Community Experience Distilled

Ext JS 6 By Example

Create powerful, real-world web applications with Ext JS 6 using sample projects

Anand Dayalan

[PACKT] PUBLISHING open source* community experience distilled

Ext JS 6 By Example

Create powerful, real-world web applications with
Ext JS 6 using sample projects

Anand Dayalan

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Ext JS 6 By Example

First published: August 2015

Production reference: 1250815

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

ISBN 978-1-78355-049-4

www.packtpub.com

Credits

Author
Anand Dayalan

Reviewers
Berend de Boer
Stefan Gehrig
Pramod Khare
James YK Moey

Commissioning Editor
Nadeem Bagban

Acquisition Editor
Meeta Rajani

Content Development Editor
Nikhil Potdukhe

Technical Editor
Siddhesh Ghadi

Copy Editor
Relin Hedly

Project Coordinator
Vijay Kushlani

Proofreader
Safis Editing

Indexer
Monica Ajmera Mehta

Graphics
Abhinash Sahu

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Author

Anand Dayalan is a software engineer who resides in the U.S. He currently works for Microsoft in Seattle. Anand has extensive experience and passion in developing scalable and cool-looking enterprise web applications.

He started building web applications in 2005 and has built numerous enterprise web applications, including an e-commerce web application, using Ext JS and Microsoft technologies.

Anand enjoys watching movies, listening to music, reading, cooking delicious food, and (most importantly) spending time with his family.

About the Reviewers

Berend de Boer has been using Ext JS since its inception as YUI-Ext in 2006. The main reason for using Ext JS was the requirement for a grid. This is probably the reason many developers still turn to Ext JS. However, Ext JS has grown tremendously since then, and this book proves it with extensive examples. Berend currently uses Ext JS standalone and as part of the control panel for customers of Xplain Hosting, a Drupal-specific hosting company he founded in 2008.

Stefan Gehrig is an aeronautical engineer and works as a software architect and senior PHP and JavaScript developer at TEQneers GmbH & Co. KG, a Stuttgart-based software development company (Germany) that specializes in building medium- to large-scale enterprise solutions based on common web technologies, such as PHP and JavaScript. He is a Zend-certified PHP and an Oracle-certified MySQL developer.

Since the past few years, Stefan has been focusing on writing a company-internal rapid application development framework based on Ext JS and PHP and implementing this framework in data-driven enterprise applications.

Pramod Khare holds an MS degree in computer science. He has almost 4 years of experience in Ext JS, right from Ext JS version 2.x to 6.0. Pramod loves to write on topics such as Ext JS custom components and widgets. In his free time, he likes to program and cycle. Pramod also enjoys hiking. He is currently working at Amazon in Seattle, WA.

James YK Moey is a devoted husband and a loving father of one. He is a technology enthusiast and has worked in various IT roles since 1999. James is very passionate about science and technology. He is also an excellent solution provider.

James is currently working for a Fortune 500 company, where he manages a lot of web property. He is also one of the activists for a group of people who share the same interest of driving innovation and finding technological solutions.

James envisions that technology could make the world better and improve quality of life.

In his busy daily life and between work and family commitments, James enjoys playing badminton with his friends and dining out.

Table of Contents

Preface

Ext JS is one of the most famous JavaScript frameworks available on the market. Ext JS is almost like a one-stop shop for developing rich UI applications. This book is all about the powerful Ext JS application framework and is written for developers who like to see less theory and more code examples and sample projects.

This book will help you to understand the concepts quickly and reduce your learning curve. The working source code and explanation of projects will help you to learn code quickly and easily.

The book starts with an introduction to Ext JS and talks about how to set up the development environment with the installation guide for Ext JS 6, Sencha Cmd, and other required tools. Each chapter starts with some concepts of Ext JS 6 and ends with a sample project using the concepts you have already learned.

What this book covers

Chapter 1, *Getting Started with Ext JS*, introduces you to Ext JS and Sencha Cmd. It provides a clear, step-by-step installation guide for Ext JS 6, Sencha Cmd, and other tools required for development in both Mac and the Windows operating system.

Chapter 2, *Core Concepts*, discusses some of the core concepts of Ext JS 6. Understanding these concepts is very important before we start building a sample application. You'll learn about the class system and how to create and extend classes, events, layouts, and containers.

Chapter 3, *Basic Components*, covers some of the basic components available in Ext JS 6. It uses the concepts learned in the previous chapters and in this chapter to create a simple project.

Chapter 4, *Data Packages*, explores the tools available in Ext JS 6 to handle data and the communication between the server and client. It concludes with a sample project using a RESTful API.

Chapter 5, *Working with Grids*, talks about the different types of grid components in Ext JS 6 at an advanced level. It covers concepts such as pagination, sorting, filtering, searching, row editing, cell editing, grouping, docking toolbars, buffered scrolling, column resizing and hiding, grouped headers, multiple sort grids, row expanders, and so on. It also tells you how to build a sample project called company directory.

Chapter 6, *Advanced Components*, covers advanced components in Ext JS 6, such as trees and data views. It introduces you to a sample project called picture explore that is built with trees and data view components.

Chapter 7, *Working with Charts*, talks about the different types of chart components in Ext JS 6. It concludes with a sample project called expense analyzer.

Chapter 8, *Theming and Responsive Design*, focuses on the basics of how to theme your Ext JS application and responsive design. It also introduces you to SASS.

What you need for this book

- Mac or the Windows operating system
- 2 GB of free disk space

This book provides steps to install the following:

- JRE
- Sencha Cmd 6
- Ext JS 6
- Ruby

Who this book is for

If you're a front-end web developer and wish to learn a new JavaScript framework, or if you already know Ext JS and are looking for a practical resource with multiple example projects to get expert-level knowledge of Ext JS, this will be a great resource for you.

A basic understanding of HTML, CSS, and JavaScript is required.

Conventions

In this book, you will find a number of text styles 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: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
Ext.define('MyApp.model.Population', {
  extend: 'Ext.data.Model',
  fields: ['year', 'population']
});
```

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

```
Ext.define('MyApp.model.Population', {
  extend: 'Ext.data.Model',
  fields: ['year', 'population']
});
```

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

```
sencha generate app --ext MyApp ./myapp
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

[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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in 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 at `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 from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. 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 with 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/0494OS_ColorImages.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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

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

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Getting Started with Ext JS

Gone are the days when you used plain Vanilla JavaScript. JavaScript is a great, powerful language, but many find it difficult to maintain the code as the web application gets bigger and bigger. So, it's very difficult and time consuming to handle everything in plain Vanilla JavaScript.

When it comes to JavaScript frameworks, there are client-side JavaScript frameworks as well as server-side JavaScript frameworks. Ext JS is a client-side JavaScript framework.

This chapter is pretty much about setting the development environment with the required tools, and we'll see the basic introduction of Ext JS. In this chapter, we will cover the following topics:

- The advantages of using Ext JS
- An introduction to Ext JS
- Setting up Ext JS and Sencha Cmd
- Scaffolding the Ext JS application with Sencha Cmd
- The application architecture
- Exploring Sencha Cmd commands
- Debugging an Ext JS application
- The development IDE

Why Ext JS?

Now, let's take a look at some of the advantages of using Ext JS over plain Vanilla JavaScript in your web application.

Cross-browser support

You may have spent several hours or even days solving the cross-browser bugs. Why should you spend time for this instead of focusing on your business functionality? Rather, if you use a good JavaScript framework, such as Ext JS, it will take care of most of these tasks, and you can focus on the business functionality.

Rich UI components

Ext JS comes with a huge number of rich UI controls, such as data grid, tab panels, tree controls, date picker, charts, and so on, and these should reduce your development time a lot.

Two-way binding

Two-way binding means that when the view data changes, your model gets updated automatically. Also, when your application updates the model, the data will be propagated to the view automatically.

For example, take the edit page. When you load the page, you have to render the data from the model to the HTML, and when the user updates the data in the view, you have to update your model. You don't have to do these programmatically yourself if you use Ext JS.

Architectural pattern for JavaScript

As more and more code is moved to the client, maintaining the client side, JavaScript becomes difficult. By bringing **MVC** (**Model View Controller**)/**MVVM** (**Model View, View Model**) to the client side, it becomes easier to maintain the client-side JavaScript code, and it increases the productivity. MVC and MVVM are explained later in this chapter.

Simplifying a complicated operation

Think about writing the AJAX `call` in plain JavaScript and making sure to support all the browsers as well. Take a look at the source code of the AJAX `call` method in any of the JavaScript framework. Think about creating a complex control like grid in plain JavaScript with features, such as pagination, sorting, filtering, grouping, keyboard navigation, editable fields, and so on.

Easy DOM access

In plain JavaScript, you can access the DOM, but it is a bit complicated.

Client-side routing

In web applications, routing means mapping of the URL to web pages and the logic behind it. Routing can be on the server side as well as on the client side. Typically, client routing is used in a **Single-Page Application** (**SPA**).

Support for accessibility

Accessibility means that the content of application must be easily accessible to people who are visually impaired and depend on assistive technologies (such as screen readers). Developing an application with very good support for accessibility is very difficult.

In USA, if you are developing software that will be used by federal and state employees, then in most cases, you have to make sure that your application is accessible as per *Section 508*. Very few JavaScript frameworks provide very good support for accessibility. Ext JS provides excellent support for accessibility.

World Wide Web Consortium (**W3C**) has created a technical specification called **WAI-ARIA** (**Web Accessibility Initiative - Accessible Rich Internet Applications**). This defines ways to make web application accessible for people with disabilities. Ext JS has excellent support for this, and all the controls/widgets in Ext JS are accessible and don't require any additional code or work from you.

An introduction to Ext JS

Ext JS is almost a one-stop shop to develop rich UI applications. It provides MVC, MVVM, two-way binding, cross-browser compatibility, routing, an extensive set of rich UI components, charts, and so on. Ext JS also has an excellent documentation for all the APIs in the framework. Ext JS is originally built as an add-on library extension of YUI by Jack Slocum, it is now a product of Sencha Inc.

In Ext JS, you'll write most of your code in JavaScript. Mostly, you do not need to write HTML. Ext JS ships with a huge set of rich UI components, which is a huge time save in your development.

All the example code and sample projects code in this book will use the latest Ext JS Version 6, but still the majority of the code are compatible with the previous Ext JS Version 5. Most concepts in Ext JS 5 and Ext JS 6 are the same. So, if you are using Ext JS 5, you can still get a great benefit out of this book. However, remember some of the code in this book will not run in Ext JS 5 and may need some minor modification to make it run in Ext JS 5.

The most important change in Ext JS 6 is that it merges two frameworks: Ext JS and Sencha Touch into one single framework. Ext JS 6 also brought a new SASS compiler called Fashion, 3D chart improvements, and so on.

To understand why the merge of Ext JS and Sencha Touch happened, we need to look back a bit.

Sencha Touch was a separate product that specialized in creating touch applications for mobiles and tablets. It leverages hardware acceleration techniques to provide high-performance UI components for mobile devices.

Ext JS 4 and Ext JS 5 are used mainly to develop web applications for the desktop. If you have created a web application for the desktop in Ext JS 5 or Ext JS 4, which will still work in a mobile and tablet but it won't have some of the touch-specific functionalities and won't leverage hardware acceleration techniques to provide high-performance UI components for mobile devices. So, to better support mobile devices, Sencha developers were told to use Sencha Touch.

There are many advantages of using Sencha Touch. Applications written in Sencha will have the native look of the mobile platform, and the performance will be better. However, many developers had a complaint about this because they were forced to maintain two set of code base for the same application.

Although Sencha Touch and Ext JS are totally a different product with many differences, they had a common shared code; the concepts and ideas of both the frameworks are very similar. If you know Ext JS, then it is extremely easy to learn Sencha Touch.

For long, many Ext JS and Sencha Touch users were asking why not merge both the products into a single product and bring the touch capabilities to Ext JS. In Ext JS 6, Sencha made the decision to merge both the products into a single product.

Now, in Ext JS 6, you can maintain a single code. For some of the views, you may need to have a separate view code, but there will be a lot of shared code.

In Ext JS 6, they merged the common code and put them as a core framework, and they brought a concept called toolkit. A **toolkit** is a package with visual components, such as button, panels, and so on. There are two toolkits: classic and modern. Ext JS visual components are placed in the classic toolkit, and Sencha Touch components are placed in the modern toolkit.

Now, in Ext JS 6, you can simply choose the toolkit that you want to target. If you are writing an application that only targets mobile devices, you can choose modern, and if you are targeting only for desktop, then you can choose the classic toolkit.

The universal application

If you want to target both desktop and mobile devices, then in Ext JS 6, you can create a universal application, which will use both the toolkits. Instead of adding the preceding toolkit config mentioned before, you have to add the following builds config section that specifies which build uses which toolkit and theme:

```
"builds": {
  "classic": {
    "toolkit": "classic",
    "theme": "theme-triton"
  },
  "modern": {
    "toolkit": "modern",
    "theme": "theme-neptune"
  }
},
```

The basic idea here is to have two set of toolkits in a single framework in order to target the desktop and mobile devices.

If you are totally new to Ext JS, these may be a bit confusing for you now, but don't worry about it now much. These will make more sense later when we work on the samples and example code.

Ext JS 6 ships with two set of themes for the classic toolkit and the modern toolkit. There are specific themes in Ext JS, which provides native looks for Windows, Android, and iPhone. You'll learn about theming later in *Chapter 8*, *Theming and Responsive Design*.

Setting up Ext JS

To make your Ext JS application development easy, you have to install a tool called Sencha Cmd. It's available for Windows, Mac, and Linux.

> Sencha Cmd is not a must for Ext JS' development application, but using it makes your life easier. So, it's highly recommended to use Sencha Cmd.

Sencha Cmd

Sencha Cmd is a powerful command-line tool for Sencha's application development. It helps in increasing the productivity by automating many tasks. Some of its features are scaffolding, package management, JS compiler, build scripts, theming, and so on.

Before installing Sencha Cmd 6, you need JRE, and if you're going to use ExtJS 5 and Sencha Cmd 5, then you'll also need Ruby.

Java Runtime Environment (JRE)

To check whether Java is running on your machine, type the following command in the terminal (Mac) or the command window (Windows):

```
java -version
```

If you have Java already running on your machine, then you should see something similar to the following code; otherwise, download and install the JRE or JDK:

```
java -version
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

If you still get an error, then you may need to add the installed path to the `PATH` variable.

Ruby

Note that you do not need Ruby if you are going to use Sencha Cmd 6, but if you are going to use Ext JS 5 and Sencha Cmd 5, then you will need Ruby. To check whether Ruby is installed on your machine, type the following command in the terminal (Mac) or the command window (Windows):

```
ruby --version
```

If you have Ruby already installed on your machine, then you should see something similar to the following code; otherwise, download and install Ruby:

```
ruby --version
ruby 2.0.0p481 (2014-05-08 revision 45883) [universal.x86_64-darwin14]
```

If you still get an error, then you may need to add the installed path to the `PATH` variable.

Installing Sencha Cmd

Now, download and install Sencha Cmd from the Sencha website. Once installed, make sure that Sencha Cmd is available for use from the terminal or the command window. To check, run the following command:

```
sencha which
```

If available in the terminal or the command window, then you should have something similar to the following code:

```
Sencha Cmd v6.0.0.92
/bin/Sencha/Cmd/6.0.0.92/
```

If you get an error instead, then you may need to add the installed path to the `PATH` variable. On Mac, run the following command in the terminal to add the installed path to the `PATH` variable:

```
export PATH=~/bin/Sencha/Cmd/6.0.0.92:$PATH
```

In the preceding command, change `/bin/Sencha/Cmd/6.0.0.92` with the correct installed path.

On Windows, run the following command in Command Prompt to add the installed path to the `PATH` variable:

```
set PATH=%PATH%;C:\Sencha\Cmd\6.0.0.92
```

In the preceding command, change `C:\Sencha\Cmd\6.0.0.92` with the correct installed path.

Generating the Ext JS application using Sencha Cmd

Open the terminal (Mac)/command (Windows) and type the following command:

```
sencha generate app --ext MyApp ./myapp
```

The preceding command will scaffold an Ext JS application called `MyApp` and place all the files in the subfolder called `myapp`.

Note that the preceding command will generate an app containing code for both toolkits: classic and modern. If you need either the classic or modern toolkit alone, then use `--modern` or `--classic`, as shown in the following command:

```
sencha generate app --ext --modern MyApp ./myapp
```

When you run this command for the first time, it should automatically download Ext JS 6. If it does not download Ext JS automatically, then you can manually download Ext JS 6, extract it to your local machine, and use the following command to specify the SDK path:

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

Sencha Cmd supports Ext JS 4.1.1a and higher and Sencha Touch 2.1 and higher. You can have multiple versions of the SDK in your machine. The preceding is the format of the command to generate an application based on a specific Sencha SDK.

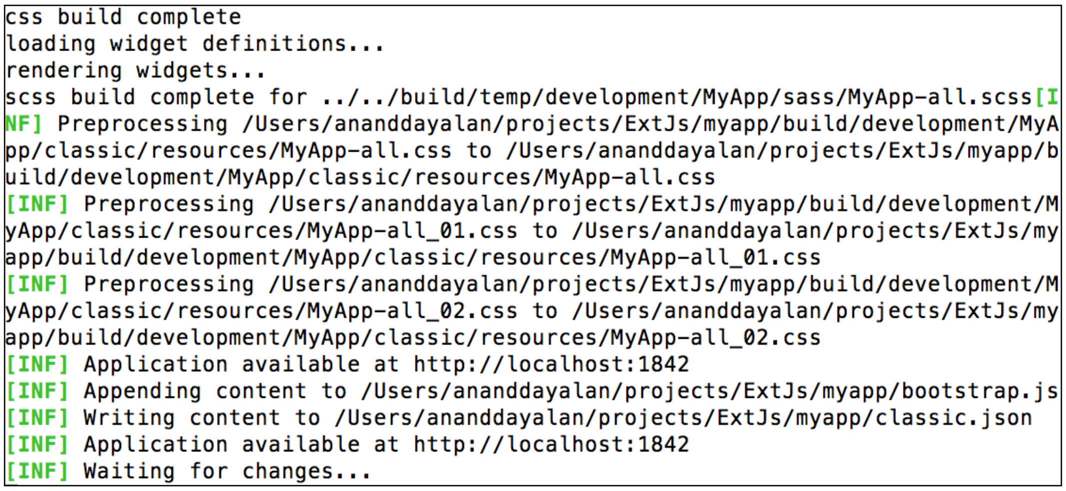
The following command is an example that will generate the Ext JS 6 application called `MyApp` under the `/projects/extjs/myapp` path:

```
sencha -sdk /bin/Sencha/ext/6.0.0/ generate app MyApp  /projects/extjs/
myapp
```

Now, to see the Ext JS application created, run the following command in the terminal or Command Prompt:

```
cd sample
sencha app watch
```

Now, this will perform a bunch of build-related tasks, and at the end, on the terminal window, you'll see something similar in *Figure 1.1*:

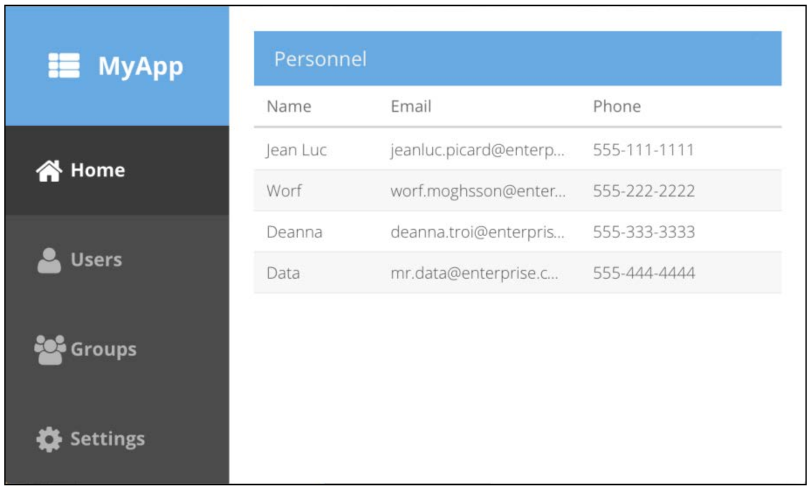


```
css build complete
loading widget definitions...
rendering widgets...
scss build complete for ../../build/temp/development/MyApp/sass/MyApp-all.scss[I
NF] Preprocessing /Users/ananddayalan/projects/ExtJs/myapp/build/development/MyA
pp/classic/resources/MyApp-all.css to /Users/ananddayalan/projects/ExtJs/myapp/b
uild/development/MyApp/classic/resources/MyApp-all.css
[INF] Preprocessing /Users/ananddayalan/projects/ExtJs/myapp/build/development/M
yApp/classic/resources/MyApp-all_01.css to /Users/ananddayalan/projects/ExtJs/my
app/build/development/MyApp/classic/resources/MyApp-all_01.css
[INF] Preprocessing /Users/ananddayalan/projects/ExtJs/myapp/build/development/M
yApp/classic/resources/MyApp-all_02.css to /Users/ananddayalan/projects/ExtJs/my
app/build/development/MyApp/classic/resources/MyApp-all_02.css
[INF] Application available at http://localhost:1842
[INF] Appending content to /Users/ananddayalan/projects/ExtJs/myapp/bootstrap.js
[INF] Writing content to /Users/ananddayalan/projects/ExtJs/myapp/classic.json
[INF] Application available at http://localhost:1842
[INF] Waiting for changes...
```

Figure 1.1

The watch monitors any code changes made, and as soon as the code changes are saved, it will refresh the browser to include the updated code changes.

If you open the application with the default URL (`http://localhost:1842`), as shown before, the application will look similar to *Figure 1.2*:

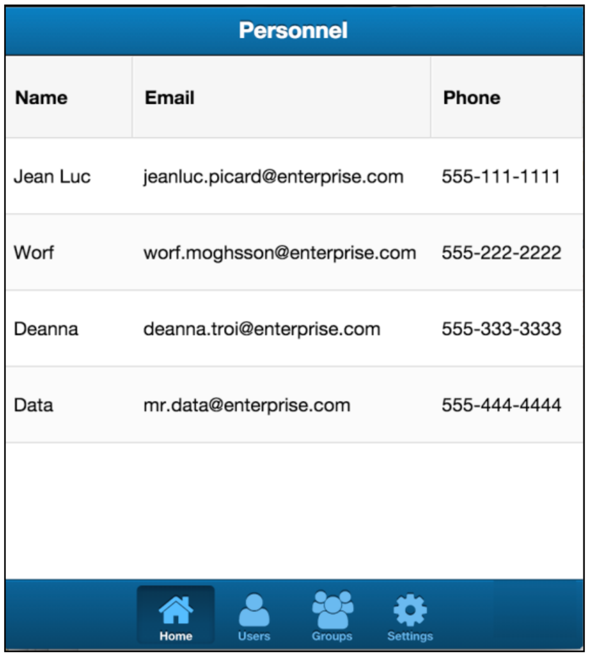


Personnel

Name	Email	Phone
Jean Luc	jeanluc.picard@enterp...	555-111-1111
Worf	worf.moghsson@enter...	555-222-2222
Deanna	deanna.troi@enterpris...	555-333-3333
Data	mr.data@enterprise.c...	555-444-4444

Figure 1.2

By default, when you navigate to the URL (`http://localhost:1842`) from a desktop computer, the app to show is automatically detected, and it shows you the classic toolkit. If this is accessed from a mobile browser, it will show you the modern toolkit. To see the modern app on the desktop computer, append `?profile=modern` to the URL, and you'll see the following screenshot:


Personnel

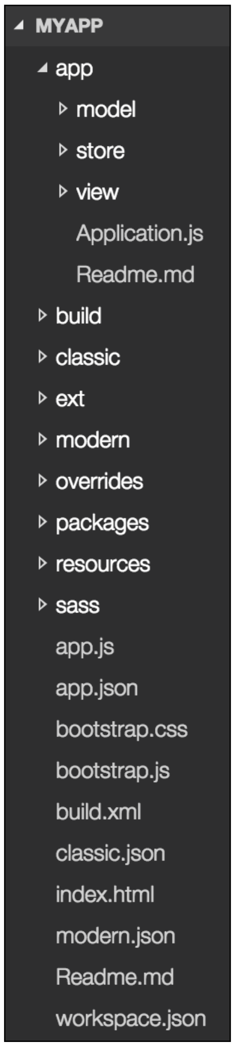
Name	Email	Phone
Jean Luc	jeanluc.picard@enterprise.com	555-111-1111
Worf	worf.moghsson@enterprise.com	555-222-2222
Deanna	deanna.troi@enterprise.com	555-333-3333
Data	mr.data@enterprise.com	555-444-4444

Figure 1.3

The contents of **MyApp** will look as shown in *Figure 1.4*. We'll take look at some of the important files of this sample application.

The app contains `model`, `store`, and `application.js`. Consider a store to be like a collection of instances of the model. The store loads data using the proxy and provides functionalities, such as sort, filter, paging, and so on. You'll learn more about the store later.

In the following screenshot, see the modern and classic folders. These folders contain the application code that uses the respective toolkits: modern and classic.



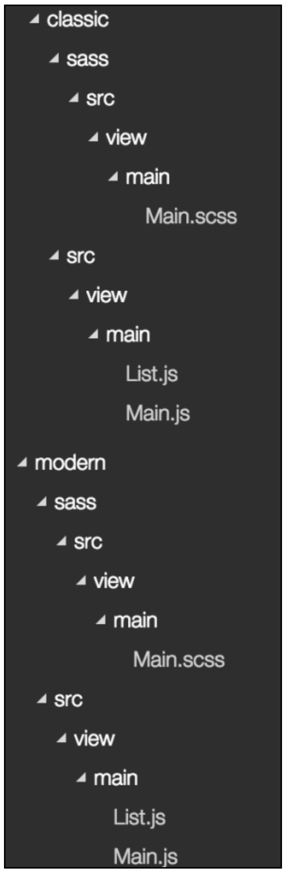
```
◢ MYAPP
  ◢ app
    ▷ model
    ▷ store
    ▷ view
      Application.js
      Readme.md
  ▷ build
  ▷ classic
  ▷ ext
  ▷ modern
  ▷ overrides
  ▷ packages
  ▷ resources
  ▷ sass
    app.js
    app.json
    bootstrap.css
    bootstrap.js
    build.xml
    classic.json
    index.html
    modern.json
    Readme.md
    workspace.json
```

Figure 1.4

Figure 1.5 shows the content of the classic folder and the modern folder. The classic folder and the modern folder contains the `src` folder that contains application views. The `main.scss` file contains the styles specific to mobile devices and the desktop. There is the `sass` folder at the root that contains the common application style.

`SASS` (`Syntactically Awesome Stylesheets`) is a stylesheet language. SASS is heavily used in Ext JS. You'll learn more about styling and theming later in *Chapter 8, Theming and Responsive Design*.

Note that these are not the framework's toolkit code, but these are application code. You can find the framework's modern and classic toolkit code in the `ext` folder:



```
◢ classic
  ◢ sass
    ◢ src
      ◢ view
        ◢ main
            Main.scss
  ◢ src
    ◢ view
      ◢ main
          List.js
          Main.js
◢ modern
  ◢ sass
    ◢ src
      ◢ view
        ◢ main
            Main.scss
  ◢ src
    ◢ view
      ◢ main
          List.js
          Main.js
```

Figure 1.5

In the next section, we'll take a look at MVC and the content that some of these files generated using Sencha Cmd in the MyApp sample application.

The application architecture

Ext JS provides support for both MVC and MVVM application architectures.

Model

This represents the data layer. The model can contain data validation and logics to persist the data. In Ext JS, mostly model is used along with a data store.

View

This represents the user interface. So, components such as button, form, and message box are views.

Controller

This handles any view-related logic, event handling of the view, and any app logic.

View model

This encapsulates the presentation logic required for the view, binds the data to the view, and handles the updates whenever the data is changed.

Now, let's examine some of the files created by Sencha Cmd for the view, controller, and view model.

If you open `app.js`, you'll see the following code, which is the starting code for your Ext JS application:

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

  extend: 'MyApp.Application',

  requires: [
  'MyApp.view.main.Main'
  ],
  mainView: 'MyApp.view.main.Main'
});
```

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.

In the preceding code, the first line defines the name of the application, and the following line extends `MyApp.Application`, which is declared in `Application.js` in the `app` folder:

```
extend: 'MyApp.Application'
```

The list of classes required for this class has to be specified in the `requires` section. These will be loaded first before instantiating this class. The last line specifies the name of the initial view to create.

Next, if you check the `app` folder, you will see `Application.js`, and `model`, `view`, `store`, and so on.

In the `application.js` file, you'll see some code as follows:

```
Ext.define('MyApp.Application', {
  extend: 'Ext.app.Application',
  name: 'MyApp',
  stores: [
  // TODO: add global / shared stores here
  ],
  launch: function () {
    // TODO - Launch the application
  }
});
```

Here, you can see that `MyApp.Application` extends `Ext.app.Application`. The `launch` function is defined in `Ext.app.Application`. This function is called after the page is loaded.

The store in `application.js` is nothing, but data stores. You'll learn about stores in detail later in the upcoming chapters.

View model – MainModel.js

Take a look at `MainModel.js` under `\app\view\main\`. This class is the view model for the `Main` view of the application. The `view` model extends from `Ext.app.ViewModel`, as shown in the following code:

```
Ext.define('MyApp.view.main.MainModel', {
  extend: 'Ext.app.ViewModel',

  alias: 'viewmodel.main',

  data: {
    name: 'MyApp',
    loremIpsum: 'Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor incididunt ut labore et
dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in culpa qui officia
deserunt mollit anim id est laborum.'
  }
});
```

Controller – MainController.js

This class is the view controller for the `main` view of the application. In the following code, you can see that the `onItemSelected` function is defined; this will be called when an item is selected from the grid in the view.

```
Ext.define('MyApp.view.main.MainController', {
  extend: 'Ext.app.ViewController',

  alias: 'controller.main',

  onItemSelected: function (sender, record) {
    Ext.Msg.confirm('Confirm', 'Are you sure?', 'onConfirm',
this);
  },

  onConfirm: function (choice) {
    if (choice === 'yes') {
      //
    }
  }
});
```

There are two types of controllers: `Ext.app.ViewController` and `Ext.app.Controller`.

You'll learn about its difference and usage later in the upcoming chapters.

View – main.js

If you have used Sencha Cmd 6, and if you generated the app only for classic or modern toolkits with `--modern` or `--classic`, then you'll find the `main.js` file under the `\app\view\main\`folder, but if you have used Sencha Cmd 6 to generate a universal application, then you can find two `main.js` files located under two paths: `\modern\src\view\main\` and `\classic\src\view\main\`.

Before we see the contents of this file, let's go through the background behind these two `main.js` files under two different paths.

Earlier in this chapter, you learned how Ext JS 6 merges Sencha Ext JS and Sencha Touch into one framework. As a result, a single framework is created with two toolkits.

The core of both these frameworks are moved to a common library, and they split the rest of the code into two parts: classic and modern. The traditional Ext JS code moved to the classic toolkit, and the modern code that supports touch and HTML5 are moved to the modern toolkit.

Applications that share core resources and logic and utilize both the toolkits are called universal applications.

Now, let's take a look at the `main.js` view file under modern:

```
Ext.define('MyApp.view.main.Main', {
  extend: 'Ext.tab.Panel',
  xtype: 'app-main',

  requires: [
  'Ext.MessageBox',
  'MyApp.view.main.MainController',
  'MyApp.view.main.MainModel',
  'MyApp.view.main.List'
  ],

  controller: 'main',
  viewModel: 'main',

  defaults: {
    styleHtmlContent: true
  },

  tabBarPosition: 'bottom',

  items: [
  {
    title: 'Home',
    iconCls: 'fa-home',
    layout: 'fit',
    items: [{
      xtype: 'mainlist'
    }]
  },{
    title: 'Users',
    iconCls: 'fa-user',
    bind: {
      html: '{loremIpsum}'
    }
  },{
    title: 'Groups',
    iconCls: 'fa-users',
    bind: {
      html: '{loremIpsum}'
    }
  },{
    title: 'Settings',
    iconCls: 'fa-cog',
    bind: {
      html: '{loremIpsum}'
    }
  }
  ]
});
```

This sample view defines `controller`, `viewmodel`, and other dependency classes required, creates four tabs, and binds the `loremIpsum` property of `ViewModel`. You'll learn more about this in detail in the upcoming chapters.

Now, let's take a look at `main.js` under `\classic\src\view\main\`:

```
Ext.define('NewApp.view.main.Main', {
  extend: 'Ext.tab.Panel',
  xtype: 'app-main',

  requires: [
  'Ext.plugin.Viewport',
  'Ext.window.MessageBox',

  'NewApp.view.main.MainController',
  'NewApp.view.main.MainModel',
  'NewApp.view.main.List'
  ],

  controller: 'main',
  viewModel: 'main',

  ui: 'navigation',
  tabBarHeaderPosition: 1,
  titleRotation: 0,
  tabRotation: 0,

  header: {
    layout: {
      align: 'stretchmax'
    },
    title: {
      bind: {
        text: '{name}'
      },
      flex: 0
    },
    iconCls: 'fa-th-list'
  },

  tabBar: {
    flex: 1,
    layout: {
      align: 'stretch',
      overflowHandler: 'none'
    }
  },

  responsiveConfig: {
    tall: {
      headerPosition: 'top'
    },
    wide: {
      headerPosition: 'left'
    }
  },

  defaults: {
    bodyPadding: 20,
    tabConfig: {
      plugins: 'responsive',
      responsiveConfig: {
        wide: {
          iconAlign: 'left',
          textAlign: 'left'
        },
        tall: {
          iconAlign: 'top',
          textAlign: 'center',
          width: 120
        }
      }
    }
  },

  items: [{
    title: 'Home',
    iconCls: 'fa-home',
    items: [{
      xtype: 'mainlist'
    }]
  }, {
    title: 'Users',
    iconCls: 'fa-user',
    bind: {
      html: '{loremIpsum}'
    }
  }, {
    title: 'Groups',
    iconCls: 'fa-users',
    bind: {
      html: '{loremIpsum}'
    }
  }, {
    title: 'Settings',
    iconCls: 'fa-cog',
    bind: {
      html: '{loremIpsum}'
    }
  }]`
});
```

In the preceding code, you can see that the content of items is almost the same as in the modern toolkit. Additionally, you can see that this file has some config that is specific to responsive design. The following line in the preceding code tells the framework to use the `ui` style component called `navigation`:

```
ui: 'navigation'
```

You'll learn more about the UI config and responsive design later in *Chapter 8, Theming and Responsive Design*.

Similarly, if you take a look at `List.js` under classic and modern, you'll only find some minor differences.

Exploring Sencha Cmd commands

Now, let's explore some of the useful commands in Sencha Cmd.

The Sencha command format

Sencha commands take the following format:

```
sencha [category] [command] [options...] [arguments…]
```

There are many commands and options available in Sencha Cmd. Let's take a look at some of the important commands.

Help

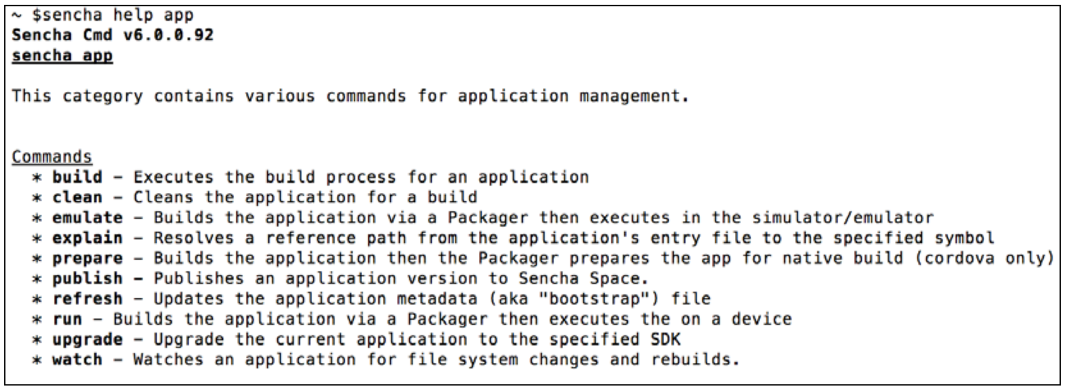
If you simply type the following command, you'll get a list of categories, a list of top-level commands, and options that are available:

```
sencha help
```

To get help on a particular category type, the category name, followed by help, for example, to get help on a category app, run the following command:

```
sencha help app
```

This will produce the following output:



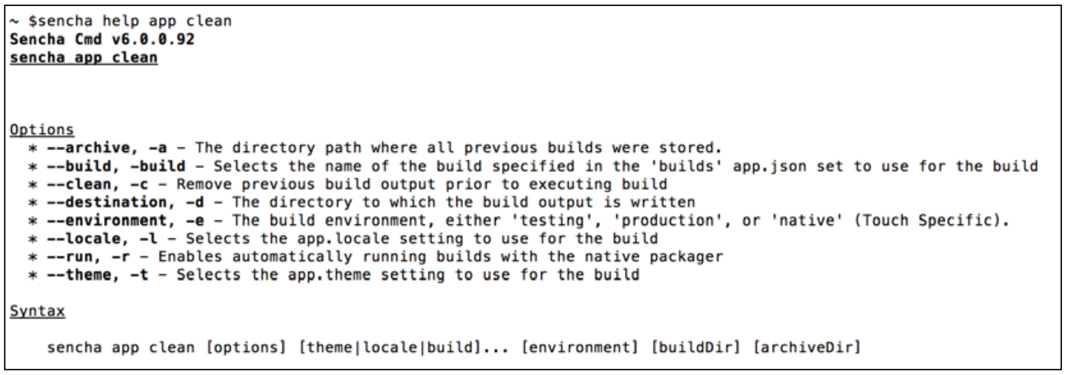
```
~ $sencha help app
Sencha Cmd v6.0.0.92
sencha app

This category contains various commands for application management.


Commands
  * build - Executes the build process for an application
  * clean - Cleans the application for a build
  * emulate - Builds the application via a Packager then executes in the simulator/emulator
  * explain - Resolves a reference path from the application's entry file to the specified symbol
  * prepare - Builds the application then the Packager prepares the app for native build (cordova only)
  * publish - Publishes an application version to Sencha Space.
  * refresh - Updates the application metadata (aka "bootstrap") file
  * run - Builds the application via a Packager then executes the on a device
  * upgrade - Upgrade the current application to the specified SDK
  * watch - Watches an application for file system changes and rebuilds.
```

Figure 1.6

If you further want to get help on subcategory commands under app, you can just add the command at the end for clean, as shown in the following code:

```
sencha help app clean
```

This will give you the following output:



```
~ $sencha help app clean
Sencha Cmd v6.0.0.92
sencha app clean



Options
  * --archive, -a - The directory path where all previous builds were stored.
  * --build, -build - Selects the name of the build specified in the 'builds' app.json set to use for the build
  * --clean, -c - Remove previous build output prior to executing build
  * --destination, -d - The directory to which the build output is written
  * --environment, -e - The build environment, either 'testing', 'production', or 'native' (Touch Specific).
  * --locale, -l - Selects the app.locale setting to use for the build
  * --run, -r - Enables automatically running builds with the native packager
  * --theme, -t - Selects the app.theme setting to use for the build

Syntax

    sencha app clean [options] [theme|locale|build]... [environment] [buildDir] [archiveDir]
```

Figure 1.7

Upgrading Sencha Cmd

If you want to check whether there are any updates available for Sencha Cmd, use the following command:

```
sencha upgrade --check
```

If you want to upgrade Sencha Cmd, just remove the `--check` option, as shown in the following code:

```
sencha upgrade
```

Generating an application

Sencha Cmd supports Ext JS 4.1.1a and higher and Sencha Touch 2.1 and higher. You can have multiple versions of the SDK on your machine. Here is the format of the command to generate an application based on a Sencha SDK, such as Ext JS or Sencha Touch:

```
sencha -sdk /path/to/sdk generate app [--modern/classic] MyApp /path/to/
myapp
```

This is an example that will generate the Ext JS 6 application called `MyApp` under the `/Users/SomeUser/projects/extjs/myapp` path:

```
sencha -sdk /Users/SomeUser/bin/Sencha/Cmd/repo/extract/ext/6.0.0/
generate app MyApp /Users/SomeUser/projects/extjs/myapp
```

Building the application

The following command will build HTML, JS, SASS, and so on:

```
sencha app build
```

In Sencha Cmd 6 and Ext JS 6, you can also run one of the following code to choose either modern or classic:

```
sencha app build modern
sencha app build classic
```

Here, `modern` and `classic` refers to the builds config in `app.json`. By default, Sencha Cmd puts two builds config: classic and modern in `app.json`; you can add the additional build config if required.

Launching the application

The watch command can be used to rebuild and launch the application. This will not only launch the application, but also monitor any code changes made, and as soon as the code changes are saved, it will refresh the browser to include the updated code changes as follows:

```
sencha app watch
```

In Sencha Cmd 6 and Ext JS 6, you can also run one of the following code to choose either modern or classic:

```
sencha app watch modern
sencha app watch classic
```

The code generation

With Sencha Cmd, you can generate the Ext JS code, such as views, controller, and model:

```
sencha generate view myApp.MyView
sencha generate model MyModel id:int,fname,lname
sencha generate controller MyController
```

If the field type is not specified when you generate the model, the default field type string will be used.

Upgrading your application

Sencha Cmd makes upgrading from one version of the SDK to another version easy. Use the upgrade command in the `app` category to upgrade to the new framework:

```
sencha app upgrade [ path-to-new-framework ]
```

Debugging an Ext JS application

You can use the browser's default debugger to debug the Ext JS code, but the debugging Ext JS code is much easier with a Firefox plugin called Illumination or the App Inspector plugin for Chrome.

Illumination

Illumination is a third-party tool. It is not a product of Sencha, and right now, it is only available for Firefox, and it requires Firebug.

The features of Illumination

Here are the some of the features of the Illumination plugin. This will reduce the amount of time you spent in debugging.

Object naming

Illumination will recognize the Ext JS components easily, so in the illumination tab you'll see Ext JS component names such as `Ext.panel.Panel` instead of showing `Object` as in Firebug's DOM tab.

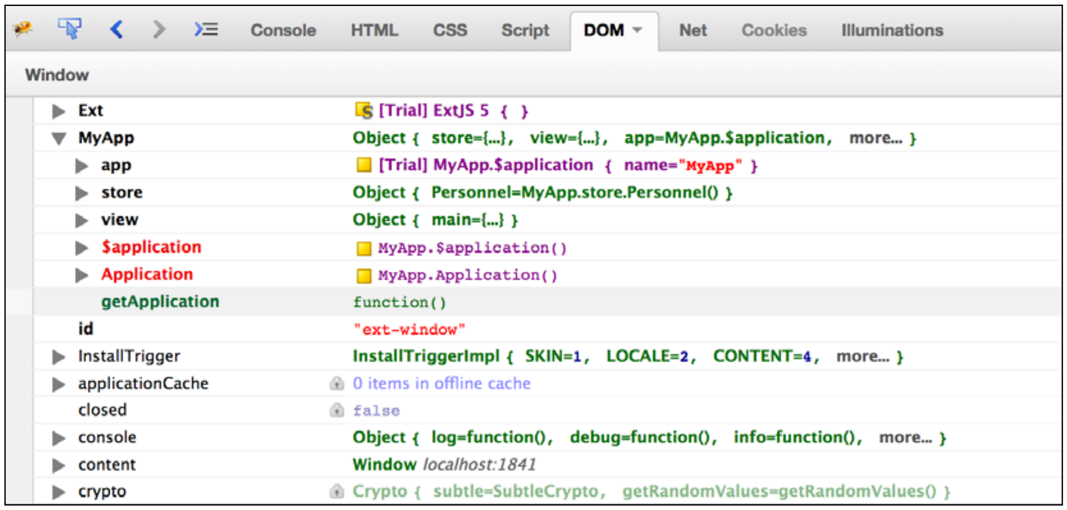
Element highlighting

If you hover over any of the objects in the Illumination window, it will highlight the whole component in the HTML page.

The contextual menu

An Ext JS component is composed of several HTML elements. If you right-click on the page and select the Firebug context menu, you'll be taken to the element nested in the Ext JS component, but if you select the Illumination context menu, it will show you the Ext JS component that makes it easier to examine the component and its methods, properties, and events.

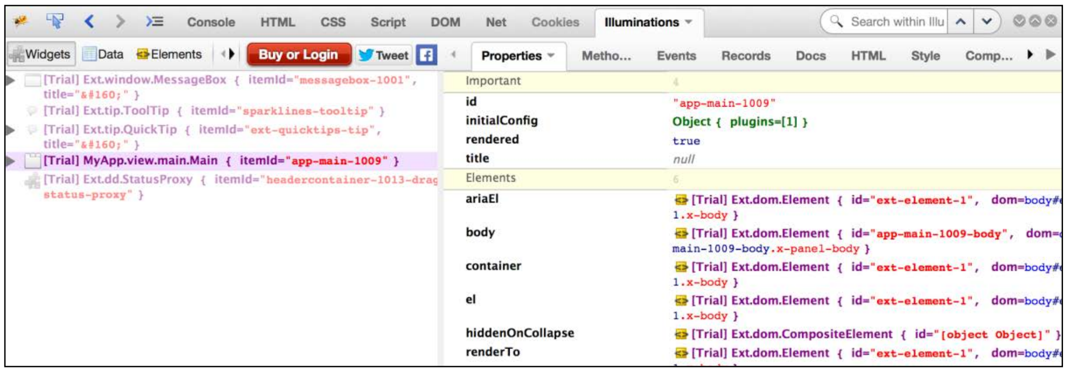
Check the Firebug **DOM** tab in *Figure 1.8* and see how the objects are represented:



```
Window
▶ Ext                  [Trial] ExtJS 5 { }
▼ MyApp                Object { store={...}, view={...}, app=MyApp.$application, more... }
  ▶ app                [Trial] MyApp.$application { name="MyApp" }
  ▶ store              Object { Personnel=MyApp.store.Personnel() }
  ▶ view               Object { main={...} }
  ▶ $application       MyApp.$application()
  ▶ Application        MyApp.Application()
    getApplication     function()
  id                   "ext-window"
▶ InstallTrigger       InstallTriggerImpl { SKIN=1, LOCALE=2, CONTENT=4, more... }
▶ applicationCache     0 items in offline cache
  closed               false
▶ console              Object { log=function(), debug=function(), info=function(), more... }
▶ content              Window localhost:1841
▶ crypto               Crypto { subtle=SubtleCrypto, getRandomValues=getRandomValues() }
```

Figure 1.8

Now, check the Illumination tab in *Figure1.9* and see how the objects are represented. You can see that Illumination recognizes all the Ext JS component, as shown in the following screenshot:



```
▶ [Trial] Ext.window.MessageBox { itemId="messagebox-1001",
  title="&#160;" }
  [Trial] Ext.tip.ToolTip { itemId="sparklines-tooltip" }
▶ [Trial] Ext.tip.QuickTip { itemId="ext-quicktips-tip",
  title="&#160;" }
▶ [Trial] MyApp.view.main.Main { itemId="app-main-1009" }
  [Trial] Ext.dd.StatusProxy { itemId="headercontainer-1013-drag
  status-proxy" }
```

```
Important          4
id                 "app-main-1009"
initialConfig      Object { plugins=[1] }
rendered           true
title              null
Elements           6
ariaEl             [Trial] Ext.dom.Element { id="ext-element-1", dom=body#
                   1.x-body }
body               [Trial] Ext.dom.Element { id="app-main-1009-body", dom=
                   main-1009-body.x-panel-body }
container          [Trial] Ext.dom.Element { id="ext-element-1", dom=body#
                   1.x-body }
el                 [Trial] Ext.dom.Element { id="ext-element-1", dom=body#
                   1.x-body }
hiddenOnCollapse   [Trial] Ext.dom.CompositeElement { id="[object Object]" }
renderTo           [Trial] Ext.dom.Element { id="ext-element-1", dom=body#
```

Figure 1.9

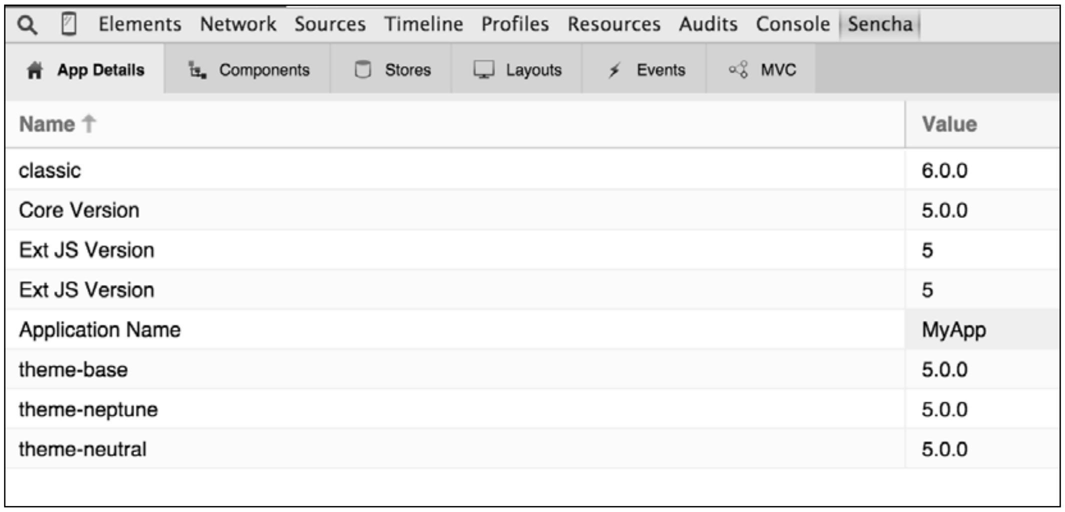
Although Illumination makes the debugging of the Ext JS application easier, it is not a must. Illumination is not a free tool. So, if you do not want to pay for it, you can still use Firebug to debug, but you may need to spend a little bit of extra time to debug, or you may need to take a look at App Inspector or Sencha Fiddle.

App Inspector

App Inspector is a free Chrome plugin developed by Sencha. It also provides all the features provided by Illumination. Some of the features provided are component inspector, store inspector, and layout profiles.

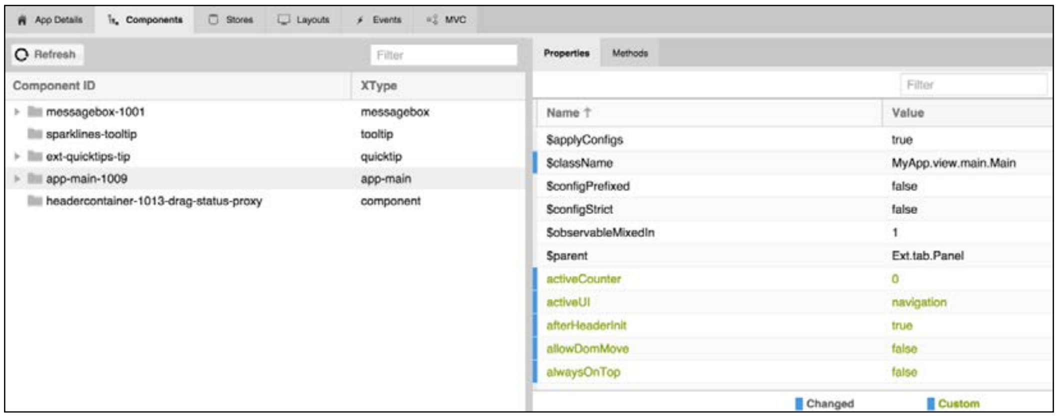
Some information is easier to find in App Inspector than Illumination, and debugging with the Illumination Ext JS application takes a long time to load than debugging with App Inspector.

Figure 1.10 and *Figure 1.11* shows couple of tabs in App Inspector:



Name ↑	Value
classic	6.0.0
Core Version	5.0.0
Ext JS Version	5
Ext JS Version	5
Application Name	MyApp
theme-base	5.0.0
theme-neptune	5.0.0
theme-neutral	5.0.0

Figure 1.10



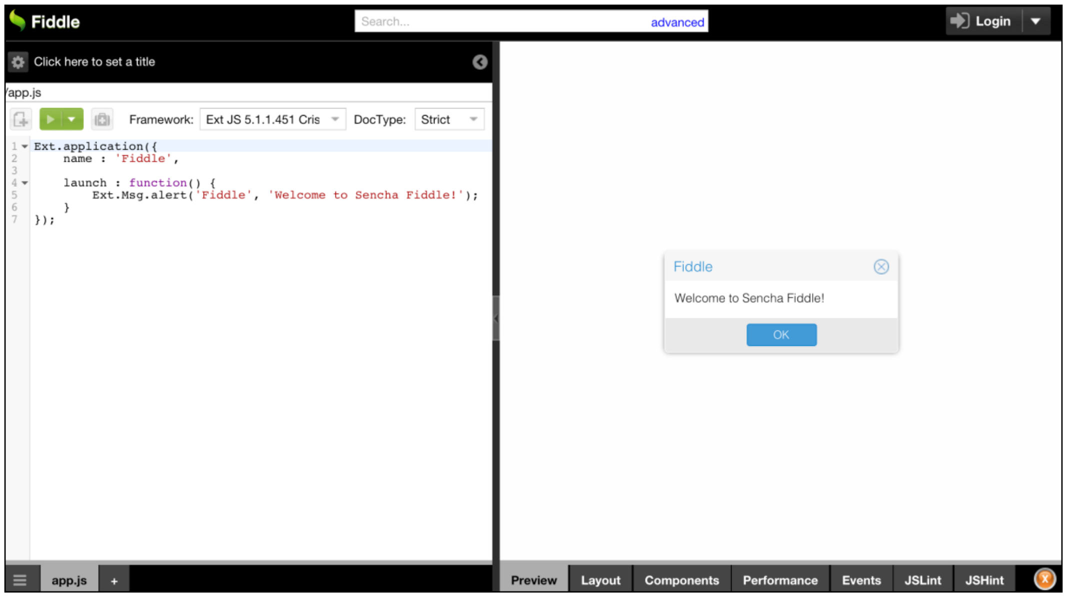
Component ID	XType
messagebox-1001	messagebox
sparklines-tooltip	tooltip
ext-quicktips-tip	quicktip
app-main-1009	app-main
headercontainer-1013-drag-status-proxy	component

Name ↑	Value
$applyConfigs	true
$className	MyApp.view.main.Main
$configPrefixed	false
$configStrict	false
$observableMixedIn	1
$parent	Ext.tab.Panel
activeCounter	0
activeUI	navigation
afterHeaderInit	true
allowDomMove	false
alwaysOnTop	false

Figure 1.11

Sencha Fiddle

This is another debugging tool that may be helpful. It is also an online web-based IDE that provides some debugging capabilities, as shown in *Figure 1.12*:



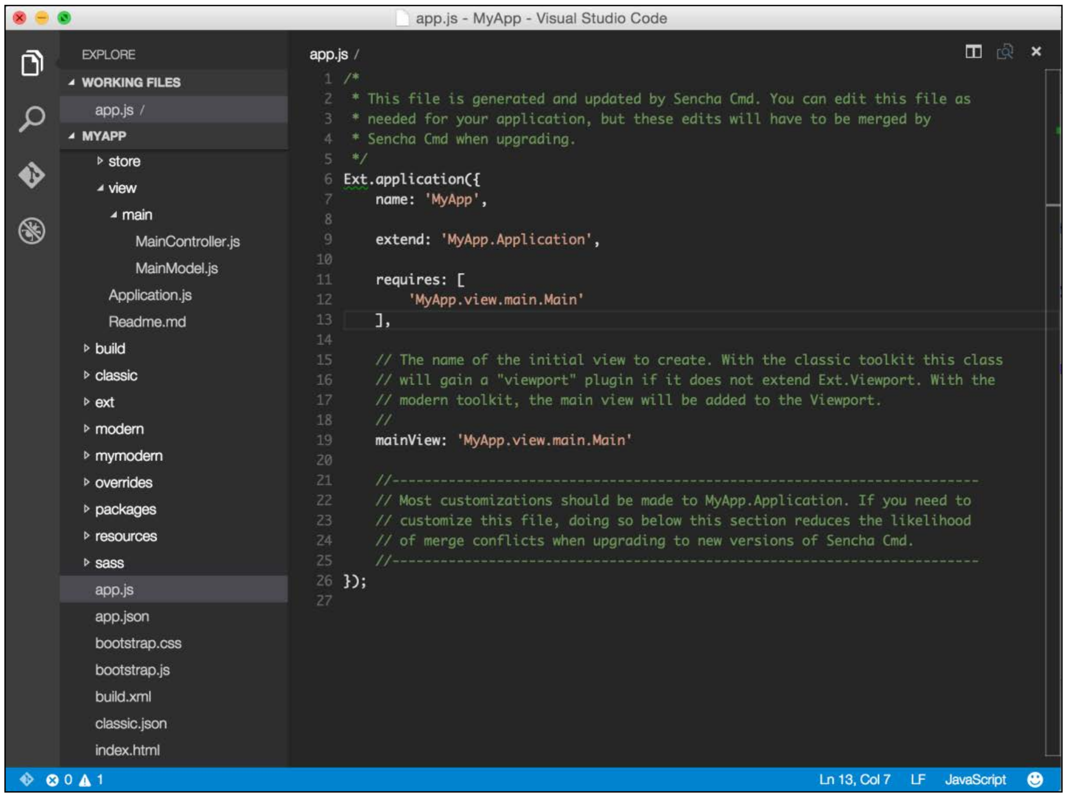
```
Ext.application({
    name : 'Fiddle',

    launch : function() {
        Ext.Msg.alert('Fiddle', 'Welcome to Sencha Fiddle!');
    }
});
```

Figure 1.12

Tha development IDE

Although you can use any simple text editors to write the Ext JS code, using the IDEs definitely makes it a bit easier. Sencha provides the Sencha JetBrains plugin for JetBrains products, such as IntelliJ, WebStrome, PHPStorm, and RubyMine.

If you're looking for some free and simple IDE, then take a look at Visual Studio Code and Brackets.io. Both of these are extremely lightweight and available for Mac, Windows, and Linux. *Figure 1.13* shows **Visual Studio Code**:



```
/*
 * This file is generated and updated by Sencha Cmd. You can edit this file as
 * needed for your application, but these edits will have to be merged by
 * Sencha Cmd when upgrading.
 */
Ext.application({
    name: 'MyApp',

    extend: 'MyApp.Application',

    requires: [
        'MyApp.view.main.Main'
    ],

    // The name of the initial view to create. With the classic toolkit this class
    // will gain a "viewport" plugin if it does not extend Ext.Viewport. With the
    // modern toolkit, the main view will be added to the Viewport.
    //
    mainView: 'MyApp.view.main.Main'

    //-------------------------------------------------------------------------
    // Most customizations should be made to MyApp.Application. If you need to
    // customize this file, doing so below this section reduces the likelihood
    // of merge conflicts when upgrading to new versions of Sencha Cmd.
    //-------------------------------------------------------------------------
});
```

Figure 1.13

Summary

In this chapter, we looked at some of the advantages with JavaScript frameworks instead of using plain JavaScript. We also looked at some of the famous JavaScript frameworks. You learned how to set up a development environment for the Ext JS application, and we scaffolded an Ext JS application with Sencha Cmd.

In the next chapter, you'll learn the core concepts and basics of Ext JS.

2
Core Concepts

Before we start building a sample project in the next chapter, you'll learn some core concepts in Ext JS, which will help you to understand the sample project easily. The following topics will be covered in this chapter:

- The class system, and creating and extending class
- Events
- Querying
- Containers
- Layouts

The class system

Ext JS provides a number of functions that make it simple to create and work with classes.

The following are the classes in the Ext JS 6 class system:

- `Ext`
- `Ext.Base`
- `Ext.Class`
- `Ext.ClassManager`
- `Ext.Loader`

Ext

`Ext` is a global singleton object that encapsulates all classes, singletons, and utility methods in the Sencha library. Many commonly used utility functions are defined in `Ext`. It also provides shortcuts to frequently used methods in other classes.

Let's take a look at some of the methods and properties in the `Ext` class:

application

Many applications are initiated with `Ext.application`. This function loads the `Ext.app.Application` class and starts it with the given configuration after the page is loaded.

`Ext.app.Application` is a class that represents the entire application that we saw in *Chapter 1, Getting Started with Ext JS*. The following is an example of `Ext.app.Application`:

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

This code creates a global variable called `MyApp`. All the application's classes will reside under this single namespace, which will reduce the chances of colliding global variables.

define

To create or override a class, you can use this function. It takes three parameters, as shown in the following code. Here, `name` is the name of the class, `data` is the properties to apply to this class, and `callback` is an optional function that will be called after the class is created:

```
Ext.define(name,data, callback)
```

The following code creates a class called `Car`:

```
Ext.define('Car', {
  name: null,
  constructor: function(name) {
    if (name) {
      this.name = name;
    }
  },
  start: function() {
    alert('Car started');
  }
});
```

You can also use `define` to extend a class:

```
Ext.define('ElectricCar', {
  extend: 'Car',
  start: function() {
    alert("Electric car started");
  }
});
```

If you want to replace the implementation of a base class, you can use `Ext.define` to override the method, as shown in the following code:

```
Ext.define('My.ux.field.Text', {
  override: 'Ext.form.field.Text',
  setValue: function(val) {
    this.callParent(['In override']);
    return this;
  }
});
```

If you want to create a singleton class, then you can use a property called singleton defined in `Ext.Class`, as shown in the following code:

```
Ext.define('Logger', {
  singleton: true,
  log: function(msg) {
    console.log(msg);
  }
});
```

create

You can use the following signature to create an instance of a class:

```
Ext.create(Class,Options);
```

The following code creates an instance of the `ElectricCar` class and passes a value:

```
var myCar = Ext.create('ElectricCar',{
    name: 'MyElectricCar'
});
```

If `Ext.Loader` is enabled, `Ext.create` will automatically download the appropriate JavaScript file if the `ElectricCar` class does not exist. By default, the `Ext.Loader` is enabled; you can disable it by setting the enabled config in `Ext.Loader` to `false`.

You can also use the new keyword to create an instance, as shown in the following code; however, if the class has not been defined yet, the new keyword won't download the appropriate JavaScript file, in which the class is defined:

```
var myCar = new ElectricCar('MyElectricCar');
```

onReady

This function is called once the page is loaded:

```
Ext.onReady(function(){
  new Ext.Component({
    renderTo: document.body,
    html: 'DOM ready!'
  });
});
```

In most cases, you don't have to use the `onReady` method in your code. Rarely, in some special cases, you may need to use it. If you're from a jQuery background, do not try to use `onReady` frequently as you can do the same in jQuery with `$( document ).ready()`.

widget

When you define a class, you can give a shorthand alias. For example, the `Ext.panel.Panel` has an alias called `widget.panel`. When you define widgets, instead of specifying the alias, you can also use `xtype` to give a shorthand name for the class.

The `xtype` is very useful to specify the widgets that a container will have without creating the instance when the definition of your container gets executed. You'll learn more about this later in this chapter when you learn about containers and layouts.

`Ext.widget` is the shorthand to create a widget by its `xtype`.

For example, without using the widget method, you can create an instance of `Ext.panel.Panel` with the following code:

```
Ext.create('Ext.panel.Panel', {
  renderTo: Ext.getBody(),
  title: 'Panel'
});
```

Instead, you can use the following shorthand to create the instance:

```
Ext.widget('panel', {
  renderTo: Ext.getBody(),
  title: 'Panel'
});
```

Here, the panel is a container, we'll learn more about the panel later. The below one equivalent to the above:

```
Ext.create('widget.panel', {
  renderTo: Ext.getBody(),
  title: 'Panel'
});
```

Note that as you read, you can execute this sample code and most of the other sample code that we will see in this book. You can either execute them on your local machine or easily on Sencha Fiddle. You can visit Sencha Fiddle at `https://fiddle.sencha.com` and put the preceding code in the launch function, run it, and see the result. So, if you go to `https://fiddle.sencha.com`, you'll see the following code:

```
Ext.application({
  name : 'Fiddle',
  launch : function() {
    Ext.Msg.alert('Fiddle', 'Welcome to Sencha Fiddle!');
  }
});
```

Now, paste the panel widget sample code as shown below, run it, and see the result. Just be sure that the quotes symbol is (`'`) and not (‘). Sometimes, when it's copied and pasted, the symbol changes:

```
Ext.application({
  name : 'Fiddle',
  launch : function() {
    Ext.create('widget.panel', {
      renderTo: Ext.getBody(),
      title: 'Panel'
    });
  }
});
```

Not all the sample code can be executed in this way. Also, not all the sample code have visual rendering.

getClass

This returns the class of the given object if the instance is created with `Ext.define`; otherwise, it returns null:

```
var button = new Ext.Button();
Ext.getClass(button);  // returns Ext.Button
```

getClassName

This returns the name of the class by its reference or its instance:

```
Ext.getClassName(Ext.Button); //returns "Ext.Button"
```

Ext.Base

This is the base of all `Ext` classes. All classes in `Ext` inherit from `Ext.Base`. All prototype and static members of this class are inherited by all other classes.

Ext.Class

This is a low-level factory that is used by `Ext.ClassManager` for the creation of a class. So, this shouldn't be accessed directly in your code; instead, you should use `Ext.define`.

Ext.ClassManager

This manages all the classes and handles mapping from a string class name to the actual class objects. It is generally accessed through these shorthands:

- `Ext.define`
- `Ext.create`
- `Ext.widget`
- `Ext.getClass`
- `Ext.getClassName`

We have already discussed these methods in this chapter.

Ext.Loader

This is used for dynamic dependency loading. Normally, the `Ext.require` shorthand is used. When you define a class, it's a good practice to specify the list of components required, as shown in the following code:

```
Ext.require(['widget.window', 'layout.border',
'Ext.data.Connection']);
```

If you need all the components/classes enclosed under a particular namespace, use wildcards, as shown in the following code:

```
Ext.require(['widget.*', 'layout.*', 'Ext.data.*');
```

To exclude what is not required, use the following syntax:

```
Ext.exclude('Ext.data.*').require('*');
```

In this way, the required classes are loaded asynchronously. If you don't specify the required classes in your definition, then when the instance is created using `Ext.Create`, the class file will be loaded synchronously if it's not loaded already. This has a little bit of an impact on performance. So, when you define your class, it is always better to specify the required classes using `Ext.require`.

Note that the file path to locate a class is calculated based on the name of the class. For example, the file path of the `MyApp.view.About` class should be `\myapp\view\about.js`.

Events

An event could be a user action, a response to an Ajax call, and so on.

Adding listeners

You can add listeners when you create the object or later. The following example adds a listener for the click event when the object is created:

```
Ext.create('Ext.Button', {
  renderTo: Ext.getBody(),
  listeners: {
    click: function() {
      Ext.Msg.alert('Button clicked!');
    }
  }
});
```

You can add listeners to multiple events, as shown in the following code:

```
Ext.create('Ext.Button', {
  renderTo: Ext.getBody(),
  listeners: {
    mouseout: function() {
      //Do something
    },
    click: function() {
      // Do something
    }
  }
});
```

Also, you can add listeners after the instance is created with the `on` method:

```
var button = Ext.create('Ext.Button');

button.on('click', function() {
    //Do something
});
```

You can also add multiple listeners with the `.on` method, as shown in the following code:

```
var button = Ext.create('Ext.Button');

button.on({
  mouseover: function() {
    //Do something
  },
  mouseover: function() {
    //Do something
  }
});
```

Removing listeners

You can also remove the listeners, but you need the reference to the function; you can't use the anonymous function.

```
var HandleClick= function() {
  Ext.Msg.alert('My button clicked!');
}

Ext.create('Ext.Button', {
  listeners: {
    click: HandleClick
  }
});

button.un('click', HandleClick);
```

The DOM node event handling

You can add listeners to the DOM element, as shown here.

Let's say that in your HTML, you have a `div` element with `id=mydiv`, as shown in the following code:

```
<div id="mydiv"></div>
```

Now, to add the listener, use the following code:

```
var div = Ext.get('mydiv');
div.on('click', function(e, t, eOpts) {
 // Do something
});
```

Accessing DOM

There are three ways to access DOM elements: `get`, `query`, and `select`.

Ext.get

This takes the ID of a DOM node and retrieves the element wrapped as `Ext.dom.Element`:

```
var mydiv = Ext.get('myDivId');
```

Ext.query

This selects the child nodes of a given root based on the passed CSS selector. It returns an array of elements (`HTMLElement[]/Ext.dom.Element[]`) that match the selector. If there are no matches, an empty array is returned.

In the following example, `myCustomComponent.getEl().dom` is passed as a root node. `Ext.query` will search the children of root and returns an array containing only the elements which has the CSS class '`oddRow`':

```
var someNodes = Ext.query('.oddRow',
myCustomComponent.getEl().dom);
```

Ext.select

Given some CSS/XPath selector, `Ext.select` returns a single object of type `CompositeElement`, which represents a collection of elements.

This `CompositeElement` has methods to filter, iterate, and perform collective actions on the whole set, and so on:

```
var rows = Ext.select('div.row'); ////Matches all divs with class
row
rows.setWidth(100); // All elements become 100 width
```

You can also combine both the lines to a single line, as shown in the following code:

```
Ext.select('div.row').setWidth(100);
```

Multiple selections

This can be used to select multiple sets of elements by specifying the multiple search criteria in one single method call:

```
Ext.select('div.row, span.title'); //Matches all divs with class
row and all spans with class title
```

Selection root

When you use select, it takes the HTML body as the root and starts searching the entire DOM tree of the body by default. You can avoid this by specifying a root element so that it will only search the children of the given root.

```
Ext.get('myEl').select('div.row');
```

Here this uses 'myEL' root element. This will first find the element having the id 'myEl', and then under that root element, it will search for 'div' tags having the class 'row'.

```
Ext.select('div.row', true, 'myEl');// This is equivalent to the
previous line.
```

Selection chaining

The following query matches `div` with a class of row and has a title attribute bar, which is the first child of its immediate parent:

```
Ext.select('div.row[title=bar]:first')
```

Ext.ComponentQuery

This allows you to find a component with the ID, xtype, and properties. You can search globally, or you can specify a root component.

The following query will return all the components with the xtype `button`:

```
Ext.ComponentQuery.query('button');
```

To get a component with the ID of `foo`, use the following code:

```
Ext.ComponentQuery.query('#foo');
```

The following code will return all the components with the xtype `button` and the `my button` title property:

```
Ext.ComponentQuery.query("button[title='my button']");;

//or

parent.query('textfield[title=my button]');
```

You can also use nested selectors as follows:

```
Ext.ComponentQuery.query('formpanel numberfield'); // Gets only
the numberfields under a form
```

The following code returns the first direct child of the container that matches the passed selector. If there is no match, then a null will be returned.

```
parent.child('button[itemId=save]');
```

Similarly, you can use other methods, such as `nextNode`, `up`, `down`, `previousSibling`, and so on.

Components, containers, and layouts

Ext JS comes with a rich set of components and layouts, which make the UI application development in Ext JS super easy even for a non-UI developer.

Components

From simple components, such as button and label, to complex components, such as Tree Panel, Grids, and so on, Ext JS has an extensive list of built-in components. All the components are derived from `Ext.Component`, which provides supports to create, resize, render, and dispose the components.

All the components have a property called `xtype`. This is very useful when you don't want to instantiate the component immediately, but you want to lazy load it.

Containers

Containers are a special type of component that are capable of holding other components. `Ext.container.Container` is the base class of all the containers in Ext JS.

`Ext.toolbar.Toolbar`, `Ext.panel.Panel`, and `Ext.Editor` are some of the examples of built-in components. These are capable of containing other components. `Ext.button.Button` doesn't derive from `Ext.container.Container`, so it's not capable of containing other components.

A typical Ext JS application contains a set of nested components. Consider this simple example:

```
Ext.create('Ext.panel.Panel', {
  renderTo     : Ext.getBody(),
  width        : 700,
  height       : 400,
  items: [
  {
    xtype: 'panel',
    title: 'Panel 1',
  },
  {
    xtype: 'panel',
    title: 'Panel 2',
    height: 200,
    items: [
    {
      xtype: 'button',
      text: 'Click Me'
    }
    ]
  },
  {
    xtype: 'panel',
    title: 'Panel 3',
    width: 150,
    height: 100,
  }
  ]
});
```

In the preceding code, the components are nested, as shown in the following diagram:

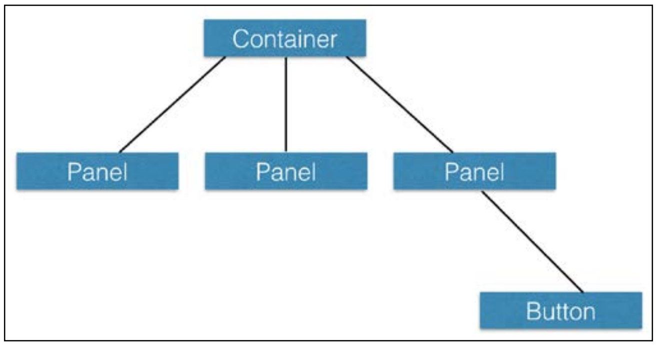


Figure 2.1

The output of the preceding code will look similar to the following screenshot:

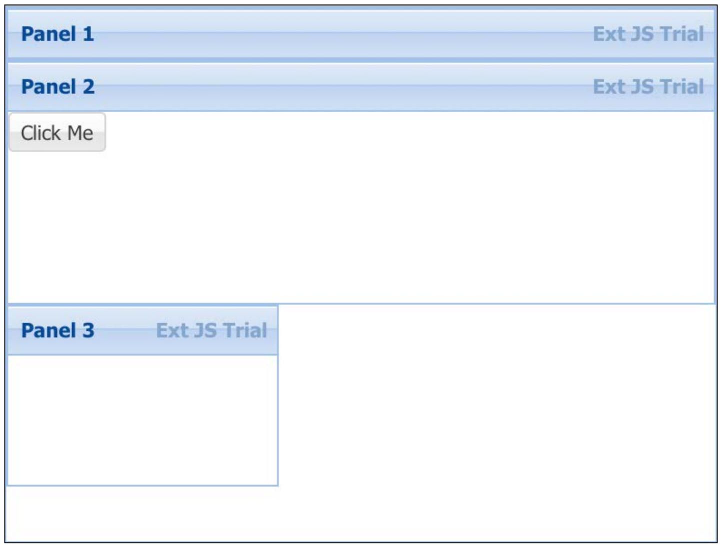


Figure 2.2

Layouts

A layout defines how the contained components are positioned and sized. Every container has a layout. The default layout is auto. This doesn't specify any rules to position and size the child components.

In preceding *Figure 2.2*, you may have noticed that the child components are just nested one after another in the parent container. This is because we haven't specified any layout for these components in the code, and by default, they use the auto layout.

Now, let's use some layouts for the same code. In the following example, we will use the column layout and the center layout.

When you use the column layout, you can specify the `columnWidth`. The sum of values of `columnWidth` of all the columns must be equal to `1`. You can also specify the fixed width for some of the columns, as shown in the following code. Here, `Panel 3` takes a fixed width of `150`, and the remaining two columns shares the remaining width based on the `columnWidth` value:

```
Ext.create('Ext.panel.Panel', {
  renderTo     : Ext.getBody(),
  width        : 700,
  height       : 400,
  layout       : 'column',
  items: [
  {
    xtype: 'panel',
    title: 'Panel 1',
    columnWidth: 0.4,
    height: 400,
  },
  {
    xtype: 'panel',
    title: 'Panel 2',
    columnWidth: 0.6,
    layout: 'center',
    height: 400,
    items: [
    {
      xtype: 'button',
      text: 'Click Me'
    }
    ]
  },
  {
    xtype: 'panel',
    title: 'Panel 3',
    width: 150,
    height: 400,
  }
  ]
});
```

Now, the output will look like this:

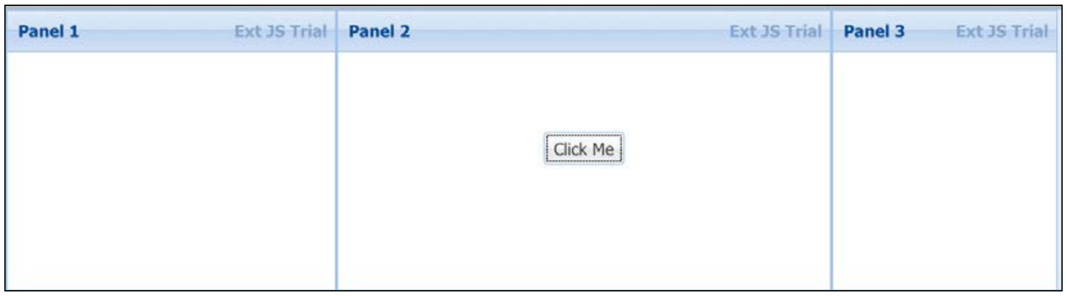


Figure 2.3

updateLayout

The `updateLayout` is a method in `Ext.container.Container`. This can be used to reposition the child components according to the layout rule.

suspendLayout

Most of the time, you don't have to call this `updateLayout` method in your code. However, there are some cases where you have to call it.

The `updateLayout` method is called during the resize and when you add or remove children. Sometimes, you may need to suspend it for a while, especially when you add/remove multiple children subsequently.

So, in these scenarios, you can set the `suspendLayout` property to `true`, and once you finish adding or removing the children, you can set `suspendLayout` to `false` and call the `updateLayout` method in your code.

Also, if you want to suspend the update layout for the whole framework, you can call `Ext.suspendLayouts()`, and after you batch update, you can resume it by calling `Ext.resumeLayouts(true)`.

The following are the list of built-in layouts available in Ext JS:

- `absolute`
- `accordion`
- `anchor`
- `border`
- `card`
- `center`

- `column`
- `fit`
- `hbox`
- `table`
- `vbox`

The absolute layout

This layout defines absolute positioning with `x` and `y` properties:

```
Ext.create('Ext.panel.Panel', {
  renderTo     : Ext.getBody(),
  width        : 700,
  height       : 400,
  layout       : 'absolute',
  items: [
  {
    xtype: 'panel',
    title: 'Panel 1',
    x: 12,
    y: 20,
    height: 250,
  },
  {
    xtype: 'panel',
    title: 'Panel 2',
    x: 200,
    y: 150,
    height: 200,
  },
  {
    xtype: 'panel',
    title: 'Panel 3',
    x: 400,
    y: 250,
    width: 150,
    height: 100,
  }
  ]
});
```

The output is shown here. You can overlap components because they are positioned with absolute position:

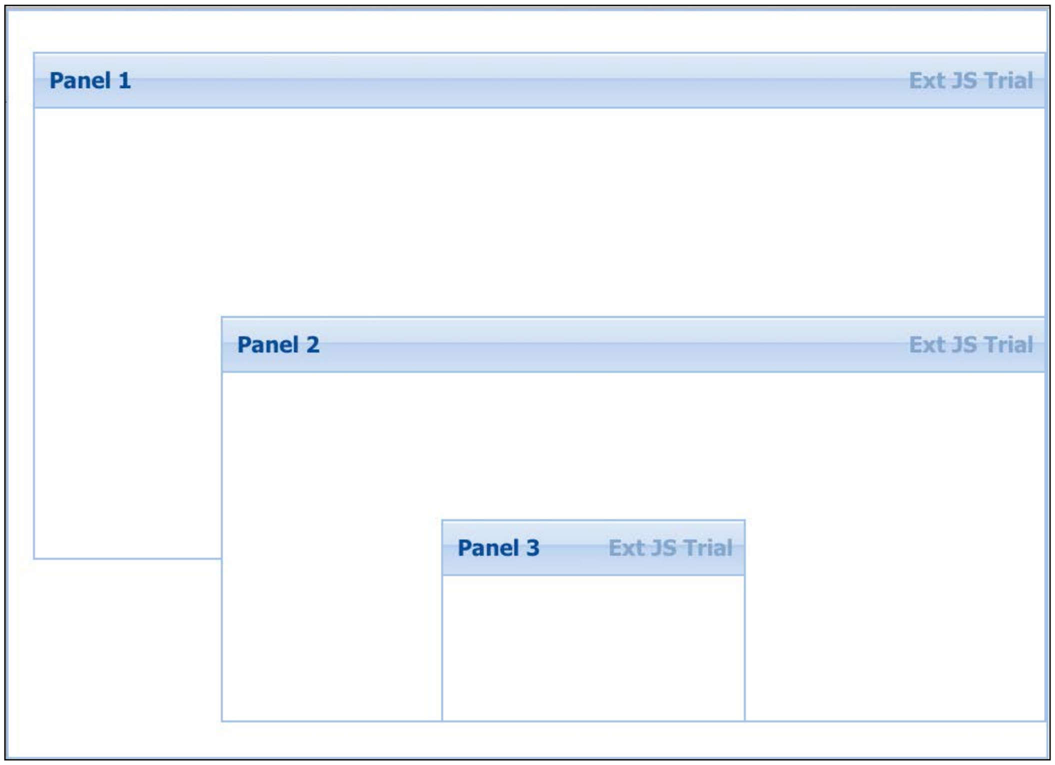


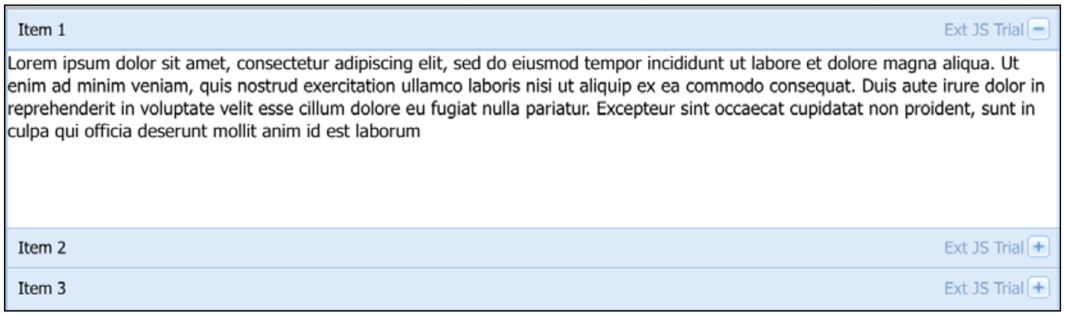
Figure 2.4

The accordion layout

This layout shows only one child panel at a time with the built-in support to collapse and expand. Take a look at the following example:

```
Ext.create('Ext.panel.Panel', {
  renderTo     : Ext.getBody(),
  width        : 700,
  height       : 400,
  layout       : 'accordion',
  items: [
  {
    title: 'Item 1',
    html: 'Lorem ipsum dolor sit amet, consectetur adipiscing
elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi
ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat
nulla
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum'
  },
  {
    title: 'Item 2',
    html: 'some content here'
  },
  {
    title: 'Item 3',
    html: 'empty'
  }
  ]
});
```

The output is shown here. The `Item 1` is expanded, whereas the other panels are collapsed:


Item 1

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

Item 2

Item 3

Figure 2.5

The anchor layout

This layout enables you to specify the size of the child components that are relative to the container. First, the containers are resized and then all the child components are resized according to the anchoring rules specified:

```
Ext.create('Ext.panel.Panel', {
  renderTo     : Ext.getBody(),
  width        : 700,
  height       : 400,
  layout       : 'anchor',
  items: [
  {
    title: 'Item 1',
    html: 'Item 1',
    anchor: '50%'
  },
  {
    title: 'Item 2',
    html: 'Item 2',
    anchor: '-20 -200'
  },
  {
    title: 'Item 3',
    html: 'Item 3',
    anchor: '-200'
  }
  ]
});
```

The output is shown in the following screenshot:

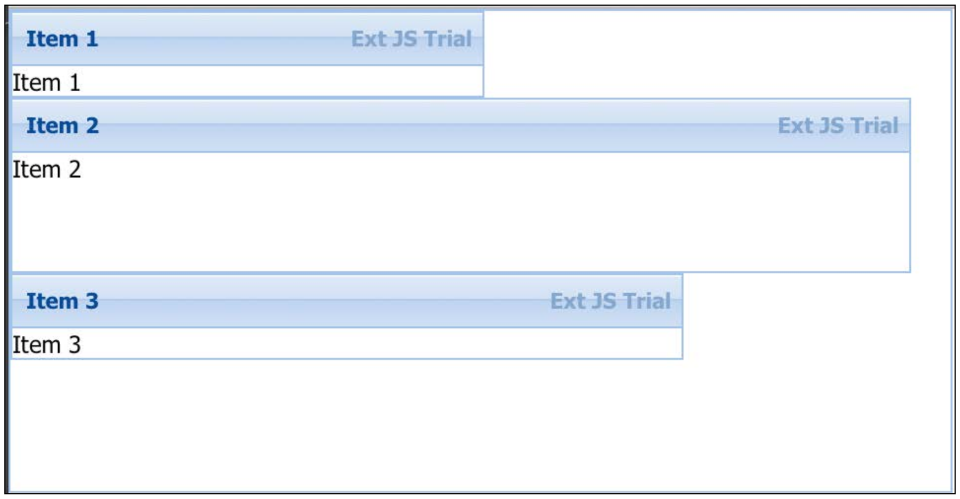


Figure 2.6

The border layout

This layout allows you to specify the position of the child components in terms of regions, such as center, north, south, west, and east. When you use the `border` layout, you must always have one component with a region as center, as shown in the following code:

```
Ext.create('Ext.panel.Panel', {
  renderTo     : Ext.getBody(),
  width        : 700,
  height       : 400,
  layout       : 'border',
  items: [
  {
    title: 'Item 1',
    html: 'Item 1',
    region: 'center'
  },
  {
    title: 'Item 2',
    html: 'Item 2',
    region: 'east',
    width: 200
  },
  {
    title: 'Item 3',
    html: 'Item 3',
    region: 'south',
    height: 100
  }
  ]
});
```

The output of the preceding code will look similar to the following figure:

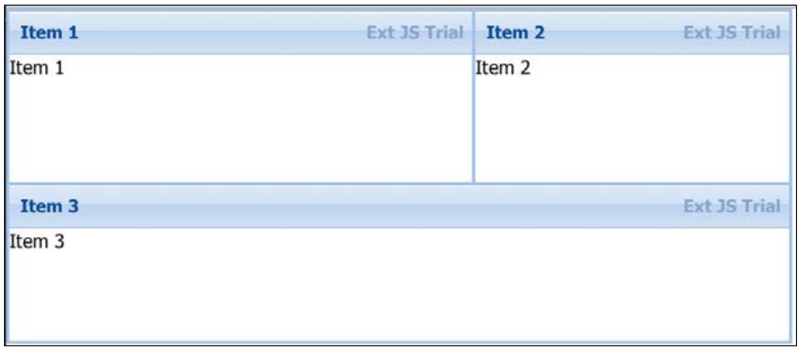


Figure 2.7

The card layout

In this layout, only one child component will be visible, which will fill almost the entire container. The `card` layout is used in wizard and tabs:

```
Ext.create('Ext.panel.Panel', {
  renderTo      : Ext.getBody(),
  width         : 700,
  height        : 400,
  layout        : 'card',
  defaultListenerScope: true,
  bbar: ['->',
  {
    itemId: 'btn-prev',
    text: 'Previous',
    handler: 'showPrevious',
    disabled: true
  },
  {
    itemId: 'btn-next',
    text: 'Next',
    handler: 'showNext'
  }
  ],
  items: [
  {
    index: 0,
    title: 'Item 1',
    html: 'Item 1'
  },
  {
    index: 1,
    title: 'Item 2',
    html: 'Item 2'
  },
  {
    index:2,
    title: 'Item 3',
    html: 'Item 3'
  }
  ],
  showNext: function () {
    this.navigate(1);
  },
  showPrevious: function () {
    this.navigate(-1);
  },
  navigate: function (incr) {
    var layout = this.getLayout();
    var index = layout.activeItem.index + incr;
    layout.setActiveItem(index);

    this.down('#btn-prev').setDisabled(index===0);
    this.down('#btn-next').setDisabled(index===2);
  }
});
```

The output of the card (wizard) layout is shown here. When you click on the **Next** button, it will show the `Item 2` panel:

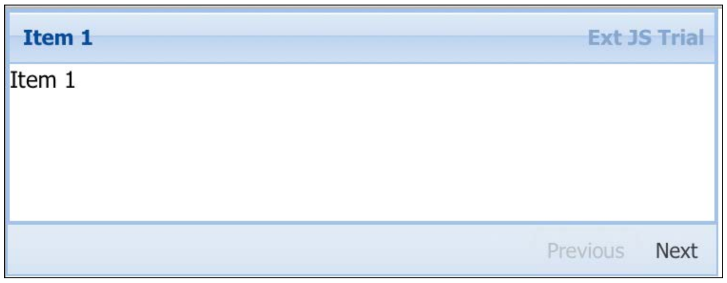


Figure 2.8

The center layout

This layout places the child in the center of the container. We already saw an example in this chapter at the beginning of the layouts section.

The column layout

With this layout, you can split the container into columns of a specific size and place the child components in these columns. We already saw an example in this chapter at the beginning of the layouts section.

The fit Layout

In this layout, the child fits the container's dimension as follows:

```
    Ext.create('Ext.panel.Panel', {
      renderTo      : Ext.getBody(),
      width         : 700,
      height       : 400,
      layout       : 'fit',
      bodyPadding: 20,
      items: [
      {
        title: 'Item 1',
        html: 'Fills the container',
      }
      ]
    });
```

The output of the preceding code is shown here. Note that the gap between the child and parent component is created by `bodyPadding` that we set in the code:

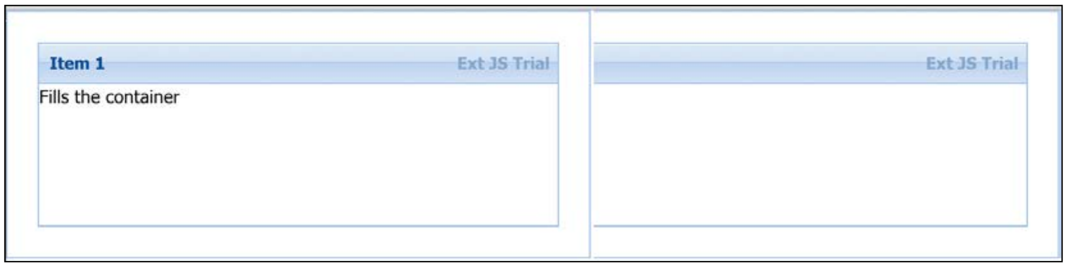


Figure 2.9

The hbox layout

This layout is almost same as the column layout, but it allows you to stretch the height of the column.

The flex option used here makes the child component to flex horizontally according to the given relative flex value:

```
  Ext.create('Ext.panel.Panel', {
    renderTo     : Ext.getBody(),
    width        : 700,
    height       : 400,
    layout       :
    {
      type: 'hbox',
      pack: 'start',
      align: 'stretch',
    },
    items: [
    {
      title: 'Item 1',
      html: 'Item 1',
      flex: 1
    },
    {
      title: 'Item 2',
      html: 'Item 2',
      width: 100
    },
    {
      title: 'Item 3',
      html: 'Item 3',
      flex: 2
    }
    ]
  });
```

The output of the preceding code will look similar to the following figure:

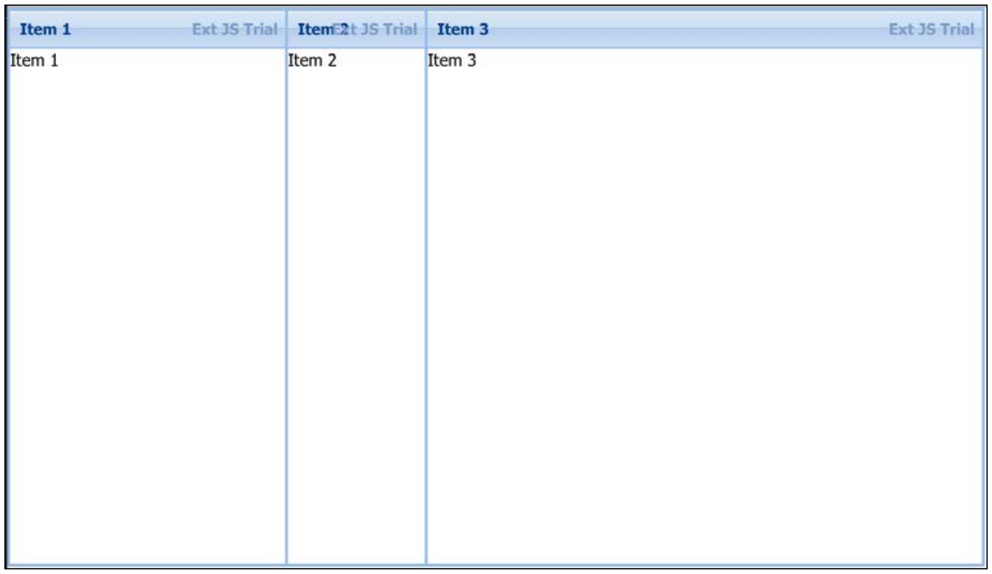


Figure 2.10

The table layout

This layout allows you to render in the table format. You can specify the number of columns and rows with `rowspan` and `colspan` to create complex layouts:

```
Ext.create('Ext.panel.Panel', {
  renderTo     : Ext.getBody(),
  width        : 700,
  height       : 400,
  layout       :
  {
    type: 'table',
    columns: 3,
    tableAttrs: {
      style: {
        width: '100%'
      }
    }
  },
  items: [
  {
    rowspan: 3,
    title: 'Item 1',
    html: 'Item 1'
  },
  {
    title: 'Item 2',
    html: 'Item 2'
  },
  {
    title: 'Item 3',
    rowspan: 2,
    html: 'Item 3'
  },
  {
    title: 'Item 4',
    html: 'Item 4'
  },
  {
    title: 'Item 5',
    html: 'Item 5'
  },
  {
    title: 'Item 6',
    html: 'Item 6'
  },
  {
    title: 'Item 7',
    html: 'Item 7'
  }
  ]
});
```

The output of the preceding table layout code is shown in the following screenshot:

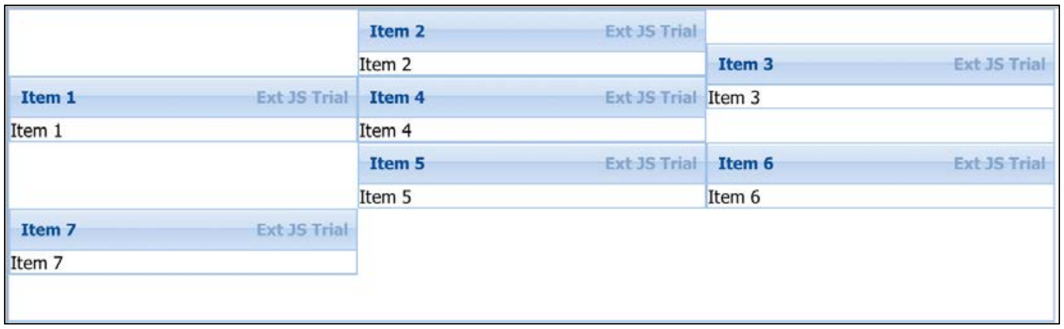


Figure 2.12

The VBox layout

This layout places the child components vertically downwards one after another.

Take a look at the following sample code:

```
Ext.create('Ext.panel.Panel', {
  renderTo     : Ext.getBody(),
  width        : 700,
  height       : 400,
  layout       :
  {
    type: 'vbox',
    pack: 'start',
    align: 'stretch',
  },
  items: [
  {
    title: 'Item 1',
    html: 'Item 1',
    flex: 1
  },
  {
    title: 'Item 2',
    html: 'Item 2',
    height: 100
  },
  {
    title: 'Item 3',
    html: 'Item 3',
    flex: 2
  }
  ]
});
```

The output of this code is shown here:

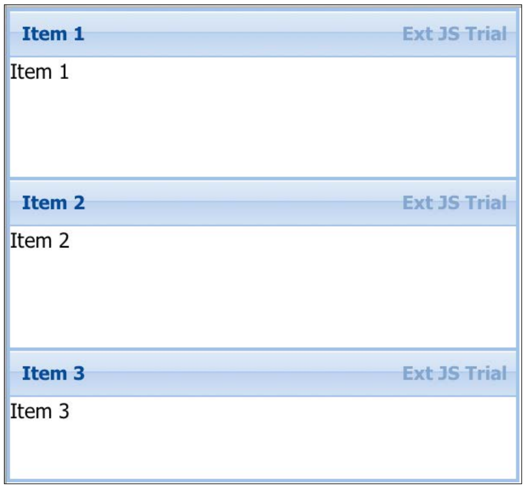


Figure 2.11

Summary

In this chapter, we looked at the base classes in Ext JS and some of the most frequently used methods in these classes. You also learned how to create and extend classes. We learned how to use events and query capabilities.

In the next chapter, we'll take a look at many useful components, such as buttons, menus, toolbars, and so on. We'll also create a small calculator application.

3
Basic Components

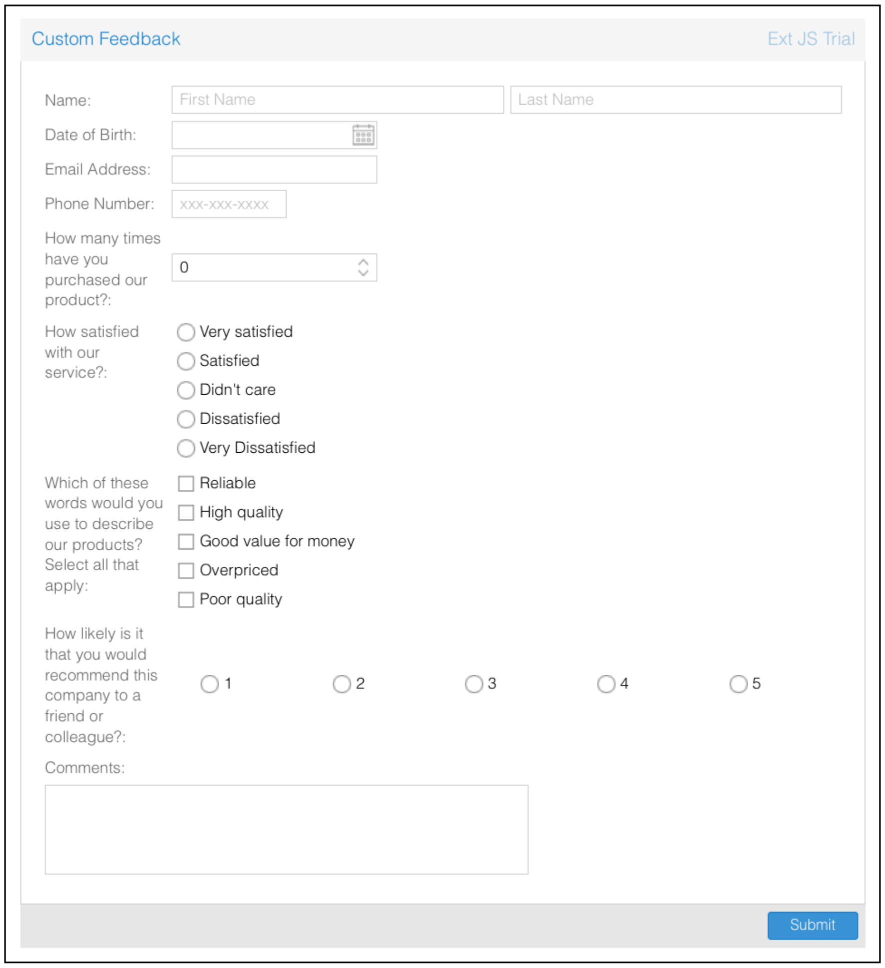
In this chapter, you'll learn some of the basic components available Ext JS. We'll use the concepts learned in the previous chapters and in this chapter to create a sample project. The following topics will be covered in this chapter:

- Getting familiar with basic components – buttons, text field, date picker, and so on
- The form field validation
- Menus and toolbars
- A customer feedback form design
- Calculator – a sample project

The main goal of the chapter is to build a form design and a calculator sample project. *Figure 3.1* and *Figure 3.2* shows the design of the customer feedback form design and the calculator design respectively.

First, if you look at the customer feedback form design in *Figure 3.1*, you'll see that we have used a lot of controls, such as a label, a text field.

This following figure is the design of the **Customer Feedback** form:



Custom Feedback

Ext JS Trial

Name: First Name | Last Name

Date of Birth:

Email Address:

Phone Number: xxx-xxx-xxxx

How many times have you purchased our product?: 0

How satisfied with our service?:
- ◯ Very satisfied
- ◯ Satisfied
- ◯ Didn't care
- ◯ Dissatisfied
- ◯ Very Dissatisfied

Which of these words would you use to describe our products? Select all that apply:
- ☐ Reliable
- ☐ High quality
- ☐ Good value for money
- ☐ Overpriced
- ☐ Poor quality

How likely is it that you would recommend this company to a friend or colleague?: ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5

Comments:

Submit

Figure 3.1 - Customer Feedback form design

Next, you can see the button and label: the most important controls used in the calculator design in *Figure 3.2*. So, you'll first learn about button and handlers. Later, at the end of this chapter, we'll build the calculator project. In the calculator project, you'll learn how the view and controller interact and work together. We'll also see how to bind the view model property to a field in the view.

This following figure is the design of the calculator:

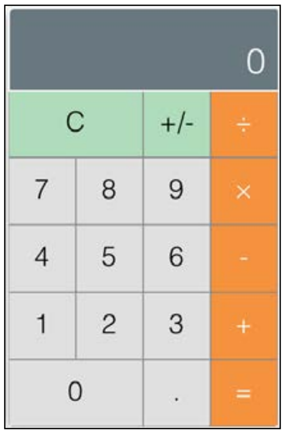


Figure 3.2: Calculator

Getting familiar with basic components

Ext JS comes with lots of useful controls. Let's take a look at some of the basic components:

Ext.Button

This is a commonly used control; the handler is used for the click event, as shown in the following code:

```
Ext.create('Ext.Button', {
  text: 'My Button',
  renderTo: Ext.getBody(),
  handler: function() {
    alert('click');
  }
});
```

The output of the preceding code is as follows:

Figure 3.3

I mentioned about running the sample code in *Chapter 2, Core Concepts* already, but I would like to reiterate this point again. As you read, you can execute this sample code and most of the other sample code that we will see in this book. You can either execute them on your local machine or on Sencha Fiddle. You can visit Sencha Fiddle at `https://fiddle.sencha.com` and put the preceding code in the launch function, run it, and see the result. So, if you go to `https://fiddle.sencha.com`, you'll see the following code:

```
Ext.application({
  name : 'Fiddle',
  launch : function() {
    Ext.Msg.alert('Fiddle', 'Welcome to Sencha Fiddle!');
  }
});
```

Now, paste the button sample code as shown here, run it, and see the result:

```
Ext.application({
  name : 'Fiddle',
  launch : function() {
    Ext.create('Ext.Button', {
      text: 'My Button',
      renderTo: Ext.getBody(),
      handler: function() {
        alert('click');
      }
    });
  }
});
```

[Not all examples can be executed in this way, and not all sample code has visual rendering.]

You can also use the `listeners` config to add one or more event handlers, as shown in the following code:

```
Ext.create('Ext.Button', {
  text: 'My Button',
  renderTo: Ext.getBody(),
  listeners: {
    click: {
      fn: function(){
        //Handle click event
        alert('click');
      }
    },
    mouseout: {
      fn: function(){
        //Handle double click event
        alert('Mouse out');
      }
    }
  }
});
```

The preceding code creates a simple button, but you can create many variations of buttons. You can also create a link button, a button with a menu, a toggle button, and so on.

To create a link button, set the `href` property, as shown in the following code:

```
Ext.create('Ext.Button', {
  renderTo: Ext.getBody(),
  text: 'Link Button',
  href: 'http://www.sencha.com/'
});
```

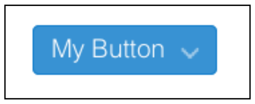
The output of the link button is shown in *Figure 3.4*. When click on it, the link will open up:

Figure 3.4

You can create a menu button by setting the `menu` property, as shown in the following code:

```
Ext.create('Ext.Button', {
  text: 'My Button',
  renderTo: Ext.getBody(),
  menu: [{
    text: 'Item 1'
  }, {
    text: 'Item 2'
  }, {
    text: 'Item 3'
  }]
});
```

The output is shown here:



Collapsed

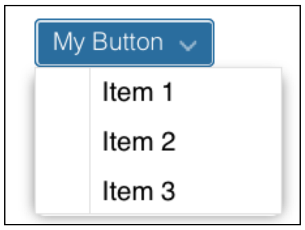


Figure 3.5: Expanded

`Ext.Button` has many other properties, such as `bind`, `cls`, `disabled`, `html`, `tooltip`, `tpl`, and so on, which you can use to customize the button.

Ext.MessageBox

The `Ext.window.MessageBox` class provides the message box implementation. `Ext.MessageBox` is a singleton instance of this class. You can use `MessageBox` to show an alert, get confirmation, prompt for input, and so on.

The following code will show a simple alert. Here, `Ext.Msg` is the alias of `Ext.Messagebox`:

```
Ext.Msg.alert('Info', 'Document saved!');
```

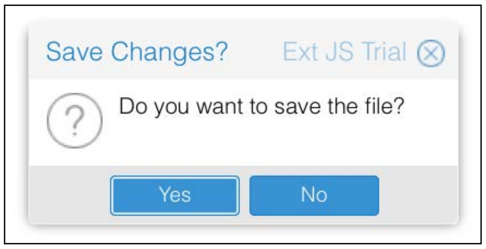
You can show a confirmation message box with a `yes` and `no` button with the following code:

```
Ext.Msg.confirm('Confirm', 'Are you want to cancel the updates?',
function(button){
  if('yes'==button) {

  }
  else {

  }
}
);
```

Also, you can customize the message box as follows:

```
Ext.MessageBox.show({
  title:'Save Changes?',
  msg: 'Do you want to save the file?',
  buttons: Ext.MessageBox.YESNO,
  fn: function(button){
    if('yes'==button)
    {

    }
    else if('no'==button)
    {
    }
  } ,
  icon: Ext.MessageBox.QUESTION
});
```

The output of the preceding code is as follows:



Do you want to save the file?

Figure 3.6

Forms and form fields

Now, let's take a look at some of the form-related components.

Ext.form.Panel

The form panel inherits from the panel and adds functionalities related to forms, such as field management, validation, submission, and so on. The default layout of the form panel is anchor layout, but you can change it if required.

The form panel has a convenient config called `fieldDefaults`, which can be used to specify the default config values for all the fields.

Fields

Ext JS comes with so many built-in ready to use form fields. Some of the commonly used fields are:

```
Ext.form.field.Checkbox
Ext.form.field.ComboBox
Ext.form.field.Date
Ext.form.field.File
Ext.form.field.Hidden
Ext.form.field.HtmlEditor
Ext.form.field.Number
Ext.form.field.Radio
Ext.form.field.Text
Ext.form.field.TextArea
Ext.form.field.Time
```

Let's take a look at some of these form fields here.

Ext.form.field.Text

This is a basic text field, but has so many useful properties and configs. One of these useful property is `vtype` that is used for validation. For example, you can set the `vtype` property as an e-mail to validate the input for the valid e-mail, as shown in the following code:

```
Ext.create('Ext.form.field.Text', {
  renderTo: Ext.getBody(),
  name: 'email',
  fieldLabel: 'Email',
  allowBlank: false,
  vtype: 'email'
});
```

Here `allowBlank` is a validation property. By setting allowBlank to `false`, it will show validation error if the field is blank.

Ext.form.field.Number

The number field extends from the spinner field, which in turn extends from the text field. The number field provides several options to handle a numeric value. The following code will render a number field shown in *Figure 3.7*:

```
Ext.create('Ext.form.field.Number', {
  renderTo: Ext.getBody(),
  name: 'Count',
  fieldLabel: 'Count',
  value: 0,
  maxValue: 10,
  minValue: 0
});
```

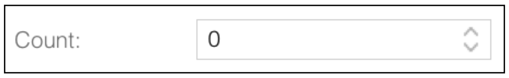


Figure 3.7

You can remove spinner buttons, arrow keys, and mouse wheel listeners with the config options: `hideTrigger`, `keyNavEnabled`, and `mouseWheelEnabled` respectively.

Ext.form.field.ComboBox

The following code creates a dropdown with a list of months. The `combobox` has a config called `store`. The datastore provides the data for the dropdown. The datastore is part of the data packages of ExtJS, which we'll cover in detail in the upcoming chapters.

Another important config in the `combobox` is `queryMode`. This can be either local or remote. If you set this as remote, then the datastore will be loaded at runtime by sending a request to the remote server:

```
  var months = Ext.create('Ext.data.Store', {
    fields: ['abbr', 'name'],
    data: [
    {"abbr":"JAN", "name":"January"},
    {"abbr":"FEB", "name":"February"},
    {"abbr":"MAR", "name":"March"},
    {"abbr":"APR", "name":"April"},
    {"abbr":"MAY", "name":"May"},
    {"abbr":"JUN", "name":"June"},
    {"abbr":"JUL", "name":"July"},
    {"abbr":"AUG", "name":"August"},
    {"abbr":"SEP", "name":"September"},
    {"abbr":"OCT", "name":"October"},
    {"abbr":"NOV", "name":"November"},
    {"abbr":"DEC", "name":"December"}
    ]
  });

  Ext.create('Ext.form.ComboBox', {
    fieldLabel: 'Choose Month',
    store: months,
    queryMode: 'local',
    displayField: 'name',
    valueField: 'abbr',
    renderTo: Ext.getBody()
  });
```

The output of the preceding code is as follows:

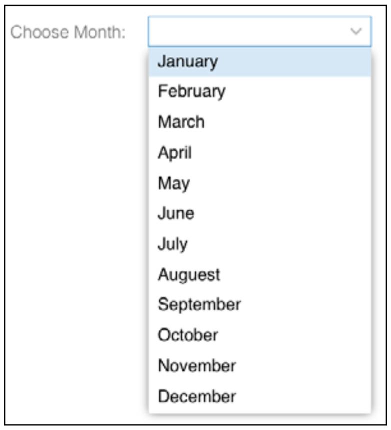


Figure 3.8

Ext.form.field.HtmlEditor

Ext JS also has a very good HTML editor that provides the common word processor features directly to your web pages, as shown in the following code:

```
  Ext.create('Ext.form.HtmlEditor', {
    width: 800,
    height: 200,
    renderTo: Ext.getBody()
  });
```

The output of the preceding code is as follows:

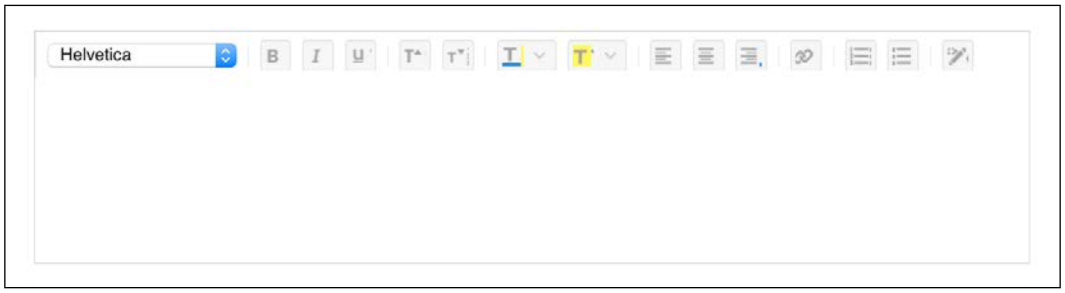


Figure 3.9

The form field validation

Most fields have its own validation rules, for example, if you enter a nonnumeric in the number field, it will show an invalid number validation. The text field comes with `allowBlank`, `minLength`, and `maxLength`. Further, Regex can be used for custom validations.

Events in the form panel

Some of the supported events in the form panel are:

- `beforeaction`: This event will be fired before executing any action
- `actionfailed`: This event will be fired when an action fails
- `actioncomplete`: This event will be fired after an action is completed
- `validitychange`: This event will be fired when the validity of the entire form changes
- `dirtychange`: This event will be fired when the dirty state of the form changes

Form field containers

The following are some of the useful containers for the form panel.

Ext.form.CheckboxGroup

`CheckboxGroup` extends `FieldContainer` and is used to group the checkbox field. In the following example, note the same name for all the items in the checkbox group; this helps in getting the values passed as a single parameter to the server.

```
Ext.create('Ext.form.CheckboxGroup', {
  renderTo: Ext.getBody(),
  fieldLabel: 'Skills ',
  vertical: true,
  columns: 1,
  items: [
  { boxLabel: 'C++', name: 'rb', inputValue: '1' },
  { boxLabel: '.Net Framework', name: 'rb', inputValue: '2',
checked: true },
  { boxLabel: 'C#', name: 'rb', inputValue: '3' },
  { boxLabel: 'SQL Server', name: 'rb', inputValue: '4' },
  ]
});
```

The output of the preceding code is as follows:

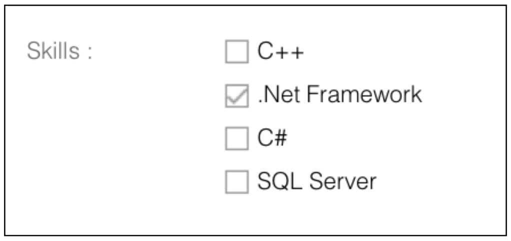


Figure 3.10

Ext.form.FieldContainer

`FieldContainer` is useful when you want to group a set of related field and attach it to a single label.

The following code creates an output, which is shown in *Figure 3.11*:

```
Ext.create('Ext.form.FieldContainer', {
  renderTo: Ext.getBody(),
fieldLabel: 'Name',
  layout: 'hbox',
  combineErrors: true,
  defaultType: 'textfield',
  defaults: {
    hideLabel: 'true'
  },
  items: [{
    name: 'firstName',
    fieldLabel: 'First Name',
    flex: 2,
    emptyText: 'First',
    allowBlank: false
  }, {
    name: 'lastName',
    fieldLabel: 'Last Name',
    flex: 3,
    margin: '0 0 0 6',
    emptyText: 'Last',
    allowBlank: false
  }]
});
```

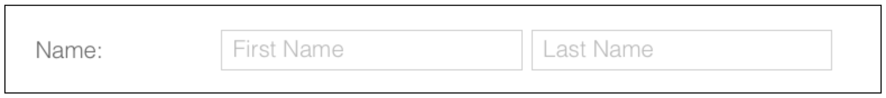


Figure 3.11

Ext.form.RadioGroup

`RadioGroup` extends `CheckboxGroup` and is used to group radio buttons. Note that the name property is the same for all the item. This keeps them grouped. Otherwise, they would be independently selectable, as shown in the following code:

```
Ext.create('Ext.form.RadioGroup', {
  renderTo: Ext.getBody(),
  fieldLabel: 'Sex ',
  vertical: true,
  columns: 1,
  items: [
  { boxLabel: 'Male', name: 'rb', inputValue: '1' },
  { boxLabel: 'Female', name: 'rb', inputValue: '2' }

]});
```

The output of the preceding code is as follows:

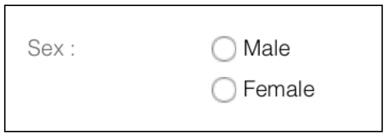


Figure 3.12

Submitting a form

To submit a form, you can use the submit method of the form. Use the `getForm` method to get the form and `isValid` to validate the form before submitting it, as shown in the following code:

```
var form = this.up('form').getForm();
if (form.isValid()) {
  form.submit({
    url: 'someurl',
    success: function () {

    },
    failure: function () {

    }
  });
} else {
  Ext.Msg.alert('Error', 'Fix the errors in the form')
}
```

Menus and toolbar

Ext JS provides complete support to build any kind of toolbar and menus you can think of. Use `Ext.toolbar.Toolbar` to build a toolbar. By default, any child items in `Ext.toolbar.Toolbar` is a button, but you can add any other controls, such as a text field, a number field, an icon, a dropdown, and so on in the toolbar.

To arrange items in the toolbar, you can use `Ext.toolbar.Spacer`, `Ext.toolbar.Separator`, and `Ext.toolbar.Fill` to have space, a separator bar, and a right-justified button container respectively. The shortcuts for these are, `' '` (space), `'|'` (pipe), and `'->'` (arrow), respectively.

`Ext.menu.Menu` is used to build a menu with `Ext.menu.Item` as menu items.

A sample code and its output are shown in the following screenshot:

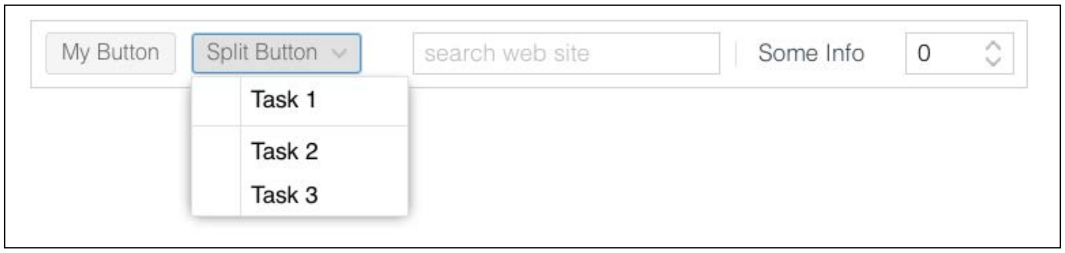


Figure 3.13

```
Ext.create('Ext.toolbar.Toolbar', {
  renderTo: Ext.getBody(),
  width: 800,
  items: [
  {
    text: 'My Button'
  },
  {
    text: 'My Button',
    menu: [{
      text: 'Item 1'
    }, {
      text: 'Item 2'
    }, {
      text: 'Item 3'
    }]
  },
  {
    text: 'Menu with divider',
    tooltip: {
      text: 'Tooltip info',
      title: 'Tip Title'
    },
    menu: {
      items: [{
        text: 'Task 1',
        // handler: onItemClick
      }, '-', {
        text: 'Task 2',
        // handler: onItemClick
      }, {
        text: 'Task 3',
        // handler: onItemClick
      }]
    }
  },
  '->',
  {
    xtype: 'textfield',
    name: 'field1',
    emptyText: 'search web site'
  },
  '-',
  'Some Info',
  {
    xtype: 'tbspacer'
  },
  {
    name: 'Count',
    xtype: 'numberfield',
    value: 0,
    maxValue: 10,
    minValue: 0,
    width: 60
  }
  ]
});
```

The customer feedback form design

Now, let's use some of the concepts learned in the previous chapters and this chapter to design a customer feedback form design.

The following form in *Figure 3.13* is the design of the form that we will design:

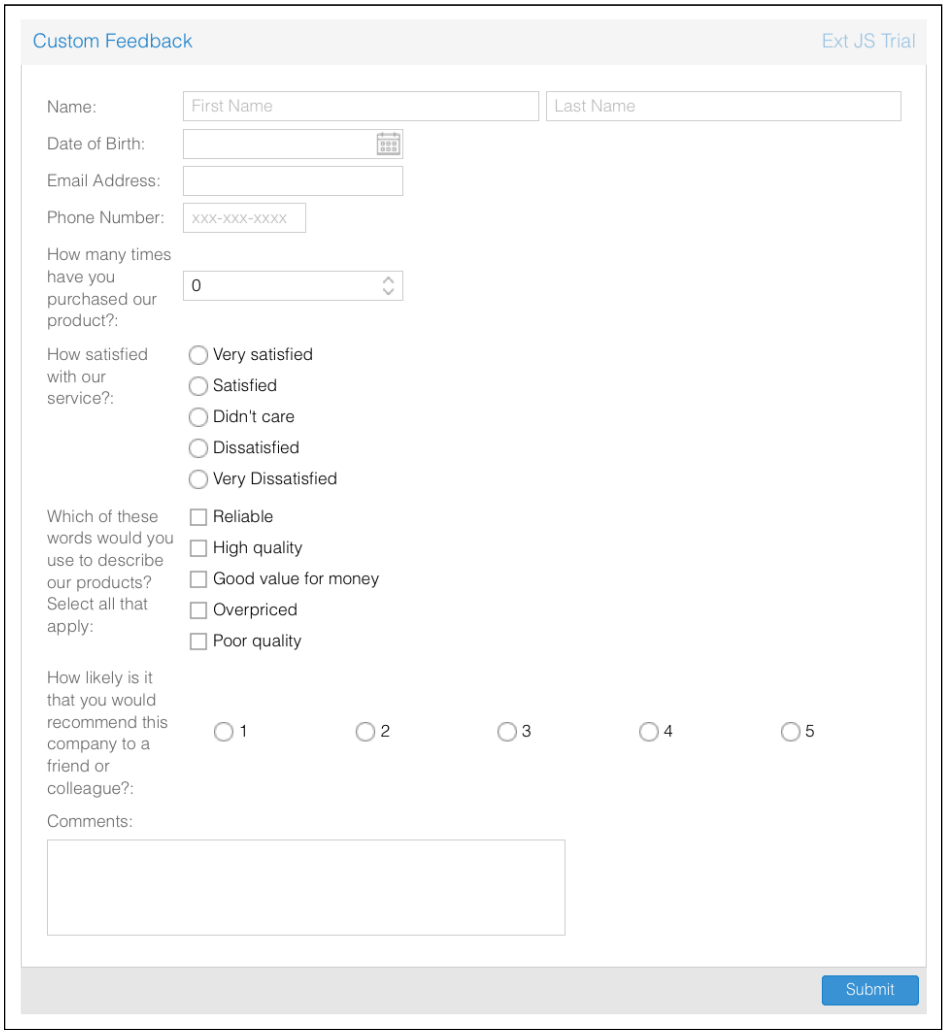


Figure 3.14

The code for the preceding design is shown in the following code. I have kept only the important code and truncated the rest. The complete source code for this project is available at `https://github.com/ananddayalan/extjs-by-example-customer-feedback-form`.

Here, we will place all the component in `Viewport`. This is a specialized container that represents the browser's application view area.

In the `Viewport`, we will set the scrollable option to make this child component scrollable. Instead of true or false, this option can also take *x* or *y* as values to enable only horizontal or vertical scroll:

```
Ext.create('Ext.container.Viewport', {
  scrollable: true,
  items: [{
    xtype: 'container',
    layout: {
  type: 'hbox',
  align: 'center',
  pack: 'center'
 },
    items: [ {
      xtype: 'form',
      bodyPadding: 20,
      maxWidth: 700,
      flex: 1,
      title: 'Custom Feedback',
      items: [ {
        xtype: 'fieldcontainer',
        layout: 'hbox',
        fieldLabel: 'Name',
        defaultType: 'textfield',
        defaults: {
          allowBlank: false,
          flex: 1
        },
        items: [{
          name: 'firstName',
          emptyText: 'First Name
        },  {
          name: 'lastName',
          margin: '0 0 0 5',
          emptyText: 'Last Name'
        }
      ]}, {
        xtype: 'datefield',
        fieldLabel: 'Date of Birth',
        name: 'dob',
        maxValue: new Date() /* Prevent entering the future date.
*/
      }, {
        fieldLabel: 'Email Address',
        name: 'email',  vtype: 'email',
        allowBlank: false
      }, {
        fieldLabel: 'Phone Number',
        labelWidth: 100,
        name: 'phone',
        width: 200, emptyText: 'xxx-xxx-xxxx',
        maskRe: /[\d\-]/,
        regex: /^\d{3}-\d{3}-\d{4}$/,
      regexText: 'The format must be xxx-xxx-xxxx' },
      //…code truncated
      {
        xtype: 'radiogroup',
        fieldLabel: 'How satisfied with our service?',
        vertical: true, columns: 1,
        items: [  {
          boxLabel: 'Very satisfied',
          name: 'rb',
          inputValue: '1'
        },  {
          boxLabel: 'Satisfied',
          name: 'rb', inputValue: '2'
        },
        //…code truncated
        ]
      },{
        xtype: 'checkboxgroup',
        fieldLabel: 'Which of these words would you use to
describe our products? Select all that apply',
        vertical: true,
        columns: 1,
        items: [{
          boxLabel: 'Reliable',
          name: 'ch',
          inputValue: '1'
        },
        //…code truncated
        ]

      },
      {
        xtype: 'radiogroup',
        fieldLabel: 'How likely is it that you would recommend
this company to a friend or colleague?',
        vertical: false,
        defaults: { padding: 20 },
        items: [ {
          boxLabel: '1',
          name: 'recommend',
          inputValue: '1'
        },
        //…code truncated
        ],
        buttons: [{
          text: 'Submit',
          handler: function () {
            var form = this.up('form').getForm();
            if (form.isValid()) {
              form.submit({
                url: 'cutomer/feedback',
                success: function () {},
                failure: function () {}
              });
            } else {
              Ext.Msg.alert('Error', 'Fix the errors in the form')
            }
          }
          //…code truncated
```

In the preceding code, by setting `defaultType` at the container level, we're avoiding the repetition of specifying `xtype` for the child components of the container. So, by default, all child components that doesn't have the `xtype` set will default to `textfield`.

On the `form` panel, the `flex` is used to make the `form` panel to fill the parent container's width, and at the same time, we will limit the max width of the form by setting `maxWidth` to `700`.

The field container is used with the `hbox` layout to put both the first name and the last name under a single label.

Calculator – a sample project

Let's build a complete sample project with the concepts learned in the previous chapter and this chapter. Here is the design of the calculator that we will build:

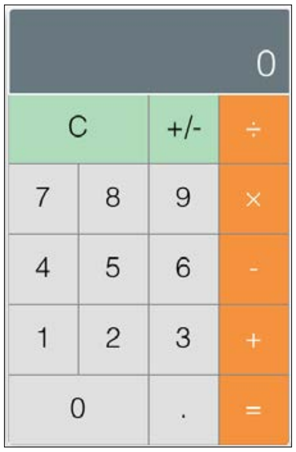
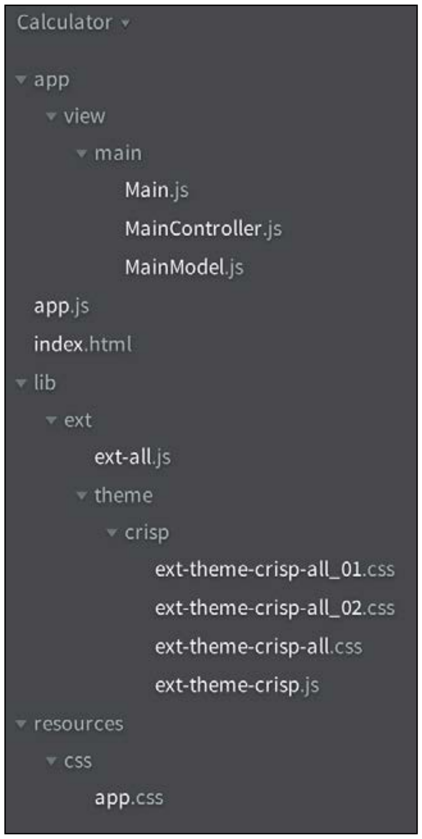


Figure 3.15

The folder structure

Here is the folder structure of the files that we have in this project. I have copied only some of the required files from Ext JS to the project folder:



```
Calculator
▾ app
    ▾ view
        ▾ main
            Main.js
            MainController.js
            MainModel.js
  app.js
  index.html
▾ lib
    ▾ ext
        ext-all.js
        ▾ theme
            ▾ crisp
                ext-theme-crisp-all_01.css
                ext-theme-crisp-all_02.css
                ext-theme-crisp-all.css
                ext-theme-crisp.js
▾ resources
    ▾ css
        app.css
```

Figure 3.16

The following are some of the important files in the project. I have excluded the HTML and CSS files from here. The complete project files are available at `https://github.com/ananddayalan/extjs-by-example-calculator`.

App – app.js

In `app.js`, we will simply create the `Main` view and show it as a floating, movable window in the browser:

```
Ext.application({
  name: 'Calc',
  launch: function () {
    Ext.create('Calc.view.main.Main').show();
  }
});
```

MVC and MVVM – Revisit

In *Chapter 1*, *Getting Started with Ext JS* you learned about **MVC** (**Model View Controller**) and **MVVM** (**Model View ViewModel**). The code in this project is a very good example to show the distinction between the view, controller, and view model.

Model

This represents the data layer. The model can contain data validation and logics to persist the data.

View

This represents the user interface. So, components such as button, form, and a message box are views. The `main.js` in this calculator project is a good example for the view.

Controller

This handles any view-related logic, event handling of the view, and any app logic.

ViewController and Controller

In Ext JS 5 and 6, there are two types of controllers: `ViewController` and `Controller`. The `ViewController` was introduced in Ext JS 5. `ViewController` is created for a specific view, but you can have the controller for application-wide cross view logic.

The view controller brings some new concepts—such as reference and listener—to simplify the connection between the view and controller. Also, `ViewController` will be destroyed when the view is destroyed. We will not use the reference and listeners in this example, but we'll use them in the next sample project.

You can use the listeners in this project instead of using the handler to handling the events.

View model

This encapsulates the presentation logic required for the view, binds the data to the view, and handles all the updates whenever the data is changed.

Unlike model, `view model` is created mostly for a specific view. A model is a pure data class and can be used across the application, but `view model` is created to use with a view and server as a data binder between the view and model. Take a look at the `main.js` in this calculator project and see the view model binding.

View — Main.js

I will create one single view for this calculator application called `Main`. This view contains all the button, the display field, and so on. Events are associated with the methods of this controller. The controller of this view has been specified using the controller config.

This view makes use of the table layout with four columns. The CSS class has been specified with the `cls` property.

Refer to the additional comments/explanation in the code as comments:

```
Ext.define('Calc.view.main.Main', {
  extend: 'Ext.window.Window',

  /* Marks these are required classes to be to loaded before
loading this view */
  requires: [ 'Calc.view.main.MainController',
'Calc.view.main.MainModel'],
  xtype: 'app-main',
  controller: 'main',

  /* View model of the view */
  viewModel: { type: 'main' },

  resizable: false,
  layout: {
    type: 'table',
    columns: 4
  },

  /* Defaults properties to be used for the child items. Any child
can override it */

  defaultType: 'button',
  defaults: {
    width: 50,
    height: 50,
    cls: 'btn',
    handler: 'onClickNumber'
  },

  /* I'm using the header config of the Ext.window.Window to
display the result in the calculator. Using this header you can
move the floating calculator around within the browser */

  header: {
    items: [
    {
      xtype: 'displayfield',
      colspan: 4,
      width: 200,
      cls: 'display',
      bind: {  value: '{display}' },
      height: 60,
      padding: 0
    }]
  },
  items: [{
    text: 'C',
    colspan: 2,
    width: 100,
    cls: 'btn-green',
    handler: 'onClickClear'
  },  {
    text: '+/-',
    cls: 'btn-green',
    handler: 'onClickChangeSign'
  }, {
    text: '&divide;',
    cls: 'btn-orange',
    handler: 'onClickOp'
  },
  {  text: '7' },
  { text: '8' },
  { text: '9' },
  {  text: '&times;',
    cls: 'btn-orange',
    handler: 'onClickOp'
  },
  { text: '4'},
  { text: '5'},
  { text: '6'},
    {
      text: '-',
      cls: 'btn-orange',
      handler: 'onClickOp'
    },
    { text: '1'},
    { text: '2'},
    { text: '3'},
    {
      text: '+',
      cls: 'btn-orange',
      handler: 'onClickOp'
    }, {
      text: '0',
      width: 100,
      colspan: 2
    }, {
      text: '.',
      handler: 'onClickDot'
    }, {
      text: '=',
      cls: 'btn-orange',
      handler: 'onClickOp'
    }]
  });
```

Controller — MainController.js

Although this controller code is a bit longer, it's a very simple code. This controller has bunch methods to handle the click events of the buttons, such as operators and operands. The controller uses a model called `Main`:

```
Ext.define('Calc.view.main.MainController', {

  extend: 'Ext.app.ViewController',

  alias: 'controller.main',
  views: ['Calc.view.main.Main'],
  models: ['Main'],

  //Here the 'state' is a custom property that use to track the
state of the calculator.
  state: {
    operatorClicked: false,
    selectedOperator: null,
    dotClicked: false,
    op1: 0,
    numberClicked: false,
    sign: true,
    decimal: false
  },
  onClickClear: function () {

    var vm = this.getViewModel();
    vm.set('display','0');
    this.state.selectedOperator=null;
    this.state.op1=0;
    this.state.isPositive = true;
    this.state.decimal = false;
    this.state.sign = true;
  },
  onClickChangeSign: function (btn) {

    var vm = this.getViewModel();
    var cur = vm.get('display');
    if(cur!='0') {
      if(this.state.sign===true ) {
        vm.set('display', '-' + cur);
      }
      else {
        vm.set('display', cur.toString().substring(1));
      }
    }
    this.state.sign=!this.state.sign;
  },

  onClickOp: function (btn) {

    if(this.state.selectedOperator!=null &&
this.state.numberClicked===true)
    {
      var vm = this.getViewModel();
      var op2 = parseFloat(vm.get('display'));
      var op1 = parseFloat(this.state.op1);
      var result = 0;

      switch(this.state.selectedOperator){
        case '+':
        result = op1 + op2;
        break;

        case '-':
        result = op1 - op2;
        break;

        case '&times;':
        result = op1 * op2;
        break;

        case '&divide;':
        result = op1 / op2;
        break;
      }
      vm.set('display', Math.round(result * 100) / 100);
      this.state.selectedOperator=null;
    }
    if(btn.text!='=') {
      this.state.operatorClicked = true;
    }
    this.state.selectedOperator = btn.text;
    this.state.numberClicked = false;
  },

  onClickDot: function (btn) {
    if(this.state.decimal===false) {
      var vm = this.getViewModel();
      vm.set('display', vm.get('display') + '.');
    }
  },
  onClickNumber: function (btn) {

    this.state.numberClicked = true;
    if(this.state.selectedOperator ==='='){
      this.onClickClear();
    }
    var vm = this.getViewModel();
    if(this.state.operatorClicked===true) {
      this.state.op1= vm.get('display');
      vm.set('display', btn.text);
      this.state.operatorClicked=false;
    }
    else{
      var cur = vm.get('display');
      if(cur == '0') {
        cur = '';
      }
      vm.set('display', cur + btn.text);
    }
  }
});
```

ViewModel — MainViewModel.js

This `ViewModel` has just one property called `display`. This is used to bind the display value of the calculator. Here, we will not create a model separately with a set of fields. Also, we have hardcoded the data directly.

```
Ext.define('Calc.view.main.MainModel', {
  extend: 'Ext.app.ViewModel',
  alias: 'viewmodel.main',
  data: {
    display: 0.0
  }
});
```

You'll learn more about view model, model, fields, field types, validation, and so on in the upcoming chapters.

Summary

In this chapter, you learned about different kinds of basic components, such as the test field, the number field, button, menu, and so on. You already learned how to design a form using form fields, and we created a nice simple project called calculator.

In the next chapter, you'll learn about data packages, such as data stores, model, proxies, and so on, which will be useful to handle data.

4
Data Packages

This chapter explores the tools available in Ext JS to handle the data and the communication between the server and client. This chapter will end with a sample project on RESTful services. The following topics will be covered in this chapter:

- Models
- Schema
- Stores
- Proxies
- Filtering and sorting
- To do — a RESTful sample project

Model

A model defines the fields, field types, validation, associations, and proxies. It is defined by extending from the `Ext.data.Model` class.

The types, validation, association, and proxies are optional. When a field is specified without type, then the default type `'auto'` will be used. Normally, proxies are specified in the data store, but you can have proxies in the model as well.

Field

`Ext.data.field.Field` is used to add properties of the model. A field can be a predefined type or a custom type. The following are the predefined types available:

- auto
- boolean
- date

- int
- number
- string

The data conversion

By default, when you specify a type for a field, the data is converted to that type before it is saved to the field. This conversion is handled by the inbuilt `convert` method. The auto field type doesn't have a `convert` method, so conversion doesn't happen for the auto field type.

All other field types have the `convert` method. If you want to avoid the conversion for other field types in order to improve the performance, you can do so by specifying the convert config of the respective fields as `null` as shown in the following `Employee` model.

Validators

The model supports the validation of the model data. The following are the supported validators:

- **presence**: This makes sure that a value is present for a particular field
- **format**: This format can be validated using a regular expression
- **length**: This length validator supports the maximum and minimum length validation
- **exclusion and inclusion**: You can pass a set of values to these validators to make sure that the value doesn't or does present in the given set of values

The following code shows an example model with some validators on the fields:

```
Ext.define('Employee', {
  extend: 'Ext.data.Model',
  fields: [
  { name: 'id',  type: 'int',  convert: null },
  { name: 'firstName', type: 'string' },
  { name: 'lastName', type: 'string' },
  { name: 'fulltime', type: 'boolean', defaultValue: true,
convert: null },
  { name: 'gender', type: 'string' },
  { name: 'phoneNumber', type: 'string'},
  ],
  validators: {
    firstName: [
    { type: 'presence'},
    { type: 'length', min: 2 }
    ],
    lastName:[
    { type: 'presence'},
    { type: 'length', min: 2 }
    ],
    phoneNumber: {
      type: 'format',
      matcher: '/^[(+{1})|(00{1})]+([0-9]){7,10}$/'
    },
    gender: {
      type: 'inclusion',
      list: ['Male', 'Female']
    },
  }
});
```

To create an instance of a model, use `Ext.create`, as shown in the following code:

```
var newEmployee = Ext.create('Employee', {
  id : 1,
  firstName : 'Shiva',
  lastName : 'Kumar',
  fulltime : true,
  gender: 'Male',
  phoneNumber: '123-456-7890'
});
```

The model has `get` and `set` methods to read and set values:

```
var lastName = newEmployee.get('lastName');
newEmployee.set('gender','Female');
```

Relationships

To define the relation between the entities, use one of the following relationship types:

One-to-one

The following code represents one-to-one relationship:

```
Ext.define('Address', {
  extend: 'Ext.data.Model',

  fields: [
  'address',
  'city',
  'state',
  'zipcode'
  ]
});

Ext.define('Employee', {
  extend: 'Ext.data.Model',

  fields: [{
    name: 'addressId',
    reference: 'Address'
  }]
});
```

One-to-many

The following code represents one-to-many relationship:

```
  Ext.define('Department', {
    extend: 'Ext.data.Model',

    fields: [
    { name: 'employeeId', reference: 'Employee' }
    ]
  });

  Ext.define('Division', {
    extend: 'Ext.data.Model',

    fields: [
    { name: 'departmentId', reference: 'Department' }
    ]
  });
```

Many-to-many

The following code represents many-to-many relationship:

```
  Ext.define('Employee', {
    extend: 'Ext.data.Model',

    fields: [
    { name: 'empId',  type: 'int',  convert: null },
    { name: 'firstName', type: 'string' },
    { name: 'lastName', type: 'string' },
    ],

    manyToMany: 'Project'
  });

  Ext.define('Project', {
    extend: 'Ext.data.Model',

    fields: [
    'name'
    ],

    manyToMany: 'Employee'
  });
```

Custom field types

You can also create custom field types by simply extending from `Ext.data.field.Field`, as shown in the following code:

```
  Ext.define('App.field.Email', {
    extend: 'Ext.data.field.Field',
    alias: 'data.field.email',

    validators: {
      type: 'format',
      matcher: /^\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$/,
      message: 'Wrong Email Format'
    }
  });
```

Store

A store represents a collection of instances of the model and proxies used to get the data. The store also defines collection operations, such as sorting, filtering, and so on. It is defined by extending the `Ext.data.Store` class.

Usually, when you define a store, you need to specify a proxy in the store. This is a configuration that tells the store how to read and write the data. We'll see more details on this and look at the different kinds of proxies in the upcoming sections of this chapter.

The following is an example of a store that uses the RESTful API request to load the data in the JSON format:

```
var myStore = Ext.create('Ext.data.Store', {
  model: 'Employee',
  storeId: 'mystore',
  proxy: {
    type: 'rest',
    url: '/employee',
    reader: {
      type: 'json',
      rootProperty: 'data'
    }
  },
  autoLoad: true,
  autoSync: true
});
```

Here, `storeId` is a unique identifier of the store. This store has a method called `load`, which is used to load the data via the configured proxy. If you set `autoLoad` to `true`, then the `load` method will be called automatically when the store is created. If you set `autoLoad` to `false`, then you can call the `load` method to load the data.

Similarly, you can set `autoSync` to `true` to enable the sync to happen as soon as you edit, add, or remove the records in the store.

In the preceding example, it will call the REST service API as soon as the edit, add, or remove operation is performed in the store. If you set this property to `false`, then you can call the `sync` method of the store to perform the sync operation.

Calling the `sync` method will trigger a batch operation. So, if you have added some records and removed some records, then calling the `sync` method will trigger multiple calls to the server in order to perform these operations. Here is an example of a sync call:

```
store.sync({
  callback: function (batch, options) {
    //Do something
  },
  success: function (batch, options) {
    //Do something
  },
  failure: function (batch, options) {
    //Do something
  },
  scope: this
});
```

Here, the success method will be called when all the sync operations are completed without any exception or failure. The failure method will be called if one or more operations in the sync fails. The `callback` method will be called upon completion of the sync operations, irrespective of its success or failure.

If the `failure` method is called, you can check the batch's exception array to see exactly what operations failed and why. Here, the options are original parameters that are passed in the sync.

The `sync` method also has an additional property called **params**, which you can use to pass any additional parameters during the sync.

The inline data store

If you don't want to bind the store to the server or an external storage such as browsers LocalStorage, but want to use some of the features of the store for some static data, then it's possible to hardcode the inline data in the store directly, as shown in the following code:

```
Ext.create('Ext.data.Store', {
  model: 'Employee',
  data: [
  {
    firstName: 'Shiva',
    lastName: 'Kumar',
    gender: 'Male',
    fulltime: true,
    phoneNumber: '123-456-7890'
  },
  {
    firstName: 'Vishwa',
    lastName: 'Anand',
    gender: 'Male',
    fulltime: true,
    phoneNumber: '123-456-7890'
  }
  ]
});
```

Filtering and sorting

The store supports both filtering and sorting either locally or remotely. The following is an example of local sorting:

```
var myStore = Ext.create('Ext.data.Store', {
  model: 'Employee',
  sorters: [{
    property: 'firstName',
    direction: 'ASC'
  }, {
    property: 'fulltime',
    direction: 'DESC'
  }],

  filters: [{
    property: 'firstName',
    value: /an/
  }]
});
```

To have the remote sort and the remote filter, set `remoteSort` and `remoteFilter` to `true`. If you set these values to `true`, then you will have to perform filtering and sorting on the server side and return the filtered and sorted data to the client.

Accessing the store

You might need to keep the store in a separate file, and you'll need to access the store in other parts of the application. There are multiple ways you can access the store in other parts of your application.

Accessing the store using StoreManager

Using the store manager's `lookup` method, you can access the store from anywhere in the application. For this, we need to use `storeId`. Note that when the store is instantiated by a controller, `storeId` will be overridden by the name of the store. Here is an example:

```
  Ext.create('Ext.data.Store', {
    model: 'Employee',
    storeId: 'mystore',
    proxy: {
      type: 'rest',
      url: '/employee',
      reader: {
        type: 'json',
        rootProperty: 'data'
      }
    }
  });
```

Let's say we have created a store as shown before. Now, you can access this store anywhere in the application by passing `storeId` to the `StoreManager.lookup` method, as shown in the following code:

```
Ext.data.StoreManager.lookup('myStore');
```

Instead, you can also use the shortcut method. `Ext.getStore`. `Ext.getStore` is a shortcut method for `Ext.data.StoreManager.lookup`.

Accessing the store using Ext.app.ViewModel

You can access the store via the `getStore` method of `Ext.app.ViewModel`. When you are in `ViewController`, it's better to use this method, as shown in the following code:

```
var myStore = this.getViewModel().getStore('myStore')
```

The `getStore` method is also defined in the view; you can access the store using it.

Store events

There are multiple store events you can listen. Some of the store events are:

- **add**: This is called when a record is added to the store
- **beforeload**: This is called before loading the data
- **beforesync**: This is called before the sync operation
- **datachanged**: This is called when records are added or removed from the store
- **load**: This is called when the store reads from a remote data store
- **remove**: This is called when a record is removed from the store
- **update**: This is called when a record gets updated

An example of listening to the store event is given here:

```
Ext.create('Ext.data.Store', {
  model: 'Employee ',
  storeId: 'mystore',
  proxy: {
    type: 'rest',
    url: '/employee',
    reader: {
      type: 'json',
      rootProperty: 'data'
    }
  },
  listeners: {
    load: function (store, records, options) {
      //Do something
    }
  }
});
```

If you want to listen to the store events in your controller, you can do so, as shown in the following code:

```
init: function() {
  this.getViewModel().getStore('myStore').on('load',
this.onStoreLoad, this);
}
```

The store in ViewModel

You can define your store and `ViewModel` separately or together. It's often desirable to define the store in the ViewModel itself. An example is shown here:

```
Ext.define('MyApp.view.employee.EmployeeModel', {
  extend: 'Ext.app.ViewModel',
  alias: 'viewmodel.employee',
  stores: {
    employee: {
      fields: [
      { name: 'id', type: 'string' },
      { name: 'firstname', type: 'string' },
      { name: 'lastname', type: 'string' }
      ],
      autoLoad: false,
      sorters: [{
        property: 'firstname',
        direction: 'ASC'
      }],
      proxy: {
        type: 'rest',
        url: 'employee',
        reader: {
          type: 'json',
        },
        writer: {
          type: 'json'
        }
```

```
          }
        }
      }
    });
```

Proxies

Stores and models use the proxy to load and save data. Using the configs in the proxy, you can specify how to read and write the data. You can also specify the URL that has to be called to read the data; you can tell the reader the format of the data and whether it's JSON or XML, and so on.

There are two types of proxies: the client-side proxy and the server-side proxy.

The client-side proxy

Client-side proxies are used to handle data loading and saving from the client itself. There are three client side proxies: memory, LocalStorage, and SessionStorage.

The memory proxy

A memory proxy is used for an in-memory local variable data. The following code shows an example of the memory proxy. Here, the data doesn't have be to be a hardcoded value. It can be any variable with data in a proper format:

```
var data = {
  data: [
  {
    firstName: 'Shiva',
    lastName: 'Kumar',
    gender: 'Male',
    fulltime: true,
    phoneNumber: '123-456-7890'
  },
  {
    firstName: 'Vishwa',
    lastName: 'Anand',
    gender: 'Male',
    fulltime: true,
    phoneNumber: '123-456-7890'
  },
};

var myStore = Ext.create('Ext.data.Store', {
  model: 'Employee',
  data : data,
  proxy: {
    type: 'memory',
    reader: {
      type: 'json',
      rootProperty: 'Employee'
    }
  }
});
```

The LocalStorage proxy

This is used to access a browser's LocalStorage. It is a key-value-pair storage added in HTML 5, so many older browsers don't support it:

```
  var myStore = Ext.create('Ext.data.Store', {
    model: 'Benefits',
    autoLoad: true,
    proxy: {
      type: 'localstorage',
      id: 'benefits'
    }
  });
```

The SessionStorage proxy

This is used to access a browser's SessionStorage. Again, this is an HTML 5 feature which is supported in recent browsers only. The data stored in SessionStorage will get destroyed when the session expires:

```
  var myStore = Ext.create('Ext.data.Store', {
    model: 'Benefits',
    autoLoad: true,
    proxy: {
      type: 'localstorage',
      id  : 'benefits'
    }
  });
```

The server-side proxy

Server-side proxies communicate to the server to read and save data. There are four kinds of proxies:

- **Ajax**: This is used to send asynchronous requests.
- **Direct**: This uses `Ext.Direct` to communicate with the server.
- **JSONP** (JSON with padding): This is useful when you need to send a request to another domain. Ajax can be used only to send requests to the same domain.
- **REST**: This is used to send an Ajax request to the server, using RESTful HTTP verbs, such as `GET`, `POST`, `PUT`, and `DELETE`.

We already saw an example of the REST proxy in this chapter. Let's take a look at an example for JSONP:

```
var myStore = Ext.create('Ext.data.Store', {
  model: 'Products',
  proxy: {
    type: 'jsonp',
    url : 'http://domain.com/products',
    callbackKey: 'productsCallback'
  }
});
```

To do – a RESTful sample project

Now, let's create a simple ToDo application that will utilize some of the concepts learned in this chapter and the previous chapter. We'll use the REST proxy of the store to connect to the REST service.

To create a simple RESTful service, we will use the Go language, also known as Golang. This language was initially developed by Google. It's a statically-typed language with its syntax loosely derived from that of C.

You don't have to learn Go to understand this project. This project will primarily focus on Ext JS. You can replace the Go service code with a RESTful service created in any other language that you're familiar with. I'll provide a brief introduction to Go.

Here is the design of the ToDo application that we will create:

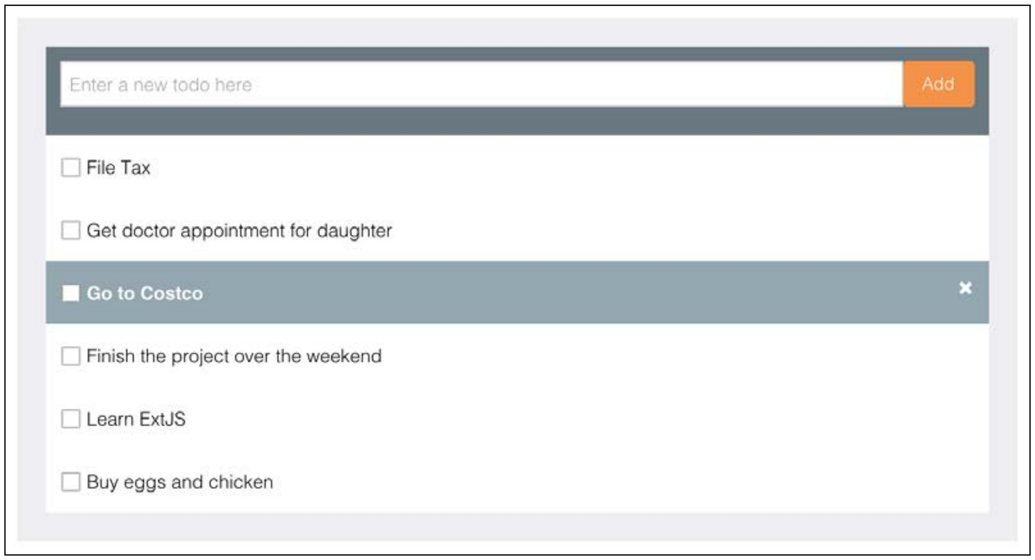


Figure 4.1

Let's take a look at some of the important files in this project. The complete source code (except libraries used, such as Ext JS, Go, and Gorilla mux) for this project is available at `https://github.com/ananddayalan/extjs-by-example-todo`.

The folder structure of this application is shown in the following screenshot:

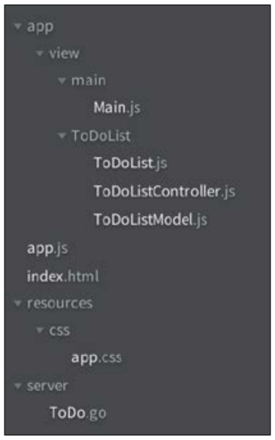


Figure 4.2

For this project, let's first create the store that we've already learned in this chapter. Here, I will keep the store inside the `ViewModel`, but it's absolutely possible to keep them separately.

The following `ViewModel` has three fields: `id`, `desc` (description), `done` (indicates whether or not ToDo has been completed). The proxy is set to `rest` with the URL as tasks. As it's a REST proxy, it will generate the service according to the operation performed with respective HTTP verbs.

For example, when you delete a record, the service request URL will be `<base URL>/tasks/{id}`. So, if you host the application in `localhost` at port `9001`, then the request URL will be `http://localhost:9001/tasks/23333`, and the HTTP verb `DELETE` will be used. Here, `23,333` is the ID of the record. When you add a record, the URL will be `http://localhost:9001/tasks`, the HTTP verb POST will be used, and the added record will be sent to the server as a JSON:

```
Ext.define('ToDo.view.toDoList.ToDoListModel', {
  extend: 'Ext.app.ViewModel',
  alias: 'viewmodel.todoList',
  stores: {
    todos: {
      fields: [ {   name: 'id', type: 'string'  },
      {   name: 'desc', type: 'string'    }],
      autoLoad: true,
      sorters: [{
        property: 'done',
        direction: 'ASC'
      }],
      proxy: {
        type: 'rest',
        url: 'tasks',
        reader: {
          type: 'json',
        },
        writer: {
          type: 'json'
        }
      }
    }
  }
});
```

Now, let's create the view. Here, we will create a view to keep the list of `todos`, a textbox to enter a new todo, and a button to add it.

The UI for the list of `todos` has to be dynamically created based on the records in the store, and whenever a record is added or removed, the view has to be updated accordingly. The Ext JS grid component can be used for this purpose, but you haven't learned the grid component yet. So, we will create it without using the grid component. This way, you'll also learn how to handle a custom UI.

As the UI for the list of `todos` has to be created dynamically, I will keep this code in the `ViewController`. Here, in the `view`, I will keep the textbox to enter a new task and a button to add it:

```
Ext.define('ToDo.view.toDoList.ToDoList', {
  extend: 'Ext.panel.Panel',

  /* Marks these are required classes to be to loaded before
loading this view */
  requires: [
  'ToDo.view.toDoList.ToDoListController',
  'ToDo.view.toDoList.ToDoListModel'
  ],

  xtype: 'app-todoList',
  controller: 'todoList',

  /* View model of the view. */

  viewModel: {
    type: 'todoList'
  },

  items: [{
    xype: 'container',
    items: [
    {
      xtype: 'container',
      layout: 'hbox',
      cls: 'task-entry-panel',
      defaults: {
        flex: 1
      },
      items: [
      {
        reference: 'newToDo',
        xtype: 'textfield',
        emptyText: 'Enter a new todo here'
      },
      {
        xtype: 'button',
        name: 'addNewToDo',
        cls: 'btn-orange',
        text: 'Add',
        maxWidth: 50,
        handler: 'onAddToDo'
      }]
    }
    ]
  }]
});
```

In the preceding view, I have specified two `cls` called `btn-orange` and `task-entry-panel`. These are CSS classes that are used to add some CSS styles. In our design, you can see that the `Add` button is not in the default color of the theme(crisp) that we used. So, to customize the button, I have specified this CSS class.

Now, let's create our `ViewController`. Here, we will create the UI for the list of `todos` by reading the records from the store in the init function. Here, as soon as the application loads, we will call the store's load method. This will make the rest service call to get the records from the server, and for each record, we will create a row in the UI.

To provide the delete functionality, I will add a delete icon to each todo. Here, the following code is used to bind the click event of an icon. As the UI for todo will be generated dynamically, we will need to use the `event` delegation, as shown in the following code:

```
Ext.getBody().on('click', function (event, target) {
  me.onDelete(event, target);
}, null, {
  delegate: '.fa-times'
});
```

If the UI is not dynamically added, then the normal way of binding the click event will look similar to the following code:

```
Ext.select('fa-times').on('click', function (event, target) {
  me.onDelete(event, target);
});
```

The following is the code for `ViewController` of the `ToDoList` view. The `onAddToDo` method uses the `lookupReference` method by passing the reference name set in the `ToDoList` view:

```
Ext.define('ToDo.view.toDoList.ToDoListController', {
  extend: 'Ext.app.ViewController',
  alias: 'controller.todoList',
  views: ['ToDo.view.toDoList.ToDoList'],

  init: function () {
    var me = this;

    //Here we're calling the load method of the store and passing
an anonymous method for the callback. So, the load will call the
server to get the data first and calls the anonymous method.  The
anonymous method then, add the data to the view.
    this.getViewModel().data.todos.load(function (records) {

      Ext.each(records, function (record) {
        //Add a container for each record
        me.addToDoToView(record);
      });
    });

    Ext.getBody().on('click', function (event, target) {
      me.onDelete(event, target);
    }, null, {
      delegate: '.fa-times'
    });

  },

  onAddToDo: function () {

    var store = this.getViewModel().data.todos;

    var desc = this.lookupReference('newToDo').value.trim();
    if (desc != '') {
      store.add({
        desc: desc
      });
      store.sync({
        success: function (batch, options) {
          this.lookupReference('newToDo').setValue('');
          this.addToDoToView(options.operations.create[0]);
        },
        scope: this
      });
    }

  },

  addToDoToView: function (record) {
    this.view.add([{
      xtype: 'container',
      layout: 'hbox',
      cls: 'row',
      items: [
      {
        xtype: 'checkbox',
        boxLabel: record.get('desc'),
        checked: record.get('done'),
        flex: 1
      },
      {
        html: '<a class="hidden" href="#"><i taskId="' +
record.get('id') + '" class="fa fa-times"></i></a>',
      }]
    }]);
  },

  onDelete: function (event, target) {
    var store = this.getViewModel().data.todos;
    var targetCmp = Ext.get(target);
    var id = targetCmp.getAttribute('taskId');
    store.remove(store.getById(id));
    store.sync({
      success: function () {
        this.view.remove(targetCmp.up('.row').id)
      },
      scope: this
    });
  }
});
```

Finally, let's create the REST service in Go. There are multiple ways to install the Go language in your machine. Go can be installed in Mac OS, Windows, Linux, and so on. The easiest way to install Go is to download the installer package from `https://golang.org`.

After you install the Go language, you will need to set the `GOROOT` if you have chosen to install it in a custom path. For example, if you have installed Go in your home directory, you should add the following commands to the `$HOME/.profile`:

```
export GOROOT=/usr/local/go
export PATH=$PATH:$GOROOT/bin
```

For this project, I will use a router called Gorilla mux. So, install the Gorilla mux using the following command after you install Go:

```
go get github.com/gorilla/mux
```

The following is the REST service code. Basically, the main method binds the different method for each HTTP verb, and the respective methods handle the respective requests to perform add, update, delete, and get:

This code doesn't persist the data in a database or any external storage. All the data will be in the memory; when you close the application, the data will be destroyed.

```
package main

import (
"fmt"
"encoding/json"
"net/http"
"strconv"
"github.com/gorilla/mux"
)

type Task struct {
  Id string `json:"id"`
  Desc string `json:"desc"`
  Done bool `json:"done"`
}

var tasks map[string] *Task

func GetToDo(rw http.ResponseWriter, req * http.Request) {

  vars := mux.Vars(req)
  task := tasks[vars["id"]]

  js, err := json.Marshal(task)
  if err != nil {
    http.Error(rw, err.Error(), http.StatusInternalServerError)
    return
  }

  fmt.Fprint(rw, string(js))
}

func UpdateToDo(rw http.ResponseWriter, req * http.Request) {

  vars := mux.Vars(req)

  task:= tasks[vars["id"]]

  dec:= json.NewDecoder(req.Body)
  err:= dec.Decode( & task)
  if err != nil {
    http.Error(rw, err.Error(), http.StatusInternalServerError)
    return
  }

  task.Id = vars["id"]

  retjs, err:= json.Marshal(task)
  if err != nil {
    http.Error(rw, err.Error(), http.StatusInternalServerError)
    return
  }

  fmt.Fprint(rw, string(retjs))
}

func DeleteToDo(rw http.ResponseWriter, req * http.Request) {

  vars := mux.Vars(req)
  delete(tasks, vars["id"])
  fmt.Fprint(rw, "{status : 'success'}")
}

func AddToDo(rw http.ResponseWriter, req * http.Request) {

  task:= new(Task)

  dec:= json.NewDecoder(req.Body)
  err:= dec.Decode( & task)
  if err != nil {
    http.Error(rw, err.Error(), http.StatusInternalServerError)
    return
  }

  tasks[task.Id] = task

  retjs, err:= json.Marshal(task)
  if err != nil {
    http.Error(rw, err.Error(), http.StatusInternalServerError)
    return
  }

  fmt.Fprint(rw, string(retjs))
}

func GetToDos(rw http.ResponseWriter, req * http.Request) {

  v := make([]*Task, 0, len(tasks))

  for  _, value := range tasks {
    v = append(v, value)
  }
  js, err:= json.Marshal(v)
  if err != nil {
    http.Error(rw, err.Error(), http.StatusInternalServerError)
    return
  }

  fmt.Fprint(rw, string(js))
}

func main() {

  var port = 9001
  router:= mux.NewRouter()
  tasks = make(map[string] *Task)
  router.HandleFunc("/tasks", GetToDos).Methods("GET")
  router.HandleFunc("/tasks", AddToDo).Methods("POST")
  router.HandleFunc("/tasks/{id}", GetToDo).Methods("GET")
  router.HandleFunc("/tasks/{id}", UpdateToDo).Methods("PUT")
  router.HandleFunc("/tasks/{id}", DeleteToDo).Methods("DELETE")
  router.PathPrefix("/").Handler(http.FileServer(http.Dir("../")))

  fmt.Println("Listening on port", port)
  http.ListenAndServe("localhost:" + strconv.Itoa(port), router)
}
```

To run the application, use the following command from the terminal or the command window:

```
go run ToDo.go
```

If there is no error, it should show something similar to the following code:

```
Listening on port 9001
```

Now, you can open the `localhost:9001` in the browser to see the application:

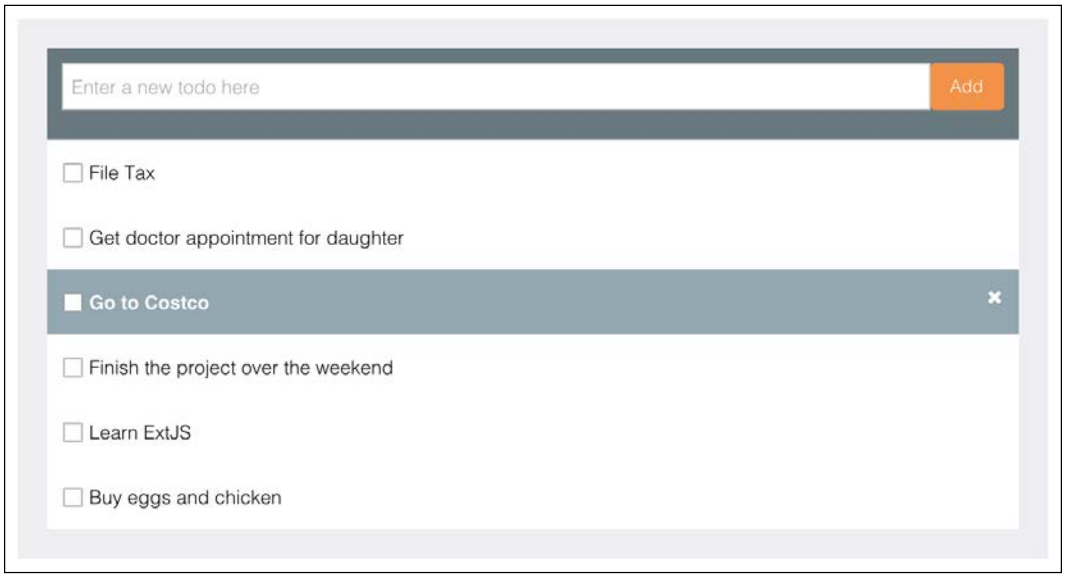


Figure 4.3

Again, here we looked at only some of the important files in this project. The complete source code (except libraries used, such as Ext JS, Go, and Gorilla mux) for this project is available at `https://github.com/ananddayalan/extjs-by-example-todo`.

Summary

In this chapter, you learned how to create the model, store, proxies, and how to work with data. We also looked at how to create a sample application with a simple RESTful API, which was created in the Go language.

5
Working with Grids

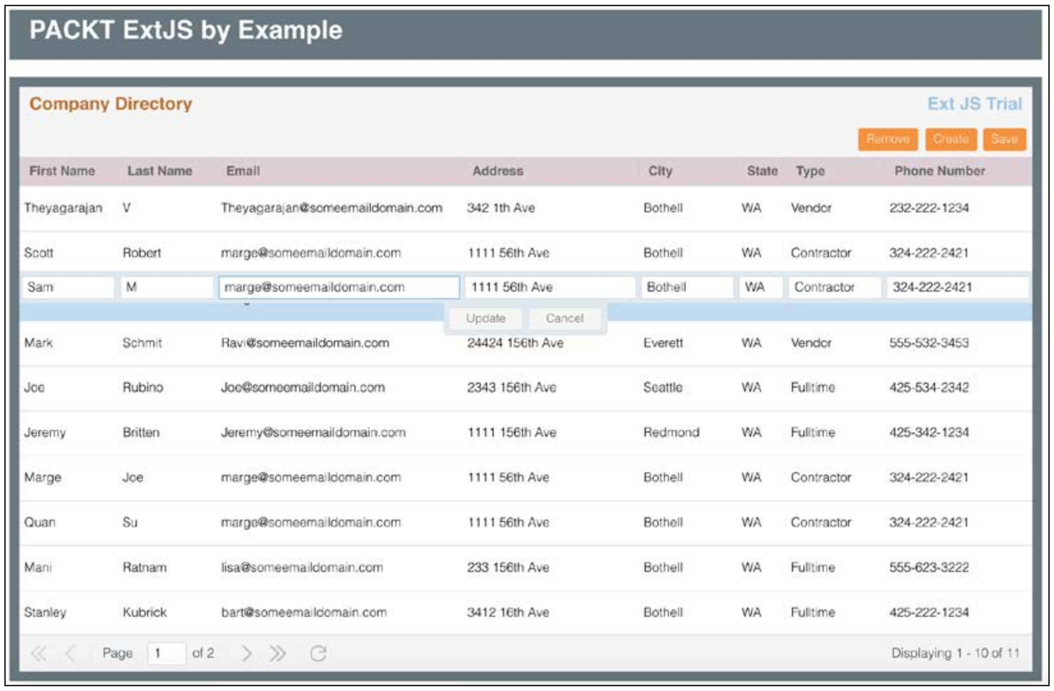
This chapter explores different types of grids components in Ext JS at an advanced level. It will also help readers build a fully functional company directory. The following topics will be covered in this chapter:

- The basic grid
- Sorting
- Renderer
- Filtering
- Pagination
- Cell editing
- Row editing
- Grouping
- The pivot grid
- Company Directory — a sample project

The grid panel is one of the most powerful components in Ext JS. With so many options and configurations, you can build any kind of grid the way you want.

Ext JS provides complete support to pagination, sorting, filtering, searching, row editing, cell editing, grouping, docking toolbars, buffered scrolling, column resizing and hiding, grouped header, multiple sort, row expander, and so on.

The main goal of this chapter is to show you the features of the Ext JS grid and how to use these features to build a sample application called **Company Directory**. The design of a company directory sample application is shown in the following screenshot.



First Name	Last Name	Email	Address	City	State	Type	Phone Number
Theyagarajan	V	Theyagarajan@someemaildomain.com	342 1th Ave	Bothell	WA	Vendor	232-222-1234
Scott	Robert	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Sam	M	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Mark	Schmit	Ravi@someemaildomain.com	24424 156th Ave	Everett	WA	Vendor	555-532-3453
Joe	Rubino	Joe@someemaildomain.com	2343 156th Ave	Seattle	WA	Fulltime	425-534-2342
Jeremy	Britten	Jeremy@someemaildomain.com	1111 156th Ave	Redmond	WA	Fulltime	425-342-1234
Marge	Joe	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Quan	Su	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Mani	Ratnam	lisa@someemaildomain.com	233 156th Ave	Bothell	WA	Fulltime	555-623-3222
Stanley	Kubrick	bart@someemaildomain.com	3412 16th Ave	Bothell	WA	Fulltime	425-222-1234

Figure 5.1

The most important new components used in this sample project are the grid, the paging toolbar, and the row editing plugin. The sample project will also use many other concepts, which you have learned in the previous chapters.

The basic grid

Let's start with a really simple basic grid. To create a grid with the Grid (`Ext.grid.Panel`) component, at the minimum, you need to specify the list columns and store to get the data. The class name for the grid in the modern toolkit is `Ext.grid.Grid`, but in the classic toolkit, it is `Ext.grid.Panel`. There are some minor differences that exist between the classic and modern toolkit, but most of the concepts are same.

Let's first create a simple store with an inline hardcoded data and a model to use.

The following is the model with three string fields. The store has the inline data and specifies the model to use:

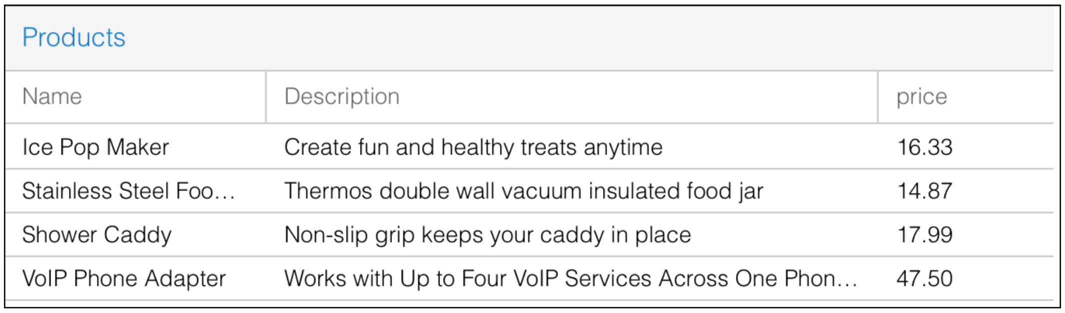
```
Ext.define('Product', {
  extend: 'Ext.data.Model',
  fields: [ 'id', 'productname', 'desc', 'price' ]
});

var productStore = Ext.create('Ext.data.Store', {
  model: 'Product',
  data: [{
    id: 'P1',
    productname: 'Ice Pop Maker',
    desc: 'Create fun and healthy treats anytime',
    price: '$16.33'
  }, {
    id: 'P2',
    productname: 'Stainless Steel Food Jar',
    desc: 'Thermos double wall vacuum insulated food jar',
    price: '$14.87'
  },{
    id: 'P3',
    productname: 'Shower Caddy',
    desc: 'Non-slip grip keeps your caddy in place',
    price: '$17.99'
  }, {
    id: 'P4',
    productname: 'VoIP Phone Adapter',
    desc: 'Works with Up to Four VoIP Services Across One Phone
Port',
    price: '$47.50'
  }
  ]
});
```

Now, let's create the grid using `Ext.grid.Panel` and use the store created before. For each column, either the width or the flex can be specified. Here, we have set the flex to `1` for the '`Description`' column, which makes this column use the rest of the available width that is not used by the other two columns.

The `dataIndex` property of the column has to match with one of the fields in the model. A column can be kept hidden by setting the hidden property of the column to `True` for the `id` column, as shown in the following code:

```
Ext.create('Ext.grid.Panel', {
  renderTo: Ext.getBody(),
  store: productStore,
  width: 600,
  title: 'Products',
  columns: [
  {
    text: 'Id',
    dataIndex: 'id',
    hidden: true
  },
  {
    text: 'Name',
    width:150,
    dataIndex: 'productname'
  },
  {
    text: 'Description',
    dataIndex: 'desc',
    sortable: false,
    flex: 1
  },
  {
    text: 'price',
    width: 100,
    dataIndex: 'price'
  }
  ]
});
```

The output of the preceding code is shown in the following screenshot. By default, the column size is resizable; you can have a fixed width if required.



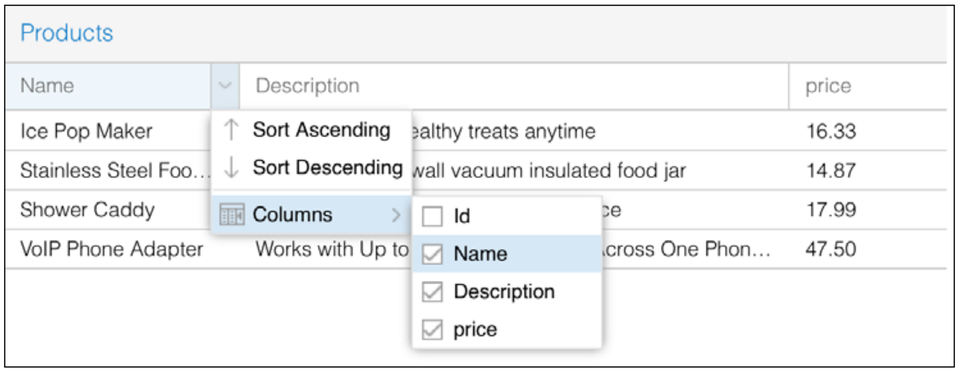
Products

Name	Description	price
Ice Pop Maker	Create fun and healthy treats anytime	16.33
Stainless Steel Foo...	Thermos double wall vacuum insulated food jar	14.87
Shower Caddy	Non-slip grip keeps your caddy in place	17.99
VoIP Phone Adapter	Works with Up to Four VoIP Services Across One Phon...	47.50

Figure 5.2

Sorting

This can be done using the default column menu. Here is how the default column menu looks. This can be used to sort, hide, or show columns. The column menu will be displayed if you click on the small arrow icon in each column. Sorters can be added using the UI or from the code:



Products

Name	Description	price
Ice Pop Maker	ealthy treats anytime	16.33
Stainless Steel Foo…	wall vacuum insulated food jar	14.87
Shower Caddy	ce	17.99
VoIP Phone Adapter	Works with Up to cross One Phon…	47.50

In the code, as we have set `sortable` to `false` for the `Description` column, the options to sort for this column will be disabled.

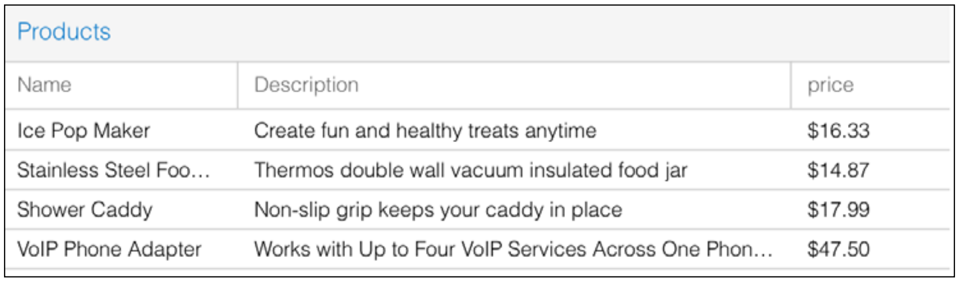
By default, the sort happens on the client side. To enable sort on the server side, you have to set the `remoteSort` property of the store to `true`. When you set `remoteSort` to `true`, the store will send the sorting information (the field and sort order) to the server in the request params.

Renderer

The `renderer` config of a column can be used to alter how the data is rendered for this column. You may have noticed that the price column in the preceding grids in *Figure 5.1* and *Figure 5.2* doesn't show the currency symbol. Now, let's use `renderer` to prefix the currency symbol for the price column:

```
Ext.create('Ext.grid.Panel', {
  renderTo: Ext.getBody(),
  store: productStore,
  width: 600,
  title: 'Products',
  columns: [{
    text: 'Id',
    dataIndex: 'id',
    hidden: true
  }, {
    text: 'Name',
    width:150,
    dataIndex: 'productname'
  }, {
    text: 'Description',
    dataIndex: 'desc',
    sortable: false,
    flex: 1
  }, {
    text: 'price',
    width: 100,
    dataIndex: 'price',

    renderer: function(value) {
      return Ext.String.format('${0}', value);
    }
  }
  ]
});
```

The output is shown here:



Products

Name	Description	price
Ice Pop Maker	Create fun and healthy treats anytime	$16.33
Stainless Steel Foo...	Thermos double wall vacuum insulated food jar	$14.87
Shower Caddy	Non-slip grip keeps your caddy in place	$17.99
VoIP Phone Adapter	Works with Up to Four VoIP Services Across One Phon...	$47.50

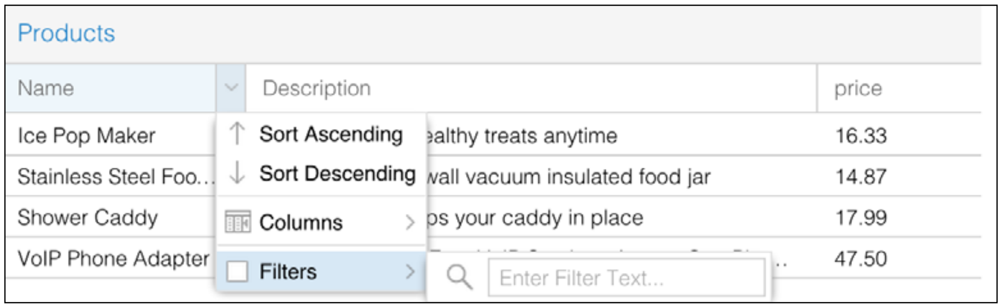
Similarly, you can use the `renderer` to render HTML tags to the column. You can also add URLs and render images in the column as well.

Filtering

The filtering capability can be easily added to the grid by simply adding the `Ext.grid.filters.Filters` (ptype: `gridfilters`) plugin and the filter config to the required columns:

```
Ext.create('Ext.grid.Panel', {
  renderTo: Ext.getBody(),
  store: productStore,
  plugins: 'gridfilters',
  width: 600,
  title: 'Products',
  columns: [
  {
    text: 'Id',
    dataIndex: 'id',
    hidden: true
  },
  {
    text: 'Name',
    width:150,
    dataIndex: 'productname',
    filter:'string'
  },
  {
    text: 'Description',
    dataIndex: 'desc',
    sortable: false,
    flex: 1,
    filter: {
      type: 'string',
      itemDefaults: {   emptyText: 'Search for…' }
    }
  },
  {
    text: 'price',
    width: 100,
    dataIndex: 'price'
  }
  ]
});
```

For each column, you can specify the filter type, such as `string`, `bool`, and so on, and the additional configs (such as `emptyText`) of the search field.



Products

Name	Description	price
Ice Pop Maker	ealthy treats anytime	16.33
Stainless Steel Foo...	wall vacuum insulated food jar	14.87
Shower Caddy	ps your caddy in place	17.99
VoIP Phone Adapter	..	47.50

Here, we will add the filter while creating the gird, but the filter can also be added later after the creation of the grid.

Pagination

Now, what if there are thousands of records? Well, you definitely don't want to load several thousands of records, so the better way is to either add the pagination support with the paging toolbar or use the buffered infinite scroll.

Let's add the paging toolbar to the preceding example using `Ext.toolbar.Paging` (`xtype: pagingtoolbar`). We will add this paging toolbar to `dockedItems`, which is a property of `Ext.panel.Panel`, and the items can be docked either at the top, bottom, left, or right-hand side of the panel.

```
Ext.create('Ext.grid.Panel', {
  renderTo: Ext.getBody(),
  store: productStore,
  width: 600,
  title: 'Products',
  columns: [
  { text: 'Id', dataIndex: 'id', hidden: true },
  { text: 'Name',  width:150, dataIndex: 'productname'},
  { text: 'Description', dataIndex: 'desc', sortable: false, flex: 
1},
  { text: 'price', width: 100, dataIndex: 'price'  }
  ],
  dockedItems: [{
    xtype: 'pagingtoolbar',
    store: productStore,
    dock: 'bottom',
    displayInfo: true
  }]
});
```

Then, the code for the store follows. This store reads the data from a JSON file:

```
var productStore = Ext.create('Ext.data.Store', {
  model: 'Product',
  pageSize: 10,
  autoLoad: true,
  proxy: {
    type: 'ajax',
    url: 'data.json',
    reader: {
      type: 'json',
            rootProperty: 'data',
            totalProperty: 'total'
        }
    }
});
```

The `rootProperty` tells the store where to find the records in the JSON file, and the `totalProperty` lets the store know from where to read the total number of records count. To get the right behavior, you need to specify `totalProperty` when you use the server-side pagination. This total value will be used by the paging toolbar.

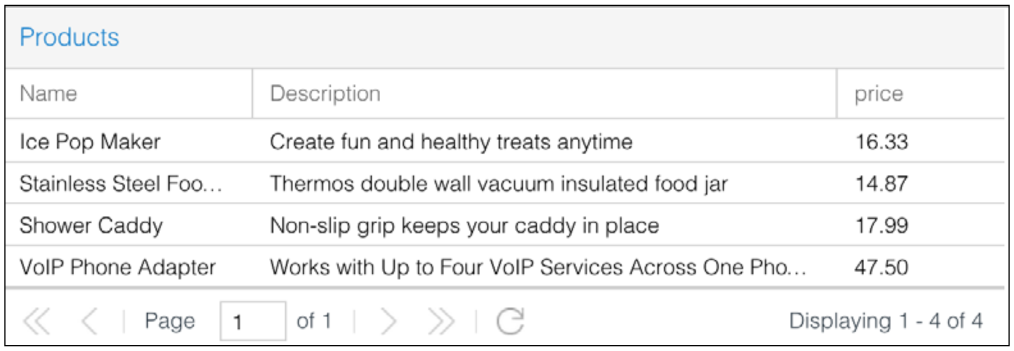
The `pageSize` value `10` will be used by the store when you send the request to the server to limit the number of records. A sample request URL is shown here:

```
http://localhost:8000/data.json?page=1&start=0&limit=10
```

Here, the limit is set to 10 because in the store, we have set pageSize as 10.

> These params have to used by the server to process the pagination and send only the requested records instead of sending the whole set. If you directly read from a JSON file, the response will contain all the records in the JSON file.

The output with the paging toolbar is shown in the following screenshot:



Products

Name	Description	price
Ice Pop Maker	Create fun and healthy treats anytime	16.33
Stainless Steel Foo...	Thermos double wall vacuum insulated food jar	14.87
Shower Caddy	Non-slip grip keeps your caddy in place	17.99
VoIP Phone Adapter	Works with Up to Four VoIP Services Across One Pho...	47.50

Cell editing

The records in the grid can be edited in the place one cell at a time with the cell editing plugin. `Ext.grid.plugin.CellEditing` can be added to the grid to support the cell editing.

So, to add the cell editing support to the grid, simply add the `celledititng` plugin and set the editor for the required column. You can add the editor in all the columns to support cell editing on all the columns, or you can add editors only to the columns, in which you want the edit support.

The editor can be a simple text field, or you can add bit complex components—such as combobox—with an attached store to provide the list of values for the column.

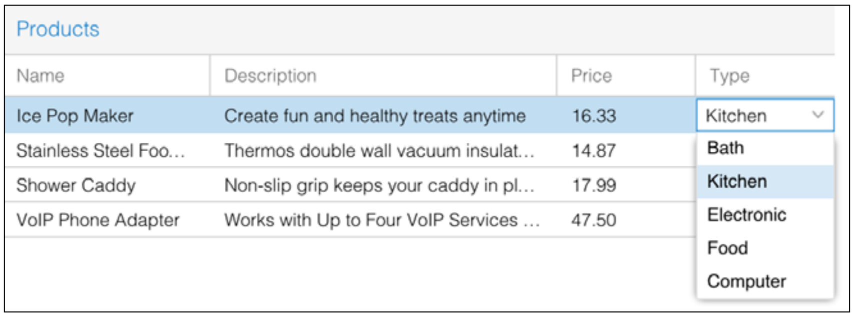
The following example adds a basic text field editor to one of the columns and combobox with an attached store to another column:

```
Ext.create('Ext.grid.Panel', {
  renderTo: Ext.getBody(),
  store: productStore,

  plugins: ['cellediting','gridfilters'],

  width: 600,
  title: 'Products',

  columns: [
  {
    text: 'Id',
    dataIndex: 'id',
    hidden: true
  },
  {
    text: 'Name',
    width:150,
    dataIndex: 'productname',
    filter:'string',
    editor: {
      allowBlank: false,
      type: 'string'
    }
  },
  {
    text: 'Description',
    dataIndex: 'desc',
    sortable: false,
    flex: 1
  },
  {   text: 'Price',
    width: 100,
    dataIndex: 'price'
  },
  {
    text: 'Type',
    width: 100,
    dataIndex: 'type',
    editor: new Ext.form.field.ComboBox({
      typeAhead: true,
      triggerAction: 'all',
      store: [
      ['Bath','Bath'],
      ['Kitchen','Kitchen'],
      ['Electronic','Electronic'],
      ['Food', 'Food'],
      ['Computer', 'Computer']
      ]
    })
  } ]
});
```

In the preceding example, the following code is used to set the editor for the `Type` column. Here, the store has the hardcoded inline data, but it can be configured to talk to the server to get the data from the server:

```
    editor: new Ext.form.field.ComboBox({
      typeAhead: true,
      triggerAction: 'all',
      store: [
      ['Bath','Bath'],
      ['Kitchen','Kitchen'],
      ['Electronic','Electronic'],
      ['Food', 'Food'],
      ['Computer', 'Computer']
      ]
    })
```

This can be any supported fields, such as a date picker, checkbox, radio options, a number field, and so on. Also, you can add validations to editors.

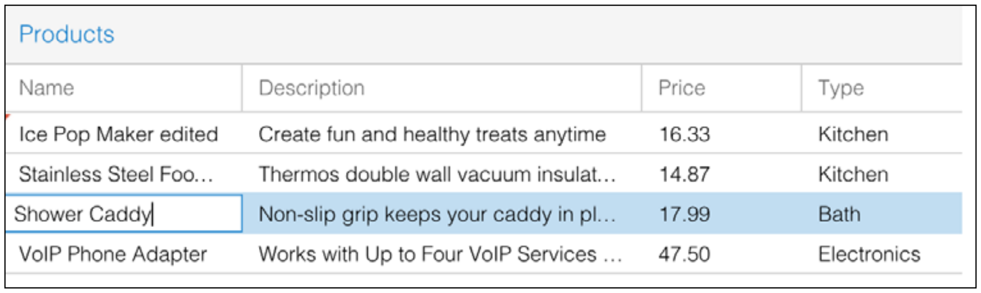

Products

Name	Description	Price	Type
Ice Pop Maker	Create fun and healthy treats anytime	16.33	Kitchen
Stainless Steel Foo...	Thermos double wall vacuum insulat...	14.87	
Shower Caddy	Non-slip grip keeps your caddy in pl...	17.99	
VoIP Phone Adapter	Works with Up to Four VoIP Services ...	47.50	

Bath
Kitchen
Electronic
Food
Computer

Figure 5.7: Combo box editor

The output for the preceding code is shown in the following screenshot:



Name	Description	Price	Type
Ice Pop Maker edited	Create fun and healthy treats anytime	16.33	Kitchen
Stainless Steel Foo...	Thermos double wall vacuum insulat...	14.87	Kitchen
Shower Caddy	Non-slip grip keeps your caddy in pl...	17.99	Bath
VoIP Phone Adapter	Works with Up to Four VoIP Services ...	47.50	Electronics

Figure 5.8: The text field editor

Once the records are edited, by default, it won't save the records to the server. You have to set the `autosync` for store property to `true`, which will trigger a request to the server for any CRUD operation.

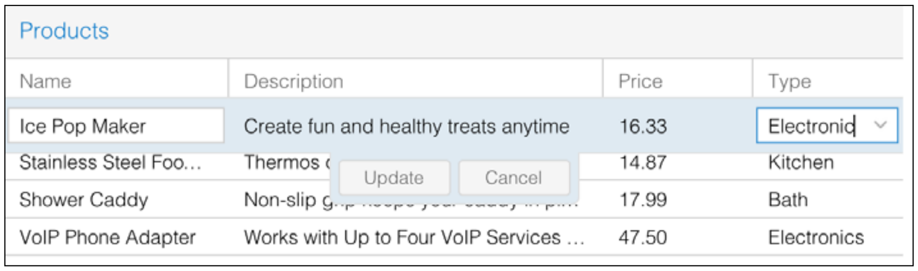
If you don't want the sync to happen immediately, then you can call the `save` or `sync` method of the store when needed. You can add the `Save` button to the grid header and call the store's `save` or `sync` method.

In *Figure 5.8: The text field editor*, note the small red mark in the top-left corner in the first cell of the first row. This is an indicator to let the user know that the record has been updated.

Row editing

In cell editing, you can edit one cell at a time, but in row editing, you can edit one row at a time.

Guess what? To use row editing instead of cell editing, all you need to do is to use the row editing plugin: `Ext.grid.plugin.RowEditing` instead of the cell editing plugin. So, if you just use the line plugins: `['rowediting','gridfilters']` instead of the line plugins: `['cellediting','gridfilters']` in the preceding code given for cell editing, you'll get the following output:



Name	Description	Price	Type
Ice Pop Maker	Create fun and healthy treats anytime	16.33	Electronic
Stainless Steel Foo...	Thermos (	14.87	Kitchen
Shower Caddy	Non-slip g...	17.99	Bath
VoIP Phone Adapter	Works with Up to Four VoIP Services ...	47.50	Electronics

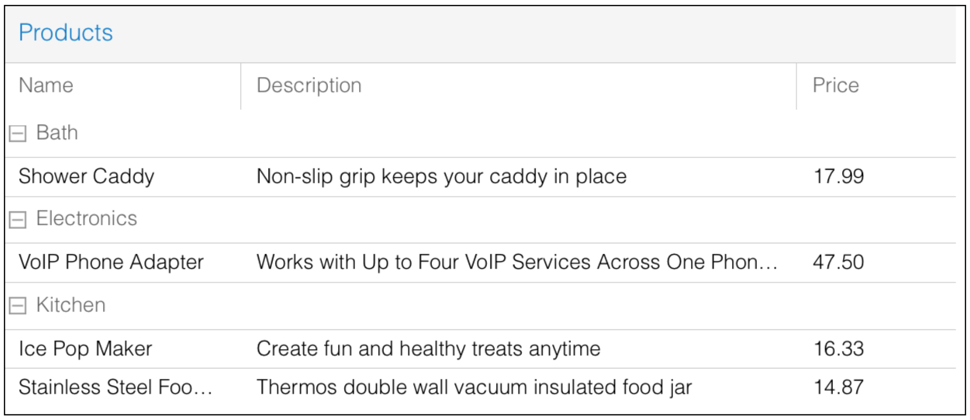
The row editing plugin is also used in the sample project in this chapter. You can find some additional information there.

Grouping

In order to group a column, you need to specify the grouping field using the `groupField` property of the store, and we need to set `Ext.grid.feature.Feature` in the grid, as shown in the following code:

```
var productStore = Ext.create('Ext.data.Store', {
  model: 'Product',
  pageSize: 10,
  autoLoad: true,
  proxy: {
    type: 'ajax',
    url: 'data.json',
    reader: {
      type: 'json',
      rootProperty: 'data',
      totalProperty: 'total'
    }
  },
  groupField: 'type'
});


Ext.create('Ext.grid.Panel', {
  renderTo: Ext.getBody(),
  store: productStore,
  width: 600,
  title: 'Products',
  features: [{
    id: 'group',
    ftype: 'grouping',
    groupHeaderTpl : '{name}',
    hideGroupedHeader: true,
    enableGroupingMenu: false
  }],
  columns: [{
    text: 'Id',
    dataIndex: 'id',
    hidden: true
  },{
    text: 'Name',
    width:150,
    dataIndex: 'productname'
  },{
    text: 'Description',
    dataIndex: 'desc',
    sortable: false,
    flex: 1,
    groupable:false
  },{
    text: 'Price',
    width: 100,
    dataIndex: 'price'
  },{
    text: 'Type',
    width: 100,
    dataIndex: 'type'
  }
  ]
});
}
});
```

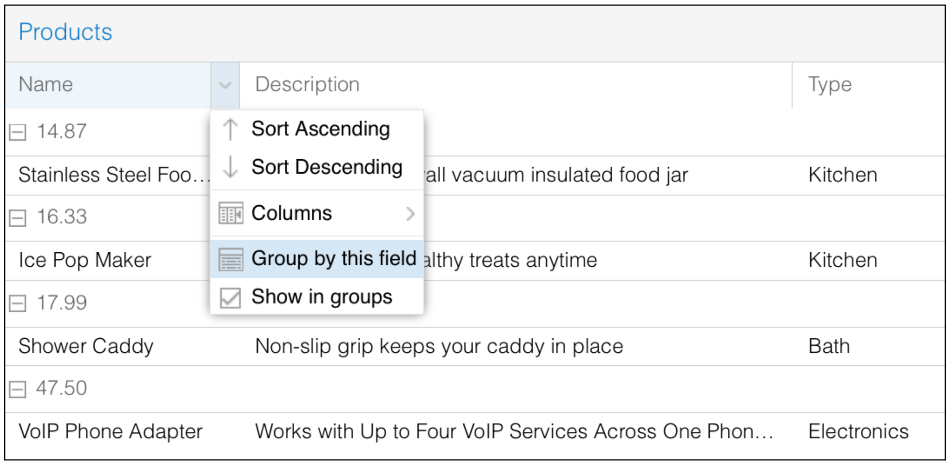
The output of the preceding code is shown here:



Products

Name	Description	Price
⊟ Bath		
Shower Caddy	Non-slip grip keeps your caddy in place	17.99
⊟ Electronics		
VoIP Phone Adapter	Works with Up to Four VoIP Services Across One Phon…	47.50
⊟ Kitchen		
Ice Pop Maker	Create fun and healthy treats anytime	16.33
Stainless Steel Foo…	Thermos double wall vacuum insulated food jar	14.87

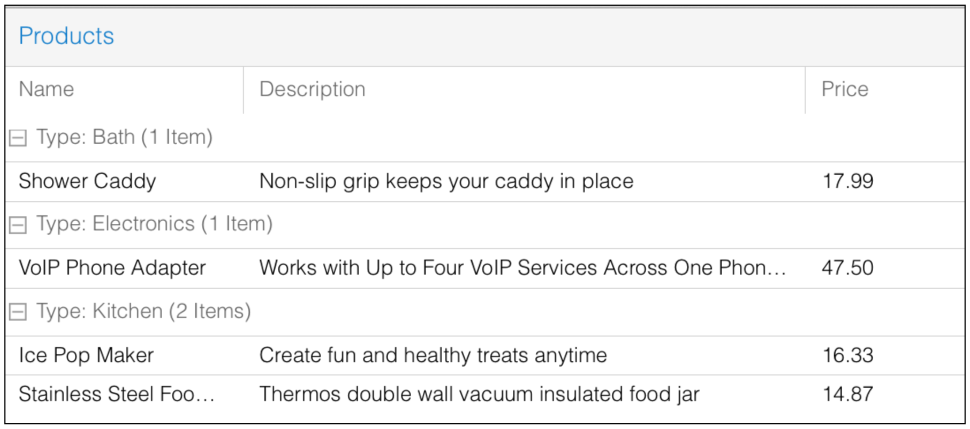
The following figure shows the grouping menu. Using this menu, at runtime, you can group by other fields in the grid. This option can be disabled for all the columns by setting the `enableGroupingMenu` property of the grouping feature to `false`. You can disable this grouping for specific columns alone by setting the `groupable` property of the column to `false`.



Products

Name	Description	Type
14.87		
Stainless Steel Foo…	all vacuum insulated food jar	Kitchen
16.33		
Ice Pop Maker	althy treats anytime	Kitchen
17.99		
Shower Caddy	Non-slip grip keeps your caddy in place	Bath
47.50		
VoIP Phone Adapter	Works with Up to Four VoIP Services Across One Phon…	Electronics

The grouping template can be used to add additional information, such as the number of items in the group or any other information that you want to include by replacing the `groupHeaderTpl : '{name}',` line in the preceding example code with the `groupHeaderTpl : '{columnName}: {name} ({rows.length} Item{[values.rows.length > 1 ? "s" : ""]})',` line. This will give the the following output:



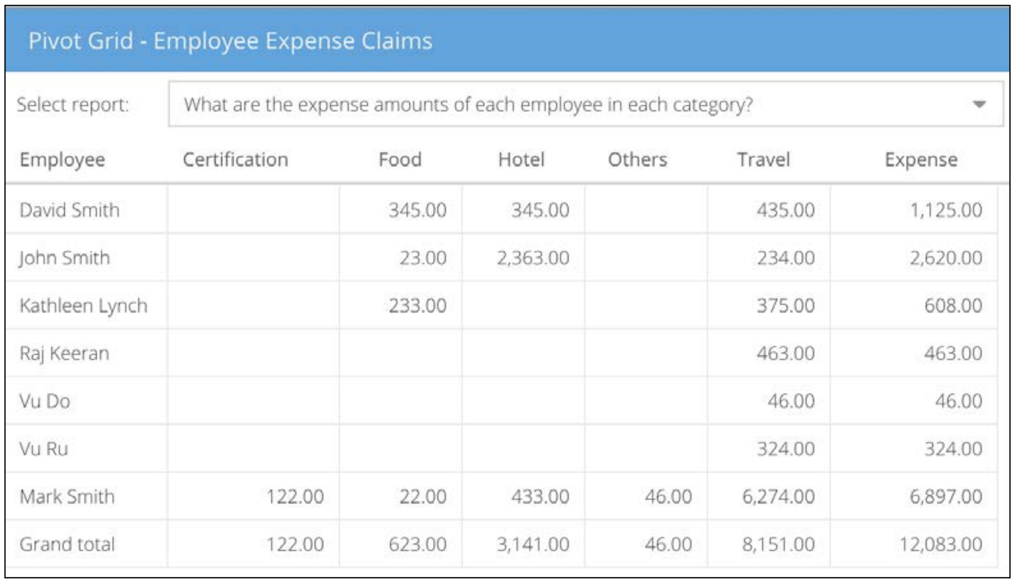
Products

Name	Description	Price
Type: Bath (1 Item)		
Shower Caddy	Non-slip grip keeps your caddy in place	17.99
Type: Electronics (1 Item)		
VoIP Phone Adapter	Works with Up to Four VoIP Services Across One Phon…	47.50
Type: Kitchen (2 Items)		
Ice Pop Maker	Create fun and healthy treats anytime	16.33
Stainless Steel Foo…	Thermos double wall vacuum insulated food jar	14.87

The pivot grid

This lets you reorganize and summarize selected columns and rows of data to obtain a desired report.

Let's say you have a list of expense data submitted by the employees of a company, and you want to see the total expense amount claimed by each employee in each category. An example of the expected result is shown in the following screenshot:



Pivot Grid - Employee Expense Claims

Select report: What are the expense amounts of each employee in each category?

Employee	Certification	Food	Hotel	Others	Travel	Expense
David Smith		345.00	345.00		435.00	1,125.00
John Smith		23.00	2,363.00		234.00	2,620.00
Kathleen Lynch		233.00			375.00	608.00
Raj Keeran					463.00	463.00
Vu Do					46.00	46.00
Vu Ru					324.00	324.00
Mark Smith	122.00	22.00	433.00	46.00	6,274.00	6,897.00
Grand total	122.00	623.00	3,141.00	46.00	8,151.00	12,083.00

So, in the grid, instead of showing the list expenses submitted, you need to reorganize and summarize the columns to get this result. This can easily be achieved using the pivot grid. Note that except the first column, the column's headers are the value of the expense category in the list of expense data. So, you see that the data representation has been reorganized and summarized.

When you use the pivot grid, you need to provide two important things: the axis and aggregates. You should use the axis to specify the row and column placement and aggregation for grouping calculations.

Here is an example:

```
leftAxis: [{
  width: 80,
  dataIndex: 'employee',
  header: 'Employee'
}]

topAxis: [{
  dataIndex: 'cat',
  header: 'Category',
  direction: 'ASC'
}]
```

In the axis, you can also specify sortable, sort direction, filter, and so on.

```
aggregate: [{
  measure: 'amount',
  header: 'Expense',
  aggregator: 'sum',
  align: 'right',
  width: 85,
  renderer: Ext.util.Format.numberRenderer('0,000.00')
}]
```

Here, the aggregator can be `sum`, `avg`, `min`, `max`, and so on.

In the pivot grid, you can also specify the `renderer` to render the data in the custom format:

```
renderer: function(value, meta, record) {

  return Ext.util.Format.number(value, '0,000.00');
}
```

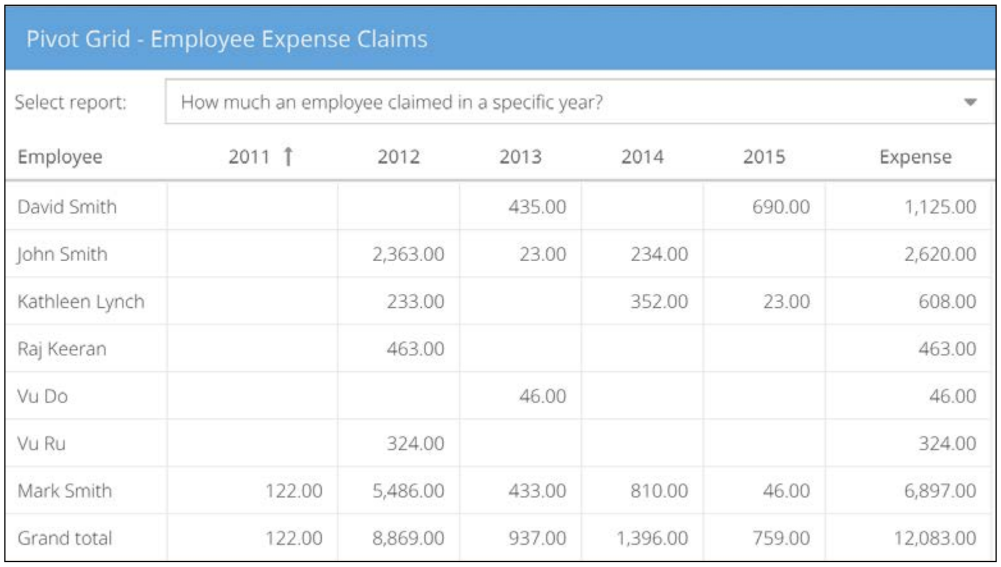
Now, let's create the pivot grid. Here is the store that will be used in the pivot grid:

```
var store = new Ext.data.JsonStore({
  proxy: {
    type: 'ajax',
    url: 'expense.json',
    reader: {
      type: 'json',
      rootProperty: 'rows'
    }
  },
  autoLoad: true,
  fields: [
  {name: 'id',       type: 'int'},
  {name: 'employee',   type: 'string'},
  {name: 'amount',       type: 'int'},
  {name: 'date',     type: 'date', dateFormat: 'd/m/Y'},
  {name: 'cat',        type: 'string'},
  {
    name: 'year',
    convert: function(v, record){
      return Ext.Date.format(record.get('date'), "Y");
    }
  }
  ]
});
```

Here is the sample pivot grid code. Here you can see the `leftAxis` is fixed, but I kept `topAxis` as dynamic in order to make it a reporting tool. Based on the dropdown value selected, the `topAxis` is changed.

```
    var pivotGrid = Ext.create('Ext.pivot.Grid', {
      renderTo: Ext.getBody(),
      title: 'Pivot Grid - Employee Expense Claims',
      height: 600,
      width: 700,
      enableLocking:  false,
      viewConfig: {
        trackOver: true,
        stripeRows: false
      },

      tbar: [{
        xtype:  'combo',
        fieldLabel: 'Select report',
        flex: 1,
        editable: false,
        value: '1',
        store: [
        ['1', 'How much an employee claimed in total?'],
        ['2', 'What are the expense amounts of each employee in each
    category?'],
        ['3', 'How much an employee claimed in a specific year?']
        ],
        listeners: {
          select: function(combo, records, eOpts){
            switch(records.get('field1')){
              case '1':
              pivotGrid.reconfigurePivot({
                topAxis: []
              });
              break;
              case '2':
              pivotGrid.reconfigurePivot({
                topAxis: [{
                  dataIndex: 'cat',
                  header: 'Category',
                  direction: 'ASC'
                }]
              });
              break;

              case '3':
              pivotGrid.reconfigurePivot({
                topAxis: [{
                  dataIndex: 'year',
                  header: 'Year',
                  direction: 'ASC'
                }]
              });
              break;
            }
          }
        }
      }],

      store: store,

      aggregate: [{
        measure: 'amount',
        header: 'Expense',
        aggregator: 'sum',
        align: 'right',
        width: 85,
        renderer: Ext.util.Format.numberRenderer('0,000.00')
      }],

      leftAxis: [{
        width: 80,
        dataIndex: 'employee',
        header: 'Employee'
      }],

      topAxis: []
    });

    }
```

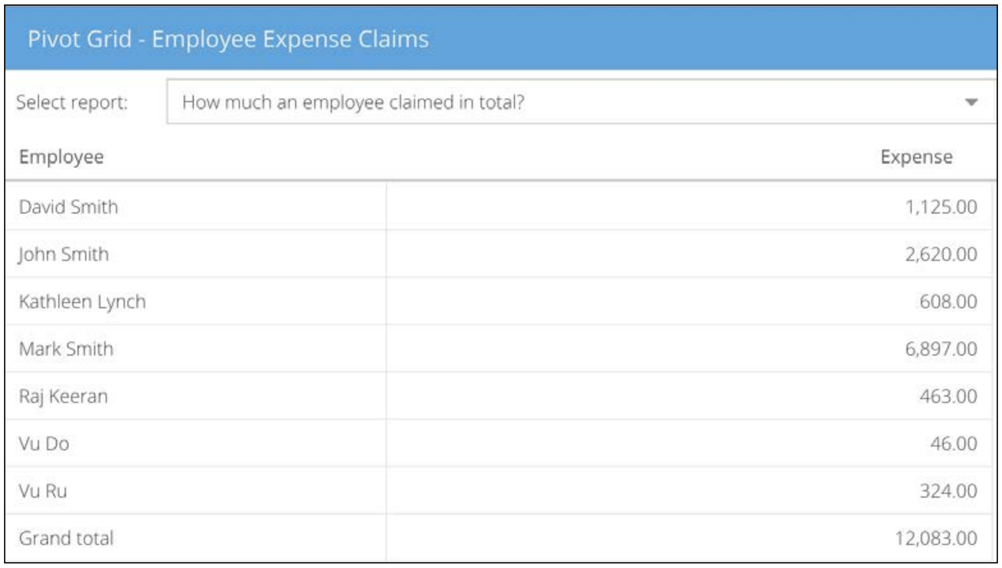
The following screenshots show the output of the preceding code:



Pivot Grid - Employee Expense Claims

Select report: How much an employee claimed in a specific year?

Employee	2011 ↑	2012	2013	2014	2015	Expense
David Smith			435.00		690.00	1,125.00
John Smith		2,363.00	23.00	234.00		2,620.00
Kathleen Lynch		233.00		352.00	23.00	608.00
Raj Keeran		463.00				463.00
Vu Do			46.00			46.00
Vu Ru		324.00				324.00
Mark Smith	122.00	5,486.00	433.00	810.00	46.00	6,897.00
Grand total	122.00	8,869.00	937.00	1,396.00	759.00	12,083.00



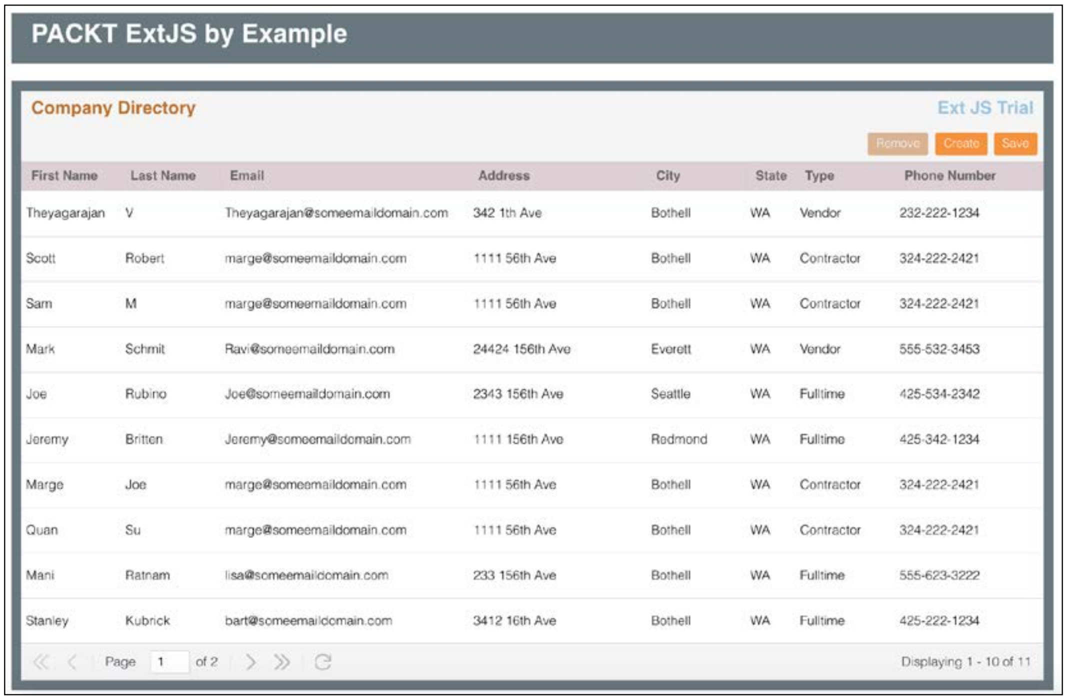
Pivot Grid - Employee Expense Claims

Select report: How much an employee claimed in total?

Employee	Expense
David Smith	1,125.00
John Smith	2,620.00
Kathleen Lynch	608.00
Mark Smith	6,897.00
Raj Keeran	463.00
Vu Do	46.00
Vu Ru	324.00
Grand total	12,083.00

The company directory – a sample project

Now, we will create a sample project using the concepts learned in this chapter and the previous chapter. The following screenshot shows the design of the sample project:

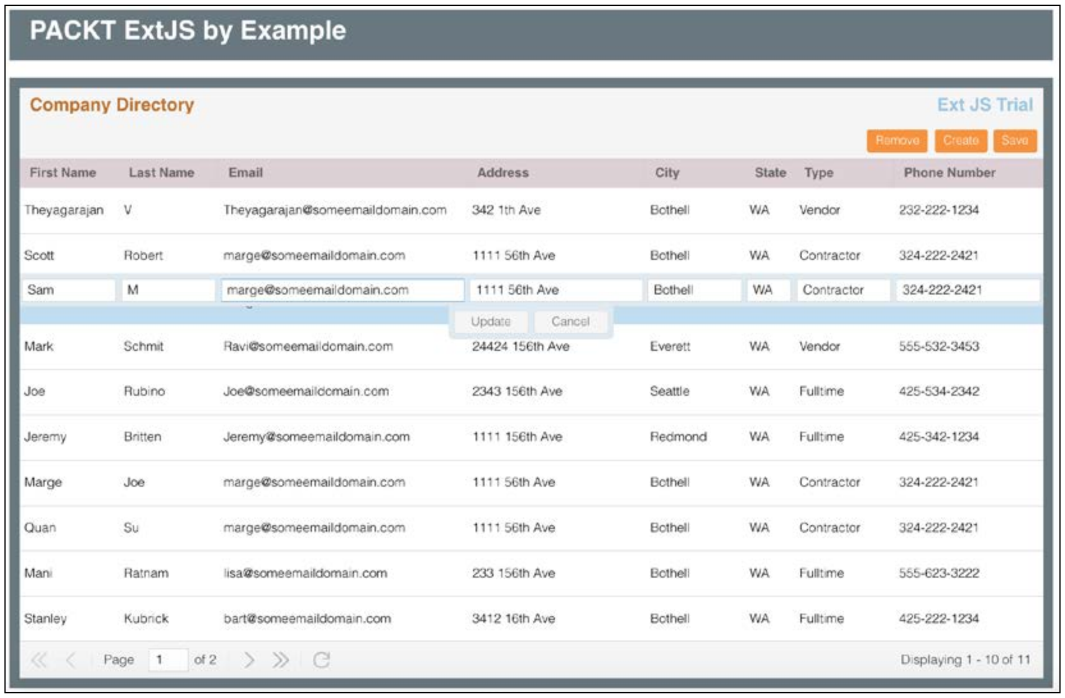


First Name	Last Name	Email	Address	City	State	Type	Phone Number
Theyagarajan	V	Theyagarajan@someemaildomain.com	342 1th Ave	Bothell	WA	Vendor	232-222-1234
Scott	Robert	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Sam	M	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Mark	Schmit	Ravi@someemaildomain.com	24424 156th Ave	Everett	WA	Vendor	555-532-3453
Joe	Rubino	Joe@someemaildomain.com	2343 156th Ave	Seattle	WA	Fulltime	425-534-2342
Jeremy	Britten	Jeremy@someemaildomain.com	1111 156th Ave	Redmond	WA	Fulltime	425-342-1234
Marge	Joe	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Quan	Su	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Mani	Ratnam	lisa@someemaildomain.com	233 156th Ave	Bothell	WA	Fulltime	555-623-3222
Stanley	Kubrick	bart@someemaildomain.com	3412 16th Ave	Bothell	WA	Fulltime	425-222-1234

By looking at the design, you know that the most important component of this project is the grid. Some of the concepts and components used in this project are:

- Grid panel
- `ViewModel`
- Model
- Data store and rest proxy
- Layouts and containers
- `RowEditing` plugins
- Pagination
- REST API in Go
- References

The following screenshot shows the view of the edit/add operation using the RowEditing plugin:



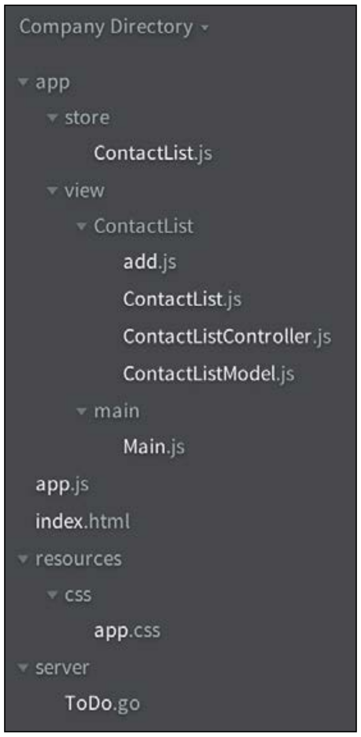
First Name	Last Name	Email	Address	City	State	Type	Phone Number
Theyagarajan	V	Theyagarajan@someemaildomain.com	342 1th Ave	Bothell	WA	Vendor	232-222-1234
Scott	Robert	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Sam	M	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Mark	Schmit	Ravi@someemaildomain.com	24424 156th Ave	Everett	WA	Vendor	555-532-3453
Joe	Rubino	Joe@someemaildomain.com	2343 156th Ave	Seattle	WA	Fulltime	425-534-2342
Jeremy	Britten	Jeremy@someemaildomain.com	1111 156th Ave	Redmond	WA	Fulltime	425-342-1234
Marge	Joe	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Quan	Su	marge@someemaildomain.com	1111 56th Ave	Bothell	WA	Contractor	324-222-2421
Mani	Ratnam	lisa@someemaildomain.com	233 156th Ave	Bothell	WA	Fulltime	555-623-3222
Stanley	Kubrick	bart@someemaildomain.com	3412 16th Ave	Bothell	WA	Fulltime	425-222-1234

Displaying 1 - 10 of 11

Let's take a look at some of the important files in this project.

> The complete source code is not provided in this book, but available for download at `https://github.com/ananddayalan/extjs-by-example-company-directory`.

The following screenshot is the folder structure of this sample project:



```
Company Directory
▾ app
  ▾ store
      ContactList.js
  ▾ view
    ▾ ContactList
        add.js
        ContactList.js
        ContactListController.js
        ContactListModel.js
    ▾ main
        Main.js
  app.js
  index.html
▾ resources
  ▾ css
      app.css
▾ server
    ToDo.go
```

The following view code is the important part of this project. This view renders most of the viewable part of the application. It uses the grid panel and the `RowEditing` plugin:

```
plugins: [{
  ptype: 'rowediting',
  clicksToMoveEditor: 1,
  autoCancel: false
}],
```

The easiest way to add editing capabilities to a grid is to use `RowEditing`. It's not necessary to use the `RowEditing` plugin; if you want, you can use the add/edit form.

For the editing to work, we have to set the field property or the `editor` property of the column. The following line is included as default to all the columns to provide the editing capability:

```
editor: { xtype: 'textfield',  allowBlank: false }
```

By setting validation rules in the editor, the `RowEditing` plugin allows only the data that satisfies the validations. Otherwise, the `Update` button in the `RowEditing` plugin will be disabled.

> As the code for this view is a bit long, in between the code, I have truncated some of the unimportant code. The complete source code for this project is available at `https://github.com/ananddayalan/extjs-by-example-company-directory`.

```
Ext.define('CD.view.contactList.ContactList', {
  extend: 'Ext.panel.Panel',

  requires: ['CD.view.contactList.ContactListController' ],
  xtype: 'app-contactList',
  controller: 'contactList',

  items: [{
    cls: 'contact-list',
    xtype: 'grid',
    reference: 'contactListGrid',
    scrollable: true,
    autoScroll: true,
    plugins: [{
      ptype: 'rowediting',
      clicksToMoveEditor: 1,
      autoCancel: false
    }],
    listeners: {
      selectionchange: 'onSelectionChange'
    },
    flex:1,
    store: 'contactList',
    pageSize: 10,
    title: 'Company Directory',
    columns: {
      defaults: {
        editor: {
          xtype: 'textfield',
          allowBlank: false
        }
      },
      items: [
      {
        text: 'First Name',
        width: 100,
        dataIndex: 'fname'
      },
      {
        text: 'Email',
        width: 250,
        dataIndex: 'email',
        editor: { vtype: 'email' }
      },
      // …Code truncated.
      ]
    },
    dockedItems: [{
      xtype: 'pagingtoolbar',
      store: 'contactList',
      dock: 'bottom',
      displayInfo: true
    }, {
      xtype: 'toolbar',
      dock: 'top',
      ui: 'footer',
      defaults: { cls: 'btn-orange' },
      items: ['->', {
        text: 'Remove',
        disabled: true,
        reference: 'btnRemoveContact',

        listeners: {
          click: 'onRemove'
        },
      },
      // …Code truncated
      ]
    }]
  } ]
});
```

In the preceding code, the grid uses two toolbars: one for the paging toolbar and one for add/remove/save buttons. These toolbars are docked to the grid using the `dockedItems` config, as shown in the following code.

You learned about this earlier in this chapter. `'dockedItems'` is a property of panel; it allows you to dock a set of components to either left, top, right, or bottom. You can dock any component you want, although typically it is used for the toolbar.

> The paging toolbar needs the store in order to properly display the number of pages, page size, and so on.

```
    dockedItems: [{
      xtype: 'pagingtoolbar',
      store: 'contactList',
      dock: 'bottom',
  displayInfo: true
}, {
  xtype: 'toolbar',
  dock: 'top',

  ui: 'footer', //This sets style to the component. The 'ui' is a
property of the component. The default value of this property for
all the component is 'default'.  For details are given in the
chapter that focus on theming and styling.

  defaults: { cls: 'btn-orange' },
  items: ['->', {
    text: 'Remove',

    disabled: true,  //We set disabled by default, and this will
be enabled when a row in the grid is selected. Check the
onSelectionChange method in the controller.
    reference: 'btnRemoveContact',
    listeners: { click: 'onRemove' },
  },
  // …Code truncated
  ]
}]
```

The `ViewController` code is very simple. It just handles the add, remove, and selection change events of the `ContactList` view.

See how the references set in the view are accessed in the controller. For example, the following line returns the reference to the grid:

```
  var grid = this.lookupReference('contactListGrid');
```

Here, `contactListGrid` is marked as a reference in the preceding view:

In the following code, the store is accessed using `grid.getStore();` it's also possible to access the store using `Ext.getStore(contactList)`:

```
  Ext.define('CD.view.contactList.ContactListController', {
    extend: 'Ext.app.ViewController',

    alias: 'controller.contactList',
    views: ['CD.view.contactList.ContactList'],
    requires: ['CD.store.ContactList'],

    onSave: function() {
      //Note, this will trigger one or more calls to the server
  based on the number of operations performed in the store.
      Ext.getStore('contactList').save();
    },

    onSelectionChange: function() {
      this.lookupReference('btnRemoveContact').enable();
    },

    onRemove: function() {
      var grid = this.lookupReference('contactListGrid');
      var sm = grid.getSelectionModel();

      //This line cancels the row/cell edit if it is active before
  we remove the item.
      grid.plugins[0].cancelEdit();

      grid.getStore().remove(sm.getSelection());
      if (grid.getStore().getCount() > 0) {
        sm.select(0);
      }
    },

    onCreate: function() {
      var grid = this.lookupReference('contactListGrid');
      grid.plugins[0].cancelEdit();

      // Create a model instance
      var r = Ext.create('Contact');
      grid.getStore().insert(0, r);
      grid.plugins[0].startEdit(0, 0);
    }
  });
```

The code for model and view is shown as follows. The `rootProperty` field lets the store know where the data records are located in the JSON response of the API.

```
Ext.define('Contact', {
  extend: 'Ext.data.Model',
  fields: ['fname', 'lname', 'email', 'address','city',
'state','phone','type']
});

Ext.define('CD.store.ContactList', {
  extend: 'Ext.data.Store',
  storeId: 'contactList',
  model: 'Contact',
  pageSize: 10,
  proxy: {
    type: 'rest',
    url: 'contactlist',
    reader: {
      type: 'json',
      rootProperty: 'data',
      totalProperty: 'total'
    }
  }
});

Ext.create('CD.store.ContactList').load();
```

As the pagination is used, the `totalProperty` field lets the store know the total number of records available in the remote server. The grid's pagination control in the view uses this values to display pagination information like "`Displaying 1 to 10 of 50`"; here, `50` is the value of `totalProperty`. Also, based on the value of this `totalProperty`, the grid knows when to disable the next button, which is used to navigate to the next page of data.

When you don't specify the type of the model field, the default type: auto will be used. If needed, you can specify the type and all the validation rules.

I have used Go to write the REST API for this sample project. The code for HTML, CSS, the REST API, and so on are not included in this book. The complete source code for this project is available at `https://github.com/ananddayalan/extjs-by-example-company-directory`.

Summary

In this chapter, we explored the grids in detail; lots of options and configurations are available in the grid, filtering, sorting, and grouping capabilities of the grid. You also learned how to use different plugins in the grid. We saw how the pivot grid can be used like a reporting tool. Later, we created a sample project called Company Directory with the grid control.

6
Advanced Components

This chapter covers advanced components, such as trees and data views, in Ext JS. It will present to readers a sample project called picture explorer, which is built with trees and data view components. The following topics will be covered:

- Trees
- Data views
- Drag and drop
- The picture explorer — a sample project

The main goal of this chapter is to explore the tree panel and data views and use them to build a sample project called picture explorer. The design of picture explorer is shown in *Figure 6.1*.

The most important components of this project are the tree panel and data views. Some of the concepts and components used in this project are:

- The tree panel
- Data views
- Model
- The store and rest proxy
- Layouts and containers
- References
- Event handling
- Filtering

You already learned about all the concepts mentioned in the preceding list in the previous chapters, apart from the tree panel and data views. So, in this chapter, let's first learn about the tree panel and data views.

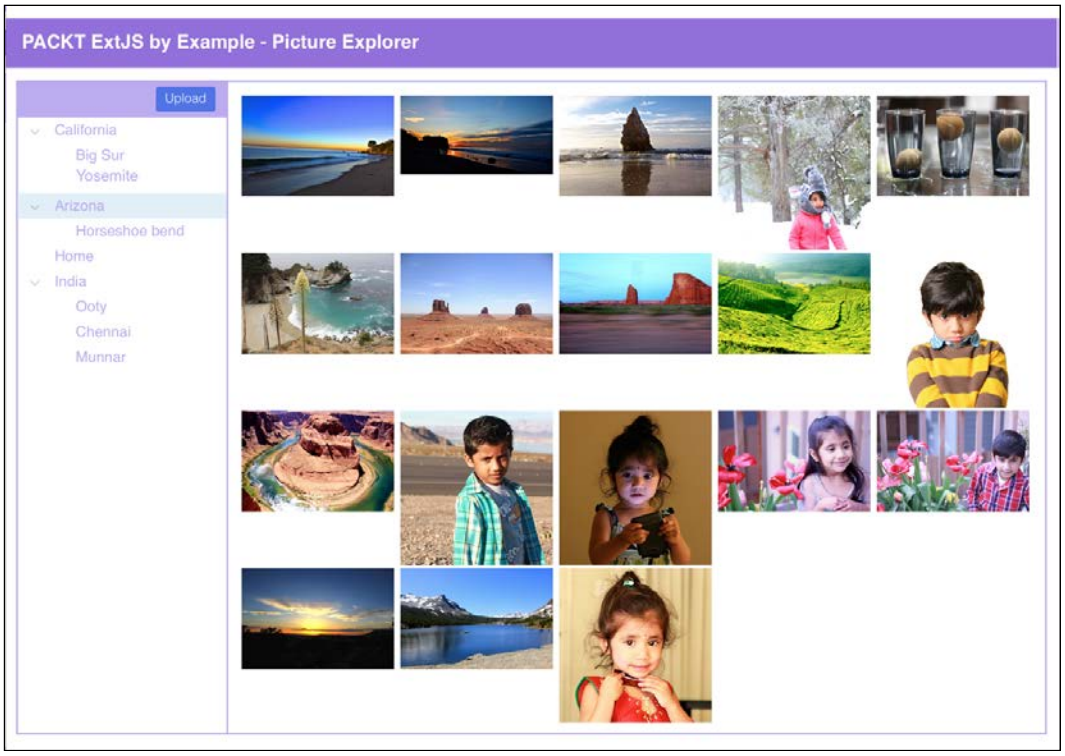


Figure 6.1: Picture Explorer

The tree panel

This is a powerful component in Ext JS with many options and configurations you can use to build any kind of tree. A tree panel represents a hierarchical data in the UI in the tree structure.

Similar to `Ext.grid.Panel`, `Ext.tree.Panel` also inherits from `Ext.panel.Table`. So, it also supports multiple columns.

Unlike the grid panel, the tree panel requires a tree store (`Ext.Store.TreeStore`). The tree store has some special properties and functionalities required for the tree panel.

The basic tree

Let's start with a simple basic tree. At the minimum, the tree panel needs a tree store to get the data. So, let's first create a simple tree store with the inline hardcoded data:

```
var store = Ext.create('Ext.data.TreeStore', {
  root: {
    expanded: true,
    text: 'Continents',

    children: [{
      text: 'Antarctica',
      leaf: true
    }, {
      text: 'South America',
      expanded: true,
      children: [{
        text: 'Brazil',
        leaf: true
      }, {
        text: 'Chile',
        leaf: true
      }]
    }, {
      text: 'Asia',
      expanded: true,
      children: [{
        text: 'India',
        leaf: true
      },{
        text: 'China',
        leaf: true
      }]
    }, {
      text: 'Africa',
      leaf: true
    } ]
  }
});
```

Now, let's create the tree using `Ext.tree.Panel`, and use the tree store created before:

```
Ext.create('Ext.tree.Panel', {
  title: 'Basic Tree',
  width: 200,
  height: 450,
  store: store,
  rootVisible: true,
  renderTo: Ext.getBody()
});
```

The output of the preceding code is shown in the following screenshot:

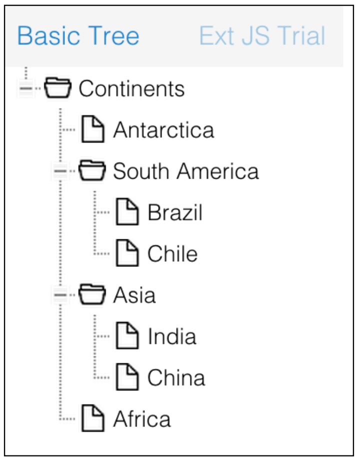


Figure 6.2

Now, let's create an advanced tree that is capable of drag and drop. Also, let's use some additional options available in the tree panel and the tree store. Just by adding a plugin called `treeviewdragdrop`, you'll be able to drag and drop the tree nodes in the tree, as shown in the following code:

```
  var store = Ext.create('Ext.data.TreeStore', {
    root: {
      expanded: true,
      text: 'Continents',

      checked: false,
      children: [{
        text: 'Antarctica',
        leaf: true ,
        checked: false
      },{
        text: 'South America',
        expanded: false,
        checked: true,
        children: [{
          text: 'Chile',
          leaf: true,
          checked: true
        }]
      },{
        text: 'Asia',
        expanded: true,
        checked: true,
        children: [{
          text: 'India',
          leaf: true,
          checked: true
        },{
          text: 'China',
          leaf: true,
          checked: true
        }]
      },{
        text: 'Africa',
        leaf: true,
        checked: true
      }]
    }
  });

  Ext.create('Ext.tree.Panel', {
    title: 'Basic Tree',
    width: 200,
    height: 450,
    store: store,
    rootVisible: true,
    useArrows: true,
    lines: false,
    renderTo: Ext.getBody(),
    viewConfig: {
      plugins: {
        ptype: 'treeviewdragdrop',
        containerScroll: true
      }
    }
  });
```

The output is shown in the following screenshot. Here, I have dragged and dropped the node **South America** under the node **Asia**:

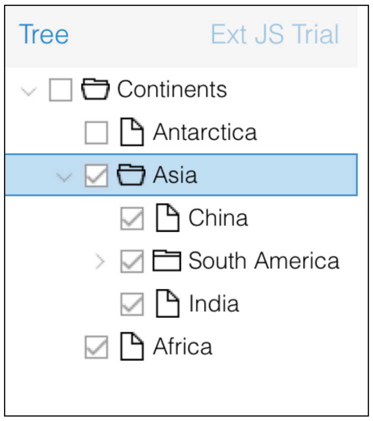


Figure 6.3

The tree grid

You can add multiple columns to the tree and create a tree grid. By default, the tree contains one column, which uses the text field of the tree store's nodes.

In the store, you can see that there are new fields added to each node. These will be used in the tree panel columns. The tree grid's features, such as column resizing, renderers, sorting, filtering, and so on, are available in the tree panel as well. Take a look at the following code:

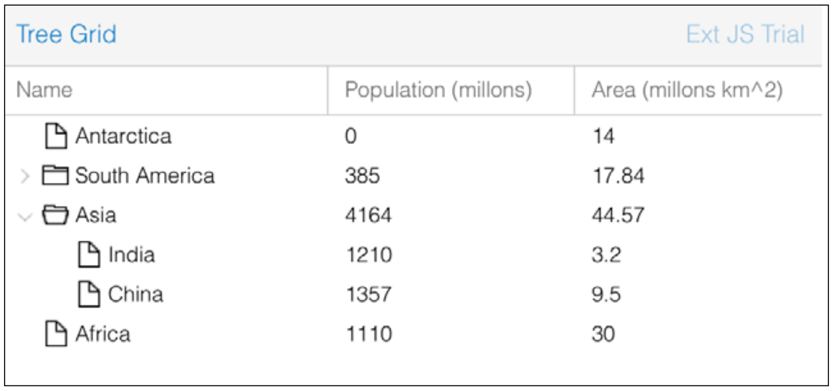
```
var store = Ext.create('Ext.data.TreeStore', {
  root: {
    expanded: true,
    text: 'Continents',
    children: [{
      name: 'Antarctica',
      population: 0,
      area: 14,
      leaf: true
    },{
      name: 'South America',
      population: 385 ,
      area: 17.84,
      expanded: false,
      children: [{
        name: 'Chile',
        population: 18,
        area: 0.7,
        leaf: true,
      }]
    },{
      name: 'Asia',
      expanded: true,
      population: 4164,
      area: 44.57,
      children: [{
        name: 'India',
        leaf: true,
        population: 1210,
        area: 3.2
      },{
        name: 'China',
        leaf: true,
        population: 1357,
        area: 9.5
      }
    ]},
    {
      name: 'Africa',
      leaf: true,
      population: 1110,
      area: 30
    }
  ]}
});
```

The following grid is almost the same as the previous tree panel, but we have added multiple columns here. The xtype `treecolumn` provides the indentation and the folder structure. Like a normal grid, the tree grid columns can be of any type; you can add the checkbox, picture, button, URL, and so on.

By default, the column size is resizable, but you can have a fixed `width` if required. Take a look at the following code:

```
    Ext.create('Ext.tree.Panel', {
      title: 'Tree Grid',
      width: 500,
      height: 450,
      store: store,
      rootVisible: false,
      useArrows: true,
      lines: false,
      scope: this,
      renderTo: Ext.getBody(),
      columns: [{
        xtype: 'treecolumn',
        text: 'Name',
        flex: 1,
        sortable: true,
        dataIndex: 'name'
      },{

        text: 'Population (millons)',
        sortable: true,
        width: 150,
        dataIndex: 'population'
      },{

        text: 'Area (millons km^2)',
        width: 150,
        sortable: true,
        dataIndex: 'area'
      }],
    });
```

The **Tree Grid** output for the preceding code is shown here:



Name	Population (millons)	Area (millons km^2)
Antarctica	0	14
South America	385	17.84
Asia	4164	44.57
India	1210	3.2
China	1357	9.5
Africa	1110	30

Figure 6.4

Data views

`Ext.view.View` (`xtype:dataview`) is used to display the data with a custom template. You need to provide the custom template and the data store so that it can be used in the data view. `Ext.XTemplate` should be used for the templating.

The data view provides built-in events, such as click, double-click, mouseover, mouseout, and so on, for the contained items.

Let's first create a simple model called `Person` and a store to hold a list of persons, as shown in the following code:

```
Ext.define('Person', {
  extend: 'Ext.data.Model',
  fields: [
  { name:'name', type:'string' },
  { name:'age', type:'int' },
  { name:'gender', type:'int' }
  ]
});


Ext.create('Ext.data.Store', {
  id:'employees',
  model: 'Person',
  data: [
  { name:'Mike', age:22, gender: 0 },
  { name:'Woo', age:32, gender: 1  },
  { name:'John', age:33, gender: 1  },
  { name:'Kalai', age:25, gender: 0  }
  ]
});
```

Then, we have to create the template. The following template uses the HTML table element to render the data in the custom format.

To bind a field of the model, you can simply mention the field name within the curly braces, for example, `{fieldname}`.

`XTemplate` supports conditional rendering with if statements, as shown in the following code:

```
var empTpl = new Ext.XTemplate(
  '<tpl for=".">',
    '<div style="margin-bottom: 10px;" class="data-view">',
    '<table style="width:100%">',
      '<tr>',
        '<td style="font-size: 100px;width:100px;" rowspan="3"><i
class="fa fa-user"></i></td>',
        '<td>Name: {name}</td>',
      '</tr>',
      '<tr>',
        '<td>Age:{age}</td>',
      '</tr>',
      '<tr>',
        '<td>Gender: <tpl if="gender == 1">',
          '<i class="fa fa-mars"></i>',
          '<tpl else>',
          '<i class="fa fa-venus"></i>',
          '</tpl>
        </td>',
      '</tr>',
    '</div>',
  '</tpl>'
);
```

For the preceding example, I have used the awesome font to render the icon. You need to add the following code in your HTML file in order to get the person icon:

```
  <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/font-
awesome/4.3.0/css/font-awesome.min.css">
```

The following code creates a data view, and it specifies the store, template, and `itemSelector`:

```
Ext.create('Ext.view.View', {
  store: Ext.getStore('employees'),

  tpl: empTpl,
  itemSelector: 'div.data-view',

  renderTo: Ext.getBody(),
  listeners:{
    itemclick: function(node, rec, item, index, e) {
      alert(rec.data.name);
    }
  }
});
```

Here, `itemSelector` is a required simple CSS selector. This is used to determine which nodes this `DataView` will be working with. The `itemSelector` is used to map DOM nodes to records. There should only be one root-level element that matches the selector for each record.

You can listen to events, such as click, double-click, and so on, by adding the listeners, as shown in the preceding code. The output is as follows:

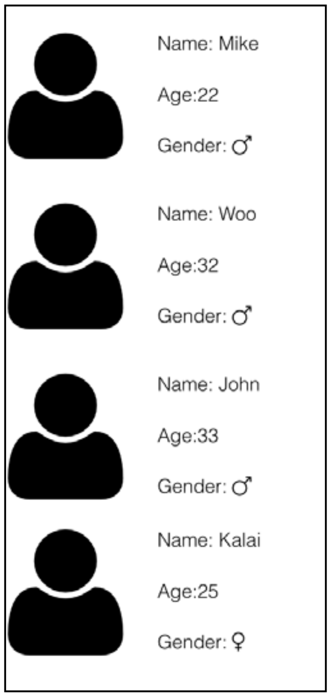


Figure 6.5

The picture explorer – a sample project

Now, we will create a sample project using the tree panel and data views, which you learned in this chapter. Here is the design of the sample project:

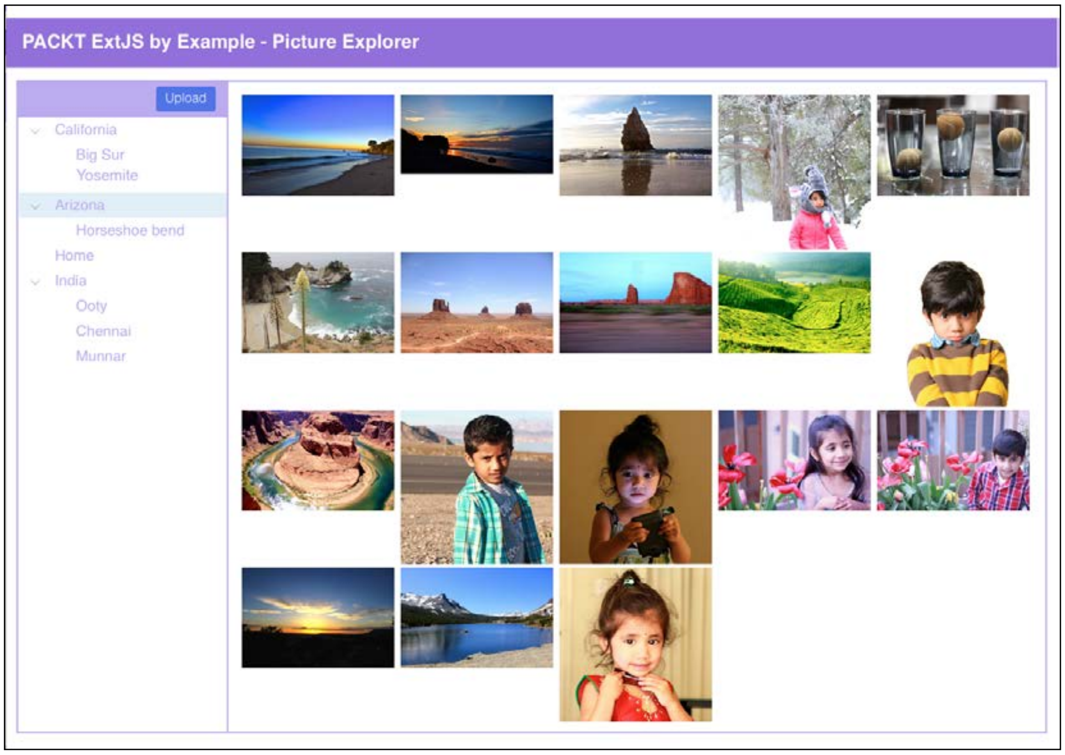


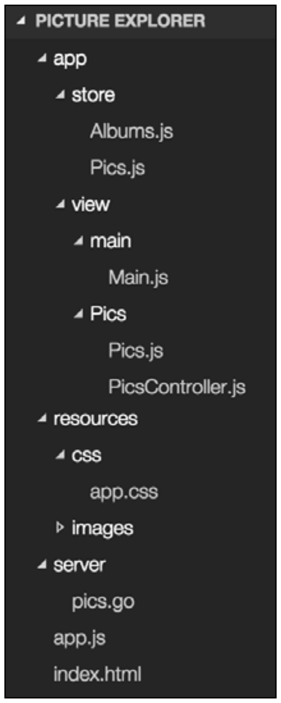
Figure 6.6

By looking at the design, you know that the most important components of this project are the tree panel and data views. Some of the concepts and components used in this project are already mentioned earlier in this chapter.

At this point, you have learned about all the concepts. Let's take a look at some of the important files and folder structure in this project.

> The complete source code is not provided in this book. The complete source code for this project is available at `https://github.com/ananddayalan/extjs-by-example-picture-explorer`.

The following is the folder structure of this sample project:



```
PICTURE EXPLORER
  app
    store
      Albums.js
      Pics.js
    view
      main
        Main.js
      Pics
        Pics.js
        PicsController.js
  resources
    css
      app.css
    images
  server
    pics.go
  app.js
  index.html
```

Figure 6.7

The following view code is the important part of this project. This view renders the most viewable part of the application. It uses the tree panel and data views:

```
Ext.define('PE.view.pics.Pics', {
  extend: 'Ext.panel.Panel',

  /* Marks these are required classes to be to loaded before
loading this view */
  requires: [ 'PE.view.pics.PicsController' ],

  xtype: 'app-pics',
  controller: 'pics',

  items: [{
    xtype: 'container',
    layout: 'hbox',

    cls: 'pics-list',
    items: [{
      xtype: 'treepanel',
      width: 200,

      height: '100%',
      store: 'albums',
      border: true,
      useArrows: true,
      cls: 'tree',
      rootVisible: false,
      listeners:{
        itemclick: 'onNodeSelect'
      },
      dockedItems: [{
        xtype: 'toolbar',
        dock: 'top',
        ui: 'footer',
        items: [
        { xtype: 'component', flex: 1 },
        { xtype: 'button', text: 'Upload', cls: 'btn-blue' }
        ]
      }]

    },
    {
      xtype:'dataview',
      reference: 'picsList',
      cls:'pics-list-content',
      store: 'pics',
      tpl: [
      '<tpl for=".">',
      '<div class="thumb"><img  src="{url}" title=""></div>',
      '</tpl>'
      ],
      multiSelect: true,
      minHeight: 400,
      flex:1,
      trackOver: true,
      overItemCls: 'x-item-over',
      itemSelector: 'div.thumb',
      emptyText: 'No images to display'
    }
    ]
  }]
});
```

The `ViewController` handles the `itemdblclick` event of the tree panel in order to filter the only picture and show the picture in the selected album only.

You can see that the upload button click event is not handled. This is because it's homework for you. Take a look at the following code:

```
Ext.define('PE.view.pics.PicsController', {
  extend: 'Ext.app.ViewController',

  alias: 'controller.pics',
  views: ['PE.view.pics.Pics'],
  requires: ['PE.store.Pics','PE.store.Albums'],

  onNodeSelect: function(node, rec, item, index, e)
  {
    var albums = [];
    albums.push(rec.id);
    rec.childNodes.forEach(function(item){
      albums.push(item.id);
    });

    Ext.getStore('pics').filter({
      property: 'albumId',
      operator: 'in',
      value   : albums
    });
  }
});
```

The code for `Model` and `Store` are shown here. Note that when you don't specify the type of the model field, the default type: `auto` will be used:

```
Ext.define('Pic', {
  extend: 'Ext.data.Model',
  fields: ['id', 'url', 'albumId']
});

Ext.define('PE.store.Pics', {
  extend: 'Ext.data.Store',

  storeId: 'pics',
  model: 'Pic',

  proxy: {
    type: 'rest',
    url: 'pics', // URL that will load data with respect to start
and limit params
```

```
      reader: {
        type: 'json'
      }
    }
  });

  Ext.create('PE.store.Pics').load();

  Ext.define('PE.store.Albums', {
    extend: 'Ext.data.TreeStore',
    storeId: 'albums',
    root: {
      expanded: true,
      children: [{
        id: 100,
        text: ' California',
        expanded: true,
        children: [{
          id: 600,
          text: ' Big Sur',
          leaf: true
        },{
          id: 500,
          text: ' Yosemite',
          leaf: true
        }]
      }, {
        id: 400,
        text: ' Arizona',
        expanded: true,
        children: [{
          id: 300,
          text: ' Horseshoe bend',
          leaf: true
        }]
      }, {
        id: 200,
        text: ' Home',
        leaf: true
      },{
        id: 700,
        text: ' India',
        expanded: true,
        children: [{
          id: 800,
          text: ' Ooty',
          leaf: true
        },{
          id: 900,
          text: ' Chennai',
          leaf: true
        },{
          id: 1000,
          text: ' Munnar',
          leaf: true
        }]
      } ]
    }
  });

  Ext.create('PE.store.Albums');
```

I have used Go to write the REST API for this sample project. The code for HTML, CSS, the REST API, and so on are not included in this book. The complete source code for this project is available at `https://github.com/ananddayalan/extjs-by-example-picture-explorer`.

The picture explorer project is a nice simple project to learn the usage of the tree panel and data views. This project can be further improved by adding a few more features. For example, what about adding the drag and drop feature to drag and drop the pictures from one album to another album? I'll leave that to you as a coding exercise, but here, I'll give you a brief overview of the drag and drop feature in the next section. This will help you to add the drag and drop feature to this sample project.

Drag and drop

Any element or a component can be enabled for drag and drop. Three important things required for drag and drop are:

- Configure the items as draggable
- Create the drop target
- Complete the drop target

Configure the items as draggable

To make an item draggable, you need to create the `Ext.dd.DD` instance for each element that you want to make it draggable.

Check the following sample code that makes all the `div` elements with the `pics` class draggable by creating the instances of `Ext.dd.DD`:

```
// Configure the pics as draggable
var pics = Ext.get('pics').select('div');

Ext.each(pics.elements, function(el) {
  var dd = Ext.create('Ext.dd.DD', el, ' picsDDGroup', {
    isTarget  : false
  });
});
```

Create the drop target

Use `Ext.dd.DDTarget` to create the drop target container. The following code creates the drop target for all the `div` elements with the `album` class:

```
var albums = Ext.get('album').select('div');

Ext.each(albums.elements, function(el) {
  var albumDDTarget = Ext.create('Ext.dd.DDTarget', el,
'picsDDGroup');
});
```

Complete the drop target

When the draggable item is dropped in a drop target, we need to move the item from the source to the drop target container. This is accomplished by overriding the `onDragDrop` method of `DD`. Take a look at the following code:

```
var overrides = {

  onDragDrop : function(evtObj, targetElId) {
    var dropEl = Ext.get(targetElId);

    if (this.el.dom.parentNode.id != targetElId) {
      dropEl.appendChild(this.el);
      this.onDragOut(evtObj, targetElId);
      this.el.dom.style.position ='';
      this.el.dom.style.top = '';
      this.el.dom.style.left = '';
    }
    else {
      this.onInvalidDrop();
```

```
        }
      },
      onInvalidDrop : function() {
        this.invalidDrop = true;
      },
    };
```

This override has to be applied to the instances of the `DD` element. So, we need to add `Ext.apply(dd, overrides)` when you make the pics as draggable, as shown in the following code:

```
    var albums = Ext.get('album').select('div');
    var pics = Ext.get('pics').select('div');

    Ext.each(pics.elements, function(el) {
      var dd = Ext.create('Ext.dd.DD', el, ' picsDDGroup', {
        isTarget  : false
      });
      Ext.apply(dd, overrides);
    });
```

Summary

In this chapter, you learned how to use the drag and drop features of Ext JS. We also looked at a couple of advanced components: the tree panel and data views. Then, we created a sample project called picture explorer with the tree panel and data views.

7
Working with Charts

This chapter explores the different types of chart components in Ext JS and ends with a sample project called expense analyzer. The following topics will be covered:

- Charts types
- Bar and column charts
- Area and line charts
- Pie charts
- 3D charts
- The expense analyzer – a sample project

Charts

In the first chapter, I mentioned that Ext JS is almost like a one-stop shop for all your JavaScript framework needs. Yes, Ext JS also includes charts with all other rich components you learned so far.

Chart types

There are three types of charts: cartesian, polar, and spacefilling.

The cartesian chart

`Ext.chart.CartesianChart (xtype: cartesian or chart)`

A cartesian chart has two directions: *X* and *Y*. By default, *X* is horizontal and *Y* is vertical. Charts that use the cartesian coordinates are column, bar, area, line, and scatter.

The polar chart

`Ext.chart.PolarChart (xtype: polar)`

These charts have two axes: angular and radial. Charts that plot values using the polar coordinates are pie and radar.

The spacefilling chart

`Ext.chart.SpaceFillingChart (xtype: spacefilling)`

These charts fill the complete area of the chart.

Bar and column charts

For bar and column charts, at the minimum, you need to provide a store, axes, and series.

The basic column chart

Let's start with a simple basic column chart. First, let's create a simple tree store with the inline hardcoded data as follows:

```
Ext.define('MyApp.model.Population', {
  extend: 'Ext.data.Model',
  fields: ['year', 'population']
});

Ext.define('MyApp.store.Population', {
  extend: 'Ext.data.Store',
  storeId: 'population',
  model: 'MyApp.model.Population',
  data: [
  { "year": "1610","population": 350 },
  { "year": "1650","population": 50368 },
  { "year": "1700", "population": 250888 },
  { "year": "1750","population": 1170760 },
  { "year": "1800","population": 5308483 },
  { "year": "1900","population": 76212168 },
  { "year": "1950","population": 151325798 },
  { "year": "2000","population": 281421906 },
  { "year": "2010","population": 308745538 },
  ]
});
```

```
var store = Ext.create("MyApp.store.Population");
```

Now, let's create the chart using `Ext.chart.CartesianChart` (xtype: `cartesian` or `chart`) and use the store created above.

```
Ext.create('Ext.Container', {
  renderTo: Ext.getBody(),
  width: 500,
  height: 500,
  layout: 'fit',
  items: [{
    xtype: 'chart',
    insetPadding: { top: 60, bottom: 20, left: 20, right: 40 },
    store: store,
    axes: [{
      type: 'numeric',
      position: 'left',
      grid: true,
      title: { text: 'Population in Millions', fontSize: 16 },
    }, {
      type: 'category',
      title: { text: 'Year', fontSize: 16 },
      position: 'bottom',
    }
    ],
    series: [{
      type: 'bar',
      xField: 'year',
      yField: ['population']
    }],
    sprites: {
      type: 'text',
      text: 'United States Population',
      font: '25px Helvetica',
      width: 120,
      height: 35,
      x: 100,
      y: 40
    }
  } ]
});
```

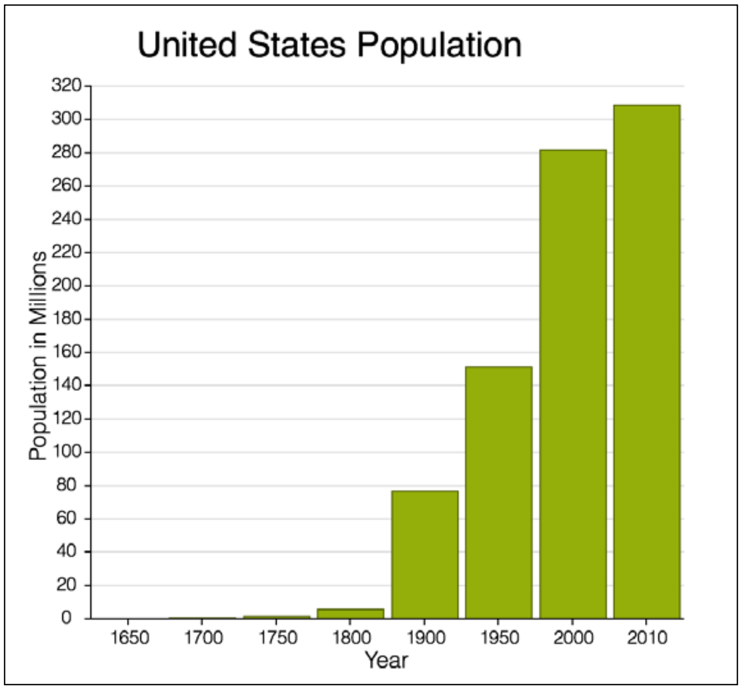
Important things to note in the preceding code are `axes`, `series`, and `sprite`. Axes can be of one of the three types: numeric, time, and category.

In series, you can see that the type is set to `bar`. In Ext JS, to render the column or bar chart, you have to specify the type as `bar`, but if you want a bar chart, you have to set `flipXY` to `true` in the chart config.

The sprites config used here is quite straightforward. Sprites is optional, not a must. The grid property can be specified for both the axes, although we have specified it only for one axis here.

The `insetPadding` is used to specify the padding for the chart to render other information, such as the title. If we don't specify `insetPadding`, the title and other information may get overlapped with the chart.

The output of the preceding code is shown here:

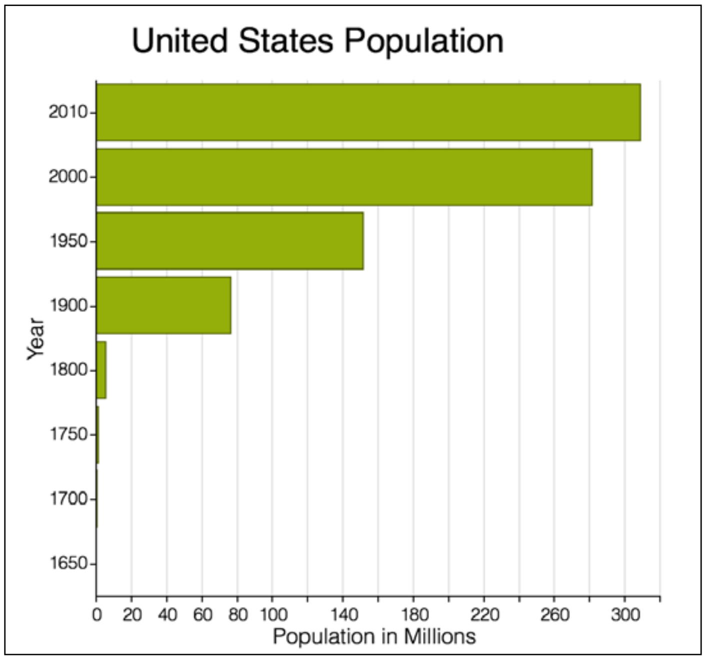


The bar chart

As mentioned before, in order to get the bar chart, you can just use the same code, but specify `flipXP` to `true` and change the positions of axes accordingly, as shown in the following code:

```
Ext.create('Ext.Container', {
  renderTo: Ext.getBody(),
  width: 500,
  height: 500,
  layout: 'fit',
  items: [{
    xtype: 'chart',
    flipXY: true,
    insetPadding: { top: 60, bottom: 20, left: 20, right: 40 },
    store: store,
    axes: [{
      type: 'numeric',
      position: 'bottom',
      grid: true,
      title: { text: 'Population in Millions', fontSize: 16 },
    }, {
      type: 'category',
      title: { text: 'Year', fontSize: 16 },
      position: 'left',
    }
    ],
    series: [{
      type: 'bar',
      xField: 'year',
      yField: ['population']
    }
    ],
    sprites: {
      type: 'text',
      text: 'United States Population',
      font: '25px Helvetica',
      width: 120,
      height: 35,
      x: 100,
      y: 40
    }
  } ]
});
```

The output of the preceding code is shown in the following screenshot:



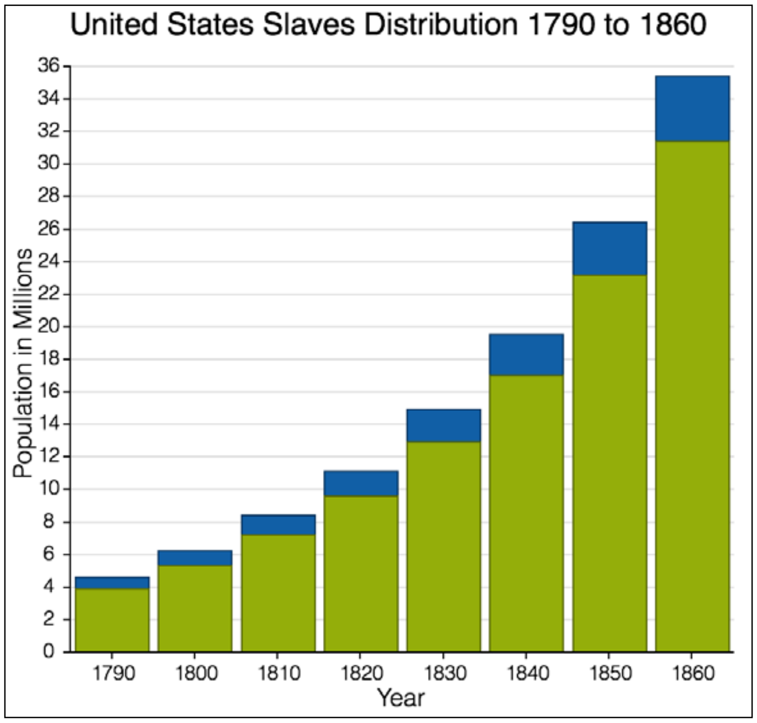
The stacked chart

Now, let's say you want to plot two values in each category in the column chart. You can either stack them or have two bar columns for each category.

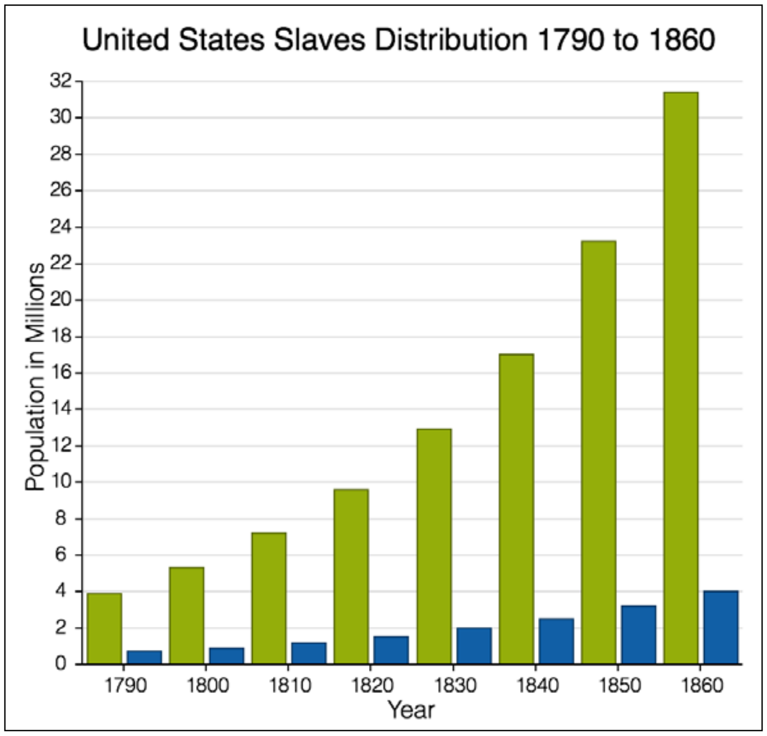
Let's modify our column chart example to render a stacked column chart. For this, we need an additional numeric field in the store, and we need to specify two fields for `yField` in the series. You can stack more than two fields, but for this example, we will stack only two fields. Take a look at the following code:

```
Ext.define('MyApp.model.Population', {
  extend: 'Ext.data.Model',
  fields: ['year', 'total','slaves']
});

Ext.define('MyApp.store.Population', {
  extend: 'Ext.data.Store',
  storeId: 'population',
  model: 'MyApp.model.Population',
  data: [
  { "year": "1790", "total": 3.9, "slaves": 0.7 },
  { "year": "1800", "total": 5.3, "slaves": 0.9 },
  { "year": "1810", "total": 7.2, "slaves": 1.2 },
  { "year": "1820", "total": 9.6, "slaves": 1.5 },
  { "year": "1830", "total": 12.9, "slaves": 2 },
  { "year": "1840", "total": 17, "slaves": 2.5 },
  { "year": "1850", "total": 23.2, "slaves": 3.2 },
  { "year": "1860", "total": 31.4, "slaves": 4 },
  ]
});
var store = Ext.create("MyApp.store.Population");

Ext.create('Ext.Container', {
  renderTo: Ext.getBody(),
  width: 500,
  height: 500,
  layout: 'fit',
  items: [{
    xtype: 'cartesian',
    store: store,
    insetPadding: { top: 60, bottom: 20, left: 20, right: 40 },
    axes: [{
      type: 'numeric',
      position: 'left',
      grid: true,
      title: { text: 'Population in Millions', fontSize: 16 },
    }, {
      type: 'category',
      title: { text: 'Year', fontSize: 16 },
      position: 'bottom',
    }
    ],
    series: [{
      type: 'bar',
      xField: 'year',
      yField: ['total','slaves']
    }
    ],
    sprites: {
      type: 'text',
      text: 'United States Slaves Distribution 1790 to 1860',
      font: '20px Helvetica',
      width: 120,
      height: 35,
      x: 60,
      y: 40
    }
  }
  ]
});
```

The output of the stacked column chart is shown here:



If you want to render multiple fields without stacking, then you can simply set the stacked property of the series to false to get the following output:



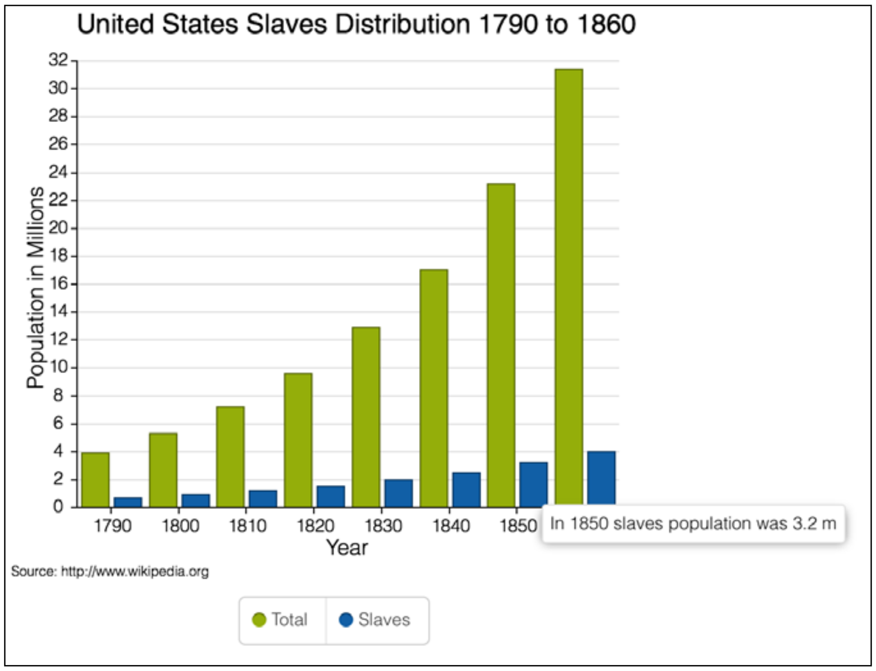
There are so many options available in the chart. Let's take a look at some of the commonly used options:

- `tooltip`: This can be added easily by setting a tooltip property in the series
- `legend`: This can be rendered to any of the four sides of the chart by specifying the legend config
- `sprites`: This can be an array if you want to specify multiple information, such as header, footer, and so on

Here is the code for the same store configured with some advanced options:

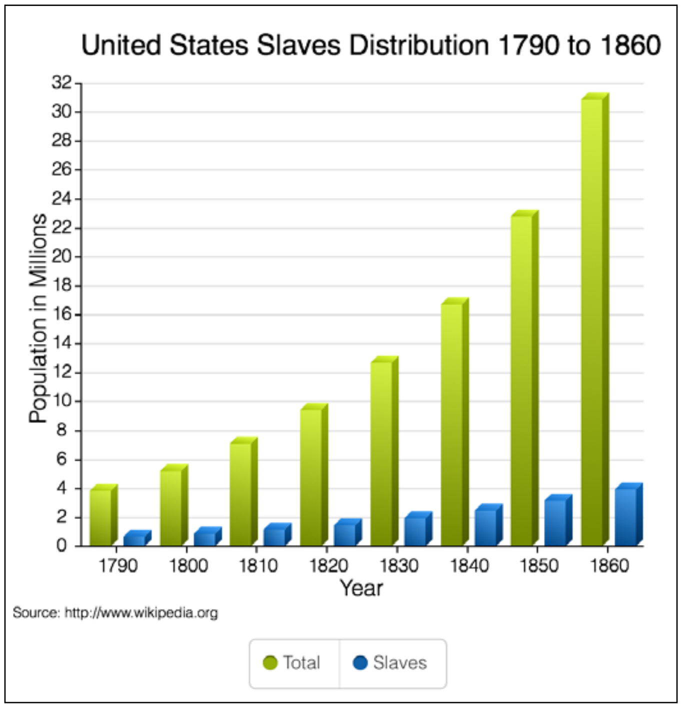
```
Ext.create('Ext.Container', {
  renderTo: Ext.getBody(),
  width: 500,
  height: 500,
  layout: 'fit',
  items: [{
    xtype: 'chart',
    legend: { docked: 'bottom' },
    insetPadding: { top: 60, bottom: 20, left: 20, right: 40 },
    store: store,
    axes: [{
      type: 'numeric',
      position: 'left',
      grid: true,
      title: { text: 'Population in Millions', fontSize: 16 },
      minimum: 0,

    }, {
      type: 'category',
      title: { text: 'Year', fontSize: 16 },
      position: 'bottom',
    }
    ],
    series: [{
      type: 'bar',
      xField: 'year',
      stacked: false,
      title: ['Total', 'Slaves'],
      yField: ['total', 'slaves'],
      tooltip: {
        trackMouse: true,
        style: 'background: #fff',
        renderer: function (storeItem, item) {
          this.setHtml('In ' + storeItem.get('year') + ' ' + item.
field + ' population was ' + storeItem.get(item.field) + ' m');
        }
      }
    ],
    sprites: [{
      type: 'text',
      text: 'United States Slaves Distribution 1790 to 1860',
      font: '20px Helvetica',
      width: 120,
      height: 35,
      x: 60,
      y: 40
    },
    {
      type: 'text',
      text: 'Source: http://www.wikipedia.org',
      fontSize: 10,
      x: 12,
      y: 440
    }]
  }]
});
```

The output with tooltip, legend, and footer is shown here:



The 3D bar chart

If you simply change the type of the series to 3D bar instead of bar, you'll get the 3D column chart, as show in the following screenshot:



Area and line charts

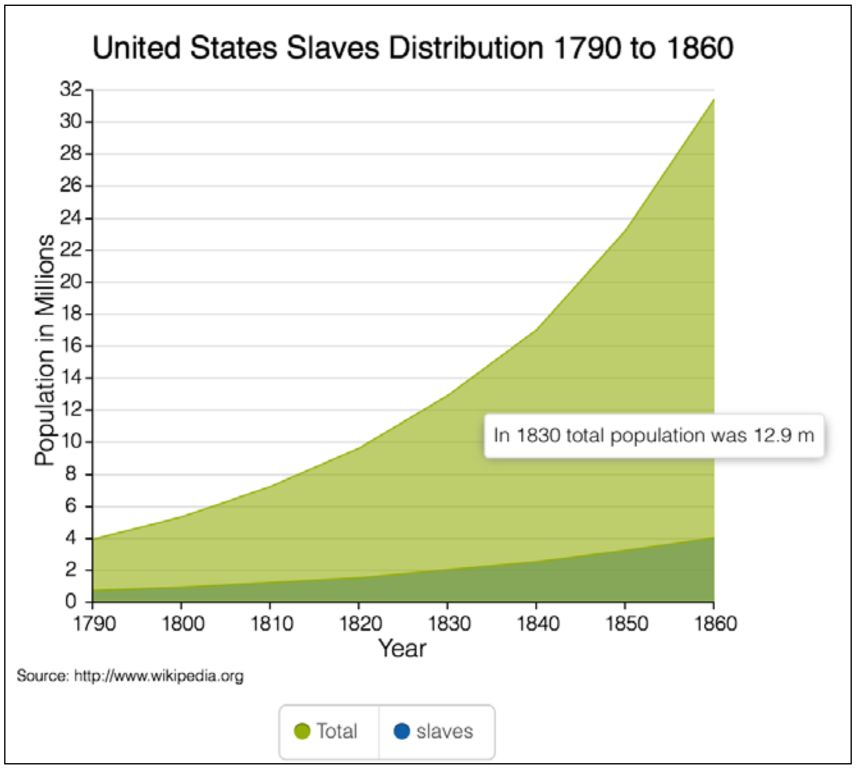
Area and line charts are also cartesian charts.

The area chart

To render an area chart, simply replace the series in the previous example with the following code:

```
series: [{
  type: 'area',
  xField: 'year',
  stacked: false,
  title: ['Total','slaves'],
  yField: ['total', 'slaves'],
  style: {
    stroke: "#94ae0a",
    fillOpacity: 0.6,
  }
}]
```

The output of the preceding code is shown here:



Similar to the stacked column chart, you can have the stacked area chart as well by setting stacked to true in the series. If you set stacked to true in the preceding example, you'll get the following output:

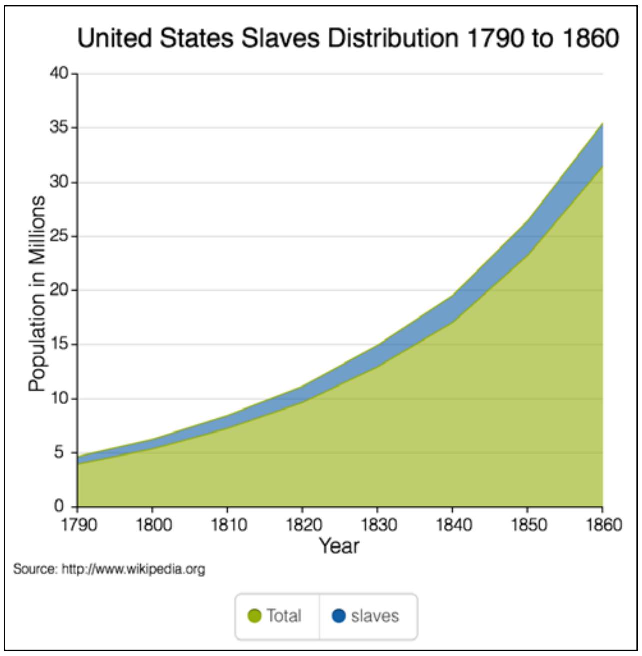
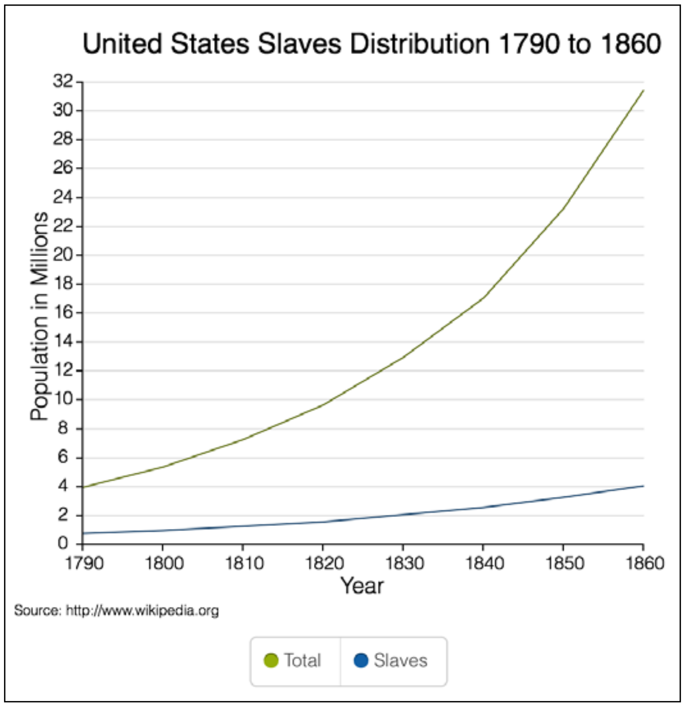


Figure 7.1

The line chart

To get the line chart shown in *Figure 7.1*, use the following series config in the preceding example instead:

```
series: [{
  type: 'line',
  xField: 'year',
  title: ['Total'],
  yField: ['total']

},
{
  type: 'line',
  xField: 'year',
  title: ['Slaves'],
  yField: ['slaves']
}],
```



The pie chart

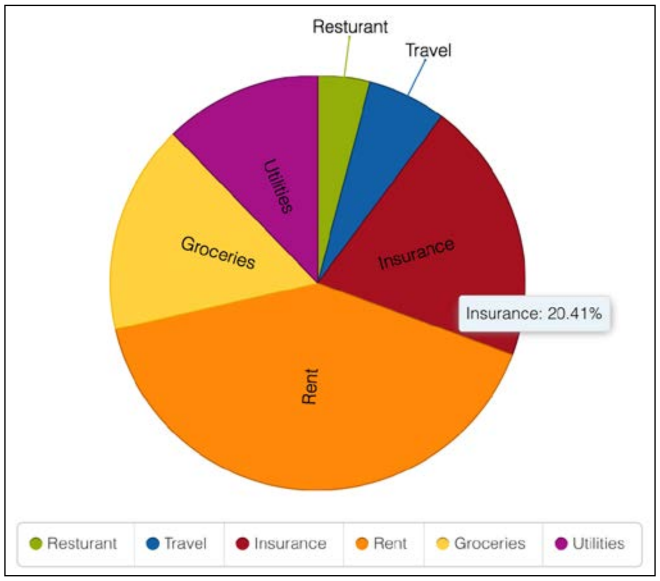
This is one of the frequently used charts in many applications and reporting tools. `Ext.chart.PolarChart (xtype: polar)` should be used to render a pie chart.

The basic pie chart

Specify the type as pie, and specify the `angleField` and label to render a basic pie chart, as as shown in the following code:

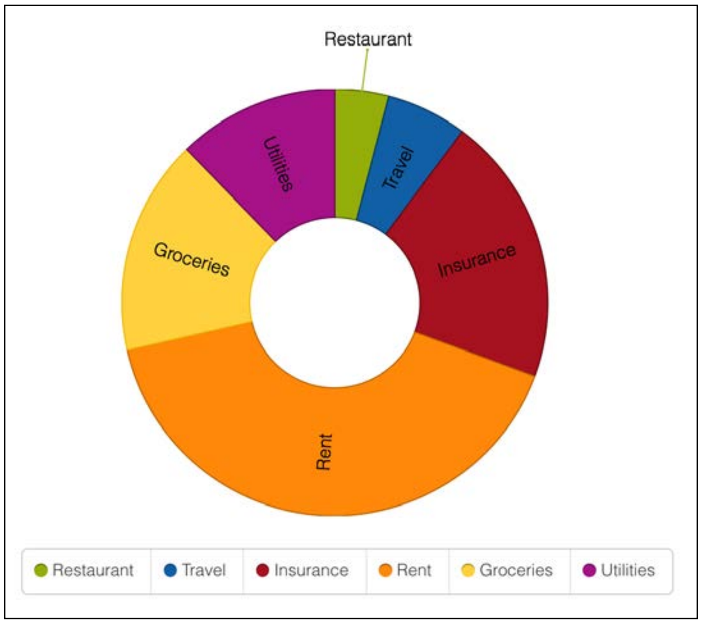
```
Ext.define('MyApp.store.Expense', {
  extend: 'Ext.data.Store',
  alias: 'store.expense',
  fields: [ 'cat', 'spent'],
  data: [
  { "cat": "Restaurant", "spent": 100},
  { "cat": "Travel", "spent": 150},
  { "cat": "Insurance", "spent": 500},
  { "cat": "Rent", "spent": 1000},
  { "cat": "Groceries", "spent": 400},
  { "cat": "Utilities", "spent": 300},
  ]
});
```

```
var store = Ext.create("MyApp.store.Expense");

Ext.create('Ext.Container', {
  renderTo: Ext.getBody(),
  width: 600,
  height: 500,
  layout: 'fit',
  items: [{
    xtype: 'polar',
    legend: { docked: 'bottom' },
    insetPadding: { top: 100, bottom: 20, left: 20, right: 40 },
    store: store,
    series: [{
      type: 'pie',
      angleField: 'spent',
      label: {
        field: 'cat',
      },
      tooltip: {
        trackMouse: true,
        renderer: function (storeItem, item) {
          var value = ((parseFloat(storeItem.get('spent') /
storeItem.store.sum('spent')) * 100.0).toFixed(2));
          this.setHtml(storeItem.get('cat') + ': ' + value + '%');
        }
      }
    }]
  }]
});
```



The donut chart

Just by setting the donut property of the series in the preceding example to `40`, you'll get the following chart. Here, donut is the percentage of the radius of the hole compared to the entire disk:



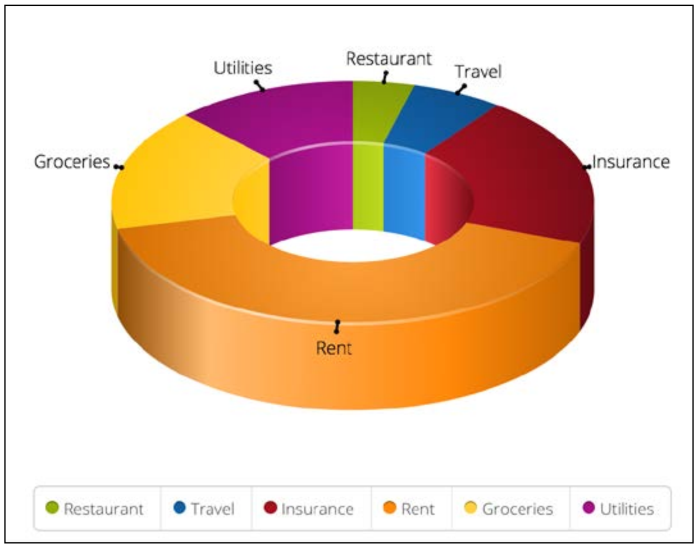
The 3D pie chart

In Ext JS 6, there were some improvements made to the 3D pie chart. The 3D pie chart in Ext JS 6 now supports the label and configurable 3D aspects, such as thickness, distortion, and so on.

Let's use the same model and store that was used in the pie chart example and create a 3D pie chart as follows:

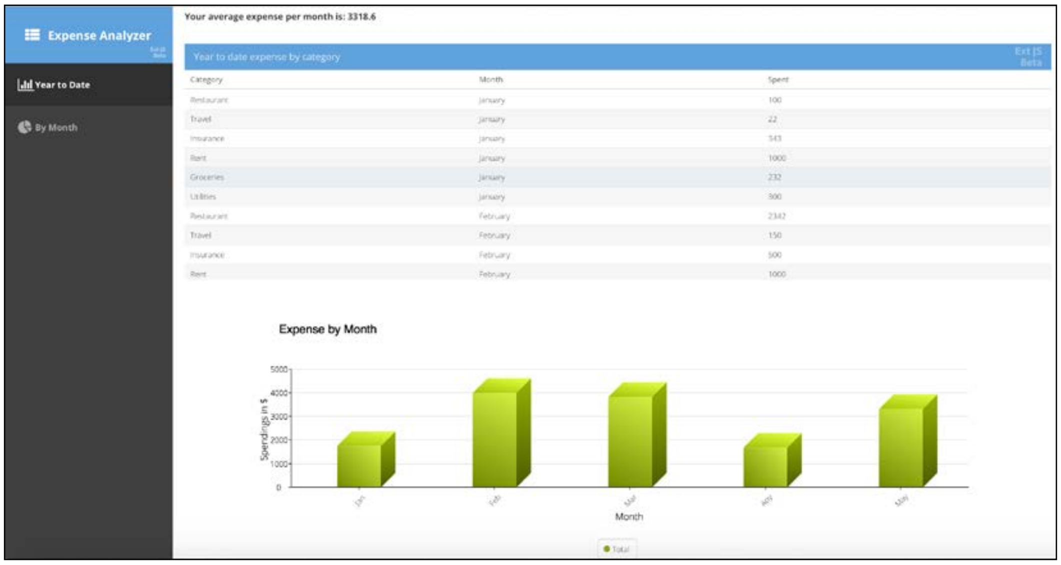
```
Ext.create('Ext.Container', {
  renderTo: Ext.getBody(),
  width: 600,
  height: 500,
  layout: 'fit',
  items: [{
    xtype: 'polar',
    legend: { docked: 'bottom' },
    insetPadding: { top: 100, bottom: 20, left: 80, right: 80 },
    store: store,
    series: [{
      type: 'pie3d',
      donut: 50,
      thickness: 70,
      distortion: 0.5,
      angleField: 'spent',
      label: { field: 'cat' },
      tooltip: {
        trackMouse: true,
        renderer: function (storeItem, item) {
          var value = ((parseFloat(storeItem.get('spent') /
storeItem.store.sum('spent')) * 100.0).toFixed(2));
          this.setHtml(storeItem.get('cat') + ': ' + value + '%');
        }
      }
    }]
  }]
});
```

The following image shows the output of the preceding code:



The expense analyzer – a sample project

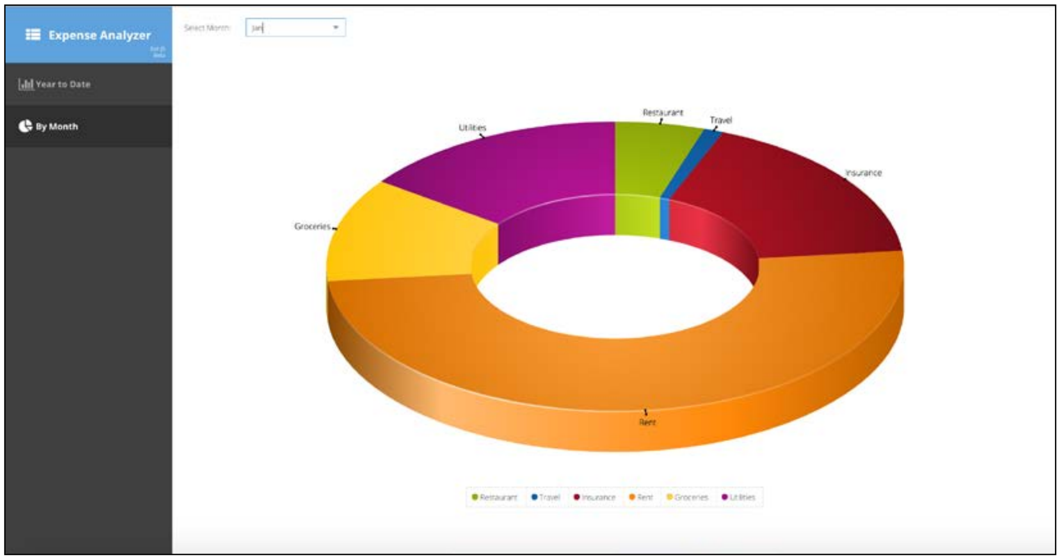
Now that you have learned the different kinds of charts available in Ext JS, let's use them to create a sample project called **Expense Analyzer**. The following screenshot shows the design of this sample project:



Your average expense per month is: 3318.6

Year to date expense by category

Category	Month	Spent
Restaurant	January	100
Travel	January	22
Insurance	January	343
Rent	January	1000
Groceries	January	232
Utilities	January	300
Restaurant	February	2342
Travel	February	150
Insurance	February	500
Rent	February	1000

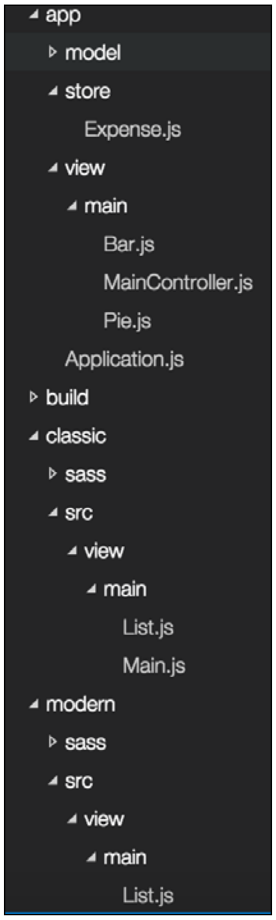


Let's use Sencha Cmd to scaffold our application. Run the following command in the terminal or command window:

```
sencha -sdk <path to SDK>/ext-6.0.0.415/ generate app EA ./expense-
analyzer
```

Now, let's remove all the unwanted files and code and add some additional files to create this project. The final folder structure and some of the important files are shown in the following *Figure 7.2*:

The complete source code is not given in this book. Here, only some of the important files are shown. In between, some less important code has been truncated. The complete source is available at `https://github.com/ananddayalan/extjs-by-example-expense-analyzer`.



```
◢ app
  ▷ model
  ◢ store
      Expense.js
  ◢ view
    ◢ main
        Bar.js
        MainController.js
        Pie.js
    Application.js
▷ build
◢ classic
  ▷ sass
  ◢ src
    ◢ view
      ◢ main
          List.js
          Main.js
◢ modern
  ▷ sass
  ◢ src
    ◢ view
      ◢ main
          List.js
```

Figure 7.2

Now, let's create the grid shown in the design. The following code is used to create the grid. This `List` view extends from `Ext.grid.Panel`, uses the `expense` store for the data, and has three columns:

```
Ext.define('EA.view.main.List', {
  extend: 'Ext.grid.Panel',
  xtype: 'mainlist',
  maxHeight: 400,
  requires: [ 'EA.store.Expense'],
  title: 'Year to date expense by category',
  store: { type: 'expense' },
  columns: {
    defaults: { flex:1 },
    items: [{
      text: 'Category',
      dataIndex: 'cat'
    }, {
      formatter: "date('F')",
      text: 'Month',
      dataIndex: 'date'
    }, {
      text: 'Spent',
      dataIndex: 'spent'
    }]
  }
});
```

Here, I have not used the pagination. The `maxHeight` is used to limit the height of the grid, and this enables the scroll bar as well because we have more records that won't fit the given maximum height of the grid.

The following code creates the `expense` store used in the preceding example. This is a simple store with the inline data. Here, we have not created a separate model and added fields directly in the store:

```
Ext.define('EA.store.Expense', {
  extend: 'Ext.data.Store',
  alias: 'store.expense',
  storeId: 'expense',
  fields: [{
    name:'date',
    type: 'date'
  },
  'cat',
  'spent'
```

```
    ],
    data: {
      items: [
      { "date": "1/1/2015", "cat": "Restaurant", "spent": 100 },
      { "date": "1/1/2015", "cat": "Travel", "spent": 22 },
      { "date": "1/1/2015", "cat": "Insurance", "spent": 343 },
      // Truncated code
    ]},
    proxy: {
      type: 'memory',
      reader: {
        type: 'json',
        rootProperty: 'items'
      }
    }
  });
```

Next, let's create the bar chart shown in the design. In the bar chart, we will use another store called `expensebyMonthStore`, in which we'll populate data from the `expense` data store.

The following 3D bar chart has two types of axis: `numeric` and `category`. We have used the month part of the date field as a category. A renderer is used to render the month part of the date field:

```
  Ext.define('EA.view.main.Bar', {
    extend: 'Ext.chart.CartesianChart',

    requires: ['Ext.chart.axis.Category',
    'Ext.chart.series.Bar3D',
    'Ext.chart.axis.Numeric',
    'Ext.chart.interactions.ItemHighlight'],

    xtype: 'mainbar',
    height: 500,
    padding: { top: 50, bottom: 20, left: 100, right: 100 },
    legend: { docked: 'bottom' },
    insetPadding: { top: 100, bottom: 20, left: 20, right: 40 },
    store: { type: 'expensebyMonthStore' },
    axes: [{
      type: 'numeric',
      position: 'left',
      grid: true,
      minimum: 0,
      title: {
        text: 'Spendings in $',
        fontSize: 16
      },
    }, {
      type: 'category', position: 'bottom',
      title: {
        text: 'Month',
        fontSize: 16
      },
      label: {
        font: 'bold Arial',
        rotate: { degrees: 300 }
      },
      renderer: function (date) {
        return ["Jan", "Feb", "Mar", "Apr", "May"][date.getMonth()];
      }
    } ],
    series: [{
      type: 'bar3d',
      xField: 'date',  stacked: false,
      title: ['Total'],  yField: ['total']
    }],
    sprites: [{
      type: 'text',
      text: 'Expense by Month',
      font: '20px Helvetica',
      width: 120,
      height: 35,
      x: 60,
      y: 40
    }]
  });
```

Now, let's create the `MyApp.model.ExpensebyMonth` store used in the preceding bar chart view. This store will display the total amount spent in each month. This data is populated by grouping the `expense` store with the `date` field. Take a look at how the data property is configured to populate the data:

```
Ext.define('MyApp.model.ExpensebyMonth', {
  extend: 'Ext.data.Model',
  fields: [{name:'date', type: 'date'}, 'total']
});
```

```
Ext.define('MyApp.store.ExpensebyMonth', {
  extend: 'Ext.data.Store',

  alias: 'store.expensebyMonthStore',
  model: 'MyApp.model.ExpensebyMonth',

  data: (function () {
    var data = [];

    var expense = Ext.createByAlias('store.expense');
    expense.group('date');
    var groups = expense.getGroups();

    groups.each(function (group) {
      data.push({ date: group.config.groupKey, total:
group.sum('spent') });
    });
    return data;
  })()
});
```

Then, the following code is used to generate the pie chart. This chart uses the `expense` store, but only shows one selected month of data at a time. A drop-down box is added to the main view to select the month.

The `beforerender` is used to filter the expense store to show the data only for the month of January on the load:

```
Ext.define('EA.view.main.Pie', {
  extend: 'Ext.chart.PolarChart',
  requires: ['Ext.chart.series.Pie3D'],
  xtype: 'mainpie',
  height: 800,
  legend: {
    docked: 'bottom'
  },
  insetPadding: {
    top: 100,
    bottom: 20,
    left: 80,
    right: 80
  },
  listeners: {
    beforerender: function () {
      var dateFiter = new Ext.util.Filter({
        filterFn: function(item) {
          return item.data.date.getMonth() ==0;
        }
      });
      Ext.getStore('expense').addFilter(dateFiter);
    }
  },
  store: {
    type: 'expense'
  },
  series: [{
    type: 'pie3d',
    donut: 50,
    thickness: 70,
    distortion: 0.5,
    angleField: 'spent',
    label: {
      field: 'cat',
    }
  }]
});
```

So far, we have created the grid, the bar chart, the pie chart, and the stores required for this sample application. Now, we need to link them together in the main view. The following code shows the `main` view from the classic toolkit section. The `main` view is simply a tab control and specifies what view to render for each tab:

```
Ext.define('EA.view.main.Main', {
  extend: 'Ext.tab.Panel',
  xtype: 'app-main',

  requires: [
  'Ext.plugin.Viewport',
  'Ext.window.MessageBox',
  'EA.view.main.MainController',
  'EA.view.main.List',
  'EA.view.main.Bar',
  'EA.view.main.Pie'
  ],

  controller: 'main',
  autoScroll: true,
  ui: 'navigation',
  // Truncated code

  items: [{
    title: 'Year to Date',
    iconCls: 'fa-bar-chart',
    items: [ {
      html: '<h3>Your average expense per month is: ' +
Ext.createByAlias('store.expensebyMonthStore').average('total') +
'</h3>',
      height: 70,
    },
    { xtype: 'mainlist'},
    { xtype: 'mainbar' }
    ]
  },
  {
    title: 'By Month',
    iconCls: 'fa-pie-chart',
    items: [{
      xtype: 'combo',
      value: 'Jan',
      fieldLabel: 'Select Month',
      store: ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
      listeners: {
        select: 'onMonthSelect'
      }
    }, {
      xtype: 'mainpie'
    }]
  }]
});
```

Summary

In this chapter, we looked at the different kinds of charts available in Ext JS. We also created a simple sample project called Expense Analyzer and used some of the concepts you learned in this chapter.

8
Theming and Responsive Design

This chapter focuses on the basics of theming your Ext JS application and responsive design. The following topics will be covered:

- Introduction to SASS
- Theming
- Responsive design

An introduction to SASS

SASS (**Syntactically Awesome Stylesheets**) is a stylesheet language. When you use SASS, instead of writing your styling in CSS, you'll write in SASS. Later, SASS will be compiled into CSS with a SASS compiler. SASS has a better syntax and a set of features that make CSS easier. Also, maintaining the SASS code is much easier than maintaining the CSS code. In SASS, you'll write less code compared to directly writing code in CSS.

Normally, when you build your application, you'll use some SASS compiler that will generate the CSS files to be used in the browser.

The topics in SASS are beyond the scope of this book. You don't have to know SASS in detail for theming in Ext JS, but some basic knowledge is required. So, let's learn the bare minimum required for SASS here.

You don't have to install SASS compiler separately, the Sencha Cmd 6 will handle it for you. The Sencha Cmd 6 uses a new SASS compiler called Fashion, which will be installed when you install Sencha Cmd 6.

[If you're using the Ext JS SDK 5 with Sencha Cmd 6 or Sencha Cmd 5, then you'll need Compass and Ruby installed on your machine.]

SASS has two syntax; the default new syntax is called SCSS (Sassy CSS), and the old SASS syntax is called SASS. We'll use the SCSS. Here, you'll learn the basic topics in SASS, such as variable, mixins, and nesting.

Variables

CSS doesn't support variables yet. SASS can help you here. SASS variables are used to store things, such as colors, font stacks, or any CSS value that you want to reuse. Take a look at the following code:

```
$body-background-color: transparent;
$base-color: #808080;
$font-family: helvetica , arial , verdana , sans-serif;

body {
  background: $body-background-color;
  font: 100% $font-family;
  color: $base-color;
}
```

When the preceding code is processed, the output will be in normal CSS, as shown in the following code. The variable defined before can be used in multiple places wherever required:

```
body {
  background: transparent;
  font: 100% helvetica , arial , verdana , sans-serif;
  color: #808080;
}
```

Mixins

These are like macros—a single instruction that expands into multiple instructions. Consider the following SCSS code:

```
@mixin border-radius($radius) {
  -moz-border-radius: $radius;
  -webkit-border-radius: $radius;
  -ms-border-radius: $radius;
  border-radius: $radius;
}

a.button {
  background: black;
  color: white;
  padding: 10px 20px;
  @include border-radius(10px);
}
```

The output of the preceding SCSS code is shown in the following code:

```
a.button {
  background: black;
  color: white;
  padding: 10px 20px;
  -moz-border-radius: 10px;
  -webkit-border-radius: 10px;
  -ms-border-radius: 10px;
  border-radius: 10px;
}
```

Nesting

Unlike HTML, CSS doesn't support the nesting of CSS selectors. SASS lets you do this.

So, instead of writing this CSS code:

```
.table {
  background: blue;
}
.table .col {
  padding: 10px;
}
.table .row .col {
  color: red;
}
```

In SASS, you can write this:

```
.table {
  background: blue;
  .col {
    padding: 10px;
    .row {
      color: white;
    }
  }
}
```

Theming

Ext JS makes theming much easier with less maintenance headache by letting you extend easily from an existing theme.

First, we will need a workspace, so run the following command to generate an application in a subfolder called `myapp` in the current folder:

```
sencha -sdk <path to SDK> generate app MyApp ./myapp
```

Ext JS comes with a set of themes, and you can choose one of them, or you can create your own custom theme by extending one of the themes that Sencha has provided.

By default, Sencha Cmd 6 uses `theme-neptune`. You will find the following code in `app.json` in the preceding generated code for `MyApp`. Here, you can change the theme for the modern and classic toolkit:

```
"builds": {
  "classic": {
    "toolkit": "classic",
    "theme": "theme-triton"
  },

  "modern": {
    "toolkit": "modern",
    "theme": "theme-neptune"
  }
},
```

If you're not using Ext JS 6, but Ext JS 5, then instead of the preceding code, you'll find the following line:

```
"theme": "ext-theme-neptune",
```

Prior to Ext JS 6, all the themes had a prefix called `ext-`, and this prefix has been dropped in Ext JS 6.

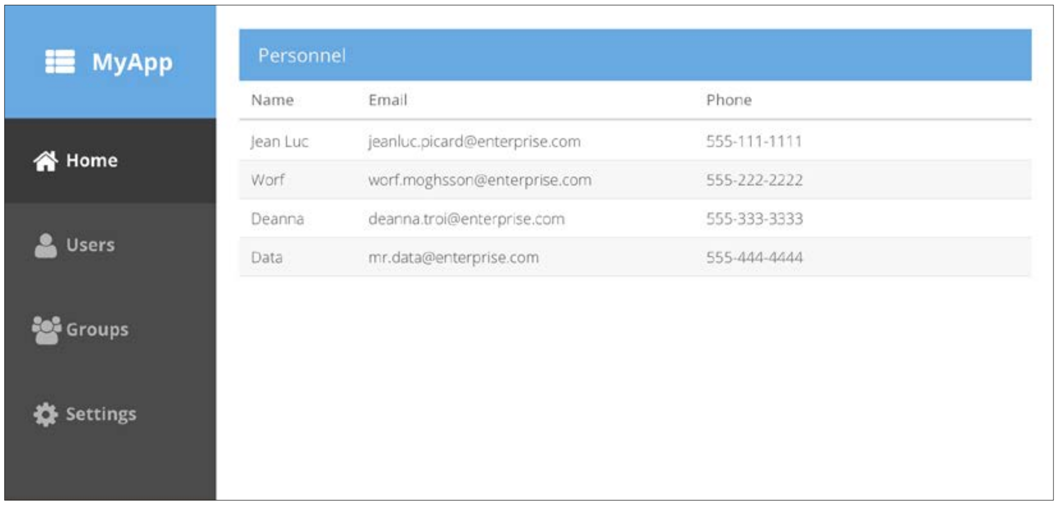
If you want to change the theme to another built-in theme, you can change its value to the theme name that you want to use. Try this and refresh the page to see the difference.

Creating a custom theme

First, let's run the `MyApp` application with the following command:

```
sencha app watch
```

This will start the application at `http://localhost:1841` (the default address). The output is in the following screenshot: uses the default `theme-neptune`.shown.



Name	Email	Phone
Jean Luc	jeanluc.picard@enterprise.com	555-111-1111
Worf	worf.moghsson@enterprise.com	555-222-2222
Deanna	deanna.troi@enterprise.com	555-333-3333
Data	mr.data@enterprise.com	555-444-4444

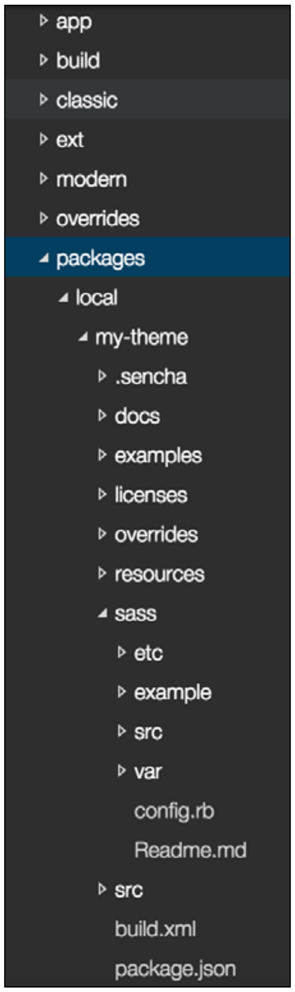
Figure 8.1

Let's use this **MyApp** application and generate a custom theme.

As a first step, run the following command to generate a custom theme from the application folder:

```
sencha generate theme my-theme
```

Now, if you check the packages folder in the application folder, you will find that a lot of files have been generated for `my-theme`. This is how the folder content will look:



```
▹ app
▹ build
▹ classic
▹ ext
▹ modern
▹ overrides
◢ packages
  ◢ local
    ◢ my-theme
      ▹ .sencha
      ▹ docs
      ▹ examples
      ▹ licenses
      ▹ overrides
      ▹ resources
      ◢ sass
        ▹ etc
        ▹ example
        ▹ src
        ▹ var
          config.rb
          Readme.md
      ▹ src
        build.xml
        package.json
```

Figure 8.2

Let's see the purpose of some of these files and folders:

- `package.json`: This file has all the configs and package properties
- `sass/var`: This folder contains all the SASS variables
- `sass/src`: This uses all the SASS rules and mixins
- `sass/etc`: This contains additional utility functions or mixins
- `resources`: This contains images and other static resources of the theme
- `overrides`: This contains JavaScript overrides required for theming

The files and folder structure of `sass/var/`, `sass/src`, and overrides should match the file path of the component you are styling or overriding. For example, variables that change the appearance of `Ext.panel.Panel` should be placed in the `sass/var/panel/` folder with the `Panel.scss` filename.

In theming, as the next step, we have to decide which existing theme to extend. Ext JS 6 ships two sets of theme: one set for the classic toolkit and one set for the modern toolkit. Some of these themes are shown in the following image. When you extend, you have to extend from any of the themes shown, except **Base** and **Neutral**:

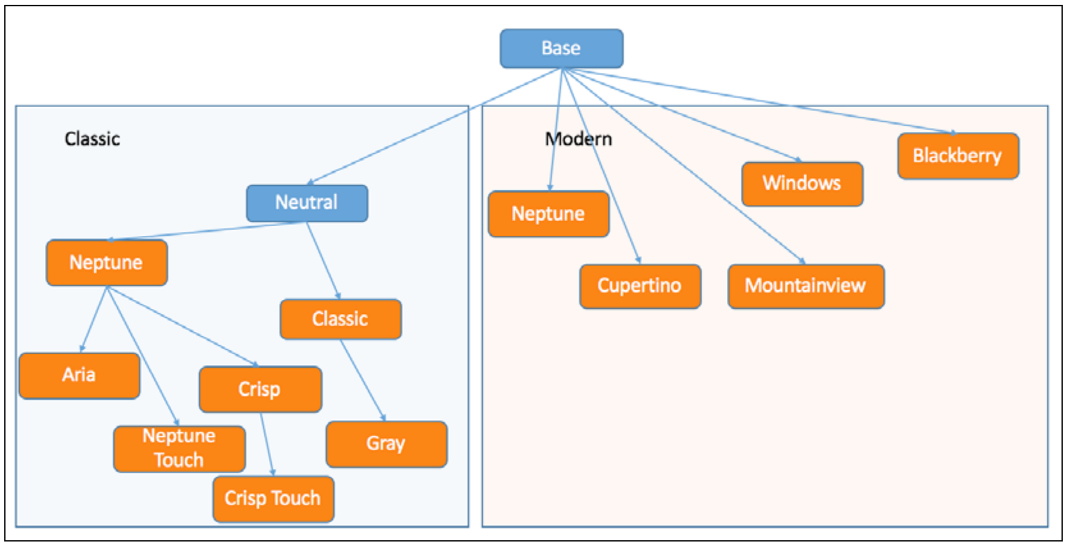


Figure 8.3

When you generate a custom theme, by default, it extends from `theme-classic`. Let's change this to `theme-crisp`. To change this, update the following line in `packages/local/my-theme/package.json`.

```
"extend": "theme-classic",
```

with

```
"extend": "theme-crisp",
```

The generate theme command also takes an optional parameter for the base theme to extend. If you don't specify the optional parameter, then it extends from `theme-classic`.

Next, you can perform one or more of the following:

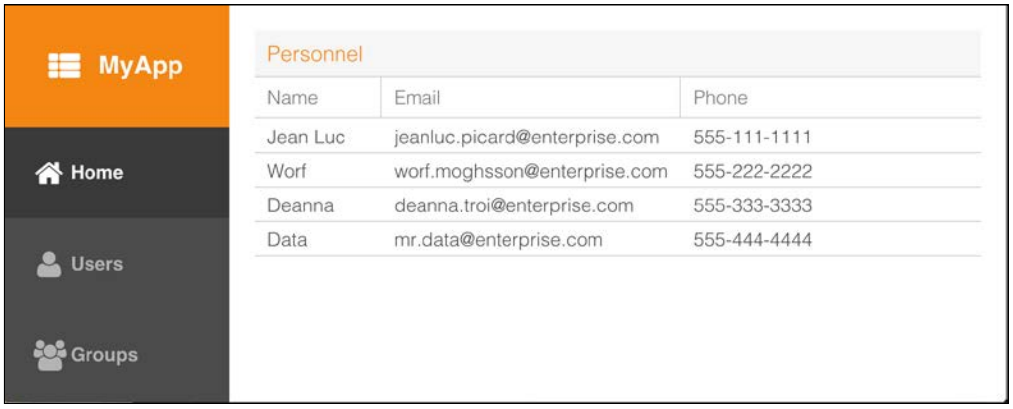
- Configure the SCSS variables
- Create the SCSS rules and mixins
- Override images
- Override the JavaScript styling

SASS variables

Ext JS has defined a whole bunch of SASS variables, and you can override them to customize according to your needs. Now, let's modify some of the global variables. Under the `var` folder, create a file called `component.scss` and add the following lines:

```
$base-color: #F17C26 !default;
$color: #404040 !default;
$font-size: 15px !default;
```

The `!default` is required to allow this to be overridden. If you refresh the page now in the browser, the application will look as follows:



Name	Email	Phone
Jean Luc	jeanluc.picard@enterprise.com	555-111-1111
Worf	worf.moghsson@enterprise.com	555-222-2222
Deanna	deanna.troi@enterprise.com	555-333-3333
Data	mr.data@enterprise.com	555-444-4444

Figure 8.4

Creating a new component UI using SASS mixins

While customizing, the first thing you should try is to check whether you can do this just by using variables alone. Variables should help to customize in most of the cases. Sometimes, it may not be enough. Only in this case, consider creating the component UI and mixins.

Most of the components in Ext JS define SASS mixins; you can call these mixins to generate the new component UI. Most of the components in Ext JS have the component UI, which defaults to the *default* component UI. You can create your own component UIs and use them.

For example, let's create one custom panel UI. Create a file called `Panel.scss` under `my-theme/sass/src/panel/` and place the following code:

```
@include extjs-panel-ui(
    $ui: 'dark',
    $ui-header-background-color: #404040,
    $ui-border-color: #404040,
    $ui-body-background-color: #404040,
    $ui-body-border-color: #404040
);
```

Now, you can use this UI, as shown in the following code:

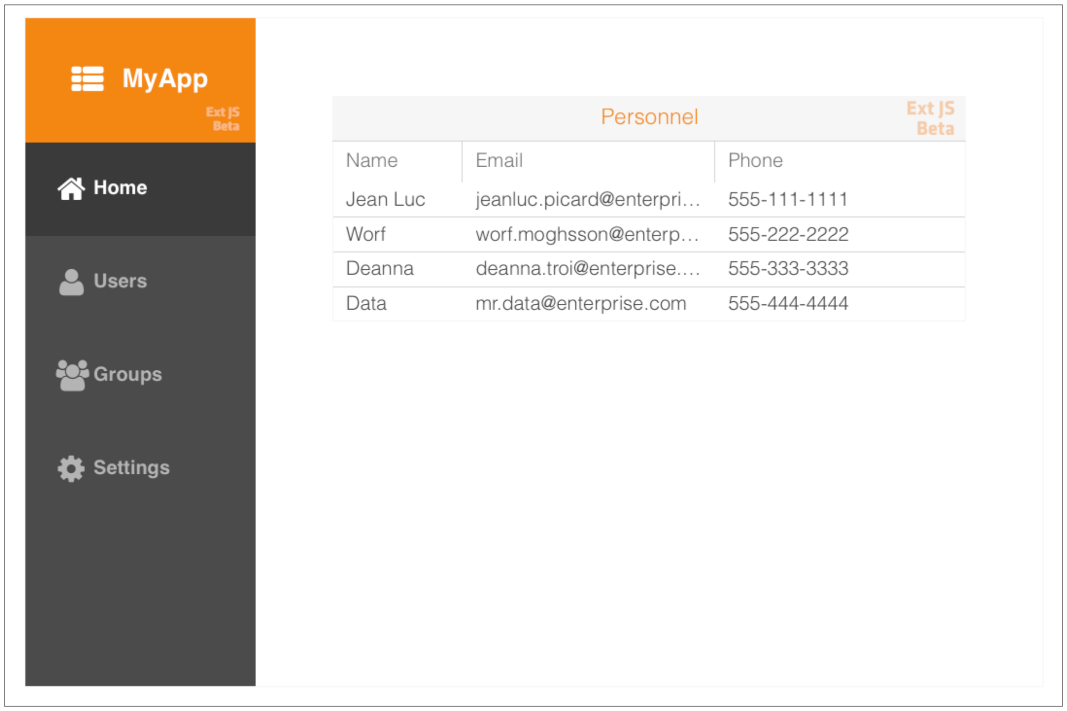
```
Ext.define('MyApp.view.main.MyPanel', {
  extend: 'Ext.panel.Panel',
  xtype: 'my-panel',
  ui: 'dark',
});
```

JS overrides

Rarely, you may want to change the appearance, which can be done only in JS. In such cases, you can use JS overrides. For example, create a file called `Panel.js` under `my-theme\overrides\panel\`, place the following code, and refresh the application to see the difference:

```
Ext.define('my-theme.panel.Panel', {
  override: 'Ext.panel.Panel',
  titleAlign: 'center',
  padding: 20
});
```

The output of the preceding code is as follows:



Name	Email	Phone
Jean Luc	jeanluc.picard@enterpri…	555-111-1111
Worf	worf.moghsson@enterp…	555-222-2222
Deanna	deanna.troi@enterprise.…	555-333-3333
Data	mr.data@enterprise.com	555-444-4444

Figure 8.5

Images

In your theme, you can also easily customize the icons by simple placing the icons under `my-theme\resources\images\`. The images have to be placed with the same filename.

Styling your application

You can do everything that you learned in theming to style your application-specific needs, which can't be in the theme.

If you want to set a variable that is specific to `classic/src/view/main/Main.js`, then you'll have to add a `scss` file called `classic/sass/src/view/main/Main.scss` and set the variables. In the same file, you can add SASS, CSS styles, and you can use mixins to create the view-specific component UIs and place them in this file.

Any valid CSS is a valid SCSS. So, the SCSS file can have the plain CSS code.

Sencha Cmd generates this `classic/sass/src/view/main/Main.scss` to style the `classic/src/view/main/Main.js` view, and you'll find the following code. Similarly, you can generate any style that is specific to this `Main.js` in this file:

```
@include extjs-panel-ui(
    $ui: 'navigation',
    $ui-header-color: #fff,
    $ui-header-glyph-color: #fff,
    $ui-header-glyph-opacity: 1,
    $ui-header-font-size: 20px,
    $ui-header-line-height: 24px,
    $ui-header-font-weight: bold,
    $ui-header-icon-height: 24px,
    $ui-header-icon-width: 24px,
    $ui-header-icon-spacing: 15px,
    $ui-header-background-color: $base-color,
    $ui-header-padding: 0,
    $ui-header-text-margin: 36px,
    $ui-header-noborder-adjust: false
);
.x-tab-icon-el-navigation {
    font-family: FontAwesome;
    color: #acacac;
    .x-tab-over & {
        color: #c4c4c4;
    }
    .x-tab-active & {
        color: #fff;
    }
}
```

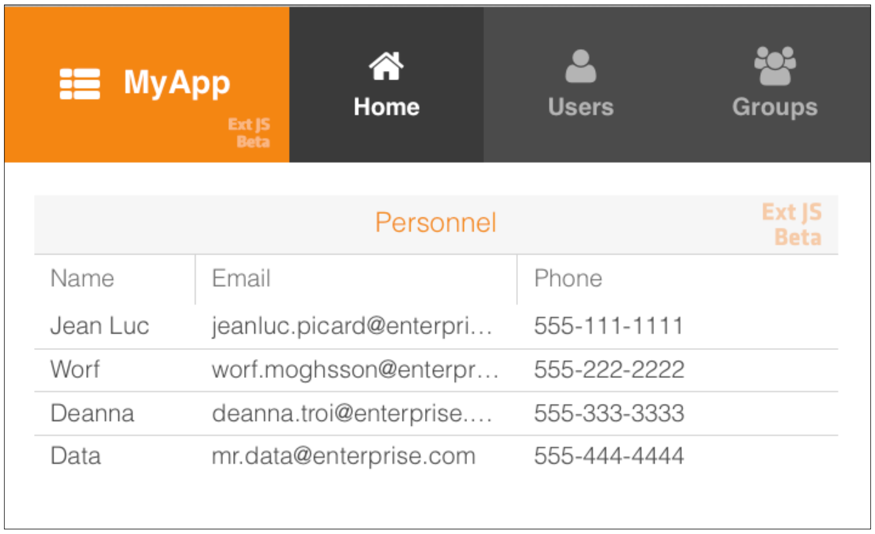
Any style that will be shared between an application should be placed in the theme and not here.

If any style, mixins, or variables are not specific to any particular view, then you can place them in `/sass/etc/all.scss`. For example, if you want to override the base color, then place the following code in `/sass/etc/all.scss`:

```
$base-color: #404040;
```

Responsive design

The default application generated by Sencha Cmd is responsive. Try to resize the application that was generated by Sencha Cmd, and you will see that the left panel moves to the top, as shown in the following screenshot:



Name	Email	Phone
Jean Luc	jeanluc.picard@enterpri…	555-111-1111
Worf	worf.moghsson@enterpr…	555-222-2222
Deanna	deanna.troi@enterprise.…	555-333-3333
Data	mr.data@enterprise.com	555-444-4444

Figure 8.6

If you check `classic\src\view\main\Main.js`, you'll find the following code:

```
responsiveConfig: {
  tall: {
    headerPosition: 'top'
  },
  wide: {
    headerPosition: 'left'
  }
},

defaults: {
  bodyPadding: 20,
  tabConfig: {
    plugins: 'responsive',
    responsiveConfig: {
      wide: {
        iconAlign: 'left',
        textAlign: 'left'
      },
      tall: {
        iconAlign: 'top',
        textAlign: 'center',
        width: 120
      }
    }
  }
},
```

responsiveConfig

Ext JS provides `responsiveConfig` to enable the application to respond dynamically based on the screen size. This is an object with keys that represent conditions under which certain configs will be applied.

By default, components are not enabled for `responsive`. To enable, add the following line to the class definition or the component instance:

```
plugins: 'responsive'
```

The following variables can be used in the `responsiveConfig` rules:

- `landscape`: This is set to true if the device orientation is landscape. This will always be `true` on desktop devices.
- `portrait`: This is set to true if the device orientation is portrait. This will always be `false` on desktop devices.
- `tall`: This is set to true if `width` is less than `height`, irrespective of the device type.
- `wide`: This is set to true if `width` is greater than `height`, irrespective of the device type.
- `width`: This denotes the width of the viewport.
- `height`: This specifies the height of the viewport.

Consider the following code present in `Main.js`. Here, you can see that the value of `iconAlign` and `textAlign` are different for wide and tall, and the width is set only for tall. So, if the width is less than the height, then the configs in tall will be used; otherwise, the configs in wide will be used:

```
tabConfig: {
  plugins: 'responsive',
  responsiveConfig: {
    wide: {
      iconAlign: 'left',
      textAlign: 'left'
    },
    tall: {
      iconAlign: 'top',
      textAlign: 'center',
      width: 120
    }
  }
}
```

These rules can be more complex, as shown in the following code. Here, the first rule checks for two conditions: `width < 768` and `tall`:

```
    responsiveConfig: {
      'width < 768 && tall': {
        visible: true
      },
      'width >= 768': {
        visible: false
      }
    }
```

Similarly, you can use the `platformConfig`, as shown in the following code:

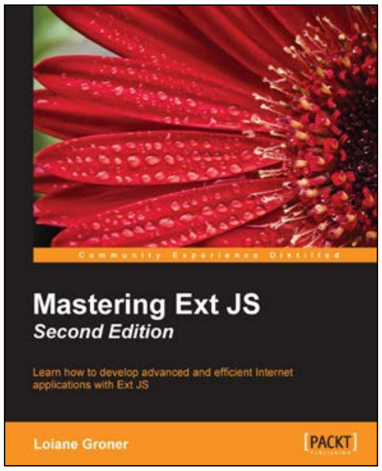
```
    platformConfig: {
      'windows || desktop': {
        visible: true
      },
      android && ios: {
        visible: false
      }
    }
```

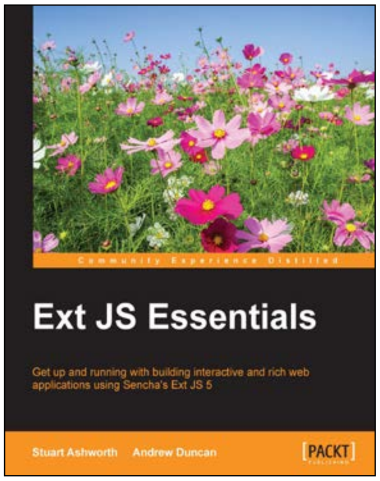
In the preceding code, Windows, desktop, Android, and iOS are defined in `Ext.platformTags`. Some of the other platform tags available include phone, Firefox, Chrome, Safari, touch, tablet, and so on.

Summary

In this chapter, you learned the basics of SASS. We understood how theming works in Ext JS. We also looked at the support available for responsive design.

Index


Community Experience Distilled
Mastering Ext JS
Second Edition
Learn how to develop advanced and efficient Internet applications with Ext JS
Loiane Groner
[PACKT]


Community Experience Distilled
Ext JS Essentials
Get up and running with building interactive and rich web applications using Sencha's Ext JS 5
Stuart Ashworth Andrew Duncan
[PACKT]