "owner": "Noel",
   "secret": "your_secret_here",
   "server": "6161",
   "farm":7,
   "title": "Ring of Fire"}
]}
```

This way, if we allow for looping in our API, we don't have to worry about whether the request came from the model or the store. From a receiving standpoint, the API only has to worry about whether the request is POST (add), PUT (edit), GET (read), or DELETE (delete).

However, there are also times when we need to communicate directly with the API and, perhaps, get a more complete response. This is where an Ajax request can come in handy.

## Ajax requests in an API

While working with an external database, there are often times when we need to make data changes to other models. We might also need to receive responses that are more complex than those available to the data store in the current version of Sencha Touch. In these cases, we can use an Ajax request object to send data directly to our backend for processing.

For example:

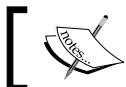
```
Ext.Ajax.request({
    url: '/api/bookmarks.php',
    method: 'GET',
    params: {
        id: '6162337597'
    },
    success: function(result, request) {
        var json = Ext.decode(result.responseText);
        console.log(json.bookmark);
    },
    failure: function(response, opts) {
        console.log('server-side failure with status code ' +
            response.status);
    }
});
```

The previous code makes a direct GET request to /api/bookmarks.php and passes an id 6162337597 value as part of the request. The API can then use this information to grab a specific bookmark and return it to the Ajax request in JSON format.

Success or failure is indicated by returning an appropriate HTTP status code. If you're returning a successful message, simply outputting JSON will return an acceptable status code. To indicate failure, you would return an error code in the 400 or 500 range; in PHP, it may look as follows:

```
<?PHP
header("Status: 400 Bad Request - Invalid Username");
?>
```

You'll need to look up how to send HTTP response headers in the documentation for your preferred API programming language.



For a list of HTTP status codes, visit [http://restpatterns.org/HTTP\\_Status\\_Codes](http://restpatterns.org/HTTP_Status_Codes).



## Going offline

Inevitably, people using your application will find themselves without Internet access. With traditional web applications, this typically means that the application was inaccessible and unusable. But, with some careful planning, you can make your mobile application available offline.

## Syncing local and remote data

The first thing to think about is your data: which data will your users need even when they are offline? Let's use a simple address book example. You would probably have a model for the contacts and a store that queries your remote address book server, along with perhaps a list view to display the contacts:

```
Ext.define('Contact', {
    extend: 'Ext.data.Model',
    config: {
        fields: [
            {name: 'id', type: 'int'},
            {name: 'firstname', type: 'string'},
            {name: 'lastname', type: 'string'},
            {name: 'email', type: 'string'}
        ]
    }
});

Ext.define('ContactStore', {
    extend: 'Ext.data.Store',
    config: {
        model: 'Contact',
        proxy: {
            type: 'jsonp',
            url: 'http://mycontactserver.com/api',
        },
        autoLoad: true
    }
});

Ext.define('ContactView', {
    extend: 'Ext.dataview.List',
    xtype: 'contactview',
    config: {
        store: 'ContactStore',
        itemTpl: '{firstname} {lastname} - {email}'
    }
});
```



This is a very simple example, and we've left out creating an `index.html` file or adding the list to a viewport, even though both these actions would be necessary to make this application actually work.

You'll notice that our application uses a `jsonp` proxy, which is fine if we only want to load its data from a remote server. If we want our application to work offline, we will have to provide some local storage. Additionally, when the user comes back online, we want them to be able to retrieve updated contact information from the remote server.

This means we'll need two stores: our current store, which uses a `jsonp` proxy, and a new store to keep a copy of the data in local storage for when we go offline. The new store looks as follows:

```
Ext.define('OfflineContactStore', {
    extend: 'Ext.data.Store',
    config: {
        model: 'Contact',
        proxy: {
            type: 'localstorage',
            id: 'contacts'
        },
        autoLoad: true
    }
});
```

Our next task is to make sure that the offline store has the most recent data from the online store. We do this by adding a listener to the online store's `load` event. Each time the online store loads new data, we'll update the offline store. In the following way, the offline store works as a cache for the online data:

```
Ext.define('ContactStore', {
    extend: 'Ext.data.Store',
    config: {
        model: 'Contact',
        proxy: {
            type: 'jsonp',
            url: 'http://mycontactserver.com/api',
            reader: {
                type: 'json'
            }
        },
        autoLoad: true,
    }
});
```

```
        listeners: {
            load: function() {
                var offlineContacts = Ext.StoreMgr.get('OfflineContactStore');

                offlineContacts.each(function(record) {
                    offlineContacts.remove(record);
                });
                offlineContacts.sync();

                this.each(function(record) {
                    offlineContacts.add(record.data);
                });

                offlineContacts.sync();
            }
        }
    });
}
```

The `load` event is called whenever the online store successfully loads new data. In our handler, we first retrieve the offline store and clear it (otherwise, we would end up duplicating our data each time we load the online store). Then, we use the online store's `.each()` function to iterate through every record, adding that record's data to the offline store.



## The .each() function

`.each()` is a function provided by the store that allows you to call a function for each record in that store. The function considers the individual record as a single argument. This allows you to perform operations on all the records one at a time rather than querying for them individually.

Now, every time the online store is updated, the offline store is updated too. More importantly, though, when the online store is unable to be updated, the offline store will still have data in it. Since the offline store will always have data to display even when the online store doesn't, we should use the offline store as the store for our list so that we're always displaying something to our users. So, we change ContactView as follows:

```
Ext.define('ContactView', {  
    extend: 'Ext.dataview.List',  
    config: {
```

```
        store: 'OfflineContactStore',
        tpl: '{firstname} {lastname} - {email}'
    }
});
```

Our online store will still autoload when our application starts, even though it's not bound to our list anymore, and if the user is online, all the data in both stores will be updated.

Of course, there are other ways in which you can accomplish the same goal. You could use the `Ext.List` component's `bindStore` function to switch between the two stores and the online store's `jsonp proxy exception` event to discover when you'd gone offline. Or, you could look at the value of the `window.navigator.onLine` variable to determine your online state and set up your stores accordingly. We'll talk about both the `jsonp proxy's exception` event and the `window.navigator.onLine` variable later in this chapter.

## Manifests

Now that we've ensured our data is available offline, we need to make sure that the rest of our application is available as well. This includes all our JavaScript code, HTML, styles, and images. If our user has gone offline, they won't be able to load our application unless they've got a local copy to work from. That's where the Application Cache comes in.

HTML5 provides a mechanism for indicating to a web browser which parts of your application it should store for offline use. This isn't a functionality provided by Sencha Touch, but is something you should be familiar with nonetheless.



If you are using Sencha Cmd to manage your application development process, the `cache.manifest` file will be created for you automatically.



A manifest is a way in which you specify which files to cache. Let's create one for our simple address book application. Open up an empty text file and add the following code:

```
CACHE MANIFEST
# Simple Address Book v1.0

CACHE:
index.html
app/app.js
css/my-app.css
lib/resources/css/sencha-touch.css
lib/sencha-touch.js

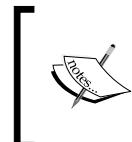
# Everything else requires us to be online.

NETWORK:
*
```

Then, save the file as `cache.manifest`. All lines starting with a hash (#) are comments and are ignored.

In the preceding code snippet, the first section following the term `CACHE:` is a list of files that the mobile device should save for offline use. If you have any images or other files that you use, those should be listed here as well.

The `NETWORK:` section lists all the files that should only be available online. The asterisk (\*) indicates that everything not listed in the `CACHE:` section should be available online only.



Most browsers limit offline storage to 5 MB. This includes both the files listed in your manifest as well as any data in local storage stores. So, if you've got an exceptionally big application, you may want to be selective about what you allow your application to do offline.

In order to let browsers know about your manifest, you have to add a reference to it in the `index.html` file. However, this isn't done in the same way in which we link to CSS or JavaScript files. Instead, we add an attribute to the opening `html` tag as follows:

```
<html manifest="cache.manifest">
```

Now, when you launch your browser, you should see your files listed in the **Application Cache** in the developer console (click on the **Resources** tab and then on **Application Cache**) as shown in the following screenshot:

The screenshot shows the Web Inspector interface with the title "Web Inspector — http://simplecontacts.dev/index2.html". The left sidebar has icons for Elements, Resources, Network, Scripts, Timeline, and Profiles, with "Resources" selected. Below this is a tree view with "simplecontacts.dev" expanded, showing "Frames", "Databases", "Local Storage", "Session Storage", "Cookies", and "Application Cache". Under "Application Cache", there is a single entry: "simplecontacts.dev". The main pane is titled "Search Resources" and contains a table with the following data:

Resource	Type	Size
http://simplecontacts.dev/app/app2.js	Explicit	2.91KB
http://simplecontacts.dev/cache.manifest	Manifest	1.07KB
http://simplecontacts.dev/css/my-app-offline...	Fallback Exp...	848B
http://simplecontacts.dev/index2.html	Master Explicit	1.53KB
http://simplecontacts.dev/lib/resources/css/s...	Explicit	144.82KB
http://simplecontacts.dev/lib/sencha-touch.js	Explicit	366.21KB

## Setting up your web server

Initially, you may find that your manifest isn't working properly. Usually, this means that your web server isn't configured to serve manifest files in the way mobile browsers expect.

Web servers use **MIME Types** to tell browsers how to handle certain files. MIME Types can get pretty complicated, but for manifests, all you have to do is add the MIME Type to your server. You should consult the documentation for your web server for instructions, but we will take the Apache web server as an example.

For Apache, you should add the following MIME Type to your `httpd.conf` file:

```
AddType text/cache-manifest .manifest
```

Then, restart your web server for the changes to take effect.

For IIS, you will want to use the Administration UI to add the MIME Type.

Take a look at the following links for setting up your web server:  
For more on setting up Apache: [http://httpd.apache.org/docs/current/mod/mod\\_mime.html](http://httpd.apache.org/docs/current/mod/mod_mime.html).  
For more on setting up IIS: [http://technet.microsoft.com/en-us/library/cc753281\(WS.10\).aspx](http://technet.microsoft.com/en-us/library/cc753281(WS.10).aspx).

## Updating your cached application

Once your application has been cached locally, the mobile device will no longer query your server to download your application files. This means that when you release updates or new versions of your application, users who've already cached your application won't get your updates.

The only way to force users to download the new version of your code is to update the manifest file itself. That's why we added the following lines at the top of the previous code snippet:

```
CACHE MANIFEST
# Simple Address Book v1.0
```

Just update the version number and save the file as follows:

```
CACHE MANIFEST
# Simple Address Book v1.1
```

This changes the manifest file, which will force anyone with cached copies to re-download all of the files in the CACHE: section of the manifest.



If you want to learn more about the Application Cache and manifest files, check out the *Beginner's Guide to Using the Application Cache* at <http://www.html5rocks.com/en/tutorials/appcache/beginner/>.

## Interface considerations

It's also important to let your users know when they're working in the offline mode. Most devices have an online icon in a status bar, but even so, it's not always apparent to the user when they've gone offline. You may want to let them know when you put your application in the offline mode.

## Alerting your users

In our address book example, we have an online store that updates a second offline store. The offline store holds the data that the user sees displayed in the `Ext.List` class. However, we never explicitly tell the user when they've gone offline. In our first example, we don't keep track of the online or offline status ourselves because the application will work in either mode.

If we want to tell our users when the application has gone offline, the most reliable method is by waiting for the online store's request to time out. In the proxy, let's add a `timeout` component and a function to call when `timeout` occurs:

```
proxy: {
    type: 'jsonp',
    url: 'http://mycontactserver.com/api',

    timeout: 2000,
    listeners: {
        exception:function (this, response, operation, eOpts)  {
            if(operation.error == 'timeout')  {
                Ext.Msg.alert('Offline Mode', 'Network unreachable, we have entered
offline mode.');
            }
        }
    }
}
```

The `exception` function will only be called after the timeout has elapsed. Timeouts in Sencha Touch are listed in milliseconds, so in this case, 2000 means two seconds. If the store doesn't get a response from the server in two seconds, the user is shown an alert informing them that the application has gone offline.

This is a good place to add other offline logic:

- If you've set up polling in your store so that it automatically refreshes every so often, you may wish to turn it off
- If there are special offline UI elements, you can enable them here
- If you have a lot of offline logic, you will probably want to put the code in a separate function so that you don't have to go hunting for it in the proxy configuration

If you are using the MVC structure discussed in the previous chapter, the controller would be a good place for this kind of logic.



If you are compiling your app via Sencha Cmd or some other method rather than running it as a web app, you may have access to the `onlinechange` event that is fired by the `Ext.device.Connection` object. Check the API docs for more on using the `Ext.device.Connection` object.

## Updating your UI

Another way to visually inform your users that they are in the offline mode is to change the color or style of your application. While setting up an entirely different theme for offline mode may be overkill, there is a handy way to specify an offline stylesheet.

Let's create a file called `my-app-offline.css` and save it to our `css` folder. In the file, place the following code:

```
.x-list .x-list-item {  
    color: #f00;  
}
```

This will turn the `contact-list` text red. Now, we need to load it when we're offline.

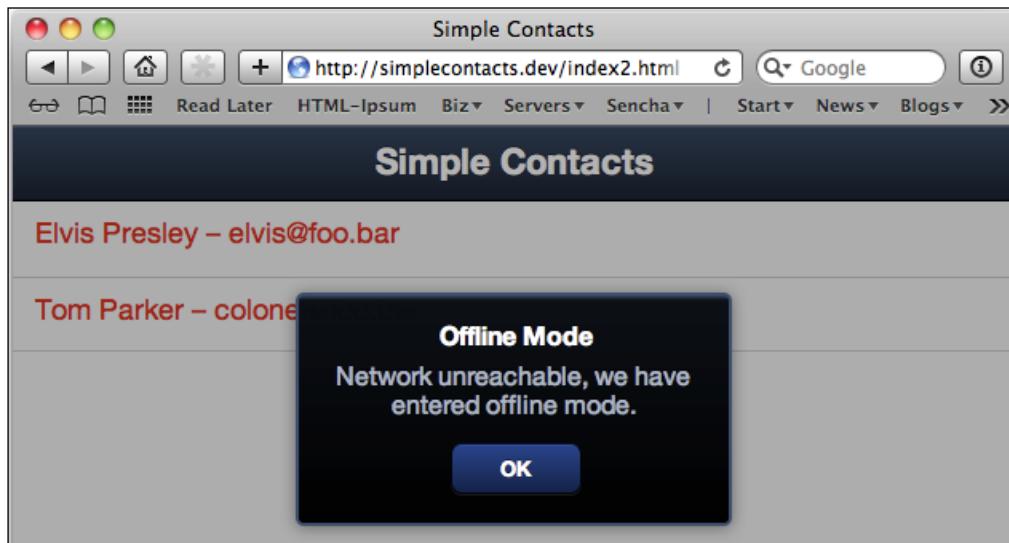
The Application Cache manifest file (`cache.manifest`) can have a section called `FALLBACK`: that is used to substitute an alternate file when a particular file is unreachable. Let's add the following to the bottom of our `cache.manifest` file:

```
FALLBACK:  
css/my-app.css css/my-app-offline.css
```

You should also change the `css/my-app.css` line from the `CACHE:` section to reference `css/my-app-offline.css` instead as follows:

```
CACHE MANIFEST  
# Simple Address Book v1.2  
  
CACHE:  
index.html  
app/app.js  
css/my-app-offline.css  
lib/resources/css/sencha-touch.css  
lib/sencha-touch.js  
  
# Everything else requires us to be online.  
NETWORK:  
*  
  
FALLBACK:  
css/my-app.css css/my-app-offline.css
```

In the `index.html` file, you should leave `css/my-app.css` in the `style` tag as that will be the file that's loaded when we're online. When we're offline, however, the manifest tells our mobile browser to implicitly use `css/my-app-offline.css` instead.



As we can see in the preceding screenshot, now, when your application is offline, it will automatically use `my-app-offline.css` instead of `my-app.css`. You could also use this to provide an offline version of images or even JavaScript files if you want to completely segregate online and offline functionality. It should be noted that this method doesn't work if someone is online and then goes offline while using your application; say, if they went through a tunnel and lost signal. In that case, you would want to use the event listener method to switch your user to the offline mode.

## Alternate methods of detecting the offline mode

As mentioned earlier in the chapter, there are two alternate methods of detecting the offline mode: the `navigator.onLine` and `online/offline` browser events.

The variable `navigator.onLine` will be `true` if the browser is online and `false` if it is not. In the exception function we discussed earlier in the chapter, we can add the following code to check it and change our message accordingly:

```
exception:function () {
    if (navigator.onLine) {
        Ext.Msg.alert('Network Error', 'We have an Internet connection,
but there is a problem communicating with the server.');
```

```

    } else {
        Ext.Msg.alert('Offline Mode', 'No Internet Connection, we have
entered offline mode.');
    }
}

```

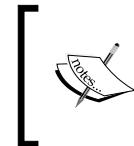
Alternately, we can set up listeners for the browser's `online` and `offline` events as follows:

```

window.addEventListener("offline", function(e) {
    alert("Application is offline.");
});
window.addEventListener("online", function(e) {
    alert("Application is online.");
});

```

You'll notice that we did not use Sencha Touch's event management concept here. This is because Sencha Touch does not provide custom events for `online` and `offline` events, so we have to use the browser's event listener function.



Not all desktop browsers support the `navigator.onLine` or `online/offline` events, so if you are making your application available to desktop users as well, you should use the `timeout` exception and manifest cache techniques instead.



## Getting into the marketplace

Sencha Touch applications offer developers a way to reach a wide audience using existing web technologies. Users can access an application via the Web and even save it to their devices for offline use. While this flexibility is extremely valuable, you may also want to distribute your application through the various application stores available for Apple and Android.

In this section, we will take a look at some of the options available and the potential hurdles for releasing a compiled application.

## Compiling your application

A compiled application is one that runs natively on the device in question. For Apple's iOS products, this means Objective C, and for Google's Android OS, this means Java. Both iOS and Android use their own **Software Development Kits (SDK)** to create these native applications.

An SDK is similar in functionality to Sencha Touch's framework, but it is much more complex and tied to a specific platform (iOS or Android). Since a native application is the only type that can be sold in the various app stores for Android and iOS, we need a way to translate our Sencha Touch JavaScript into one that the SDK can use. Fortunately, Sencha Touch developers have a few options for translating their JavaScript-based applications into either of these languages and creating compiled applications. The two most popular translation programs are Sencha Cmd and PhoneGap.

Both Sencha Cmd and PhoneGap use specialized command-line tools that allow you to take your existing code and place it into the SDK for iOS or Android. Both tools make extensive use of the Xcode and Android SDK libraries to translate your code into compiled applications. We will look at obtaining these SDKs in the *Registering for developer accounts* section.

In addition to translating your Sencha Touch application to a native application, Sencha Cmd and PhoneGap also allow you to access some of the native features of the device. These features include access to the filesystem, camera, and sound and vibration options on the device.

Let's take a look at the Sencha Cmd and PhoneGap translation programs.

## **Sencha Cmd**

If you have been working alongside this book in Sencha Cmd, it is probably your best bet for compiling applications. By compiling the application, you gain access to more of the features in your iOS or Android device. These features include the following:

- **Camera:** This feature allows you to take photos with the camera or access previously taken photos.
- **Connection:** This feature allows you to see whether or not the device is online and what type of connection is being used.
- **Contacts:** This feature allows access to search, sort, and filter contacts on the device.
- **Geolocation:** This feature allows access to the device's geolocation API (this is a more robust implementation of the browser's geolocation functionality).
- **Notification:** This feature shows simple notifications on the device. These notifications appear at the OS level and not just at the application level.
- **Orientation:** This feature gathers feedback on the orientation of the device.
- **Push:** This feature sends push notifications to the device (iOS only).

These features are accessed using an object called `Ext.device`. For example, the `Ext.device.Camera.capture(...)` method allows you to grab an image from the camera or gallery and use it in your application.

A step-by-step guide to native packaging can be found at [http://docs.sencha.com/cmd/3.1.2/#!/guide/native\\_packaging](http://docs.sencha.com/cmd/3.1.2/#!/guide/native_packaging).

## PhoneGap

Much like Sencha Cmd, PhoneGap offers a wide range of native functions through a global object called `navigator`. This object allows you to make JavaScript calls using commands in your JavaScript, such as the following:

```
navigator.camera.getPicture(...)  
navigator.compass.getCurrentHeading(...)
```

The first command opens the camera on the device and lets your application take a picture. The picture is returned as a data string to your application, where you can manipulate it in JavaScript.

The second function returns the orientation of the device in degrees. This can be very useful in games where play can be driven by tilting the device.

PhoneGap also offers access to the following features:

- **Accelerometer:** This feature gets information from the device's motion sensor
- **Camera:** This feature takes a photo using the device's camera
- **Capture:** This feature captures audio and video
- **Compass:** This feature identifies the direction to which the device is pointing
- **Connection:** This feature checks the network status and gets cellular network information
- **Contacts:** This feature works with the onboard contact database
- **Device:** This feature gathers device-specific information
- **Events:** This feature listens to native events on the device
- **File:** This feature reads and writes to the native filesystem
- **Geolocation:** This feature gathers more detailed location information
- **Media:** This feature plays back audio files
- **Notification:** This feature creates device notifications
- **Storage:** This feature stores data directly on the device

PhoneGap also offers options for compiling your applications in the Blackberry, WebOS, and Symbian platforms.



Take a look at the following link for more resources on PhoneGap:

<http://docs.phonegap.com/en/edge/>



## Other options

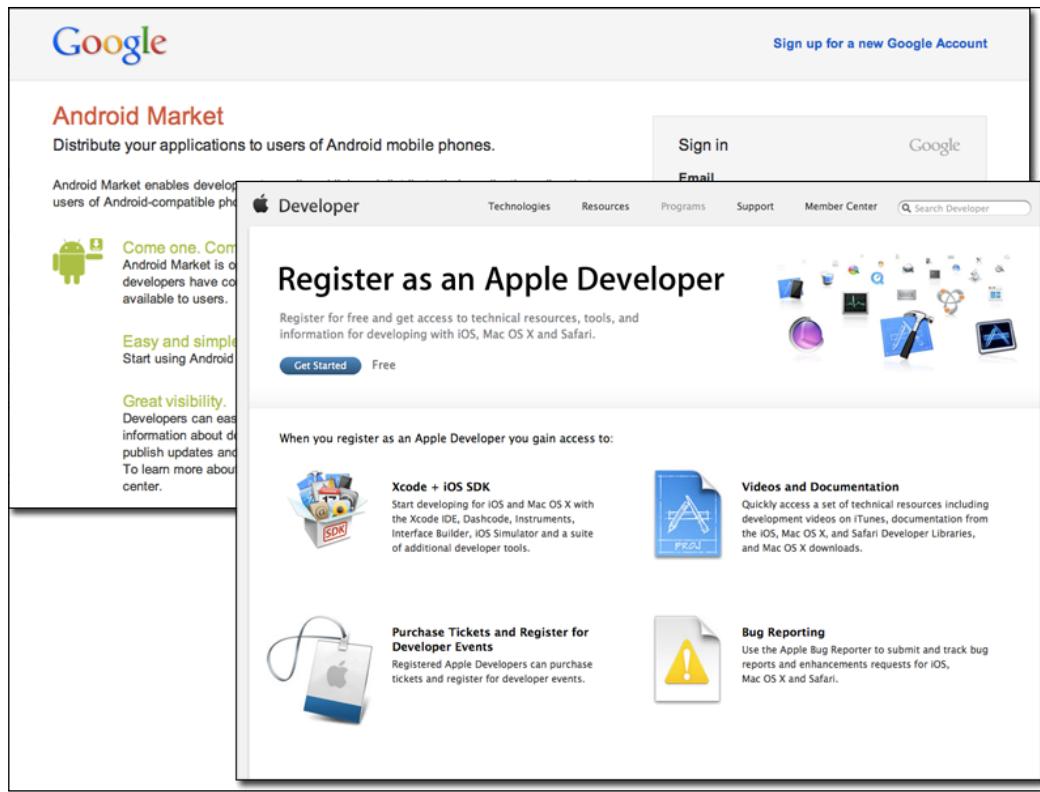
PhoneGap has also launched a cloud-based service for compiling applications called **PhoneGap Build** (<https://build.phonegap.com/>). This unique service does away with the need to download the SDKs for each platform you wish to compile for. Files are simply uploaded to the Build Service and the system generates the application for the platforms that you specify.

**Sencha Architect** is a graphical application builder for Sencha Touch and ExtJS. Architect now has the ability to compile applications for both iOS and Android built right into the application. More information can be found at <http://docs.sencha.com/architect/2/#!guide/deploy>.

As with any of these options, you will need to be a licensed developer for the platform you want to compile for. This can be a bit of a lengthy process, so let's take a look at what's involved.

## Registering for developer accounts

In order to publish your application onto the Apple Store or onto Google Play, you are going to have to sign up for their respective developer accounts. Both stores charge you a fee to become a developer and require quite a bit of information about you. They require this information for several reasons. Firstly, they have to know who you are so that you can get paid for apps that you sell in their stores. Secondly, they need to know how to contact you if there's a problem with your application. And lastly, they need to be able to track you down if you try to do something evil with your application. Not that you would, of course!



You will also need to download and install the appropriate SDK for that store in order to be able to package your application appropriately.

## Becoming an Apple developer

To become an Apple developer, first you must go to <http://developer.apple.com/programs/register/>.

You will need to either supply your existing Apple ID or sign up for a new one, fill out some lengthy profile information, agree to some legal documents, and then perform an e-mail verification. From there, you will have access to the Apple Developer Center. The two points that are of utmost interest to us as mobile developers are the **iOS Dev Center** and the **iOS Provisioning Portal**.

The iOSDev Center is where you can download the iOS SDK (known as **Xcode**), read documentation, see sample code and how-tos, and view some videos about iOS development.

The iOS Provisioning Portal is where you add your application to the Apple Store or publish test versions of your application.



In order to use Xcode or publish your application onto the Apple Store, you must have a computer running OS X. Windows and Linux computers cannot run Xcode or publish onto the Apple Store.

## Becoming an Android developer

Signing up for the Android Market is a very similar process. First, go to <https://market.android.com/publish/signup>.

There, you will be asked to fill out more profile information and pay your developer registration fee. You will also want to download the Android SDK at <http://developer.android.com/sdk/index.html>, although, unlike Apple's SDK, the Android SDK will work on Windows, OS X, and Linux.

The Android Developer Dashboard also has links to guides, reference material, and instructional videos.

## Summary

In this chapter, we've covered a few advanced topics for the aspiring Sencha Touch developer. We first talked about creating your own API to communicate with a database server. We've covered the REST method of communication for sending and receiving data from the server and discussed some options for building your own API.



More resources on creating an API are as follows:

How to create an API: <http://www.webresourcesdepot.com/how-to-create-an-api-10-tutorials/>

Creating an API-centric web application: <http://net.tutsplus.com/tutorials/php/creating-an-api-centric-web-application/>

We've then covered how to take your application offline using manifests and the Application Cache. We've talked about best practices for alerting the user that the application is offline and how you can detect the availability of an Internet connection using Sencha Touch and the device's web browser.

More resources on how to take your application offline are as follows:

Taking Sencha Touch applications offline:

<http://www.sencha.com/learn/taking-sencha-touch-apps-offline/>

The HTML manifest attribute:

[http://www.w3schools.com/tags/att\\_html\\_manifest.asp](http://www.w3schools.com/tags/att_html_manifest.asp)

We've closed the chapter with a look at getting into the application marketplace by compiling your application with Sencha Cmd and PhoneGap. We've also talked about the process for becoming an Apple or Android developer so you can sell your application in the marketplace.

More resources on building Sencha Touch applications:

Enhancing iOS Sencha Touch applications using Sencha Cmd:

<http://docs.sencha.com/cmd/3.1.2/>

Building a Sencha Touch application using PhoneGap:

<http://docs.phonegap.com/en/edge/>



# Index

## Symbols

\$base\_color 81  
.each() function 281  
/path/to/myapp command 38  
/path/to/sdk command 37  
-sdk command 37

## A

action button 65, 150  
ActionSheet component 124, 125  
Add button 263, 264  
addContact function 190  
add method 180  
addNewTab function 150  
Ajax 141  
AjaxProxy 171  
Ajax requests  
  in API 278  
align property 67  
Android developer  
  becoming 294  
  URL 294  
Android Emulator 32  
Android SDK  
  URL 294  
Apache  
  setting up, URL 284  
API (Application Programming Interface)  
  about 238  
  Ajax requests 278  
  creating, URL 294  
  designing 272, 273  
  using 270-272

API-centric web application  
  creating, URL 294  
API, left-hand side bar  
  tabs 136  
app command 38, 39  
app.js file  
  about 39  
  creating 40-43  
Apple developer  
  becoming 293, 294  
  URL 293  
application  
  app.js file, creating 40-45  
  compiling 289, 290  
  debugging 54  
  Main.js file, creating 44, 45  
  missing files 55  
  panel, adding 47, 48  
  putting, into production 57-59  
  Sencha Cmd 36  
  setting up 36  
  tab panel, exploring 45-47  
  testing 54  
  updating, for production 56  
  web inspector console 55  
arguments variable 156  
arrays 173  
ArrayStore data format 173  
asynchronous  
  versus synchronous actions 140, 141  
Asynchronous JavaScript and XML. See  
  Ajax  
auto 166

## B

**back button** 65  
**Balsamiq Mockups**  
  URL 17  
**bar chart** 225-227  
**base component class** 96  
**beforeLoad event** 201  
**beforerequest event** 141  
**bindStore function** 282  
**boolean** 166  
**build.xml** 39  
**button**  
  action button 65  
  back button 65  
  forward button 65  
  normal button 65  
  round button 65  
  small button 65  
  styling 65-68  
  using 162

## C

**callback function** 118  
**card layout**  
  creating 97, 98  
**Carousel component**  
  creating 110-112  
**case sensitivity** 55  
**chart component** 224  
**Chrome Developer Tools**  
  URL 56  
**clearFilter method** 198  
**compass**  
  about 77  
  CSS Resets 77  
  Image Spriting 78  
  Layouts and Grids 77  
  mixins 77  
  Text Replacement 78  
  Typography 78  
  URL 86  
**compass compile**  
  versus compass watch 81  
**compass watch**  
  versus compass compile 81

**component**  
  finding 136  
  page 137, 138  
  versus themes, styling 61-63  
**component object** 9  
**ComponentQuery**  
  multiple items, referencing with 151-157  
**ComponentQuery syntax** 148  
**config section** 137, 150, 166  
**container component** 127  
**container object** 10  
**controllers**  
  about 146-148  
  creating, with Sencha Cmd 241  
**control section** 150  
**Create, Read, Update, and Delete (CRUD)**  
  170, 273  
**CSS**  
  and SCSS 80  
**CSS Mixins** 138  
**CSS Vars** 138  
**custom events** 158

## D

**database**  
  direct creation, limitation 269  
**data formats**  
  about 173  
  arrays 173, 174  
  Extensible Markup Language (XML)  
    174-176  
  JavaScript Object Notation (JSON) 176, 177  
  JSONP 177, 178  
**data stores.** *See also stores*  
**data stores**  
  about 128  
  and panels 202-208  
  binding, by store 196, 197  
  changes, loading 201, 202  
  filters 197, 198  
  paging 199-201  
  sorters 197-199  
  using, for display 195  
**data, XTemplates**  
  looping through 212, 213  
  manipulating 210, 211

**date** 166  
**date function** 211  
**DatePicker component**  
  adding 115, 116  
**defaultValue field** 166, 167  
**DELETE** 272  
**destroy** 163  
**detailsPanel component** 206, 208  
**developer accounts**  
  registering 292, 293  
**developer accounts registration**  
  Android developer, becoming 294  
  Apple developer, becoming 293, 294  
**development environment**  
  Sencha Touch framework, downloading 24  
  Sencha Touch framework, installing 25  
  setting up 21  
  web server, installing on Microsoft  
    Windows 22-24  
    web sharing, setting up on Mac OS X 22  
**direct proxy** 171  
**disabledchange** 163  
**down() method** 191  
**DroidDraw**  
  URL 17

**E**

**ellipsis function** 211  
**emailfield** 114  
**Email Keyboard** 184  
**errors**  
  URL 64  
**events**  
  about 137, 140, 163  
  arguments for 143  
  beforerequest event 141  
  buttons, using 162  
  common events 162, 163  
  custom events 158  
  exploring 140  
  handlers, using 162  
  listener options 158, 159  
  listeners, removing 161  
  requestcomplete event 141  
  requestexception event 141  
  scope 160, 161

**exception function** 286  
**exclusionMessage** 169  
**Ext.Button component**  
  documentation, URL 69  
**Ext.ComponentQuery() string** 251  
**Ext.Container component** 112  
**Ext.data.Model component** 166  
**Ext.data.proxy.Ajax proxy** 272  
**Ext.dataview.List class** 246  
**Ext.define()** 166  
**Extensible Markup Language.** *See XML*  
**Ext.getCmp()** 146  
**Ext.List class** 285  
**Ext.Msg.alert** 118  
**Ext.Msg.confirm** 118  
**Ext.Msg.prompt** 118-121  
**Ext.Sheet class** 122  
**Ext.tab.Panel** 240  
**Ext.util.DelayedTask component** 159  
**Ext.util.Geolocation**  
  using 252-254

**F**

**filterBy method** 198  
**filter method** 198  
**filters** 197, 198  
**fit layout**  
  creating 102  
  hboxTest code 104  
  using 50  
**Flickr API**  
  website, URL 238  
**Flickr Finder Application**  
  basic application, creating 229, 230  
  foundation, building with Sencha Cmd  
  234, 235  
  icons, adding 267  
  improving 268  
  Model View Controller 231-233  
  polishing 267  
  SavedPhotos components, building 259  
  SearchPhotos components, creating 244-246  
**FlickrFindr.store.SearchPhotos store** 245  
**float** 166  
**fn configuration option** 142  
**formatMessage** 169

**FormPanel component**  
    creating 112-114  
    DatePicker component, adding 115, 116  
**form.reset()** 186  
**forms**  
    and stores 181-183  
    editing with 188, 189  
**forums** 21  
**forward button** 65, 67  
**frameworks**  
    about 8-10  
    building, from foundation 10  
    building, with community 12  
    building, with plan 11, 12

## G

**generate command** 37  
**GET** 272  
**getCount() method** 200  
**Google Maps API documentation**  
    URL 128  
**grouped lists**  
    adding 130, 131

## H

**handler configuration** 106  
**handlers**  
    adding 141-146  
    switching 189-192  
    using 162  
**hbox layout**  
    creating 99  
    using 53  
**hboxTest code** 104  
**heightchange** 163  
**hide** 162  
**hide() method** 123  
**htmlDecode function** 211  
**htmlEncode function** 211  
**HTML manifest attribute**  
    URL 295  
**HTTP status codes**  
    URL 278

## I

**icons**  
    mixins 84  
    URL 84  
**IIS**  
    setting up, URL 284  
**images**  
    on multiple devices, with Sencha.io Src  
        89, 90  
    sizes, specifying with Sencha.io Src 91  
    sizing, by formula 91  
    sizing, by percentage 91, 92

## Image Spriting 78

**image URL**  
    in Sencha.io Src, URL 90  
**iMockups**  
    URL 17  
**inclusionMessage** 169  
**incrementValue attribute** 117  
**index.html** 39  
**Inline JavaScript, XTemplates** 217  
**int** 166  
**interactions section** 224  
**iOS SDK** 294  
**iOS Sencha Touch applications**  
    enhancing, Sencha Cmd used 295  
**isEmpty function, XTemplates** 220  
**isValid()** 169  
**itemhighlight interaction** 224  
**iteminfo interaction** 224  
**itemsingletap listener** 188

## J

**Jasmine** 32  
**JavaScript Console** 26  
**JavaScript Object Notation. See JSON**  
**JSLint** 32  
**JSON** 176, 177  
**JSONP** 171, 177

## K

**Kitchen Sink application** 20

## L

**label** 114  
**labelAlign** 114  
**labelWidth** 114  
**launch function**  
    setting up 251, 252  
**layouts**  
    card layout 97  
    complexity, adding 103  
    fit layout, using 50  
    fit layout 102  
    for e-mail application 107, 108  
     hbox layout, using 53  
    hbox layout 98, 99  
    used, for controlling look 48  
    vbox layout, using 51, 52  
    vbox layout 100, 101  
**Learn section** 21  
**leftPad function** 211  
**lengthMessage** 170  
**listeners**  
    adding 141-146  
    configuration 142  
    options 158, 159  
    removing 161  
**lists**  
    creating 128  
    data store 128  
    grouped lists 130, 131  
    List panel 128  
    nested lists 131-135  
    XTemplate 128  
**load event** 281  
**load() function** 201  
**localStorage proxy** 171, 172  
**locationerror** 253, 255  
**locationupdate** 253  
**loop**  
    numbering within 213  
    parent data 214

## M

**Main.js file**  
    creating 44  
**Mainpanel component**  
    about 161

locating 105  
**Map component**  
    creating 126-128  
**mapping configuration option** 174  
**math functionality, XTemplates** 216  
**maxLength** 114  
**member functions, XTemplates** 218-220  
**MemoryProxy** 171  
**MessageBox component**  
    creating 118  
    Ext.Msg.alert 118  
    Ext.Msg.confirm 118  
    Ext.Msg.prompt 118  
**methods** 137  
**MIME Types** 284  
**missing files** 55  
**mixins**  
    in SassSass 72, 73  
**Mobile application frameworks**  
    about 12  
    native application versus web application  
        12, 14  
    web-based mobile frameworks 14, 15  
    web frameworks and touch technology 15  
**mobile resolutions** 16  
**models**  
    about 165  
    basic model 166, 167  
    creating 273-275  
    fields, mapping to 184-186  
    methods 170-172  
    proxies 171  
    readers 171, 172  
    validations 167-170  
**Model View Controller (MVC)**  
    about 231, 232  
    architecture 146  
    pieces, splitting 233, 234  
**mouseup event** 143  
**MyApp command** 38

## N

**name field** 114, 166  
**native application**  
    versus web application 12-14

**Navigation view**  
creating 248, 249  
**nested lists**  
adding 131-135  
**nesting**  
in SassSass 73-76  
**Network tab** 27  
**normal button** 65  
**Notepad++** 31  
**numberfield** 114  
**Number Keyboard** 184

## O

**Object-oriented Programming (OOP)** 8, 9

**Offline mode**  
about 279  
cached application, updating 285  
detecting, ways 288  
local and remote data, syncing 279-282  
manifests 282, 283  
UI, updating 287, 288  
users, altering 285, 286  
web server, setting up 284

**Omni Graffle**  
for Mac, URL 17

## P

**packager.json** 39  
**packages directory** 39  
**pageChange function** 258  
**panel**  
adding 47, 48  
**panzoom interaction** 224  
**parent attribute** 217  
**parse errors** 54, 55  
**personList component** 156  
**PhoneGap**  
about 291  
features 291  
resources, URL 292  
**PhoneGap Build**  
URL 292  
**Photo data model**  
creating 243, 244  
**Pictos**  
about 69

font, URL 69  
icons, URL 84  
**POST** 272  
**presenceMessage** 170  
**properties** 137  
**proxies**  
direct proxy 171  
local 171  
LocalStorageProxy 171  
MemoryProxy 171  
remote 171  
rest proxy 171  
Sql proxy 171  
**PUT** 272

## R

**readers**  
array format 172  
json format 172  
xml format 172  
**recorder**  
for Android, URL 163  
**refs section** 148  
**regexes.** *See Regular expressions*  
**regexpss.** *See Regular expressions*  
**Regular expressions**  
about 168  
URL 168  
**remote proxy**  
AjaxProxy 171  
JsonP 171  
**removePhoto function** 266  
**Representational State Transfer.** *See REST*  
**request**  
making 276, 277  
**requestcomplete event** 141  
**requestexception event** 141  
**required** 114  
**resources directory** 39  
**Resources tab** 29, 30  
**REST** 272  
**rest proxy** 171  
**root folder** 36  
**rotate interaction** 224  
**round button** 65

**Ruby**  
 installing, on Windows 78

**Ruby installer**  
 URL 78

**S**

**SASS + Compass = themes** 78

**Sass functions**  
 UL 71

**SassSass**  
 and compass 77  
 Mixins 72  
 nesting 73075  
 SASS + Compass = themes 78  
 selector inheritance 76  
 variables 70, 71

**SassSassSyntactically Awesome Stylesheets.**  
*See SassSass*

**SavedPhotos components**  
 Add button 263, 264  
 building 259  
 SavedPhotos controller, updating 265, 266  
 SavedPhotos store, creating 260  
 SavedPhoto views, creating 261, 263

**SavedPhotos controller**  
 updating 265, 266

**SavedPhotos store**  
 creating 260

**SavedPhoto views**  
 creating 261-263

**SCSS**  
 and CSS 80

**SearchPhotoDetails view**  
 creating 249

**SearchPhotos components**  
 creating 244  
 Navigation view, creating 248, 249  
 SearchPhotoDetails view, creating 249  
 SearchPhotos Controller, creating 250, 251  
 SearchPhotos list, creating 246-248  
 SearchPhotos store, creating 244-246

**SearchPhotos Controller**  
 creating 250, 251  
 Ext.util.Geolocation, using 252-255  
 launch function, setting up 251, 252  
 list, listening to 255-259

**SearchPhotos list**  
 creating 246

**SearchPhotos store**  
 creating 244-246

**selector inheritance**  
 in SassSass 76, 77

**Sencha Animator** 30

**Sencha Architect** 30, 292

**Sencha Cmd**  
 about 30, 36-38, 290  
 controllers, creating with 241  
 files, including 242  
 Flickr API, using 238  
 installing 235-238  
 Photo data model, creating 243, 244  
 URL, for downloading 36  
 used, for building foundation 234

**sencha command** 37

**Sencha Docs**  
 about 135  
 URL 163

**Sencha.io Src**  
 file types, changing 93  
 formula. sizing by 91  
 images, on multiple devices 89, 90  
 percentage. sizing by 91, 92  
 sizes, specifying with 91  
 URL 93

**Sencha Touch**  
 about 18  
 Application Programming Interface (API)  
 18  
 developing, additional tools used 25  
 events 139  
 examples 19  
 forums 21  
 Kitchen Sink application 20  
 Learn section 21  
 mixins, URL 86  
 themes 70  
 themes, URL 86  
 variables, URL 86

**Sencha Touch applications**  
 Ajax requests, in API 278  
 API, designing 273  
 building 269  
 building, PhoneGap used 295

model, creating 273-275  
own API, using 270, 271  
request, making 276, 277  
REST 272  
store, creating 273-275  
**Sencha Touch applications offline**  
  URL 295  
**Sencha Touch Charts**  
  about 222  
  bar chart 225, 227  
  installing 222  
  pie chart 223-225  
  URL 228  
**Sencha Touch development**  
  Android Emulator 32  
  Chrome Developer Tools 25  
  Jasmine 32  
  JavaScript Console 26  
  JSLint 32  
  Network tab 27  
  Notepad++ 31  
  other Sencha products 30  
  Resources tab 29  
  Sencha Animator 30  
  Sencha Architect 30  
  Sencha Cmd 30  
  third-party developer tools 31  
  web inspector 28  
  WebStorm 31  
  Xcode 5 31  
  YUI test 32  
**Sencha Touch Docs**  
  URL 256  
**Sencha Touch framework**  
  downloading 24  
  installing 24  
**Sencha website**  
  URL 11, 12  
**setActiveItem() command** 97  
**setActiveItem() method** 112  
**Sheet component** 121-123  
**show** 162  
**singletap event** 142  
**sliders**  
  adding 116  
**small button** 65  
**Software Development Kits (SDK)** 289

**sorters** 197-199  
**sort method** 198  
**spinnerfield value** 116  
**spinners**  
  adding 116  
**Sql proxy** 171  
**stores**  
  about 178  
  and forms 181-183  
  creating 273-275  
  data, clearing 186, 187  
  data, deleting from 192, 193  
  fields, mapping to models 184-186  
  forms, editing 188, 189  
  simple 179, 180  
  specialty text fields 183  
**string** 166  
**style block** 62  
**submit() method** 114

## T

**tab bar** 69  
**tabBarPosition value** 110  
**TabPanel component**  
  creating 45, 46, 108, 109  
**tap event** 140  
**tempFunction function** 156  
**Text Replacement** 78  
**theme**  
  base color 81  
  custom theme, creating 79, 80  
  default themes 86, 88  
  mixins 81-83  
  new icons, adding 83, 84  
  Sass resources 85, 86  
  UI configuration 81-83  
  variables 84, 85  
  versus components, styling 61-63  
**this.getMainView() function** 151  
**toggle function** 211  
**toggles**  
  adding 116  
**toolbar**  
  adding 63-65  
**toolbar component** 182  
**totalContacts property** 200

**totalProperty** property 175

**touch**

gestures 16

need for 17

technology 15

**touch** directory 39

**touch-specific events**

URL 163

**trebuchet** 158

**trim** function 211

**Typography** 78

## U

**ui configuration** 65

**urlfield** 114

**URL Keyboard** 184

**User Interface (UI)** 18

## V

**validate** method 168

**values** attribute 217

**variables**

about 84, 85

in SassSass 70, 71

**vbox layout**

creating 100, 101

using 51, 52

## W

**web-based mobile frameworks** 14, 15

**web frameworks** 15

**web inspector console** 28, 55, 56

**web server**

installing, on Microsoft Windows 22-24

**web sharing**

setting up, on Mac OS X 22

**WebStorm** 31

**widthchange** 163

**window.navigator.onLine** variable 282

**Windows**

Ruby, installing 78

## X

**xaccount** attribute 217

**Xcode** 294

**Xcode 5** 31

**xindex** attribute 217

**XML** 174, 175

**XTemplate.overwrite**

used, for changing panels content 221

**XTemplates**

about 128, 208-210

arithmetic functionality 216

conditional display 214-216

data, looping through 212, 213

data, manipulating 210, 211

Inline JavaScript 217

isEmpty function 220, 221

member functions 217-220

panel content changing,

XTemplate.overwrite used 221

**xtype** attribute 68

## Y

**YUI test** 32





## Thank you for buying Sencha Touch 2 Mobile JavaScript Framework

### About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.packtpub.com](http://www.packtpub.com).

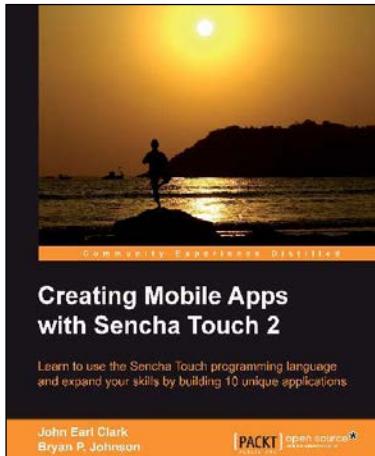
### About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

### Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

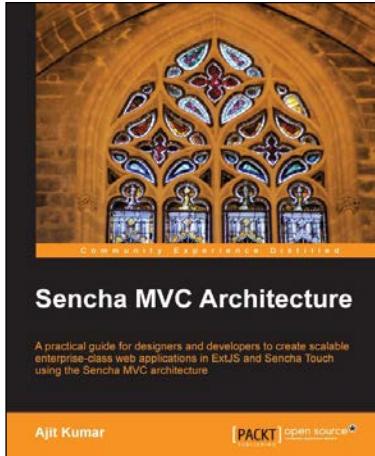


## Creating Mobile Apps with Sencha Touch 2

ISBN: 978-1-84951-890-1      Paperback: 348 pages

Learn to use the Sencha Touch programming language and expand your skills by building 10 unique applications

1. Learn the Sencha Touch programming language by building real, working applications
2. Each chapter focuses on different features and programming approaches; you can decide which is right for you
3. Full of well-explained example code and rich with screenshots



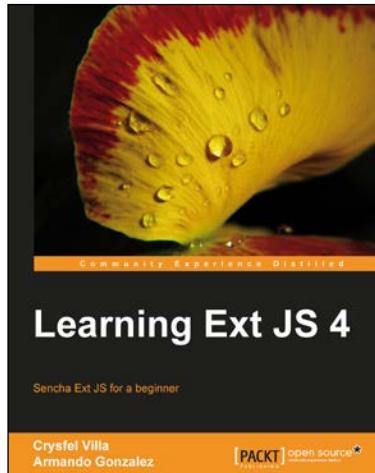
## Sencha MVC Architecture

ISBN: 978-1-84951-888-8      Paperback: 126 pages

A practical guide for designers and developers to create scalable enterprise-class web applications in ExtJS and Sencha Touch using the Sencha MVC architecture

1. Map general MVC architecture concept to the classes in ExtJS 4.x and Sencha Touch
2. Create a practical application in ExtJS as well as Sencha Touch using various Sencha MVC Architecture concepts and classes
3. Dive deep into the building blocks of the Sencha MVC Architecture including the class system, loader, controller, and application

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles



## Learning Ext JS 4

ISBN: 978-1-84951-684-6      Paperback: 434 pages

Sencha Ext JS for a beginner

1. Learn the basics and create your first classes
2. Handle data and understand the way it works, create powerful widgets and new components
3. Dig into the new architecture defined by Sencha and work on real world projects



## Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-84951-686-0      Paperback: 488 pages

Over 110 easy-to-follow recipes backed up with real-life examples, walking you through basic Ext JS features to advanced application design using Sencha's Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles