Quick answers to common problems

Highcharts Cookbook

80 hands-on recipes to create, integrate, and extend dynamic and interactive charts in your web projects

Nicholas Terwoord

[PACKT] PUBLISHING

Highcharts Cookbook

80 hands-on recipes to create, integrate, and extend dynamic and interactive charts in your web projects

Nicholas Terwoord

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

Highcharts Cookbook

First published: March 2014

Production Reference: 1120314

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

ISBN 978-1-78355-968-8

www.packtpub.com

Cover Image by Catherine Garland (catherinegarland@comcast.net)

Credits

Author
Nicholas Terwoord

Reviewers
Gert Vaartjes
Juanjo Fernandez
Jugal Thakkar
Steve P. Sharpe

Acquisition Editors
Nikhil Karkal
Kartikey Pandey

Content Development Editor
Balaji Naidu

Technical Editors
Arwa Manasawala
Veena Pagare
Manal Pednekar
Anand Singh

Copy Editors
Roshni Banerjee
Sayanee Mukherjee
Deepa Nambiar

Project Coordinator
Wendell Palmer

Proofreaders
Mario Cecere
Stephen Copestake

Indexer
Priya Subramani

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Author

Nicholas Terwoord is a software developer, professional geek, and graduate from the University of Waterloo with a Bachelor of Computer Science (Honors). When not developing software, which is not often, he can be found helping his wife, Amanda, with her business, or more likely working his way through a growing list of distractions on Steam. He can be reached at `http://nt3r.com`.

He is happily employed at Willet Inc., a company in Kitchener, Ontario that develops Second Funnel, a marketing solution for brands, and online retailers. More information can be found at `http://secondfunnel.com`.

I would like to take this opportunity to thank my lovely wife, Amanda, for being so supportive as I wrote this book as well as my good friends who encouraged me through the long (and sometimes arduous) journey towards completing my first published work.

About the Reviewers

Gert Vaartjes started as a specialist in Geographical Information Systems. While customizing these programs, he was intrigued by what's actually under the hood, and thus started his passion for programming. This programming journey led him through all kinds of programming languages. As a technical consultant, he worked for several governmental and non-governmental companies. He has been developing software for more than 10 years. Now he's working as a senior developer at Highsoft, working on the Highcharts products and focusing on backend integrations of Highcharts.

When not programming, you can find him working on his small-scale farm in Norway, where he grows potatoes, chases sheep, chops wood, and does other basic stuff.

Juanjo Fernandez is a software developer with 10 years of professional experience. He was self taught in Flash when he began using it and tried to combine the best possible design/programming and user experience.

For several years he has been in the backend, struggling with databases and servers, using PHP, MySQL, and Apache, but he's also a certified Java programmer. Now he has returned to the area of development that he is passionate about, the frontend. He's a strong advocate of web standards, and he's very excited about the future and the great possibilities offered by HTML5, CSS3, and JavaScript.

Currently he works in Incubio, a startup incubator located in Barcelona, and helps to develop frontend of several startups such as ZeedProtection, Quizlyse, NotedLinks, Signaturit, and Trakty. Also, he is working on his first personal project, Wallastic.

If you want to know him, you can follow him on Twitter and his Twitter handle is `@juanjo_fr`.

Jugal Thakkar is a very passionate and enthusiastic software developer since his youth. He is also curious about new technologies and relishes sharing knowledge. Enterprise web applications are his forte, with usability and user friendliness as his prime focus. He is an active supporter of the Stack Overflow community and is one of the top respondents to Highcharts' queries. He appreciates open source technologies and is a keen follower of Android. He loves to solve Sudoku, Rubik's Cube, and play ping pong in his free time. All the views expressed are his own and do not reflect those of his employer or anyone else.

Steve P. Sharpe has been a software engineer for more than a decade, specializing in designing and building scalable web apps. Primarily a Ruby programmer, he is also a Zend Certified Engineer and has solid knowledge of frontend technologies and utilizes best practices and latest industry trends.

He is Chief Technology Officer at EthOS Labs, an Ethnographic Research Solution company, and he has been instrumental in the company's growth and innovation. He has previously worked with various well-known brands and organizations including Coca-Cola, ITV, NHS, Sonneti ®, Autocar, Oasis, Drambuie®, Motorola, and KPMG.

Follow him on Twitter; his Twitte handle is `@stevepsharpe`.

www.PacktPub.com

Support files, eBooks, discount offers and more

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

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

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

http://PacktLib.PacktPub.com

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

Preface

Welcome to *Highcharts Cookbook*. Highcharts is a charting library that makes it easy to create interactive, configurable charts using just pure JavaScript and HTML5. It supports a variety of different chart types, has an extensive set of documentation, and even has helpful support available. This book explores how it is possible to integrate Highcharts into a variety of applications, focusing on some of the more common applications.

If it seems daunting to get started with something new, such as Highcharts, there's no need to worry. Everyone has been where you are now: beginning a journey to learn something new. In this case, if you're unfamiliar with Highcharts (or even JavaScript) that's fine; step by step, this book will walk you through simple recipes in the first few chapters to get you up-to-speed and make you more comfortable.

If you've used Highcharts before, then you can take a look through the different recipes at your leisure, and you can work to improve your understanding of the library and how it can fit into applications. You can build on the examples to create something great. Each recipe and chapter will help you to focus on a particular area to grow and improve.

If you're a JavaScript expert, then this book will provide a lot of shortcuts. There's no need to reinvent the wheel; just find out what you want to do to accomplish your goals, get a feel for what needs to be done, and use this book to speed yourself along. Whether you are an expert or a novice, I hope that you find the recipes of this book useful, and that they aid you in accomplishing your goals.

What this book covers

Chapter 1, *Getting Started with Highcharts*, covers the basics of setting up a simple page with Highcharts and quickly explores common scenarios a developer may encounter.

Chapter 2, *Processing Data*, dives into the different input sources for a chart and how those sources connect to our chart.

Chapter 3, *Handling User Interaction*, shows how we can customize charts to provide richer interactions and visualizations.

Chapter 4, *Sharing Charts on the Web*, demonstrates how we can send charts to others, online or offline.

Chapter 5, *Integrating with ExtJS*, shows how we can start building rich desktop-like applications using Highcharts.

Chapter 6, *Integrating with jQuery*, covers how we can leverage jQuery and its various plugins to create and display charts.

Chapter 7, *Integrating with the Yii Framework*, demonstrates how we can use Highcharts in a PHP application.

Chapter 8, *Integrating with Other Frameworks*, looks at some of the more popular Web frameworks and tools and how we can get them up and running with Highcharts.

Chapter 9, *Extending Highcharts*, takes us one step further into working with the internals of Highcharts and how we can create our own chart extensions.

Chapter 10, *Math and Statistics*, dives into how we can use Highcharts to graph and display data of a more mathematical and scientific nature.

Chapter 11, *System Integration*, covers a few interesting connections with system resources and how we can use Highcharts to visualize that data.

Chapter 12, *Other Inspirational Uses*, takes a look at how we can use what we've learned in the previous chapters as well as leveraging HTML5 APIs and other odds and ends to create really interesting applications without a lot of code.

What you need for this book

While this book focuses primarily on Highcharts, there are a number of tools that we will leverage to make the recipes possible. Usually, all the required tools are mentioned in the *Getting ready* section of a recipe. The following are a few of the required tools:

- **Node.js** (`http://nodejs.org/`): This is a platform for creating JavaScript applications on the server side. This book was written assuming version 0.10.24 or higher is being used.
- **Bower** (`http://bower.io/`): This is a package manager for our JavaScript dependencies. This book was written assuming version 1.2.8 or higher is being used.
- **Git** (`http://git-scm.com`): This is a distributed version control system needed for certain recipes and to install certain packages with Bower. This book was written assuming version 1.8 or higher is being used.

- **Python** (`http://www.python.org/`): This is a programming language used in some recipes for server-side examples. This book was written assuming version 2.7 of Python is being used, and it is unlikely that these examples will work in Python 3 or higher.
- **pip** (`http://pip-installer.org/`): This is a package manager for Python. This book was written assuming version 1.4 or higher is being used.
- **PHP** (`http://php.net`): This is a general-purpose scripting language used in some recipes for server-side examples. This book was written assuming version 5.3 or higher is being used.
- **Web browser**: Any recent version of Firefox, Chrome, Internet Explorer, or Safari should work fine.

Who this book is for

I've done my best to make this book as easy to read as possible for anyone with a technical background. However, this book will be easier to understand and more useful for JavaScript developers or other developers working on web applications.

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, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Charts are created by making instances of a `Highcharts.Chart` object, either directly via its constructor or indirectly using plugins developed for different JavaScript frameworks."

A block of code is set as follows:

```
{
  "name": "my-project",
  "dependencies": {
    "highcharts": "~3.0",
    "jquery": "^1.9"
  }
}
```

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

```
var options = {
    // ...
    tooltip: {
        formatter: function() {
            return 'We have ' + this.y + ' ' + this.point.options.
            category + 's'
        }
    }
}
```

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

```
pip install bottle==0.11.6
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the **By hour** button, as shown in the following screenshot."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/9688OT_Images.pdf`.

Errata

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

Piracy

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

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

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

Questions

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

1

Getting Started with Highcharts

In this chapter, we will cover the following recipes:

- Finding documentation on Highcharts
- Creating your first chart
- Including multiple series in one chart
- Displaying multiple charts in one graph
- Using the same data in multiple charts
- Creating spiderweb graphs for comparison
- Creating custom tooltips
- Adding extra content to tooltips
- Making charts internationalizable/localizable
- Creating a new theme
- Creating reusable graphs

Introduction

This chapter explains the basics of creating and rendering a chart using Highcharts and how to work with different Highcharts options to configure charts. All charts are created by providing a chart with the `options` object; `options` allows the user to define the behavior and look and feel of the chart.

Charts are created by making instances of a `Highcharts.Chart` object, either directly via its constructor or indirectly using plugins developed for different JavaScript frameworks.

Finding documentation on Highcharts

Highcharts has a very well-documented **Application Programming Interface** (**API**), and while many of the examples we go through will include details of the various options and settings used, this book is by no means a complete reference.

How to do it...

To get started, follow the ensuing instructions:

1. Visit `http://docs.highcharts.com` to find an introduction to core concepts in Highcharts, learn about chart features, and get an introduction to working with charts.
2. Highcharts also has a searchable API document found at `http://api.highcharts.com`, which has details of every method, property, and configuration option available to set up a chart. Many of the configuration options in the API include links to examples where it is possible to see the option in action or modify an existing chart.
3. Lastly, there are the demos that can be found either at `http://www.highcharts.com/demo`, or within the `examples` folder from the Highcharts ZIP file. Demos show a variety of examples used and configurations to give some idea of what Highcharts is capable of creating.

Creating your first chart

To create and render a chart, we'll need to create a `Highcharts.Chart` instance and provide it with some options.

Getting ready

There are a few things that we need to do before we get started:

1. Install `bower` (`http://bower.io`), a package manager for JavaScript.
2. Create a `bower.json` file that lists information about our project, most importantly, its dependencies, as shown in the following code:

```
{
  "name": "my-project",
  "dependencies": {
    "highcharts": "~3.0",
    "jquery": "^1.9"
  }
}
```

> **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.

3. From the same folder, run `bower install` to install our dependencies.
4. Create a simple HTML page where we will create our chart, as shown in the following code:

```
<html>
    <head>
        <style type='text/css'>
            #container {
                width: 300px;
                height: 300px;
                border: 1px solid #000;
                padding: 20px;
                margin: 10px;
            }
        </style>
    </head>
    <body>
        <div id='container'></div>


        <script src='./bower_components/jquery/jquery.js'></
         script>
        <script src='./bower_components/highcharts/highcharts-all.
         js'></script>

        <script type='text/javascript'>
            $(document).ready(function() {
                // our code will go here
            });
        </script>
    </body>
</html>
```

> In our examples, we will be using jQuery, but there are plugins and adapters for many different toolkits and frameworks.

How to do it...

To get started, follow the ensuing instructions:

1. First, we create an `options` object that will define what our chart looks like, as shown in the following code:

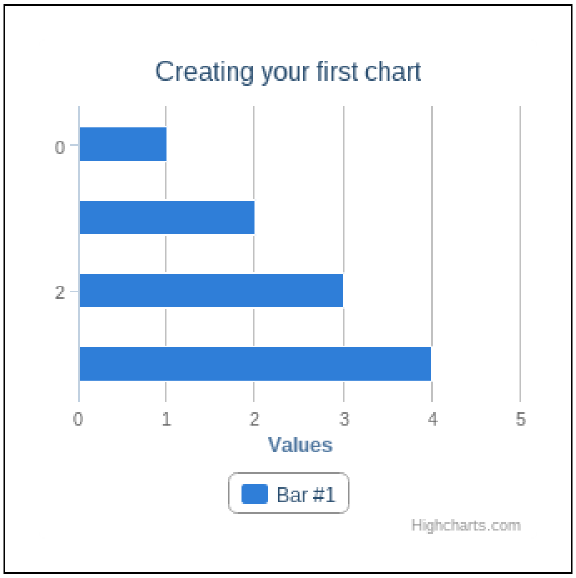
   ```
   var options = {
       chart: {
           type: 'bar'
       },
       title: {
           text: 'Creating your first chart'
       },
       series: [{
           name: 'Bar #1'
           data: [1, 2, 3, 4]
       }]
   }
   ```

   > It is possible to create a chart with an empty set of options (that is, `options = {}`) but this generates a very bland chart.

2. Next, we render our new chart by calling the `.highcharts` jQuery function on some element on the page. In this case, we select an element on the page with an `id` value equal to `container`, as shown in the following code:

   ```
   var options = {
       chart: {
           type: 'bar'
       },
       title: {
           text: 'Creating your first chart'
       },
       series: [{
           name: 'Bar #1',
           data: [1,2,3,4]
       }]
   };

   $('#container').highcharts(options);
   ```

The following is the rendered chart:



How it works...

The `.highcharts` function is actually a part of a jQuery plugin used to create the `Highcharts.Chart` objects. It uses jQuery's element selector (for example, `$('#container')`) to find the element we want to render the chart to and renders the chart inside that element. Even if we supply a more general selector (for example, `$('div')`), it will only render the first element.

There's more...

As previously mentioned, it is not necessary to use jQuery to render a chart. We can create a chart instance manually using the `chart.renderTo` option and the `Highcharts.Chart` constructor. Using this method, we can either pass in the ID of an element or a reference to an element, as shown in the following code:

```
// Using an element id
var options = {
    chart: {
        renderTo: 'container'
    },
     // ...
}
```

```
var chart = new Highcharts.Chart(options);

// Using an element reference
var otherOptions = {
    chart: {
        renderTo: document.getElementById('container');
    },
     // ...
}

var otherChart = new Highcharts.Chart(options);
```

Including multiple series in one chart

While it is useful to display one data series, we may want to add more data to a chart. For example, we may want to compare two different sets of data over the same period of time.

In Highcharts, we can display additional data in a separate `series` array. The `series` arrays are just lists of data with a name. In Highcharts, this list is represented by a JavaScript array.

How to do it...

To get started, follow the ensuing instructions:

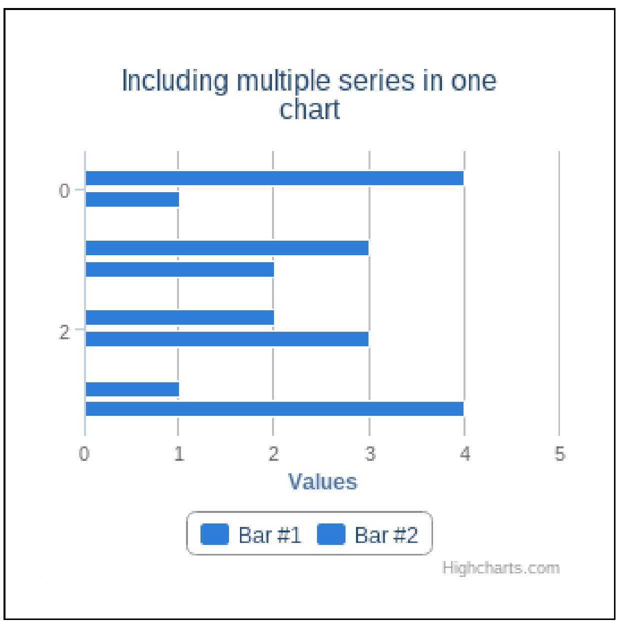
1. Define options for our chart as in the previous recipe, as follows:

```
var options = {
    chart: {
        type: 'bar'
    },
    title: {
        text: 'Including multiple series in one chart'
    },
    series: [{
        name: 'Bar #1',
        data: [1, 2, 3, 4]
    }]
};
```

2. Add a second series object as shown in the following code:

```
var options = {
    chart: {
        type: 'bar'
    },
    title: {
        text: 'Including multiple series in one chart'
    },
    series: [{
        name: 'Bar #1',
        data: [1, 2, 3, 4]
    },  // Add a new series
    {
        name: 'Bar #2',
        data: [4, 3, 2, 1]
    }]
};
```

3. Finally, render the chart using the `highcharts` function `$('#container').highcharts(options)"`. This output is shown in the following screenshot:



There's more...

If we want to add another series to the chart after it has been rendered, you can use the `addSeries` method and pass it in the series object. We can get a reference to the chart in one of the following two ways:

- Create the chart, then call `.highcharts()` with the appropriate jQuery selector, as shown in the following code:

```
$('#container').highcharts(options);
var chart = $('#container').highcharts();
```

- When creating the chart, chain together a call to `.highcharts()` as follows:

```
var chart = $('#container').highcharts(options).highcharts();
```

Using the chaining method, we can add a series as follows:

```
var chart = $('#container').highcharts(options).highcharts();
chart.addSeries({
    name: 'Series 2',
    data: [4,3,2,1]
});
```

The `addSeries` method also has a few other arguments that can be passed. The `addSeries` method also has optional second and third arguments that determine whether the chart should be redrawn (defaults to `true`) and how the new series should be animated (defaults to `true`, but we could supply an animation object that is best described in the documentation).

Displaying multiple charts in one graph

We are not limited to displaying a single series in a chart, and likewise, we are not limited to displaying a single type of chart within the same chart. In some circumstances, we may want to display the same data using different types of charts.

Earlier, we saw that a `series` object can have data associated with it, such as `name`. Similarly, a `series` object can also have a type, which changes how the data is displayed in the rendered chart.

How to do it...

To get started, follow the ensuing instructions:

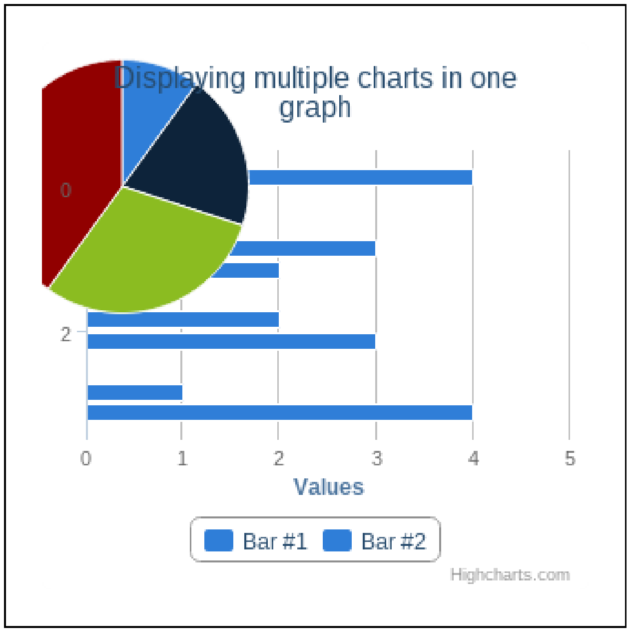
1. Define our chart options as we did in the previous recipe, as shown in the following code:

```
var options = {
    chart: {
        type: 'bar'
    },
    title: {
        text: 'Displaying multiple charts in one graph'
    },
    series: [{
        name: 'Bar #1',
        data: [1, 2, 3, 4]
    }, {
        name: 'Bar #2',
        data: [4, 3, 2, 1]
    }]
```

2. Add a new series to our chart with the type `pie` using the following code:

```
var options = {
    chart: {
        type: 'bar'
    },
    title: {
        text: 'Displaying multiple charts in one graph'
    },
    series: [{
        name: 'Bar #1',
        data: [1, 2, 3, 4]
    }, {
        name: 'Bar #2',
        data: [4, 3, 2, 1]
    }, { // add new series
        type: 'pie',
        data: [1,2,3,4],
        center: [0,0]
    }]
}
```

The following is the rendered chart:



How it works...

By default, Highcharts will use the `chart.type` string to determine how the different series should be displayed. However, if a series has its own type provided, it will use that type when it is rendered in the chart.

There's more...

Just changing the `type` string of the series will probably result in something ugly or otherwise undesired, especially in the case of a pie chart where it will render on top of the existing chart. Fortunately, it is possible to adjust the positioning and style of a pie series by providing a `center` option.

If we wanted to enable the labels on the pie chart, we could set `dataLabels.enabled` to `true`, as shown in the following code:

```
var options = {
    // ...
    series: [{
        type: 'pie',
```

```
            name: 'Bar #1',
            data: [1,2,3,4],
            dataLabels: {
                enabled: true
            }
        }]
    };
```

In fact, a `series` object can have any options that you would normally set inside `plotOptions.<chartType>`. For more details, visit `http://api.highcharts.com/highcharts#plotOptions`.

Using the same data in multiple charts

Oftentimes, we have data that we want to display in different ways on the same page, but we may not want to show that data in the same chart. For example, we may want to have an aggregate view of a set of data in one chart and another where we can see the same data over a time period. In this case, we want to share the same data in different charts.

How to do it...

To get started, follow the ensuing instructions:

1. Create a set of data or have a set of data available, as shown in the following code:

   ```
   var data = [1,2,3,4];
   ```

2. Define options for our two charts as shown in the following code:

   ```
   var chartOptions = {
       // other fields omitted for brevity
       series: [{
           name: 'X',
           data: data
       }];
   };

   var chart2Options = {
       // other fields omitted for brevity
       series: [{
           name: 'Y',
           data: data
       }];
   }
   ```

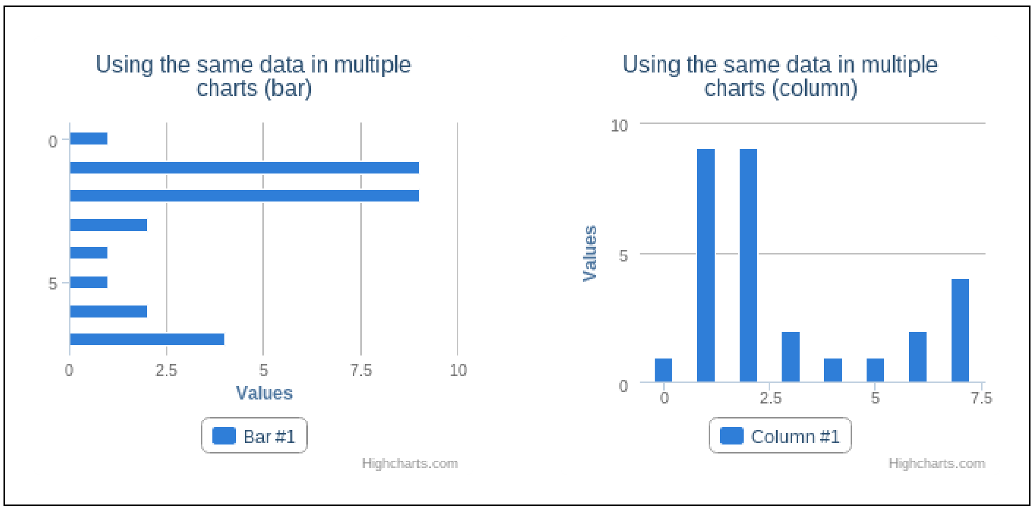
3. Render our two charts as shown in the following code:

```
$('#container').highcharts(chartOptions);
$('#container2').highcharts(chart2Options);
```

4. If the data changes, call `<series>.setData` on each chart to reflect the changes in the charts as follows:

```
// e.g. data = [1, 9, 9, 2, 1, 1, 2, 4];
$('#container').highcharts().series[0].setData(data);
$('#container2').highcharts().series[0].setData(data);
```

The following chart reflects these changes:



How it works...

Highcharts uses the same reference to the data for both charts. Unfortunately, it does not maintain a reference to the original data, so we need to call `setData` to update the chart with the new data.

Creating spiderweb graphs for comparison

We like to compare different things, but sometimes, the things that we want to compare differ in more than just one or two axes. Rather than displaying multiple graphs, we can amalgamate these different axes into one graph and use the spiderweb graph.

How to do it...

To get started, follow the ensuing instructions:

1. Define options for our basic chart, setting the `polar` property of the chart to `true` using the following code:

```
var options = {
    chart: {
        polar: true,
        type: 'line'
    },
    title: {
        text: 'Creating spiderweb graphs for comparison'
    }
}
```

> To create a spiderweb graph, we'll need to make a polar chart. The previous options will change our display from an ordinary two-axes chart into an arbitrary-axes chart that is more like a circle.

2. Label the axes of our graph by setting `xAxis.categories`, as shown in the following code:

```
options= {
    // ...
    xAxis: {
        categories: ["Strength", "Speed", "Defense"],
        tickmarkPlacement: 'on'
    }
};
```

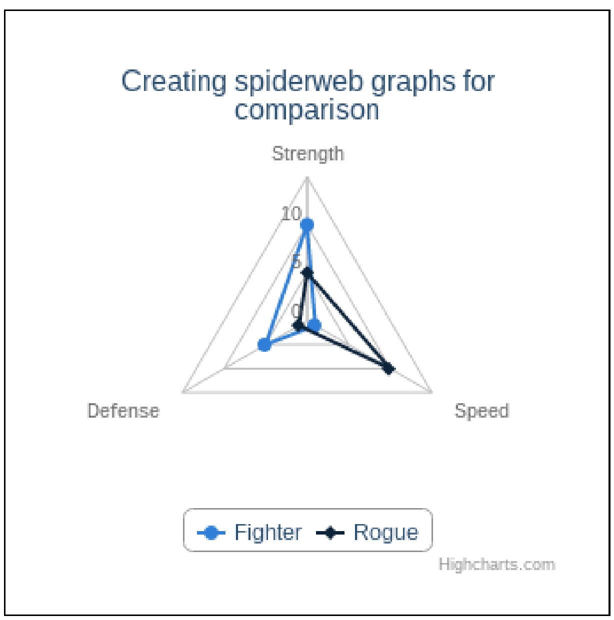
3. Set `yAxis.gridLineInterpolation` to `polygon` to make the chart less rounded, as shown in the following code:

```
var options= {
    // ...
    yAxis: {
        gridLineInterpolation: 'polygon'
    }
};
```

4. Define the data for our spiderweb graph as follows:

```
var options = {
    // ...
    series: [{
        name: 'Fighter',
        data: [10, 1, 5],
        pointPlacement: 'on'
    }, {
        name: 'Rogue',
        data: [5, 10, 1],
        pointPlacement: 'on'
    }]
};
```

The following is the rendered graph:



Creating custom tooltips

So far, we haven't done a lot with the behavior of charts. One common behavior in charts is the `tooltip` object, which can display useful information about a data point in the graph when a user hovers the mouse over that point. Tooltips are added by default to a graph, but it is useful to be able to extend this basic functionality.

How to do it...

To get started, perform the following instructions:

1. Create a function for our tooltip as follows:

```
var formatter = function () {
    var tooltipMessage = '';

    tooltipMessage += 'X value: ' + this.x + '<br>';
    for (var i=0; i < this.points.length; i++) {
        tooltipMessage += 'Y[' + i + '] value: ' + this.points[i].
        y+ '<br>'
    }

    return tooltipMessage;
}
```

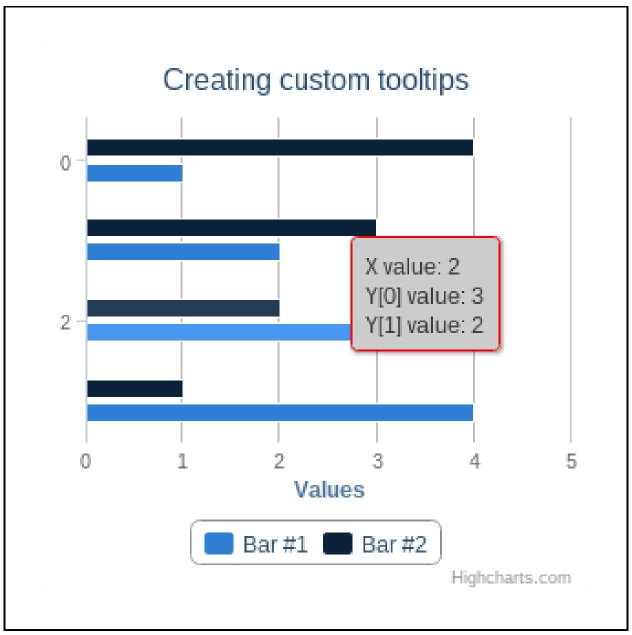
2. Define options for our chart as follows:

```
var options = {
    chart: {
        type: 'bar'
    },
    title: {
        text: 'Creating custom tooltips'
    },
    series: [{
        name: 'Bar #1',
        data: [1,2,3,4]
    }, {
        name: 'Bar #2',
        data: [4,3,2,1]
    }]
};
```

3. Assign this function to our options as `tooltip.formatter`, as shown in the following code:

```
var options = {
    // ...
    tooltip: {
        formatter: formatter,
        borderColor: '#f00',
        backgroundColor: '#ccc',
        shared: true
    }
};
```

The following is the output chart:



> The formatter function will render any string within the tooltip window. The this keyword refers to the data point that we are hovering over, so we can access the x and y values of the current point via this.x or this.y.
>
> We can also change the appearance of the tooltip via options such as changing the border with tooltip.borderColor or the background with tooltip.backgroundColor.
>
> It is even possible to disable the tooltip entirely by setting tooltip.enabled to false.
>
> More details on tooltip options can be found at http://api.highcharts.com/highcharts#tooltip or in the individual plot options for a chart at http://api.highcharts.com/highcharts#plotOptions.

There's more...

By default, tooltips are not shared—every series displays only the data for its own tooltip. If you want to have all the data to be available from a single tooltip, you can set tooltip.shared to true. In this case, if we are using tooltip.formatter, we need to change how we refer to our y values, that is, instead of this.y, we need to use this.points[i].y (where i is the series index). In fact, any value that we would normally access via this.<value> needs to be accessed via this.points[i].<value> when we have the shared Boolean value set to true. The one exception to this rule is this.x, which is always common.

If we wanted, we could also make our tooltips look better by adding HTML to our formatter function. The formatter function supports the <b>, <strong>, <i>, <em>, and
 tags, which gives us a bit more flexibility in how we design our tooltips. This is shown in the following code:

```
var options = {
    // ...
    tooltip: {
        formatter: function() {
            return 'The <b>value</b> at this point is <em>' + this.y +
            '</em>';
        }
    }
}
```

Adding extra content to tooltips

We've already seen that tooltips can add useful behavior to our charts; however, we are not merely limited to changing colors or text in the tooltip. It is possible to access more data than just what Highcharts provides.

How to do it...

To get started, follow the ensuing instructions:

1. Create or make available a set of additional data that we would like to use in our tooltips as follows:

   ```
   var altName = ["apple", "banana", "pear"];
   ```

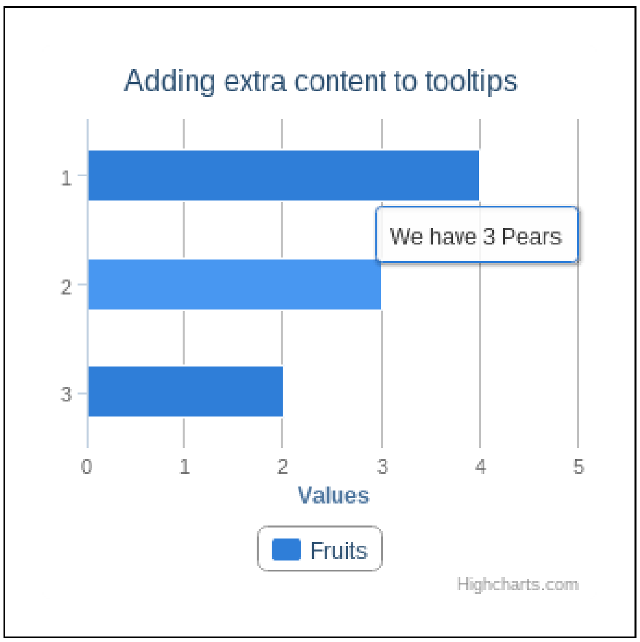
2. Define options for our chart, including our series and its additional data, as shown in the following code:

   ```
   var options = {
       chart: {
           type: 'bar'
       },
       title: {
           text: 'Adding extra content to tooltips'
       },
       series: [{
           data: [{
               'x': 1,
               'y': 4,
               'category': 'Apple'
           }, {
               'x': 2,
               'y': 3,
               'category': 'Pear'
           }, {
               'x': 3,
               'y': 2,
               'category': 'Banana'
           }]
       }]
   };
   ```

3. Access our desired fields in the `formatter` function via `this.point.options` as shown in the following code:

```
var options = {
    // ...
    tooltip: {
        formatter: function() {
            return 'We have ' + this.y + ' ' + this.point.options.
            category + 's'
        }
    }
}
```

The following is the output chart:



How it works...

Highcharts supports multiple data formats. Earlier, we were using its most basic format—an array of numeric data. However, as in this example, we have seen that the `data` array can be more complex. A data series can support two other formats—an array of objects with named values (as we used previously) and an array of the `[x,y]` coordinates, as you can see in the following code:

```
var options = {
    // ...
    series: {
        data: [
            [1,1],
            [2,2],
            [3,3]
        ]
    }
}
```

When we specify data as an array of objects, we can access the information about the individual points via `this.point.options`. Since the `formatter` function is just a JavaScript function, we can do whatever we might normally do inside a JavaScript function, such as displaying our additional information.

Making charts internationalizable/ localizable

Making charts for our own purposes is fine; however, in a business environment, we may be working with people who need to view charts in a different language or localize the chart. Fortunately, Highcharts makes it possible to change language and display settings by changing Highcharts global settings.

Getting ready...

We will need to have access to some translated words. We could use Google Translate or a similar service; but for our purposes, we will use French, just to ensure that changes have been made.

How to do it...

To get started, follow the ensuing instructions:

1. Define our chart options as shown in the following code:

```
var options = {
    chart: {
        type: 'spline'
    },
    xAxis: {
        type: 'datetime',
        dateTimeLabelFormats: {
            month: '%B'
        }
    },
    title: {
        text: 'Making charts internationalizable / localizable'
    },
    series: [{
        name: 'Temperature?',
        data: [
            [Date.UTC(2013,  0, 1), 1],
            [Date.UTC(2013,  1, 1), 10],
            [Date.UTC(2013,  2, 1), 100],
            [Date.UTC(2013,  3, 1), 1000],
            [Date.UTC(2013,  4, 1), 10000],
            [Date.UTC(2013,  5, 1), 1000],
            [Date.UTC(2013,  6, 1), 100],
            [Date.UTC(2013,  7, 1), 1000],
            [Date.UTC(2013,  8, 1), 10000],
            [Date.UTC(2013,  9, 1), 1000],
            [Date.UTC(2013,  10, 1), 100],
            [Date.UTC(2013,  11, 1), 1],
        ]
    }]
};
```

2. Create a `lang` object with the appropriate keys for the text that we want to change, as shown in the following code snippet. For example, if we want to change the months, we would create `lang.months`. We can also change the thousands separator (`lang.thousandsSep`) or symbols (`lang.numericSymbols`) we use for different counters (for example, normally 1,000 = 1K, but we can instead use 1 mille).

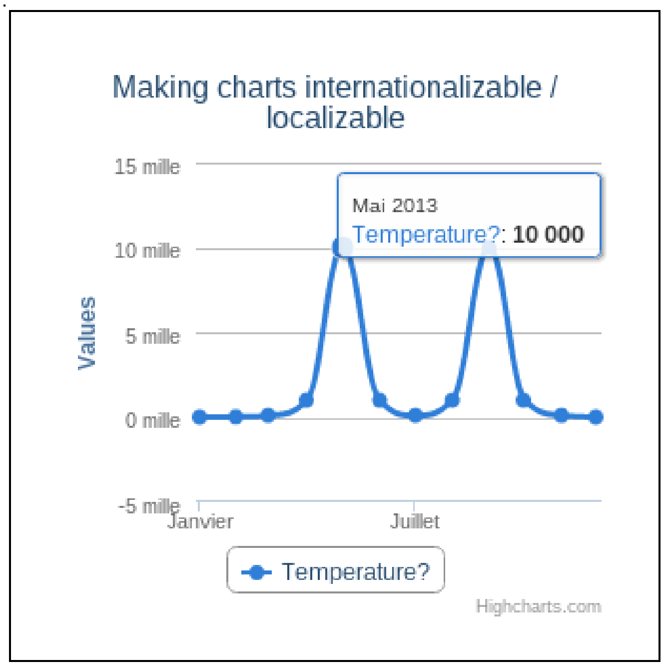
```
var lang = {
    months: ['Janvier', 'Février', 'Mars', 'Avril', 'Mai',
    'Juin', 'Juillet', 'Août', 'Septembre', 'Octobre', 'Novembre',
    'Décembre'],
    thousandsSep: ' ',
    numericSymbols: [' mille']
};
```

3. Call `Highcharts.setOptions` to change these language settings globally using the following code:

```
Highcharts.setOptions({lang: lang});
```

> We must call `Highcharts.setOptions` before any charts are rendered, otherwise the changes will not take effect.

The following is the new graph:



How it works...

The `Highcharts.setOptions` function allows us to change the options of all the charts prior to the charts being rendered. Unlike other options, `lang` can only be set in this way. Internally, Highcharts has a number of different settings for language strings, and what we have done in our example is overwritten the default English strings with French ones.

Number formats and other numeric details can also be changed for languages that differ from English. Again, using French as the example, we will change the decimal separator to a comma and the thousands separator to a space instead of a comma, as shown in the following code:

```
Highcharts.setOptions({
    lang: {
        decimalPoint: ',',
        thousandsSep: ' '
    }
});
```

We can also change the numeric symbols if we like. Numeric symbols are used when we have large numbers to display, such as one million (1,000,000). By default, metric prefixes are used for every power of one thousand (1,000), such as **k** for one thousand (1k) and **M** for one million (1M). However, we can change these values as we like. We can also disable shortening altogether if we set `lang.numericSymbols` to `null`. This is shown in the following code:

```
Highcharts.setOptions({
    lang: {
        numericSymbols: [' thousand', ' millions']
    }
});
```

Some languages are displayed **right-to-left** (**RTL**) rather than **left-to-right** (**LTR**), as English is. If the language we are working with is an RTL language, we may want to move the positions of the x and y axes such that they are also right oriented. We can do this by setting `yAxis.opposite` to `true` (to move the y axis to the right-hand side) and `xAxis.reversed` to `true` (to start the x axis on the right-hand side), as shown in the following code snippet:

```
var options = {
    // ...
    yAxis: {opposite: true},
    xAxis: {opposite: true}
}
```

By calling `Highcharts.setOptions` for different `lang` options (refer to `http://api.highcharts.com/highcharts#lang` for more details), we can change just about any of the default strings in Highcharts including the strings used to determine the loading text (for example, `lang.loading`), the strings for downloading the chart (for example, `lang.downloadJPEG.`), and other date fields (for example, `lang.weekdays`).

There's more...

If, for whatever reason, we need to render the chart before we set language options, there is a way to do so. All that we have to do is redraw the existing chart after it has rendered, as shown in the following code:

```
Highcharts.setOptions(options);
$('#container').highcharts().redraw();
```

Creating a new theme

When working on charts, we may find that there is a set of colors that works well or that there are settings that we may want to use in other charts. This is where themes are helpful. Themes are just a collection of common options that we can apply to all charts.

How to do it...

To get started, follow the ensuing instructions:

1. Create an empty options object using the following code:

   ```
   var myTheme = {};
   ```

2. Assign the properties we want in the theme, such as colors or a background color as follows:

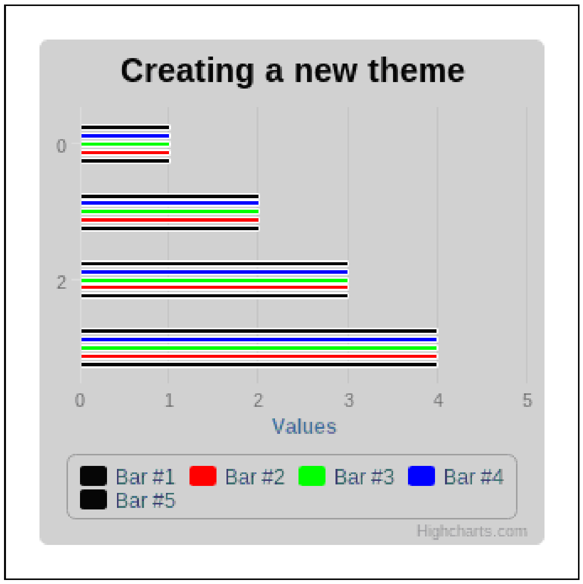
   ```
   myTheme.colors = ["#000000", "#ff0000", "#00ff00", "#0000ff"]
   myTheme.chart = {
       backgroundColor: '#cccccc'
   };
   myTheme.title = {
       style: {
           fontSize: '20px',
           fontFamily: '"Georgia", "Verdana", sans-serif',
           fontWeight: 'bold',
           color: '#000000'
       }
   }
   ```

> When all colors have been used, Highcharts will pull new colors from the beginning of the array.

3. Call `Highcharts.setOptions` to apply our theme to all charts using the following code:

```
Highcharts.setOptions(myTheme);
```

The following is the output chart:



How it works...

The `Highcharts.setOptions` function, as previously discussed, sets options globally; a theme is just a set of options that we want applied to all charts.

If we want to store the theme in a separate file, we only need to make a few small changes. First, we will create our theme in a new file. In this file, we will create our theme in the `Highcharts` namespace, include our theme file after `highcharts.js` on our main page, and call `Highcharts.setOptions`, as shown in the following code:

```
// myTheme.js
Highcharts.myTheme = {
    // Our theme goes here
};
```

```
// main page
<script type='text/javascript' src='highcharts.js'></script>
<script type='text/javascript' src='myTheme.js'></script>

<script type='text/javascript'>
    // ... chart creation ...

    Highcharts.setOptions(Highcharts.myTheme);

    // ... chart rendering …
</script>
```

There's more...

Highcharts provides a few basic themes out of the box. This includes `grid`, `skies`, `gray`, `dark blue`, and `dark green`, in addition to the default colors. They can be found in the `themes` folder and are included just as you would include any other theme. More details on theming can be found at `http://highcharts.com/docs/chart-design-and-style/themes`.

Creating reusable graphs

So far we have experimented with a lot of different graph options and configurations where themes defined a common set of styles; we may find a time where we have a very common type of graph that we want to create and we do not want to define the same options over and over again. We can avoid this tedium by creating reusable charts.

How to do it...

To get started, follow the ensuing instructions:

1. Determine what type of a chart you want to make reusable. Suppose that we want to take our existing spiderweb chart.
2. Create a new function `spiderWebChart`. This chart will take an `options` argument to let us configure the chart and return a `Highcharts.Chart` instance, as shown in the following code:

    ```
    var SpiderWebChart = function (options) {
        return Highcharts.Chart(options);
    };
    ```

3. Define default values for the chart that will give it the correct appearance, as we did in the recipe *Creating spiderweb graphs for comparison*, using the following code:

```
var SpiderWebChart = function (options) {
    // create options if they don't exist
    var modifiedOptions = options || {};

    // create a chart option if it does not exist
    modifiedOptions.chart = modifiedOptions.chart || {};
    modifiedOptions.chart.polar = true;

    // create an xAxis option if it does not exist
    modifiedOptions.xAxis = modifiedOptions.xAxis || {};
    modifiedOptions.xAxis.tickmarkPlacement = 'on';
    modifiedOptions.xAxis.lineWidth = 0;

    // create a yAxis option if it does not exist
    modifiedOptions.yAxis = modifiedOptions.xAxis || {};
    modifiedOptions.yAxis.gridLineInterpolation = 'polygon';
    modifiedOptions.yAxis.lineWidth = 0;

    return new Highcharts.Chart(modifiedOptions);
};
```

4. Create a spiderweb graph using the options from the previously mentioned recipe, using the following code:

```
var chart = SpiderWebChart({
    chart: {
        renderTo: 'container'
    },
    title: {
        text: 'Creating spiderweb graphs for comparison'
    },
    series: [{
        name: 'Fighter',
        data: [10, 1, 5],
        pointPlacement: 'on'
    }, {
        name: 'Rogue',
        data: [5, 10, 1],
        pointPlacement: 'on'
    }]
});
```

How it works...

We have created a wrapper function for common options. Instead of using jQuery, we use the `renderTo` option to find an element with `container` as its ID and render the chart within that element. As we only overwrite certain properties in our `SpiderWebChart` function, we can pass in as many other options as we like and only the ones relevant to the `SpiderWebChart` function will be affected.

2
Processing Data

In this chapter, we will cover the following recipes:

- Working with different data formats
- Using AJAX for polling charts
- Using WebSockets for real-time updates
- Drilling down and filtering data
- Using CSV, XML, or JSON with Highcharts
- Handling cross-domain data
- Handling dates

Introduction

Highcharts makes it easy to chart existing data, but one problem that often comes up is how to get data from a backend service into a chart. This chapter covers the specifics of how Highcharts can read different data formats, how to fetch fresh data, how to drill down and filter data, and generally, how data can be processed in Highcharts.

Working with different data formats

When creating charts, we may have little control over which format the data comes back in. Due to this, it's important that we be aware of how to work with the different formats that Highcharts supports.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Define the options as shown in the following code:

```
var options = {
 series: []
};
```

2. Assign the series data as an array of arrays as shown in the following code:

```
var options = {
  series: [{
    name: 'Array of arrays',
    data: [
      [0, 0],
      [1, 1],
      [2, 4],
      [3, 9]
    ]
  }]
};
```

3. Create a second series as an array of objects as shown in the following code:

```
var options = {
  series: [{
    name: 'Array of arrays',
    data: [
      [0, 0],
      [1, 1],
      [2, 4],
      [3, 9]
    ]
  }, {
    name: 'Array of objects',
    data: [
      {x: 0, y: 0},
      {x: 1, y: 1},
      {x: 2, y: 4},
      {x: 3, y: 9}
    ]
  }]
};
```

How it works...

Our examples so far had our data formatted as a simple array of numbers. This format can be useful if we have data that will be used in a bar, column, or pie chart, especially where the categories or x axis are provided for us.

If our data is more complex, for instance, a series of (x, y) pairs, we will want to use one of Highcharts' other supported formats, namely, the array of arrays, as we used earlier. In this case, each individual element of data contains two points stored in an `[x , y]` array. Highcharts also supports a variation of this format if we want to express a data range. If we had data on temperature and wanted to show its highs and lows, for example, we could enter our data as `[index, low, high]`:

```
var options = {
  series: [{
    name: 'temperature (Celsius)'
    data: [
      [0, 15, 30],
      [1, 12, 27],
      [2, 14, 23],
      [3, 10, 20]
    ]
  }]
};
```

Highcharts does support one last format that is more flexible than the previous two. Highcharts has the idea of a single data point that has its own `Point` format. The `Point` format is where each element in the data array is a JavaScript object with, at least, a `y` value:

```
var options = {
  series: [{
    data: [{
      'y': 4, // required
      'x': 0, // optional fields
      'id': '0',
      'myField': 'value'
    }]
  }]
};
```

Using these three different formats gives us a lot of flexibility in how we can use Highcharts without any adjustments to our data. More details of these formats can be found in the Highcharts documentation at `http://api.highcharts.com/highcharts`.

Using AJAX for polling charts

We have made static charts so far. Often, we'll want to update charts periodically to represent changes in the data over time. The best way to do that is using AJAX.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

If you already have a server set up or will be setting up your own, many of the following steps can be omitted:

1. Download Python 2.7 from `http://www.python.org/download` and install it.
2. Download pip from `http://www.pip-installer.org/en/latest/installing.html` and install it.
3. Run the following command to install Bottle:

   ```
   pip install bottle==0.11.6
   ```

4. Create a `bottle_server.py` file and include the following code in it:

   ```
   #!/usr/bin/env python2.7
   from bottle import run, route, static_file, template, request, response
   import json
   import random

   r = lambda: random.randint(0,255)

   def jsonp(request, data):
     if (request.query.callback):
       return "{callback}({result})".format(
         callback=request.query.callback,
         result=data
       )
     return data

   @route('/jsonp/series')
   def jsonp_series():
     if (request.query.callback):
       response.content_type = 'application/javascript'
     response.status = 200
     return jsonp(request, series())
   ```

```
@route('/jsonp/point')
def jsonp_point():
  if (request.query.callback):
    response.content_type = 'application/javascript'
  response.status = 200
  return jsonp(request, point())

@route('/csv/series')
def csv_series():
  response.content_type = 'text/css'
  response.status = 200
  results = [];
  for x in xrange(0,11):
    results.append(str(r()))

  return ",".join(results)

@route('/xml/series')
def xml_series():
  response.content_type = 'application/xml'
  response.status = 200
  xml = "<xml>\n";
  for x in xrange(0,11):
    xml += "\t<row>\n\t\t<y>{0}</y>\n\t</row>\n".format(r())
  xml += "</xml>"
  return xml;

@route('/ajax/series')
def series():
  response.content_type = 'application/javascript'
  response.status = 200
  series = []
  for x in xrange(0,11):
    series.append({
      'y': r(),
      'color': '#%02X%02X%02X' % (r(), r(), r())
    })

  return json.dumps(series)

@route('/ajax/point')
def point():
  response.content_type = 'application/javascript'
  response.status = 200
```

```
  point = {
    'y': r(),
    'color': '#%02X%02X%02X' % (r(), r(), r())
  }
  return json.dumps(point)


# Static files
# e.g. HTML page and Javascript
@route('/')
def index():
  return static_file('index.html', root='.')

@route('/bower_components/<filename:path>')
def index(filename):
  return static_file(filename, root='bower_components')

run(host='localhost', port=8000)
```

5. To start the web server, run the following command from the command line:

```
python bottle_server.py
```

How to do it...

To get started, perform the following steps:

1. Define the options as shown in the following code:

```
var options = {
  chart: {
    type: 'bar',
  },
  title: {
    text: 'Using AJAX for polling charts'
  },
  series: [{
    name: 'AJAX data (series)',
    data: []
  }]
};
```

2. Create a new event handler for the chart load event:

```
var options = {
  chart: {
    type: 'bar',
    events: {
      load: function() {
        // maintain a reference to the chart
        var self = this;

        // code goes here
      }
    }
  }
  /* … */
};
```

3. Within our event handler, create an interval via `setInterval` with a duration of how frequently we would like the chart to refresh as shown:

```
events: {
  load: function() {
    // maintain a reference to the chart
    var self = this;

    setInterval(function() {
      // our code goes here
    }, 3000); // 3000 milliseconds = 3 seconds
  }
}
```

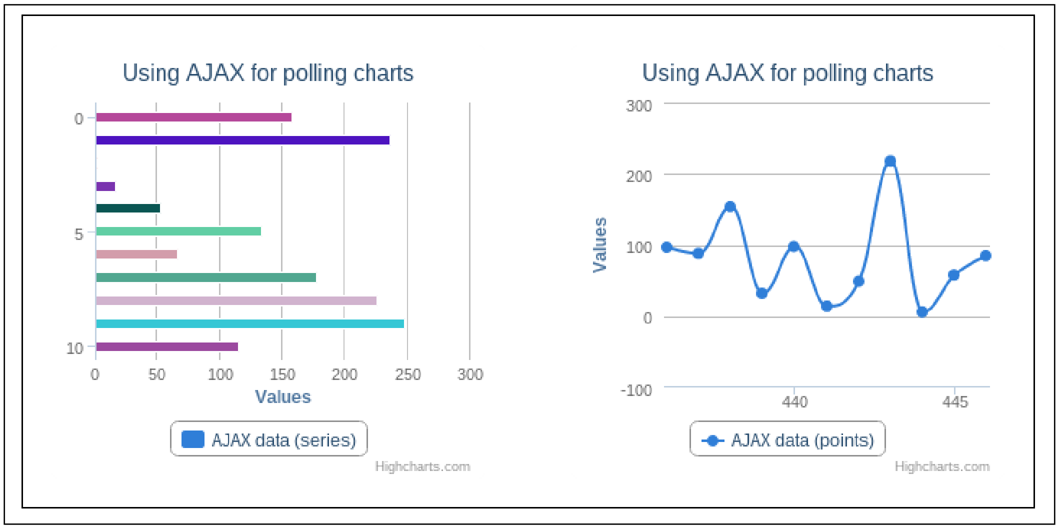
4. Inside our interval function, call `$.getJSON` to fetch our data as shown in the following code:

```
load: function() {
  // maintain a reference to the chart
  var self = this;

  setInterval(function() {
    $.getJSON('http://localhost:8000/ajax/series', function(data)
{
      // our code goes here
    });
  }, 3000);
}
```

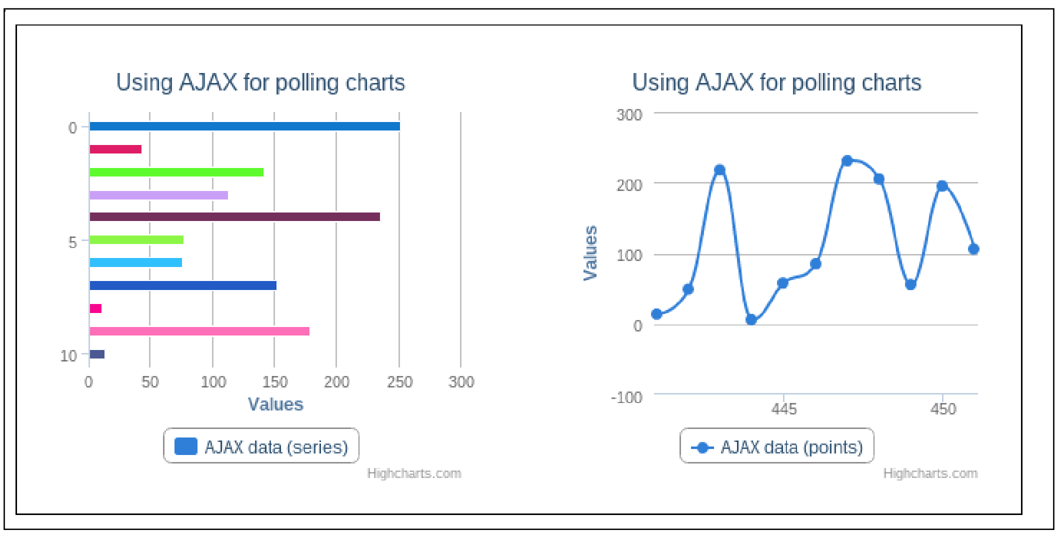
5. Inside our `$.getJSON` call, replace the existing data in the chart with `<series>.setData()` as shown:

```
$.getJSON('./example.json', function(data) {
  self.series[0].setData(data);
});
```

The resultant chart is displayed as follows:



6. Observe the graph change every three seconds, as shown in the following screenshot:



How it works...

Highcharts supports a variety of events that are triggered at different points. In our case, we have created an event handler to execute when the chart is complete. Normally, an event handler would include some information about the event that we are handling, but we do not need this feature in this example.

Also, using `setInterval`, we can periodically execute some functions over and over again. In this case, we can periodically fetch the new data via `$.getJSON` and then redraw the chart when the new data is available.

There's more...

Presently, our function will completely replace the existing data series. While this works, it is not necessarily efficient, or ideal, as it destroys the existing data and replaces it with the new data. Instead, if we have a data source that can provide us with updates point by point, we can use `<series>.addPoint()`. In this case, we just need to change our `$.getJSON` function:

```
load: function() {
  var self = this;
  $.getJSON('./example.json', function(data) {
    var series = self.series[0];
    series.addPoint(data);
  });
}
```

It is also worth noting that in this recipe we used an event (`load`). Events are triggered at different points in the chart, for example, when individual data points are selected or removed (for example, `series.data.events`), or after axes change (for example, `xAxis.events`).

Using WebSockets for real-time updates

Using AJAX for chart updates, as done in the last recipe, is helpful but may lead to a lot of unnecessary calls to whichever backend service is providing the data. Also, regardless of whether there is new data, we will have to ask for it every three seconds (or whichever interval we've configured). One alternative to this is to use WebSockets, which allows us to receive updates as soon as the server has updates.

For this recipe, we will be using Tornado, a python library that is available at `http://www.tornadoweb.org`, to provide the server-side component for our chart, but the client-side code will be similar for any server-side component that provides the WebSockets connectivity.

> While WebSockets is gaining support in many, if not the most modern, browsers—at the time of writing—they are not supported in all browsers or may experience some unusual behavior in certain network configurations. For this reason, please be aware of the limitations of your application environment when using them.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

If you already have a WebSocket-capable server set up or will be setting up your own, many of these steps can be omitted:

1. Download Python 2.7 from `http://www.python.org/download` and install it.
2. Download pip from `http://www.pip-installer.org/en/latest/installing.html` and install it.
3. Run the following command to install Tornado:

   ```
   pip install tornado==3.1
   ```

4. Create a `websocket_server.py` file and include the following code in it:

   ```
   #!/usr/bin/env python2.7
   import json
   import random
   from tornado import websocket, web, ioloop
   import datetime
   from time import time

   # Random number generator
   r = lambda: random.randint(0,255)

   # Boilerplate WebSocket code
   class WebSocketHandler(websocket.WebSocketHandler):
     def open(self):
   ```

```
    print 'Connection established.'
    # Set up a call to send_data in 5 seconds
    ioloop.IOLoop.instance().add_timeout(datetime.
    timedelta(seconds=1), self.send_data)

  def on_message(self, message):
    print 'Message received {0}.'.format(message)

  def on_close(self):
    print 'Connection closed.'

  # Our function to get new (random) data for charts
  def send_data(self):
    point_data = {
      'x': int(time()),
      'y': r(),
      'color': '#%02X%02X%02X' % (r(), r(), r())
    }

    self.write_message(json.dumps(point_data))
    timeout = r() / 10

    # Call this again within the next 0-25 seconds
    ioloop.IOLoop.instance().add_timeout(datetime.
    timedelta(seconds=timeout), self.send_data)

application = web.Application([
  (r'/websocket', WebSocketHandler)
])

if __name__ == "__main__":
  application.listen(8001)
  ioloop.IOLoop.instance().start()
```

5. To start the WebSocket server, run the following command from the command line:

```
python websocket_server.py
```

How to do it...

To get started, perform the following steps:

1. Define the chart options as shown in the following code:

```
var options = {
  chart: {
    type: 'spline',
  },
  title: {
    text: 'Using WebSockets for realtime updates'
  },
  xAxis: {
    type: 'datetime'
  },
  series: [{
    name: 'Websockets data (points)',
    data: []
  }]
};
```

2. Create a new event handler for the chart load event as shown in the following code:

```
var options = {
  chart: {
    type: 'spline',
    events: {
      load: function() {
        // maintain a reference to the chart
        var self = this;

        // code goes here
      }
    }
  }
  /* … */
}
```

3. Within our event handler, create a new `WebSocket` instance as shown in the following code:

```
load: function() {
  // maintain a reference to the chart
  var self = this;

  var connection = new WebSocket('ws://localhost:8001/websocket');
}
```

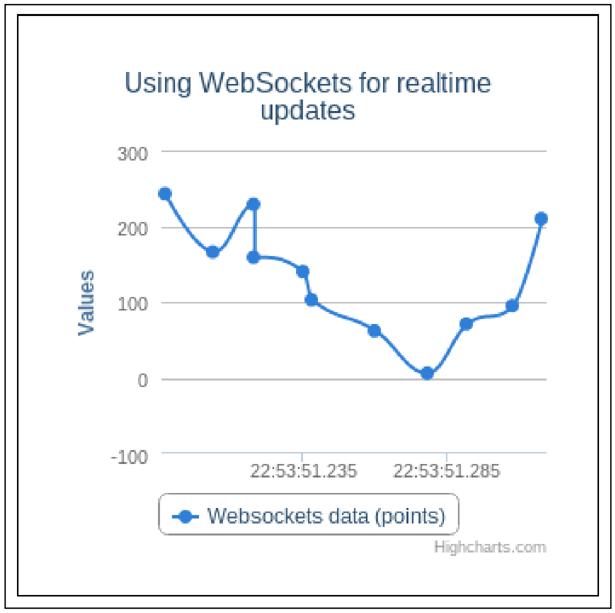
4. Create an `onmessage` event handler on our WebSocket as shown in the following code:

```
load: function() {
  // maintain a reference to the chart
  var self = this;

  var connection = new WebSocket('ws://localhost:8001/websocket');
  connection.onmessage = function(event) {
  }
}
```

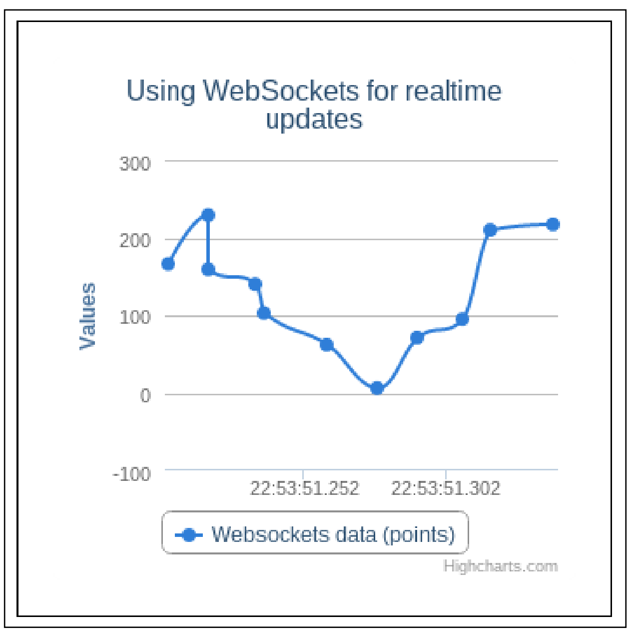
5. In the `onmessage` event handler, process the data by accessing `event.data` and replace the chart data using `<series>.setData` as shown in the following code:

```
load: function() {
  // maintain a reference to the chart
  var self = this;

  var connection = new WebSocket('ws://localhost:8001/websocket');
  connection.onmessage = function(event) {
    var data = JSON.parse(event.data);
    self.series[0].setData(data);
  }
}
```

The resultant chart is displayed as follows:



6. Observe the graph change periodically, as shown in the following screenshot:



How it works...

When the chart has loaded, we create a `WebSocket` object, which acts like a regular socket: it listens for new information from the other end of the socket (the server) and calls the appropriate event handler (for example, `onmessage`) whenever there is new data. In our case, every time we have new data, we reload the chart.

In this example, we created a `self` variable equal to `this`. We had to do this because otherwise, when our `connection` event handler is executed, it wouldn't be able to access our chart because `this` would refer to some other object. Alternatively, if we had some global reference to the chart, we could have just accessed the chart via that global reference.

Drilling down and filtering data

Sometimes, we have data that is not only categorical but also hierarchical. While it can be useful to combine that data into top-level elements, we may want to drill down into the data or filter it to see different relationships. This recipe deals with how we can drill down and filter our existing data.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

For our example, we will assume that our data looks as follows:

```
var name = 'Drilling Down and Filtering Data'
var categories = ['IE Users', 'Chrome Users', 'FF Users'];
var data =[{
  y: 50,
  category: 'IE Users',
  categories: [
    'IE 10', 'IE 9', 'IE 8', 'IE 7'
  ],
  data: [{
    y: 25,
    name: 'IE 10'
  }, {
      y: 55,
    name: 'IE 9'
  }, {
    y: 15,
    name: 'IE 8'
  }, {
```

```
      y: 5,
      name: 'IE 7'
    }]
  }, {
    y: 30,
    category: 'Chrome Users',
    categories: ['Most recent', 'Older versions'],
    data: [{
      y: 95,
      name: 'Most recent'
    }, {
      y: 5,
      name: 'Older versions'
    }]
   ];
```

How to do it...

To get started, perform the following steps:

1. Create a function to handle drilldown, and redraw the chart as shown in the following code:

   ```
   var redrawChart = function(name, categories, data) {
     chart.xAxis[0].setCategories(categories, false);
     chart.series[0].remove(false);
     chart.addSeries({
       name: name,
       data: data
     });
     chart.redraw();
   };
   ```

2. Define options for our chart as shown in the following code:

   ```
   var options = {
     chart: {
       type: 'column'
     },
     title: {
       text: name
     },
   ```

```
  xAxis: {
    categories: categories
  },
  series: [{
    name: name,
    data: data
  }]
};
```

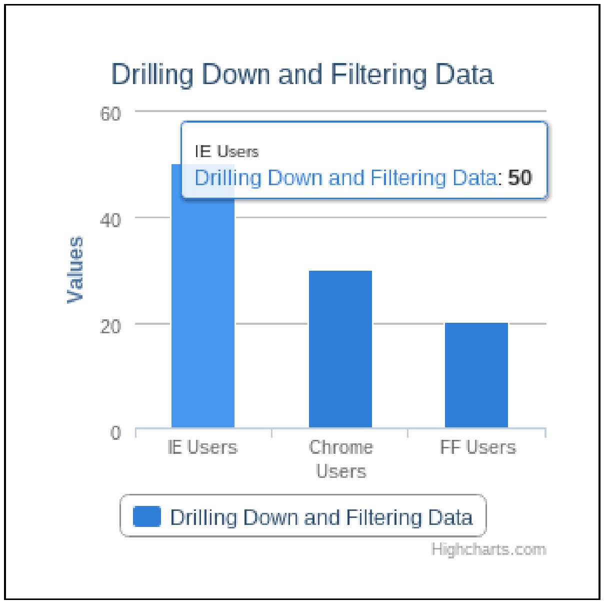
3. Create an event handler for `plotOptions.<type>.point.events.click` to handle the `click` events on the points in our chart, as shown in the following code:

```
var options = {
  /* … */
  plotOptions: {
    // we could also change this to 'bar' or 'pie' depending on
    the type
    column: {
      point: { // options for a 'point' in the chart
        events: {
          click: function () {
            // Our code goes here
          }
        }
      }
    }
  }
};
```

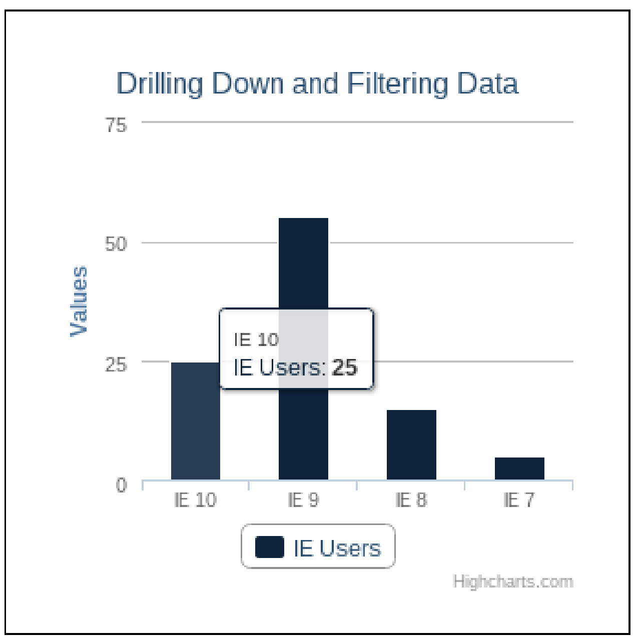
4. Within our `click` handler, call our function with different parameters depending on whether the drilldown data is available, as shown in the following code:

```
click: function() {
  if (this.categories) {
    // drilldown data is available!
    redrawChart(this.category, this.categories, this.data);
  } else {
    // drilldown data unavailable; use top level data
    redrawChart(name, categories, data);
  }
}
```

The resultant chart is displayed as follows:



5. Observe the chart after clicking on a column.



How it works...

As previously mentioned, event handlers allow us to handle different interactions on the chart. In this case, we can listen for click events on a point in the chart (here, represented by a column) and take some action depending on the value of the point we clicked on. We could just as easily have done something similar with a bar or pie chart.

There's more...

As long as you have a legend and the data is split into series, you can automatically filter data by clicking on the series legend.

Using CSV, XML, or JSON with Highcharts

Even though much of the data we work with in charts comes via a backend source, there are times when we may need to get our data from an external source, such as a CSV file or an XML file.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*. Refer to the steps as described in the *Using AJAX for polling charts* recipe discussed earlier in this chapter.

How to do it...

To get started, perform the following steps:

1. Create a new event handler for the chart load event as shown in the following code:

```
var options = {
  chart: {
    events: {
      load: function() {
        // maintain a reference to the chart
        var self = this;

        // code goes here
      }
    }
  }
};
```

2. Within our event handler, call `$.ajax` with a `success` function to handle the retrieved data. We will also need to provide `dataType`, which is dependent on the type of data we are fetching. For a CSV file, the `dataType` is `text`, for XML it is `xml`, and for JSON it is `json`, as shown in the following code:

```
load: function() {
  // maintain a reference to the chart
  var self = this;

  $.ajax('http://path/to/data/source', {
    dataType: 'json',
    success: function(data) {
      // code goes here
    }
  });
}
```

3. Inside our `success` function, process the data as desired, then finally replace the existing series data by using `<series>.setData`, as shown in the following code:

```
success: function(data) {
  // do any data processing

  // pick the first series, but could be any series
  self.series[0].setData(data);
}
```

For CSV files, data will be returned as a string. In order to use the data, we would need to process the data as shown in the following code:

```
success: function(data) {
  var delimiter = ',';
  var explodedData = data.split(delimiter);
  var csvArray = [];
  for(var i=0; i < explodedData.length; i++) {
    // need to convert strings to numbers
    explodedData.push(parseInt(explodedData[i]))
  }

  self.series[0].setData(explodedData);
}
```

For XML files, the data will be returned as a string, as well. However, we can traverse it using jQuery. How we process that data is highly dependent on what that XML looks like.

Assuming that we have XML data as follows:

<xml><row><x>0</x><y>1</y></row></xml>

We can process the data as follows:

```
success: function(data) {
  var xmlData = [];
  $(data).find('row').each(function(idx, elem) {
    var x = parseInt($(elem).find('x').text());
    var y = parseInt($(elem).find('y').text());
    xmlData.push([x, y]);
  });

  self.series[0].setData(xmlData);
}
```

How it works...

The `$.ajax` function, such as `$.getJSON`, allows us to fetch the data from an external source and execute a callback function when it is completed. That callback function includes whatever resultant data it has retrieved, and all we need to do is process it. Even after setting up all the required tools, as we have done in this example, the chart will not render until our AJAX call is successful.

There's more...

We can combine the idea from this recipe with our example from the *Using AJAX for polling charts* recipe to have polling charts with a CSV or XML source. This can be done if we replace the `$.getJSON` call with the appropriate `$.ajax` call from this recipe.

Handling cross-domain data

We may not always have a direct access to the data we want to work with; it may not be on our server or in a file we have access to. It may be a public URL on a different domain. Due to the way in which AJAX requests are handled, we can't use the exact same methods we used previously. There are still means to access cross-domain data, namely, **JSON with Padding** (**JSONP**).

> It is worth noting that we can only use JSONP for services that support JSONP. If we do not control the backend service or the backend service does not support JSONP, we will not be able to use this technique.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*. Refer to the steps as described in the *Using AJAX for polling charts* recipe discussed earlier in this chapter.

How to do it...

To get started, perform the following steps:

1. Create a new event handler for the chart load event as shown in the following code:

```
var options = {
  chart: {
    events: {
      load: function() {
        // maintain a reference to the chart
        var self = this;

        // code goes here
      }
    }
  }
}
```

2. Within our event handler, call `$.ajax` with `jsonp` for the `dataType` parameter and a `success` function to handle the retrieved data as shown in the following code:

```
load: function() {
  // maintain a reference to the chart
  var self = this;

  $.ajax('http://localhost:8000/jsonp/series', {
    dataType: 'jsonp',
    success: function(data) {
      // code goes here
    }
  });
}
```

3. Inside our `success` function, process the data as desired, then finally replace the existing series data using `<series>.setData` as shown in the following code:

```
success: function(data) {
  // do any data processing

  // pick the first series, but could be any series
  self.series[0].setData(data);
}
```

How it works...

Normally, AJAX requests can't access other domains due to the same-origin policy, which states that AJAX requests can only obtain data from the same origin, that is, the same domain. JSONP avoids this problem as the server wraps the results in a JavaScript function, and that entire result is loaded as a `<script>` tag and executed. Since a `<script>` tag is exempt from the same-origin policy, we can make requests from other domains. However, since we use jQuery to handle our AJAX requests, all of this complexity is abstracted away for us. For more details on cross-domain JavaScript, check out the Mozilla Developer Network article on the same-origin policy at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript` or the article on Defining Safer JSON-P at `http://json-p.org`.

Handling dates

So far none of our charts have dealt with dates, which is critical for time-series charts. In this recipe, we'll look at some of the different ways in which Highcharts can handle and display dates.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Define the options as shown in the following code:

```
var options = {
  title: {
    text: 'Handling Dates',
    type: 'spline'
  }
};
```

2. Set `xAxis.type` or `yAxis.type` to `datetime` for our chart, as shown in the following code:

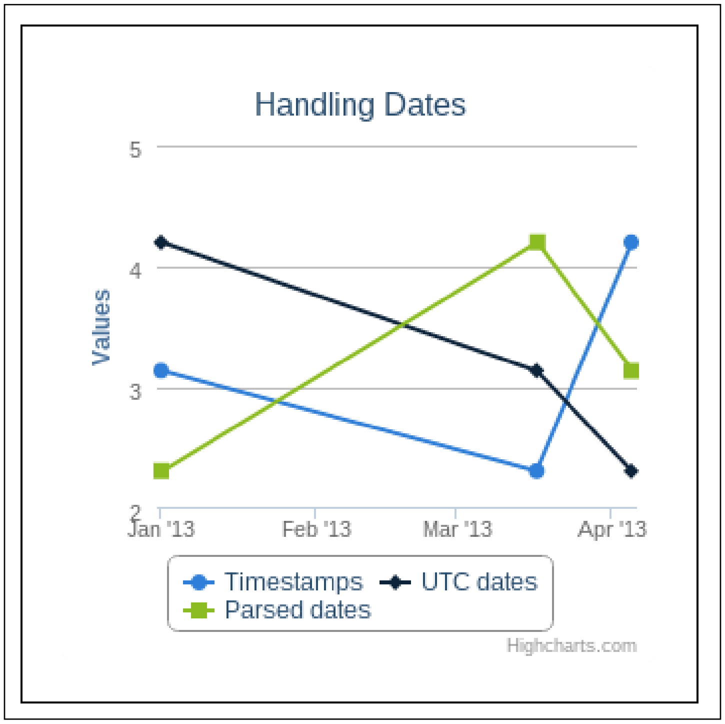
```
var options = {
  /* … */
  xAxis: {
    type: 'datetime'
  }
};
```

3. Create a new series for our chart with timestamp values for our data as shown in the following code:

```
var options = {
  /* … */
  series: [{
    name: 'Timestamps',
    data: [
      [1356998400000, 3.141592], // 2013-01-01 ET
      [1363478400000, 2.314], // 2013-03-17 ET
      [1365120000000, 4.2] // 2013-04-05 ET
    ]
  }]
};
```

Highcharts expects all dates to be a timestamp. These timestamps are very similar to the UNIX timestamps (for example, seconds since January 1, 1970) except that they are measured in milliseconds instead of seconds. For example, a UNIX timestamp for January 1, 2000 would be 946684800, but Highcharts would expect it to be 946684800000.

To include the timestamp, we'll need to use either an array of the arrays format (for example, `[timestamp, y]`) or an array of the objects format (for example, `{x: timestamp, y: y}`). The resultant chart is shown as follows:



How it works...

As long as the data provided to the chart is a timestamp, Highcharts will automatically, correctly handle labeling axes and re-labeling (if the chart is zoomable or if the chart is redrawn), and it will plot the data correctly even if it is in irregular intervals.

If our data is not a timestamp, we might still be able to format it correctly using `Date.parse`. For example, our data may look as follows:

```
var data = [
  ['2013-01-01', 4.2]
  ['2013-03-17', 3.14159]
  ['2013-04-05', 2.314]
];
```

We can convert this data using either of the two methods as shown in the following code:

```
var newData = [];
var elem, newDate, oldDate;
for (var i=0; i < data.length; i++) {
  elem = data[i];
  oldDate = elem[0];

  newDate = Date.parse(oldDate);
  newData.push([newDate, elem[1]])
}
```

Rather than preprocessing the data, we can process it immediately using an immediate function. Immediate functions take the `(function () {/*function contents*/}())` form. Using our previous example, we can process the data immediately as shown in the following code:

```
var options = {
  series: [{
    data: (function(){
      var newData = [];
      var elem;
      for (var i=0; i < data.length; i++) {
        elem = data[i];
        newData.push([Date.parse(elem[0]), elem[1]]);
      }
      return newData;
    }())
  }]
};
```

There's more...

We do not necessarily need a timestamp if the data is in regular intervals. In this case, we can use `Series.pointStart`, which determines the date or time when the first data point begins, and `Series.pointInterval`, which determines the time interval between points, as shown in the following code:

```
var options = {
  series: [{
    data: [1,2,3,4],
    pointStart: Date.UTC(2013,0,1), // January 1, 2013
    pointInterval: 24*3600*1000 // 1 day in milliseconds
  }]
};
```

Highcharts will automatically use certain date/time formats depending on the scale of the chart data, which is shown in the following code. We can override these formats using `xAxis.dateTimeLabelFormats`. The format used is a subset of the formats provided by PHP's `strftime` function found at `http://php.net/manual/en/function.strftime.php`:

```
var options = {
  xAxis: {
    dateTimeLabelFormats: {
      millisecond: '%H:%M:%S.%L', //e.g. 00:01:02.000
      second: '%H:%M:%S', // e.g. 00:01:02
      minute: '%H:%M', // e.g. 21:42
      hour: '%l:%M%P', //e.g. 3:14pm
      day: '%a %d', // e.g. Sun 01
      week: '%b-%e', //e.g. Jan-2
      month: '%B \'%y', // e.g. February '13
      year: '%Y' //e.g. 2013
    }
  }
};
```

It is even possible to create our own date label format strings. `Highcharts.dateFormats` is an object where each key is a string that we can use in `xAxis.dateTimeLabelFormats`, and each value is a function that takes a timestamp and returns the formatted string. For example, if we want to create a "week of the year" format, we can do it in the following way:

```
Highcharts.dateFormats = {
  W: function(timestamp) {
    var date = new Date(timestamp);
    var dateYear = date.getUTCFullYear();
    var firstYearDay = Date.UTC(dateYear, 0, 1);
    var dayInMs = 24 * 60 * 60 * 1000;
    var dayOfYear = (date - firstYearDay) / dayInMs;
    return Math.floor(dayOfYear / 7);
  }
};
```

More details on date formats can be found in the API documentation at `http://api.highcharts.com/highcharts#xAxis.dateTimeLabelFormat`.

3

Handling User Interaction

In this chapter, we will cover the following recipes:

- Creating a simple poll
- Making graphs zoomable
- Creating master detail graphs
- Slicing and dicing time data
- Annotating a chart
- Developing dynamic tooltips
- Taking actions on events
- Adding events after the chart has rendered

Introduction

So far we have dealt with input coming from other sources such as various backends, CSV files, and XML files, all sources external to the user. This chapter focuses on how to handle input from the user, specifically how to handle different events that are fired within a chart.

Creating a simple poll

One of the simplest charts that we've introduced is a poll, which is a tally of votes or choice selections between a number of options, often displayed as a histogram with the option for a user to make a selection and have it added to the tally. This recipe covers the basics of how to handle that interaction and update the chart.

Getting ready

For setting up a basic page and installing jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. In addition to our container `<div>`, create voting buttons as shown in the following code:

```
<div id='container'></div>
<div>
    Is it easy to create a poll?
    <input type='button' value='Yes' id='yes' class='vote-
      btn'></input>
    <input type='button' value='No' id='no' class='vote-
      btn'/></input>
</div>
```

2. Define options for a new column or bar chart as shown in the following code:

```
var options = {
    chart: {
        type: 'column'
    },
    title: {
        text: 'Creating a simple poll'
    },
    subtitle: {
        text: 'Is this easy?'
    },
    series: [{
        name: 'Yes',
        data: [0]
    }, {
        name: 'No',
        data: [0]
    }]
};
```

3. Render the chart and obtain a reference to it as shown in the following code:

```
$('#container').highcharts(options);
var chart = $('#container').highcharts();

// Alternatively
var chart =
  $('#container').highcharts(options).highcharts();
```

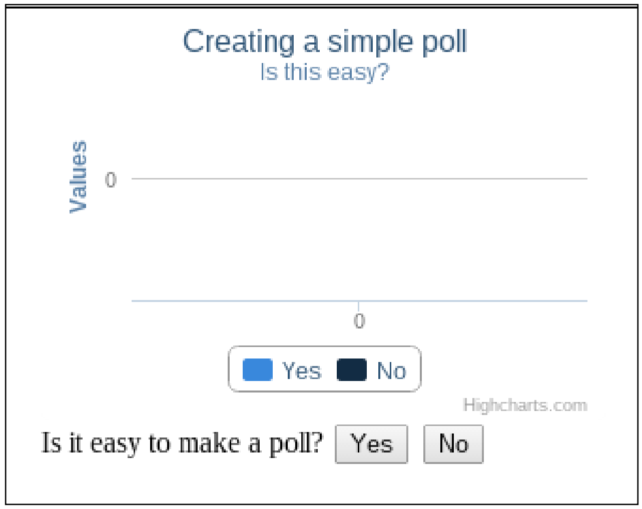
4. Create a vote function as shown in the following code:

```
var vote = function(event) {
    var $button = $(this),
        value = $button.attr('id'),
        series, data;

    if(value === 'yes') {
        series = chart.series[0];
    } else {
        series = chart.series[1];
    }

    votes = series.data[0].y || 0;
    votes += 1;

    series.setData([votes]);
};
```

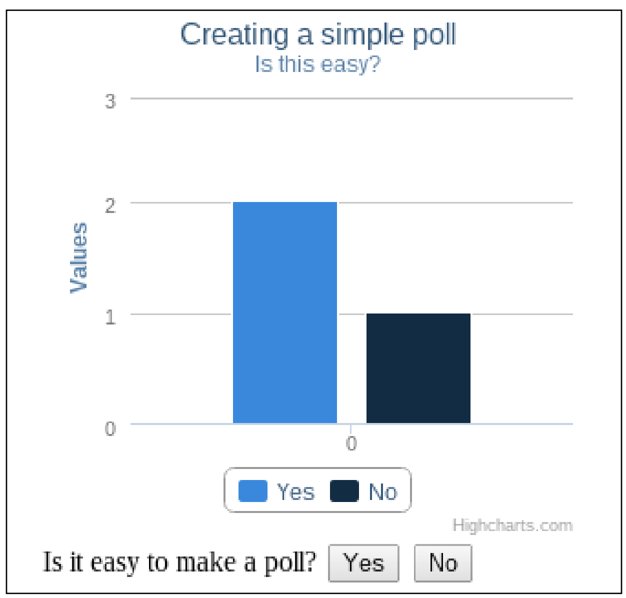
5. Attach the `vote` function as an event handler for the buttons as shown in the following code:

```
$('.vote-btn').on('click', vote);
```

6. Click on the different vote buttons and observe the difference as shown in the following screenshot:



Before voting



And after

How it works...

There isn't anything special about our chart; however, once we've obtained a reference to it via `$(selector).highcharts()` (or `$(selector).highcharts(options).highcharts()` when we're rendering the chart), we can take actions on the chart without defining those actions in the chart options.

In our case, we use jQuery's `.on(event, handler)` method to register and click on **handler** on the voting buttons. The handler has one argument, event, which contains information about the click event. From here, it is possible to find out what element was clicked on using `event.target`, then we just need to figure out which series to get the previous value from and to update using `series.setData()`.

For more information on jQuery event handling, check out the jQuery API documentation at `http://api.jquery.com/on/`.

Making graphs zoomable

When working with a small amount of data, it is possible to fit it all within the bounds of the graph. When working with larger sets of data, it becomes necessary to manage the data differently: summarize the data, filter it, and so on. One alternative to these approaches, especially in the case of chronological data, is to make a graph that we can zoom in and out of.

Getting ready

For setting up a basic page and installing jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Define chart options as we normally would for our chart, as shown in the following code:

```
var options = {
    chart: {
        type: 'spline',
    },
    title: {
        text: 'Making graphs zoomable'
    },
```

```
        series: [{
            name: 'Our data',
            data: [/* Our data goes here */]
        }]
    };
```

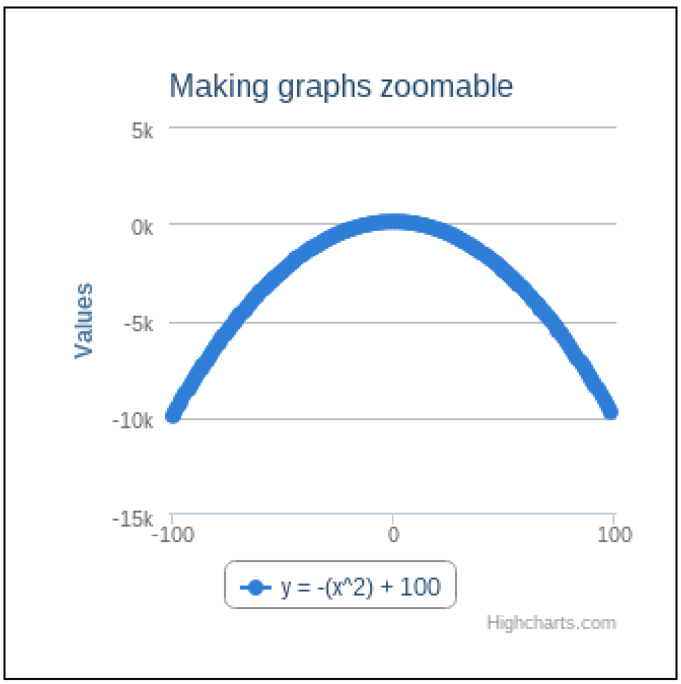
2. Set `options.chart.zoomType` to `x`, as shown in the following code:

```
var options = {
    chart: {
        type: 'spline',
        zoomType: 'x'
    },
    /* … */
};
```

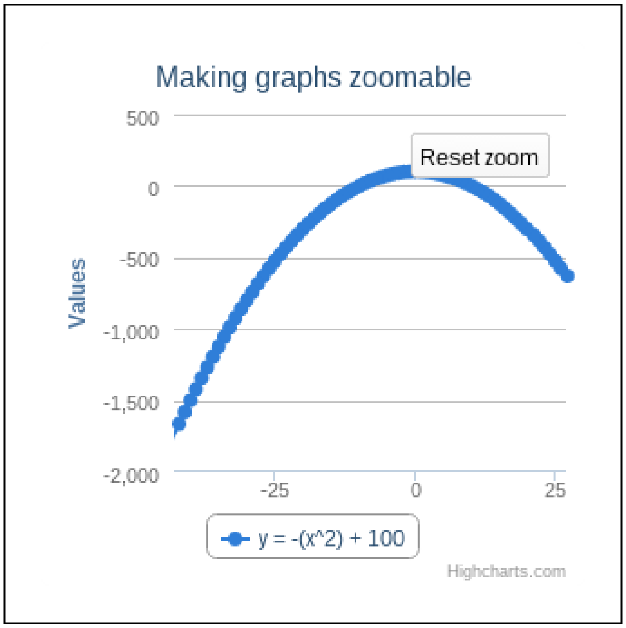
3. Render the chart using the following code:

```
$('#container').highcharts(options);
```

4. Zoom in to the chart to see it in more detail, as shown in the following screenshot:



Before zooming



And after

How it works...

If we click-and-drag over an area on our chart, we'll notice that Highcharts begins selecting an area of our chart. When we finish this selection, the chart will automatically zoom in and show the selected area in detail, along with a **Reset Zoom** button for us to go back to the full view of the data.

In our example, our selection was along the x axis; this is because we set `chart.zoomType` to `x`. We can set it to `y` too if we want to select data along the y axis only, or set it to `xy` if we want to select a specific area of the chart.

Creating master details graphs

Often times in data, especially chronological data, there are patterns. These patterns can get lost when we are presented with too much, or too little, data. There are also times when we want to keep both views at hand: the details as well as the overall data. These types of graphs are known as master details graphs.

Getting ready

For setting up a basic page and installing jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Unlike normally, we'll create three containers: one for the master graph, one for the details graph, and one to wrap both of them, as shown in the following code:

```
<div id='graph-container'>
    <div id='detail'></div>
    <div id='master'></div>
</div>
```

2. Create a variable for what will be our details chart, as shown in the following code:

```
var detailsChart;
```

3. Create a selection handler for when we select an area in our master chart, as shown in the following code:

```
var selectionHandler = function(event) {
    var selection = event.xAxis[0],
        xAxis = this.xAxis[0],
        extremes = xAxis.getExtremes(),
        max = selection.max,
        min = selection.min,
        data = [];

    // mask unselected areas
    xAxis.removePlotBand('before-selected');
    xAxis.addPlotBand({
        id: 'before-selected',
        color: 'rgba(0, 0, 0, 0.2)',
        from: extremes.min,
        to: min
    });
    xAxis.removePlotBand('after-selected');
    xAxis.addPlotBand({
        id: 'after-selected',
        color: 'rgba(0, 0, 0, 0.2)',
        from: max,
        to: extremes.max
    });
```

```
        jQuery.each(this.series[0].data, function(i, point) {
            if (min < point.x && point.x < max) {
                data.push({x: point.x, y: point.y});
            }
        });
        detailsChart.series[0].setData(data);

        // don't do whatever we would normally do on select
        return false;
    };
```

4. Define options for our master chart using the next code snippet.

> As the master chart is intended to give a general idea of the data, we will disable most labels and markers aside from those on the x axis.

```
var masterOptions = {
    chart: {
        zoomType: 'x',
        events: { selection: selectionHandler}
    },
    title: {text: null},
    tooltip: {formatter: function (){return false;}},
    legend: {enabled: false},
    credits: {enabled: false},
    yAxis: {
        title: {text: null},
        labels: {enabled: false}
              },
    xAxis: {
        title: {text:null},
        plotBands: [{
            id: 'before-selected', color: 'rgba(0, 0, 0,
              0.2)', from: -100, to: -50
        },{
            id: 'after-selected', color: 'rgba(0, 0, 0,
              0.2)', from: 50, to: 100
        }]
    },
    series: [{
        data: (function() {
            var data = [], x= -100;
            while (x < 100) {
```

```
                data.push([x, Math.pow(x,3)]);
                x++;
            }
            return data;
        }())
    }]
};
```

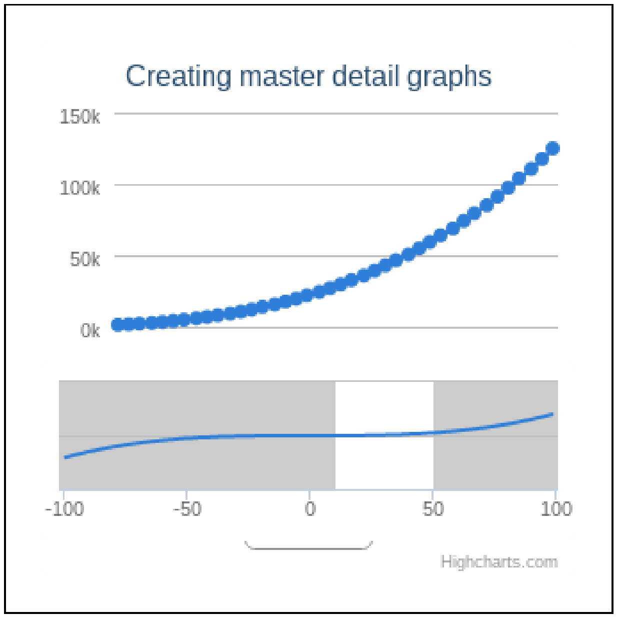
5. Create a callback function for when the master chart has rendered, as shown in the following code:

```
var createDetailsChart = function (master) {
    var options, data = [];

    //define our options for the details chart as we like
    options = {
        title: {
            text: 'Creating master detail graphs'
        },
        series: [{
            data: master.series[0].data
        }]
    };
    detailsChart =
  $('#detail').highcharts(options).highcharts();
};
```

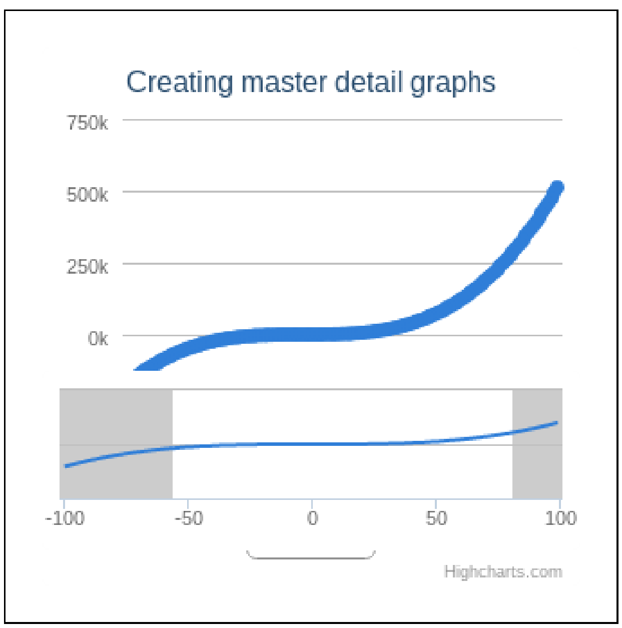
6. Render the master chart with our callback function to render the details chart, as shown in the following code:

```
$('#master').highcharts(masterOptions, createDetailsChart);
Re-position the charts
$('#graph-container').css('position', 'relative');
$('#master').css({
    position: 'absolute',
    width: '100%'      });
```

7. Select a large portion of the master graph, as shown in the following screenshot:



Zoomed in very closely



And examining a larger section of the graph

How it works...

There are a few important things discussed in this recipe:

- Creating the details chart after the master chart is loaded
- Handling the selection on the master chart
- Masking and unmasking areas according to the selection

The `$(selector).highcharts()` function can also take a second argument, as we've done in this example, to handle the `load` event for the chart. In this case, the first argument of our load handler is a reference to the rendered chart. This is equivalent to setting a handler on `chart.events.load`.

We have also created a selection handler on our master chart. This handler allows us to take some action after we have selected an area on the chart, and the first argument (`event`) to the handler contains information about the selection via `event.xAxis[0]` or `event.yAxis[0]` depending on how we set `chart.zoomType`.

Lastly, we also use `<axis>.addPlotBand` and `<axis>.removePlotBand` to create a mask over our chart. With `<axis>.addPlotBand`, we define a from value, a to value, a color, and an ID; we can remove this band with `<axis>.removePlotBand`, provided that we have the ID of the correct band.

Slicing and dicing time data

As we've seen, displaying the same data in a different way can often reveal patterns we hadn't previously seen. Another way to view data is to group it into different buckets; for example, by hour, or by month.

Getting ready

For setting up a basic page and installing jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

In addition to this basic setup, we will need to install `underscore`, a functional programming library for JavaScript:

1. Edit `bower.json` to add `underscore` as a dependency, as shown in the following code:

```
{
  "name": "highcharts-cookbook-chapter-3",
  "dependencies": {
    "highcharts": "~3.0",
    "jquery": "^1.9",
```

```
    "underscore": "~1.5.2"
  }
}
```

2. Install your dependencies using `bower`, as shown in the following code:

```
bower install
```

How to do it...

To get started, perform the following steps:

1. In addition to the chart container `<div>`, create three buttons, as shown in the following code:

```
<div id='container'></div>
<div class='buttons'>
    Group data:
    <input type='button' value='Chronologically'
      id='chrono'></input>
    <input type='button' value='By day' id='day'></input>
    <input type='button' value='By hour' id='hour'></input>
</div>
```

2. Store or create a copy of your data outside of your chart options so that you can use the data later, as shown in the following code:

```
var chartData = [{
    x: Date.UTC(2013,0,1),
    y: 42
}, /* More data goes here */];
```

3. Define the options for your chart, as shown in the following code:

```
var options = {
    chart: { type: 'column' },
    title: {text: 'Slicing and dicing time data' },
    subtitle: {text: 'All changes conducted on original
      data' },
    xAxis: { type: 'datetime' },
    series: [{ data: chartData}]
};
Render the chart and obtain a reference to it
var chart =
  $('#container').highcharts(options).highcharts();
```

4. Create an event handler for your buttons, as shown in the following code:

```
var groupData = function (event) {
    var $button = $(event.target),
        groupedData,
        displayData = {},
        xAxisOptions = {},
        data = _.clone(chartData);

    switch($button.attr('value')) {
        case 'Chronologically':
            break;
        case 'By day':
            break;
        case 'By hour':
            break;
    }
    chart.xAxis[0].update(xAxisOptions);
    chart.series[0].update(displayData);
};
```

5. Create the case to handle chronological data, as shown in the following code:

```
case 'Chronologically':
    xAxisOptions = { type: 'datetime' };
    displayData = { data: data };
    break;
```

6. Create the case to handle day data, as shown in the following code:

```
case 'By day':
    xAxisOptions = { type: 'category' };

    groupedData = _.groupBy(data, function(point) {
        return new Date(point.x).getUTCDay();
    });

    groupedData = _.chain(groupedData).map(function(value,
      index) {
        var y = _.chain(value).pluck('y').reduce(function(sum,
        num){
            return sum + num;
        }).value();
        return { y: y, name: index } ;
    });

    displayData = { data: groupedData.value() }
    break;
```

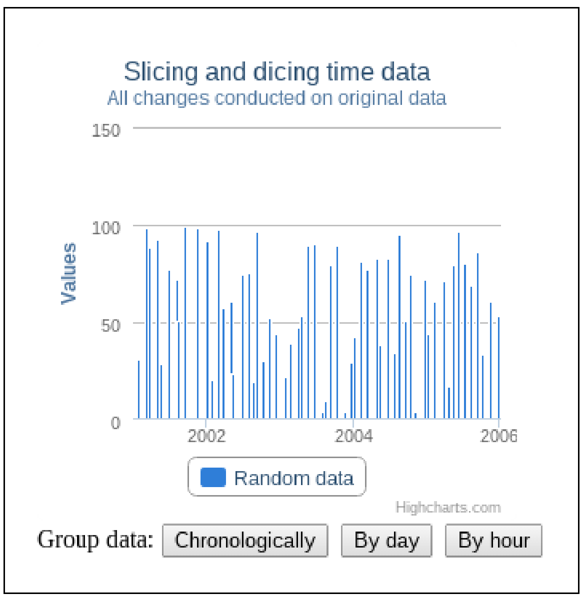
7. Create the case to handle hour data (very similar to 'day' data), as shown in the following code:

```
case 'By hour':
    xAxisOptions = { type: 'category' };

    groupedData = _.groupBy(data, function(point) {
        return new Date(point.x).getUTCHours();
    });

    groupedData = _.chain(groupedData).map(function(value,
      index) {
        var y = _.chain(value).pluck('y').reduce(function(sum,
        num){
            return sum + num;
        }).value();
        return { y: y, name: index } ;
    });

    displayData = { data: groupedData.value() }
    break;
```

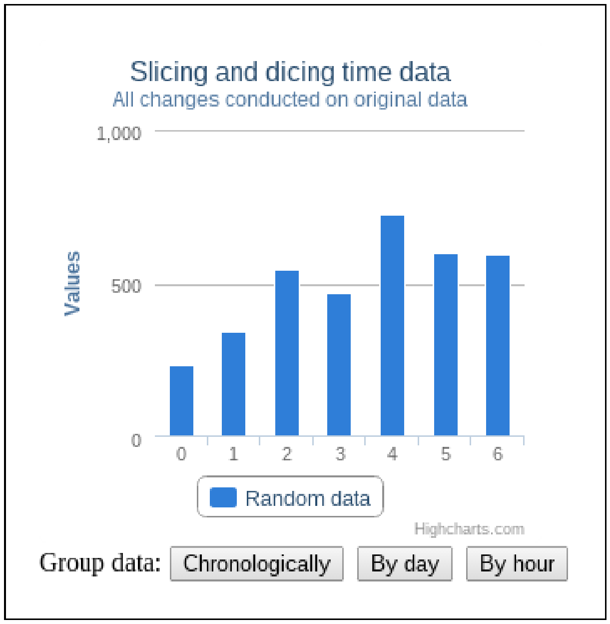
8. Bind your buttons to the event handler, as shown in the following code:

```
$('#chrono').on('click', groupData);
$('#day').on('click', groupData);
$('#hour').on('click', groupData);
```

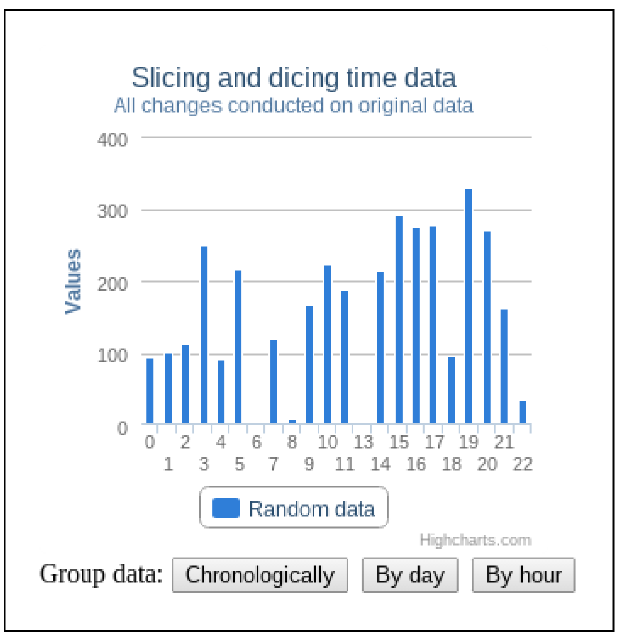
9. Click on the **Chronologically** button, as shown in the following screenshot:



10. Click on the **By day** button, as shown in the following screenshot:



11. Click on the **By hour** button, as shown in the following screenshot:



How it works...

When we click on our different grouping buttons, our event handler is called, and we can begin transforming the data depending on the button clicked. A lot of the data transformation is handled by a number of clever underscore functions.

The `_.clone(obj)` script makes a copy of `obj`, which is good as it allows working on the original data without having to worry about side effects.

The `_.groupBy(collection, groupFn)` script takes a collection (in this case, our data) and transforms it into an object, where the keys are our grouping, and the values are arrays of our original objects that are in that group. When we call this function the first time, it takes our original data and transforms it into something like the following data:

```
{'0': […], '1': […], …}
```

The next part may look complicated, but can actually be broken down into simpler steps.

The `_.chain(obj)` script wraps an object in an `underscore` wrapper. This is helpful as it makes it easier to chain function calls together without always having to provide `obj` as the first argument. We do, however, need to call `.value()` in order to convert the wrapper object into an actual result.

The `_(obj).map(mapFn)` script applies `mapFn` to every element of `obj`, and replaces each value with whatever we return in `mapFn`.

Our map function has two arguments, `value` (the array of values in the group) and `index` (the group). By using `_.chain(value).pluck('y')`, we can get just the `y` values from all of our points, and by calling `.reduce(reduceFn)`, we can reduce our array to a single value by repeatedly calling `reduceFn` with the aggregated value (`sum`, for our function) and the next value (`num`, in our case).

Annotating a chart

It can be useful to include information about a chart on a chart. While tooltips is one option, there are times when we may want to leave a note to ourselves. Due to the various events that Highcharts is able to capture, it is possible for us to annotate charts.

Getting ready

For setting up a basic page and installing jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. In addition to the regular chart container, create a text area to enter your annotations, as shown in the following code:

```
<div id='container'></div>
<textarea id='annotation'></textarea>
```

2. Create a click handler for your chart, as shown in the following code:

```
var annotateChart = function (event) {
    var x, y, text, box, content;
    x = event.chartX;
    y = event.chartY;

    // get content from the textarea
    content = $('#annotation').val();

    // create the text
    text = this.renderer.text(content, x, y).attr({
        zIndex: 2
    }).add();

    // create the border
    box = text.getBBox();
    this.renderer.rect(box.x-5, box.y-5, box.width+10,
      box.height+10, 5)
        .attr({
            fill: '#ffffff', stroke: 'gray', 'stroke-
              width': 1, zIndex: 1
        }).add();

    // empty out the textbox
    $('#annotation').val('');
};
```

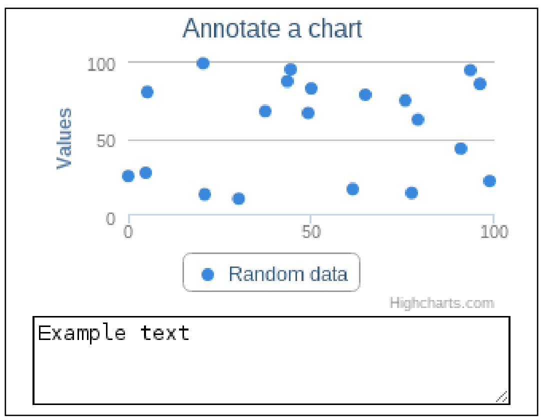
3. Define your chart options as shown in the following code:

```
var options = {
    chart: {
        type: 'scatter',
        events: {click: annotateChart}
    },
    title: {text: 'Annotate a chart'},
    series: [{/* Our data goes here */}]
};
```

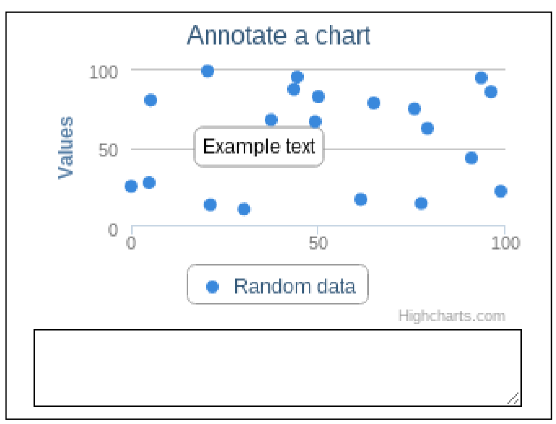
4. Render your chart as shown in the following code:

```
$('#container').highcharts(options);
```

5. Annotate the chart as shown in the following screenshot:



Before annotating the chart



And after

How it works...

The `chart.events.click` script allows us to define a `click` handler. The event that we get from that handler includes information about the point that we've clicked on in the chart in terms of x and y coordinates (via `event.chartX` and `event.chartY`). By leveraging the chart's renderer, we can draw all sorts of different shapes. In this recipe, we use `this.renderer.text(text, x, y)` to render the text, and then add it to the chart via `.add()`. We can get the bounding box of our text using `<element>.getBBox()` and use it to figure out where to place our border.

Developing dynamic tooltips

Highcharts tooltips are fairly powerful by default, but they do have some limitations. It can be difficult to attach different controls inside tooltips. For that reason, we can develop dynamic tooltips.

Getting ready

For setting up a basic page and installing jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

In addition to this basic setup, we will need to install `underscore`, a functional programming library for JavaScript:

1. Edit `bower.json` to add `underscore` as a dependency, as shown in the following code:

```
{
  "name": "highcharts-cookbook-chapter-3",
  "dependencies": {
    "highcharts": "~3.0",
    "jquery": "^1.9",
    "underscore": "~1.5.2"
  }
}
```

2. Install your dependencies using `bower`, as shown in the following code:

```
bower install
```

How to do it...

To get started, perform the following steps:

1. In addition to your chart container, create a tooltip container and a wrapper `<div>` for the two elements, as shown in the following code:

```
<div id='wrapper' style='position: relative'>
    <div id='container'></div>
    <div id='tooltip'></div>
</div>
```

2. Create a function to position your tooltip, as shown in the following code:

```
var positionTooltip = function (chart, point) {
    var selectedPoint = (chart.getSelectedPoints() ||
      [])[0],
        referencePoint, x, y;

    // Stick the tooltip next to the selected point
    if (selectedPoint) {
        referencePoint = selectedPoint;
    } else {
        referencePoint = point;
    }

    x = referencePoint.plotX + chart.plotLeft + 25;
    y = referencePoint.plotY + chart.plotTop + 25;

    $('#tooltip').css({
         position: 'absolute',
         top: y,
         left: x
     });
};
```

3. Create templates necessary for your tooltip, as shown in the following code:

```
var buttonTemplate = _.template('<input type="button"
  class="<%= cls %>" value="<%= value %>"/>');
var coordsTemplate = _.template('(<%= x %>, <%= y %>)');
```

4. Create a function to handle updating the template, as shown in the following code:

```
var updateTooltipText = function(x, y) {
    var tooltipString = coordsTemplate({x: x, y: y}) +
      '<br/>';
    tooltipString += buttonTemplate({cls: 'modifyValue',
      value: '+'});
    tooltipString += buttonTemplate({cls: 'modifyValue',
      value: '-'}) + '<br/>';
    tooltipString += buttonTemplate({cls: 'removePoint',
      value: 'Remove Point?'});

    $('#customTooltip').html(tooltipString);
};
```

5. Create a `select` handler to handle point selection, as shown in the following code:

```
var pointSelect = function (event) {
    var tooltipString = "";

    // position tooltip
    var point = this;
    positionTooltip(this.series.chart, point);

    // create tooltip text
    updateTooltipText(parseInt(this.x), parseInt(this.y));

    // unbind any previous handlers and add this one
    $('#tooltip').off('click').on('click', '.modifyValue',
      function (event){
        var $button = $(this);
        var y = point.y;

        if ($button.attr('value') === '+') {
            y += 1;
        } else {
            y -= 1;
        }

        point.update(y);

        // Update tooltip
        updateTooltipText(parseInt(this.x), parseInt(this.y));
    });
```

```
        $('#tooltip').one('click', '.removePoint', function
          (event) {
            point.remove();
            $('#tooltip').hide().empty();
        });

        $('#tooltip').show();
    };
```

6. Create an `unselect` handler to handle when a point is unselected, as shown in the following code:

```
var pointUnselect = function (event) {
    var chart = this.series.chart;

    // A point is still selected when the `unselect` event
      fires
    if (this.selected) {
        $('#tooltip').hide().empty();
    } else {
        // reposition tooltip
        positionTooltip(chart, this);
    }
};
```

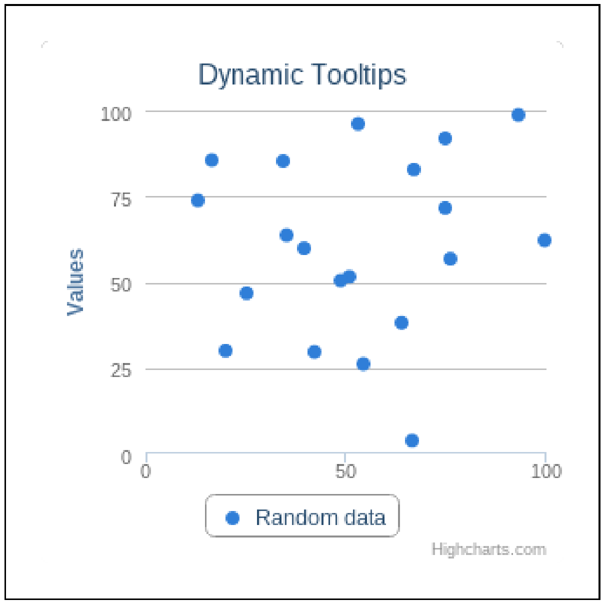
7. Define the options for your chart, including your select and unselect handlers and options to disable tooltips, as shown in the following code:

```
var options = {
    title: {text: 'Dynamic tooltips'},
    tooltip: {enabled: false},
    series: [{/* Our data goes here */}],
    plotOptions: {
        series: {
            stickyTracking: true,
            allowPointSelect: true,
            point: {
                events: {select: pointSelect, unselect:
                  pointUnselect}
            }
        }
    }
};
```

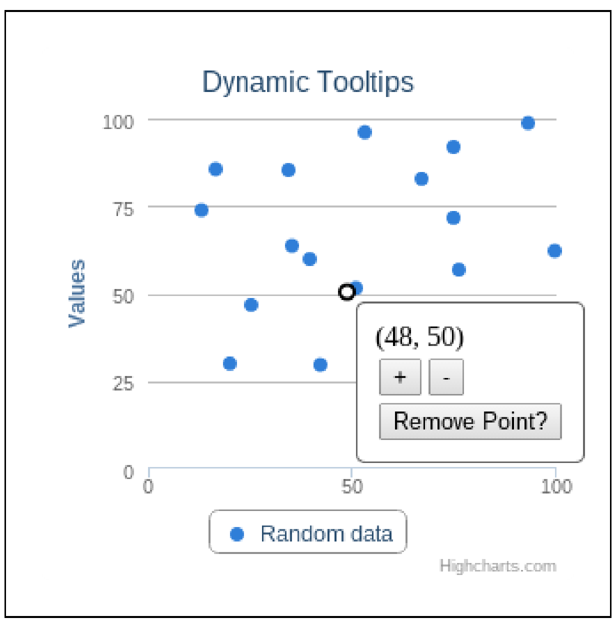
8. Render your chart using the following code:

```
$('#container').highcharts(options);
```

9. Select a point, as shown in the following screenshot:



Before selecting a point



After selecting a point

How it works...

It may appear that there is a lot going on in our example, but we can break things down into two main parts:

- Positioning our custom tooltip
- Handling events for our tooltip

Our positionTooltip function is straightforward. Since we have a reference to the chart, we can call chart.getSelectedPoints() to find out which points in the chart are selected (and we can select points because we have chart.plotOptions.series.allowPointSelect set to true). We then take the selected point (or the current point, if one isn't available) and determine where to plot our tooltip by adding the current position of the point relative to the chart (<point>.plotX and <point>.plotY) to the chart offsets (chart.plotLeft and chart.plotTop). Then we just set the positioning of the tooltip with ordinary CSS.

Since our tooltip is custom-made, we also need to handle all the relevant events. In our example, we create and position the tooltip whenever a point is selected, and display the proper coordinates via this.x and this.y. The trickiest part is ensuring that the buttons in our tooltip are bound properly. We do this by unregistering (.off('click')) any previous click handlers on our .modifyValue buttons and registering a new click handler to handle adjusting the point values. We use .one(event, selector, handler) to ensure that our remove handler is registered exactly once, and when we click on the **Remove Point?** button, we only remove the one point.

Since creating the tooltip itself is a bit repetitive, we leverage the template function from underscore. This allows us to define a template string where underscore will substitute a variable name for its value. This means when we call _.template(variable_str, {key: value}), underscore will replace any instances of <%= key %> in variable_str with value. In our recipe, we use a slightly different version of the template function—by omitting the data, underscore will pre-compile the template so that it is ready to be rendered at any point. For more information on the underscore template function, check out the underscore documentation at http://underscorejs.org/#template.

Taking actions on other events

So far, we have touched on a number of different events such as `selection`, `unselect`, `click`, and `load` that we can handle through Highcharts. There are many other user interactions that we are able to track.

Getting ready

For setting up a basic page and installing jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Define options for your chart as shown in the following code:

```
var options = {
    chart: {type: 'spline'},
    title: {text: 'Taking actions on other events'},
    series:[{ /* Our series data goes here */}],
    xAxis: { min: 0, max: 100 },
    yAxis: { min: 0, max: 100 }
}
```

2. Add event handlers for when the chart is redrawn, as shown in the following code:

```
var options = {
    chart: {
        type: 'spline',
        events: {
            redraw: function(event) {
                console.log('Chart redrawn!', event, this);
            }
        }
    },
    /* … */
};
```

3. Add event handlers for hovering over a series, as shown in the following code:

```
var options = {
    /* … */
    plotOptions: {
        series: {
```

```
                events: {
                    mouseOver: function(event) {
                        console.log('Series mouseover!', event,
                          this);
                    }
                }
            }
        }
    };
```

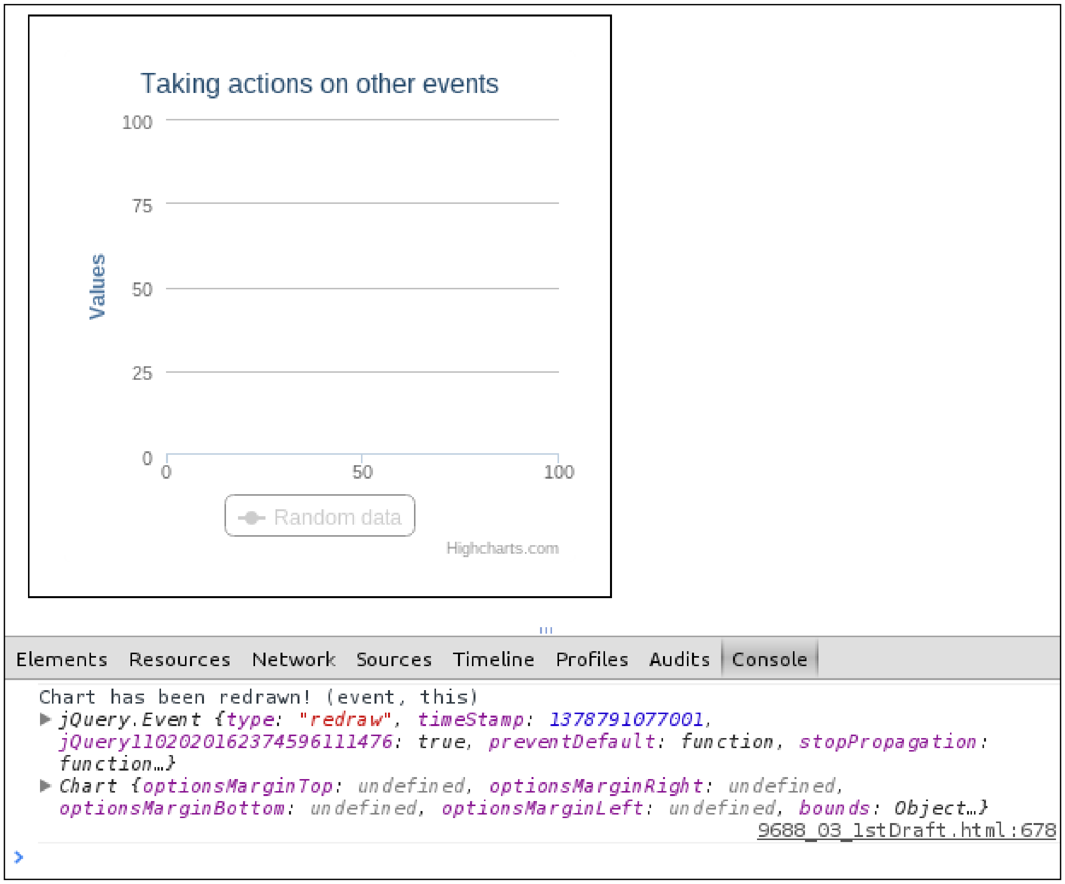
4. Render the chart as shown in the following code:

```
$('#container').highcharts(options);
```

5. Examine the developer console for a redraw message, as shown in the following screenshot:

View of the console before a redraw event

6. Examine the developer console for a `redraw` message, as shown in the following screenshot:



An example after the redraw event has fired

How it works...

Adding these event handlers is no different from the ones we've added in any other recipe. There are all sorts of different events that can be handled on the chart (for example, `chart.event.addSeries` and `chart.event.selection`), on a series (for example, `plotOptions.series.events.hide` and `plotOptions.series.events.mouseOut`), or even on an individual data point (`plotOptions.series.events.remove` and `plotOptions.series.events.update`). Events can even be handled for specific chart types (for example, `plotOptions.<type>.events` and `plotOptions.<type>.point.events`).

Adding events after the chart is rendered

There are occasions where we are unable to attach the proper event handlers to a chart before the chart has rendered. Fortunately, in these cases, we can leverage some handy Highcharts functions.

Getting ready

For setting up a basic page and installing jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe from *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Create an event handler for when we click on a series, as shown in the following code:

```
var clickSeries = function (event) {
    console.log('Captured click event!', event);
};
```

2. Get a reference to your chart as shown in the following code:

```
var chart = $('#container').highcharts();

// Alternatively, if the chart has not already rendered
var chart =
  $('#container').highcharts(options).highcharts();
```

3. Call `Highcharts.addEvent` to attach the event handler, as shown in the following code:

```
Highcharts.addEvent(chart.series[0], 'click', clickSeries);
```

How it works...

As long as we have a reference to the part of the chart we want to attach an event handler to, we can call `Highcharts.addEvent(element, event, handler)` to call `handler` when `event` is fired on `element`.

As previously mentioned, there are different events available to listen to. For more details on where to find information on different events, see API documentation on `chart.events` at `http://api.highcharts.com/highcharts#chart.events` or under the various `plotOptions` at `http://api.highcharts.com/highcharts#plotOptions`.

If we're more familiar with jQuery, it's possible to just use its event binding mechanism:

```
$(chart.series[0]).on('click' , clickSeries);
```

4

Sharing Charts on the Web

In this chapter, we will cover the following recipes:

- Rendering charts on the server side
- Exporting images to different formats
- E-mailing static charts
- E-mailing dynamic charts
- Preparing charts for printing

Introduction

Creating a chart for our own use on our own computer is certainly helpful, but there are many times when we would like to distribute our charts. This chapter will cover how to make changes in order to print, e-mail, or otherwise export our charts for different formats.

Rendering charts on the server side

One of the first steps towards being able to send charts to others is rendering the chart in a static format. While Highcharts does make it possible to render charts using its **Content Distribution Network** (**CDN**), this recipe will cover how we can render the charts on our own, which is especially important if we don't want our data being made available to the public.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe from *Chapter 1*, *Getting Started with Highcharts*.

As the bower installation does not include some important scripts, we'll need to download Highcharts normally by performing the following steps:

1. Visit the Highcharts website and download Highcharts from `http://www.highcharts.com/download`.
2. Extract the contents of the downloaded zip file to a folder called `Highcharts`.

   We will also need to install and run PhantomJS.

3. Install PhantomJS following the instructions on the PhantomJS website from `http://phantomjs.org`.
4. Change directories to `Highcharts/exporting-server/phantomjs`.
5. Modify `highcharts-convert.js` to use Highcharts instead of Highstock as shown in the following code:

```
var config = {
      /* define locations of mandatory JavaScript files */
      HIGHCHARTS: 'highcharts.js',
      HIGHCHARTS_MORE: 'highcharts-more.js',
      HIGHCHARTS_DATA: 'data.js',
      JQUERY: 'jquery.1.9.1.min.js',
      TIMEOUT: 2000 /* 2 seconds timeout for loading images
        */
```

6. Within that folder, run the following command to start a PhantomJS instance using the following code:

```
phantomjs highcharts-convert.js -host 127.0.0.1 -port 3003
```

How to do it...

To get started, perform the following steps:

1. Include `exporting.js` on our page as shown in the following code:

```
<!-- Include the verbose version of Highcharts extras -->
<script src='./bower_components/highcharts/highcharts-
  more.src.js'></script>

<!-- Include the exporting module -->
<script
```

```
src='./bower_components/highcharts/
modules/exporting.src.js'></script>
```

2. Add `exporting.enabled` and `exporting.url` to our chart configuration as shown in the following code:

```
var options = {
    /* … */
    exporting: {
        enabled: true,
        url: 'http://localhost:3003/'
    }
};
```

3. Render our chart as shown in the following code:

```
$('#container').highcharts(options);
```

How it works...

PhantomJS is a headless JavaScript API. Basically, this means that it can execute JavaScript files without having a browser, or even a display (such as a monitor). When `highcharts-convert.js` is executed with PhantomJS, we start a web server; by including `exporting.js` and configuring the `exporting` object, Highcharts knows to make requests to our web server, and our web server responds to these requests.

Had we not included `exporting.url` and pointed it at our server, Highcharts would have defaulted to using its CDN (`http://export.highcharts.com`). If the information we are exporting is not sensitive or if the user is connected to the Internet, we could have just omitted the parameter.

If, for whatever reason, we wanted to use different versions of the files listed in the recipe, for example, a non-minified version of jQuery, we can make changes to `config` in `highcharts-convert.js`, as in the following code:

```
config = {
    /* define locations of mandatory javascript files */
    /* Note: all files / paths are relative to the exporting-
      server/phantomjs directory */
  HIGHCHARTS: 'highstock.js',
  HIGHCHARTS_MORE: 'highcharts-more.js', /
  HIGHCHARTS_DATA: 'data.js',
  JQUERY: 'jquery.1.9.1.js',
  TIMEOUT: 2000 /* 2 seconds timeout for loading images */
},
```

Exporting images to different formats

There may be occasions where we don't want an entire server running to render a few charts. In this case, we can opt for a slightly simpler solution and render to specific image formats at the same time.

Getting ready

To set up and run PhantomJS, refer to the *Getting ready* section of the *Rendering charts on the server side* recipe given earlier in this chapter.

How to do it...

To get started, perform the following steps:

1. Create a file called `options.json` in Highcharts/exporting-server/phantomjs, and include all of our chart options there. Note that this file must be JSON data (that is, no Javascript). The following is an example:

   ```
   {
       "chart": {"type": "bar"},
       "title": {"text": "Creating your first chart"},
       "series": [{
           "name": "Bar #1",
           "data": [1, 2, 3, 4]
       }]
   }
   ```

2. Change directories to `Highcharts/exporting-server/phantomjs`.
3. Run the following command to generate a chart from `options.json`:

   ```
   phantomjs highcharts-convert.js -infile options.json -
     outfile chart.png
   ```

How it works...

The `highcharts-convert.js` script can run as either a server or as a single command as we've seen. In this scenario, it will take JSON (as in our charts) and render a chart as output.

Generally, the outputted chart will match whatever configuration we pass via `options.json`, but we can change a few properties, such as the image format or the size of the image, using different command-line flags. The following are examples of how we can change properties:

- By changing the file extension in our command, we can change the file format. The script supports four image formats: PNG, JPEG, PDF, and SVG as shown in the following code:

```
phantomjs highcharts-convert.js -infile options.json -
  outfile chart.pdf
```

- We can alter the size of the image by using the `-scale` argument, which will scale the image by the factor that is provided. Images can scale to at most four times their original size as shown in the following command:

```
phantomjs highcharts-convert.js -infile options.json -scale
  2 -outfile chart.png
```

- We can also define a fixed width (in pixels) for the image (which will override the scaling) using the `-width` argument as shown in the following code:

```
phantomjs highcharts-convert.js -infile options.json -width
  400 -
  outfile chart.png
```

E-mailing static charts

When we want to share information, sometimes the fastest way to do so is to just send an e-mail. In this recipe, we'll cover how we can generate a static chart and then e-mail it to someone programmatically.

Getting ready...

1. Install PhantomJS following the instructions found on the PhantomJS website at `http://phantomjs.org`.
2. Install Python 2.7 following the instructions found on the Python website at `http://www.python.org/getit/`.

> It is possible to accomplish this recipe using almost any server side language. The steps as outlined and explained are still relevant, but the details will vary from language to language.

How to do it...

To get started, perform the following steps:

1. Create a new Python file `email_static.py`.
2. Import the different Python modules we'll be using as shown in the following code:

```
import smtplib, os
from email.MIMEMultipart import MIMEMultipart
from email.MIMEBase import MIMEBase
from email.MIMEText import MIMEText
from email.Utils import COMMASPACE, formatdate
from email import Encoders
from subprocess import call
```

3. Define our variables as shown in the following code:

```
send_to = ['whoever@account.com']
send_from = 'your_address@account.com';
subject = 'Test email'
text = 'This is a test email for sending email attachments'
outfile = 'chart.png'
```

4. Render our chart using the following code:

```
call(['phantomjs', 'highcharts-convert.js', '-infile',
  'options.json', '-outfile', outfile])
```

5. Define our message and its fields as shown in the following code:

```
msg = MIMEMultipart()
msg['From'] = send_from
msg['To'] = COMMASPACE.join(send_to)
msg['Date'] = formatdate(localtime=True)
msg['Subject'] = subject
msg.attach( MIMEText(text) )
```

6. Convert the rendered file into the proper encoding and attach it as shown in the following code:

```
part = MIMEBase('application', "octet-stream")
part.set_payload( open(outfile,"rb").read() )
Encoders.encode_base64(part)
part.add_header('Content-Disposition', 'attachment;
  filename="%s"'
  % os.path.basename('chart.png'))
msg.attach(part)
```

7. Send the message using Gmail as shown in the following code:

```
smtp = smtplib.SMTP('smtp.gmail.com:587')
smtp.starttls()
smtp.login('gmail_username@gmail.com','your password')
smtp.sendmail(send_from, send_to, msg.as_string())
smtp.quit()
```

8. Run `email_static.py` as shown in the following code:

```
python email_static.py
```

How it works...

What we've done is very similar to our *Exporting images to different formats* recipe, with some changes.

First, we render the chart, as we had done previously using Python's `subprocess` module to call `phantomjs` outside the function. Then, we start building up our e-mail's basic fields (for example, subject, to, from), and then encode and include that image in the e-mail. Finally, we send the e-mail.

There's more...

If we do not want to use Gmail, we can instead leverage any SMTP server by making the following changes in the code:

```
smtp = smtplib.SMTP('IP Address')
smtp.sendmail(send_from, send_to, msg.as_string())
smtp.close()
```

E-mailing dynamic charts

While sending a static chart is nice to give someone an idea about some dataset, they may want to explore the data themselves. In this case, we'll need to design our charts such that we can share them.

> The following technique will only work for non-local websites, that is, websites that are reachable either on a local network or over the Internet.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe from *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Create the following `getParams` function as shown in the following code:

```
var getParams = function() {
    var searchParams = window.location.search.substr(1),
        paramPairs = searchParams.split('&'),
        params = {},
        i, temp;

    for (i=0; i < paramPairs.length; i++) {
        if (paramPairs[i].trim() === "") {
            continue;
        }
        temp = paramPairs[i].split('=');
        params[temp[0]] = JSON.parse(temp[1]);
    }

    return params;
};
```

2. Call our `getParams` function using the following code:

```
var params = getParams();
```

3. Define our chart options as shown in the following code:

```
var otherData = [/* Assume this is defined */];
var options = {
    /* … */
    series: [{
        // options that are in our GET params
        data: params.data || otherData
    }]
};
```

4. Render our chart as shown in the following code:

```
var chart =
  $('#container').highcharts(options).highcharts();
```

5. Create the following `chartLink` function:

```
var chartLink = function(chart) {
    var config = chart.options,
        url = window.location.origin +
          window.location.pathname + '?',
        key, value;

    // determine which configurable options to be used
    key = 'data';
    value = config.series[0].data;
    url += key + '=' + JSON.stringify(value);

    return url;
};
```

6. Define a button and text field on the page to call our `chartLink` function as shown in the following code:

```
<input type='button' value='Generate Link' id='generate' />
<input type='text' id='link'/>
```

7. Attach an event handler to our button to generate the link as shown in the following code:

```
$('#generate').click(function(){
    $('#link').val(chartLink(chart));
});
```

> Depending on how Highcharts is integrated with your project, this technique may interfere with other aspects of the web page, given that the technique leverages the GET parameters.

How it works...

As implemented previously, our `chartLink` function is a bit incomplete, but enough is present to get the gist of things. Our `getParams` function will look for any search parameters (for example, anything after the ? in a URL) in the URL, turn the keys into dictionary values, and convert the JSON-encoded values into the relevant JavaScript objects.

Our `chartLink` function will return a URL to the current page with relevant parameters converted to JSON.

When the page loads, our chart will look for GET parameters; if it finds them, it will use them in the chart. When we send the chart, the parameters will be in the URL, so they will be used on the page that renders.

There's more...

The technique we've used previously isn't limited to Highcharts; we could also use those encoded parameters in other aspects of a JavaScript application.

We can encode whichever parameter we want, provided that we change our `chartLink` function. For example, if we want to copy all the chart parameters, we could perform the following code:

```
var chartLink = function(chart) {
    var config = chart.options,
        url = window.location.origin + window.location.pathname +
          '?',
        key, value;

    // determine which configurable options to be used
    for(key in config) {
        if (config.hasOwnProperty(key)){
            value = config[key];
            url += key + '=' + JSON.stringify(value);
        }
    }

    return url;
};
```

> We could do something similar to what we did in the *E-mailing static charts* recipe, and render the entire HTML page. However, this process is both more involved and more error-prone, so it is left as an exercise for the reader.

Preparing charts for printing

Although we tend to do things entirely digitally, there may be occasions where we will want to print a chart, be it for a report or some other purpose. In cases such as these, we need to make some small adjustments to make our charts look good for printing.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe from *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Include `exporting.js` on our page as shown in the following code:

```
<!-- Include the verbose version of Highcharts extras -->
<script src='./bower_components/highcharts/highcharts-
  more.src.js'></script>

<!-- Include the exporting module -->
<script
  src='./bower_components/highcharts/modules/
  exporting.src.js'></script>
```

2. Create a button as shown in the following code:

```
<div id='container'></div>
<input type='button' id='print-all' value='Print All' />
```

3. Create the following `printCharts` function and include it on your page as shown in the following code:

```
var printCharts = function (charts) {
    var origDisplay = [],
        origParent = [],
        body = document.body,
        childNodes = body.childNodes,
        ELEMENT = 1;

    // (1) default to all charts
    charts = charts || Highcharts.charts;

    // (2) hide all body content
    Highcharts.each(childNodes, function (node, i) {
        if (node.nodeType === ELEMENT) {
            origDisplay[i] = node.style.display;
            node.style.display = "none";
        }
    });

    // (3) put the charts back in
    $.each(charts, function (i, chart) {
        origParent[i] = chart.container.parentNode;
        body.appendChild(chart.container);
    });

    // (4) print
```

```
        window.print();

        // (5) allow the browser to prepare before reverting
        setTimeout(function () {
            // (6) put the charts back in
            $.each(charts, function (i, chart) {
                origParent[i].appendChild(chart.container);
            });

            // (7) restore all body content
            Highcharts.each(childNodes, function (node, i) {
                if (node.nodeType === 1) {
                    node.style.display = origDisplay[i];
                }
            });
        }, 500);
    }
```

4. Attach our `printCharts` function to the button with an event handler as shown in the following code:

```
$('#print-all').click(function(event) {
    printCharts();
});
```

How it works...

By default, Highcharts will only print a single chart when we click on **Print chart** even if multiple charts are present on the screen. We've wired our button to the `printCharts` function, which allows us to print an arbitrary number of charts. The way that it does this is very similar to `Chart.print`. To print charts, we perform the following steps:

1. Determine which charts we want to print. If none are provided, we default to using all charts available.
2. Find all HTML elements that are part of the document and hide them by setting `display: none` for the element's style. Keep track of the element's original style.
3. Iterate over the charts we want to print, keep track of their original parent element, then insert them into the document again.
4. Tell the browser to print.
5. When we've finished printing, allow a brief pause before we start putting things back to the way they were.
6. Put the charts back underneath their old parent elements.
7. Set the `display` property back to whatever it was before we set it to `display: none`.

There's more...

If we wanted this behavior to be the default and override the existing **Print chart** button, we could accomplish that as follows:

```
// Get a reference to the default menu items
var defaultOptions = Highcharts.getOptions(),
    buttons =
      defaultOptions.exporting.buttons.contextButton.menuItems;

// replace the 'Print Chart button'
buttons[0].onclick = function () {
    printCharts();
} ;

// In our chart options, use our new menu items
var options = {
    /* … */
    exporting: {
        buttons: {
            contextButton: {
                menuItems: buttons
            }
        }
    }
};
```

One other thing that we can do to prepare charts for printing is to change the colors used to make them more print-friendly. These changes are left as an exercise for the reader.

For best results, save the chart as a PDF, and be sure to compare the printed version to what is seen on the screen.

5
Integrating with ExtJS

In this chapter, we will cover the following recipes:

- Setting up a simple ExtJS project
- Using Highcharts in ExtJS
- Connecting your chart using `Ext.data.Store`
- Observing live data using other Store types
- Connecting your chart to `Ext.app.Controller`
- Creating charts that inherit from other charts

Introduction

ExtJS is a well-documented JavaScript framework for rich desktop-like applications. It provides a lot of tools for common development patterns right out of the box, and also has a lot of reusable, pre-built components. In this chapter, we'll take a brief look at some of the components in an ExtJS application and how we can integrate Highcharts into that application.

> All examples in this chapter will be written from the perspective of using ExtJS 4.2.1. While the same advice should apply to later versions, or possibly other 4.x Versions, please keep this in mind.

Setting up a simple ExtJS project

In order to get started, we'll need to first set up a basic ExtJS project, and then get a bit of an understanding of how ExtJS projects are laid out.

Getting ready

Before we begin, we will need to download the ExtJS framework and set up our project.

1. Visit the Sencha website to download ExtJS from `http://www.sencha.com/products/extjs/`. Note that both free and commercial versions of the software are available, so we need to pick an appropriate version for our project.
2. Create a folder named `my_project` that will contain our project.
3. Unzip ExtJS to `my_project/extjs`.
4. Create a file named `my_project/app.js` that will define our application. Include the following code in it:

   ```
   // Added so that Viewport code is always loaded before application
   runs
   Ext.require('Ext.container.Viewport');
   ```

5. Create a file named `my_project/index.html` and include the following code within it:

   ```
   <html>
     <head>
       <title>Chapter 5 – Integrating with ExtJS</title>
       <!-- Include default stylesheets -->
       <link rel="stylesheet" type="text/css" href="extjs/resources/
       css/ext-all.css">

       <!-- Include the core JS in a developer friendly format -->
       <script type="text/javascript" src="extjs/ext-dev.js"></script>

       <!-- Include our application -->
       <script type="text/javascript" src="app.js"></script>

     </head>

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

We will also need a way of running this project from a server. This is beyond the scope of this chapter, but we may be able to re-use the techniques used to set up a Bottle server, which is discussed in the *Using Ajax for polling charts* recipe in *Chapter 2*, *Processing Data*.

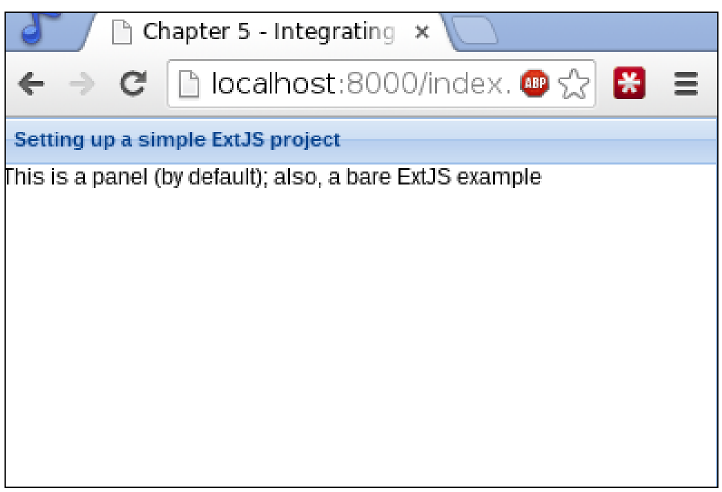
How to do it...

To get started, perform the following steps:

1. Within `my_project/app.js`, include the following code:

```
// Added so that Viewport code is always loaded before application
runs
Ext.require('Ext.container.Viewport');

Ext.application({
  name: 'Chapter5',
  launch: function() {
    Ext.create('Ext.container.Viewport', {
      layout: 'fit',
      items: [{
        title: 'Setting up a simple ExtJS project',
        html: 'This is a panel (by default); also, a bare ExtJS
        example'
      }]
    });
  }
});
```

2. Visit the `index.html` page in the browser:



How it works...

If you've noticed, our `index.html` page, unlike usual, doesn't really do all that much: It links some ExtJS stylesheets to the developer version of the core libraries (`ext-dev.js`) and to our application (`app.js`).

All of the detail of our project comes from `app.js` where we create our main `Application` object. We can include any component that we'll need by using `Ext.require`, which we've done for our viewport (that is, an object representing the browser). We'll also notice that unlike other JavaScript code, where we might follow some pattern such as `var obj = new MyClass()`, here we use `Ext.create`. ExtJS handles all object creation, inheritance, and so on for us, so we can focus on writing application-level code in ExtJS.

Our example is fairly straightforward: when the application launches, we want to create a viewport that will fill the screen (`layout: 'fit'`) and put some items in our viewport. We'll notice this pattern when creating other nested items (`Ext.create('Class', {items: [/*list of child items*/]}`). Normally, we'd specify an `xtype`, which is a short name for a particular class; but if it is omitted, the parent will usually render the item as a panel, which is a simple container object.

There's more...

ExtJS is a very feature-rich library, so it is definitely worth reading up on its well-documented API at `http://docs.sencha.com/extjs/`. Rather than setting up a project manually, Sencha provides a tool called Sencha Cmd (`http://www.sencha.com/products/sencha-cmd/`), which not only makes it easy to scaffold applications, as we did in our example, but also makes it easy to handle other steps in the process of building an ExtJS application.

Using Highcharts in ExtJS

Now that we have a basic understanding of how to set up an ExtJS project, we are able to get started and create our first chart in ExtJS. Much of what we have already learned can be applied, but there are differences worth noting.

Getting ready

We will need to download the Highcharts extension for ExtJS.

1. Create an account on the Sencha network at `https://id.sencha.com/users/sign_up` because this is necessary to access the market.
2. Download the Highcharts extension (Version 2.3) found in the Sencha market (`http://market.sencha.com/extensions/highcharts/`).

3. Unzip the Highcharts extension so that our project looks like the following hierarchy:

```
my_project/
  app.js
  index.html
  extjs/
  resources/
    Chart/
      ux/
```

4. Highcharts and jQuery are not included in the Highcharts extension, so download these files, if necessary, and copy them to `my_project/resources` as follows:

```
my_project/
  app.js
  index.html
  extjs/
  resources/
    highcharts.src.js
    highcharts-more.src.js
    jquery.dev.js
    Chart/
```

5. Update `my_project/index.html` to include Highcharts and jQuery, as shown in the following code:

```
<!-- Include the core JS in a developer friendly format -->
<script type="text/javascript" src="extjs/ext-dev.js"></script>

<!-- Include Highcharts, jQuery; Not provided by Highcharts
extension -->
<script type="text/javascript" src="resources/jquery.dev.js"></
script>
<script type="text/javascript" src="resources/highcharts.src.
js"></script>
<script type="text/javascript" src="resources/highcharts-more.src.
js"></script>

<!-- Include our application -->
<script type="text/javascript" src="app.js"></script>
```

6. Ensure that the Highcharts extension is loaded in our application, as shown in the following code:

```
// Added so that Viewport code is always loaded before application
runs
Ext.require('Ext.container.Viewport');

// Needs to know where Highcharts class is located
Ext.Loader.setPath('Chart', '../resources/Chart');

// Require Highcharts Extension
Ext.require('Chart.ux.Highcharts');

// Require whichever series we will be using
Ext.require('Chart.ux.Highcharts.Serie');
Ext.require('Chart.ux.Highcharts.LineSerie');
```

How to do it...

To get started, perform the following steps:

1. Create a new application in `app.js`, as shown in the following code:

```
Ext.require(/* … */);

Ext.application({
  name: 'MyApp',
  launch: function() {
    var chart = Ext.create('Chart.ux.Highcharts', { });
  }
});
```

2. Add configuration to our chart, as shown in the following code:

```
launch: function() {
  var chart = Ext.create('Chart.ux.Highcharts', {
    region: 'center',
    id: 'chart',
    initAnimAfterLoad: false,
    chartConfig: {
      chart: {
        type: 'line',
        showAxes: true
      },
      title: {
        text: 'A simple graph'
      }
    }
  });
}
```

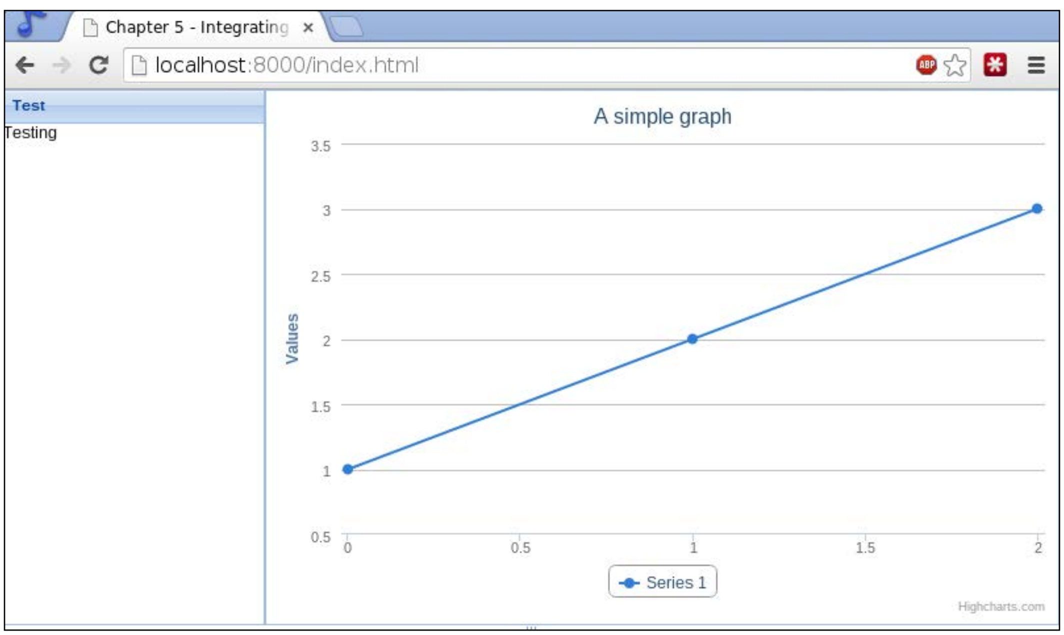
3. Create `Viewport`, as shown in the following code:

```
launch: function() {
  var chart = Ext.create('Chart.ux.Highcharts', { /*...*/});

  var viewport = Ext.create('Ext.container.Viewport', {
    layout: 'border',
    items: [{
      region: 'west',
      width: 200,
      title: 'Test',
      html: 'Testing'
    }, chart]
  });
}
```

4. Add data to the chart, as shown in the following code:

```
launch: function() {
  var chart = Ext.create('Chart.ux.Highcharts', { /*...*/});
  var viewport = Ext.create('Ext.container.Viewport', {/*...*/});
  chart.addSeries([{
    name: 'Series #1',
    data: [1,2,3]
  }], false);
}
```

5. Visit the `index.html` page in the browser:



How it works...

ExtJS loads files dynamically as they are needed, so we must first tell ExtJS where it can find the Highcharts extension by using `Ext.Loader.setPath`. With `Ext.Loader.setPath`, we tell ExtJS what class we want to find (for example, `Chart`) and where it can be found relative to `app.js` (in this case, it is present in `../resources/Chart`). Then, as is the case with any module, we need to tell ExtJS that we require that module by using `Ext.require`.

Configuration works differently when using the extensions as compared to how they would be defined in JavaScript. First, most of the chart configuration is stored inside a `chartConfig` element. Second, all data that we would normally define using `series[x].data` has been removed from `config` and instead been added by using `chart.addSeries(series, append)`. This method allows us to add one or more charts in a way similar to how we would normally define our series.

We need to include a few options to render the chart without initial data. We need to set `initAnimAfterLoad` to `false` to prevent the chart from animating after loading, and `chartConfig.chart.showAxes` to `true` so that the axes will render the chart initially.

> In the case of the Highcharts extension, if you do not include the dependencies for the chart's series types that are used, the graph will not load. For example, if `chartConfig.chart.type` is `area`, `Chart.ux.Highcharts.AreaSerie` must be included by using `Ext.require`. For more information on the names and series types available, please visit the Highcharts extension documentation at `http://joekuan.org/demos/Highcharts_Sencha/docs/`.

There's more...

Because of the differences between ExtJS and plain JavaScript, the Highcharts extension for ExtJS makes some changes to how configuration is defined and how certain actions take place. For this reason, it's worthwhile to take a look at the online documentation for the extension (`http://joekuan.org/demos/Highcharts_Sencha/docs/`) and the source code (`https://github.com/JoeKuan/Highcharts_Sencha`) for more details about the differences.

Connecting your chart using Ext.data.Store

When working with Highcharts, most of our work is focused on defining the configuration and the data. In ExtJS, this work is abstracted into different components. In particular, data is managed using a store. This recipe examines how we can leverage the store to add data to our charts.

Getting ready

For setting up a basic ExtJS project with Highcharts, refer to the *Getting ready* section of the *Using Highcharts in ExtJS* recipe discussed earlier in this chapter.

How to do it...

To get started, perform the following steps:

1. Define the application as follows:

```
Ext.require('Ext.container.Viewport');

Ext.Loader.setPath('Chart', '../resources/Chart');
Ext.require('Chart.ux.Highcharts');
Ext.require('Chart.ux.Highcharts.Serie');
Ext.require('Chart.ux.Highcharts.LineSerie');

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

2. Create `ArrayStore` for your data as follows:

```
launch: function() {
  data = [
    ['September', 42],
    ['October', 31.4],
    ['November', 23.18]
  ];
```

```
    var store = Ext.create('Ext.data.ArrayStore', {
      fields: [
        {name: 'month', type: 'string'},
        {name: 'value', type: 'float'}
      ],
      data: data
    });
  }
```

3. Create the chart as follows:

```
launch: function() {
  var store = Ext.create('Ext.data.ArrayStore', { /*...*/});
  var chart = Ext.create('Chart.ux.Highcharts', {
    region: 'center',
    id: 'chart',
    store: store,
    series: [{
      dataIndex: 'value'
    }],
    xField: 'month',
    chartConfig: {
      chart: {
        type: 'line',
      },
      title: {
        text: 'A simple graph'
      }
    }
  });
}
```

4. Create a viewport as follows:

```
launch: function() {
  var store = Ext.create('Ext.data.ArrayStore', { /*...*/});
  var chart = Ext.create('Chart.ux.Highcharts', { /*...*/});
  var viewport = Ext.create('Ext.container.Viewport', {
    layout: 'border',
    items: [chart]
  });
}
```

5. Visit the `index.html` page in the browser.

How it works...

The biggest difference between this recipe and how we've created our charts previously is the introduction of `ArrayStore`. An `ArrayStore` is a `Store`: a class that encapsulates a client-side collection of models, which are just individual data points. Using an `ArrayStore` means that the data is expected to be in an array format, even though other stores exist for other data formats.

The way in which it works is that we tell our Store what the fields of each data point are named (for example, month), what format the data will be in (for example, string), and we then provide the Store with the data as an array of arrays that follows the format that we laid out, namely, `[[month, value], ...]`.

We also add a few new configuration options to our chart apart from `store`. We use `xField` to determine which field in the Store should be used for the x axis labels, and `series` lets us list which series will be included in the chart and what will be used for the y axis values by setting `dataIndex`.

Observing live data using other Store types

ExtJS supports a number of different Store types, but stores are only a part of the data management puzzle. The other piece is proxies, which are used to access the data from different sources. This recipe will look into how we can use certain stores, or stores and proxies, to get live data from a server.

Getting ready

For setting up a basic ExtJS project with Highcharts, refer to the *Getting ready* section of the *Using Highcharts in ExtJS* recipe discussed earlier in this chapter.

For this recipe, we will assume that the data we get back from our source looks like the following code snippet:

```
[
  {"y": 132, "color": "#98D8AF"},
  {"y": 254, "color": "#A80877"},
  {"y": 119, "color": "#569A3C"}
  /* … */
]
```

How to do it...

To get started, perform the following steps:

1. Define the application as follows:

```
Ext.require('Ext.container.Viewport');

Ext.Loader.setPath('Chart', '../resources/Chart');
Ext.require('Chart.ux.Highcharts');
Ext.require('Chart.ux.Highcharts.Serie');
Ext.require('Chart.ux.Highcharts.LineSerie');

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

2. Create a JsonStore for our data as follows:

```
launch: function() {
  var store = Ext.create('Ext.data.JsonStore', {
    fields: [
      {name: 'color', type: 'string'},
      {name: 'y', type: 'float'}
    ],
    proxy: {
      type: 'ajax',
      url: '/ajax/series'
    },
    autoLoad: true
  });
}
```

3. Create the chart as follows:

```
launch: function() {
  var store = Ext.create('Ext.data.ArrayStore', { /*...*/});
  var chart = Ext.create('Chart.ux.Highcharts', {
    region: 'center',
    id: 'chart',
    store: store,
    series: [{
      dataIndex: 'y'
    }],
    chartConfig: {
      chart: {
        type: 'line',
      },
      title: {
```

```
          text: 'A simple graph'
        }
      }
    });
  }
```

4. Create a viewport as shown in the following code:

```
launch: function() {
  var store = Ext.create('Ext.data.ArrayStore', { /*...*/});
  var chart = Ext.create('Chart.ux.Highcharts', { /*...*/});
  var viewport = Ext.create('Ext.container.Viewport', {
    layout: 'border',
    items: [chart]
  });
}
```

5. Visit the `index.html` page in the browser.

How it works...

In this recipe, we've used a proxy, in particular `AjaxProxy`. Proxies make it easy to create, read, update, and delete resources. All we need to do is provide our store with a proxy definition with a type and relevant details (`url` in this case), and the store will be able to fetch whatever results it needs. The `autoLoad` callback just determines whether we should load data immediately or not.

There's more...

If we want to create a polling example where the chart updates periodically (for example, once in a second), we just need to periodically call `store.reload()`:

```
launch: function {
  var store = Ext.create('Ext.data.ArrayStore', { /*...*/});
  var chart = Ext.create('Chart.ux.Highcharts', { /*...*/});
  var viewport = Ext.create('Ext.container.Viewport', { /*...*/});
  setInterval(function() {
    store.reload();
  }, 1000);
}
```

Also, other proxies exist for RESTful services (RestProxy, `http://docs.sencha.com/extjs/4.2.1/#!/api/Ext.data.proxy.Rest`) and for JSONP (JsonPProxy, `http://docs.sencha.com/extjs/4.2.1/#!/api/Ext.data.proxy.JsonP`).

Connecting your chart to Ext.app.Controller

Being able to create a chart can be useful, but it is even better to be able to control the chart; for example, being able to take certain actions and listen for events on the chart. That is where `Ext.app.Controller` comes in, and this is what we'll be investigating in this recipe.

Getting ready

For setting up a basic ExtJS project with Highcharts, refer to the *Getting ready* section of the *Using Highcharts in ExtJS* recipe discussed earlier in this chapter.

How to do it...

To get started, perform the following steps:

1. Set up the folder structure. We will also need to create an empty `GenericController.js` file as follows:

```
my_project/
  app.js
  index.html
  extjs/
  resources/
  app/
    controller/
      GenericController.js
```

2. Define the controller in `GenericController.js` as follows:

```
Ext.define('MyApp.controller.GenericController', {
  extend: 'Ext.app.Controller',

  init: function() {
    console.log('Initialized GenericController!');
    this.control({
      '#refresh': {
        click: function() {
          var chart = Ext.getCmp('chart');
          chart.store.reload();
        }
      }
    });
  }
});
```

3. Define the application in `my_project/app.js` as follows:

```
Ext.require('Ext.container.Viewport');
Ext.Loader.setPath('Chart', '../resources/Chart');
Ext.require('Chart.ux.Highcharts');
Ext.require('Chart.ux.Highcharts.Serie');
Ext.require('Chart.ux.Highcharts.LineSerie');

Ext.application({
  name: 'MyApp',
  appFolder: 'app',
  launch: function() { }
});
```

4. Create a reference for our controller in our application as follows:

```
Ext.application({
  name: 'MyApp',
  appFolder: 'app',
  controllers: ['GenericController'],
  launch: function() { }
});
```

5. Create a `JsonStore` for our data:

```
launch: function() {
  var store = Ext.create('Ext.data.JsonStore', {
    fields: [
      {name: 'color', type: 'string'},
      {name: 'y', type: 'float'}
    ],
    proxy: {
      type: 'ajax',
      url: '/ajax/series'
    },
    autoLoad: true
  });
}
```

6. Create the chart as shown in the following code:

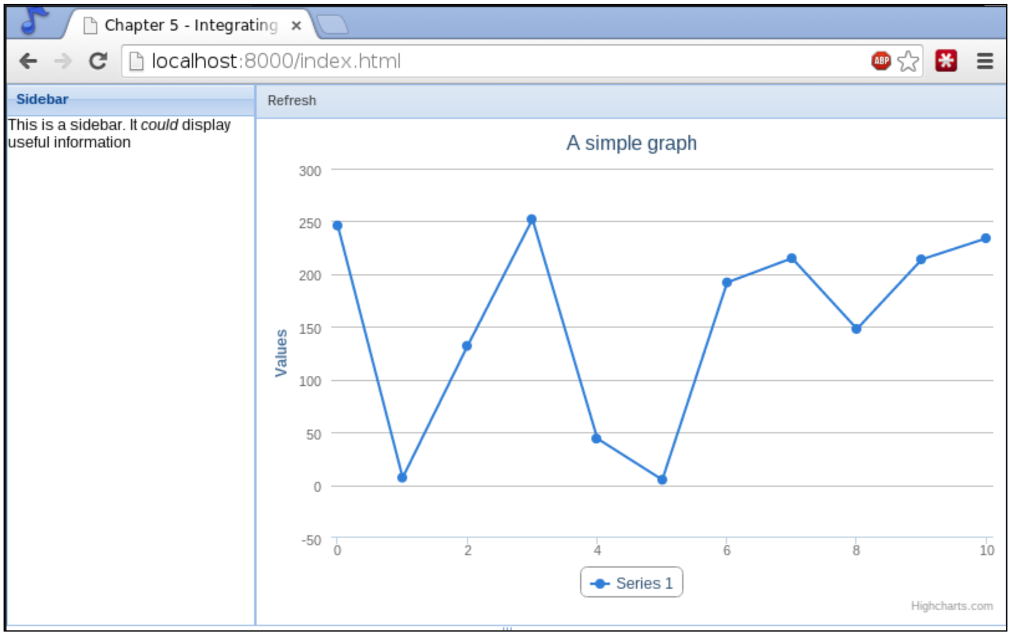
```
launch: function() {
  var store = Ext.create('Ext.data.ArrayStore', { /*...*/});
  var chart = Ext.create('Chart.ux.Highcharts', {
    region: 'center',
    id: 'chart',
    store: store,
```

```
        series: [{
          dataIndex: 'y'
        }],
        chartConfig: {
          chart: {
            type: 'line',
          },
          title: {
            text: 'A simple graph'
          }
        }
      });
    }
```

7. Create a viewport as shown in the following code:

```
launch: function() {
  var store = Ext.create('Ext.data.ArrayStore', { /*...*/});
  var chart = Ext.create('Chart.ux.Highcharts', { /*...*/});
  var viewport = Ext.create('Ext.container.Viewport', {
    layout: 'border',
    items: [{
      region: 'west',
      width: 200,
      title: 'Sidebar',
      html: 'This is a sidebar. It <i>could</i> display useful
      information'
    }, {
      region: 'center',
      tbar: [{
        xtype: 'button',
        text: 'Refresh',
        id: 'refresh',
        height: 50
      }],
      items: [chart]
    }]
  });
}
```

8. Visit the `index.html` page in the browser:



How it works...

A controller is just a means of handling and redirecting events, such as an element being clicked or rendered. If we set up everything correctly, then ExtJS can load our controller automatically. More specifically, it will load our controller if the following conditions are satisfied:

- We have set `appFolder` to the folder containing our other classes
- We have created the appropriate folders in that folder (that is, `controller`)
- The controller's name in our controllers' arrays matches the name in the JavaScript file

Apart from our viewport, which now has a bar at the top (`tbar`) with a **Refresh** button on it, most of the interesting work takes place inside our `GenericController`. The most important part of our controller is the `init` method where we can set up our event handlers. With the help of `this.control`, we can pass a set of key-value pairs, where the keys are component selectors (very similar to DOM selectors, but for ExtJS components), and the values are a map of events to handler functions.

We'll also notice that we use an `Ext.getCmp` function, which is an abbreviation for **get component**; it will get an ExtJS component from anywhere on the page by the component's ID. Once we have a reference to the chart, we can access its store and refresh the data at the click of a button.

There's more...

ExtJS component selectors are similar to DOM selectors, but have some interesting differences and improvements from basic DOM selectors. More information on the selectors can be found in the Ext.ComponentQuery documentation (`http://docs.sencha.com/extjs/#!/api/Ext.ComponentQuery`).

For more information on how to structure an application in ExtJS and the general architectural details of ExtJS, it is worth reviewing their MVC architecture guide (`http://docs.sencha.com/extjs/#/guide/application_architecture`).

Creating charts that inherit from other charts

Creating the same charts over and over again can become tiresome, especially if the only elements that change are the data or a few small details. In this recipe, we'll uncover how we can define charts in a way that we can extend and change to make new charts.

Getting ready

For setting up a basic ExtJS project with Highcharts, refer to the *Getting ready* section of the *Using Highcharts in ExtJS* recipe discussed earlier in this chapter.

How to do it...

To get started, perform the following steps:

1. Create a new `my_project/Custom` folder, and create a `CustomChart.js` file within it.
2. Within `CustomChart.js`, define our class using the following code:
   ```
   Ext.define('Custom.SplineChart', {
   });
   ```
3. Our class should extend `Chart.ux.Highcharts`, as shown in the following code:
   ```
   Ext.define('Custom.SplineChart', {
     extend: 'Chart.ux.Highcharts'
   });
   ```

4. Define any option we want for our new chart. Also, include anything we would require:

```
Ext.require('Chart.ux.Highcharts.SplineSeries');
Ext.define('Custom.SplineChart', {
  extend: 'Chart.ux.Highcharts',
  constructor:function(config) {
    this.callParent(arguments);
    this.initConfig(config);

    this.chartConfig = this.chartConfig || {};
    this.chartConfig.chart = this.chartConfig.chart || {};
    this.chartConfig.chart.type = 'spline';
  }
});
```

5. Use our new chart in an application, as shown in the following code:

```
Ext.Loader.setPath('Chart', '../resources/Chart);
Ext.Loader.setPath('Custom', '../Custom');

Ext.require('Chart.ux.Highcharts');
Ext.require('Custom.SplineChart');
```

How it works...

The `Ext.define` function handles the task of selecting the class we want to extend (`Chart.ux.Highcharts`) and extending it to create a new class (`Custom.SplineChart`). We can also use this technique to extend or replace any other methods or properties of the base class. We still need to tell ExtJS where to find our new class by using `Ext.Loader.setPath`, and we still require both the class and the Highcharts extension as dependencies for the application by using `Ext.require`.

6

Integrating with jQuery

In this chapter, we will cover the following recipes:

- Creating charts with jQuery
- Using the `data-` attributes to load charts
- Binding events using `jQuery.on`
- Handling user interaction with jQuery
- Updating a chart on the backend
- Using jQuery UI tabs and Highcharts
- Modifying charts using jQuery UI widgets
- Putting charts in pages using jQuery Mobile

Introduction

jQuery is one of the most popular JavaScript libraries in use today, and it's easy to understand why. It abstracts away a lot of the inconsistencies between browsers, and, since its introduction, it has also grown to include loads of other useful functionalities such as animations and plugins. Its existence has also paved the way for similar, more focused projects such as jQuery Mobile (focused on improvements for mobile browsers) and jQuery UI (provides widgets and other functionalities for web pages and web applications). This chapter will focus on how we can integrate Highcharts with some of the varied functionality that jQuery provides.

Creating charts with jQuery

Before we really dig into what we can create, we need to make a basic chart. Fortunately, Highcharts provides a wrapper for jQuery, making chart creation very simple.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Create a set of chart options as shown in the following code:

```
$(document).ready(function() {
  var options = {
    chart: {
      type: 'column'
    },
    title: {
      text: 'Creating Charts with jQuery'
    },
    subtitle: {
      text: 'Look familiar?'
    },
    series: [{
      name: '2^x',
      data: [1,2,4,8]
    }, {
      name: '2^(x+1)',
      data: [2,4,8,16]
    }]
  };
```

2. Render the chart by using `.highcharts()` as shown in the following code:

```
$('#container').highcharts(options);
```

How it works...

By default, Highcharts includes a wrapper that makes it easy to create charts. This wrapper is the `.highcharts()` plugin. This function automatically takes the options provided and uses it to generate a chart within the DOM node with the `container` ID (that is, `#container`). We can optionally provide a second argument that will execute some function on load (similar to `chart.events.load`).

There's more...

We can also obtain a reference to the chart, which will occasionally be needed to act on the chart after it has already been created. This can be done when the chart is being created, as shown in the following code:

```
var chart = $('#container').highcharts(options).highcharts();
```

It can also be done after the chart has been created, as shown in the following code:

```
var chart = $('#container').highcharts();
```

Using the data- attributes to load charts

There may be occasions where we do not have a data source for our charts per se. Perhaps we're only given a pre-rendered HTML code and need to make a chart from that, or the data is stored using `data-` attributes. In such cases, it is still relatively easy to create a chart.

> This recipe assumes that we have a piece of HTML code with the following `data-` attributes. If our HTML is not of this format, we will need to change it accordingly:

```
<div id='container'
  data-chart='{"type": "column"}'
  data-title='{"text": "Using data-attributes to load
  charts"}'
  data-series='[{"name": "2^x", "data": [1,2,4,8]},
{"name": "2^(x+1)", "data": [2,4,8,16]}]'
></div>
```

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Obtain the data from our DOM node with the following code:
   ```
   var options = $('#container').data();
   ```
2. Process the data as necessary as shown in the following code. This step is optional:
   ```
   options.title = 'My new title';
   ```
3. Render the chart with the following code:
   ```
   $('#container').highcharts(options);
   ```

How it works...

jQuery has a function called `.data()` that can be called on any jQuery object. This function will return any data on that element that is set using `.data(key, value)`, or initially set using `data-` attributes. Any `data-` attribute is automatically converted. Keys are converted to camel case and have their `data-` prefix removed (for example, `data-my-attr` becomes `myAttr` in the returned object). In fact, if our data is formatted in a proper JSON format (for example, all keys are enclosed in double quotes), the `.data` function will even extract nested fields, as it did in our example. If we only wanted to extract some of the information from our element, we could have used `.data(key)`, which would have returned data only for that key (for example, `.data('chart')` would return `{"type": "column"}`).

There's more...

We could also accomplish what we've done in this recipe in a shorter form as follows:

```
$('#container').highcharts(
  $.extend($('#container').data(), { title: "My new title" })
);
```

The major difference being that we use `$.extend`, which takes the first object argument and adds or overwrites it with any keys found in the subsequent argument objects.

Binding events using jQuery.on

Static charts are used for various purposes, but in order for our charts to become more dynamic, we need to be able to handle events. Handling events is possible with the use of `jQuery.on`.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Define the chart options with the following code:

   ```
   var options = { /* our chart options */};
   ```

2. Render our chart, maintaining a reference to the chart, as shown in the following code:

   ```
   var chart = $('#container').highcharts(options).highcharts();
   ```

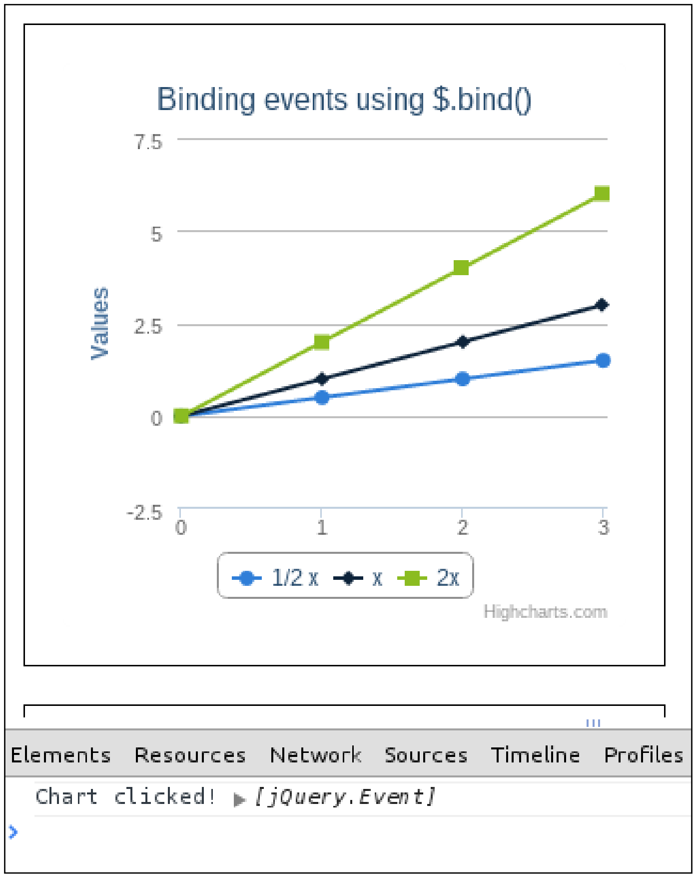
3. Bind a click handler to the chart as shown in the following code:

   ```
   $(chart).on('click', function() {
     console.log('Our chart has been clicked!');
   });
   ```

4. Bind a handler for the show event on the first series in the chart as shown in the following code:

   ```
   $(chart.series[0]).on('show', function() {
     console.log('The first series has been shown in our chart!');
   });
   ```

The resultant chart is displayed as follows:



How it works...

The `.bind()` method is very similar to using the various events' options we would normally put in our chart configuration, except that it works after the chart has already been rendered. We can call `.bind()` on any element in our chart we would normally be able to define events on (for example, the chart object itself, individual series, and points), and we can bind handlers to any event that these objects would normally respond to (for example, the `click` event on a series or a point). The format is straightforward, `.bind()` takes two arguments, a string of events (or multiple events, for example, `show click` to use the same handler for the `show` and `click` events), and an event handler.

Handling user interaction with jQuery

Using `jQuery.on` is a start, it allows us to handle events. However, what if `.on()` is too verbose for us? What about handling events on multiple objects?

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Define the chart options as shown in the following code:

   ```
   var options = { /* our chart options */};
   ```

2. Render our chart, maintaining a reference to the chart, as shown in the following code:

   ```
   var chart = $('#container').highcharts(options).highcharts();
   ```

3. Register a handler as shown in the following code:

   ```
   $(chart).click(function() {
     console.log('Our chart has been clicked!');
   });
   ```

4. Register a handler using `.on()` as shown in the following code:

   ```
   $(chart.series[0].data[0]).on('mouseOver', function() {
     console.log('Mouseover on first point!');
   });
   ```

5. Register a handler on several elements, as shown in the following code:

   ```
   $(chart.series[1].data).each(function(index, point) {
     $(point).click(function() {
       console.log('Clicked on a point!');
     });
   });
   ```

How it works...

jQuery supports a number of shortcuts such as `.click()` to handle events. The difference between this and `.bind()` is that we do not need to supply the event name as it's included in the function (that is, `.click()` handles the `click` events).

The `.bind()` and `.click()` methods are really just aliases of `.on()`, which is how jQuery handles all events. Of course, there may be subtle differences, so it's always best to consult the documentation (`http://api.jquery.com`).

Also in this recipe, we used `$(element).each(eachFn)`, which is a simple helper function that allows us to iterate over the element. The `eachFn` parameter that we provide, takes an index and an element as its first and second parameters respectively. From there, we just add our handler to each element.

Updating a chart on the backend

Until this point, we've mostly been getting data from some source. We never modify or change data at the source. This recipe will briefly go through the details of how we might send data to some server that affects our chart.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*. For our example, assume that the JSON we retrieve looks like the following code:

```
{'y': 0}
```

How to do it...

To get started, perform the following steps:

1. Create a textbox and a button on our page as follows:

```
<div id='container'></div><br/>
<input type='text' id='new_value' />
<input type='submit' id='replace_value' value="Replace value"/>
```

2. Create a handler for the instance when the button is pressed, as shown in the following code:

```
$('#replace_value'').on('click', function(event) {
  var newValue = $('#new_value').val();

  event.preventDefault();
```

```
  $.ajax({
    url: '/ajax/point',
    contentType: 'application/json',
    type: 'POST',
    data: JSON.stringify({y: newValue})
  }) ;
});
```

3. Define the chart options as follows:

```
var options = {
  /* … */
};
```

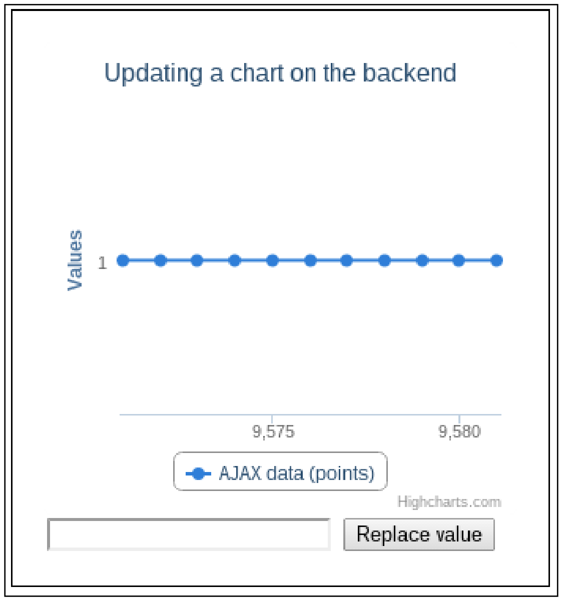
4. Have our chart fetch data periodically using `setInterval` and the `load` event on the chart, as shown in the following code:

```
var options = {
  chart: {
    events: {
      load: function () {
        var self = this;
        setInterval(function() {
          $.getJSON('/ajax/point', function(data) {
            var series = self.series[0];
            var redrawVal = true;
            var shiftVal = false;
            if (series.data && series.data.length > 10) {
              shiftVal = true;
            }
            series.addPoint(data, redrawVal, shiftVal);
          });
        }, 1000);
      }
    }
  }
};
```

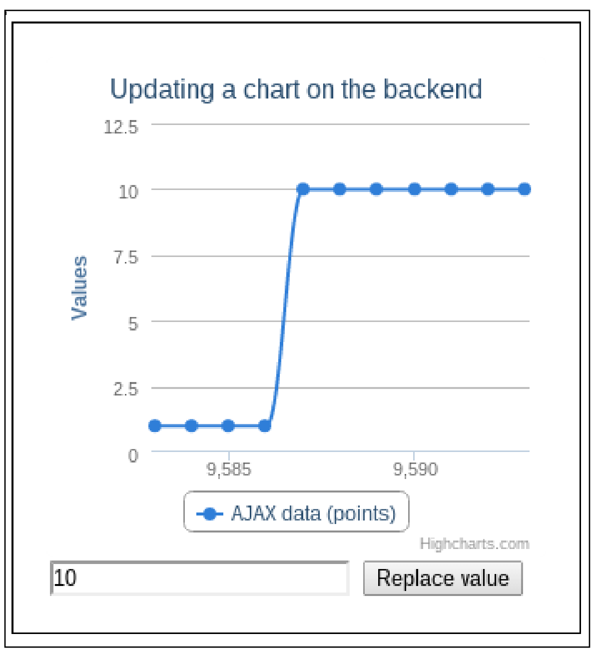
5. Render the chart as follows:

```
$('#container').highcharts(options);
```

The resultant chart is displayed as follows:



6. Change the values as shown in the following screenshot:



How it works...

Similarly, to know how we can obtain data using `$.ajax` or `$.getJSON`, we can also send data using these methods. In our case, since we're sending data, we set the type to `POST`, we set our `contentType` so that the server knows what to expect, and since we're not submitting a form, we need to send over our data as a string, which we do using `JSON.stringify(obj)`. Then, our backend gets the new value, and whenever we poll for new data, our chart will add that new value on to the end (as we did in previous examples).

Using jQuery UI tabs and Highcharts

So far, we've been able to create charts and modify their containers using CSS, or the chart itself using options. One element that we haven't looked into is layout. This recipe will show us how we can put Highcharts in jQuery UI tabs, but these steps will be very similar to the steps for using other jQuery UI elements, such as an accordion.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

We will also need to include CSS and JavaScript for jQuery UI by performing the following steps:

1. Modify `bower.json` to include `jquery-ui` as follows:

```
{
 "name": "highcharts-cookbook-chapter-6",
 "dependencies": {
  "jquery": "^1.9",
  "jquery-ui": "^1.10",
  "highcharts": "~3.0"
 }
}
```

2. Install bower dependencies with the following code:

```
bower install
```

3. Include jQuery UI theming on the page as follows:

```
<title>Chapter 6 - Integrating with jQuery - Examples</title>
<link rel="stylesheet" href="./bower_components/jquery-ui/themes/
ui-lightness/jquery-ui.min.css" />
<link rel="stylesheet" href="./bower_components/jquery-ui/themes/
ui-lightness/jquery.ui.theme.css" />
```

4. Include jQuery UI JavaScript in the page as follows:

```
<!-- Include the verbose version of jQuery -->
<script src='./bower_components/jquery/jquery.js'></script>

<!-- Include jQuery UI -->
<script src="./bower_components/jquery-ui/ui/jquery-ui.js"></
script>
```

How to do it...

To get started, perform the following steps:

1. Define the tab markup as shown in the following code:

```
<div id='tabs'>
  <ul>
    <li><a href='#tab1'>Tab 1</a></li>
    <li><a href='#tab2'>Tab 2</a></li>
  </ul>
  <div id='tab1'></div>
  <div id='tab2'></div>
</div>
```

2. Define the chart options as follows:

```
var options1 = {
  chart: {
    type: 'areaspline'
  },
  title: {
    text: 'Tab 1 Chart'
  },
  series: [{
    name: 'x^2',
    data: [0, 1, 4, 9, 16, 25]
  }, {
    name: '-(x^2) + 30',
    data: [30, 29, 26, 21, 14, 5]
  }]
};
var options2 = {
  chart: {
    type: 'area'
  },
  title: {
    text: 'Tab 2 Chart'
  },
```

```
    series: [{
      name: '1/2 x',
      data: [0, 0.5, 1, 1.5]
    }, {
      name: 'x',
      data: [0, 1, 2, 3]
    }, {
      name: '2x',
      data: [0, 2, 4, 6]
    }]
  };
```

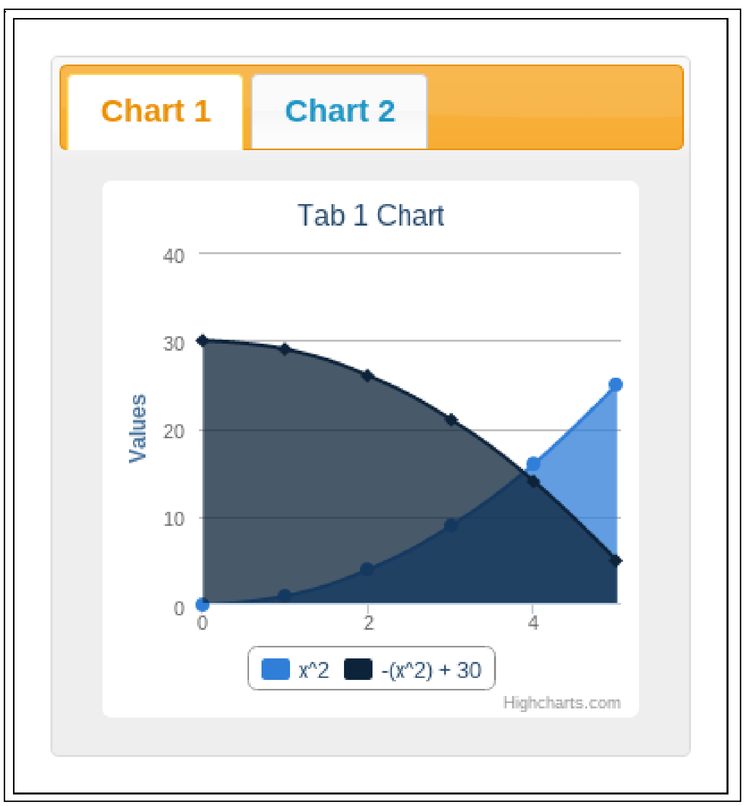
3. Render the charts as shown in the following code:

```
$('#tab1').highcharts(options1);
$('#tab2').highcharts(options2);
```

4. Render the tabs as shown in the following code:

```
$('#tabs').tabs();
```

The resultant chart appears as follows:



How it works...

The `.tabs()` method is part of jQuery UI that takes a given element and converts it into a set of tabs. By default, it expects a `<ul>` or `<ol>` element where it will obtain the tab information from the `<li>` elements present. Each `<li>` element must have an anchor with an `href` attribute, and it will link to whichever ID is specified in the anchor (for example, `<a href='#banana'>Banana</a>` will connect the tab to the element with ID `banana`).

In addition to all this, `.tabs()` will automatically add styles to the tabs and containers making them easy to style, or use some of jQuery UI's existing themes. It is also possible to add configurations to tabs by using `.tabs(config)`. For more details on `.tabs(config)`, visit the jQuery UI documentation (`http://api.jqueryui.com/tabs`).

Modifying charts using jQuery UI widgets

jQuery UI provides a number of different controls that are easy to create and style. This recipe looks at how we can leverage a few of these widgets in our charts.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Using jQuery UI tabs and Highcharts* recipe discussed earlier in this chapter.

How to do it...

To get started, perform the following steps:

1. Create elements for a slider and a button, as follows:

```
<div id='container'></div>
<div id='slider'></div><br/>
<button id='increase'>Increase</button>
<button id='decrease'>Decrease</button>
```

2. Define the chart options as shown in the following code:

```
var options = {
  chart: {
    type: 'column'
  },
  title: {
```

```
    text: 'Modifying charts using jQuery UI Controls'
  },
  series: [{
    name: 'Slider',
    data: [0]
  }, {
    name: 'Button',
    data: [0]
  }]
};
```

3. Render the chart, keeping a reference to it, as shown in the following code:

```
var chart = $('#container').highcharts(options).highcharts();
```

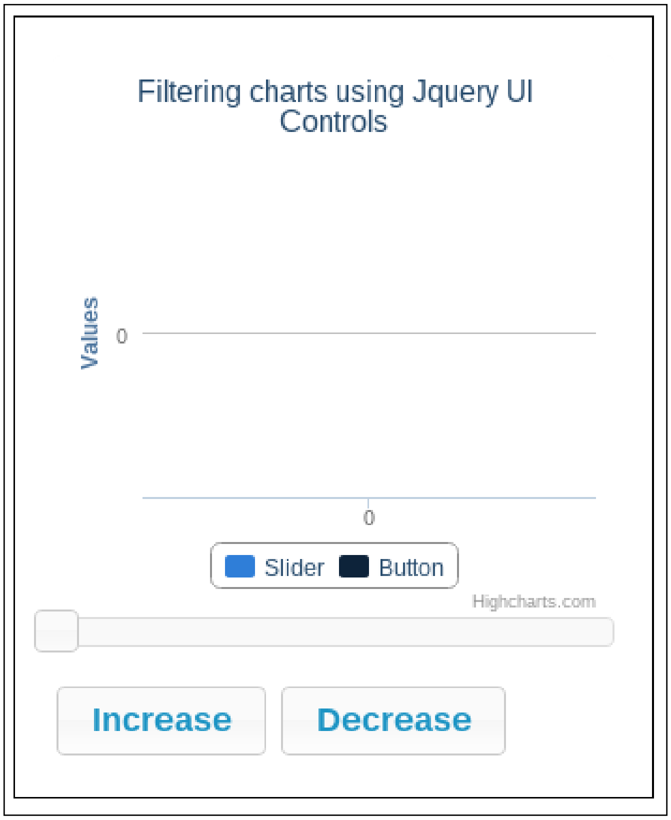
4. Create the slider as follows:

```
$('#slider').slider({
  min: 0,
  max: 100,
  step: 10,
  slide: function (event, ui) {
    chart.series[0].setData([ui.value]);
  }
});
```

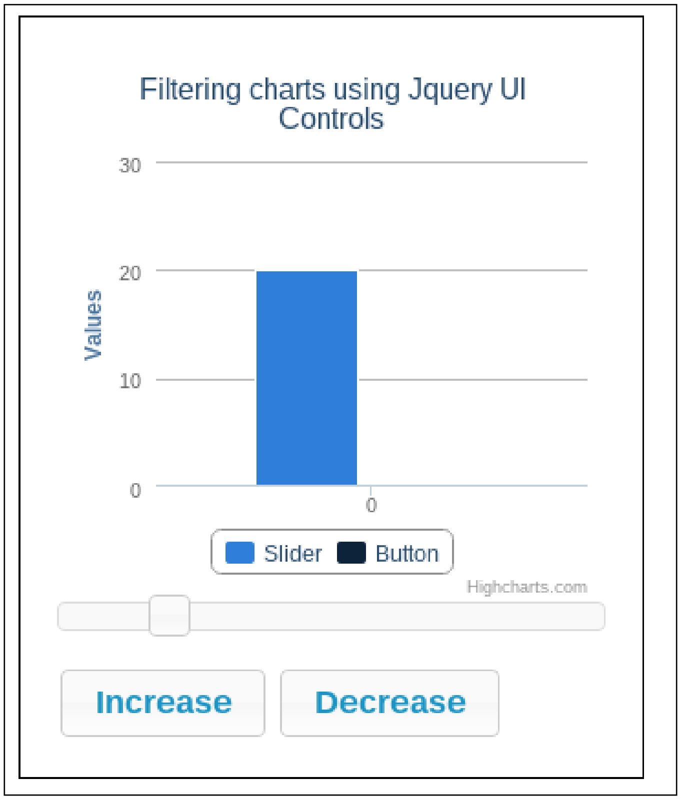
5. Create the buttons using the following code:

```
$('#increase').button().click(function() {
  var data = chart.series[1].data[0].y || 0;
  chart.series[1].setData([data += 1]);
});
$('#decrease').button().click(function() {
  var data = chart.series[1].data[0].y || 0;
  chart.series[1].setData([data -= 1]);
});
```

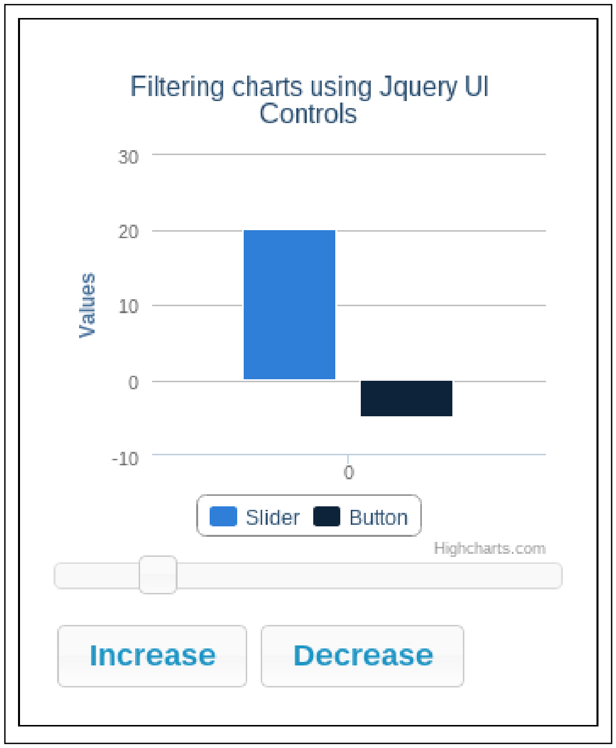
6. Observe the following chart that is rendered initially:



7. Observe the following chart that is rendered after adjusting the slider:



8. Observe the following chart that is rendered after pressing the **Decrease** button a few times:



How it works...

jQuery UI provides almost all of its widgets as plugins. In our case, we use these plugins to create our slider and buttons.

The `.slider()` method takes an object as configuration, and in our case, we define the minimum and maximum values and what each tick of the slider represents (that is, the step value). We also provide the plugin with a `slide()` function to take some action while the slider is moving. In our case, we get the current value of the slider by using `ui.value`, and we set the first series to this value (causing the chart to shrink or grow as we drag the slider). The creation of buttons is even more straightforward. We call `.button()` to convert the buttons into jQuery buttons, and then we can attach a click handler to them.

Putting charts in pages using jQuery Mobile

Our charts have mostly been designed for desktop browsers. While Highcharts works in a large variety of browsers (including mobile browsers on Android and iPhone; see `http://www.highcharts.com/documentation/compatibility` for more details), we haven't really designed our charts for these devices. This recipe will look at how we can use jQuery Mobile to make pages more mobile-friendly.

How to do it...

To get started, perform the following steps:

1. Define the base HTML page as follows:

```
<!DOCTYPE html>
<html>
<head>
 <title>My Page</title>
 <meta name="viewport" content="width=device-width, initial-
  scale=1">
 <link rel="stylesheet" href="http://code.jquery.com/mobile/1.2.1/
  jquery.mobile-1.2.1.min.css" />
 <script src="http://code.jquery.com/jquery-1.8.3.min.js">
 </script>
 <script src="http://code.jquery.com/mobile/1.2.1/jquery.mobile-
 1.2.1.min.js"></script>

</head>
<body>

<div data-role="page">

 <div data-role="header">
  <h1>Highcharts with jQuery Mobile</h1>
 </div>

 <div data-role="content">
  <ul data-role='listview' data-inset='true' data-filter='true'>
    <li><a href='#chart1'>First Chart</a></li>
    <li><a href='#chart2'>Second Chart</a></li>
  </ul>
 </div>
</div>
</body>
</html>
```

2. Define a page for our first chart as follows:

```
       <li><a href='#chart2'>Second Chart</a></li>
  </ul>
 </div>
</div>

<div data-role="page" id='chart1'>
 <div data-role="header">
  <h1>First Chart</h1>
 </div>

 <div data-role="content">
  <div id='container1'></div>
 </div>
</div>
```

3. Define a page for our second chart as follows:

```
    <div id='container1'></div>
 </div>
</div>

<div data-role="page" id='chart2'>
 <div data-role="header">
  <h1>Second Chart</h1>
 </div>

 <div data-role="content">
  <div class='chart' id='container2'
      data-chart='{"type": "column"}'
      data-title='{"text": "Using data-attributes to load
      charts"}'
      data-series='[{"name": "2^x", "data": [8,2,4,1]},
      {"name":"2^(x+1)", "data": [2,16,8,4]}]'
    ></div>
 </div>
</div>
```

4. Define the code to render our first chart as follows:

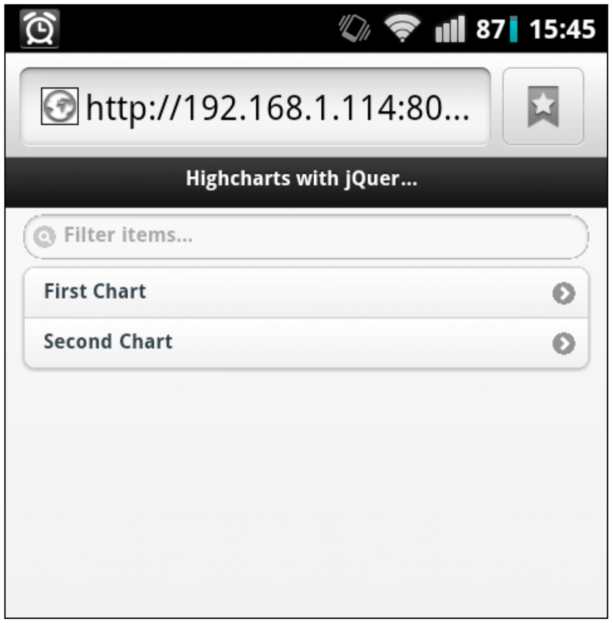
```
      data-series='[{"name": "2^x", "data": [8,2,4,1]}, {"name":
      "2^(x+1)", "data": [2,16,8,4]}]'
    ></div>
 </div>

</div>
```

```
<!-- Include the verbose version of highcharts -->
<script src='./highcharts.src.js'></script>

<!-- Include the verbose version of highcharts extras -->
<script src='./highcharts-more.src.js'></script>

<!-- Include our scripts -->
<script type='text/javascript'>
  $('#chart1').on('pageshow', function() {
    var chart1Options = {
      chart: {
        type: 'column'
      },
      title: {
        text: 'Creating Charts with jQuery'
      },
      subtitle: {
        text: 'Look familiar?'
      },
      series: [{
        name: '2^x',
        data: [1,2,4,8]
      }, {
        name: '2^(x+1) (scrambled)',
        data: [2,4,8,16]
      }]
    };

    $('#container1').highcharts(chart1Options);
  }) ;

</script>
</body>
```

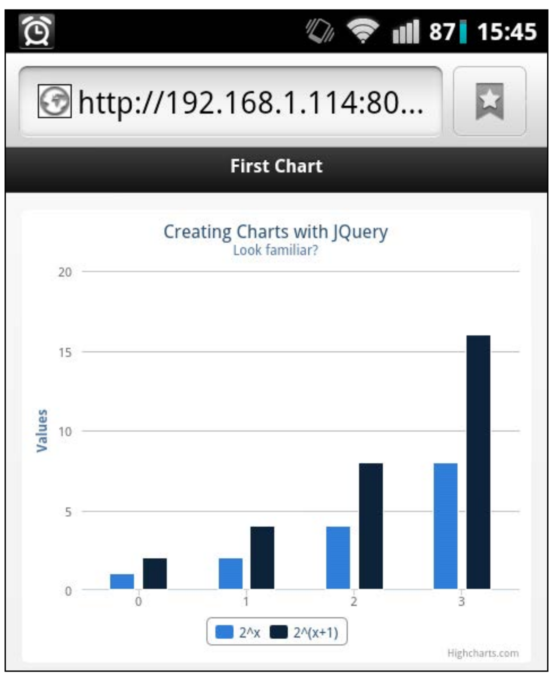
5. Define the code to render our second chart as follows:

```
    $('#container').highcharts(chart1Options);
  }) ;

  $('#chart2').on('pageshow', function() {
    var chart2Settings = $('#container2').data();
    $('#container2').highcharts(chart2Settings);
  });

</script>
</body>
```

6. Examine the list view page shown in the following screenshot:



7. Examine the chart page shown in the following screenshot:



How it works...

A lot of what jQuery Mobile does is handled using `data-` attributes. In our example, we've created three pages (`data-role='page'`) each with its own header (`data-role='header'`) and content (`data-role='content'`). When we create links using `<a href=''>`, jQuery Mobile will automatically create a smooth transition between pages and will visit the next page. In fact, if the link and page IDs match, it will treat the different jQuery Mobile pages in our example as though they were completely separate pages.

On our first page, we've created a special type of element, `listview`. Based on the settings provided, this creates a list of items that we can filter by typing in the textbox it creates.

Rendering our charts is quite different in jQuery Mobile; for starters, we do not use `$(document).ready()`. Again, jQuery Mobile is page-focused, so instead of waiting until the document is ready, we take actions based on page status. The `pageshow` event will fire when a jQuery Mobile page is rendered for the first time, or subsequent times, so we create our charts when their respective pages are shown. Otherwise, it's remarkably similar.

7

Integrating with the Yii Framework

In this chapter, we will cover the following recipes:

- Setting up a simple Yii project
- Creating a chart from model data
- Generating a chart with a Yii CLI command
- Creating charts with a RESTful controller
- Updating the model when the chart changes

Introduction

Integrating with a well-defined backend component can be very easy, but it is often less simple when we are starting from scratch and need to build an entire piece of software from start to end. This chapter focuses on how to build a backend component rapidly and integrate it with the Highcharts library.

Setting up a simple Yii project

In order to get started with Highcharts and Yii, we'll need to set up a base project. While Yii provides tools to handle a lot of this work, we'll need to make some changes in order to get things working.

Getting ready

In order to get started, we will need to do a bit of preparation to get PHP and Yii ready. We'll do this using the following steps:

1. Install PHP. We will need Version 5.1.0 or above. Details on how to install PHP can be found on the PHP website `http://php.net/manual/en/install.php`.
2. Install any necessary **PHP Data Object** (**PDO**) extensions. For these examples, we will be using SQLite which should be included by default, but this may vary from system to system. Details on installing other PDO drivers can be found online (`http://php.net/manual/en/pdo.drivers.php`).
3. Enable the SQLite PDO driver. This is done by editing `php.ini` on your system and finding and uncommenting the appropriate lines, as shown in the following code:

```
; If you only provide the name of the extension, PHP will look for
it in its
; default extension directory.
;
; … other PHP extensions
extension=pdo_sqlite.so
extension=sqlite3.so
```

> To find your `php.ini` file, run `php -ini`.
>
> How PDO extensions are installed varies from system to system. Please consult the manual for more details (`http://php.net/manual/en/pdo.installation.php`).
>
> In this chapter, we will use SQLite for our database; it is possible to use any database, but SQLite is fairly lightweight and easy to get started with. For details on installing SQLite, please refer to your operating system's package manager or the SQLite website (`http://www.sqlite.org/download.html`).

4. Download the Yii framework (`http://www.yiiframework/download/`).

> This chapter was written with reference to Version 1.1 of the Yii framework. The author gives no guarantee that the instructions in this chapter will work with other versions of the Yii framework.

5. Create a folder `yii-highcharts` and unzip the Yii framework to `yii-highcharts/yii`.
6. Start a local server with PHP from `yii-highcharts` using the following command:

```
php -S localhost:8080
```

7. Verify that Yii's requirements are met by visiting `http://localhost:8080/yii/requirements/index.php` and following the instructions on screen.

Also, we will need to make some small changes to the setup that follow from the *Getting Ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

1. In `bower.json`, add `underscore` to our list of dependencies as shown in the following code:

   ```
   "dependencies": {
     "highcharts": "~3.0",
     "underscore": "~1.5"
   }
   ```

2. Create a file, `.bowerrc`, in the same folder as `bower.json`. Place the following contents in that file to ensure that the bower components are installed to the right location, as shown in the following code:

   ```
   {
       "directory": "yii-highcharts/example_app/js"
   }
   ```

3. Install our JavaScript dependencies using the following command:

   ```
   bower install
   ```

How to do it...

To get started, follow the ensuing instructions:

1. Create a skeleton project by running the following command from the `yii-highcharts/yii/framework` folder:

   ```
   ./yiic webapp ../../example_app
   ```

2. Verify whether the skeleton project works by visiting `http://localhost:8080/example_app/index.php`.
3. Configure Gii for model creation. Make the following changes in `example_app/protected/config/main.php`:

   ```
   // ...
   'modules'=>array(
       // uncomment the following to enable the Gii tool
       'gii'=>array(
           'class'=>'system.gii.GiiModule',
           'password'=>'password',
       ),
   ),
   // ...
   ```

4. Create a new folder, `yii-highcharts/example_app/js`, and include the Highcharts files here.
5. Edit `example_app/protected/views/layouts/main.php` to include the required JavaScript files:

```
</div><!-- page -->
<?php
Yii::app()->clientScript->registerCoreScript('jquery');
Yii::app()->clientScript->registerScriptFile(Yii::app()-
>baseUrl.'/js/highcharts/highcharts.src.js');
Yii::app()->clientScript->registerScriptFile(Yii::app()-
>baseUrl.'/js/highcharts/highcharts-more.src.js');
Yii::app()->clientScript->registerScriptFile(Yii::app()-
>baseUrl.'/js/underscore/underscore.js');
?>
</body>
</html>
```

How it works...

Most of what we have coded executes when we run `yiic webapp <name>`. This command uses the Yii framework to create a skeleton project and set up the examples, views, controllers, and whatever else we might need to get started.

Gii is a code generator provided by the Yii framework that we will use later in this chapter to automatically create models and controllers from a database schema. We have enabled it and set a password for its use.

Lastly, we actually include Highcharts on a page; `example_app/protected/views/layouts/main.php` is the wrapper view for the main page layout, so whatever changes we make to this file will affect all other pages. The Yii framework includes some JavaScript of its own including jQuery—these core scripts are registered by name and can be included via `registerCoreScript(name)` as we did earlier. We can also register our own JavaScript for inclusion using `registerScriptFile(filepath)`. The `Yii::app()->baseUrl` string is just a shortcut to get the root folder of our application.

Creating a chart from model data

Now that we've set up our base project, we can really get started with the Yii framework. Since non-static data is typically stored in models, we'll begin by creating a chart from the model data.

Getting ready

To set up the basic Yii project, refer to the *Setting up a simple Yii Project* recipe from this chapter.

How to do it...

Perform the following steps to create a chart from the model data:

1. Open `example_app/protected/data/testdrive.db` using the following command:

   ```
   sqlite3 example_app/protected/data/testdrive.db
   ```

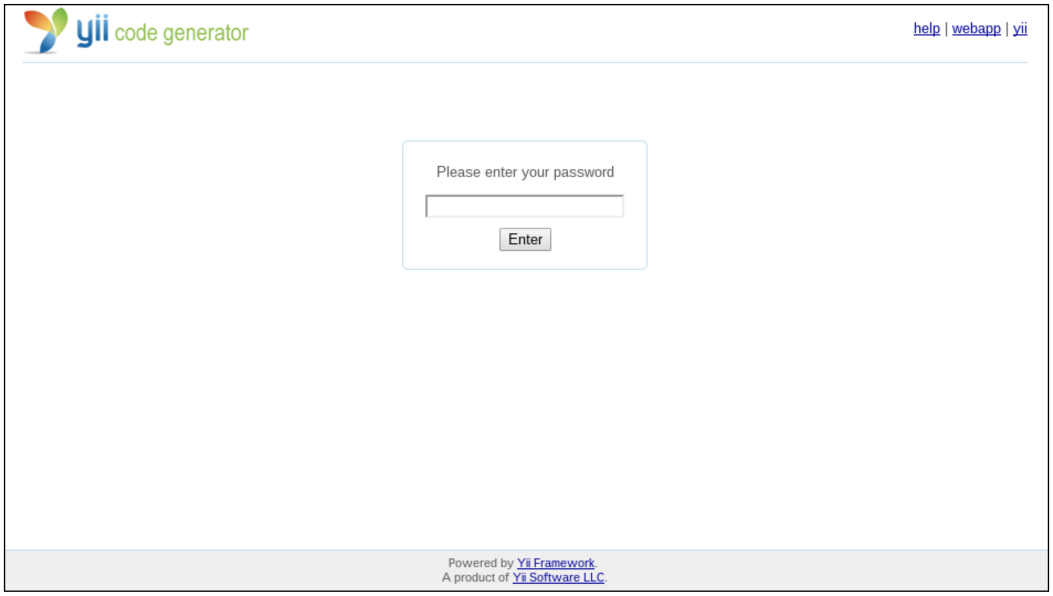
2. Create a table for our data using the following code:

   ```
   CREATE TABLE tbl_monster (
       id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
       name VARCHAR(32) NOT NULL,
       height REAL,
       weight REAL,
       hp INTEGER,
       attack INTEGER,
       defense INTEGER,
       special_attack INTEGER,
       special_defense INTEGER,
       speed INTEGER
   );
   ```

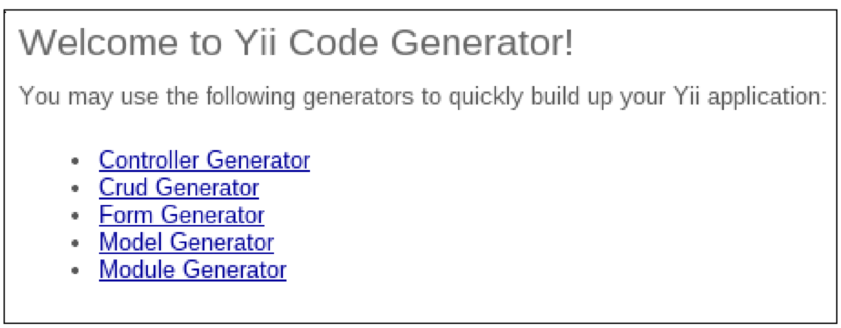
3. Add the example data to our table using the following code:

   ```
   INSERT INTO tbl_monster (name, height, weight, hp, attack,
   defense, special_attack, special_defense, speed) VALUES ('Turtle',
   0.5, 9.0, 44, 48, 65, 50, 64, 43);
   INSERT INTO tbl_monster (name, height, weight, hp, attack,
   defense, special_attack, special_defense, speed) VALUES
   ('Salamander', 0.6, 8.5, 39, 52, 43, 60, 50, 65);
   INSERT INTO tbl_monster (name, height, weight, hp, attack,
   defense, special_attack, special_defense, speed) VALUES
   ('Dinosaur', 0.7, 6.9, 45, 49, 49, 65, 65, 45);
   ```

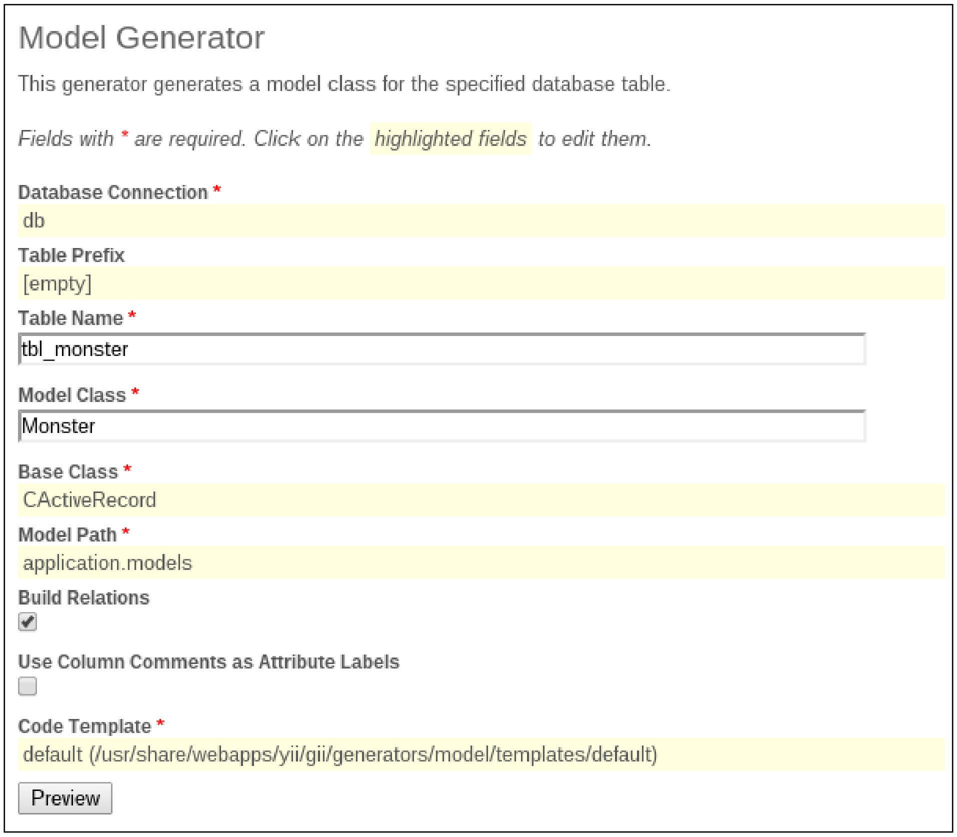
4. Visit `http://localhost:8080/example_app/?r=gii`, and enter the password we defined in `example_app/protected/config/main.php` in the field shown in the following screenshot:



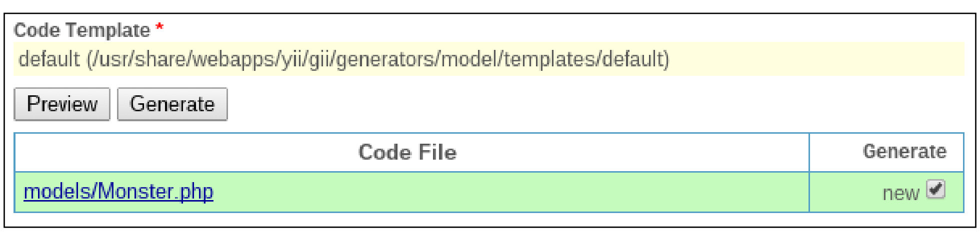
5. Visit **Model Generator** as shown in the following screenshot:



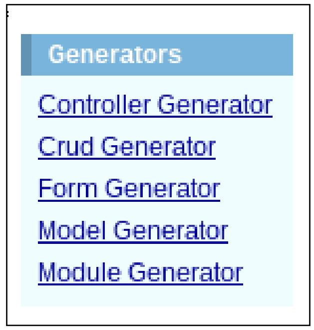
6. Enter the name of the table we created (`tbl_monster`), the name we would like for our model class `Monster`, and then click on **Preview**, as shown in the following screenshot:



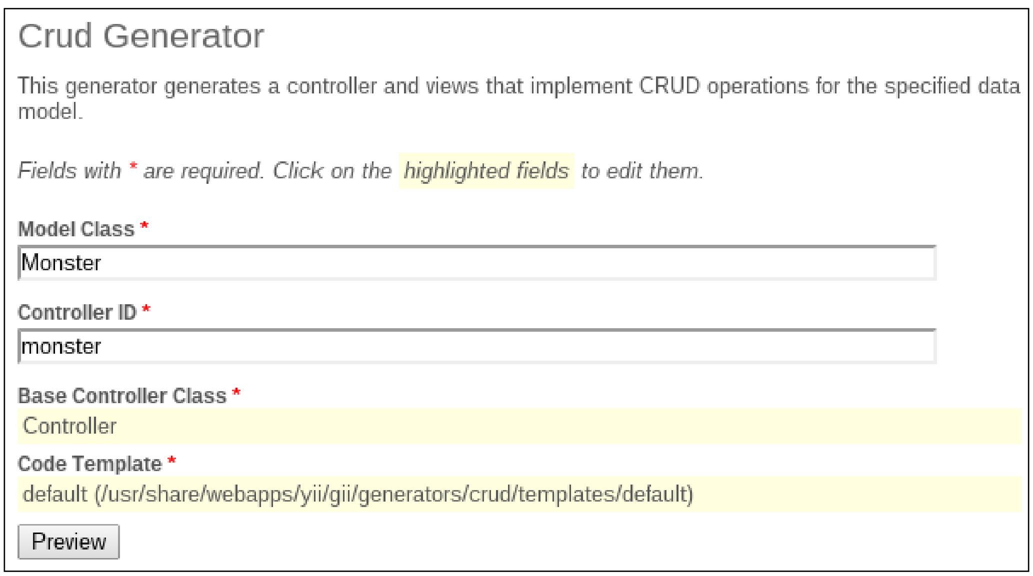
7. On the following page, click on **Generate** as shown in the following screenshot:



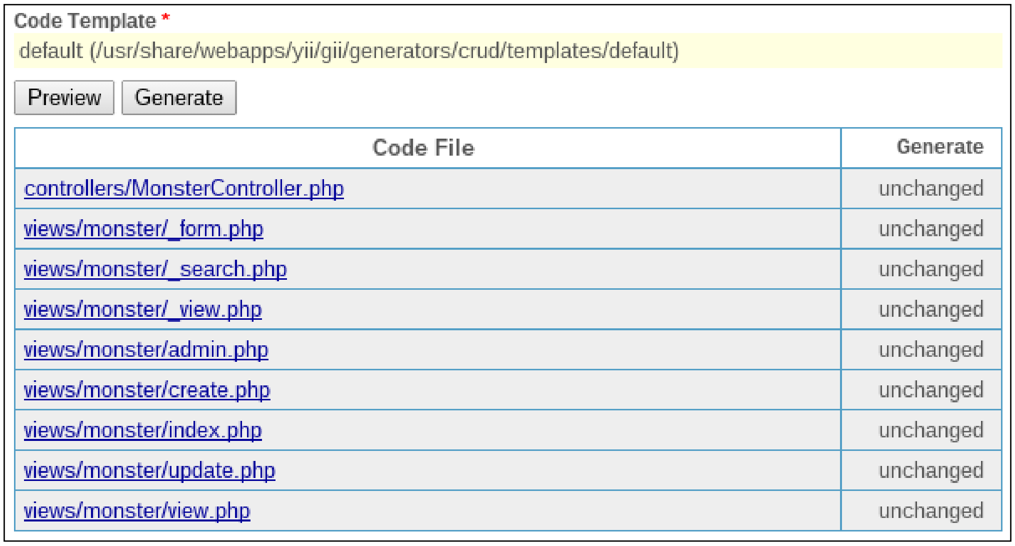
8. From the sidebar, click on **Crud Generator**:



9. Enter the name of the model that we created from a previous step, then click on **Preview**:



10. On the following page, click on **Generate**, as shown in the following screenshot:



11. Create a container for our charts in `example_app/protected/views/monster/_view.php` using the following code:

```
<?php
/* @var $this MonsterController */
/* @var $data Monster */
?>

<div class="view">
     <div
         style='float: right; height:150px; width:250px; '
         id='monster<?php echo CHtml::encode($data->id); ?>'>
         Testing
     </div>
```

12. Uncomment the commented attributes in the generated `example_app/protected/views/monster/_view.php` file as shown in the following code:

```
<b><?php echo CHtml::encode($data->getAttributeLabel('defense'));
?>:</b>
  <?php echo CHtml::encode($data->defense); ?>
  <br />
```

```
<b><?php echo CHtml::encode($data->getAttributeLabel('special_
attack')); ?>:</b>
<?php echo CHtml::encode($data->special_attack); ?>
<br />

<b><?php echo CHtml::encode($data->getAttributeLabel('special_
defense')); ?>:</b>
<?php echo CHtml::encode($data->special_defense); ?>
<br />

<b><?php echo CHtml::encode($data->getAttributeLabel('speed'));
?>:</b>
<?php echo CHtml::encode($data->speed); ?>
<br />
```

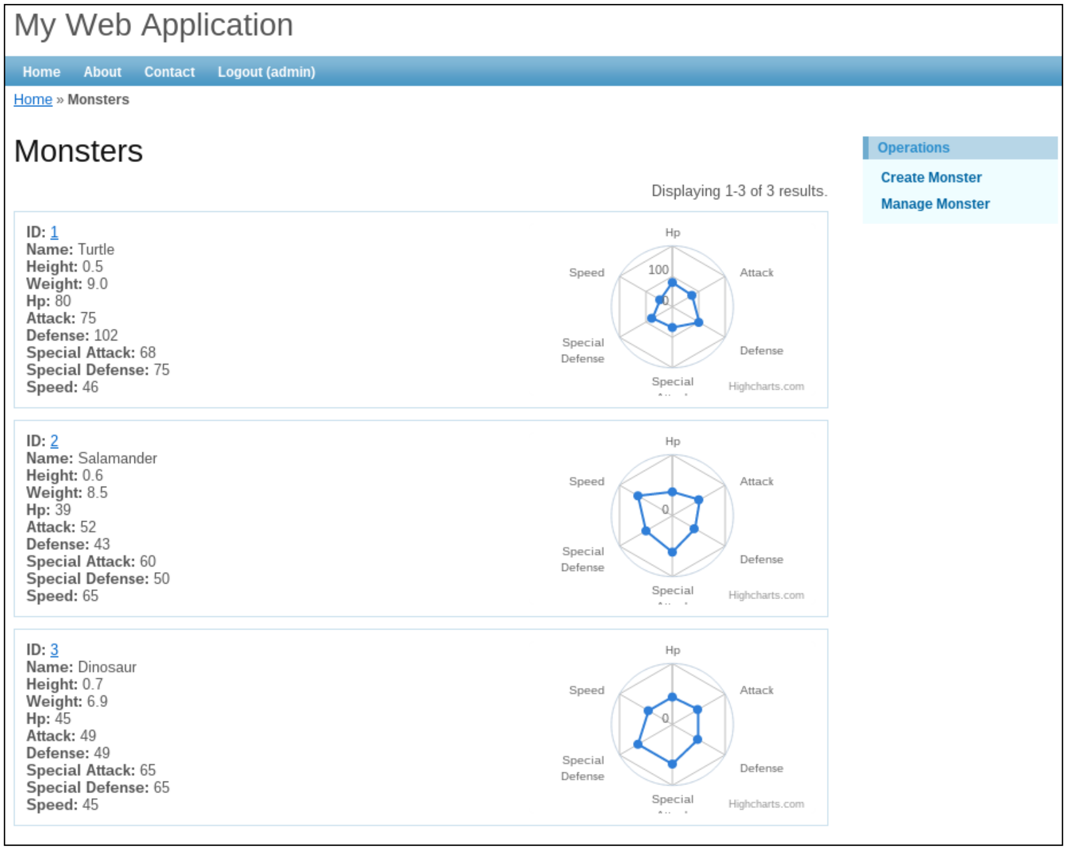
13. Add a JavaScript block at the end of `example_app/protected/views/monster/_view.php` for our data as shown in the following code:

```
<?php echo CHtml::encode($data->speed); ?>
<br />

<script type='text/javascript'>
    (function() {
        var srcData = <?php echo CJSON::encode($data-
        >attributes) ?>;
        var srcCategories = <?php echo CJSON::encode($data-
        >attributeLabels()) ?>;

        var categories = _.chain(srcCategories)
            .omit(['id', 'weight', 'height', 'name'])
            .values()
            .value();

        var data = _.chain(srcData)
            .omit(['id', 'weight', 'height', 'name'])
            .values()
            .map(function(val) {
                return parseInt(val, 10);
            })
            .value();

        var id = '#monster<?php echo CHtml::encode($data->id);
        ?>';
    }());
</script>
</div>
```

14. Add the code for our spiderweb chart to `example_app/protected/views/monster/_view.php`, as shown in the following code:

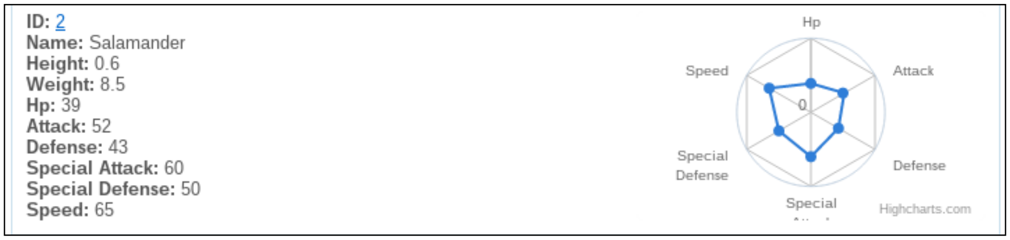
```
var id = '#monster<?php echo CHtml::encode($data->id);
?>'

var options = {
       chart: {polar: true, type: 'line'},
       title: {text: null},
       xAxis: {
           tickmarkPlacement: 'on',
           categories: categories,
           labels: {
               overflow: 'justify',
               style: {
                   fontSize: '10px'
               }
           }
       },
       yAxis: {gridLineInterpolation: 'polygon', min: 0},
       tooltip: {
           shared: true,
           pointFormat: '<span style="color:series.
           color}">{series.name}: <b>${point.y:,.0f}
           </b><br/>'
       },
       legend: {
           enabled: false
       },
       series: [{
           data: data,
           pointPlacement: 'on'
       }]
   };

   jQuery(id).highcharts(options);
}());
```

15. Visit `http://localhost:8080/example_app/?r=monster`. You should get the page as shown in the following screenshot:



How it works...

Gii does most of the work in this recipe. After we've added a table and some sample data to our database (`example_app/protected/data/testdrive.db`), we use its code generation tools to create the model, controller, and views that we use to create, edit, update, and delete our data. Then, all we need to do is make changes to our view, which ends up with something as follows:



In this recipe, we made changes to `_view.php`. The `_view.php` view is used for each element in the list view. Gii also created a number of other views, such as `view.php` (used when looking at a single model), `update.php` (used to create the form to update the `Monster` model), and `search.php` (used when searching for `Monsters`).

Most of what we did in this recipe should look familiar, the big difference being how we obtain our data. In `_view.php`, `$data` represents the model instance. In order to access the relevant data, we needed to access it via `$data->attributes`, and then convert it into something that we can manipulate in JavaScript. We used `CJSON::encode($data)` to convert the model data from PHP to JSON. We could have also accessed the various attributes via `$data->attributeName` (as we did with `$data->id`), but in this case, it is more convenient to convert all the data and use underscore to filter out the data we didn't need.

Generating a chart with a Yii CLI command

In past recipes, we've explored how to generate a chart from the command line, or more correctly, on the server side. In this recipe, we'll be leveraging Yii's command-line capabilities to create a chart from the command line.

Getting ready

To set up the basic Yii project, refer to the *Setting up a simple Yii Project* recipe from this chapter.

We will also need to install PhantomJS. As the bower installation does not include some important scripts, we'll also need to download Highcharts normally using the following steps:

1. Visit the Highcharts website and download Highcharts (`http://www.highcharts.com/download`).
2. Extract the contents of the downloaded ZIP file, and copy the `exporting-server/phantomjs` folder to `yii-highcharts/example_app/js/highcharts-phantomjs`.
3. Install PhantomJS following the instructions found on the PhantomJS website (`http://phantomjs.org`).

How to do it...

To get started, follow the ensuing instructions:

1. Create a new file `highchartsCommand.php` in `example_app/protected/commands` using the following code:

```
<?php

class highchartsCommand extends CConsoleCommand
{
}
?>
```

2. Add a method called `actionGenerate` as shown in the following code:

```
class highchartsCommand extends CConsoleCommand
{
    public function actionGenerate($file, $scale=False,
    $width=False, $format='pdf') {
        $PHANTOM_JS = exec('which phantomjs');
        $SCRIPT ="\"".Yii::app()->basePath.'/../js/highcharts-
        phantomjs/highcharts-convert.js'."\"";

        echo "Generating chart...\n";

        $cmd = $PHANTOM_JS." ".$SCRIPT;

        if ($scale) {
            $cmd .= " -scale ".$scale;
        }

        if($width) {
            $cmd .= " -width ".$width;
        }

        $cmd .= ' -infile '.$file;
        $cmd .= ' -outfile file.'.$format;

        echo exec($cmd);
    }
}
```

3. Define our chart in `yii-highcharts/options.json` as shown in the following code:

```
{
    "chart": {
        "type": "bar"
    },
    "title": {
        "text": "Exporting images to different formats"
    },
    "series": [{
        "name": "Bar #1",
        "data": [1,2,3,4]
    }]
}
```

4. Navigate to `example_app/protected`, and generate a chart with the following command:

```
./yiic highcharts generate –file=../../options.json
```

How it works...

Yii allows us to create command-line arguments easily. All that we need to do is create a new file `<myCommand>Command.php` in the `example_app/protected/commands` folder, extend `CConsoleCommand` in that file, and our command will show up in the list of available commands when we run `yiic`.

We can then create specific actions (such as `generate` in our case) by adding a `action<actionName>` public function to that class. Any arguments that are included as the function definition will become available as command-line options; for example, if one of our arguments was `$banana`, we could pass that to our action by appending `--banana=value` to our command-line arguments. We can also set the default values by setting `$arg='value'` in our function definition.

More information on creating console commands in Yii can be found in the Yii documentation (`http://yiiframework.com/doc/guide/1.1/en/topics.console`).

There's more...

Our preceding example is pretty simple. However, what if we wanted to pull data from a model instead of using a passed-in file? In this case, we can leverage the connection to the database. We could do something like the following in our action:

```
$sql = "SELECT * FROM {{monster}}";
$connection = Yii::app()->db;
$monsters=$connection->createCommand($sql)->queryAll();
```

This code would connect to the database and get all of the rows from the `monster` table. With this information, we could then inject the data into our chart configuration. This is left as an exercise for the reader.

More information on working with the database can be found in the **Data Access Object** (**DAO**) documentation (`http://yiiframework.com/doc/guide/1.1/en/database.dao`).

Creating charts with a RESTful controller

For ease of understanding or just to make our API calls tidier, we may want to use RESTful services. This recipe looks at how we can make our existing services RESTful and how we can leverage these changes in our charts.

Getting ready

To set up the basic Yii project, refer to the *Setting up a simple Yii Project* recipe from this chapter.

We will also need to download and set up **RestfullYii**, an extension for the Yii framework using the following steps:

1. Download the appropriate version of RestfullYii from the RestfullYii page (`http://yiiframework.com/extension/restfullyii`) and extract the contents to `example_app/protected/extensions`.
2. Add an `aliases` entry in `example_app/protected/config/main.php` as shown in the following code:

```
return array(
    'basePath'=>dirname(__FILE__).DIRECTORY_SEPARATOR.'..',
    'name'=>'My Web Application',

    'aliases' => array(
        'RestfullYii' =>realpath(__DIR__.'/../extensions/starship/
         RestfullYii'),
    ),
```

3. Uncomment the `urlManager` array and add the following rules:

```
'components'=>array(
    'user'=>array(
        // enable cookie-based authentication
        'allowAutoLogin'=>true,
    ),
    // uncomment the following to enable URLs in path-format
    'urlManager'=>array(
        'urlFormat'=>'path',
        'rules'=>[
            'api/<controller:\w+>'=>['<controller>/REST.GET',
            'verb'=>'GET'],
            'api/<controller:\w+>/<id:\w*>'=>['<controller>/
REST.GET', 'verb'=>'GET'],
            'api/<controller:\w+>/<id:\w*>/<param1:\
w*>'=>['<controller>/REST.GET', 'verb'=>'GET'],
            'api/<controller:\w+>/<id:\w*>/<param1:\
w*>/<param2:\w*>'=>['<controller>/REST.GET', 'verb'=>'GET'],

            ['<controller>/REST.PUT',
'pattern'=>'api/<controller:\w+>/<id:\w*>', 'verb'=>'PUT'],
            ['<controller>/REST.PUT',
'pattern'=>'api/<controller:\w+>/<id:\w*>/<param1:\w*>',
'verb'=>'PUT'],
            ['<controller>/REST.PUT',
'pattern'=>'api/<controller:\w*>/<id:\w*>/<param1:\
w*>/<param2:\w*>', 'verb'=>'PUT'],

            ['<controller>/REST.DELETE',
'pattern'=>'api/<controller:\w+>/<id:\w*>','verb'=>'DELETE'],
            ['<controller>/REST.DELETE',
'pattern'=>'api/<controller:\w+>/<id:\w*>/<param1:\w*>',
'verb'=>'DELETE'],
            ['<controller>/REST.DELETE',
'pattern'=>'api/<controller:\w+>/<id:\w*>/<param1:\
w*>/<param2:\w*>', 'verb'=>'DELETE'],

            ['<controller>/REST.POST',
'pattern'=>'api/<controller:\w+>', 'verb'=>'POST'],
            ['<controller>/REST.POST',
'pattern'=>'api/<controller:\w+>/<id:\w+>', 'verb'=>'POST'],
            ['<controller>/REST.POST',
'pattern'=>'api/<controller:\w+>/<id:\w*>/<param1:\w*>',
'verb'=>'POST'],
```

```
                ['<controller>/REST.POST',
'pattern'=>'api/<controller:\w+>/<id:\w*>/<param1:\
w*>/<param2:\w*>', 'verb'=>'POST'],

                '<controller:\w+>/<id:\d+>'=>'<controller>/view',
                '<controller:\w+>/<action:\w+>/<id:\
                 d+>'=>'<controller>/<action>',
                '<controller:\w+>/<action:\
                 w+>'=>'<controller>/<action>',
            ],
        ),
```

4. Change the `filters` method in our `MonsterController` (`example_app/protected/controllers/MonsterController.php`) as shown in the following code:

```
public function filters()
    {
        return array(
            'accessControl', // perform access control for CRUD
             operations
           // 'postOnly + delete', // we only allow deletion via
           POST request
           array(
                'ext.starship.RestfullYii.filters.ERestFilter +
                REST.GET, REST.PUT, REST.POST, REST.DELETE'
            ),
        );
    }
```

5. Add an `actions` method in our `MonsterController` as shown in the following code:

```
public function actions()
{
    return array(
        'REST.'=>'ext.starship.RestfullYii.actions.
         ERestActionProvider',
    );
}
```

6. Change the `accessRules` method in our `MonsterController` as shown in the following code:

```
public function accessRules()
{
    return array(
       // remove previous allow rules
        array('allow',
            'actions'=>array('REST.GET', 'REST.PUT', 'REST.POST',
            'REST.DELETE'),
            'users'=>array('*'),
        ),
        array('deny',  // deny all users
            'users'=>array('*'),
        ),
    );
}
```

7. Verify that our new API works using the following command:

```
curl -i -H "Accept: application/json" -H "X_REST_USERNAME:
admin@restuser" -H "X_REST_PASSWORD: admin@Access" http://
localhost:8080/example_app/api/monster

@Access" http://localhost:8080/example_app/api/monster

HTTP/1.1 200 OK

Host: localhost:8080

Connection: close

X-Powered-By: PHP/5.5.5

Set-Cookie: PHPSESSID=3gm61kbl54ptm3iqrbcsd363i1; path=/

Expires: Thu, 19 Nov 1981 08:52:00 GMT

Cache-Control: no-store, no-cache, must-revalidate, post-check=0,
pre-check=0

Pragma: no-cache

Content-type: application/json


{"success":true,"message":"Record(s) Found","data":{"totalCou
nt":"3","monster":[{"id":"1","name":"Turtle","height":"0.5","w
eight":"9.0","hp":"44","attack":"48","defense":"65","special_
attack":"50","special_defense":"64","speed":"43"},{"id":"2","na
me":"Salamander","height":"0.6","weight":"8.5","hp":"39","attac
k":"52","defense":"43","special_attack":"60","special_defense":
"50","speed":"65"},{"id":"3","name":"Dinosaur","height":"0.7","
weight":"6.9","hp":"45","attack":"49","defense":"49","special_
attack":"65","special_defense":"65","speed":"45"}]}}
```

> In this example, we've used curl, but just about any application that can make an HTTP request can perform this check.
>
> For more information on curl, visit the curl website (`http://curl.haxx.se`).

How to do it...

To get started, follow the ensuing instructions:

1. Edit `example_app/protected/controllers/SiteController.php` as shown in the following code:

```
 public function actionIndex()
{
    // renders the view file 'protected/views/site/index.php'
    // using the default layout 'protected/views/layouts/main.
    php'
    $this->render('index');
}

public function actionExample()
{
    $this->render('example');
}
```

2. Create a new file, `example_app/protected/views/site/example.php`, as shown in the following code:

```
<?php
$this->pageTitle=Yii::app()->name . ' - Example';
$this->breadcrumbs=array(
    'Example',
);
?>
<h1>Example page</h1>

<div id='container'>
</div>
```

3. Create JavaScript to fetch data as shown in the following code:

```
</div>

<script type='text/javascript'>
    jQuery.ajax({
        url: '/example_app/api/monster',
        type: 'GET',
        headers: {
            "X_REST_USERNAME": "admin@restuser",
            "X_REST_PASSWORD": "admin@Access"
        },
        success: function(response) {
        }
    });
</script>
```

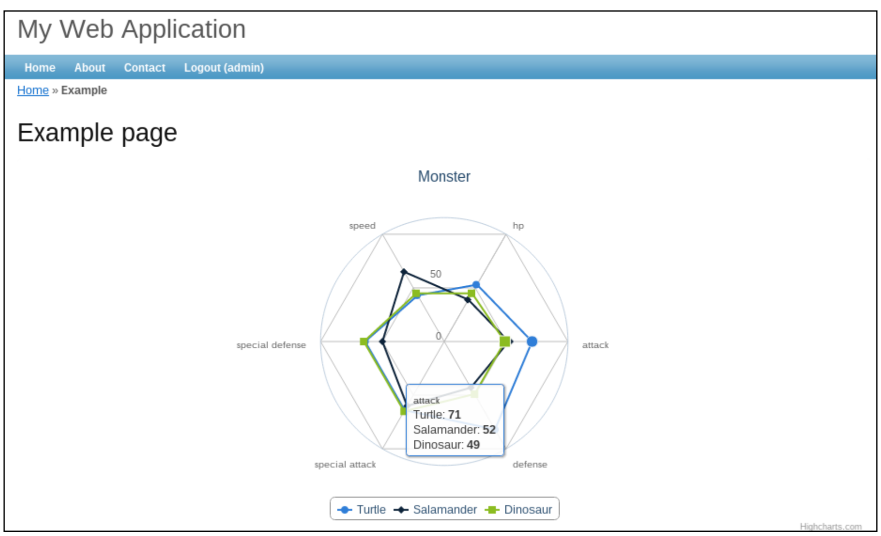
4. Transform the data for easier use in our chart using the following code:

```
success: function(response) {
            var options, categories, seriesData;
            console.log(response);

            categories = _.chain(response.data.monster[0])
                .omit(['id', 'weight', 'height', 'name'])
                .keys()
                .map(function(elem) {
                    return elem.split('_').join(' ');
                }).value();

            seriesData = _.chain(response.data.monster)
                .map(function(elem){
                    var result = {};

                    result.name = elem.name;
                    result.data = _.chain(elem)
                        .omit(['id', 'weight', 'height', 'name'])
                        .values()
                        .map(function(val) {
                            return parseInt(val, 10);
                        })
                        .value();

                    return result;
                }).value();
                //...
```

5. Create our chart using the following code:

```
options = {
    chart: {
        polar: true,
        type: 'line'
    },
    title: {text: 'Monster'},
    xAxis: {
        tickmarkPlacement: 'on',
        categories: categories,
        labels: {
            overflow: 'justify',
            style: {
                fontSize: '10px'
            }
        }
    },
    yAxis: {gridLineInterpolation: 'polygon', min: 0},
    tooltip: {
        shared: true,
        pointFormat: '<span style="color:series.
        color}">{series.name}: <b>{point.y:,.0f}
        </b><br/>'
    },
    series: seriesData
};
$('#container').highcharts(options);
```

6. View the chart at `http://localhost:8080/example_app/site/example`. It should look as follows:



How it works...

restfulyii takes our existing controllers and makes a few changes to make our URLs RESTful. Changing `urlManager` in `config/main.php` sets up the rules for what our URLs can look like and what actions should take place. Then, when we made changes to the `MonsterController`, this allowed us to set rules for accessing resources and informed us about which HTTP verbs we can use.

Our JavaScript, in this example, is very similar to what we've seen in the past. The biggest difference is that we make a call to our API first using `jQuery.ajax` and then, we create our chart.

Updating the model when the chart changes

In previous chapters, we've focused on getting data from a server and sending it to the browser and rendering a chart, with a few examples on how we can send data in the other direction. In this recipe, we'll be demonstrating how we can update the model on the server from the browser. This recipe will leverage the work done in the previous recipe.

Getting ready

Complete all the steps in the previous recipe *Creating charts with a RESTful controller*.

How to do it...

To get started, follow the ensuing instructions:

1. Replace our existing `example_app/protected/views/site/example.php` file with the following:

```
<?php
$this->pageTitle=Yii::app()->name . ' - Example';
$this->breadcrumbs=array(
    'Example',
);
?>
<h1>Example page</h1>

<div id='container'>
</div>
```

```
<center>
    Give a treat to:
    <div class='buttons'>
    </div>
<center>

<script type='text/javascript'>
</script>
```

2. Create a function to get the categories for our chart using the following code:

```
<script type='text/javascript'>
var getStats = function(categories) {
    return _.chain(categories)
       .omit(['id', 'weight', 'height', 'name'])
       .keys()
       .map(function(elem) {
           return elem.split('_').join(' ');
       }).value();
};
```

3. Create a function to get series data for our chart using the following code:

```
var getStats = function(categories) {/* … */};
var getSeries = function(series) {
    return _.chain(series)
       .map(function(elem){
           var result = {};
           result.name = elem.name;
           result.data = _.chain(elem)
               .omit(['id', 'weight', 'height', 'name'])
               .values()
               .map(function(val) {
                   return parseInt(val, 10);
               })
               .value();
             return result;
       }).value();
};
```

4. Create a function to draw the chart as shown in the following code:

```
            var getStats = function(categories) {/* … */};
            var getSeries = function(series) {/* … */};
var drawChart = function (response) {
    var options = {
        chart: {
            polar: true,
            type: 'line'
        },
        title: {text: 'Monster'},
        xAxis: {
            tickmarkPlacement: 'on',
            categories: getStats(response.data.monster[0]),
            labels: {
                overflow: 'justify',
                style: {
                    fontSize: '10px'
                }
            }
        },
        yAxis: {gridLineInterpolation: 'polygon', min: 0},
        tooltip: {
            shared: true,
            pointFormat: '<span style="color:series.
            color}">{series.name}: <b>{point.y:,.0f}</b><br/>'
        },
        series: getSeries(response.data.monster)
    };
    $('#container').highcharts(options);
};
```

5. Create a function to update a model as shown in the following code:

```
var getStats = function(categories) {/* … */};
var getSeries = function(series) {/* … */};
      var drawChart = function (response) {/* … */ };
var levelUpMonster = function (monster) {
    var id = monster.id
    $.ajax({
        url: '/example_app/api/monster/' + id,
        type: 'PUT',
```

```
            headers: {
                "X_REST_USERNAME": "admin@restuser",
                "X_REST_PASSWORD": "admin@Access"
            },
            data: JSON.stringify(monster),
            success: function() {
                reload();
            }
        });
    };
```

6. Create a function handle by clicking on different monster buttons (for example, **Turtle**, **Salamander**, and **Dinosaur**) using the following code:

```
var getStats = function(categories) {/* … */};
var getSeries = function(series) {/* … */};
var drawChart = function (response) {/* … */};
var levelUpMonster = function (monster) {/* … */};
var feedMonster = function(data) {
    // Randomly select a stat
    key = _.chain(data).omit(['id', 'weight', 'height', 'name']).
    keys().shuffle().first().value();

    // Randomly increase the stat
    data[key] = parseInt(data[key], 10) + _.random(1,10);

    levelUpMonster(data);
};
```

7. Create a function to create buttons as shown in the following code:

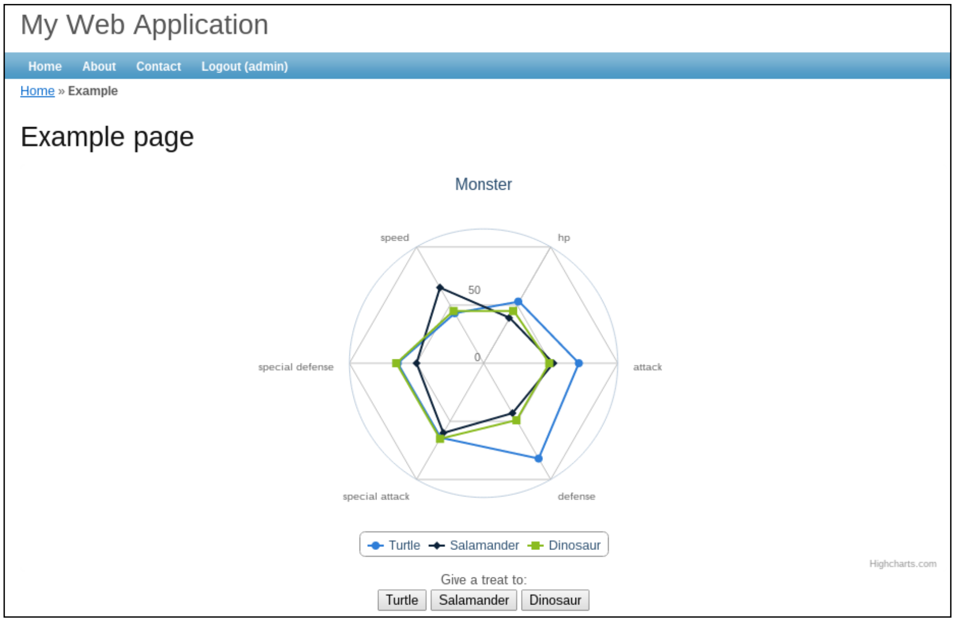
```
var getStats = function(categories) {/* … */};
var getSeries = function(series) {/* … */};
      var drawChart = function (response) {/* … */};
var levelUpMonster = function (monster) {/* … */};
var feedMonster = function (monster) {/* … */};
var createButtons = function (monsters) {
    $('.buttons').empty();
    _.chain(monsters)
```

```
        .each(function(data) {
            var $button = $('<input>', {
                type: 'button',
                value: data.name,
                id: data.name
            });
            $button.click(function() {
                feedMonster(data);
            });
            $('.buttons').append($button);
        });
};
```

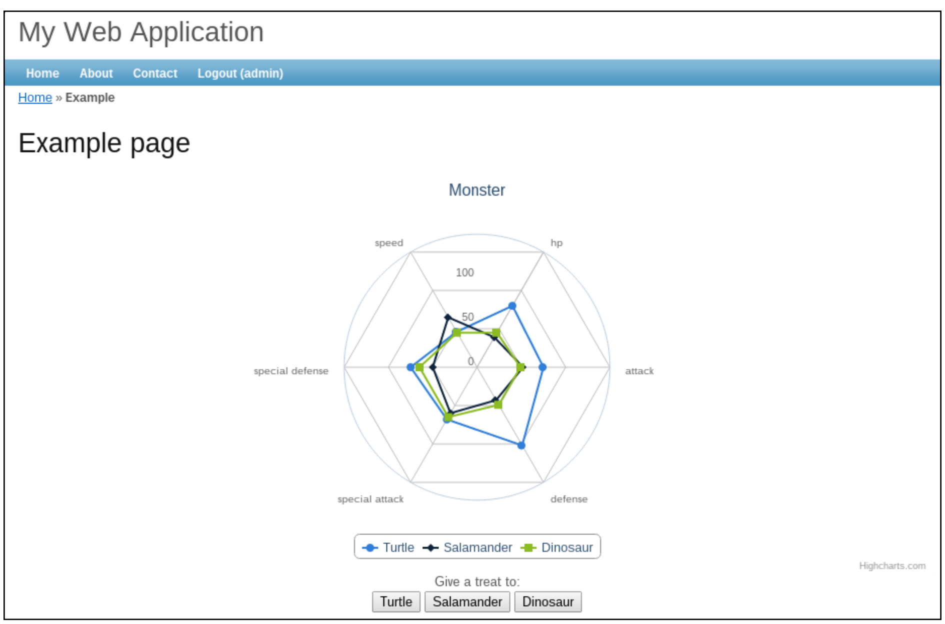
8. Create a function to reload the data as shown in the following code:

```
var getStats = function(categories) {/* … */};
var getSeries = function(series) {/* … */};
      var drawChart = function (response) {/* … */};
var levelUpMonster = function (monster) {/* … */};
var feedMonster = function (monster) {/* … */};
var createButtons = function (monsters) {/* … */};
    var reload = function() {
        jQuery.ajax({
            url: '/example_app/api/monster',
            type: 'GET',
            headers: {
                "X_REST_USERNAME": "admin@restuser",
                "X_REST_PASSWORD": "admin@Access"
            },
            success: function(response) {
                drawChart(response);
                createButtons(response.data.monster);
            }
        });
    };
    reload();
</script>
```

9. Visit `http://localhost:8080/example_app/site/example`. You should get the following page:



10. Give a treat to the Turtle multiple times and observe the changes in the chart as shown in the following screenshot:



How it works...

It may seem like we have a lot going on, but what we've done is just broken down our previous recipe into smaller pieces and added on to that.

When we call the `reload()` method, we fetch data via our RESTful controller, draw the chart, and create buttons.

When we click on a button, we increment one of the monsters stats (via `feedMonster`), and when that happens, we submit a `PUT` request to change the model (via `levelUpMonster`). After that has happened, we reload the chart to display the new data.

8
Integrating with Other Frameworks

In this chapter, we will cover the following recipes:

- Using NodeJS as a data provider
- Using Django as a data provider
- Using Flask/Bottle as a data provider
- Integrating with Backbone
- Using AngularJS data bindings and controllers
- Using NodeJS for chart rendering

Introduction

There exists a wide variety of different tools and frameworks spanning different languages and paradigms, and this list of tools continues to grow and expand. This chapter examines a few of the more popular tools and gives us some idea on how to integrate these different tools with Highcharts.

Using NodeJS as a data provider

JavaScript has become a formidable language in its own right. Google's work on the V8 JavaScript engine has enabled others to develop NodeJS, and with it, allowed the development of JavaScript on the server side. This chapter will take a look at how we can serve data using NodeJS, specifically using a framework known as **express**.

Getting ready

We will need to set up a simple project before we can get started using the following steps:

1. Download and install NodeJS (`http://nodejs.org/download/`).
2. Create a folder `nodejs` for our project.
3. Create a file `nodejs/package.json` and fill it with the following contents:

```
{
  "name": "highcharts-cookbook-nodejs",
  "description": "An example application for using highcharts
   with nodejs",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "3.4.4"
  }
}
```

4. From within the `nodejs` folder, install our dependencies locally (that is, within the `nodejs` folder) using **npm** (**NodeJS package manager**) using the following command:

```
npm install
```

> If we wanted to install packages globally, we could have instead done the following:
>
> ```
> npm install -g
> ```

5. Create a folder `nodejs/static`, which will later contain our static assets (for example, a web page and our JavaScript).
6. Create a file `nodejs/app.js`, which will later contain our express application and data provider.

7. Create a file `nodejs/bower.json` to list our JavaScript dependencies for the page using the following code:

```
{
  "name": "highcharts-cookbook-chapter-8",
  "dependencies": {
    "jquery": "^1.9",
    "highcharts": "~3.0"
  }
}
```

8. Create a file `nodejs/.bowerrc` to configure where our JavaScript dependencies will be installed, as shown in the following code:

```
{
    "directory": "static/js"
}
```

How to do it...

Let's begin. Perform the following steps:

1. Create an example file `nodejs/static/index.html` to view our charts using the following steps:

```
<html>
    <head>
    </head>
    <body>
        <div id='example'></div>
        <script src='./js/jquery/jquery.js'></script>
        <script src='./js/highcharts/highcharts.js'></script>

        <script type='text/javascript'>
            $(document).ready(function() {
                var options = {
                    chart: {
                        type: 'bar',
                        events: {
                            load: function () {
                                var self = this;
                                setInterval(function() {
                                    $.getJSON('/ajax/series',
                                    function(data) {
                                        var series = self.
                                        series[0];
```

```
                                        series.setData(data);
                                    });
                                }, 1000);
                            }
                        }
                    },
                    title: {
                        text: 'Using AJAX for polling charts'
                    },
                    series: [{
                        name: 'AJAX data (series)',
                        data: []
                    }]
                };
                $('#example').highcharts(options);
            });
        </script>
    </body>
</html>
```

2. In `nodejs/app.js`, import the express framework using the following code:

```
var express = require('express');
```

3. Create a new express application using the following code:

```
var app = express();
```

4. Tell our application from where to serve static files using the following code:

```
var app = express();
app.use(express.static('static'));
```

5. Create a method to return the data using the following code:

```
app.use(express.static('static'));

app.get('/ajax/series', function(request, response) {
    var count = 10,
        results = [];

    for(var i = 0; i < count; i++) {
        results.push({
            "y": Math.random()*100
        });
    }

    response.json(results);
});
```

6. Listen on port 8888 using the following code:

```
    response.json(results);
});
app.listen(8888);
```

7. Start our application using the following command:

```
node app.js
```

8. View the output on `http://localhost:8888/index.html`.

How it works...

Most of what we've done in our application is fairly simple: creating an express instance, creating request methods, and listening on a certain port.

With express, we could also process different HTTP verbs such as `POST` or `DELETE`. We can handle these methods by creating a new request method. In our example, we handled the `GET` requests (that is, `app.get`), but in general, we can use `app.VERB` (where `VERB` is an HTTP verb). In fact, we can also be more flexible in what our URLs look like: we can use JavaScript regular expressions. More information on the express API can be found at `http://expressjs.com/api.html`.

Using Django as a data provider

Django is likely one of the more robust Python frameworks and certainly one of the oldest. As such, Django can be used to tackle a variety of different cases and has a lot of available support and extensions. This recipe will look at how we can leverage Django to provide data for Highcharts.

Getting ready

Perform the following steps before we proceed:

1. Download and install Python 2.7 (`http://www.python.org/`
2. Download and install Django (`http://www.djangoproject.com/download/`).
3. Create a new folder `django` for our project.
4. From within the `django` folder, run the following command to create a new project:

```
django-admin.py startproject example
```

5. Create a file `django/bower.json` to list the following JavaScript dependencies:

```
{
  "name": "highcharts-cookbook-chapter-8",
  "dependencies": {
    "jquery": "^1.9",
    "highcharts": "~3.0"
  }
}
```

6. Create a file `django/.bowerrc` to configure where our JavaScript dependencies will be installed. The following code gives this location:

```
{
    "directory": "example/static/js"
}
```

7. Create a folder `example/templates` for any templates we may have.

How to do it...

To get started, follow the ensuing instructions:

1. Create a folder `example/templates` and include a file `index.html` as follows:

```
{% load staticfiles %}
<html>
    <head>
    </head>
    <body>
        <div class='example' id='example'></div>

        <script src='{% static "js/jquery/jquery.js" %}'></script>
        <script src='{% static "js/highcharts/highcharts.js"
%}'></script>

        <script type='text/javascript'>
            $(document).ready(function() {
                var options = {
                    chart: {
                        type: 'bar',
                        events: {
                            load: function () {
                                var self = this;
                                setInterval(function() {
                                    $.getJSON('/ajax/series',
function(data) {
```

```
                                        var series = self.
                                        series[0];
                                        series.setData(data);
                                    });
                                }, 1000);
                            }
                        }
                    },
                    title: {
                        text: 'Using AJAX for polling charts'
                    },
                    series: [{
                        name: 'AJAX data (series)',
                        data: []
                    }]
                };
                $('#example').highcharts(options);
            });
        </script>
    </body>
</html>
```

2. Edit `example/example/settings.py` and include the following code at the end of the file:

```
STATIC_URL = '/static/'

TEMPLATE_DIRS = (
    os.path.join(BASE_DIR, 'templates/')
)

STATICFILES_DIRS = (
    os.path.join(BASE_DIR, 'static/'),
)
```

3. Create a file `example/example/views.py`, and create a handler to show our page as shown in the following code:

```
from django.shortcuts import render_to_response

def index(request):
    return render_to_response('index.html')
```

4. Edit `example/example/views.py`, and create a handler to serve our data as shown in the following code:

```
import json
from random import randint
from django.http import HttpResponse
from django.shortcuts import render_to_response

def index(request):
    return render_to_response('index.html')

def series(request):
    results = []

    for i in xrange(1, 11):
        results.append({
            'y': randint(0, 100)
        })

    json_results = json.dumps(results)
    return HttpResponse(json_results, mimetype='application/json')
```

5. Edit `example/example/urls.py` to register our URL handlers using the following code:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

import views

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'example.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),

    url(r'^admin/', include(admin.site.urls)),

    url(r'^/?$', views.index, name='index'),
    url(r'^ajax/series/?$', views.series, name='series'),
)
```

6. Run the following command from the `django` folder to start the server:

```
python example/manage.py runserver
```

7. Observe the page by visiting `http://localhost:8000`.

How it works...

There are a lot of different things going on here, so let's try to understand some of the specifics.

The `django-admin.py startproject <name>` command creates a skeleton project for us. The `settings.py` file includes any settings relevant to running our application such as where we can find templates (that is, `TEMPLATE_DIRS`) or static files (that is, `STATICFILES_DIRS`). The `urls.py` file lists the different routes in our application. Each route has a path listed as a regular expression, a reference to a Python function, and a name that we can use to look up the view from our code. Lastly, there is `views.py`, which contains handler functions and serves content.

One might notice that in our `index.html` file, we've included some unusual syntax. Our `index.html` file is actually a template—a file that we can dynamically alter and inject content into. Our particular page is not very exciting: we tell Django to load a set of template tags (that is, `{% load staticfiles %}`). Later, we use some of these tags to generate proper URLs for files (that is, `{% static "<filename>" %}`) that can be found in our `example/static` folder.

Our handler functions are only useful when they return something meaningful. In our example, we use two different ways to return a value. For our `index.html` page, we use `render_to_response`. This method takes the name of a template in our `example/templates` folder, renders it, and returns it with the proper MIME type and HTTP status code. The other way to return a value is to return an HTTP response object via `HTTPResponse`, in which case we can set our own MIME type, response body, and HTTP status code as we like.

Using Flask/Bottle as a data provider

Many different micro frameworks have emerged to tackle small, specific problems that developers may have. In the Python world, there are two prominent examples: **Flask** and **Bottle**. Flask and Bottle are very similar, and so in this recipe, we examine how we can use either as a data provider for Highcharts.

Getting ready

First, we will need to set up Python using the following steps:

1. Download and install Python 2.7 (`http://www.python.org/getit/`).
2. Download and install Flask (`http://flask.pocoo.org`).
3. Download and install Bottle (`http://bottlepy.org/`).
4. Create a folder `flask_bottle` for our project.
5. Create a file `flask_bottle/bower.json` to list our JavaScript dependencies. The dependencies are as follows:

```
{
  "name": "highcharts-cookbook-chapter-8",
  "dependencies": {
    "jquery": "^1.9",
    "highcharts": "~3.0"
  }
}
```

6. Create a file `flask_bottle/.bowerrc` to configure where our JavaScript dependencies will be installed. The following code gives this location:

```
{
    "directory": "static/js"
}
```

How to do it...

To get started, follow the ensuing instructions:

1. Create a file `static/index.html` as follows:

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

```
        <div id='example'></div>
        <script src='./js/jquery/jquery.js'></script>
        <script src='./js/highcharts/highcharts.js'></script>

        <script type='text/javascript'>
            $(document).ready(function() {
                var options = {
                    chart: {
                        type: 'bar',
                        events: {
                            load: function () {
                                var self = this;
                                setInterval(function() {
                                    $.getJSON('/ajax/series',
                                    function(data) {
                                        var series = self.
                                        series[0];
                                        series.setData(data);
                                    });
                                }, 1000);
                            }
                        }
                    },
                    title: {
                        text: 'Using AJAX for polling charts'
                    },
                    series: [{
                        name: 'AJAX data (series)',
                        data: []
                    }]
                };
                $('#example').highcharts(options);
            });
        </script>
    </body>
</html>
```

2. Create a file `server.py`, and create a Flask instance as shown in the following code:

```
from flask import Flask

app = Flask(__name__)
```

3. Create a route to serve our data as shown in the following code:

```
from flask import Flask

app = Flask(__name__)

@app.route('/ajax/series')
def series():
    return None
```

4. Return some data in our route using the following code:

```
from flask import Flask, Response
import json
import random

app = Flask(__name__)

@app.route('/ajax/series')
def series():
    series = []
    for x in xrange(0,11):
        series.append({
            'y': random.randint(0,100)
        })

    return Response(json.dumps(series), mimetype='application/
    json')
```

5. If the file is run as an executable, run our application in the debug mode as shown in the following code:

```
from flask import Flask, Response
import json
import random

app = Flask(__name__)

@app.route('/ajax/series')
def series():
    series = []
    for x in xrange(0,11):
        series.append({
            'y': random.randint(0, 100)
        })
```

```
        return Response(json.dumps(series), mimetype='application/
        json')

    if __name__ == '__main__':
        app.run(debug=True)
```

6. Start the server using the following command:

   ```
   python server.py
   ```

7. Visit `http://localhost:5000/static/index.html`.

How it works...

`app.route` is a decorator we can apply to Python methods to handle HTTP requests. In addition to specifying the path, we can also specify which HTTP methods we want to handle (for example, `app.route('/path', methods=['GET'])` to just handle the `GET` requests).

Flask will automatically call server files from the static folder at `http://localhost:5000/static`, which is why we did not need to add any special configuration to see `index.html`.

There's more...

In Bottle, we'll find the code quite similar to the following:

```
from bottle import run, route, static_file, request, response
import json
import random

@route('/ajax/series')
def series():
    response.content_type = 'application/json'
    response.status = 200
    series = []
    for x in xrange(0,11):
        series.append({
            'y': random.randint(0, 100)
        })

    return json.dumps(series)

# Static files
# e.g. HTML page and JavaScript
```

```
@route('/')
def index():
    return static_file('index.html', root='.')

@route('/static/<filename:path>')
def index(filename):
    return static_file(filename, root='static')

run(host='localhost', port=5000)
```

The following are the more significant differences:

- Bottle doesn't use a core application object; it just handles everything globally via various methods imported from Bottle.
- We can return just about anything from a method and Bottle will serve it. If we want to set some property on the response though, we need to import `response` and alter that.
- Bottle has specific methods for serving static files.

Integrating with Backbone

In addition to server-side micro frameworks, a number of client-side micro frameworks have also appeared. Many aim to provide a simple means to send data from end-to-end with a clear separation of concerns. In this recipe, we'll take a look at Backbone, specifically at integration of Backbone with its models (an abstraction of our interface with a backend) and collections (a means of managing multiple models).

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe from *Chapter 1*, *Getting Started with Highcharts*.

We will, however, need to make some small changes as shown in the following steps:

1. Create a folder `backbone` for our project, and set up a basic project in that folder as described previously.
2. Modify `backbone/bower.json` as follows:

```
{
  "name": "highcharts-cookbook-chapter-8",
  "dependencies": {
    "highcharts": "~3.0",
    "jquery": "^1.9",
```

```
        "underscore": "^1.5",  // Used by Backbone, functional
         programming
        "backbone": "~1.1.0",  // Model-view-controller library for
         JavaScript
        "backbone.localStorage": "~1.1.7"  // Handles persistence
         using brower's localStorage
    }
}
```

3. Install our dependencies from within the `backbone` folder using the following command:

```
bower install
```

How to do it...

To get started, follow the ensuing instructions:

1. Create our skeleton HTML file `backbone/index.html`, as shown in the following code:

```
<!doctype html>
<html>
    <head>
        <script src='./bower_components/jquery/jquery.js'></
        script>
        <script src='./bower_components/highcharts/highcharts.src.
        js'></script>
        <script src='./bower_components/highcharts/highcharts-
        more.src.js'></script>
        <script src='./bower_components/underscore/underscore.
        js'></script>
        <script src='./bower_components/backbone/backbone.js'></
        script>
        <script src='./bower_components/backbone.localStorage/
        backbone.localStorage.js'></script>
        <script src='./example.js'></script>

        <style type='text/css'>
            #monsters {
                list-style: none;
                padding: 0px;
            }
            #monsters li {
                margin-bottom: 5px;
            }
```

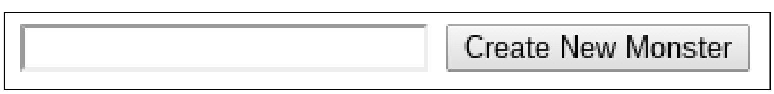
```
            #monsters .card {
                clear: both;
                padding: 10px;
                border: 1px solid #aaa;
            }
            #monsters .card .graph {
                float: left;
                width: 300px;
                height: 300px;
            }
            #monsters .card .stats .row > label {
                display: inline-block;
                width: 100px;
                font-size: 1.0em;
                text-transform: capitalize;
            }
            #monsters .card .stats .row > input {
                width: 50px;
                padding: 5px;
                text-align: right;
                font-size: 1.2em;
            }
        </style>
    </head>

    <body>
        <div id='main'>
            <input id='monster-name' type='text' />
            <input id='new-monster' type='button' value='Create
             New Monster' />
            <ul id='monsters'>
            </ul>
        </div>
    </body>
</html>
```

2. Create a template for our model instances as shown in the following code:

```
<!doctype html>
<html>
    <head>
        <!-- … -->
    </head>
```

```
    <body>
        <div id='main'>
            <!-- … -->
        </div>

        <script type='text/template' id='monster-template'>
            <div class='card'>
                <div class='graph' id='monster-<%= stats.name %>'>
                </div>
                <div class='stats'>
                    <h2><%= stats.name %> <input class='feed'
                    type='button' value='Feed Me!' /></h2>

                    <% var keys = ['hp', 'attack', 'defense',
                    'special_attack', 'special_defense', 'speed'];
                    %>
                    <% var key_stats = _.chain(stats).pick(keys).
                    value(); %>
                    <% _.each(key_stats , function(value, key) {
                    %>
                    <div class='row'>
                        <label for='<%= key %>'><%= key %></label>
                        <input type='text' name='<%= key %>'
                         value='<%= value %>' />
                    </div>
                    <% }); %>
                </div>
            </div>
        </script>
    </body>
</html>
```

On the page, observe the **Create New Monster** button we have created so far:



3. Create a file `backbone/example.js`, and include an immediate function using jQuery as shown in the following code:

```
$(function() {
});
```

4. Define a Backbone model as shown in the following code:

```
$(function() {
    var Monster = Backbone.Model.extend({
        defaults: {
            name: 'Unknown',
            height: 0.0,
            weight: 0.0,
            hp: 0,
            attack: 0,
            defense: 0,
            special_attack: 0,
            special_defense: 0,
            speed: 0
        }
    });
});
```

5. Define a Backbone collection using the following code:

```
$(function() {
    //...
    var MonsterCollection = Backbone.Collection.extend({
        model: Monster ,
        localStorage: new Backbone.LocalStorage("example")
    });
});
```

6. Create a `MonsterCollection` as shown in the following code:

```
$(function(){
    // ...
    var MonsterCollection = Backbone.Collection.extend(/* … */);

    var Monsters = new MonsterCollection();
});
```

7. Create a view for our `Monster` model as shown in the following code. This will handle interaction with our model as well as make any changes to the UI for a model.

```
$(function() {
    // ...
    var MonsterView = Backbone.View.extend({
        tagName: 'li',

        template: _.template($('#monster-template').html()),
```

```
        initialize: function () {
            this.listenTo(this.model, 'change', this.render);
        },

        render: function() {
            this.$el.html(this.template({
                'stats': this.model.toJSON()
            }));

            return this;
        }
    });
});
```

8. Create a view for our application in general, as shown in the following code:

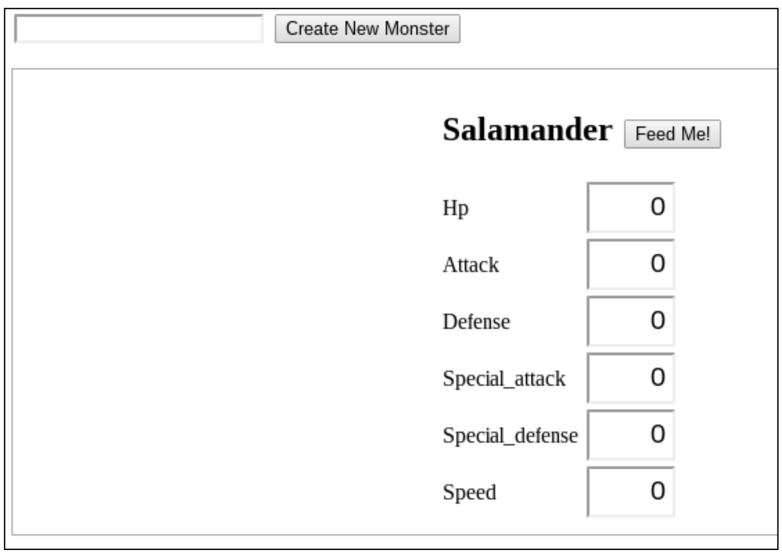
```
$(function() {
    // ...
    var AppView = Backbone.View.extend({
        el: $('#main'),

        events: {
            'click #new-monster': 'createMonster',
            'keypress #monster-name': 'createMonster'
        },

        initialize: function() {
            this.listenTo(Monsters, 'add', this.addMonster);
        },

        createMonster: function(event) {
            var $name = $('#monster-name');

            if (event.type === 'keypress' && event.keyCode !== 13)
            {
                return;
            }
            if (!$name.val()) {
                return;
            }

            Monsters.create({name: $name.val()});

            $name.val('');
        },
```

```
        addMonster: function(monster) {
            var view = new MonsterView({model: monster});
            this.$("#monsters").append(view.render().el);
        },
    });
});
```

9. Create an instance of our application view using the following code:

```
$(function() {
    // ...
    var App = new AppView();
});
```

On the page, observe the progress on our template monster view so far:



10. Modify `MonsterView.render` to render our Highcharts as shown in the following code:

```
    var MonsterView = Backbone.View.extend({
        // …
        chartOptions: {
            chart: {
                polar: true,
                type: 'line'
            },
```

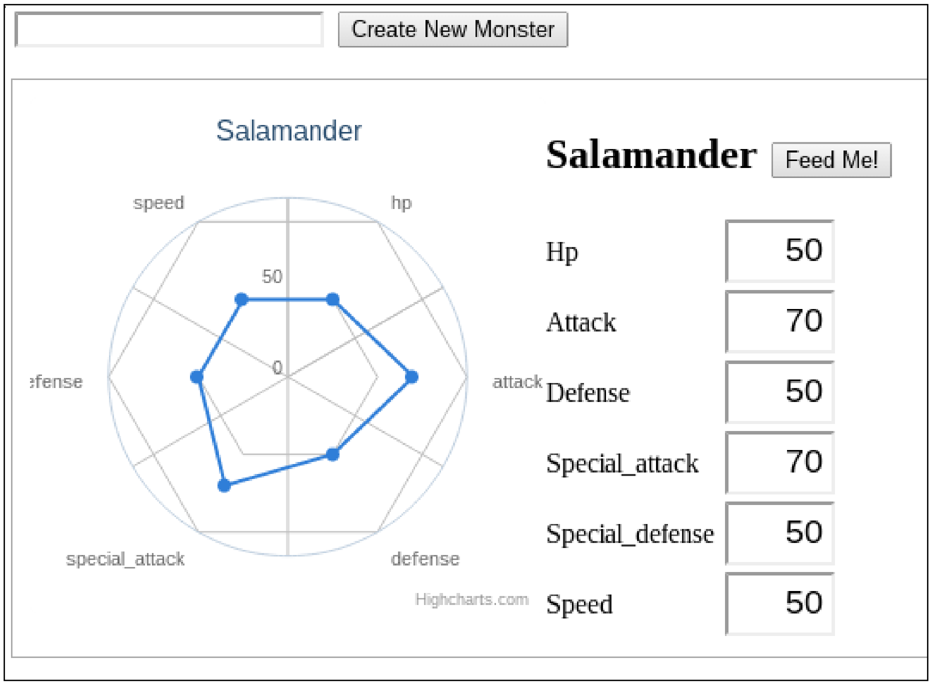
```
        legend: {
            enabled: false
        },
        xAxis: {
            tickmarkPlacement: 'on',
            labels: {
                overflow: 'justify',
                style: {
                    fontSize: '10px'
                }
            }
        },
        yAxis: {gridLineInterpolation: 'polygon', min: 0},
        tooltip: {
            pointFormat: '<span style="color:series.
            color}">{series.name}: <b>{point.y:,.0f}</b><br/>'
        }
    },

    render: function() {
        this.$el.html(this.template({
            'stats': this.model.toJSON()
        }));

        // get the key stats from the model
        var key_stats = this.model.pick([
            'hp',
            'attack',
            'defense',
            'special_attack',
            'special_defense',
            'speed'
        ]);

        // turn those stats into a highcharts data series
        var series = _.chain(key_stats).map(function(value,
        key) {
            return parseInt(value, 10);
        });

        // extend the default options
        var options = _.extend(this.chartOptions, {
            title: {
```

```
                    text: this.model.get('name')
                },
                xAxis: {
                    categories: _.chain(key_stats).keys().value()
                },
                series: [{
                    data: series.value()
                }]
            });

            this.$('.graph').highcharts(options);

            return this;
        },
    });
```

11. Modify `MonsterView` to help us to handle changing different stats using the following code:

```
    var MonsterView = Backbone.View.extend({
        // ...
        events: {
            'keyup .row input': 'statChange'
        },

        statChange: function(event) {
            // figure out which element this is from
            var $target = this.$(event.target)

            // get the text value from the element
            var value = parseInt($target.val(), 10);
            var key = $target.attr('name');
            if (!_.isNumber(value) || _.isNaN(value) || value < 0)
            {
                return;
            }

            // update the underlying model
            this.model.set(key, value);
        }
});
```

On the page, create a new monster and assign it some stats to see our chart change, as shown in the following screenshot:



How it works...

Our models and collections are fairly straightforward, especially as there is presently no backend interaction: we only store information in a local copy of our models. Our models only contain some simple information and a set of default values, and our collection only has one really interesting piece, the key model which defines what type of models this collection holds.

Most of the interesting work in our example comes from our two views. Backbone is able to handle events after we've defined either a template (as we did in our `MonsterView`) or an existing element with `el` (as we did in `AppView`). After that, we're able to register events via the `events` object using jQuery-like selectors of the form `'<event> <selector>': 'function_name'`.

Our views also have a few important methods. The `initialize` method is called immediately after a view is created, which allows us to set up any additional event handling we need to do. For example, we use `this.listenTo` to listen certain events on a view's model to tell the view to re-render itself.

Using AngularJS data bindings and controllers

Interactivity, especially responsiveness, in web applications has become very important. One important concept to fostering further responsiveness is the idea of two-way binding, where changes made to a model or a view are automatically made to the other or vice versa. This recipe looks at how we can leverage data bindings in AngularJS and integrate them with Highcharts.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe from *Chapter 1*, *Getting Started with Highcharts*.

We will, however, need to make some small changes as shown in the following steps:

1. Create a folder `angularjs` for our project, and set up a basic project in that folder as described previously.
2. Modify `angularjs/bower.json` as follows:

```
{
  "name": "highcharts-cookbook-chapter-8",
  "dependencies": {
    "highcharts": "~3.0",
    "jquery": "^1.9",
    "angular": "^1.2",  // JavaScript framework
    "highcharts-ng": "~0.0.4"  // Highcharts adapter for AngularJS
  }
}
```

3. Install our dependencies from within the `backbone` folder using the following command:

```
bower install
```

How to do it...

To get started, follow the ensuing instructions:

1. Create a skeleton HTML file `angularjs/index.html` as shown in the following code:

```
<!doctype html>
<html>
    <head>
        <script src='./bower_components/jquery/jquery.js'></
        script>
        <script src='./bower_components/highcharts/highcharts.src.
        js'></script>
        <script src='./bower_components/angular/angular.js'></
        script>
        <script src='./bower_components/highcharts-ng/dist/
        highcharts-ng.js'></script>
        <script src='./example.js'></script>

        <style type='text/css'>
            .example {
                clear: both;
            }

            .container {
                width: 300px;
                float: left;
            }

            .controls {
                padding: 20px;
            }

            input.title {
                width: 150px;
            }
```

```
                input {
                    width: 50px;
                    padding: 6px;
                }

                span {
                    display: inline-block;
                    width: 30px;
                }
            </style>
        </head>

        <body ng-app="example">
            <div ng-controller='ctrl'>
                <div class='example'>
                    <div class='container'>
                        <highchart id='example1' config='config'></
                        highchart>
                    </div>
                    <div class='controls'>
                        <input ng-model='config.title.text'
                        class='title'><br/>
                        <span>Yes:</span> <input type="number" ng-
                        model='config.series[0].data[0]'><br/>
                        <span>No:</span> <input type="number" ng-
                        model='config.series[0].data[1]'>
                    </div>
                </div>
            </div>
        </body>
    </html>
```

2. Create our controller `angularjs/example.js` using the following code:

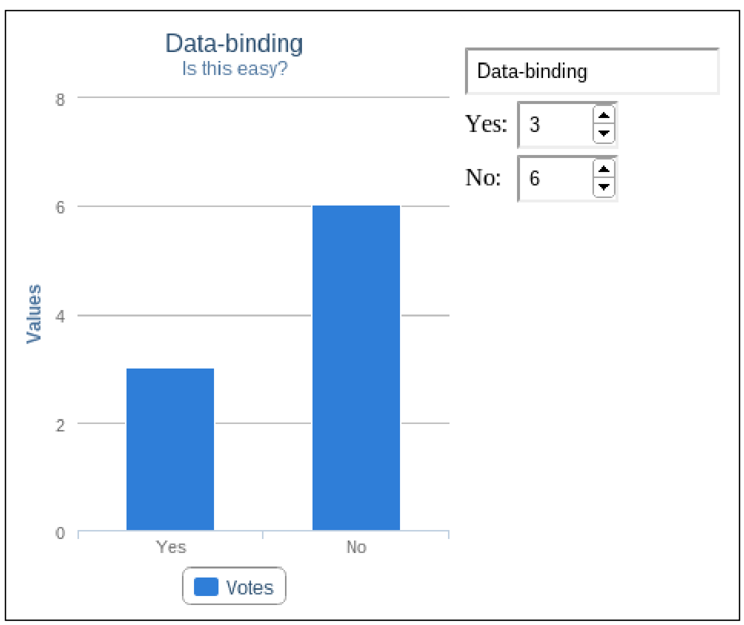
```
var app = angular.module('example', ['highcharts-ng']);

app.controller('ctrl', function($scope){
    $scope.config = {
        options: {
            chart: {
                type: 'column'
            },
        },
```

```
            xAxis: {
                categories: ['Yes', 'No']
            },
            series: [{
                name: 'Votes',
                data: [0, 0]
            }],
            title: {
                text: 'Data-binding Example'
            },
            subtitle: {
                text: 'Is this easy?'
            },
            credits: {
                enabled: false
            },
            loading: false
        };
    });
```

The following is the output from the controller:

3. Change the title text as shown in the following screenshot:

4. Adjust the voting buttons as shown in the following screenshot:



How it works...

This AngularJS example is deceptively simple in that a lot is happening *for free*. Let's start by looking at our `example.js` file.

First, we create a new application via `angular.module(app_name, dependencies)`. This, in conjunction with the accompanying `ng-app="app_name"` in the HTML page, binds the application in JavaScript to the HTML code; if there was no `ng-app` attribute on the page, our JavaScript just wouldn't do anything.

Once we've done that, we can add controllers to our application. Controllers are responsible for handling any model and action that take place in the scope of that controller. That's why, when we create our controller (for example, `app.controller(control_name, callback)`), everything has access to the controller's scope.

In our case, we define a `config` model, which is basically our chart options. Models don't really require anything special to define; they're just objects.

The magic of handling the two-way bindings is handled by directives. Directives are attributes (for example, `ng-model`) or elements (`highchart`) that watch for changes in the view or model and propagate the changes in both directions. Directives are one of the most complicated parts of AngularJS.

There is a lot more to AngularJS than what has been described here; it is well worth reading the documentation (`http://angularjs.org`) to get a better understanding of how it works.

Using NodeJS for chart rendering

In previous chapters, we've seen different ways of rendering charts on the server side. In this recipe, we look at how we can do the same using NodeJS.

Getting ready

To begin, we'll need to set up a few NodeJS dependencies, as shown in the following steps:

1. Create a folder `nodejs-rendering`.
2. Download and install NodeJS (`http://nodejs.org/download/`)
3. Create a file `nodejs-rendering/package.json`, and fill it with the following content:

```
{
  "name": "highcharts-cookbook-nodejs",
  "description": "An example application for using highcharts with
  nodejs",
```

```
    "version": "0.0.1",
    "private": true,
    "dependencies": {
      "express": "3.4",
            "node-highcharts-exporter": "0.0.5"
    }
  }
```

4. Install our dependencies (for example, express) using the following command:

```
npm install
```

How to do it...

Let's begin. Perform the following steps:

1. Create `nodejs-rendering/app.js`, and set up express as shown in the following code:

```
var express = require('express');
var app = express();

app.use(express.bodyParser());

app.post('/generate', function(req, response) {
    // future code
});
app.listen(8888);
```

2. Include `node-highcharts-exporter` library using the following code:

```
app.use(express.json());

var nhe = require('node-highcharts-exporter');

// ...
```

3. Call out to the `node-highcharts-exporter` library using the following code:

```
var nhe = require('node-highcharts-exporter');

app.post('/generate', function(req, response) {
    nhe.exportChart(req.body, function(error, chart) {
        // future code
    });
});
// ...
```

4. Open the image and serve it if it is available, otherwise, show an error, as shown in the following code:

```
var nhe = require('node-highcharts-exporter');
var fs = require('fs')

app.post('/generate', function(req, response) {
    nhe.exportChart(req.body, function(error, chart) {
        if (error) {
            console.log(error)
            response.writeHead(500);
            response.end(error);
        } else {
            var img = fs.readFileSync(chart.filePath);
            console.log(req.body.type)
            response.writeHead(200, {'Content-Type': req.body.type
            });
            response.end(img, 'binary');
        };
    });
});

app.listen(8888);
```

5. On the page using this export functionality, set `exporting.url` to point at our NodeJS app and include the exporting module, as shown in the following code:

```
<script type='text/javascript' src='path/to/highcharts/modules/
exporting.js'></script>

// ...
var options = {
    // …
    exporting: {
         enabled: true,
         url: 'http://localhost:8888/generate'
    }
}
$('#container').highcharts(options);
```

How it works...

Our NodeJS application hands over most of the work to the `node-highcharts-exporter` library. This library has one very important method `exportChart`, which takes two arguments: an object containing the parameters to generate the chart (`req.body` in our case) and a callback for when the chart has been rendered. The callback will give us an error message and the generated chart as its arguments, and so we just need to serve `img` (a stream of binary data, or `error`, JSON containing information about the error) as a response in our `/generate` function.

9
Extending Highcharts

In this chapter, we will cover the following recipes:

- Wrapping existing functions
- Creating new chart types
- Creating your own Highcharts extension
- Adding new functions to your extension
- JSHinting your code
- Unit - testing your new extension
- Packaging your extension
- Minifying your code

Introduction

We've been working mostly with Highcharts as a library to create a specific sort of chart or connect data to a certain source. However, there may be occasions where we would prefer to do more than just that. We may want to expand on the Highcharts library to add our own functionality. This chapter shows how we can expand on the core library to add our own improvements.

Wrapping existing functions

Sometimes, we don't want to change the entire way that a method works, just a part of it. In these cases, it would be fantastic if we could just wrap a method or a property by calling our code either before or after the desired method. This recipe will show how we can wrap one such method in Highcharts, `drawGraph`, but a similar technique can be applied to wrap other methods.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe from *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, perform the following steps:

1. Create an immediate function that takes `Highcharts` as an argument, as shown in the following code:

```
(function(H) {
    // Later code goes here
}(Highcharts));
```

2. Within that function, call the `wrap` function and give it an object and method to wrap, as shown in the following code:

```
(function(H) {
    H.wrap(
        H.Series.prototype,
        'drawGraph',
        function(original_fn) {
            var original_arguments = Array.prototype.slice.call(
                arguments, 1
            );
            original_fn.apply(this, original_arguments);
        }
    );
}(Highcharts));
```

3. Create a wrapping function to call code either before or after the wrapped method is called. In our example, we will just use `console.log` to show a message as shown in the following code:

```
(function(H) {
    H.wrap(
        H.Series.prototype,
        'drawGraph',
        function(original_fn) {
            var original_arguments = Array.prototype.slice.call(
                arguments, 1
            );
            console.log('Called before H.series.prototype.
drawGraph');
            original_fn.apply(this, original_arguments);
```

```
            console.log('Called after H.series.prototype.
drawGraph');
        }
    );
}(Highcharts));
```

4. Create a chart normally, as shown in the following code:

```
var options = {
    title: {
        text: 'Wrapping existing functions'
    },
    series: [{
        type: 'spline',
        name: 'Spline #1',
        data: [1,2,3,4]
    }]
};
```

5. Render the chart using the following code:

```
$('#container').highcharts(options);
```

How it works...

We execute all our calls to wrap in an immediate function (`(function(args){}(args))`) to ensure that any work that needs to be done happens before we render any charts, or before anything else has the chance to take place. This also has the benefit of not polluting the global scope (that is, our variables and functions will not have any side-effect on those found outside our immediate function); doing so is common practice.

Thanks to `Highcharts.wrap`, most of what we do is straightforward. `Highcharts.wrap` takes three arguments: the object to wrap, its method, and a function to call in its place. Then, when Highcharts goes to call the named method on that object, it instead calls our method. In fact, it will be called in the same scope, so calls to this will behave the same as if the original method were called.

In order to avoid breaking Highcharts, our wrapping function needs to do a few things. First, it must take the original function as an argument (`original_fn`). Secondly, if desired, the original function needs to be called at some point; in some cases, we may not want to call the original function, in which case we could omit the call. In our example, since we aren't modifying any arguments, we pass all the original arguments to the original function, `Array.prototype.slice.call(arguments, 1)`, that takes the list of arguments passed to our wrapping function and slices off the first (for example, `original_fn`), leaving us with the remaining arguments. Similarly, `original_fn.apply(this, original_arguments)` calls the wrapped function and makes sure it is called within the same scope.

Creating new chart types

We've worked with a variety of different chart types so far: column, bar, pie, and spline, to name a few. What if we could make our own chart type? As daunting as that may sound, this recipe looks at how we might make our own chart type, picto, which is like a column chart but uses images for bars.

> Much of how this recipe has been prepared depends on specifics of the Highcharts source code. This recipe was written assuming that Highcharts Version 3.0.7 is used. The information provided here may change depending on the version of Highcharts or the chart type.
>
> To make changes to other chart types, refer to the existing series types in the Highcharts source code.

How to do it...

To get started, perform the following steps:

1. Create an immediate function that takes Highcharts as an argument as shown in the following code:

```
(function(H) {
    // Future code goes here
}(Highcharts));
```

2. Create a wrapper function for `Highcharts.Renderer.prototype.image` as shown in the following code:

```
(function(H) {
    // Fix for incorrect image renderer function
    H.wrap(
        H.Renderer.prototype,
        'image',
        function(original_fn) {
            var passed_arguments,
                first_argument,
                original_arguments;

            original_arguments = Array.prototype.slice.call(
                arguments, 1
            );
            first_argument = original_arguments[0];
```

```
            if (typeof first_argument === 'object') {
                passed_arguments = [];
                passed_arguments.push(first_argument.source);
                passed_arguments.push(first_argument.x);
                passed_arguments.push(first_argument.y);
                passed_arguments.push(first_argument.width);
                passed_arguments.push(first_argument.height);

            } else {
                passed_arguments = original_arguments;
            }

            return original_fn.apply(this, passed_arguments);
        }
    );
}(Highcharts));
```

3. Create a copy, `PictoSeries`, of `ColumnSeries` as shown in the following code:

```
(function(H) {
    H.wrap(H.Renderer.prototype, 'image', function(original_fn) {
        // Wrapping code, as previously
    });
     var ColumnSeries = H.seriesTypes['column'];
     var PictoSeries = H.extendClass(ColumnSeries, {});
}(Highcharts));
```

4. Add a translate method to our `PictoSeries` to mix in our image attributes, as shown in the following code:

```
(function(H) {
    H.wrap(H.Renderer.prototype, 'image', function(original_fn) {
        // Wrapping code, as previously
    });
    var ColumnSeries = H.seriesTypes['column'];
    var PictoSeries = H.extendClass(ColumnSeries, {
        translate: function() {
            var series = this;

            // call translate normally
            ColumnSeries.prototype.translate.apply(this,
arguments);

            // Fix shape arguments
```

```
            H.each(series.points, function (point) {
                point.shapeType = 'image',
                point.shapeArgs = {
                    source: series.options.image,
                    unitHeight: series.options.unitHeight,
                    width: point.shapeArgs.width,
                    height: point.shapeArgs.height,
                    x: point.shapeArgs.x,
                    y: point.shapeArgs.y
                };
            });
        }
    });
}(Highcharts));
```

5. Add a `drawPoints` method to handle drawing multiple points. Note that most of this comes from the `ColumnSeries` method `drawPoints`, with some modifications as shown in the following code:

```
(function(H) {
    H.wrap(H.Renderer.prototype, 'image', function(original_fn) {
        // Wrapping code, as previously
    });
    var ColumnSeries = H.seriesTypes['column'];
    var PictoSeries = H.extendClass(ColumnSeries, {
        translate: function() {/* … as previously ... */},
        drawPoints: function() {
            var series = this,
                options = series.options,
                renderer = series.chart.renderer,
                shapeArgs;

                $.each(series.points, function (index, point) {
          var plotY = point.plotY,
                    graphic = point.graphic,
                    graphicArgs,
                    unitHeight,
                    remainder;

                if (plotY !== undefined && !isNaN(plotY) &&
point.y !== null) {
                    shapeArgs = point.shapeArgs;
                    unitHeight = shapeArgs.unitHeight || 1
```

```
                    if (graphic) {
                        stop(graphic);
                        graphic.animate(merge(shapeArgs));
                  } else {
                        for(var units=0; units < shapeArgs.height;
units += unitHeight) {
                            graphicArgs = $.extend({}, shapeArgs,
{
                                'height': unitHeight,
                                'y': shapeArgs.y + units
                            });

                            point.graphic = graphic =
renderer[point.shapeType](graphicArgs)
                                .attr(point.pointAttr[point.
selected ? 'select' : ''])
                                .add(series.group)
                                .shadow(options.shadow, null,
options.stacking && !options.borderRadius);
                        }
                    }

                } else if (graphic) {
                    point.graphic = graphic.destroy();
                }
            });
        }
    });
}(Highcharts));
```

6. Register `PictoSeries` as a new chart type as shown in the following code:

```
(function(H) {
    H.wrap(H.Renderer.prototype, 'image', function(original_fn) {
        // Wrapping code, as previously
    });
    var ColumnSeries = H.seriesTypes['column'];
    var PictoSeries = H.extendClass(ColumnSeries, {
        translate: function() {/* … as previously ... */}
    });
    H.seriesTypes.picto = PictoSeries;
}(Highcharts));
```

7. Define our chart options and make sure to set the chart type and the image, as shown in the following code:

```
var options = {
    title: {
        text: 'PictoChart!'
    },
    series: [{
        image: 'check.gif',
        type: 'picto',
         name: 'Picto #1',
         data: [1,2,3,4]
     }]
};
```

8. Render the chart using the following code:

```
$('#container').highcharts(options);
```

How it works...

Highcharts is able to determine which type of chart to render by the different series objects that are stored in `Highcharts.seriesTypes`. Many chart types are simply extensions of other chart types (for example, bar and column charts) whereas other chart types (for example, polar) are not. Our example is similar to bar and column charts in that we extend an existing series and make a small set of modifications, rather than implementing our own series class from scratch.

`Highcharts.extendClass` allows us to take an existing class, in this case `ColumnSeries`, and extend it. In our case, we do everything that a column chart would normally do, but we'll modify some existing methods to suit our needs.

We also wrapped `Highcharts.Renderer.prototype.image`. We do this so that the object we pass to the method can be unpacked into individual arguments; if we didn't, our images wouldn't have any dimensions and it would appear as though our chart was empty.

Creating your own Highcharts extension

In past examples, our code has been fairly ad hoc; we've created functions when necessary on a per-project basis. If we could compartmentalize our code, as the Highcharts library does, then we could leverage what we've learned in multiple projects or even share our changes. This recipe will look at how we can create our own library that builds off Highcharts.

How to do it...

To get started, perform the following steps:

1. Create a new file `myExtension.js`, and, in it, include an immediate function, as shown in the following code:

```
(function(w, H, $) {
}(window, Highcharts, jQuery));
```

2. Create a new object in the `window` scope, if one does not exist, as shown in the following code:

```
(function(w, H, $) {
    w.MyExtension = w.MyExtension || {};
}(window, Highcharts, jQuery));
```

3. Create a `Chart` function as shown in the following code:

```
(function(w, H, $) {
    var me;

    w.MyExtension = w.MyExtension || {};
    me = w.MyExtension;

    me.Chart = me.Chart || Highcharts.Chart;
}(window, Highcharts, jQuery));
```

4. On a page, create options for our chart as shown in the following code:

```
var options = {
    chart: {
        renderTo: 'container'
    },
    title: {
        text: 'Creating a chart extension'
    },
    series: [{
        name: 'Series #1',
        data: [1,2,3,4]
    }]
};
```

5. Render our chart using the following code:

```
var chart = new MyExtension.Chart(options);
```

How it works...

It is true that our extension does not appear to be doing much; our extension does the minimum possible to lay the foundation for our library.

Our immediate function ensures that everything that takes place within it doesn't leak out into the global scope. We pass in any libraries or variables that we require as well (for example the window object, Highcharts, and jQuery, for later).

Within our immediate function, we create our extension and attach it to the window object; if it already exists, we instead use this function (`w.MyExtension = w.MyExtension || {}`). In addition, we create a local reference (that is, `me`) for convenience, and any of the functions we attach to that local reference are exposed as public methods. We then create our own chart constructor that references Highcharts.

Adding new functions to your extension

In our last recipe, we set up a foundation for extension that at the moment, just aliases Highcharts. In this recipe, we will see how we can add functions to our extension to make it more useful.

How to do it...

To get started, perform the following steps:

1. Add some new variables to our extension, as shown in the following code:

```
(function(w, H, $) {
    var me,
        NAME,
        MAJOR,
        MINOR,
        PUBLISHED;

    w.MyExtension = w.MyExtension || {};
    me = w.MyExtension;

    // --- Private variables ---
    NAME = 'MyExtension';
    MAJOR = 1;
```

```
    MINOR = 0;
    PUBLISHED = new Date(2013, 11, 26);

    // --- Global functions ---
    me.Chart = me.Chart || Highcharts.Chart;
}(window, Highcharts, jQuery));
```

2. Create a `getVersionInfo` function, as shown in the following code:

```
(function(w, H, $) {
    var me,
        NAME,
        MAJOR,
        MINOR,
        PUBLISHED,
        formatDate;

    w.MyExtension = w.MyExtension || {};
    me = w.MyExtension;

    // --- Private variables ---
    NAME = 'MyExtension';
    MAJOR = 1;
    MINOR = 0;
    PUBLISHED = new Date(2013, 11, 26);

    // --- Private function ---
    formatDate = function(d) {
        return "" + d.getFullYear() + "-" + d.getMonth() + "-" +
d.getDate();
    }

    // --- Global functions ---
    me.Chart = me.Chart || Highcharts.Chart;

    // Privileged functions
    me.getVersionInfo = function() {
        return ""
            + NAME + " - " + MAJOR + "." + MINOR
            + " (" + formatDate(PUBLISHED) + ")";
    };
}(window, Highcharts, jQuery));
```

3. Create a `SpiderWebChart` function as shown in the following code:

```
(function(w, H, $) {
    var me,
        NAME,
        MAJOR,
        MINOR,
        PUBLISHED,
        formatDate;

    w.MyExtension = w.MyExtension || {};
    me = w.MyExtension;

    // --- Private variables ---
    NAME = 'MyExtension';
    MAJOR = 1;
    MINOR = 0;
    PUBLISHED = new Date(2013, 11, 26);

    // --- Private function ---
    formatDate = function(d) {
        return "" + d.getFullYear() + "-" + d.getMonth() + "-" +
d.getDate();
    }

    // --- Global functions ---
    me.Chart = me.Chart || Highcharts.Chart;

    me.SpiderWebChart = function (options) {
        // create options if they don't exist
        var modifiedOptions = options || {};

        // create a chart option if it does not exist
        modifiedOptions.chart = modifiedOptions.chart || {};
        modifiedOptions.chart.polar = true;

        // create an xAxis option if it does not exist
        modifiedOptions.xAxis = modifiedOptions.xAxis || {};
        modifiedOptions.xAxis.tickmarkPlacement = 'on';
        modifiedOptions.xAxis.lineWidth = 0;
```

```
            // create a yAxis option if it does not exist
            modifiedOptions.yAxis = modifiedOptions.xAxis || {};
            modifiedOptions.yAxis.gridLineInterpolation = 'polygon';
            modifiedOptions.yAxis.lineWidth = 0;

            new me.Chart(modifiedOptions);
        };

        // Privileged functions
        me.getVersionInfo = function() {
            return ""
                + NAME + " - " + MAJOR + "." + MINOR
                + " (" + formatDate(PUBLISHED) + ")";
        };
    }(window, Highcharts, jQuery));
```

4. On our page, define our chart options as shown in the following code:

```
var options = {
    chart: {
        renderTo: 'container'
    },
    title: {
        text:  MyExtension.getVersionInfo()
    },
    xAxis: {
        categories: ["Strength", "Speed", "Defense"]
    },
    series: [{
        name: 'Fighter',
        data: [10, 1, 5],
        pointPlacement: 'on'
    }, {
        name: 'Rogue',
        data: [5, 10, 1],
        pointPlacement: 'on'
    }]
};
```

5. Render our chart using the following code:

```
var chart = new MyExtension.SpiderWebChart(options);
```

How it works...

Our extension now contains private, public, and privileged elements. We've already seen how we can create public functions and attributes by adding on to the `w.MyExtension` object. Creating private variables and functions is also simple; any functions or variables declared in our immediate function will not be accessible outside the immediate function.

The exception to this is privileged functions. Privileged functions are public functions that can access private variables. Normally, a JavaScript function can only access variables defined in the scope of the function. However, if we have an inner function (for example, `getVersionInfo`) that accesses variables from an outer function (for example, our immediate function), we can create a closure. When we have a closure, the inner function maintains a reference to variables from the outer function, even though that function has already been executed; that's basically how privileged functions work.

JSHinting your code

Now that we have a fledgling extension, we may want to remove any extraneous code or potential errors. A common way to do this is to use a lint program (that is, a program designed to find suspicious language usage). This recipe will examine how we can use `JSHint` to find errors in our extension.

Getting ready

Install JSHint (`http://www.jshint.com/install/`). If you already have NodeJS and `npm` set up, you can just install it by running the following command:

```
npm install jshint
```

> It may be necessary to install JSHint globally (rather than locally, which is the default method). To install JSHint globally, run the following command:
>
> `npm install jshint -g`
>
> It is also possible to avoid installing JSHint entirely by using the JSHint website at `http://www.jshint.com`.

How to do it...

1. Run JSHint using the following command:

   ```
   jshint myExtension.js
   ```

2. Correct the errors/warnings listed.

How it works...

By default, JSHint will run some simple checks: looking for missing semicolons, undeclared variables, and so on. We can customize how strict JSHint is by using either comments or a `.jshintrc` file in the same directory. For example, if we want to tell JSHint that the code will be running in a browser in strict mode, we could add the following comment at the beginning of our document:

```
/* jshint strict: true, browser: true */
```

We can also have JSHint tell us more about potential problems with the `--show-non-errors` flag. There are many, many options though, so it's worth looking at the documentation to determine which options you may want to enable or disable (`http://www.jshint.com/docs/options/`).

Unit testing your new extension

Unit testing is a common means to provide some certainties that the code is working correctly. While unit testing, it is important that we only test our code (there is no need to test other people's code). In this recipe, we'll be unit testing our extension using QUnit.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe from *Chapter 1*, *Getting Started with Highcharts*.

We will also need to make a small change to our `bower.json` file as shown in the following code:

```
{
  "name": "highcharts-cookbook-chapter-9",
  "dependencies": {
    "highcharts": "~3.0",
    "jquery": "^1.9",
    "qunit": "~1.14.0"
  }
}
```

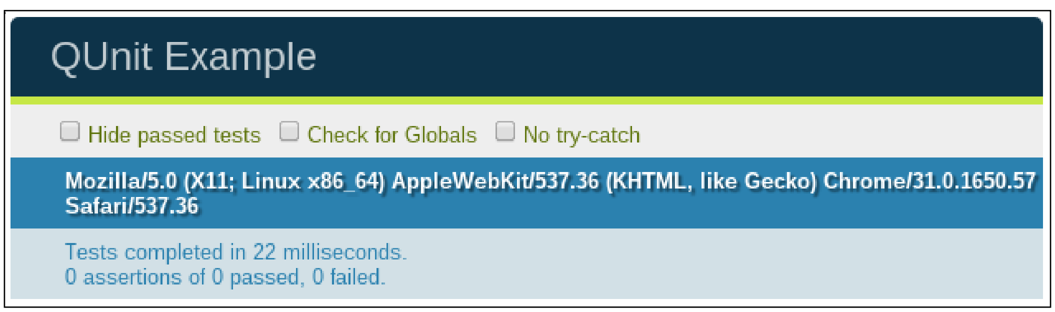
How to do it...

To get started, perform the following steps:

1. Create a file, `test.html`, and then open it in a browser, as shown in the following code:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>QUnit Example</title>
  <link rel="stylesheet" href="./bower_components/qunit/qunit/
qunit.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="./bower_components/qunit/qunit/qunit.js"></script>
  <script src="./bower_components/jquery/jquery.js"></script>
  <script src="./bower_components/highcharts/highcharts.js"></
script>
  <script src="myExtension.js"></script>
  <script src="tests.js"></script>
</body>
</html>
```

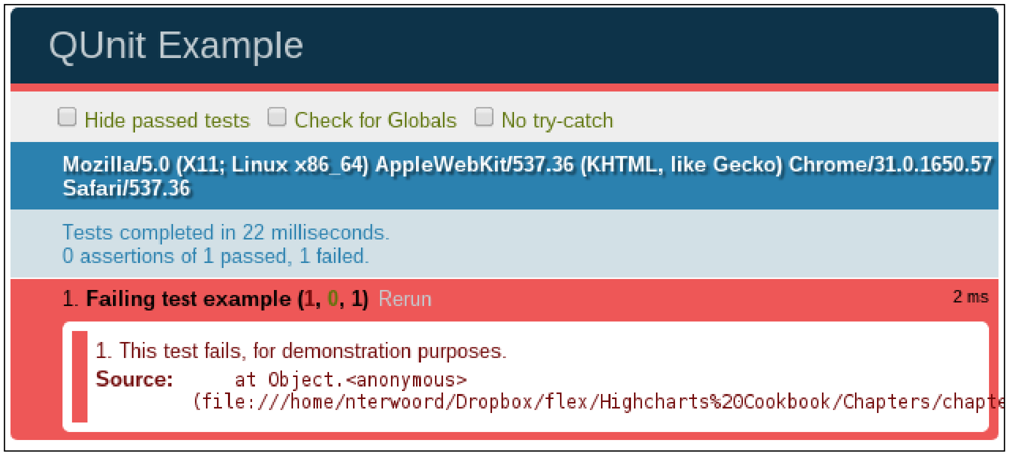
   After the code is executed, we get the following screenshot:



2. Create a file, `test.js`, as shown in the following code:

```
test("Failing test example", function() {
    ok(false, "This test fails, for demonstration
      purposes.");
});
```

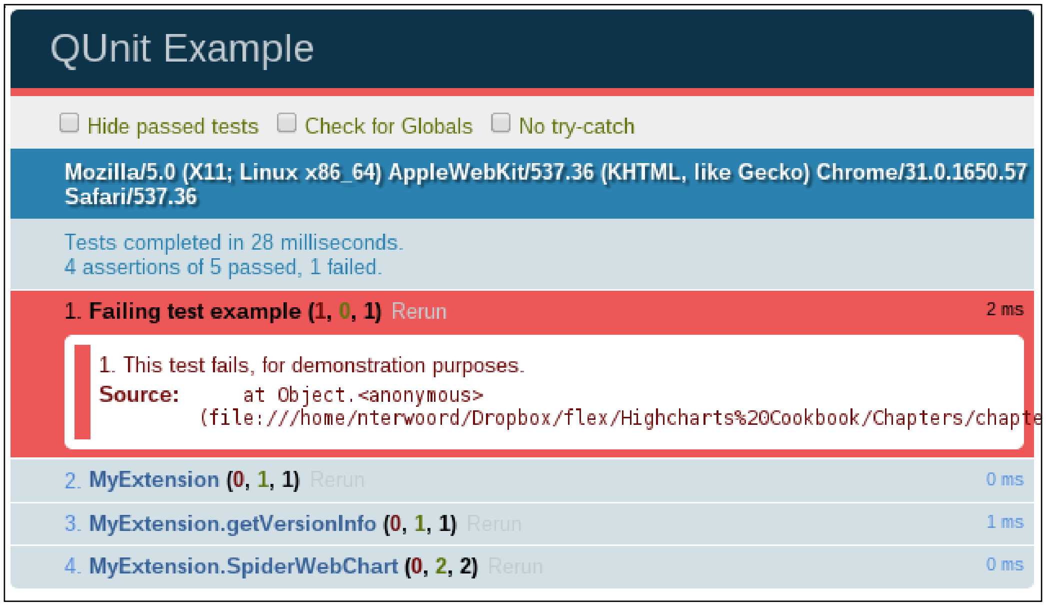
3. Open `test.html` to see the failing test run, as shown in the following screenshot:



4. Add some additional tests as shown in the following code:

```
test("Failing test example", function() {
    ok(false, "This test fails, for demonstration purposes.");
});

test("MyExtension", function() {
    ok(MyExtension, "MyExtension doesn't exist.");
});

test("MyExtension.getVersionInfo", function() {
    var actual, expected;

    actual = MyExtension.getVersionInfo();
    expected = "MyExtension - 1.0 (2013-11-26)";
    equal(actual, expected, "Wrong version info.")
});

test("MyExtension.SpiderWebChart", function() {
    var actual, expected;

    ok(MyExtension.SpiderWebChart, "SpiderWebChart doesn't
exist.")

    throws(function(){
        new MyExtension.SpiderWebGraph();
    }, "SpiderWebChart needs arguments.");
});
```

5. Open `test.html` to see other tests succeed as shown in the following screenshot:



How it works...

Our `test.html` page sets up a test runner that does all the hard work of making it possible to run our tests. After `qunit.js` has been included, we include all of our source files and then our `tests.js` file.

QUnit will automatically execute any test function we've included on the page. The `test` function is just a simple function for grouping tests. Within that method, we can execute any code that we want to execute, then use various assertions, such as ok, equal, or throws, to verify that some expected value matches our actual value.

More information on QUnit can be found in the documentation at `http://api.qunitjs.com`.

Packaging your extension

Our extension as implemented works, but it isn't as polished as the Highcharts library itself. For example, we can't create a chart using jQuery as we previously could. This recipe will look at how we can package our extension as a jQuery plugin.

Getting ready

To set up a basic extension, refer to the *Creating your own Highcharts extension* recipe presented earlier in this chapter.

How to do it...

To get started, perform the following steps:

1. In `myExtension.js`, create a new function as shown in the following code:

```
(function(w, H, $) {
    // … previous code
    $.fn.spiderwebchart = function() {
        return this;
    };
}(window, Highcharts, jQuery));
```

2. Get our chart options from the arguments, as shown in the following code:

```
(function(w, H, $) {
    // … previous code
    $.fn.spiderwebchart = function() {
        var args = arguments,
            options;

        options = args[0];

        return this;
    });
}(window, Highcharts, jQuery));
```

3. Add code to create the chart, as shown in the following code:

```
(function(w, H, $) {
    // … previous code
    $.fn.spiderwebchart = function() {
        var args = arguments,
      options,
      chart;

    options = args[0];

    // Create the chart
    options.chart = options.chart || {};
    options.chart.renderTo = this[0]; // use just the first
      element
    chart = new me.SpiderWebChart(options);

        return this;
    });
}(window, Highcharts, jQuery));
```

4. Define our chart options, as shown in the following code:

```
var options = {
    title: {
        text: 'SpiderWebGraph'
    },
    series: [{
        name: 'Bar #1',
        data: [1,2,3,4]
    }]
};
```

5. Render our chart using the following code:

```
$('#container').spiderwebchart(options);
```

How it works...

jQuery plugins are created by adding functions onto `$.fn`. By using the arguments option, our plugin can accept an arbitrary number of arguments (though, in this case, we only use the first argument). Within our plugin, this is a collection of elements that the jQuery selector returns. For our plugin, we just take the first element from the selector and use that element as the path to which the chart should render.

Minifying your code

We now have something we may want to share with the world, something that we may even want to deploy to one of our own applications. As our extension may be unnecessarily large now, it might take longer for slower devices to run our application. A technique that is often used to handle this issue is minification. This recipe looks at how we can reduce the size of our extension using UglifyJS.

Getting ready

Install UglifyJS (`http://lisperator.net/uglifyjs/`). If you already have NodeJS and `npm` set up, you can just install it as follows:

```
npm install uglify-js
```

> Depending on your system, it may be necessary to install UglifyJS globally. If this is the case, you can install UglifyJS globally with the following command:
>
> ```
> npm install uglify-js -g
> ```

How to do it...

1. Minify our extension using the following code:

   ```
   uglifyjs myExtension.js -o myExtension.min.js
   ```

How it works...

UglifyJS is a JavaScript parser, compressor, and beautifier. This means that is can take a JavaScript file as an input and compress it by shortening variables and functions. In our example, we do this by calling the `uglifyjs` command with the source as the first argument (that is, `myExtension.js`) specifying the output file using the `-o` flag, and listing the desired output file (for example, `myExtension.min.js`).

10
Math and Statistics

In this chapter, we will cover the following recipes:

- Graphing equations
- Showing descriptive statistics with box plots
- Plotting distributions with jStat
- Displaying experimental data with scatter plots
- Displaying percentiles with area range graphs

Introduction

One of the most common uses of charts is to display mathematical formulas. Anything ranging from business to education, simple to complex models can be demonstrated with the help of charts. This chapter looks at a few ways in which we can leverage math and statistics in our Highcharts.

Graphing equations

Mathematics classes are filled with many equations, and students are often required to learn how to plot these equations. In this recipe, we'll look at how it is possible to create a simple application to graph equations.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

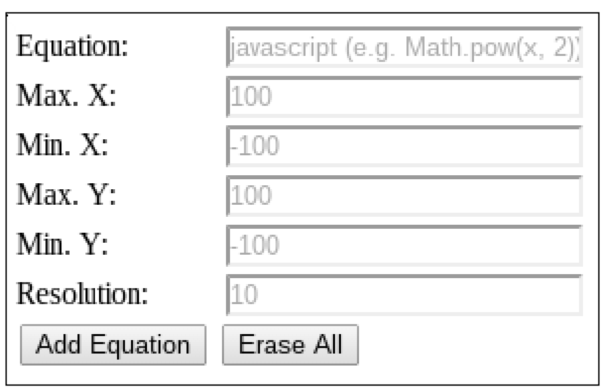
How to do it...

To get started, perform the following steps:

1. Create input fields for our graph's parameters and observe the page, as shown in the following code:

```
<div id='container'></div>
<label for='equation'>Equation:</label>
<input type='text' id='equation' placeholder='javascript (e.g.
Math.pow(x, 2))'/><br/>

<label for='maxX'>Max. X:</label>
<input type='text' id='maxX' placeholder='100'/><br/>

<label for='minX'>Min. X:</label>
<input type='text' id='minX' placeholder='-100'/><br/>

<label for='maxY'>Max. Y:</label>
<input type='text' id='maxY' placeholder='100'/><br/>

<label for='minY'>Min. Y:</label>
<input type='text' id='minY' placeholder='-100'/><br/>

<label for='resolution'>Resolution:</label>
<input type='text' id='resolution' placeholder='10'/>

<input type='button' id='addSeries' value='Add Equation'/>
<input type='button' id='removeSeries' value='Erase All'/>
```

The result of this code is shown in the following screenshot:



2. Create a `dataFromEquation` function to generate data for our series as follows:

```
var dataFromEquation = function(config) {
  var config = config || {},
    equation = config.equation || function(x) {
      return x;
    },
    minX = parseInt(config.minX, 10) || -100,
    maxX = parseInt(config.maxX, 10) || 100,
    minY = parseInt(config.minY, 10) || -100,
    maxY = parseInt(config.maxY, 10) || 100,
    resolution = parseInt(config.resolution, 10) || 10,
    data = [];


  var data = [];

  for(var x = minX; x <= maxX; x+=resolution) {
    data.push({
      'x': x,
      'y': equation(x)
    })
  }

  return data;
};
```

3. Create an `addSeries` function that will add an equation to the chart, as shown in the following code:

```
var addSeries = function() {
  var equationStr = $('#equation').val(),
    minX = $('#minX').val(),
    maxX = $('#maxX').val(),
    minY = $('#minY').val(),
    maxY = $('#maxY').val(),
    resolution = $('#resolution').val();

  var equation = new Function('x', 'return ' + equationStr);

  $('#container').highcharts().addSeries({
    name: equationStr,
    data: dataFromEquation({
      minX: minX,
      maxX: maxX,
```

```
        minY: minY,
        maxY: maxY,
        resolution: resolution,
        equation: equation
      })
    });
  };
```

4. Create a `removeSeries` function that will remove all equations from the chart, as shown in the following code:

```
var removeSeries = function() {
  var chart = $('#container').highcharts();

  while(chart.series.length > 0) {
    chart.series[0].remove(true);
  }
};
```

5. With the help of the following code, attach handlers as necessary:

```
$('#addSeries').click(addSeries);
$('#removeSeries').click(removeSeries);
```

6. Define the chart options as follows:

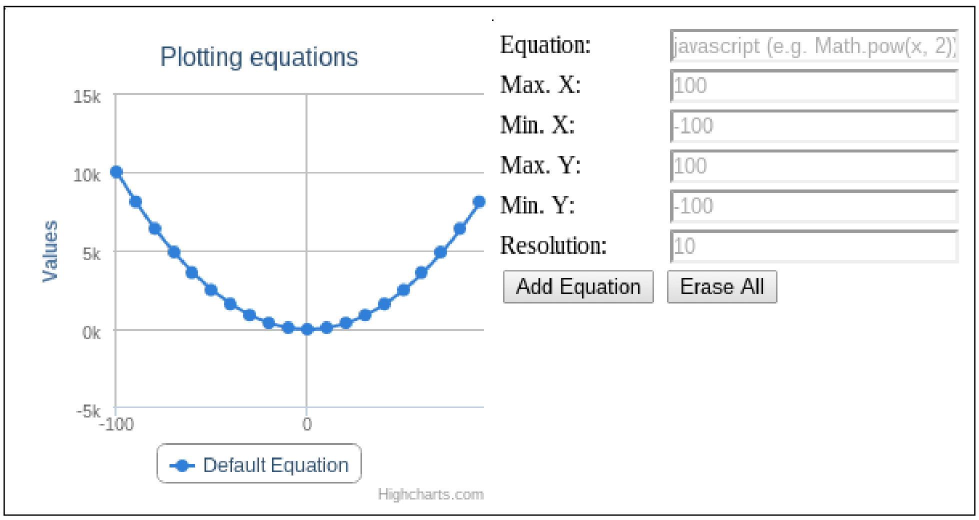
```
var options = {
  chart: {
    zoomType: 'xy',
    type: 'spline'
  },
  title: {
    text: 'Plotting equations'
  },
  xAxis: {
    gridLineWidth: 1,
  },
  yAxis: {
    gridLineWidth: 1,
  },
```

```
    series: [{
      name: 'Default Equation',
      data: dataFromEquation({
        equation: function(x) {
          return Math.pow(x, 2);
        }
      })
    }]
  };
```

7. Render our chart with the following code:

   ```
   $('#container').highcharts(options);
   ```

   The resultant chart is displayed as follows:

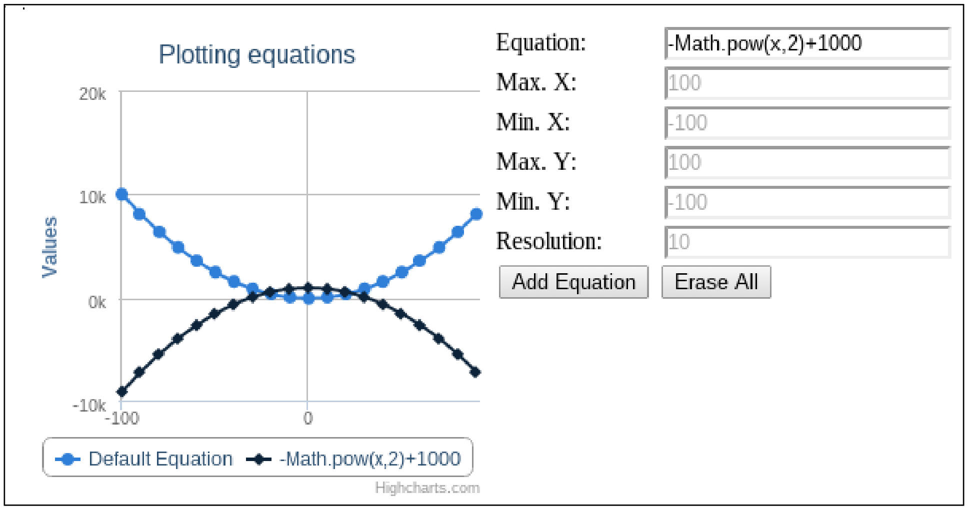


How it works...

Unfortunately, Highcharts cannot plot arbitrary functions, but our `dataFromEquation` function can generate a list of data points for us to plot (as seen in the previous screenshot). It takes a configuration object which allows us to set the maximum and minimum x and y values for our chart as well as the resolution (that is, the number of points to display).

Our `addSeries` function generates the configuration object for `dataFromEquation`. Equations, in our example, are just JavaScript code (for example, `Math.pow(x,4) + x*3 - 2`); due to this, we can use JavaScript's `Function` constructor to create a function that we will evaluate at every x value in `dataFromEquation`.

Lastly, we make sure that our chart has `chart.zoomType` set to `xy` so that we are able to zoom in anywhere we like on the chart. After we add an equation in our application (for example, `-Math.pow(x,2)+1000`), we have a result similar to the following screenshot:



Showing descriptive statistics with box plots

Few concepts in statistics are as well understood as descriptive statistics: the mean (average), minimum, maximum, and quartiles. Often, it is possible to condense all of this data into one simple graph. In this recipe, we will plot all of these different data points using a box plot, occasionally referred to as a box-and-whisker plot graph.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*. In addition to these instructions, box plot charts are only available by using `highcharts-more.js`. So, the following code can be used to ensure that we have included it on our page after we have included Highcharts:

```
<script src='./bower_components/highcharts/highcharts.js'></script>
<script src='./bower_components/highcharts/highcharts-more.js'></
script>
```

How to do it...

To get started, perform the following steps:

1. Define the chart options as follows:

```
var options = {
  chart: {
    type: 'boxplot'
  },
  title: {
    text: 'Box Plots'
  },
  series: [{
    name: 'Sample #1',
    data: [[0, 25, 50, 75, 100]]
  }]
};
```

2. Render the chart with the following code:

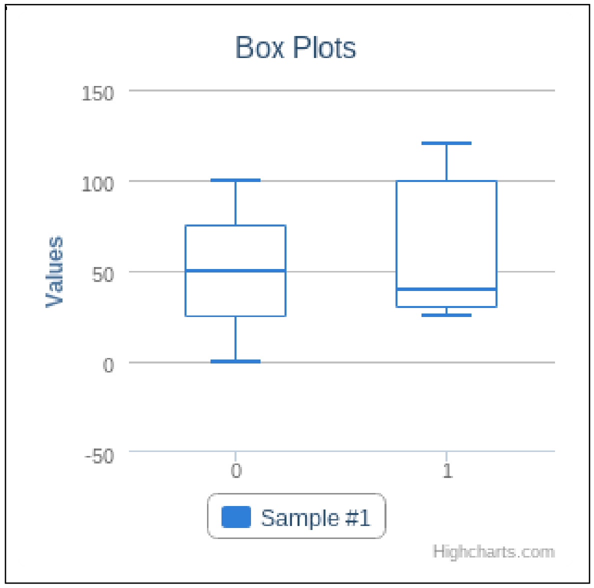
```
$('#container').highcharts(options);
```

How it works...

Provided that we add the proper type to our chart using `chart.type` or `series[0].type`, Highcharts will do the rest. In our example, we only showed one box plot, but we could have easily added more plots by adding them on to `series[0].data`. The following code is an example of how we can add more plots:

```
series: [{
  name: 'Sample #1',
  data: [
    [0, 25, 50, 75, 100],
    [25, 30, 40, 100, 120]
  ]
}]
```

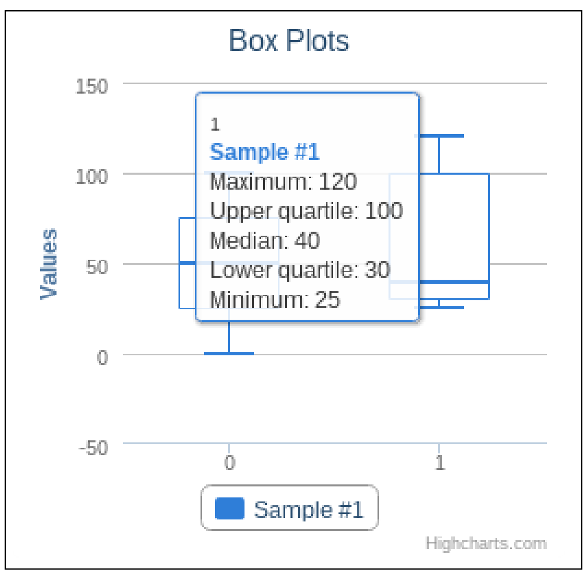
The resultant graph would look like the following screenshot:



It is worth noting that the format of the data as well. Box plots take data in a specific format. Each array must be of the following form:

```
[minimum, first quartile, mean, third quartile, maximum]
```

A sample box plot is shown as follows:



Plotting distributions with jStat

Some areas of statistics are more used and referred to than others; distributions, **probability density functions** (**PDFs**), and **cumulative density functions** (**CDFs**) are just some of the topics that tend to come up often. In this recipe, we will look at how we can use jStat to plot certain statistical distributions such as the normal distribution.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

We will also need to add some additional dependencies to our project in `bower.json` as shown in the following code:

```
{
  "name": "highcharts-cookbook-chapter-10",
  "dependencies": {
    "jquery": "^1.9",
    "highcharts": "~3.0",
    "rm-jstat": "~1.0.0",
    "underscore": "~1.6.0"
  }
}
```

Lastly, we need to include the following files on our page:

```
<script src='./bower_components/highcharts/highcharts.js'></script>
<script src='./bower_components/rm-jstat/jstat.js'></script>
<script src='./bower_components/underscore/underscore.js'></script>
```

How to do it...

To get started, perform the following steps:

1. Create a normal distribution and its points as follows:

   ```
   var range = jstat.seq(-5,5,100);
   var dNorm = jstat.dnorm(range, 0.0, 1.0);
   var dNormPairs = _.zip(range, dNorm);
   ```

2. Create a log-normal distribution and its points as follows:

```
var range = jstat.seq(-5,5,100);
var dNorm = jstat.dnorm(range, 0.0, 1.0);
var dNormPairs = _.zip(range, dNorm);

var dlNorm = jstat.dlnorm(range, 0.0, 1.0);
var dlNormPairs = _.zip(range, dlNorm);
```

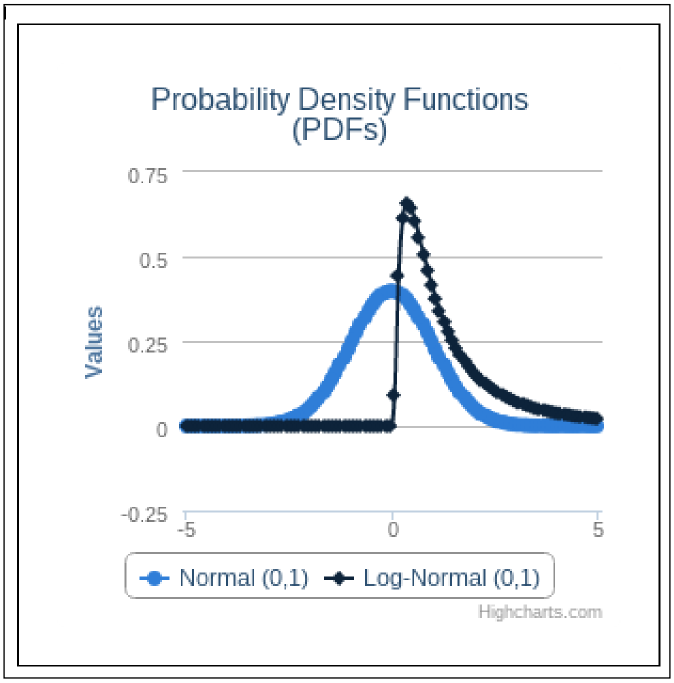
3. Define the chart options as follows:

```
var options = {
  chart: {
    type: 'spline',
    zoomType: 'xy'
  },
  title: {
    text: 'Probability Density Functions (PDFs)'
  },
  series: [{
    name: 'Normal (0,1)',
    data: dNormPairs
  }, {
    name: 'Log-Normal (0,1)',
    data: dlNormPairs
  }]
};
```

4. Render our chart with the following code:

```
$('#container').highcharts(options);
```

The resultant chart is displayed as follows:



How it works...

As we can see, jStat is able to generate the points for both our functions when given some information:

- `jstat.seq(min, max, points)`: This function generates a range that we can use when generating the points for the different distributions; in our case, we generate `100` points between `-5` and `5` on the x axis.
- `jstat.dnorm(range, mean, standard_deviation)`: This function generates the data points for a normal probability density function at the given values defined in the range. The `jstat.dlnorm(range, log_mean, log_standard_deviation)` function is similar to `jstat.dnorm(range, mean, standard_deviation)`, except that it takes `log(mean)` and `log(standard_deviation)` as its parameters.

Lastly, since these functions only give us values for the y axis, we need to map them to their x values, which we do by using `_.zip()`:

```
var x = [1, 2, 3];
var y = [4, 5, 6]
_.zip(x,y); // [[1, 4], [2, 5], [3, 6]]
```

Displaying experimental data with scatter plots

There are occasions where we don't know the relationship between our data points. In these instances, a scatter plot can show us the data, and patterns may emerge from the plotting of the points. This recipe will show us how we can display data using a scatter plot.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

We will also need to add some additional dependencies to our project in `bower.json`, as shown in the following code:

```
{
  "name": "highcharts-cookbook-chapter-10",
  "dependencies": {
    "jquery": "^1.9",
    "highcharts": "~3.0",
    "underscore": "~1.6.0"
  }
}
```

Lastly, we need to include these files on our page as follows:

```
<script src='./bower_components/highcharts/highcharts.js'></script>
<script src='./bower_components/underscore/underscore.js'></script>
```

How to do it...

To get started, perform the following steps:

1. Generate example data as shown in the following code:

```
var samples = 10;
var rangeFn = _.partial(_.random, 1, 100);
var experiment1 = _.zip(
  _.times(samples, rangeFn),
  _.times(samples, rangeFn)
);

var experiment2 = _.zip(
  _.times(samples, rangeFn),
  _.times(samples, rangeFn)
);
```

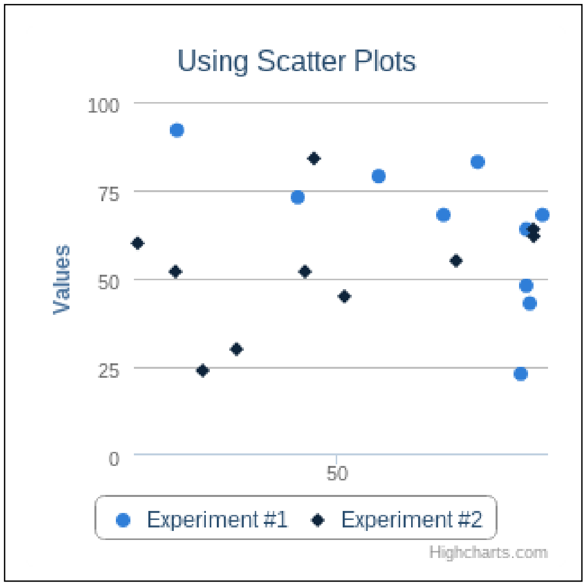
2. Define the chart options as follows:

```
var options = {
  chart: {
    type: 'scatter',
    zoomType: 'xy'
  },
  title: {
    text: 'Using Scatter Plots'
  },
  series: [{
    name: 'Experiment #1',
    data: experiment1
  }, {
    name: 'Experiment #2',
    data: experiment2
  }]
};
```

3. Render our chart with the following code:

```
$('#container').highcharts(options);
```

The resultant chart is displayed as follows:



How it works...

Creating a scatter plot is easy, as long as we perform the following steps:

1. Set `chart.type` or `series[0].type` to `scatter`.
2. Ensure that our series data is formatted as a series of x, y pairs, as either `[[x1, y1], [x2, y2] …]` or `[{x: x1, y: y1}, {x: x2, y: y2}, …]`.

As for our generated data, it may look complex, but it can be explained simply as follows:

- `_.partial(fn, arg1, ...)`: This function takes a function, applies arguments to it, then returns a new function. In this case, we take the function `_.random`, pass the arguments 1 and 100 to it, and return a function that calls it. In our example, it creates a function that returns a random number between 1 and 100.
- `_.times(n, fn)`: This function will call a function *n* times and add the result of each call to an array, which it returns. In our example, it creates an array of 10 random numbers between 1 and 100.
- `_.zip(arg1, …)`: This function takes an arbitrary number of arrays and zips them together (refer to the *Plotting distributions with jStat* recipe). In our case, we take our array of random numbers, and zip it together with another array of random numbers, yielding a bunch of random x, y pairs.

Displaying percentiles with area range graphs

Percentile data is interesting because it provides an insight into the distribution of the data. Different percentile can provide us with a greater picture than just the average. Take historical weather data for instance; a single day may be cold compared to the average of the temperature of various days but still within the 25th percentile (that is, it is still relatively common). This recipe will look at how we can use layered area range graphs to explain percentile data better.

Getting ready

To set up a basic page and installing jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

We will also need to add some additional dependencies to our project in `bower.json` as shown in the following code:

```
{
  "name": "highcharts-cookbook-chapter-10",
  "dependencies": {
    "jquery": "^1.9",
    "highcharts": "~3.0",
    "underscore": "~1.6.0"
  }
}
```

We will need to include `highcharts-more.js` as well as the following files on our page:

```
<script src='./bower_components/highcharts/highcharts.js'></script>
<script src='./bower_components/highcharts/highcharts-more.js'></
script>
<script src='./bower_components/underscore/underscore.js'></script>
```

Lastly, include the following weather data on our page:

```
var highs = [{
  'time': 1386306000000,
  'max': 17,
  '90%': 8,
  '75%': 5,
  '50%': 2,
  '25%': -1,
```

```
      '10%': -3,
      'min': -4
    }, {
      'time': 1386392400000,
      'max': 17,
      '90%': 8,
      '75%': 5,
      '50%': 2,
      '25%': -1,
      '10%': -4,
      'min': -5
    }, {
      'time': 1386478800000,
      'max': 10,
      '90%': 8,
      '75%': 5,
      '50%': 2,
      '25%': -1,
      '10%': -4,
      'min': -6
    }];

    var lows = [{
      'time': 1386306000000,
      'max': 3,
      '90%': 2,
      '75%': 0,
      '50%': -4,
      '25%': -7,
      '10%': -10,
      'min': -14
    }, {
      'time': 1386392400000,
      'max': 3,
      '90%': 2,
      '75%': 0,
      '50%': -4,
      '25%': -8,
      '10%': -11,
      'min': -11
    }, {
      'time': 1386478800000,
```

```
    'max': 5,
    '90%': 2,
    '75%': 0,
    '50%': -4,
    '25%': -8,
    '10%': -11,
    'min': -15
  }];
```

How to do it...

To get started, perform the following steps:

1. Define a function for the maximum and minimum values of our high and low weather data points as follows:

   ```
   var getExtremes = function(item) {
     return [
       item['time'],
       item['min'],
       item['max']
     ];
   };
   var highLines = _.map(highs, getExtremes);
   var lowLines = _.map(lows, getExtremes);
   ```

2. Define a function for the 10th and 90th percentiles of our high and low weather data points as follows:

   ```
   var highPercentiles = function(item) {
     return [
       item['time'],
       item['10%'],
       item['90%']
     ];
   };
   var highOuter = _.map(highs, highPercentiles);
   var lowOuter = _.map(lows, highPercentiles);
   ```

3. Define a function for the 25th and 75th percentiles of our high and low weather data points as follows:

   ```
   var q1q3 = function(item) {
     return [
       item['time'],
       item['25%'],
   ```

```
    item['75%']
  ];
};
var highInner = _.map(highs, q1q3);
var lowInner = _.map(lows, q1q3);
```

4. Define the chart options as follows:

```
var options = {
  chart: {
    type: 'arearange',
   zoomType: 'xy'
  },
  title: {
    text: 'Displaying Percentile Data'
  },
  xAxis: {
    type: 'datetime'
  },
  tooltip: {
    crosshairs: true,
    shared: true,
    valueSuffix: 'C'
  }
};
```

5. Add the extreme values to the graph as follows:

```
tooltip: {/* … */}
series: [{
  id: 'highs',
  name: 'Highs',
  data: highLines,
  fillOpacity: 0.1,
  color: '#ff0000',
  zIndex: 0
}, {
  id: 'lows',
  name: 'Lows',
  data: lowLines,
  fillOpacity: 0.1,
  color: '#0000ff',
  zIndex: 3
}]
```

6. Add our 10th and 90th percentiles to the graph as follows:

```
series: [{
  id: 'highs',
  name: 'Highs',
  data: highLines,
  fillOpacity: 0.1,
  color: '#ff0000',
  zIndex: 0
}, {
  name: '10% - 90%',
  linkedTo: 'highs',
  data: highOuter,
  lineWidth: 0,
  fillOpacity: 0.1,
  color: '#ff0000',
  zIndex: 1
}, {
  id: 'lows',
  name: 'Lows',
  data: lowLines,
  fillOpacity: 0.1,
  color: '#0000ff',
  zIndex: 3
}, {
  name: '10% - 90%',
  linkedTo: 'lows',
  data: lowOuter,
  lineWidth: 0,
  fillOpacity: 0.1,
  color: '#0000ff',
  zIndex: 4
}]
```

7. Add our 25th and 75th percentiles to the graph as follows:

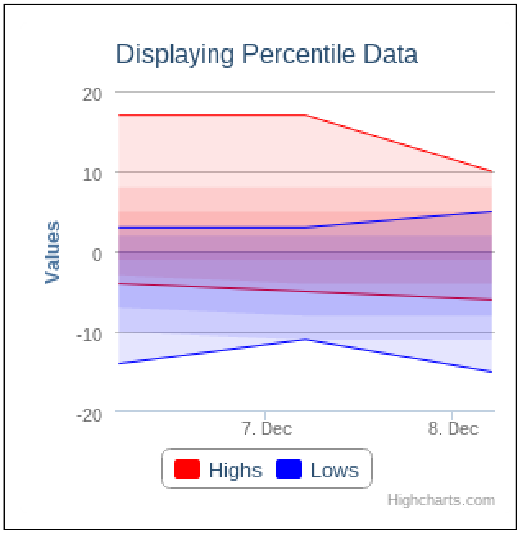
```
series: [{
  id: 'highs',
  name: 'Highs',
  data: highLines,
  fillOpacity: 0.1,
  color: '#ff0000',
  zIndex: 0
}, {
```

```
    name: '10% - 90%',
    linkedTo: 'highs',
    data: highOuter,
    lineWidth: 0,
    fillOpacity: 0.1,
    color: '#ff0000',
    zIndex: 1
  }, {
    name: '25% - 75%',
    linkedTo: 'highs',
    data: highInner,
    lineWidth: 0,
    fillOpacity: 0.1,
    color: '#ff0000',
    zIndex: 2
  }, {
    id: 'lows',
    name: 'Lows',
    data: lowLines,
    fillOpacity: 0.1,
    color: '#0000ff',
    zIndex: 3
  }, {
    name: '10% - 90%',
    linkedTo: 'lows',
    data: lowOuter,
    lineWidth: 0,
    fillOpacity: 0.1,
    color: '#0000ff',
    zIndex: 4
  }, {
    name: '25% - 75%',
    linkedTo: 'lows',
    data: lowInner,
    lineWidth: 0,
    fillOpacity: 0.1,
    color: '#0000ff',
    zIndex: 5
  }]
```

8. Render the chart with the following code:

```
$('#container').highcharts(options);
```

The resultant chart is displayed as follows:



How it works...

As we can see, the combination of the overlapping high and low values gives us a clearer view of where the high and low values are concentrated. Darker areas are where a greater percentage of our data lies.

The `_.map(iterable, fn)` function calls `fn` on each element in `iterable` and returns an array of the results of each call. Most of our calls were made to just partition the data into different subsets.

Once we set `chart.type` or `series[0].type` to `arearange`, Highcharts expects our data to be in the following format:

```
[x, low, high]
```

When we create our series, most of our options are merely for styling (for example, `lineWidth: 0` removes lines, `fillOpacity: 0.1` determines how opaque our colors are, and `color` sets the color of the fill area).

However, for our main series, we've defined the ID fields, and in our other series, we've defined the `linkedTo` fields referring to these IDs. This is done so that even though we have multiple series in our chart, our percentile data can appear as though it is a part of a series in the legend, which means that we can show or hide all high/low data at once.

Lastly, we use `zIndex` to determine the order in which the different series should appear so that we can layer them properly. Elements with a higher value for `zIndex` appear on top of elements that have a lower value for `zIndex`.

11
System Integration

In this chapter, we will cover the following recipes:

- Exploring hard drive usage
- Understanding CPU and memory usage graphs
- Showing Git commits by contributor
- Showing Git commits over time

Introduction

Highcharts can be very powerful on its own, but it is only as useful as the data we can obtain; the more data sources we can integrate with Highcharts, the more useful it can be. One such data source that can be useful to integrate with is the system itself, whether that be our own computer or some remote server. In this chapter, we will look at how we can use Highcharts along with Node.js to integrate with certain systems.

> The recipes in this chapter are highly server-side dependent; we will need to have Node.js set up and installed. For instructions on installing Node.js, visit `http://nodejs.org/download`.

Exploring hard drive usage

The hard drive is something that we use often, and many tools exist for exploring it. It is often easy to get simple data about a hard drive (for example, how much space is available or used) but not as easy to visualize that data, especially if that data is only available when logged on to a remote machine. In this recipe, we will look at how we can leverage Node.js and explore hard drive usage using Highcharts.

Getting ready

Create a folder `nodejs` for all of the files in this recipe; all instructions for this recipe will assume that we are operating from this folder, unless specified otherwise.

To set up a basic page and install jQuery and Highcharts, refer to the *Getting ready* section of the *Creating your first chart* recipe from *Chapter 1*, *Getting Started with Highcharts*.

We will also need to perform some steps to ensure our page and Node.js are set up correctly. They are as follows:

1. Include Underscore as one of our dependencies in `bower.json`, as shown in the following code:

   ```
   {
     "name": "highcharts-cookbook-chapter-11",
     "dependencies": {
       "jquery": "^1.9",
       "highcharts": "~3.0",
       "underscore": "~1.6.0"
     }
   }
   ```

2. Install our JavaScript dependencies using the following command:

   ```
   bower install
   ```

3. Create a file `package.json` in the same folder as `bower.json` for our Node.js dependencies, as shown in the following code:

   ```
   {
     "name": "highcharts-cookbook-chapter-11",
     "version": "0.0.0",
     "dependencies": {
       "gift": "~0.1.1",
       "underscore": "~1.5.2",
       "express": "~3.4.7"
     }
   }
   ```

4. Install our Node.js dependencies using `npm`:

```
npm install
```

5. Create a basic Node.js application `app.js` to serve our files, as shown in the following code:

```
var express = require('express');
var app = express();
app.use(express.json());
app.use(express.static(__dirname));

app.listen(8888);
console.log('Listening on port 8888');
```

How to do it...

To get started, perform the following steps:

1. Create a handler to obtain a directory listing, as shown in the following code:

```
app.use(express.static('static'));
app.post('/directory', function(request, response) {
});
app.listen(8888);
```

2. Get a list of files for a given directory using the following code:

```
var fs = require('fs');
app.post('/directory', function(request, response) {
    // request.body → {'path': '/some/directory'}
    var dir = request.body.path;

    // Get a list of files synchronously
    var file_list = fs.readdirSync(dir);
});
```

3. Convert the list of files into a usable format, as shown in the following code:

```
var fs = require('fs');
var path = require('path');
var underscore = require('underscore');
app.post('/directory', function(request, response) {
    // request.body → {'path': '/some/directory'}
    var dir = request.body.path;
```

```
    // Get a list of files synchronously
    var file_list = fs.readdirSync(dir);

    // Generate file statistics
    var files = underscore.map(file_list, function(file) {
        var filepath = path.join(dir, file);
        var file_obj = fs.lstatSync(filepath);
        var is_directory = file_obj.isDirectory();
        var size;
        var type;

        if (is_directory) {
            type = 'directory';
            size = 0;
        } else {
            type = 'file';
            size = file_obj.size;
        }


        // Note: Sizes are in bytes
        return {
            'name': file,
            'path': filepath,
            'type': type,
            'size': size
        }
    });
});
```

4. Group, sort, and return the list of files and directories, as shown in the following code:

```
// ...
app.post('/directory', function(request, response) {
    // … from previous step
    /*
        [{file}, …, {directory}, …]
        →
        {file: [files], directory: [files]}
    */
    var objects = underscore.groupBy(files, function(elem) {return
elem.type;})

    var sorted_files = underscore.sortBy(objects.file, function(x)
{return x.size;}).reverse();
```

```
    var sorted_dirs = underscore.sortBy(objects.directory,
function(x) {return x.name;});

    response.json({
        files: sorted_files,
        directories: sorted_dirs
    });
});
// ...
```

5. Include some controls for loading a directory, as shown in the following code:

```
    <body>
        <div id='container'></div>
        <label for='directory'>Initial Directory:</label>
        <input type='text' name='directory' id='directory' />
        <input type='button' name='go' id='go' value='Explore' />
        <br/>
        <label for='directories'>Sub-directories:</label>
        <select name='directories' id='directories'>
        </select>

        <script src='./bower_components/jquery/jquery.js'></
script>
```

6. Create a function to handle drawing a chart for each directory, as shown in the following code:

```
$(document).ready(function() {
    var drawDirectoryChart = function(files, directory) {
        // Convert files to proper format
        var slices = _.map(files, function(item) {
            var sizeInMB = parseFloat((item.size / (1024 * 1024)).
toFixed(2));
            return [item.name, sizeInMB];
        });

        var options = {
            chart: {type: 'pie'},
            title: {text: directory || 'Explore Hard Drive
Usage'},
            series: [{name: 'Size (MB)', data: slices}]
        };

        $('#container').highcharts(options);
    };
});
```

7. Create a function for updating the list of subdirectories, as shown in the following code:

```
$(document).ready(function() {
    var drawDirectoryChart = function(files, directory) {
        // ...
    };

    var updateDirectorySelector = function (directories) {
        var $selector = $('#directories');
        $selector.empty();
        _.each(directories, function(directory) {
            var newElem = $('<option>', {
                value: directory.path,
                text: directory.name
            })
            $selector.append(newElem);
        });
    };
});
```

8. Create a function to get data from our server-side application, as shown in the following code:

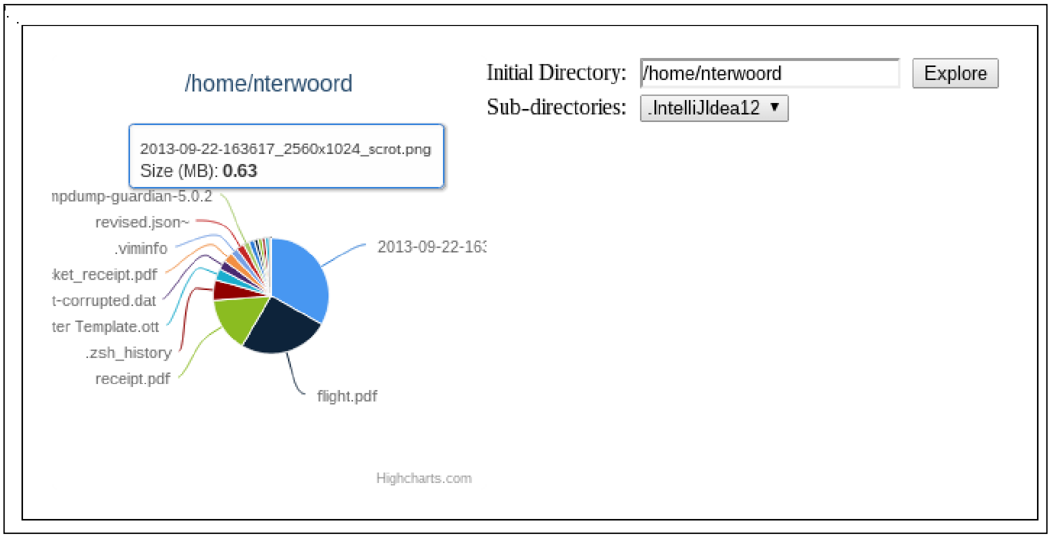
```
$(document).ready(function() {
    var drawDirectoryChart = function(files, directory) {
/*...*/};
    var updateDirectorySelector = function (directories) {
/*...*/};
    var getData = function (value) {
        var path = value || $('#directory').val();

        $.ajax({
            url: '/directory',
            data: JSON.stringify({'path': path}),
            type: 'POST',
            contentType: 'application/json; charset=utf-8',
            dataType: 'json'
        }).done(function(data) {
            drawDirectoryChart(data.files, path);
            updateDirectorySelector(data.directories);
        }).fail(function() {
            console.log(arguments);
        });
    };
});
```

9. Attach event handlers so that our chart updates when we make selections or enter a directory, as shown in the following code:

```
$(document).ready(function() {
    var drawDirectoryChart = function(files, directory) { /*...*/
};
    var updateDirectorySelector = function (directories)
{/*...*/};
    var getData = function (value) { /*...*/};
    $('#directory').on('keypress', function(e) {
        var key = e.keyCode || e.which;
        if (key == 13) {
            getData();
        }
    });

    $('#go').on('click', function() {getData();});

    $('#directories').on('change', function(e) {getData(this.
value);});
});
```

10. Start the NodeJS application, `node app.js`.
11. Visit `localhost:8888` in a browser window, and observe our initial application state:

12. Enter a directory on your computer in the **Initial Directory** field, click on the **Explore** button, and observe:



How it works...

On the server side, what we've done is simple and is mostly handled by `fs`—Node.js's file system library. The `fs.readdirSync` method takes the path to a directory and returns a listing of the files in the directory. In order to get the information we need, we go through each file listed using `underscore.map` and call `fs.lstatSync` to get details about the individual file objects. The last thing we do at the server side is take our list of file objects, separate out directories from files, and otherwise tidy up the value that we return.

Understanding CPU and memory usage graphs

When working with remote systems, it can be useful to know the status of the server. For example, it can be beneficial to know when a machine has run out of memory, or whether it is spending a lot of time doing computationally expensive work. In this recipe, we'll look at how we can use Node.js and Highcharts to understand a system's CPU and RAM usage graphs.

Getting ready

To set up a basic Node.js project and its dependencies, refer to the *Getting Ready* section of the *Exploring hard drive usage* recipe from this chapter.

How to do it...

To get started, perform the following steps:

1. Create a handler for obtaining CPU information using the following code:

```
app.use(express.static('static'));
app.get('/cpu/', function(request, response) {});
app.listen(8888);
```

2. Calculate and return the CPU usage, as shown in the following code:

```
var os = require('os');
var underscore = require('underscore');
var sum = function(p, n) { return p + n; };
app.get('/cpu/', function(request, response) {
    var cpus = os.cpus();

    var cpu_percentages = underscore.map(cpus, function(cpu, key)
{
        var values = underscore.values(cpu.times);
        var total = underscore.reduce(values, sum, 0);
        var idle = cpu.times.idle;

        return {
            'percent': parseFloat((((total - idle) * 100) /
total).toFixed(1)),
            'usage': (total - idle),
            'total': total,
            'time': timestamp.getTime(),
            'id': key
        }
    });
    response.json(cpu_percentages);
});
```

3. Create a handler to obtain RAM information using the following code:

```
app.get('/cpu/', function(request, response) {/*...*/});

app.get('/memory/', function(request, response) {});
```

4. Calculate RAM usage and return, as shown in the following code:

```
app.get('/memory/', function(request, response) {
    var timestamp = new Date();
    var free = os.freemem();
    var total = os.totalmem();
```

```
        var used = total - free;

        response.json([{
            'percent': parseFloat(((used * 100) / total).toFixed(1)),
            'usage': used,
            'total': total,
            'time': timestamp.getTime(),
            'id': 'RAM'
        }]);
    });
```

5. Meanwhile, on our page (`index.html`), create a function to add data to our chart, as shown in the following code:

```
$(document).ready(function() {
    var addOrUpdateSeries = function(chart, dataSeries, name) {
        var id = dataSeries.id,
            display = (name) ? '' + name + id : id,
            series = chart.get(id),
            percent = dataSeries.percent,
            timestamp = dataSeries.time,
            redraw = true,
            newSeries, existingPoints, point;

        if (series) {  // Update
            existingPoints = series.data.length
            point = { x: timestamp, y: percent }
            series.addPoint(point, redraw, existingPoints > 10);
        } else {  // New
            newSeries = {
                id: id,
                name: display,
                data: [{ x: timestamp, y: percent }]
            }

            chart.addSeries(newSeries);
        }
    };
});
```

6. Create a set of options common to our charts, as shown in the following code:

```
$(document).ready(function() {
    var addOrUpdateSeries = function(chart, dataSeries, name) {
/*...*/};
    var commonOptions = {
        chart: {type: 'spline',},
        xAxis: {type: 'datetime'},
        yAxis: {max: 100, min: 0},
        series: []
    }
});
```

7. Define options for our CPU chart, as shown in the following code:

```
$(document).ready(function() {
    var addOrUpdateSeries = function(chart, dataSeries, name) {
/*...*/};
    var commonOptions = {/*...*/}
    var cpuChart = _.extend({}, commonOptions);
    cpuChart.chart.events = {
        load: function () {
            var self = this;
            setInterval(function() {
                $.getJSON('/cpu/', function(data) {
                    for (var i=0; i < data.length; i++) {
                        addOrUpdateSeries(self, data[i], 'CPU #');
                    }
                });
            }, 1000);
        }
    };
    cpuChart.title = {text: 'CPU usage'};
});
```

8. Define options for our RAM chart, as follows:

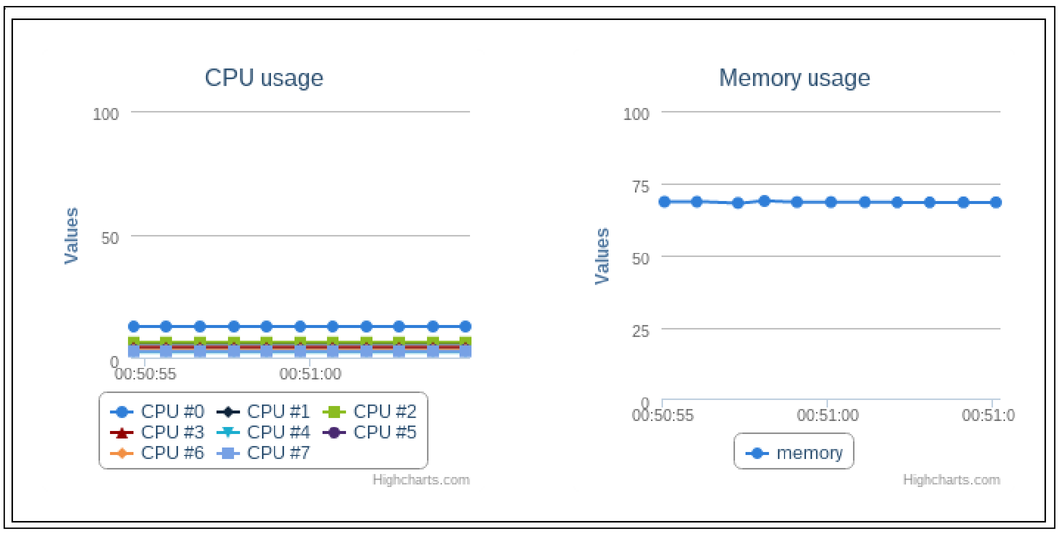
```
var ramChart = _.extend({}, commonOptions);
ramChart.chart.events = {
    load: function () {
        var self = this;
        setInterval(function() {
            $.getJSON('/memory/', function(data) {
                for (var i=0; i < data.length; i++) {
```

```
                        addOrUpdateSeries(self, data[i]);
                    }
                });
            }, 1000);
        }
    };
    ramChart.title = {
        text: 'RAM usage'
    };
```

9. Render our charts using the following code:

```
$('#cpu').highcharts(cpuChart);
$('#ram').highcharts(ramChart);
```

10. Visit `localhost:8888` and observe:



How it works...

Node.js's `os` library handles a lot of the heavy lifting in this recipe. It allows us to access information about memory (`os.freemem()` for free memory and `os.totalmem()` for total memory) as well as CPU information (`os.cpus()`).

Showing Git commits by contributor

Git is a fantastic tool that has done a lot to improve version control for developers. Out of the box, it is possible to get a lot of useful meta-information about a Git repository such as who has made the most commits. By default, Git provides this information as text, but what if we wanted to visualize the data differently? This recipe will examine how we can set up Node.js to obtain Git information and how we can use Highcharts to display it.

Getting ready

To set up a basic Node.js project and its dependencies, refer to the *Getting ready* section of the *Exploring hard drive usage* recipe from this chapter.

How to do it...

To get started, perform the following steps:

1. Create a handler to obtain the Git user information with the following code:

```
app.use(express.static('static'));
app.get('/git/users', function(request, response) {
});
app.listen(8888);
```

2. Create a connection to a repo, and get all commits from a particular branch (in this case, `master`), as shown in the following code:

```
var git = require('gift');
var repo = git('/full/path/to/repository');
app.get('/git/users', function(request, response) {
    repo.commits('master', -1, function(err, commits) {
    });
});
```

> The path provided to the `git` function can be absolute or relative. If the path is relative, it will be relative to the folder in which `app.js` is started.

3. Filter and return the commits by user, as shown in the following code:

```
var underscore = require('underscore');
app.get('/git/users', function(request, response) {
    repo.commits('master', -1, function(err, commits) {
        // Count the user
```

```
        user_counts = underscore.countBy(commits, function(commit)
{
            return commit.author.name;
        });

        // Convert to list of [name, commits]
        counts = underscore.pairs(user_counts);

        // Sort by most commits first
        sorted_counts = underscore.sortBy(counts, function(x) {
            return x[1];
        });

        // Respond with result
        response.json(sorted_counts);
    });
});
```

4. Create a function to get our Git data, as shown in the following code:

```
$(document).ready(function() {
    $.getJSON('/git/users', function(data) {
    });
});
```

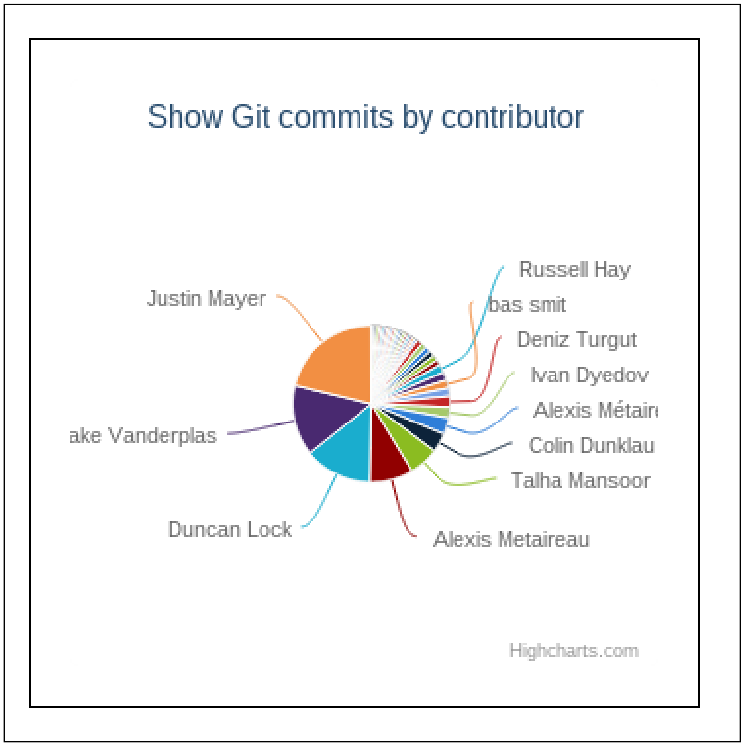
5. Create a set of options for our chart using the following code:

```
$(document).ready(function() {
    $.getJSON('/git/users', function(data) {
        var options = {
            chart: {type: 'pie'},
            title: {text: 'Show Git commits by contributor'},
            series: [{name: 'Commits',data: data}]
        };
    });
});
```

6. Render our chart, as follows:

```
$(document).ready(function() {
    $.getJSON('/git/users', function(data) {
        var options = {/* … */};
        $('#container').highcharts(options);
    });
});
```

Your desired output will look something like the following:



How it works...

Gift (`http://npmjs.org/package/gift`) is a Node.js library that wraps the Git command-line interface. We use it to obtain a reference to an existing repository (`repo = git('path/to/repo')`), and we can also use it to obtain a list of commits (`repo.commit(branch_name, number_of_commits, callback)`).

Showing Git commits over time

Displaying Git commits by users is interesting, but what if we wanted to observe when commits are taking place instead of observing who is making commits? Perhaps commits occur more likely at certain times of the day, or on certain days of the week? This recipe will look at how we can take the same Git data we have and instead show it as commits over time.

Getting ready

To set up a basic Node.js project and its dependencies, refer to the *Getting ready* section of the *Exploring hard drive usage* recipe from this chapter.

How to do it...

1. Create a handler to obtain Git timeline information as shown in the following code:

```
app.use(express.static('static'));
app.get('/git/timeline', function(request, response) {/*...*/});
app.listen(8888);
```

2. Create a connection to a Git repo, and get all commits from a particular branch (in this case, `master`) as shown in the following code:

```
var git = require('gift');
var repo = git('/full/path/to/repository');
app.get('/git/timeline', function(request, response) {
    repo.commits('master', -1, function(err, commits) {
    });
});
```

> The path provided to the `git` function can be absolute or relative. If the path is relative, it will be relative to the folder in which `app.js` is started.

3. Filter and return the commits by user, as shown in the following code:

```
var underscore = require('underscore');
app.get('/git/timeline', function(request, response) {
    repo.commits('master', -1, function(err, commits) {
        // Group commits by day
        commits_per_day = underscore.countBy(commits,
function(commit) {
            var date = commit.authored_date;  // Alt. use
`committed_date`
```

```
            var day = new Date(date.getFullYear(), date.
getMonth(), date.getDate());
            return day.getTime();
        });

        // Convert to list
        commits = underscore.pairs(commits_per_day);

        // JSON can't have integer keys; convert string keys to
ints again
        commits = underscore.map(commits, function(item) {
            key = parseInt(item[0]);
            value = item[1];
            return [key, value];
        });

        // Sort the dates; Highcharts has problems with unsorted
data
        commits = underscore.sortBy(commits, function(x) {return
x[0]; });

        // Respond with result
        response.json(commits);
    });
});
```

4. Create a function to get our Git data, as follows:

```
$(document).ready(function() {
    $.getJSON('/git/timeline', function(data) {
    });
});
```

5. Create a set of options for our chart, as shown in the following code:

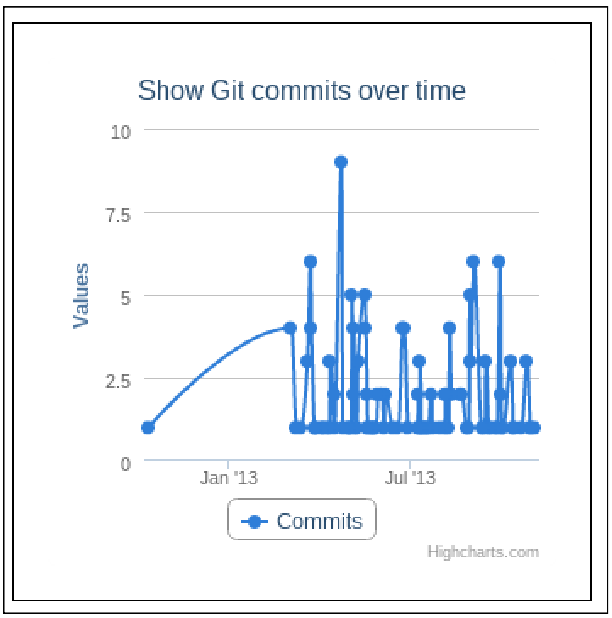
```
$(document).ready(function() {
    $.getJSON('/git/timeline', function(data) {
        var options = {
            chart: {
                type: 'pie'
            },
            title: {
                text: 'Show Git commits over time'
            },
            series: [{
                name: 'Commits',
```

```
                data: data
            }]
        };
    });
});
```

6. Render our chart and observe:

```
$(document).ready(function() {
    $.getJSON('/git/timeline', function(data) {
        var options = {/* … */};
        $('#container').highcharts(options);
    });
});
```

Your desired output will look something like the following:



How it works...

Again, Gift (`http://npmjs.org/package/gift`) does a lot of the work in this example. The notable difference in this recipe is in how we slice the data.

Instead of using `countBy` to get the count of commits by author, we use it to get the count by the authored date of the commit. From there, all we need to do is clean up the data (as we do in our `map` function), sort by the date, and return the data.

12
Other Inspirational Uses

In this chapter, we will cover the following topics:

- Demonstrating time zones with gauge charts
- Exploring a Highcharts stopwatch
- Counting words per minute
- Measuring the distance travelled
- Plotting tweets per day
- Creating a compass
- Creating a weight watching application

Introduction

In the previous chapters, we mostly experimented with the somewhat useful or typical examples of what we can do with Highcharts. This chapter explores some of the remaining chart types. In addition to that, we will also learn how we can integrate interesting APIs (for example, HTML5's `geolocation` or `localStorage` APIs) to come up with even more interesting uses of Highcharts.

For example, what if we could use Highcharts to tackle habits that we'd like to change? Such as watching our weight? Recording the changes over time and observing improvements can be incredibly motivating. Or what if we wanted to work on our typing speed and see it update in real time like the speedometer of an automobile? These are just a few of the interesting things that we can do with Highcharts. Hopefully, after reading this chapter, you'll find other inspirational uses.

Demonstrating time zones with gauge charts

There are a lot of straightforward uses of charts, especially for charts that are used in reports. However, one of the advantages of using a library for rendering dynamic charts is, well, **dynamism**. By leveraging the dynamic nature of Highcharts along with its gauge chart, we can do some very interesting things, such as creating a clock.

Getting ready

To set up a basic page and installing jQuery and Highcharts, refer to the instructions in the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, follow the ensuing instructions:

1. Create HTML fields for our time zones using the following code:

```
<div id='container'></div>
<label for='timezone'>Timezone:</label>
<select name='timezone' id='timezone'>
    <option value='localtime'>Localtime</option>
    <option value='-8'>Pacific (UTC-08:00)</option>
    <option value='-7'>Mountain (UTC-07:00)</option>
    <option value='-6'>Central (UTC-06:00)</option>
    <option value='-5'>Eastern (UTC-05:00)</option>
    <option value='-4'>Atlantic (UTC-04:00)</option>
    <option value='-3.5'>Newfoundland (UTC-03:30)</option>
</select>
<script src='./bower_components/jquery/jquery.js'></script>
```

2. Define a variable to store the time zone (`clockTimezone`) and create a function to get the positions of the clock hands, as shown in the following code:

```
$(document).ready(function() {
    var clockTimezone;

    var getClockPositions = function(options) {
        var date, now, tzOffset, tzDifference;

        options = options || {};
```

```
            date = new Date();
            tzOffset = parseFloat(options.tz);

            if (tzOffset) {
                tzDifference = (tzOffset * 60) + date.
                getTimezoneOffset();
                now = new Date(date.getTime() + tzDifference * 60 *
                1000);
            } else {
                now = date;
            }

            return {
              hours: now.getHours() + (now.getMinutes() / 60),
              minutes: ((now.getMinutes() * 12) / 60) + ((now.
              getSeconds() * 12) / 3600),
              seconds: ((now.getSeconds() * 12) / 60)
          };
        };
});
```

3. Get the initial position of the hands and define the chart options, as shown in the following code:

```
var getClockPositions = function(options) {/* … */}

var initialPosition = getClockPositions();

var options = {
    chart: { type: 'gauge' },
    title: { text: 'Time' },
    subtitle: { text: 'Localtime' },
    tooltip: { enabled: false },
    yAxis: {
        min: 0,
        max: 12,
        showFirstLabel: false,
        tickInterval: 1
    },
    series: [{
        animation: false,
        dataLabels: { enabled: false},
        data: [{
            id: 'hours',
            y: initialPosition.hours,
            dial: { radius: '70%' }
        }, {
```

```
                id: 'minutes',
                y: initialPosition.minutes,
                dial: { radius: '90%' }
            }, {
                id: 'seconds',
                y: initialPosition.seconds,
                dial: {
                    radius: '90%',
                    baseWidth: 1,
                    backgroundColor: '#faa',
                    borderColor: '#faa'
                }
            }]
        }]
    };
```

4. Render our chart and get a reference to the chart using the following code:

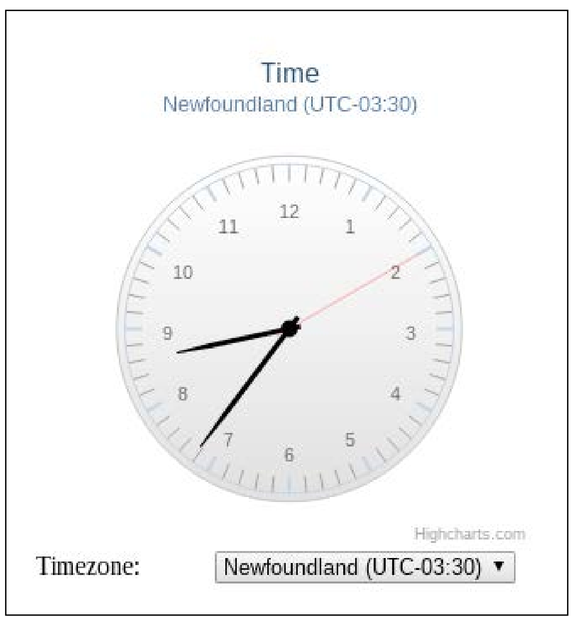
```
var options = {/* ... */};
var chart = $('#container').highcharts(options).highcharts();
Create an interval timer to update the clock hands using the
following code:
var chart = $('#container').highcharts(options).highcharts();

var clockInterval = setInterval(function() {
    var hours = chart.get('hours'),
        minutes = chart.get('minutes'),
        seconds = chart.get('seconds'),
        now = getClockPositions({
            tz: clockTimezone
        }),
        redraw = true;
        animate = false;

    hours.update(now.hours, redraw, animate);
    minutes.update(now.minutes, redraw, animate);
    seconds.update(now.seconds, redraw, animate);
}, 1000);
```

5. Create a handler for changing the time zone as shown in the following code:

```
var clockInterval = setInterval(function() {/* … */}, 1000);

$('#timezone').change(function(event) {
    var tz = this.value,
        tzText = $(this).find(':selected').text();

    clockTimezone = tz;
    chart.setTitle(undefined, {text: tzText});
});
```

6. Observe the clock on the page. You should see the clock shown in the following screenshot:

7. Change the **Timezone** value and observe the clock change, as shown in the following screenshot:



How it works...

All of this is possible due to the type of gauge chart. This chart type maps the *y* values along the outside of the gauge. Since we don't define a start angle (`options.pane.startAngle`) or end angle (`options.pane.endAngle`), our gauge wraps around completely, meaning we have the perfect setup to create a clock.

The `getClockPositions` method may look complicated. However, if we break it down, it does the following:

- Gets the current time (`date = new Date()`) and offset (if available, `tzOffset = parseFloat(options.tz)`).
- Gets the adjusted time by converting a time zone offset (for example, `-5`, for eastern time) into minutes `((tzOffset * 60) + date.getTimezoneOffset())` and adding it to the current time (`now = new Date(date.getTime() + tzDifference * 60 * 1000)`).
- Lastly, it gets the position of the hands from the adjusted time. Since our clock values go from `0` to `12`, we need to ensure that hours, minutes, and seconds all appear correctly. We do this by normalizing the values by performing the following steps:
    - Hours are the most straightforward. We take the current hour value (`now.getHours()`) and add any fractional hours (`now.getMinutes() / 60`) to it.
    - Next, we need to map minute values (that is, `0` to `59`) to hour values (`0` to `12`). We do this by multiplying the current minute value (`now.getMinutes()`) by `12` and dividing this by `60` (effectively, making each minute one-fifth of a clock value, as we would expect). We do something similar to add any fractional minutes.
    - We do the same thing with seconds, mapping each second to one-fifth of a clock value. The biggest difference is that we are not concerned with fractional seconds (that is, milliseconds).

Our interval function is fairly straightforward: we use `chart.get(id)` to get whichever series we need to update, get the updated positions for the clock hands, and then use `<series>.update(value, redraw, animate)` to set the value of the hand positions.

Since we set the time zone in a private variable (`clockTimezone`) via our change handler, our clock will update on the next tick of the clock. Lastly, we adjust the appearance of the clock hands via `series.dial`.

Exploring a Highcharts stopwatch

If we can create a clock, what other concepts can we create? A clock is just a way to measure time; perhaps there are other ways in which we can measure time? There are. With a few adjustments, we can create not only a clock, but also a stopwatch. This recipe will show how we can leverage the gauge chart to make a realistic-looking stopwatch.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the instructions in the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, follow the ensuing instructions:

1. Define the HTML code for our stopwatch controls as shown in the following code:

```
<div id='container'></div>
<button type='button' id='start'>Start</button>
<button type='button' id='reset'>Reset</button>
<script src='./bower_components/jquery/jquery.js'></script>
```

2. Define our chart options as shown in the following code:

```
$(document).ready(function() {
    var options = {
        chart: {type: 'gauge' },
        title: { text: 'Stopwatch' },
        yAxis: {
            min: 0,
            max: 60,
            showFirstLabel: false,
            tickInterval: 5
        },
        tooltip: { enabled: false },
        series: [{
            animation: false,
            dataLabels: {
              formatter: function() {
                  // Output in the following format:
                    //      HH:mm:ss.ms
                    var hours,
```

```
                        minutes,
                        seconds,
                        milli,
                        s = this.y,
                        H = 3600, // 1h = 3600 s
                        M = 60,   // 1m = 60 s

                    hours   = Math.floor(s / H);
                    minutes = Math.floor((s - (hours * H)) / M)
                    seconds = Math.floor((s - (hours * H) -
                   (minutes * M)));
                    milli = parseInt((s*1000) % 1000);

                    // Add padding to numbers
                    if (hours   < 10)  { hours   = "0" + hours; }
                    if (minutes < 10)  { minutes = "0" + minutes;
                    }
                    if (seconds < 10)  { seconds = "0" + seconds;
                    }
                    if (milli   < 100) { milli   = "0" + milli; }
                    if (milli   < 10)  { milli   = "0" + milli; }

                  return hours + ':' + minutes + ':' + seconds +
                  '.' + milli;
                }
        },
          data: [{
              id: 'seconds',
              y: 0,
              dial: {
                  radius: '90%',
                  baseWidth: 1,
                  backgroundColor: '#faa',
                  borderColor: '#faa'
              }
          }]
      }]
    };
});
```

3. Render our chart and get a reference to it, using the following code:

```
var options = {/* … */};
var chart = $('#container').highcharts(options).highcharts();
```

4. Define a variable to keep track of our timer as shown in the following code:

```
var chart = $('#container').highcharts(options).highcharts();
var stopWatchInterval;
var minimumResolution = 100;
```

5. Create a function to update our chart as shown in the following code:

```
var stopWatchInterval;
var minimumResolution = 100;

var updateTimer = function(value) {
    var seconds = chart.get('seconds'),
        prevValue = seconds.y,
        nextValue,
        redraw = true,
        animate = false;

    if (value != undefined) {nextValue = value;}
    else {nextValue = prevValue + (minimumResolution / 1000);}

    seconds.update(nextValue, redraw, animate);
};
```

6. Create the functions that affect the timer as shown in the following code:

```
var updateTimer = function(value) {/* ... */};

var timerRunning = function() {
    return !!stopWatchInterval;
};

var startTimer = function() {
    stopWatchInterval = setInterval(function() {updateTimer();},
    minimumResolution);
    $('#start').text('Stop');
};

var stopTimer = function() {
    clearInterval(stopWatchInterval);
    stopWatchInterval = 0;
    $('#start').text('Start');
};
```

7. Create handlers for our `start` and `reset` buttons as shown in the following code:

```
var stopTimer = function() {/* ... */};

$('#start').click(function() {
    if (!timerRunning()) {startTimer();}
    else {stopTimer();}
});

$('#reset').click(function() {
    if (timerRunning()) {stopTimer();}
    updateTimer(0);
});
```

8. Visit the page and observe the stopwatch at rest. You should see the stopwatch that is shown in the following screenshot:

9. Start the stopwatch by clicking on the **Start** button, and then watch it in action as shown in the following screenshot:

How it works...

Our setup in this scenario is similar to our clock in the previous recipe. Our `updateTimer` function handles the specifics of modifying the actual chart. If we provide it with a value (as we do when we reset the timer), it will set the value of the gauge to whatever value we provide.

Other than that, much of our work revolves around ensuring that our timer is in a good state. Our **Start** button will start or stop the timer, depending on whether it is running or not, and our **Reset** button will stop the timer (if it is running) and also reset it.

Counting words per minute

In our previous recipes, we used gauge charts in "less-than-typical" scenarios. In this recipe, we will use a gauge for a more typical example, such as a speedometer in an automobile. In the case of a speedometer, we measure velocity; in our example, we'll be measuring words per minute.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the instructions in the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

We will also need to include `underscore` by adding it to our dependencies in `bower.js`, as shown in the following code:

```
{
  "name": "highcharts-cookbook-chapter-12",
  "dependencies": {
    "jquery": "~1.9",
    "highcharts": "~3.0",
    "underscore": "~1.6.0"
  }
}
```

Next, we need to install our dependencies using the following command:

```
bower install
```

Lastly, we need to include `underscore.js` to our page using the following code:

```
<script src='./bower_components/highcharts/highcharts-more.src.js'></
script>
<script src='./bower_components/underscore/underscore.js'></script>
```

How to do it...

To get started, follow the ensuing instructions:

1. Define the HTML component where we will type (and measure) our words per minute as shown in the following code:

   ```
   <div id='container'></div>
   <textarea id='wpm'></textarea>
   <script src='./bower_components/jquery/jquery.js'></script>
   ```

2. Define the chart options as shown in the following code:

   ```
   $(document).ready(function() {
       var options = {
           chart: { type: 'gauge' },
           title: { text: 'Words Per Minute (WPM)' },
           pane: {
               startAngle: -150,
               endAngle: 150,
           },
           yAxis: {
               min: 0,
               max: 200,
               tickInterval: 10
           },
           tooltip: { enabled: false },
           series: [{
   ```

```
                animation: false,
                data: [{
                    id: 'wpm',
                    y: 0,
                    dial: {
                        radius: '90%',
                        baseWidth: 1,
                        backgroundColor: '#faa',
                        borderColor: '#faa'
                    }
                }]
            }]
        };
    });
```

3. Render the chart and get a reference to it using the following code:

```
var options = {/* … */};
var chart = $('#container').highcharts(options).highcharts();
```

4. Create a function to calculate words per minute as shown in the following code:

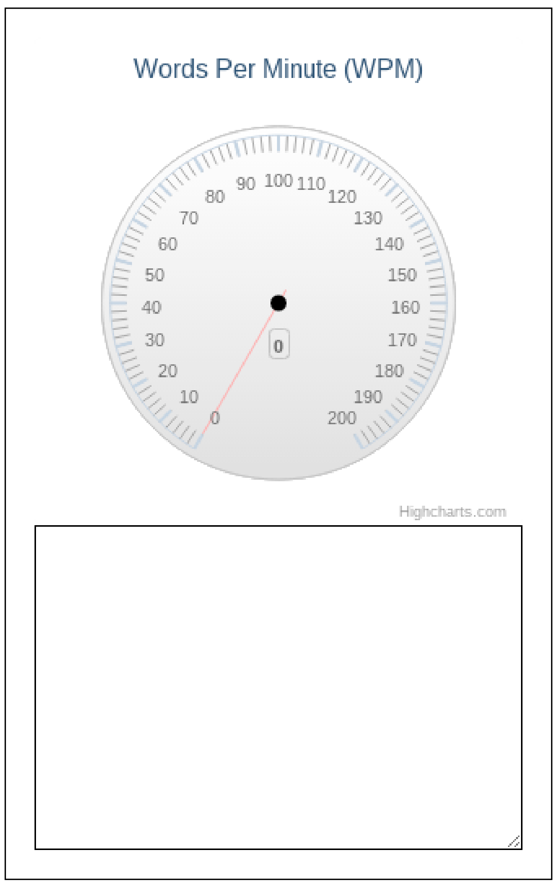
```
var chart = $('#container').highcharts(options).highcharts();

//WPM: standardized as 5 keystrokes = 1 word
var keystrokes = totalTime = lastTime = currTime = wpm = 0,
    pastWPM = [];

$('#wpm').on('keyup', function(){
    var words, wpmilli, subset,
        animate = true,
        redraw = true,
        sumOver = 3;

    currTime = new Date().getTime();

    if (lastTime != 0) {
        keystrokes++;

        totalTime += currTime - lastTime;

        words = keystrokes / 5;

        // take the average of the past few values to
        // smooth things out
        wpmilli = words / totalTime;
        pastWPM.push(wpmilli * 60000);

        if (pastWPM.length >= sumOver ) {
            subset = _.last(pastWPM, sumOver);
```

```
                wpm = _.reduce(subset, function(a, b) {return a + b;})
/ sumOver;
        }
    }

    lastTime = currTime;
});
```

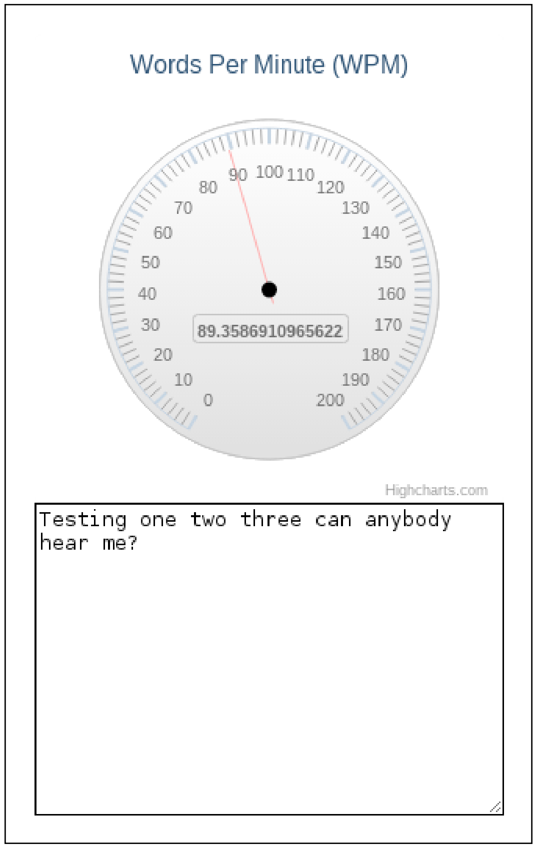
5. Update the gauge periodically (instead of doing it immediately) using the following code:

```
$('#wpm').on('keyup', function() {/* … */});

setInterval(function() {
    var redraw = animate = true;
    chart.get('wpm').update(wpm, redraw, animate) ;
}, 1000);
```

6. Visit our page and observe our **Word Per Minute (WPM)** detector at rest. You should see it as shown in the following screenshot:



7. Start typing and then observe the detector measure your speed. You should see the change in the speedometer as shown in the following screenshot:



How it works...

Whenever a key is pressed in our text area, the code does the following:

- If some amount of time has passed, increase the number of keystrokes by one (`keystrokes++`)
- Convert keystrokes to words; we use five keystrokes to represent one word
- Calculate the current words per minute and store it temporarily (`pastWPM.push(wpmilli * 60000)`)
- Take the average of the past few (three) WPM values to provide a more accurate measure
- Lastly, instead of updating the gauge on every `keyup` event (which would make our chart very sluggish), we only update it once every second

Measuring the distance travelled

Bubble charts are a good way to represent three-dimensional values in a two-dimensional space (that is, a chart). In this way, it is similar to a scatter chart that only tracks the *x* and *y* coordinates. In this recipe, we'll use a bubble chart (and a few HTML5 technologies) to log our location with Highcharts.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the instructions in the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, follow the ensuing instructions:

1. Create some controls to manage our chart using the following code:

```
<div id='container'></div>
<button id='record-location'>Record Location</button><br/>
<ul id='log-messages'></ul>
<script src='./bower_components/jquery/jquery.js'></script>
```

2. Create a function to calculate the distance between two points (that is, latitude and longitude) on the globe as shown in the following code:

```
$(document).ready(function() {
    // See http://en.wikipedia.org/wiki/Great-circle_
    distance#Formulas
    var calculateDistance = function(p1, p2) {
        var radius = 6378.137; // km

        // convert measures to radians
        var dLat = ((p1.y - p2.y) * Math.PI) / 180;
        var dLon = ((p1.x - p2.x) * Math.PI) / 180;

        var a = Math.sin(dLat/2) * Math.sin(dLat/2) +
            Math.sin(dLon/2) * Math.sin(dLon/2) * Math.cos(p1.y) *
            Math.cos(p2.y);

        var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));

        var d = radius * c;
```

```
            return d;
        };
    });
```

3. Define our chart options as shown in the following code:

```
var calculateDistance = function(p1, p2) {/* ... */};

var options = {
    chart: { type: 'bubble' },
    title: { text: 'My Location' },
    subtitle: { text: 'Over time' },
    xAxis: { title: { text: 'Longitude' } },
    yAxis: { title: { text: 'Latitude' } },
    tooltip: {
        formatter: function() {
            var coordsLn, recordedLn, distanceLn, distance, index,
            p;

            // indexOf isn't supported in IE8 or less,
            // but neither is geolocation!
            index = this.series.data.indexOf(this.point);

            if (index > 0) {
                distance = calculateDistance(this.point, this.
                series.data[index-1]);
            } else {
                distance = 0;
            }

            recordedLn = '<b>Recorded:</b> ' + new Date(this.
            point.timestamp) + '<br/>';
            coordsLn = '<b>Coordinates:</b> (' + this.x + ', ' +
            this.y + ')<br/>';
            distanceLn = '<b>Geographical distance from last
            logged:</b> ~' + distance + 'km';
            return recordedLn + coordsLn + distanceLn;
        },
    },
    legend: { enabled: false },
    series: [{
        id: 'position',
        lineWidth: 1,
        data: []
    }]
};
```

4. Render our chart and obtain a reference to it, using the following code:

```
var options = {/* ... */};
var chart = $('#container').highcharts(options).highcharts();
```

5. Define functions for accessing `localStorage` as shown in the following code:

```
var chart = $('#container').highcharts(options).highcharts();

var key = 'highcharts_location';
var addToLocalStorage = function(point) {
    var data = JSON.parse(localStorage[key] || "[]");
    data.push(point);
    localStorage[key] = JSON.stringify(data);
};
```

6. Prompt the user to allow location access using the following code:

```
var addToLocalStorage = function(point) {/* ... */};

try {
    navigator.geolocation.getCurrentPosition(function(){});
} catch(e) {
    console.log('It appears that your browser does not support the
Geolocation API :(');
}
```

The following screenshot shows the output:

An example of what the prompt looks like in Chrome

7. Define handlers that log our position using the following code:

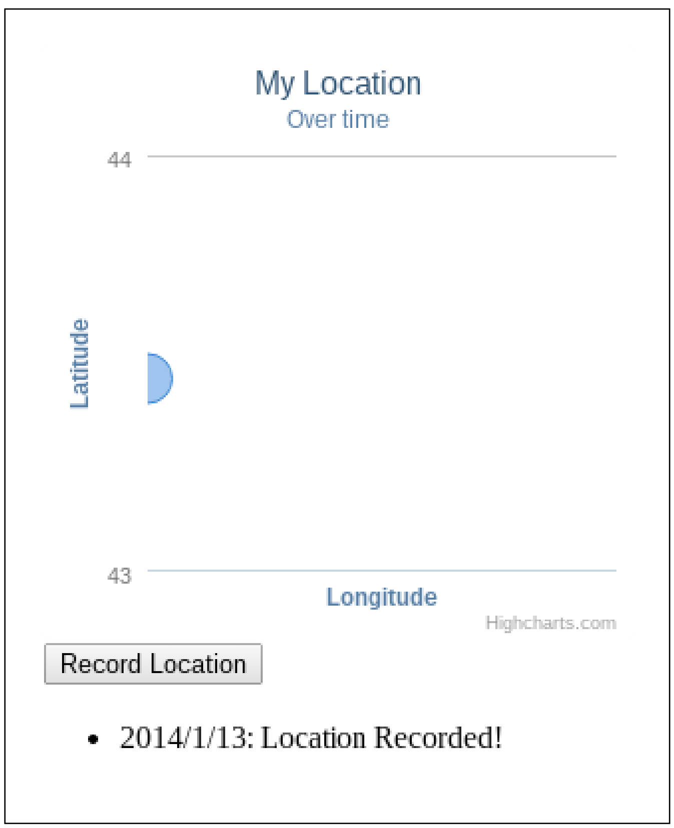
```
try {
    navigator.geolocation.getCurrentPosition(function(){});
} catch(e) {
    console.log('It appears that your browser does not support the
Geolocation API :(');
}

$('#record-location').click(function() {
    navigator.geolocation.getCurrentPosition(function(p){
        var position = chart.get('position'), point;

        point = {
            y: p.coords.latitude,
```

```
                x: p.coords.longitude,
                z: p.coords.accuracy,
                altitude: p.coords.altitude,
                timestamp: p.timestamp
            } ;

            position.addPoint(point);
            addToLocalStorage(point);
        });
});
```

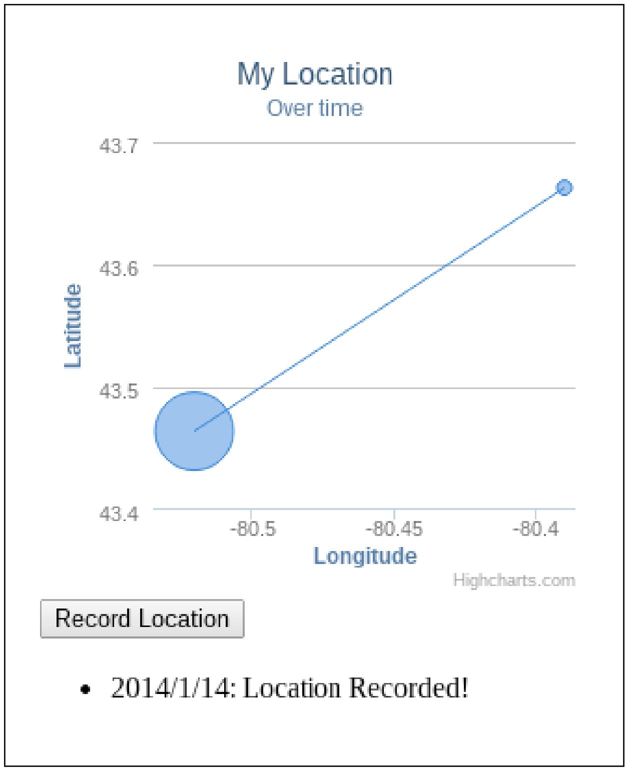
8. Load the initial data into the chart using the following code:

```
$('#record-location').click(function() {/* … */});

var data = JSON.parse(localStorage[key] || "[]");
chart.get('position').setData(data);
```

9. Open the page and record a position using the **Record Location** button as shown in the following screenshot.



10. Travel to some other location, and then click on the **Record Location** button again. You should now see an output similar to the following screenshot:



How it works...

Using the `geolocation` API allows us to access information about the user's location (provided they allow us to do so). More specifically, if we provide a single-argument (`p`) callback to `navigation.geolocation.getCurrentPosition`, we can get the following information:

- Latitude (`p.coords.latitude`)
- Longitude (`p.coords.longitude`)
- Accuracy (`p.coords.accuracy`)
- Altitude (if available, `p.coords.altitude`)
- Altitude accuracy (if available, `p.coords.altitudeAccuracy`)

- Heading (if available, `p.coords.heading`)
- Speed (if available, `p.coords.speed`)

The `localStorage` API acts a lot like a dictionary in that we can get and set keys with `localStorage[key] = value`. Unfortunately, it doesn't handle nested objects too well. So, we convert all of the values to strings before we store them, and then convert them back to objects when we retrieve keys from `localStorage`.

Plotting tweets per day

Sometimes, patterns emerge when we look at data differently. In this recipe, we'll take data from our existing Twitter feed, summarize it using Node.js, and then display it using a bubble chart where the size of the bubble is the number of tweets in that day.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the instructions in the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

We will also need to include `underscore` by adding it to our dependencies in `bower.js` as shown in the following code:

```
{
  "name": "highcharts-cookbook-chapter-12",
  "dependencies": {
    "jquery": "~1.9",
    "highcharts": "~3.0",
    "underscore": "~1.6.0"
  }
}
```

Next, we need to install our dependencies using the following command:

```
bower install
```

We will also need to include `underscore.js` to our page using the following code:

```
<script src='./bower_components/highcharts/highcharts-more.src.js'></
script>
<script src='./bower_components/underscore/underscore.js'></script>
```

We will also need to make some changes in order to obtain tweets on the server side. This can be done using the following steps:

1. Create `package.json`, and add the following code to it:

```
{
  "name": "highcharts-cookbook-chapter-12",
  "version": "0.0.0",
  "dependencies": {
    "underscore": "~1.5.2",
    "express": "~3.4.7",
    "twitter": "~0.2.5"
  }
}
```

2. Install the Node.js dependencies using the following command:

```
npm install
```

3. In order to access the Twitter API, we will need to obtain a consumer key, consumer secret, access token key, and access token secret, These values can be obtained after creating an application (`https://dev.twitter.com/apps`).

How to do it...

To get started, follow the ensuing instructions:

1. Set up a basic `express` application in a file `app.js` as shown in the following code:

```
var express = require('express');
var app = express();

app.use(express.json());
app.use(express.static(__dirname));

app.listen(8888);
console.log('Listening on port 8888');
```

2. Create a handler to obtain tweets using the following code snippet:

```
app.use(express.static(__dirname));

app.get('/tweets/summary', function(request, response) {});

app.listen(8888);
```

3. Create a Twitter client and then obtain tweets using the following code:

```
var twitter = require('twitter');
client = new twitter({
    consumer_key: 'YOUR KEY GOES HERE',
    consumer_secret: 'YOUR SECRET GOES HERE',
    access_token_key: 'YOUR ACCESS TOKEN',
    access_token_secret: 'YOUR TOKEN SECRET'
});

app.get('/tweets/summary', function(request, response) {
    client.get('/statuses/user_timeline.json', {
        screen_name: 'YOUR_SCREEN_NAME',
        include_entities: false,
        count: 200,
        trim_user: 1,
        include_rts: false
    }, function(data) {
});
});
```

4. Simplify the Twitter data using the following code:

```
var underscore = require('underscore');
app.get('/tweets/summary', function(request, response) {
        /*...*/
    }, function(data) {
        var tweets;

        tweets = underscore.map(data, function(tweet) {
            return {
                created: new Date(tweet.created_at),
                text: tweet.text
            }
        });
    }
});
```

Next, group the tweets by day as shown in the following code:

```
app.get('/tweets/summary', function(request, response) {
    client.get('/statuses/user_timeline.json', {
        /*...*/
    }, function(data) {
        var tweets;

        tweets = underscore.map(data, function(tweet) {/* ...*/});
```

```
            tweets = underscore.countBy(tweets, function(tweet) {
                var date = tweet.created;
                var day = new Date(date.getFullYear(), date.
                getMonth(), date.getDate());
                return day.getTime();
            });
        }
    });
```

5. Clean up, sort, and respond with tweets using the following code:

```
app.get('/tweets/summary', function(request, response) {
    client.get('/statuses/user_timeline.json', {
        /* ... */
    }, function(data) {
        var tweets;

        tweets = underscore.map(data, function(tweet) {/*...*/});

        tweets = underscore.countBy(tweets, function(tweet)
        {/*...*/});

        // Convert to list of lists
        tweets = underscore.pairs(tweets)

        // JSON can't have integer keys; convert string keys to
        ints again
        tweets = underscore.map(tweets, function(tweetSummary) {
            var date, monthYear, day;

            timestamp = parseInt(tweetSummary[0]);
            count = tweetSummary[1];

            date = new Date(timestamp);
            monthYear = (
                new Date(date.getFullYear(), date.getMonth(), 1)
            ).getTime();
            day = date.getDate();

            return [monthYear, day, count];
        });

        // Sort the dates; Highcharts has problems with unsorted
        data
        tweets = underscore.sortBy(tweets, function(x) {
            return x[0];
        });
```

```
            // Respond with result
            response.json(tweets);
        }
    });
```

6. In our page, create a callback to handle the response from our `/tweets/summary` service, as shown in the following code snippet:

```
$(document).ready(function() {
    $.getJSON('/tweets/summary', function(data) {
    });
});
```

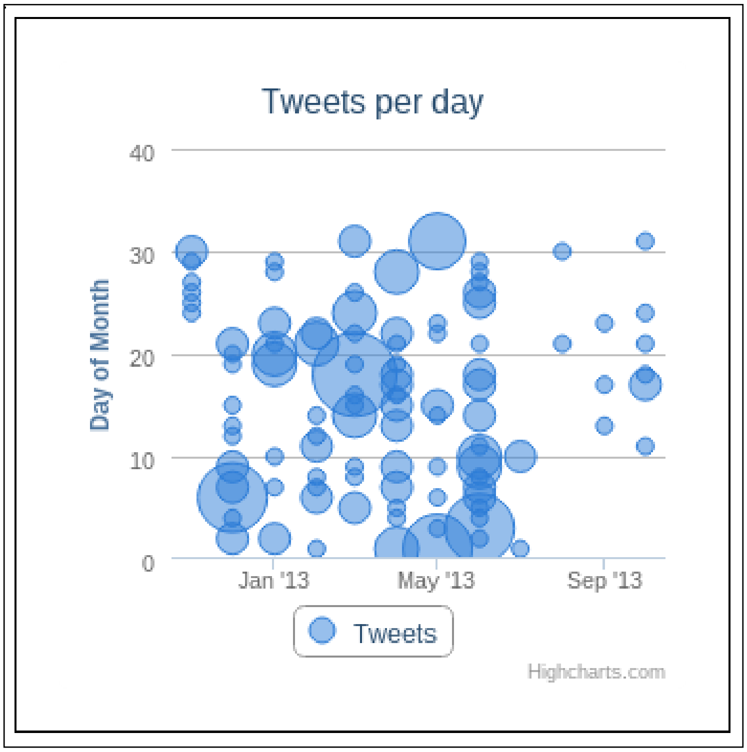
7. Define our chart options as shown in the following screenshot:

```
$.getJSON('/tweets/summary', function(data) {
    var options = {
        chart: {type: 'bubble' },
        title: { text: 'Tweets per day' },
        xAxis: { type: 'datetime' },
        yAxis: {
            min: 1,
            max: 31,
            title: { text: 'Day of Month' }
        },
        tooltip: {
            formatter: function() {
                var line1, line2, date, day, count, fmtDate;

                date = new Date(this.point.x);
                day = this.point.y;
                count = this.point.z;

                date.setDate(date.getDate() + day - 1);
                fmtDate = date.getFullYear() + '/' + (date.
                getMonth() + 1) + '/' + date.getDate();

                line1 = '<b>' + fmtDate + '</b><br/>';
                line2 = '<b>Tweets:</b> ' + count;

                return line1 + line2;
            }
        },
        series: [{name: 'Tweets', data: data}]
    };
});
```

8. Render our chart using the following code:

```
var options = {/* ... */};
$('#container').highcharts(options);
```

The following screenshot shows the rendered chart for a particular person:



An example of someone's tweets displayed over time

How it works...

Most of the work in this recipe is actually on the backend. After we've created a Twitter client, we can access any REST endpoint in the Twitter documentation (`https://dev.twitter.com/docs/api/1.1`) via `twitter.<verb>(url, parameters, callback)`.

In our recipe, we get data from `/statuses/user_timeline.json`, which gives the status updates from a user. In our parameters, we specify who we want to get updates from (`screen_name`), how many tweets we want (`count: 200`), and whether we want to include retweets (`include_rts`). We could also set many other fields based on what is stated in the documentation of the endpoint.

Creating a compass

In this recipe, we'll use what we've learned about gauge charts and the HTML5 geolocation API to create a compass. Our compass may not be as accurate as a physical compass, but it should give us a decent approximation of our heading.

> Not all browsers/devices provide the necessary data to get heading information, so this recipe may not work in all circumstances.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the instructions in the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

How to do it...

To get started, follow the ensuing instructions:

1. Define the controls for our chart as shown in the following code:

```
<div id='container'></div>
<button id='start-compass'>Start Compass</button>
<script type='text/javascript'></script>
```

2. Define the options for our chart as shown in the following code:

```
$(document).ready(function() {
    var options = {
        chart: {type: 'gauge' },
        title: { text: 'Compass' },
        yAxis: {
            min: 0,
            max: 360,
            showLastLabel: false,
            tickInterval: 45,
            labels: {
                formatter: function() {
                    var direction,
                        directions = {
                            0: 'N', 45: 'NE', 90: 'E', 135: 'SE',
                            180: 'S', 225: 'SW', 270: 'W', 315:
                            'NW'
                        };
                    direction = directions[this.value] || '';
```

```
                    return '<b>' + direction + '</b>'
                }
            }
        },
        tooltip: { enabled: false },
        series: [{
            animation: false,
            dataLabels: false,
            data: [{
                id: 'north',
                y: 0,
                dial: {
                    radius: '90%',
                    backgroundColor: '#f66',
                    borderColor: '#faa',
                    baseWidth: 5,
                    baseLength: '90%'
                }
            }, {
                id: 'south',
                y: 180,
                dial: {
                    radius: '90%',
                    backgroundColor: '#bbb',
                    borderColor: '#000',
                    baseWidth: 5,
                    baseLength: '90%'
                }
            }]
        }]
    };
});
```

3. Render and obtain a reference to our chart as shown in the following code:

```
var options = {/* ... */};
var chart = $('#container').highcharts(options).highcharts();
```

4. Create methods to start and stop watching our heading using the following code:

```
var chart $('#container').highcharts(options).highcharts();

var compassInt;
var compassOn = function () {return !!compassInt;};

var startCompass = function() {
compassInt = navigator.geolocation.watchPosition(function(p){
    var northSeries = chart.get('north'),
        southSeries = chart.get('south'),
        redraw = true,
```

```
            animate = false,
            north, south;

        north = p.coords.heading;
            south = (180 + north) % 360;

            northSeries.update(north, redraw, animate);
            southSeries.update(south, redraw, animate);
        });
        $('#start-compass').text('Stop Compass');
    };

    var stopCompass = function () {
        navigator.geolocation.clearWatch(compassInt);
        compassInt = 0;
        $('#start-compass').text('Start Compass');
    };
```

5. Create handlers to control starting and stopping our compass using the following code:

```
var stopCompass = function() {/* … */};

$('#start-compass').click(function() {
    if(compassOn()) {stopCompass();}
    else {startCompass();}
});
```

The following screenshot shows the compass in its default state:

How it works...

The HTML5 geolocation API (`navigator.geolocation`) provides the following two methods that are used to obtain a user's location information; each method accepts a single-argument callback function (whose argument we will refer to, as in the code, as `p`):

- The `getCurrentPosition` method returns the user's current latitude (`p.coords.latitude`), longitude (`p.coords.longitude`), and the accuracy in meters (`p.coords.accuracy`), of the coordinates. Other values may be available, depending on the device. For example, the heading (`p.coords.heading`) that we use in our compass.
- The `watchPosition` method does the same, except that it is similar to `setInterval` in that it will poll for the user's current position periodically.

In our example, when a user clicks on the **Start** button, we start watching the user's position to get the current heading. If we click on the button again, the compass will stop polling for the user's position. The only special work that we need to do is to draw the *south* arm 180 degrees from the *north* arm to make our chart more closely resemble a compass.

Creating a weight-watching application

Once a year, people work to come up with a list of things to change for the new year—new year's resolutions. Losing weight is a new year's resolution that comes up quite often, and an important part of losing weight (or making progress in just about anything) is weight measurement. In this recipe, we will see how we can track our weight, or just about any measurable quality, on a regular basis.

Getting ready

To set up a basic page and install jQuery and Highcharts, refer to the instructions in the *Getting ready* section of the *Creating your first chart* recipe in *Chapter 1*, *Getting Started with Highcharts*.

We will also need to include `underscore` by adding it to our dependencies in `bower.js`, as shown in the following code:

```
{
  "name": "highcharts-cookbook-chapter-12",
  "dependencies": {
    "jquery": "~1.9",
    "highcharts": "~3.0",
    "underscore": "~1.6.0"
  }
}
```

Next, we need to install our dependencies using the following command:

```
bower install
```

We will also need to include `underscore.js` to our page, as shown in the following code:

```
<script src='./bower_components/highcharts/highcharts-more.src.js'></
script>
<script src='./bower_components/underscore/underscore.js'></script>
```

How to do it...

To get started, follow the ensuing instructions:

1. Define controls for our chart as shown in the following code:

```
<div id='container'></div>

<label for='date-list'>Date:</label>
<select name='date-list' id='date-list'>
</select><br/>
<label for='weight'>Weight (kg):</label>
<input type='text' name='weight' id='weight' />
<button name='modify' id='modify'>Add / Modify</button>

<script src='./bower_components/jquery/jquery.js'></script>
```

2. Define our chart options as shown in the following code:

```
$(document).ready(function() {
    var options = {
        chart: { type: 'spline' },
        title: { text: 'Weight Watcher' },
        xAxis: { type: 'datetime' },
        series: [{
            id: 'weight',
            name: 'Weight',
            data: []
        }]
    };
});
```

3. Render and obtain a reference to our chart using the following code:

```
var options = {/* … */};
var chart = $('#container').highcharts(options).highcharts();
```

4. Create a function to add new elements to our selected box as shown in the following code:

```
var chart = $('#container').highcharts(options).highcharts();
var addToList = function(timestamp, weight) {
    var $elem, $list = $('#date-list');

    $elem = $('<option/>', {
        value: timestamp,
        text: new Date(timestamp)
    });
    $elem.data('weight', weight);

    $list.append($elem);
};
```

5. Create functions to interact with `localStorage` as shown in the following code:

```
var addToList = function(timestamp, weight) {/* … */};

var key = 'highcharts_weightwatcher';
var getDataFromStorage = function() {
    var obj, list

    obj = JSON.parse(localStorage[key] || "{}");

    list = _.pairs(obj);
        return _.map(list, function(item) {
            return [ parseInt(item[0]), item[1] ];
    });
};

var loadInitial = function() {
    var data = getDataFromStorage();
    var $list = $('#date-list');

    $list.empty();

    $list.append('<option value="new" selected="selected">New
    Entry</option>');

    _.each(data, function(item) {
        addToList(item[0], item[1]);
    });

    // Add Data to chart
    var series = chart.get('weight');
```

```
        series.setData(data);
    };

    var modifyData = function(weight, date) {
        var data = JSON.parse(localStorage[key] || "{}");
        date = date || new Date();
        data[date.getTime()] = weight;
        localStorage[key] = JSON.stringify(data);
    };
```

6. Create handlers for our selected box and our button as shown in the following code:

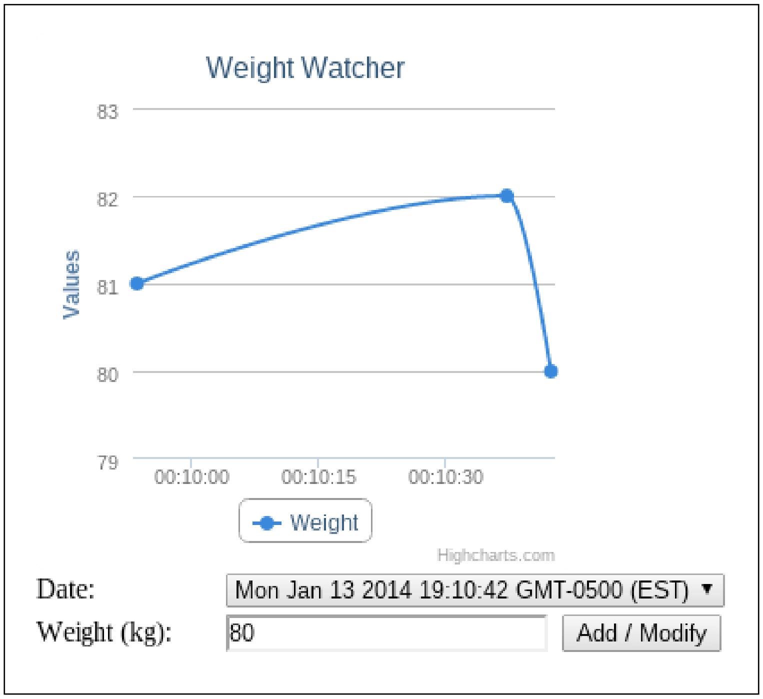
```
var modifyData = function(weight, data) {/* ... */};
$('#modify').click(function() {
    var listValue, weight, date, isNew, timestamp, series, redraw;

    weight = parseInt($('#weight').val());
    listValue = $('#date-list').val();
    isNew = (listValue === 'new');

    series = chart.get('weight');
    if (isNew) {          date = new Date();
        timestamp = date.getTime();

        addToList(date.getTime(), weight);
        series.addPoint([timestamp, weight]);
    } else {
        timestamp = parseInt(listValue);
        date = new Date(timestamp)
    }

    modifyData(weight, date);

    redraw = true;
    if (!isNew) {series.setData(getDataFromStorage(), redraw)}
});

$('#date-list').change(function() {
    var value = $(this).find(':selected').data('weight') || '';
    $('#weight').val(value);
});
```

7. Load any previously saved data using the following code:

```
$('#date-list').change(function() {/* ... */});

loadInitial();
```

8. Visit the page and observe our application's initial state. You should get something similar to the following screenshot:

9. Record a few weight values via the **Add / Modify** button as shown in the following screenshot:



How it works...

Most of our functions are self explanatory, but the following areas do need clarification:

- We store all of our data in `localStorage` as a dictionary (`{}`) with the timestamp as the key. This makes it easy for us to add new values or to replace values from previous points in time.
- Since we use `timestamp` as a key for several things and an `option` element can only have one `value` attribute, we store `weight` on the `option` element as a data attribute using `$elem.data('field', value)`.
- As previously mentioned, `localStorage` doesn't support object nesting very well. So, we convert everything to a string before storing and to an object when retrieving.

Index

D

E

Q

R

S

T

U

V

W

X

Y

PACKT PUBLISHING

Thank you for buying
Highcharts Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

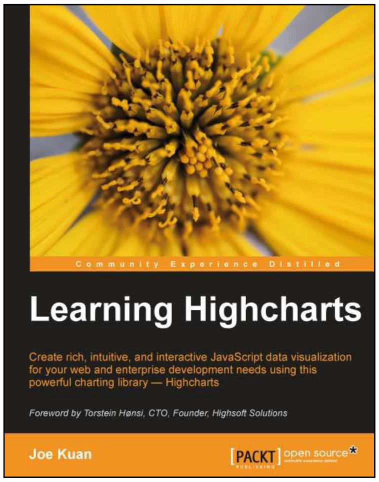
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] PUBLISHING

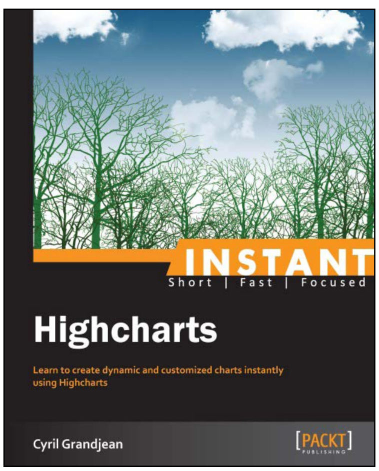


Learning Highcharts

ISBN: 978-1-84951-908-3 Paperback: 362 pages

Create rich, intuitive, and interactive JavaScript data visualization for your web and enterprise development needs using this powerful charting library — Highcharts

1. Step-by-step instructions with real-live data to create bar charts, column charts and pie charts, to easily create artistic and professional quality charts.
2. Learn tips and tricks to create a variety of charts such as horizontal gauge charts, projection charts, and circular ratio charts.
3. Use and integrate Highcharts with jQuery Mobile and ExtJS 4, and understand how to run Highcharts on the server-side.



Instant Highcharts

ISBN: 978-1-84969-754-5 Paperback: 50 pages

Learn to create dynamic and customized charts instantly using Highcharts

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Create your first customized and interactive Highcharts chart.
3. Get to grips with the core concepts of Highcharts.
4. Learn how to progress with the Highcharts library.

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