



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]
PUBLISHING

Ext JS Essentials

Get up and running with building interactive and rich web applications using Sencha's Ext JS 5

Stuart Ashworth

Andrew Duncan



BIRMINGHAM - MUMBAI

Ext JS Essentials

Copyright © 2015 Packt Publishing

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

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

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

First published: April 2015

Production reference: 1200415

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

ISBN 978-1-78439-662-6

www.packtpub.com

Credits

Authors

Stuart Ashworth
Andrew Duncan

Copy Editors

Puja Lalwani
Laxmi Subramanian

Reviewers

Dale Chinault
Stefan Gehrig
Dmitry Pashkevich

Project Coordinator

Danuta Jones

Commissioning Editors

Ashwin Nair

Proofreaders

Simran Bhogal
Dan McMahon
Linda Morris

Acquisition Editors

Meeta Rajani
Sam Wood

Indexer

Priya Sane

Content Development Editor

Mohammed Fahad

Production Coordinator

Komal Ramchandani

Technical Editor

Pankaj Kadam

Cover Work

Komal Ramchandani

About the Authors

Stuart Ashworth is a freelance web and mobile developer and an all-round web geek currently living in Glasgow, Scotland, with his fiancée, Sophie, and wee dog, Meg. Since graduating with a first class honors degree in design computing from the University of Strathclyde, Stuart has worked in the IT industry creating software for both large and small companies across the UK and around the world.

Stuart has worked with Sencha technologies for over 5 years, creating numerous web applications, mobile applications, and framework plugins along the way. He coauthored *Ext JS 4 Web Application Development Cookbook*, Packt Publishing and also curates content for the fortnightly Sencha Insights e-mail newsletter (www.senchainsights.com).

Stuart enjoys playing football, snowboarding, and traveling. He blogs about Sencha and web technology on his website (www.stuartashworth.com), and can be contacted through Twitter at @StuartAshworth9, through e-mail at stuart@stuartashworth.com, or through the Sencha forums.

Andrew Duncan is a technologist and businessman from Scotland. He has a combined passion for IT and business and now runs his own company, SwarmOnline, which specializes in cloud and mobile applications. Although Andrew is a managing director, he is also an experienced and highly competent technologist and solutions architect. His first experience with Ext JS was in 2009 when he introduced the framework into the NHS. Since then, he's become passionate about Sencha and its technologies.

As experts in Sencha technologies, Andrew and his team employ them regularly to build rich and complex business applications. His experience lead him to coauthor *Ext JS 4 Web Application Development Cookbook*, Packt Publishing and he is now a sought-after conference speaker, having previously spoken at numerous technology events across the globe.

About the Reviewers

Dale Chinault has built enterprise applications using nearly two dozen software languages and managed more than a dozen different database platforms. Over the past 5 years, he has been primarily developing Ext JS web frontends, but occasionally finds the opportunity to brush off his Java and database skills by working on server-side functionality. He is currently a UI staff engineer at Virtual Instruments, a company that provides tools that help large enterprises manage mission-critical applications and massive datacenter infrastructures.

With a degree in computer science and a minor in math from Virginia Tech, along with 25 years of diverse work experience, Dale brings unique insight to the challenges that development teams face each day. Having performed nearly every role in the development and business ownership arena, he is able to talk shop with anyone, be it CTOs, testers, DBAs, architects, project managers, business owners, infrastructure administrators, or developers. This often pays dividends by helping him to build a level of understanding and consensus among team members.

In addition to software, Dale is also interested in electronics, microcontrollers, and robotics and especially likes to tinker with Arduino and Raspberry Pi projects. Otherwise, Dale can be found constructing things around the house, biking, hiking, backpacking, or traveling to some far-off place.

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

His main focus for the past few years has been writing a company-internal rapid application development framework based on Ext JS and PHP, and implementing the framework in data-driven enterprise applications.

Dmitry Pashkevich is a software engineer at Lucid Software, where he works on Lucidchart and its integrations with third-party SaaS products. He has a passion for crafting web applications that keep great user experience in mind.

When away from the keyboard, he enjoys traveling, photography, mountain biking, and tea.

You can find him online at <http://dpashk.com>.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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

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



<https://www2.packtpub.com/books/subscription/packtlib>

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

Why subscribe?

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

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	ix
Chapter 1: Getting to Know Ext JS	1
What is Ext JS	2
What Ext JS isn't	3
Use cases	3
What's new in Ext JS 5	4
Touch support	4
Architectural improvements	4
Responsive layouts	4
Component enhancements	4
What you need to know	5
Development environment	5
Project introduction	5
Creating our application with Sencha Cmd	6
Installing Sencha Cmd	6
Downloading the framework	7
Applications, packages, and workspaces	7
Generating our workspace	7
Generating our application	8
Getting production ready	9
Deployment recommendations	9
GZip	10
Minification and concatenation	10
Image optimization	10
Anatomy of our application	10
How it works	11
The bootstrapped launch process	11
JavaScript to HTML	11
The event system	12

Managing data	12
Browser API interaction	12
Routing	12
Summary	12
Chapter 2: Mastering the Framework's Building Blocks	13
Defining classes	13
The magic behind Ext.Loader	15
The class definition process	16
Defining dependencies	16
Loader paths	17
Ext.Loader and Sencha Cmd	18
Our dependency root	18
Adding class members	19
Properties	19
Methods	19
Statics	20
Statics in subclasses	20
Singletons	21
Extending classes	21
Overriding classes	22
Requiring our override class	23
Targeting overrides to framework versions	24
Configuring classes	24
Setting a configuration value	25
Overriding defaults	27
Platform-specific configs	27
Summary	28
Chapter 3: Reacting to User and Application Interactions	29
Background	30
Defining event handlers in config objects	30
Button handlers	31
The on method	32
Listener options	33
Firing events	34
Listening for events on elements	35
Event delegation	35
Mouse events	36
Keyboard events	36
KeyMap	36

Touch events	37
Event normalization	37
Gestures	37
Summary	38
Chapter 4: Architecting an Ext JS Application	39
Getting the most from Sencha Cmd	40
Generating application components	40
Generating models	40
Generating views	40
Generating controllers	41
Upgrading your application	42
Refreshing application metadata	42
Triggering automatic builds	42
MVC and MVVM	43
What is MVC?	43
Explaining the abbreviations	43
Putting this together	44
Ext JS naming convention and directory structure	45
The benefits and drawbacks of using MVC	45
What is MVVM?	46
Addressing the concerns	46
Explaining the abbreviations	46
Business logic	48
Cross-class communication with events	49
Taking your application offline	49
Why should we design offline first?	50
What can we do about this?	50
How can we do this?	51
Offline architecture	51
Syncing data	51
Summary	52
Chapter 5: Modeling Data Structures for Your UI	53
Defining models	54
Fields	55
Field validation	55
Custom field types	57
Custom data converters	58
Working with stores	59
Simple stores	59
Store stats	60
Retrieving a record	60
Finding specific records	61

Complex searches	61
Filtering a store	62
Sorting a store	63
Grouping	64
Chained stores	66
TreeStores	68
Ext.data.TreeModels	68
Creating a TreeStore	70
Populating a TreeStore	70
Getting data into your application	72
Ext.Ajax	72
Simple AJAX calls	72
Handling errors	73
Other useful configurations	74
Proxies	74
AJAX proxies	75
LocalStorage proxies	77
REST proxies	79
Data associations	79
One-to-many	80
Configuring a proxy and data source	80
Defining the association	81
The reference config	83
Exploring requests	84
Many-to-many	85
Configuring a proxy and data source	85
Defining the association	86
Loading the associated data	87
Saving data	87
CRUD endpoints	89
Data writers	89
Summary	91
Chapter 6: Combining UI Widgets into the Perfect Layout	93
Layouts and how they work	94
How the layout system works	95
The component layout	96
Using the border layout	96
Starting with the Viewport	97
Configuring the border layout	97
Using the fit layout	99
Using the HBox layout	100
Widths in HBox layouts	102
Fixed width	102
Flex width	102

Packing items together	103
Using the VBox layout	103
Alignment and packing	104
align: String	104
pack: String	104
Responsive layouts	104
Ext.mixin.Responsive and Ext.plugin.Responsive	105
ResponsiveConfig rules	107
Summary	107
Chapter 7: Constructing Common UI Widgets	109
Anatomy of a UI widget	109
Components and HTML	110
The component lifecycle	111
The creation lifecycle	111
constructor	112
Config options processed	112
initComponent	112
render	112
boxready	112
activate (*)	112
show (*)	112
The destruction process	113
hide (*)	114
deactivate (*)	114
destroy	114
Component Queries	114
xtypes	115
Sample component structure	115
Queries with Ext.ComponentQuery	118
Finding components based on xtype	118
Finding components based on attributes	118
Finding components based on itemIds	119
Finding components based on member functions	119
Scoped Component Queries	119
up	119
down	119
query	120
Hierarchical data with trees	120
Binding to a data source	120
Defining a tree panel	120
Displaying tabular data	122
Product data	122
Product grid	123

Customizing column displays	125
Column renderers	125
Template columns	126
Grid widgets	127
Inputting data with forms	128
Defining a form	128
Displaying our form	129
Populating our form	131
Persisting updates	133
Data-bound views	134
Defining our users' data view	135
Store	135
Template	135
Item selector	136
Styling the view	136
Summary	137
Chapter 8: Creating a Unique Look and Feel with SASS	139
Applying themes to your application	139
Configuring a new theme	141
Creating a custom theme	141
Theme architecture	141
Generating a theme package	142
Anatomy of a theme	143
Cross-browser styling	144
Theme inheritance	144
Applying the new theme	144
Basic theme customizations	144
Theme variables	145
Changing the main color	145
Changing the font size	146
Changing a button's color	146
Custom component UIs	147
Defining UIs	147
Applying UIs	148
Other UIs	149
Summary	150
Chapter 9: Visualizing Your Application's Data	151
Anatomy of chart components	151
Series	151
Axes	152
Labels	152

Interactions	152
Creating a line chart	152
Creating the store and model	153
Polling the server for new data	154
Presenting data in a bar chart	156
Creating a pie chart in Ext JS	158
Integrating visualizations in grids	160
Summary	163
Chapter 10: Guaranteeing Your Code's Quality with Unit and UI Testing	165
Writing testable JavaScript	166
Single responsibility	166
Accessible code	166
Nested callbacks	167
Separate event handlers from actions	168
Testing frameworks	169
Jasmine	169
Siesta	169
Writing unit tests	170
Testing project structure	171
Creating the test harness	171
Adding the first test	173
Executing tests	174
Extending tests	174
Testing UI interaction	175
Testing cell contents	175
Setting up expected data	176
Checking cell contents	177
Simulating clicks	177
Event recorder	179
Test automation and integration	182
Test reports	182
Summary	183
Index	185

Preface

Ext JS 5 is a heavyweight JavaScript framework used by millions to build rich and interactive web applications. Its numerous widgets and advanced data package make it especially well-suited for enterprise class software. The framework encourages good architecture and is extremely customizable.

Ext JS Essentials is written to give you a fast-track understanding of Ext JS. This book covers the most important aspects of the framework in a concise but comprehensive way ensuring your success using its many features.

Written around an example application, the book is packed with practical insights on how the framework works, architecting your applications, working with data, and the many widgets on offer.

What this book covers

Chapter 1, Getting to Know Ext JS, introduces the Ext JS framework and outlines its strengths and weaknesses, including the types of applications it is well suited to. We will then create our example project using Sencha Cmd.

Chapter 2, Mastering the Framework's Building Blocks, covers Ext JS' class system, where we will learn to define, create, and configure classes. Concepts such as inheritance and overriding will also be touched upon.

Chapter 3, Reacting to User and Application Interactions, helps us explore how users interact with our application, including how events are triggered and listened for.

Chapter 4, Architecting an Ext JS Application, covers the MVC and MVVM patterns and how they can be used to architect your applications.

Chapter 5, Modeling Data Structures for Your UI, goes into detail on how to model data structures with the Ext JS data package. We will then cover loading and saving data to and from external data sources.

Chapter 6, Combining UI Widgets into the Perfect Layout, introduces the Ext JS layout system and explains the main layout types in depth with an example of each.

Chapter 7, Constructing Common UI Widgets, introduces the main UI widgets that the framework offers, including trees, grids, forms, and data views. The component lifecycle and Component Query syntax are also covered.

Chapter 8, Creating a Unique Look and Feel with SASS, shows you how to create a theme for your Ext JS application and covers creating a custom theme, customizing it with SASS mixins and component UIs.

Chapter 9, Visualizing Your Application's Data, explores how to visualize data using the Ext JS chart packages with examples of all of the major chart types.

Chapter 10, Guaranteeing Your Code's Quality with Unit and UI Testing, explores techniques for writing testable code in detail, with a hands-on introduction to the testing framework, Siesta.

What you need for this book

To follow along with this book, we recommend you have your IDE ready along with a browser with some developer tools – we recommend Google Chrome and its Developer Tools.

You will also need a local webserver available to run the example project.

The example project requires the Ext JS 5 SDK, which is available from the Sencha website <http://www.sencha.com/products/extjs>. We will also make use of Sencha Cmd, which can be downloaded from <http://www.sencha.com/products/sencha-cmd>.

The Siesta testing tool is also introduced and can be downloaded from <http://www.bryntum.com/products/siesta>.

Who this book is for

If you are a developer looking to develop rich Internet applications, then this book is for you. This book assumes that you have experience developing software and enables you to quickly hit the ground running with this amazing framework.

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: "The `ext` folder contains the Ext JS framework code which our new application will make use of."

A block of code is set as follows:

```
Ext.define('BizDash.config.Config', {  
  
    }, function() {  
        console.log('Config class created');  
    });
```


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('BizDash.config.Config', {  
    singleton: true,  
    }, function() {  
        console.log('BizDash.config.Config defined');  
    });
```

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

Sencha app build

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 on 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.

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 to Know Ext JS

The term "app" has revolutionized the way we access information and interact with each other and organizations. Designing and developing apps presents numerous challenges to software engineers, which can hinder our ability to release these apps quickly and remain agile in our development process.

Assistance from frameworks and libraries is better now than ever before, and with Sencha's latest release, Ext JS 5, we can produce even richer, real-time, and engaging experiences for our users.

Ext JS Essentials aims to touch upon all the major topics on Ext JS. By the end of this book, you'll understand:

- How to use Ext JS to build rich and responsive applications
- Web application architecture with the **Model-View-Controller (MVC)** and **Model-View-ViewModel (MVVM)** patterns
- The framework's fundamentals, including classes, events, and user interactions
- Ext JS' core layouts, widgets, and components
- The main concepts of the data package and two-way data binding
- Methods to visualize datasets with charts
- The tools you can use to enhance your development experience
- Where to start when customizing your interface with SASS
- What you can do to test your application

This chapter will provide a high-level overview of the framework, and explain why you should invest time and energy into learning its techniques, technology, and tools.

It's expected that you already have a programming background, but you'll discover the knowledge you require to get the most from the framework. The examples worked upon throughout the book will center on a single application, and by the end of the chapter, your development environment will be ready, and the application will run!

What is Ext JS

Sencha Ext JS is an application development platform that allows you to build rich user experiences using web technologies and standards. Ext JS is ideally suited to building single-page applications and provides all the tools required to enable this. However, if you only want to use a small subset of the framework, it is possible to incorporate just the widgets and classes you require. You can embed these directly on a web page, just like you would a jQuery component.

Unlike other application development frameworks, Ext JS comes with a few tricks up its sleeve. Firstly, it has been specifically designed to work cross-platform and cross-browser. Much of the headache that comes with cross-browser development has been removed for us. With little effort, your apps will run on:

- Internet Explorer 8 and above
- Firefox
- Chrome
- Safari 6 and above
- Opera 12 and above

Sencha hasn't forgotten iOS and Android either. From Ext JS 5, you'll be able to run your apps smoothly on iOS 6+ and Android 4.1+.

Secondly, Ext JS encourages best practices and the use of architectural patterns in web apps. Ext JS is an object-oriented framework with a clearly defined structure and naming convention. Ext JS supports MVC and MVVM architectures, but you're not limited by Sencha; feel free to override and extend the framework to meet your needs.

Finally, the framework is supported by thorough and well-written documentation alongside an outstanding community of users, often found on the Sencha forums or StackOverflow. There is also a community of plugins available from the Sencha Market or on GitHub. The resources are out there for you to find. Go looking, and if you can't find something just ask.

What Ext JS isn't

Before you go any further, it might be worth taking a few seconds to consider what Ext JS is not. It's not for you if you're looking to design for mobile first. Sencha has made major advancements with the framework recently to give it touch support, but it's still not best suited for mobile phones. Sencha Touch, on the other hand, is a framework specifically designed for mobile applications.

Ext JS also isn't best suited for use on websites, unless you have a specific use case. For example, if you're looking for a drop-down menu or tab panel for your website, you might be better off looking at other frameworks, such as jQuery. We're not saying that Ext JS can't do these things, but the framework has a large overhead and might be too much for these simple use cases.

As the framework is complex and contains a large number of components, it can sometimes be cumbersome to customize components and/or the look and feel of your application to suit your needs. Doing it well requires time, patience, and knowledge, but the results speak for themselves.

Use cases

Ext JS is used across the globe by thousands of organizations to deliver web applications to their users. The architectural patterns and components, such as grids, make it particularly well-suited to business-related applications.

Commonly, developers are using Ext JS for:

- Trading apps with real-time data
- Enterprise data management apps
- Intranet applications
- Visualization applications
- Consumer-facing web apps
- Native-wrapped desktop apps
- Data capturing and monitoring systems
- Dashboards and portals

Obviously, this list is not exhaustive, but it should give you an idea of how others are using Ext JS in the real world.

What's new in Ext JS 5

Ext JS 5 is yet another step forward for application developers and the end user. Its new features make it one of the world's most advanced multi-device JavaScript frameworks. Here's a summary of the major enhancements in this framework.

Touch support

From one codebase, you can now build truly cross-platform apps. This release of Ext JS introduces touch capability, enabling you to deliver your desktop applications on touch-enabled devices. This puts an end to the frustration felt by users on tablets, such as iPads, or touch-screen laptops.

Sencha has provided a theme called *Neptune Touch* which is more appropriate for use on touch-enabled devices. This is primarily done by increasing the size of the tappable components on-screen.

A *Crisp*, version of *Neptune Touch* gives you further choice as a starting point for your apps.

Architectural improvements

Further efficiencies and architectural improvements help make the framework more responsive and better for building applications. The most notable improvement here is the new MVVM architectural pattern, which enables us to develop with Ext JS using less application logic. *Chapter 3, Reacting to User and Application Interactions*, will take a much deeper dive into MVVM and will explain how to produce apps using it.

Responsive layouts

Ext JS 5 provides the ability for your apps to have an optimal viewing experience across desktops and tablets, regardless of orientation changes, using a new responsive config system.

Component enhancements

Sencha has made a number of enhancements to the components in the framework. For example, grids have the ability to have widgets added directly to the grid cell. This is going to be great for data visualization and flexible user experiences. The charting package has been upgraded for financial charting as well as to be touch-optimized.

What you need to know

Getting started with Ext JS doesn't require a huge amount of upfront knowledge, as there are plenty of examples and resources to help. However, having experience of general programming and object-oriented programming will stand you in good stead. If you happen to know JavaScript (and JSON), that is even better.

We'll be covering the MVC and MVVM architectural patterns in this book, but any prior knowledge you might have from another framework will make it easier for you. The same applies for theming your application: we'll demonstrate using SASS in an Ext JS app, but won't cover the technology in detail.

Development environment

The development environment for Ext JS need be no more than a basic text editor, a local web server, and a web browser to view the output. Having said that, there are tools, some freely available, that will make your experience better. We would recommend getting familiar with your browser's built-in tools and add-ons:

- Developer Tools in Chrome
- Firebug in Firefox
- Developer Tools in Internet Explorer
- Dragonfly in Opera
- Developer Tools in Safari

Finally, it's worth noting that in recent years, JavaScript support for **Integrated Development Environments (IDEs)** has improved greatly. Ext JS works particularly well with JetBrains IntelliJ Idea (or WebStorm if you're looking for something more basic), Eclipse, and Spket among others.

Project introduction

In order to make our journey through the Ext JS framework as informative and relevant as possible, we will be developing a real-world application from start to finish. We will cover each step of the application building process and we will incorporate every new concept, widget and class that we come across into this application to impart practical knowledge that can be put to use straight away.

Our application will be a business dashboard application that will present a variety of information in a number of different formats; it will allow users to create and manipulate data structures and create a customized look and feel.

By the end of the book, our application will look like the screenshot that follows and will include the following features:

- MVVM architecture
- Data displayed in a variety of different charts
- Interactive data grids
- Custom data views
- Two-way data-bound forms

Navigation	Name	Description	Quantity	Price	Action	Historic Sales
Home	Product 1	Product 1 Description	1	£9.99 (€9.99)	Details	
> Users	Product 2	Product 2 Description	5	£2.99 (€14.95)	Details	
	Product 3	Product 3 Description	1000	£5.49 (€5490.00)	Details	

Creating our application with Sencha Cmd

Sencha Cmd is a command-line tool that automates a variety of tasks relating to the creation, development, and deployment of Ext JS and Sencha Touch applications. It has a huge number of features, many of which will be discussed in detail in *Chapter 3, Reacting to User and Application Interactions*. In this section, we will discuss how to install Sencha Cmd, use it to generate our business dashboard application, and ready it for production deployment.

Installing Sencha Cmd

Sencha Cmd is a cross-platform tool and has variations available for each major platform, but it relies on a couple of dependencies that must be installed first. These are:

- Java Runtime Environment v1.7
- Ruby
 - This comes preinstalled on OS X
 - For Windows, this is available from <http://rubyinstaller.org/downloads/>
 - For Ubuntu, download this with `sudo apt-get install ruby2.0.0`

With these installed, head over to the Sencha website (<http://www.sencha.com/products/sencha-cmd/download>) to download the package relevant to your OS. Follow the instructions within the installer and you should be ready to go.

To verify that the installation has been successful, open a new Terminal or Command Prompt window and run the command `sencha`. You should see the Sencha Cmd help text appear, listing the available commands.

Downloading the framework

Before we can create our application, we must first download the Ext JS framework from the Sencha website (<http://www.sencha.com/products/extjs/#try>). Extract this archive to a suitable location. We're now ready to create our application.

Applications, packages, and workspaces

There are three main entities to understand when it comes to structuring your application: applications, packages, and workspaces.

An **application** is a complete product that brings all your functionality and features together. Each application has its own `index.html` page and generally stands on its own.

A **package** is a self-contained piece of code designed to be shared between applications and could be local to your workspace or distributed through the Sencha Package Manager and a remote repository. An example of a package may be a custom UI component.

Finally, a **workspace** is a special folder that groups multiple applications and packages, allowing them to share common code and framework instances.

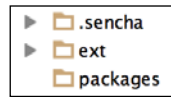
Generating our workspace

We will use Sencha Cmd's `generate` command to create our basic application structure, but we will start by creating a workspace.

First, we open a Terminal/Command Prompt window and navigate to the Ext JS framework folder we extracted earlier. We then run the following command:

```
sencha generate workspace /your/workspace/path
```

This tells Sencha Cmd to generate a new workspace in the specified folder, which will be created if it doesn't already exist. The contents of the workspace can be seen in the following screenshot:



The `.sencha` folder is a hidden folder that contains configuration files that Sencha Cmd makes use of. You will only need to delve into this folder if you are customizing the build process or heavily customizing the application.

The `ext` folder contains the Ext JS framework code which our new application will make use of.

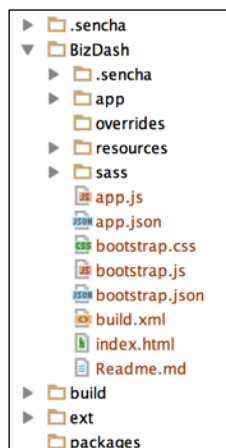
Finally, the empty `packages` folder will be the home of any packages that we choose to include. This could be user extensions, themes, or simply common code to be shared between applications.

Generating our application

Now that we have a workspace for our application to live in, we can create it using the `generate app` command and pass to it the name of our application and the path to the application's folder, within the workspace:

```
sencha generate app BizDash /your/workspace/folder/BizDash
```

This command will create a basic application and folder structure inside a folder named `BizDash`, as shown in the following screenshot. We will explain what all the folders and files inside this folder are in the next section.



This structure will form the basis of our business dashboard application.

You can navigate to the `index.html` file with your browser and see what it looks like. You should see a simple application with a tab bar and a button.

Getting production ready

Obviously, we're a long way off from the production of our application, but it is important to have things set up correctly from the start so that we can quickly deploy when we need to and have this process continually running.

When we view our application now, and monitor the file requests in the Developer Tools, we will see a lot of activity – totaling over 350 requests and 6 MB of data transfer! This is far from ideal for end users, and so, we want to combine these requests into a single minified and compressed file.

We do this by building our application with Sencha Cmd. This process will combine all of the class files used by the application (that is, the 350 requests we mentioned earlier) into a single JS file, along with other building tasks. There are two types of builds: testing and production. A testing build will combine the code but leave it unminified, whereas a production build will minify it fully.

We build the application for production with the following command:

```
sencha app build production
```

This outputs our built application to the `build/production/BizDash` folder. If you load this page in your browser, you will see that the number of requests has dropped to 6 and only totals 1.3 MB. Much better!

In addition to concatenating and minifying our JavaScript, this process will compile our SASS styles, generate image sprites for older browsers, and build a cache manifest file.

Deployment recommendations

Although we have already saved ourselves over 5 MB, there are still further enhancements that should be made to ensure our application loads as fast as possible. We will outline a few of these methods here, but for a more complete list, check out Google's PageSpeed or Yahoo's YSlow, which will analyze the application and make suggestions.

GZip

GZipping allows compression of the content sent to the browser, making it much quicker to download. This is a setting that can be enabled on most web servers.

Minification and concatenation

We have already talked about these two processes, which are taken care of by the Sencha Cmd process. However, it is important to remember when including third-party libraries and frameworks into your applications.

Image optimization

A lot of emphasis is placed on JavaScript size when it comes to page weight, but often, removing a single image will cut your page size in half. If images can't be removed, then make sure they are fully optimized, using a tool such as ImageOptim.

Anatomy of our application

Let's take a step back and understand what Sencha Cmd has created in our application's folder, and where our app's code will belong:

- `.sencha`: Our app's `.sencha` folder is similar to the one found in the workspace. The files within it allow us to gain fine-grain control over aspects of the app itself and its build process.
- `app`: The `app` folder is where we'll spend most of our time, as it contains all of our JavaScript source code. Each of the main class types has its own folder in here by default, including controllers, models, stores, and views. New folders can be added here at any time, as required by the application.

Ext JS 5 introduces a new architecture concept to the framework called MVVM, which will be discussed further in a later chapter. When using this structure, we will include our ViewModels and ViewControllers in the `view` folder.

- `Application.js`: This file is where the application is defined and where it will be launched from. In this file, we will define which controllers, stores, and views we want to load, and the code we want to run when the browser and framework are ready.

You will note that there is also an `app.js` file at the root level. This file should not need to be edited and with any "application" customizations being added to `Application.js`.

- `overrides`: Any overrides we want to make to the framework's code can be added here.
- `resources`: This folder will contain any assets (images, icons, fonts, and so on) that our application will make use of. These are all copied into our production `build` folder when the application is built.
- `sass`: The `sass` folder will be the home of all of our custom SASS styling rules, which will be compiled during the Sencha Cmd build process.
- `app.json`: Our `app.json` file contains a large number of configuration options for our application and can be used to configure things such as JavaScript and CSS files included in the build, AppCache details, and the active theme.
- `build.xml`: This file allows us to hook into each step of the automated build process and add our own steps. This is useful if we want to customize the process to fit our workflow.
- `bootstrap.css`, `bootstrap.js`, and `bootstrap.json`: These three files are required to launch the application, but are generated by the Sencha Cmd build process, so should not be edited by hand.

How it works

At this point, we have created a working skeleton application ready to be fleshed out with our business logic and user interface, but how does the framework actually work and what does it do for us that other frameworks don't?

The bootstrapped launch process

The process for launching and running our applications is simple with the framework taking responsibility for including all assets, in the correct order, as and when they're needed. We simply define which classes rely on which other classes, and the framework builds this relationship map for us.

JavaScript to HTML

Ext JS manages the entire HTML generated to display our user interface to the user. We deal primarily in JavaScript configuration of interface components that are then rendered as HTML. By doing this, we are able to be abstracted from the complex HTML and CSS that is needed to render the rich widgets perfectly across all platforms.

The event system

Ext JS classes make use of an event system that allows them to communicate seamlessly with each other. This makes it easy to keep coupling low and is perfect to simplify the handling of the asynchronous nature of JavaScript.

Managing data

One of the big strengths of the framework is that it allows you to model your data structures and manage data within your applications effectively. The support for creating associated data models, reading and saving data to various sources, and binding interface components directly to these data sources, makes the framework extremely powerful.

Browser API interaction

There are numerous browser APIs that Ext JS abstracts and interacts with on our behalf, to simplify and unify the way we use them. For example, using this approach, switching between saving data to a server API or a LocalStorage data store is a simple configuration change.

Routing

Ext JS 5 has introduced a new routing system to allow us to enable the back button in our single-page web applications, and to give direct access to specific areas of an application.

Summary

This chapter has focused on setting the scene for the Ext JS framework and explained how to use it to create incredible web applications. Its use cases are varied, and by using it you can guarantee that you are building your project on a solid base with the features on hand to create an application that is reliable, maintainable, and most importantly, functional.

We have also created the basis of the book's project application, which will be extended in each chapter of the book. By the end, we will have developed a real-life, living application. We feel this approach is essential to keeping the lessons relevant and practical, ensuring that you can go on to create your own application immediately.

2

Mastering the Framework's Building Blocks

The Ext JS class system forms the basis of the framework and provides us with an object-oriented structure to construct our applications. This chapter will introduce the fundamentals of the class system, and how we can use it to define the building blocks of our applications.

We will discuss the following topics in this chapter:

- How basic object-oriented principles, such as inheritance, are used
- How to dynamically load our classes with the Ext.Loader class
- How to override class methods
- How to use Ext JS' configuration model

Defining classes

Our application's first class will be a configuration class to hold various options for our application. We define a class using the `Ext.define` method as shown here:

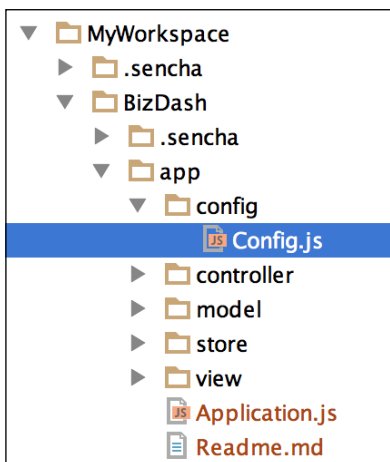
```
Ext.define('BizDash.config.Config', {  
    },  
    function() {  
        console.log('Config class created');  
    });
```

The first parameter is a string defining the class' name, the second, an object containing all the class' members, and the third is an optional callback function, executed when the class has been defined.

Class names must follow a strict naming convention so that they can be auto loaded by the `Ext.Loader` class. This class automatically loads classes when they are needed and uses their fully qualified name to find them within the directory structure.

- The first part of our class name — `BizDash` — is our application's name and is our root namespace. This is mapped, by default, to the `app` folder inside our application folder.
- The second part — `config` — is our class' subnamespace and is used to organize our classes into folders. This name maps to a subfolder inside the `app` folder. We can create namespaces to any depth we wish, to allow us to organize our code in a way that makes sense for the application.
- Finally, `Config` is the name of our class and forms the file name where our class definition will live.

The following screenshot shows how our fully qualified class name equates to its directory structure:



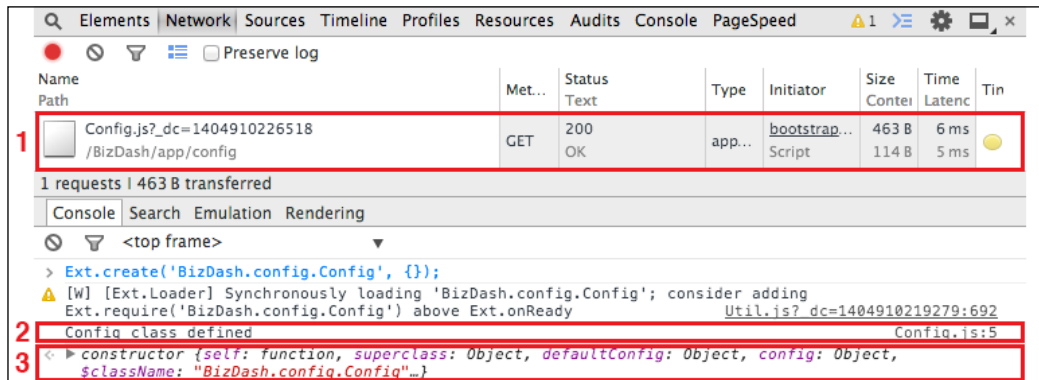
The standard Sencha naming convention outlines that all subnamespaces should be lowercase, with all class names being upper camel case. For example, `BizDash.view.users.UserForm` is preferred over `BizDash.view.Users.userForm`.

We will now instantiate this class using the `Ext.create` method. This method accepts a class' fully qualified name and an object whose properties and values will be used to configure the class. We will talk about how to add our own configuration options later in this chapter.

Open the application in a web browser and run the following code in the Developer Tools' console:

```
Ext.create('BizDash.config.Config', {});
```

Upon running the code, we should see a console similar to the following screenshot:



In the screenshot, we can see three things:

1. The source file loaded into the page automatically via a GET request
2. The console log from the `Ext.define` callback
3. Our new class instance logged in the console

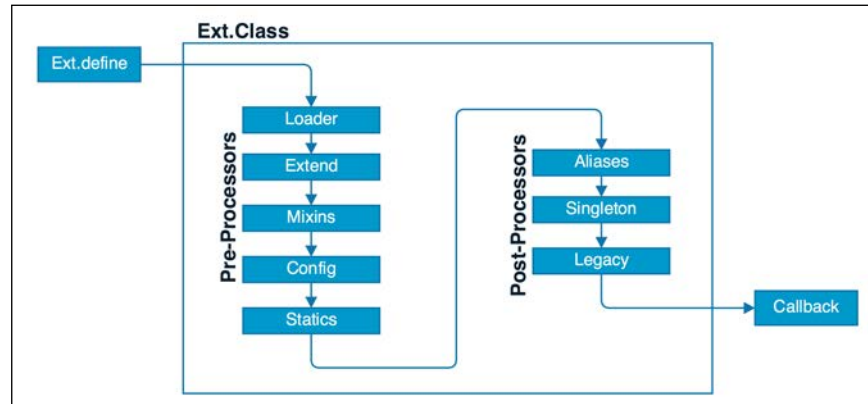
You will have noticed that we were able to instantiate our `BizDash.config.Config` class without ever referencing it in our application or HTML page. This is taken care of by the magic of the `Ext.Loader` class, which we will talk about next.

The magic behind Ext.Loader

In the last section, we saw an example of how class source files can be automatically loaded into the application as and when they are needed. This is taken care of by the `Ext.Loader` class, which analyzes the dependencies of each class defined and ensures every reliant class is loaded in via AJAX. For more details on how this relates to creating production builds, see the *Creating our application with Sencha Cmd* section of *Chapter 1, Getting to Know Ext JS*.

The class definition process

The following diagram shows the process of how a class is defined, starting with the `Ext.define` method call, and how the `Loader` class fits into this process:



Defining dependencies

The best way to ensure that all necessary classes have loaded into the page is to build a dependency tree using the `requires` configuration option. This option accepts an array of fully qualified class names, which are then all loaded into the page before the class definition is deemed complete. Upon loading a dependent class, the process is repeated with the new class where all of its dependencies are loaded until all required classes are present.

We can extend our `Config` class by introducing a dependency in a new class called `BizDash.config.Constants`:

```
Ext.define('BizDash.config.Constants', {
},
function() {
    console.log('BizDash.config.Constants defined.');
```

```
});
```

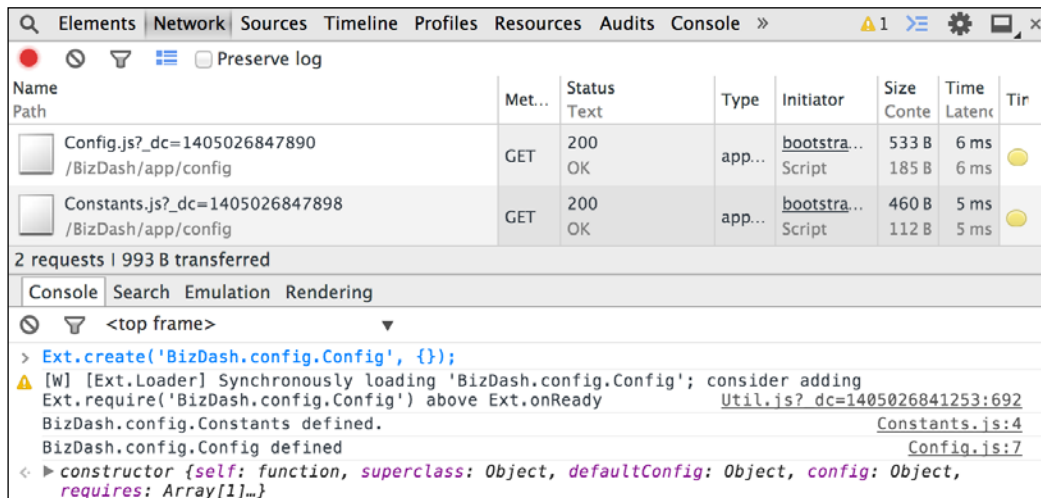
We want our `Config` class to make use of this class, so we must ensure it is available to be instantiated when the class is defined. We do this by adding the `requires` configuration to the class definition, telling the framework to load it in automatically:

```
Ext.define('BizDash.config.Config', {
    requires: ['BizDash.config.Constants']
},
```

```
function() {
  console.log('BizDash.config.Config defined');
});
```

With this line in place, we can use `Ext.create` to instantiate our class again and see the `Constants` class being loaded in automatically:

```
Ext.create('BizDash.config.Config', {});
```



Loader paths

By default, the root namespace of an app (in our case `BizDash`) is mapped to the app folder. This means that any subnamespace is mapped directly to a folder inside the app folder and the class name to the JavaScript file itself. It is important to keep to these naming conventions so that these source files can be found.

If you want to have the application load files from a different location, you can specify a custom path and namespace convention for the loader to follow, by adding a call to the `Ext.Loader.setConfig` method to the top of your `Application.js` file.

The default `paths` object map `Ext` and the application namespace, as the following snippet shows:

```
Ext.Loader.setConfig({
  paths: {
    Ext      : '../ext/src',
    BizDash  : 'app'
  }
});
```

You can customize this by adding your own namespace as the key (this can be more than one portion of a namespace; for example, `Custom.namespace`), and the path to the correct folder as the value:

```
Ext.Loader.setConfig({
  paths: {
    Ext           : '../ext/src',
    BizDash       : 'app',
    'Custom.path' : '..CustomClasses/path'
  }
});
```

Ext.Loader and Sencha Cmd

As we have seen, Sencha Cmd can concatenate all the required class files into a single file. This is achieved by producing this dependency graph and combining all the source files in the order that they are loaded when launching the application. This ensures that only the classes that are actually used are included in the application's final source file, making it much smaller than if the entire framework were always included.

Our dependency root

In our examples, we have forced our `Config` to be loaded via the `Ext.create` call, which synchronously loads the source file. By relying on this to load class files, the dependency tree is only established and fulfilled during runtime. This will mean Sencha Cmd will be unable to create a fully concatenated source file and will be forced to load missing classes at runtime.

To combat this, we should include our root classes as required classes in the `Application.js` file. This file is the entry point for the application, so from here we can establish the dependency graph all the way to the extremities, before the app is launched.

To ensure our `BizDash.config.Config` class is loaded and ready for use upon application launch, we would add it inside a `requires` array in the `Application.js` file:

```
Ext.define('BizDash.Application', {
  extend: 'Ext.app.Application',
  name: 'BizDash',
  requires: [ 'BizDash.config.Config' ],
  ...
});
```

Adding class members

So far, we have looked at defining and creating classes and getting these classes loaded into our page. We will now explore how to make them useful by adding class members such as properties, methods, and statics.

Properties

Public properties can be added to a class by including a key-value pair to the class' definition object. The following code example shows how we add a version number property to our `Config` class:

```
Ext.define('BizDash.config.Config', {
  requires: ['BizDash.config.Constants'],
  version: '0.0.1-0'
},
function(){
  console.log('BizDash.config.Config defined');
});
```

This property can be accessed by simply using the dot notation in an instance of the `Config` class; for example:

```
var config = Ext.create('BizDash.config.Config', {});
// logs 0.0.1-0 console.log(config.version);
```

Methods

Methods can be added in an identical way to properties. We can add a `getBuildNumber` method to the `Config` class, which will extract the build number from the version property, as shown here:

```
Ext.define('BizDash.config.Config', {
  requires: ['BizDash.config.Constants'],
  version: '0.0.1-0',
  getBuildNumber: function(){
    var versionSplit = this.version.split('-');
    return versionSplit[1];
  }
},
function() {
  console.log('BizDash.config.Config defined');
});
```

This method can be executed in the normal way:

```
// logs "0"
console.log(config.getBuildNumber());
```

Note that by default, the methods are executed in the scope of the class instance, allowing us to access the version property.

Statics

The Ext JS class system provides us with the ability to include static properties and methods to our classes to remove the need to create class instances. These can be added inside an object assigned to the `statics` configuration option.

The following example shows a static property called `ENVIRONMENT` added to the `BizDash.config.Constants` class to track if we are in a development or production environment:

```
Ext.define('BizDash.config.Constants', {
    statics: {
        ENVIRONMENT: 'DEV'
    }
},
function() {
    console.log('BizDash.config.Constants defined.');
```

We can now access this static property using the following code:

```
// logs "DEV"
console.log(BizDash.config.Constants.ENVIRONMENT);
```

Ext JS' naming conventions says that static property names should always be in uppercase.

Statics in subclasses

If you would like your class' static properties to be available in any subclasses, then they must be defined using the `inheritableStatics` property.

Singletons

As an alternative to using static methods, classes can be defined as singletons, which will, after successful definition, instantiate the class and assign it back to the class property. This can be done by simply adding the `singleton: true` configuration to the class:

```
Ext.define('BizDash.config.Config', {
    singleton: true,
    ...
},
function() {
    console.log('BizDash.config.Config defined');
    // logs "true"
    console.log(BizDash.config.Config.isInstance);
});
```

The logging of the `isInstance` property, which is available in all class instances, shows that once defined, the `BizDash.config.Config` property is now an instance of the class.

Extending classes

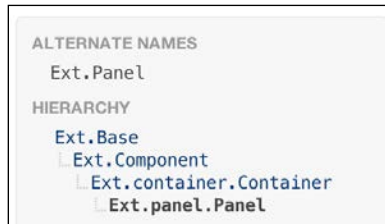
The Ext JS framework's architecture means the functionality is built up and shared using inheritance. It is extremely easy to have one of our custom classes or views inherit from another or from an existing framework class.

For example, when defining your view classes, you will likely extend from the base `Ext.Component` class or from one of its subclasses, such as `Ext.panel.Panel` or `Ext.grid.Panel`.

We define our class' superclass by including the `extend` configuration and giving it the name of the class to extend. In the following example, we will extend our `Config` class from the `Ext.util.Observable` class, allowing it to raise its own custom events. This will give our `Config` class access to all of the methods and properties of this class, such as the `on` method:

```
Ext.define('BizDash.config.Config', {
    extend: 'Ext.util.Observable',
    ...
}, function() {
    // logs "function(){..}"
    console.log(BizDash.config.Config.on);
});
```


The Ext JS documentation shows the inheritance tree of each class at the top right of its documentation page, as shown here, for the `Ext.panel.Panel` class:



When omitted, as in our original `Config` class, the class will extend from the `Ext.Base` class by default, which is the root base class for all of the framework's classes.

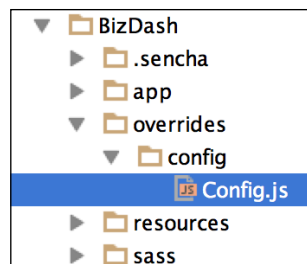
Overriding classes

It is sometimes necessary to change the functionality of a class without directly modifying the original source. We can do this by defining a new class along with the `overrides` configuration option to tell Ext JS which class to modify.

This technique should always be used when altering the framework behavior (for example, to fix a bug in the framework's code) as the Ext JS code should never be modified directly. This will come back to bite you when making framework upgrades in the future.

In the following example, we will override our `getBuildNumber` method so that it returns the build number with a prefix of **Build Number:**. You will have seen that Sencha Cmd has already created an `overrides` folder in our application structure. This is where we will put our class override files.

Inside this folder, we create a new `config` directory to mirror our main app folder structure and define a new class named `Overrides.config.Config` in a file called `Config.js`.



The override follows the same structure as a normal class definition, as shown here:

```
Ext.define('Overrides.config.Config', {
    override: 'BizDash.config.Config',
});
```

We can now add in our override for the `getBuildNumber` method, which uses the `callParent` method to execute the original method and combine its output with our label:

```
Ext.define('Overrides.config.Config', {
    override: 'BizDash.config.Config',
    getBuildNumber: function() {
        return 'Build Number: ' + this.callParent(arguments);
    }
});
```

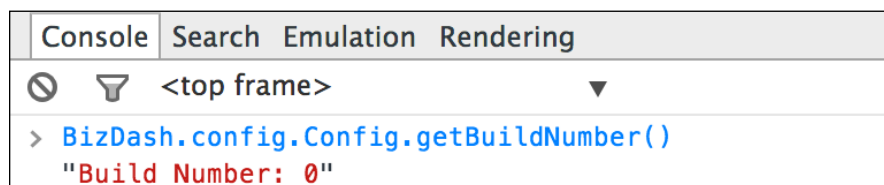
Requiring our override class

There is one extra step required to include this override in our app and for the changes to take effect. We must tell the application where our override classes reside so `Ext.Loader` knows where to find them. We do this by adding the `overrides` path to our `sencha.cfg` file so that Sencha Cmd can update the app's dependencies. We do this by adding the following line to the bottom of the `BizDash/.sencha/app/sencha.cfg` file:

```
app.overrides=${app.dir}/overrides
```

We can then run the `sencha app refresh` command from within the `BizDash` folder to regenerate the app's bootstrap files.

If we now reload our application and call the `getBuildNumber` method, we should see the output as **Build Number: 0**.



Although this override is overriding one of our own custom methods, an identical process can be followed to override any of the framework's own classes whose behavior you would like to change.

Targeting overrides to framework versions

It is now possible to target overrides to specific framework or package versions by using the compatibility configuration. This can accept a single string containing a version number, an array of version strings that will be matched with an OR operator, or an object using the `and` or `or` keys to create more complex matches. The following examples show how these can be used:

```
/* matches Ext JS 5.0.0 compatibility: '5.0.0'
matches Ext JS 5.0.0 OR Ext JS 4.2.1 compatibility: ['5.0.0', '4.2.1']
matches the 'Ext JS' package with 5.0.0 and 'Sencha Core' with 5.0.0
*/
compatibility: {
  and: [
    'extjs@5.0.0',
    'sencha-core@5.0.0'
  ]
}
```

If the compatibility option is a match, then the override is included; otherwise it is not.

For full details of what version expressions can be used, see the documentation on the `Ext.checkVersion` method. <http://docs.sencha.com/extjs/5.1/5.1.0-apidocs/#!/api/Ext-method-checkVersion>.

Configuring classes

Ext JS offers us a useful way of creating configurable properties, which gives us an autogenerated process to get and set their values, and also to perform any updates required in other areas; for example, reflecting the updated value in the UI.

When a class is defined, any properties found in the `config` object become completely encapsulated from other class members and are given their own getter and setter methods.

For example, if we move our `Config` class' `version` property into a `config` object, as shown in the following code snippet, the class will be given two new methods named `getVersion` and `setVersion`. Notice that we must create a constructor function and call the `initConfig` method to have the class system initialize these new methods. We also must update our `getBuildNumber` method, which references the version number using `this.version`, to use the new getter method `getVersion`.

```
Ext.define('BizDash.config.Config', {
  extend: 'Ext.util.Observable',
```

```

    singleton: true,
    requires: ['BizDash.config.Constants'],
    config: { version: '0.0.1-0' },
    constructor: function(config){
        this.initConfig(config);
        this.callParent([config]);
    },
    getBuildNumber: function() {
        var versionSplit = this.getVersion().split('-');
        return versionSplit[1];
    }
}, function() {
    ...
});

```

In Ext JS 5, the call to `initConfig` is now not required when extending the `Ext.Component` class or one of its subclasses, as it is called internally.

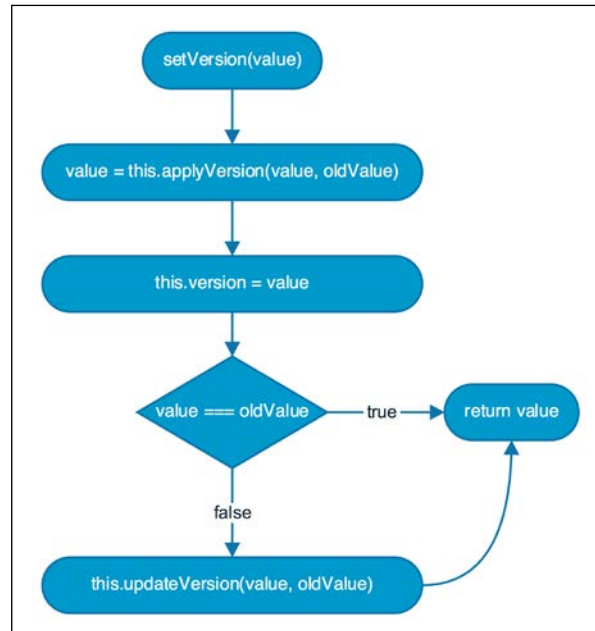
Setting a configuration value

When giving the `config` option a new value, the generated setter method goes through a more complex process than just assigning the given value to a property in the class. It introduces the concept of `applier` and `updater` methods, which are optional and, if present, called in turn within the setter method.

The `applier` function should be named with the same convention as the getter and setter, with the name being prefixed with `apply` and the property name's first letter in uppercase; for example, `applyVersion`. This method is used to transform the given value as required before it is stored in the class. Potential usage may be to perform a lookup of the value to get its real instance (for example, a store instance from its ID). This method must return a value; otherwise the property won't be updated.

The `updater` function follows the same naming pattern (for example, `updateVersion`) and is called after the value has been transformed with the `applier` and set in the class. This function is primarily used to update the UI to reflect the latest value within a component.

The following diagram shows how this process works when calling the generated setter method:



The following example shows how we use and apply the method to ensure the format of the version number; and an update method to raise a custom event, allowing other areas of the app to be notified of the change:

```
Ext.define('BizDash.config.Config', {
    extend: 'Ext.util.Observable',
    singleton: true,
    requires: ['BizDash.config.Constants'],
    config: {
        version: '0.0.1-0',
        ...
    },
    ...
    applyVersion: function(newVersion, oldVersion){
        return newVersion;
    },
    updateVersion: function(newVersion, oldVersion){
        this.fireEvent('versionchanged', newVersion, oldVersion);
    }
}, function() { ... });
```

Overriding defaults

The default values given to configuration properties can be easily overridden when a new class instance is created, by simply adding the property and the desired value to the configuration object passed to the `Ext.create` method. The following snippet shows the version option being set (assuming the `Config` class wasn't set to be a singleton):

```
var config = Ext.create('BizDash.config.Config', {
    version: '0.2.0-0'
});
// logs "0.2.0-0"
console.log(config.getVersion());
```

Platform-specific configs

Ext JS 5 introduces the ability to configure a class differently based on the platform the application is being run on, so we can tailor the experience based on the platform's capabilities.

This is done by using the `platformConfig` config option, giving it an array of configuration objects. These configuration options must contain a platform property, which will be used to find the appropriate config to use on the current platform.

This option should contain an array of strings describing the platform the config should target. This can be one or more of the following options: phone, tablet, desktop, iOS, Android, Blackberry, Safari, Chrome, or IE10.

If any of these platforms match the current one, then the other properties will be merged into the class configuration. It is possible that multiple rules will evaluate to true, in which case the properties of all matched rules will be applied.

The following code shows how we can configure our `Config` class differently based on the platform:

```
Ext.define('BizDash.config.Config', {
    extend: 'Ext.util.Observable',
    singleton: true,
    requires: ['BizDash.config.Constants'],
    config: {
        version: '0.0.1-0',
        isPhone : false,
        isTablet : false,
        isDesktop: false
    },
});
```

```
platformConfig: [ {
  platform: ['phone'],
  isPhone : true
},
{
  platform: ['tablet'],
  isTablet: true
},
{
  platform : ['desktop'],
  isDesktop: true
}]
...
});
```

We can then access these properties using their getter methods. The following code, run on a laptop, will output `true` for the `isDesktop` property and `false` for the other two:

```
// logs "true", "false", "false"
console.log(BizDash.config.Config.getIsDesktop());
console.log(BizDash.config.Config.getIsPhone());
console.log(BizDash.config.Config.getIsTablet());
```

Summary

In this chapter, we have covered all aspects of the Ext JS class system, including:

- Defining and instantiating classes
- Adding properties, methods, and statics
- Extending other classes and overriding the framework behavior
- Configuring classes and making use of `applier` and `updater` methods

We have also demonstrated how to ensure our class files are loaded into our application when needed, using the `Ext.Loader` class.

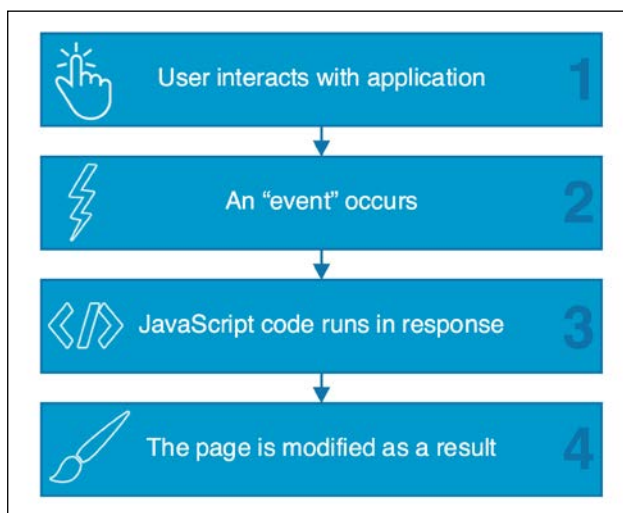
The next chapter will discuss how Ext JS handles events, originating both from user input and from other classes.

3

Reacting to User and Application Interactions

Developing effective applications with Ext JS requires a solid understanding of event-driven programming and how actions are performed, based on user and application interactions. Ext JS is an event-driven framework and uses events to control the flow of the application.

Events can be raised by user input, internally by the framework or by our own application code. For example, when a user clicks a button with their mouse, a click event will be fired by the button's instance. We can then attach a listener to this event and execute our handler code when it is fired. Have a look at the following diagram:



This chapter will explore in detail the events in Ext JS. The topics covered in this chapter include:

- Listening for events
- Raising custom events
- Attaching event handlers to components
- Listening for user input via mouse, keyboard, and touch screens

Background

Applications built with Ext JS will make use of events at numerous points throughout their lifecycle. Even if you're unaware of it, in the background, the framework will raise events when something interesting happens. As developers, we want our application code to respond to these events, either to process something or to give the user feedback on what has happened.

Ext JS implements an `Ext.mixin.Observable` class that provides a common interface for publishing events. Let's look at the options we have to listen to events.

Defining event handlers in config objects

A common way to define listeners is to use the `listeners` config option, which allows us to define an object containing event handlers. This object should be defined in the `config` object of the Ext JS component.

Let's jump straight into the `BizDash.view.main.Main` class that was created automatically when we generated our application in *Chapter 1, Getting to Know Ext JS*.

We will bind an event listener to the `afterrender` event of the Tab Panel component. The function is executed when the event is fired. In this case, the output of the handler is a simple console message:

```
Ext.define('BizDash.view.main.Main', {
    extend: 'Ext.container.Container',
    ...
    items: [{
        region: 'center',
        xtype: 'tabpanel',
        listeners: {
            afterrender: function(component, eOpts) {
                console.log('Center tabpanel has rendered.')
```

```

    }
  }
}]
});

```

It is also possible to attach handlers to multiple events at once. This example has both, a `beforerender` and `afterrender` handler for events fired by the Tab Panel component:

```

Ext.define('BizDash.view.main.Main', {
  extend: 'Ext.container.Container',
  ...
  items: [{
    region: 'center',
    xtype: 'tabpanel',
    listeners: {
      afterrender: function(component, eOpts) {
        console.log('Center tabpanel has rendered.')
      },
      beforerender: function(component, eOpts) {
        console.log('Center tabpanel before rendering.')
      }
    }
  ]
});

```

Button handlers

The framework provides a shortcut to define a handler to a button as it's highly likely you'll want your buttons to respond to a mouse click. The handler in this case is bound to the `onClickButton` method in the view's View Controller, `BizDash.view.main.MainController`:

```

Ext.define('BizDash.view.main.Main', {
  extend: 'Ext.container.Container',
  ...
  items: [{
    ...
    tbar: [{
      text: 'Button',
      handler: 'onClickButton'
    }]
  ]
});

```

It's considered good practice to keep your business logic out of views instead of putting it in global controllers or ViewControllers.

The on method

Alternatively, we can use the `on` method, which is an alias of the `addListener` method. This method comes from the mixed in `Ext.mixin.Observable` class and enables us to add listeners to a class, or component after the class is instantiated:

```
Ext.define('BizDash.view.main.MainController', {
    extend: 'Ext.app.ViewController',
    ...
    init: function () {
        var button = this.getView().query('button[text="Button"]')[0];
        button.on('mouseover', 'onMouseOver');
        button.on({ mouseover: 'onMouseOver' });
    },
    onMouseOver: function () {
        console.log('Button Mouseover Event Fired');
    }
});
```

The preceding example shows how to use an Ext JS Component Query to search for the button in our application and add a `mouseover` listener bound to the method `onMouseOver` in our `ViewController`. We will go into Component Queries in more detail in *Chapter 7, Constructing Common UI Widgets*.

As with the listeners config object, the `on` method also accepts an alternative parameter set that allows multiple event handlers to be assigned at once. By providing a JavaScript object as the first parameter, with name/value pairs specifying the event name and its handling function, the listeners will all be assigned at once. By defining a `scope` property within this object, the handler functions will all be executed within the scope of the specified object (or what `this` will refer to within the function):

```
Ext.define('BizDash.view.main.MainController', {
    extend: 'Ext.app.ViewController',
    ...
    init: function () {
        var button = this.getView().query('button[text="Button"]')[0];
        button.on({
            mouseover: 'onMouseOver',
            mouseout: 'onMouseOut',
            scope: this
        });
    }
});
```

```

    },
    onMouseOver: function () {
        console.log('Button Mouseover Event Fired');
    },
    onMouseOut: function () {
        console.log('Button Mouseout Event Fired');
    }
});

```

Scope defaults to the object that fires the event. In our case, that would be the button. You can customize this with the `scope` option if you require the handler function to be executed in a different scope. The preceding example demonstrates this using the `this` reference to alter the scope to that of the `BizDash.view.main.MainController`.

Listener options

There are numerous listener options for you to configure as well. For example, you can buffer an event that fires in quick succession or target an event to a specific element rather than the entire component. Event delegation like this is useful during component construction to add DOM event listeners to elements of components, which will exist only after the component is rendered.

```

Ext.define('BizDash.view.main.MainController', {
    extend: 'Ext.app.ViewController',
    ...
    onClickButton: function () {
        Ext.Msg.confirm('Confirm', 'Are you sure?', 'onConfirm', this);
        this.getView().getButton().disable();
    },
    ...
    init: function () {
        var button = this.getView().query('button[text="Button"]')[0];
        button.on({
            mouseover: 'onMouseOver',
            mouseout: 'onMouseOut',
            click: {fn: 'onClickButton', single: true},
            scope: this
        });
    },
    ...
});

```

This example shows the `single` option added to the click event. This option automatically removes the click event after the first time it's fired. We've further enhanced the `onClickButton` method to disable the button too.

The documentation for `Ext.mixin.Observable` contains useful examples and further information on configuring events. You can find it at <http://docs.sencha.com/extjs/5.1/5.1.0-apidocs/#!/api/Ext.mixin.Observable>.

Firing events

The `Ext.mixin.Observable` class also provides a way to fire events, whether these are framework events or custom events. The `fireEvent` method will fire any event we require and pass parameters for consumption by the handler function. The following example shows us how to fire a custom confirmed event, passing the choice parameter on the button, and binding it to an `onConfirmed` handler:

```
Ext.define('BizDash.view.main.MainController', {
    extend: 'Ext.app.ViewController',
    onConfirm: function (choice) {
        if (choice === 'yes') {
            var button = this.getView().getButton();
            button.fireEvent('confirmed', choice)
        }
    },
    onConfirmed: function(choice){
        console.log('The CONFIRMED event was fired');
    },
    init: function () {
        button.on({
            mouseover: 'onMouseOver',
            mouseout: 'onMouseOut',
            click: {
                fn: 'onClickButton',
                single: true
            },
            confirmed: 'onConfirmed',
            scope: this
        });
    }
});
```

Listening for events on elements

As some events, for example click, aren't available on all components, it's possible to attach event handlers directly to any element. The `Ext.dom.Element` class, a framework class that wraps DOM elements, will relay all of the underlying DOM events, and its documentation contains a full list of these.

```
Ext.define('BizDash.view.main.MainController', {
    extend: 'Ext.app.ViewController',
    init: function () {
        var el = this.getView().getEl();
        el.on('tap', function() {
            console.log('The Viewport was tapped/clicked.');
```

The preceding example shows how to listen for a tap event on the entire Viewport.

Event delegation

Event handlers, however, are a common cause of memory leaks and can cause performance degradation when not managed carefully. The more event handlers we create, the more likely we are to introduce such problems; so, we should try to avoid creating huge numbers of handlers when we don't have to.

Event delegation is a technique where a single event handler is created on a parent element, which leverages the fact that the browser will bubble any events raised on one of its children to this parent element. If the target of the original event matches the delegate's selector, then it will execute the event handler; otherwise nothing will happen.

This means that instead of attaching an event handler to each individual child element, we only have to create a single handler on the parent element, and then, within the handler, query which child element was actually clicked and react appropriately. To achieve this, we use the `delegate` option available to the `listeners` config.

The following example shows how you might use event delegation with an element containing multiple links:

```
/* assume navigationEl is an Ext.Element instance containing multiple
<a> tags */
navigationEl.on('click', function(e){
```

```
/* Handle a click on any element inside the 'navigationElement'. Use
e.getTarget to determine which link was clicked.*/

}, {
  delegate: 'a'
});
```

Mouse events

The mouse events the framework can handle are `mousedown`, `mousemove`, `mouseup`, `mouseover`, `mouseout`, `mouseenter`, and `mouseleave`. The `Ext.event.Event` class handles the cross-browser and cross-device differences for us to ensure our application behaves the same on all supported browsers.

Keyboard events

The `Ext.event.Event` class also provides a list of key constants:

```
var constants = {
  BACKSPACE: 8,
  TAB: 9,
  NUM_CENTER: 12,
  ENTER: 13,
  RETURN: 13,
  SHIFT: 16,
  ...
}
```

We can, for instance, get a reference to the enter key's code using `Ext.event.Event.ENTER`.

KeyMap

The `Ext.util.KeyMap` class is used to bind keyboard strokes to a handling function. Using this, you can enable the user to control the application using their keyboard:

```
Ext.define('BizDash.view.main.MainController', {
  extend: 'Ext.app.ViewController',
  init: function () {
    var map = new Ext.util.KeyMap({
      target: this.getView().getEl(),
```

```
        key: Ext.event.Event.ENTER,
        fn: this.onEnterPress,
        scope: this
    });
},
onEnterPress: function() {
    console.log('ENTER key was pressed');
}
});
```

Touch events

Not only do we often have to support multiple browsers, but most of the time we need our applications to be device agnostic. Ext JS enables us to support users with other types of pointers, such as a mouse, pen, or finger.

Ext JS 5 provides support for `touchstart`, `touchmove`, and `touchend` events.

Event normalization

To support touch-screen devices, the framework automatically translates touch-screen events into their equivalent mouse events for us. This is called event normalization.

As developers, we don't need to worry about the extra coding. All we have to do is consider the event being used by a mouse. For instance, `mousedown` will seamlessly be translated to `touchdown` and `pointerdown` for us.

Gestures

While normalization saves us coding, we still need to understand the gesture the user carries out on our application. Ext JS does almost all the heavy lifting for us. It will interpret gestures such as tap, swipe, drag, and double tap on any element and raise events for us to listen for.

In order to do this, the framework builds upon the Sencha Touch gesture system, which interprets the sequence and timing of three primary events: `touchstart`, `touchmove`, and `touchend`. Ext JS 5 translates these to the equivalent pointer and mouse events (for example, `pointerdown` or `mousedown`) so that gestures are understood regardless of the device being used for input.

For example, gestures such as tap and swipe will work for both touch and mouse input.

Summary

In this chapter, we learned how to work with events in an Ext JS application. You should now feel more comfortable with:

- Listening for events
- Raising custom events
- Attaching event handlers to components
- Listening for user input via mouse, keyboard, and touch screens

The next chapter builds on your knowledge of classes and events to cover the entire application and its architecture. Ext JS 5 now provides support for MVVM as well as MVC. While the chapter's focus is these paradigms, we also explore other considerations it's worth making early in the development cycle.

4

Architecting an Ext JS Application

Whether you're only developing one application or plan to develop many, it's an excellent idea to consider your application architecture well in advance. The architecture is the internal structure of your application and the programming patterns employed within it.

Ultimately, following commonly used patterns provides continuity and consistency in your applications. This gives us four major advantages:

- The framework and your applications are easier to learn
- It takes less time to switch between applications
- Code sharing between applications is possible
- You gain consistency among your build and testing tools

As this chapter covers application architecture with Ext JS, the core topics we're going to look at are:

- Sencha Cmd and how it can be used to help us build our apps
- The **Model-View-Controller (MVC)** architectural pattern
- The newly introduced **Model-View-ViewModel (MVVM)** architectural pattern
- Cross-class communication with an event-driven model
- Considerations for taking your application offline and an offline first design

Ext JS 5 now provides support for both MVC and MVVM application architectures. Essentially, the two patterns split the application, resulting in well-organized code in a well-organized file system.

Getting the most from Sencha Cmd

We described how to get started with Sencha Cmd in *Chapter 1, Getting to Know Ext JS*, but it is possible to do a lot more with it. In this section, we will explore some of its powerful commands and how they can speed up and improve our workflow.

Generating application components

Sencha Cmd can help by generating MVC/MVVM components to speed up the development process and allow us to focus on our application's logic rather than writing repetitive code.

Generating models

To add a model to your application, make `/path/to/MyWorkspace/BizDash` your current directory and run Sencha Cmd, like this:

```
cd /path/to/MyWorkspace/BizDash
sencha generate model User Name:string,Email:string,TelNumber:string
```

This command adds a `model` class in a `User.js` file in the `model` directory. The file looks like this:

```
fields: [
{
  name: 'Name',
  type: 'string'
},
{
  name: 'Email',
  type: 'string'
},
{
  name: 'TelNumber',
  type: 'string'
}
]
});
```

Generating views

Add a view to your application in the same way:

```
cd /path/to/MyWorkspace/BizDash
sencha generate view location.Map
```

This will generate the following files:

- `app/view/Location/`: The folder for the classes implementing the new view
- `Map.js`: The new view
- `MapModel.js`: The `Ext.app.ViewModel` for the new view
- `MapController.js`: The `Ext.app.ViewController` for the new view

The output from the preceding code for `Map.js` is as follows:

```
Ext.define("BizDash.view.location.Map", {
    extend: "Ext.panel.Panel",
    controller: "location-map",
    viewModel: {
        type: "location-map"
    },
    html: "Hello, World!"
});
```

The `ViewController` (`MapController.js`) is:

```
Ext.define('BizDash.view.location.MapController', {
    extend: 'Ext.app.ViewController',
    alias: 'controller.location-map'
});
```

There are no required parameters in this case beyond the view name. You can, however, add a base class if desired:

```
cd /path/to/MyWorkspace/MyApp
sencha generate view -base Ext.tab.Panel location.Map
```

This will change the `extend` used by the view class to `Ext.tab.Panel`.

Generating controllers

In Ext JS 5, each view generated by Sencha Cmd has a default `Ext.app.ViewController`, so it is not necessary to generate global controllers based on `Ext.app.Controller` in most cases. If you need a new controller, you can generate one in the same basic way as with Models and Views:

```
cd /path/to/MyWorkspace/BizDash
sencha generate controller Location
```

This will generate a file called `Location.js` in the controller directory with the following content:

```
Ext.define('BizDash.controller.Location', {  
    extend: 'Ext.app.Controller'  
});
```

Upgrading your application

As framework enhancements, features, and bug fixes are implemented, you may find yourself in a position where you wish to upgrade your application to a newer version of the framework. You can do this with the following command:

```
sencha app upgrade path/to/new/framework
```

This command will upgrade both, the Sencha Cmd scaffold and the framework used by the application. Complete instructions on upgrading your application are available in the framework documentation.

It is worth noting that reversing an upgrade operation isn't possible with Sencha Cmd. We recommend ensuring you are in a position to revert the upgrade changes in your version control system before starting.

Refreshing application metadata

The following command regenerates the metadata file containing *bootstrap* data for the dynamic loader and class system. This must be done any time a class is added, renamed or removed.

```
sencha app refresh
```

Triggering automatic builds

The `watch` command is extremely useful as it watches your code base for changes (edits, deletes, and so on) and triggers a rebuild of the application to speed up the development process.

When you run the following command, a web server is started to host the application:

```
sencha app watch
```



The default port of the web server is 1841.

MVC and MVVM

The next part of the chapter is going to focus on the architectural patterns that work best with Ext JS applications. Sencha was the pioneer of MVC in web applications when it introduced the latter in the early versions of Sencha Touch. Since then, MVC (and MVVM) has gained traction and popularity in the web development community as web apps become larger, more complicated, and harder to maintain. One of the main purposes of these application architectures is to provide structure and consistency to your code base. Nowadays, most major frameworks support them, and the same is true of Ext JS.

We will explain what MVC and MVVM are, their pros and cons, and how they work in a typical Ext JS application.

What is MVC?

Model-View-Controller (MVC) is an architectural pattern for writing software. It splits the user interface of an application into three distinct parts, which helps organize the code base into logical representations of information, depending upon the function. In an Ext JS application, the end result of this paradigm is to have well-organized code and a well-organized filesystem.

Explaining the abbreviations

Sometimes, MVC implementations vary slightly between applications, but in general, each part of the architecture has specific responsibilities. In MVC architecture, every object in the program is a model, a view, or a controller.

Model

The Model represents the data we are planning to use within the application. It describes a common format for the data – in most cases simple fields – but it may also contain business rules, validation logic, conversions, formatting rules, and various other functions.

View

The View visually represents the data to the user. It is defined using the standard JSON configuration of Ext JS by extending a framework component/widget. For example, a typical view may be a grid, a form, or a chart.

It is possible that more than one view may display the same data in different ways. For example, a chart and a grid look visually different even though they share the same data.

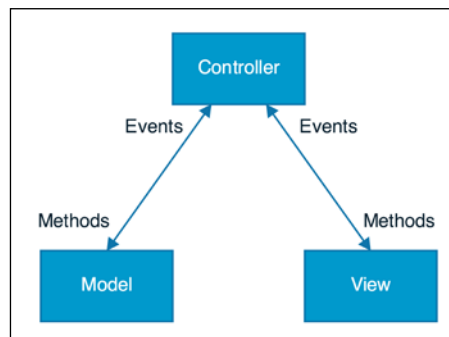


Best practices dictate that little, if any, business logic exists in the view.

Controller

The Controller is the central piece of an MVC application. It is a singular construct and responds to the events in the application and delegates commands between the Model and the View. Due to the clear separation of concerns in MVC, a controller acts as a global message bus listening for events on configured components.

The following diagram will give you a clearer picture:



Putting this together

In your application, a user will interact with the views, which often contain data that is held in the models. The controller plays the pivotal role of monitoring the various interactions in the view and making the necessary updates to the model or view.

Controllers hold almost all of the business logic of the application, leaving the views and models mostly unaware of each other. You can think of it as a Publish-Subscribe model for an application.

Ext JS has specific classes to manage controllers and models, namely `Ext.app.Controller` and `Ext.data.Model`. The views should be defined by extending framework widgets.

It's a good idea to define your model first, as it contains the data you plan on using in the application. Simply extending the `Ext.data.Model` class with basic field configurations is all that is required to get you started.

Following this, create a view by extending a component or widget. Try to avoid putting business logic in the view; instead, put the logic in the controller. For example, a button in a view should not contain event logic – it should be placed inside the controller.

Finally, create a controller by extending `Ext.app.Controller`. The controller may not be aware of the view, and in many cases, you will be required to manually get a reference to the view before you can do more work. Sencha's documentation on refs (<http://docs.sencha.com/extjs/5.1/5.1.0-apidocs/#!/api/Ext.app.Controller-cfg-refs>) and Component Query (<http://docs.sencha.com/extjs/5.1/5.1.0-apidocs/#!/api/Ext.ComponentQuery-method-query>) explains this in more detail.

Ext JS naming convention and directory structure

Sencha uses a clearly defined naming convention to keep all our files together. For example, as was explained in *Chapter 2, Mastering the Framework's Building Blocks*, the class `BizDash.view.Main` correlates to a location in your filesystem. In this case, the file is called `Main.js` within the `view` directory.

Ensure that you store your models in the `model` directory, views in the `view` directory, and controllers in the `controller` directory.

The benefits and drawbacks of using MVC

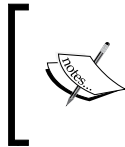
There are a number of advantages of using MVC, which include:

- Consistency across applications, which reduces learning time
- Ease of sharing code between applications
- Ability to build optimized apps using Sencha's build tools when using MVC

However, the greatest advantage of MVC architecture is that it helps developers avoid writing excessively large files that are difficult to maintain. By clearly splitting the responsibilities of each piece of the application, the classes are stored in a directory structure that's consistent and easy to work with.

Sadly, MVC architecture does come with its drawbacks. Controllers are globally scoped to the application in the Ext JS approach to MVC, which results in additional business logic to grab references to views, models, and other objects. Controllers could be written to watch any object at any time, so any given controller might have logic for both view A and view B, leading to additional confusion in large applications.

Unit testing is another issue that crops up time and time again in MVC applications. In MVC models, views and controllers are supposed to be loosely coupled, but testing a controller necessitates some knowledge of the greater application. More often than not, unit testing requires the entire application to be launched in order to test the individual pieces. This is obviously cumbersome, time-consuming, and brittle.



Don't fall into the trap of building applications with a relatively small number of controllers, each spanning thousands of lines of code. Ultimately, this causes poor performance and long-term maintenance issues.

What is MVVM?

While the MVC architecture clearly does have some major benefits, its drawbacks need addressing. The Model-View-ViewModel (MVVM) architecture is an answer. Sencha only introduced MVVM architecture support in Ext JS 5, so you will not be able to apply these principles to your Ext JS 4 or Sencha Touch 2 applications.

Ext JS 5 still offers support for MVC, so upgrading from Ext JS 4 to 5 will not break your application. Sencha decided to support MVVM in Ext JS 5 to address the drawbacks of maintainability and testing.

Addressing the concerns

In theory, global controllers are very useful, but as you can see, they can be difficult to manage in practice. MVVM overcomes this by introducing a new class called `ViewModel`, which manages the data specific to the view. It does this using data bindings. This means we write less code, it's easier to maintain, and a lot easier to test.

Explaining the abbreviations

Just like MVC, MVVM is another architectural pattern for writing software. It's based on the MVC pattern, so a lot of this should be familiar.

Model

The principle of a Model in MVVM architecture is the same as in MVC.

View

As with the Model, the View is also the same in MVVM as it is in MVC. The only difference is that we must set up our data bindings for the view. This is done by adding a ViewModel to the view:

```
Ext.define("BizDash.view.location.Map", {  
    extend: "Ext.panel.Panel",  
    viewModel: {  
        type: "location-map"  
    }  
});
```

Unlike MVC, MVVM architecture tightly couples views to their associated ViewModels and ViewControllers.

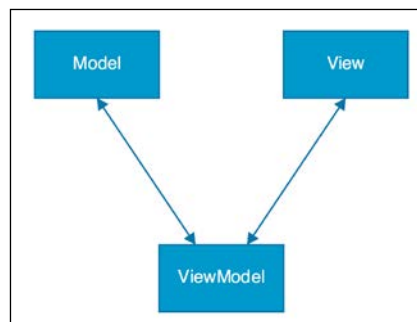
ViewModel

The ViewModel is the key difference between MVVM and MVC. In MVC, using events, our Controller is responsible for managing the communication between a Model and a View. In MVVM, the framework does the heavy lifting behind the scenes and does this using data binding.

Data binding is simply a mechanism to connect the user interface with the business logic. For instance, when the values in the UI are changed, the underlying data value in the model also changes.

The result of introducing ViewModels is that the Model and framework perform much more work than before, which minimizes the application logic required to manipulate the View.

Have a look at the following diagram:



A typical ViewModel may look like this:

```
Ext.define('BizDash.view.location.MapModel', {
    extend: 'Ext.app.ViewModel',
    alias: 'viewmodel.location-map',
    data: {
        name: 'Map Location'
    }
});
```

ViewModels provide a bridge between the Models' data and its visual representation. They are tied closely to the views and provide the data that they represent.

Business logic

There's still the question of application logic, however. There are two options for this, but in general, your business logic should go in ViewControllers.

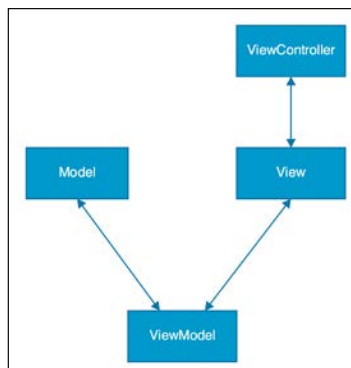
ViewControllers

A ViewController is very similar to a Controller, as it still uses a Publish-Subscribe model by listening for events. Where it differs is in how it's coupled to the view.

A ViewController is:

- Scoped directly to a View
- Has a one-to-one relationship
- Created for every instance of the View

This drastically reduces overheads, as your application doesn't have as many events and component references floating about. Memory leaks and state management are easier to identify and maintain, as the ViewController is tied to the View that referenced it.



Your ViewController will start off looking like this:

```
Ext.define('BizDash.view.location.MapController', {  
    extend: 'Ext.app.ViewController',  
    alias: 'controller.location-map'  
});
```

The alias defined here can be used in a View definition's controller configuration to tie the two classes together.

Controllers

You can, however, continue to use controllers for your application-wide message bus. They will continue to listen for events in multiple views just like in the MVC architecture.

Cross-class communication with events

Your Ext JS application makes use of events to handle user interaction, but it is also easy to have your classes communicate with each other via events.

Ext JS is very mature in the way it handles events, as it's something that's been at the core of the framework since the beginning. It uses an observer pattern to allow your classes to publish events and others to subscribe to those events. Your subscribing class will have its logic triggered as soon as the publishing class fires the event. This paradigm is asynchronous and modular.

As we covered in *Chapter 2, Mastering the Framework's Building Blocks*, the event-driven logic in Ext JS is handled by the `Ext.mixin.Observable` class.

Taking your application offline

When building applications, developers often debate the problem of designing a system that's capable of working offline. Ext JS as a framework is designed to imitate the components and widgets one may typically see in a standard desktop environment. It gives us the tools we need to build truly rich applications and an experience that's similar, if not better, than a legacy desktop application. For web developers, the issue is that users are already used to working with software that's been designed "offline first" The same applies to many mobile or tablet apps.

Take an e-mail client as an example: the desktop clients continue to work even when your data connection is lost. You can read e-mail, search, organize folders, and do much more. On the other hand, the equivalent web-based client is not likely to fare so well. Try it yourself.

Why should we design offline first?

Apart from ensuring that the web is, and remains, a viable solution for delivering applications, there are many advantages of considering taking your applications offline:

- The most important fact is that *we go offline*. It might not be intended and it may not be desirable, but connectivity is sporadic and not always guaranteed. This might result in your user losing data or being unable to complete their job.
- We don't have ubiquitous Internet. We've got to work within the confines of our mobile and fixed line network providers.
- Mobile, remote working and accessing systems on the move are more popular than ever before. This trend is continuing, and most developers are required to consider this when developing new applications.
- Performance is greatly enhanced as the user does much of the work locally, leaving your servers for other, more important tasks.
- Reliability and trust are improved, as your users see the application as something that doesn't come with the usual flaws.
- Finally, robustness is improved, as, for example, your server downtime doesn't necessarily mean application downtime.

To summarize, an offline first approach will enable you to deliver a better user experience.

What can we do about this?

Different apps have different approaches and the difficulty rating of each is different. Common approaches, in order of simplest to most complex, are to:

- Warn the user they are offline
- Provide the user cached data
- Allow users minimal interaction
- Allow users full interaction with a complex application

How can we do this?

There are a number of ways to achieve this in web applications, and choosing the right method for your application isn't straightforward.

- Native packaging is an option, similar to that of hybrid mobile apps. It is possible to package your web applications with tools such as Embedded WebKit, Cordova, or Chrome/Firefox apps.
- Using the web app manifest to define the details and APIs present in your app.
- Using AppCache is useful to have the browser cache files and resources.
- ServiceWorkers are useful if you need to do background processing; for example, data syncing. It's still early days for ServiceWorkers, but these might be ideal for many JavaScript developers.
- LocalStorage is great for storing data in key-value pairs. Ext JS provides excellent support for working with LocalStorage, but be wary of its browser enforced storage limitations—typically around 5 MB.
- IndexedDB or WebSQL are other ways storing application data in the client side. WebSQL has been deprecated but IndexedDB is an extremely viable alternative and has excellent support from Ext JS.

Offline architecture

There's a lot to consider, and it goes without saying that clearly, an offline web application requires a different architecture.

You should always plan for the worst and hope for the best.

One solution to the problem is to put all the state in the client and then sync it whenever possible. Essentially, you want to design your application to download and store files in cache on the user's hard disk and interact with locally stored data. In the background, the locally stored data is synced with your server through some form of proxy.

Syncing data

Syncing data with your backend is perhaps the trickiest part of developing an application with the offline first principle. Writing a sync protocol is a difficult and time-consuming process. You should consider using frameworks and tools, such as Hoodie, PouchDB, and remoteStorage.io to alleviate the problem that syncing causes.

In order to get the best results, we recommend that you follow these guidelines:

- Do it often and as soon as possible
- Transfer the minimal amount of data you can get away with
- Be prepared for unreliable data networks
- Have a strategy to manage conflicts

Summary

This chapter has focused on the principles of application architecture and some of the tools we have available in Ext JS 5 applications. We have covered:

- Sencha Cmd to generate models, views, and controllers
- MVC and MVVM architecture patterns
- Cross-class communication
- Working with offline data

The next chapter will deepen your knowledge further by introducing data packages in detail. A solid understanding of data modeling and stores will stand you in good stead to develop advanced web applications.

5

Modeling Data Structures for Your UI

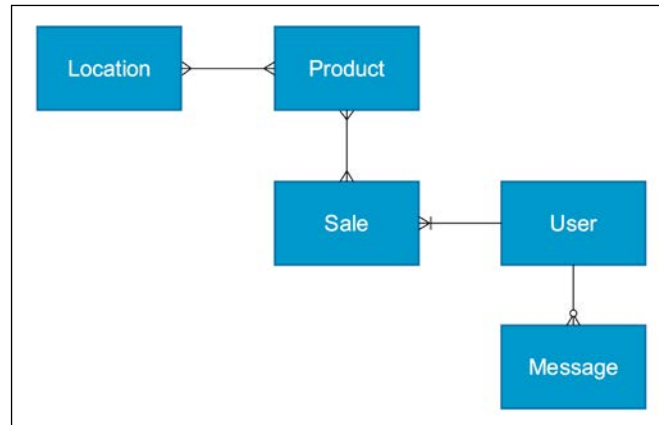
Representing your system's data accurately is extremely important for every application. Being able to access and manipulate datasets in a logical and organized way is essential to creating a maintainable code base.

Ext JS has a comprehensive data package which makes this task very easy. It has a huge number of features, including:

- Numerous data field types
- Automatic value conversion
- Validation rules
- Associations, including one-to-many, one-to-one, and many-to-many
- Abstracted reading and writing mechanisms, including AJAX, LocalStorage, and REST

In this chapter, we will demonstrate how to create the data structures required for our business dashboard application, how to join these data models with associations, and how to read and write data from a backend.

The following diagram demonstrates the data entities and their relationships that we will be representing in our application:



The main relationships are as follows:

- A **Location** can contain one or more Products
- A **Product** can be found in one or more Locations
- A **Product** can have multiple Sales
- A **Sale** can involve one or more Products
- A **User** can have multiple Sales
- A **User** can have multiple Messages

Defining models

Data models are defined as regular Ext JS classes and should extend the `Ext.data.Model` class. Model classes are generally located in the app's model folder and can have any number of subnamespaces to allow for grouping of related models.

We will start by defining our Product model in a file named `Product.js` in the model folder:

```
Ext.define('BizDash.model.Product', {
    extend: 'Ext.data.Model',
    fields: [
        {
            name: 'Name',
            type: 'string'
        }
    ]
});
```

```
    },  
    {  
      name: 'Description',  
      type: 'string'  
    },  
    {  
      name: 'Quantity',  
      type: 'int'  
    },  
    {  
      name: 'Price',  
      type: 'float'  
    }  
  ]  
});
```

To save on some key strokes, you can generate models in the command line using Sencha Cmd. Refer to *Chapter 4, Architecting an Ext JS Application*, for details on how to do this.

Fields

The `fields` property in the product model defines the name and type of each piece of data attached to the model. Each object in the `fields` array is a configuration object for the `Ext.data.Field` class, which provides various options for how the field is interpreted and stored.

The main configurations are:

- `name`: This is the name of the field. It is used as the mapping key when reading data from a data source and retrieving it later.
- `type`: This is data type the assigned value should be parsed into. Possible values include `string`, `int`, `float`, `date`, `boolean`, and `auto`. If set to `auto`, no automatic conversion will take place.
- `mapping`: This allows the field's value to be pulled from a property that doesn't match the field's name.

Field validation

Validation logic for data structures can be baked right into the definition, thus ensuring that the same logic isn't scattered all through your application. This can be added with the `validators` option which lets you define what validation rules should be applied to each field.

We can add validation rules to our Product model to ensure each field's data is correct. We do this by defining an object with each key referring to a field's name. The value of this property can be a simple string referring to the name of a validation type, an object configuring the validation, or an array of objects so multiple rules can be applied.

The following snippet shows the configuration used to validate the fields in the following way:

- The Name field is valid by being present (that is, not empty) and by being at least three characters long
- The Quantity field must be present
- The Price field is greater than 0

```
...
  validators: {
    Name: [
      {
        type: 'presence'
      },
      {
        type: 'length',
        min: 3
      }
    ],
    Quantity: 'presence',
    Price: {
      type: 'range',
      min: 0
    }
  }
...

```

The type config defines the rule we want to apply: out of the box the possibilities are presence, length, format, inclusion, exclusion, range, and email. Details of each can be found in the `Ext.data.validator.* namespace`.

Any other properties added are then used as options to customize that specific validator; for example, min defines the minimum length the Name must have.

We can apply the validation rules on a model instance (sometimes referred to as a *record*) using the `getValidation` method. This method will return an instance of the `Ext.data.Validation` class, which we can then query to determine which fields are invalid and retrieve error messages from.

Custom field types

Although Ext JS has five field types that cover the majority of scenarios required, the latest version introduces the ability to create your own custom field type that can have conversions, validations, and serializations built in. This feature can help reduce duplicated code across models that share the same field type.

In our Product model, we defined a validation rule for the `Price` field to ensure it has a positive value. We will have to apply this same rule to every model which has a field holding a monetary value. This would be a prime candidate for a custom field type that would allow this configuration to be shared.

Custom field types are declared in the same way as other classes and should extend the `Ext.data.field.Field` class (or one of its subclasses if you want to build upon their existing functionality). We will extend the `Ext.data.field.Number` type, which provides us with the logic to ensure the value is numeric.

The validation we added to the Product model can then be added to this class:

```
Ext.define('BizDash.model.field.Money', {
    extend: 'Ext.data.field.Number',
    alias: 'data.field.money',
    validators: [
        {
            type: 'range',
            min : 0
        }
    ],
    getType: function() {
        return 'money';
    }
});
```

This custom field type can now be used by assigning any field a type of `money`. The Product model's `Price` field will be rewritten as follows, and will automatically have the validation rules we specified applied to it:

```
...
{
    name: 'Price',
    type: 'money'
}
...
```

Custom data converters

A field's value can be automatically processed before being stored, in order to modify it in some way based on other field values, or to parse its value into a different type; for example, splitting a currency value received as "USD10.00" to its two component parts: currency and value.

The first way to do this is to add a `convert` property to the field's configuration and assign it a function. This function will be passed the field's value and the record instance it is being stored in and should return the processed value which will then be stored.

It should be noted that the record instance may be incomplete depending on the order in which the fields are populated, so it can't be guaranteed that other field values will be available.

The following example shows how we might implement a `StockValue` field, which uses the current `Quantity` and `Price` values to calculate the total value of the stock:

```
{
  name    : 'StockValue',
  type    : 'money',
  convert: function(val, rec) {
    return rec.get('Quantity') * rec.get('Price');
  },
  depends: ['Price', 'Quantity']
}
```

We have also included the `depends` config, which creates a dependency between our `StockValue` field and the fields we use in its `convert` function. This means that when the `Price` or `Quantity` fields are updated, the `StockValue` field is recalculated ensuring everything is kept in sync.

Another way of doing this, is to use the `calculate` option, which behaves in a very similar way, but is designed to be used for purely calculated fields rather than manipulating actual values. We could rewrite the `StockValue` field using `calculate`, as follows:

```
{
  name      : 'StockValue',
  type      : 'money',
  calculate: function(data) {
    return data.Quantity * data.Price;
  }
}
```

The `calculate` function accepts one parameter containing the record's data object, which can be used to access the other fields. As with the `convert` function, it should return the value to be stored.

By using `calculate`, the dependencies are automatically determined based on the function's contents. So, there is no need for an explicit `depends` configuration.

Working with stores

Stores are a collection of model instances that allow these models to be manipulated (for example, sorted, filtered, searched, and so on). They also provide a platform for backend interaction. Many of Ext JS' components can be bound to data stores and take care of a lot of the plumbing required to react to changes in the data held within it.

In this section, we will discuss how to construct a simple store and how to perform simple manipulation of the data within it; how to create different views of a dataset using Chained stores; and finally, how hierarchical data can be stored using TreeStores.

Simple stores

To define a store, you must extend the `Ext.data.Store` class and configure it with a model class that it will hold a collection of. The following store definition shows a store containing a collection of user records:

```
Ext.define('BizDash.store.Users', {
    extend: 'Ext.data.Store',
    model: 'BizDash.model.User'
});
```

We can now add our store to the `stores` config option in our `Application.js` file. This will result in the file being loaded and automatically instantiated, making it accessible using the `Ext.getStore` method. We can then populate it with some sample data using the `add` method:

```
// Application.js
stores: [
    'Users'
],
launch: function() {
    var usersStore = Ext.getStore('Users');
    usersStore.add([
        {
```

```
        Name : 'John',
        Email : 'john@gmail.com',
        TelNumber: ' 0330 122 2800',
        Role : 'Administrator'
    },
    {
        Name : 'Sarah',
        Email : 'sarah@hotmail.com',
        TelNumber: ' 0330 122 2800',
        Role : 'Customer'
    },
    {
        Name : 'Brian',
        Email : 'brian@aol.com',
        TelNumber: ' 0330 122 2800',
        Role : 'Supplier'
    },
    {
        Name : 'Karen',
        Email : 'karen@gmail.com',
        TelNumber: ' 0330 122 2800',
        Role : 'Administrator'
    }
  ]);
}
```

Now that we have a store filled with data, we can start exploring the various methods to interrogate and manipulate the data.

Store stats

We often need to know how many records there are within a store, and this can be done easily with the `getCount` method:

```
usersStore.getCount() // returns 4
```

Retrieving a record

The simplest way to retrieve a record instance is to access it based on its (zero-based) index. To do this, we use the `getAt` method, which returns the `Ext.data.Model` subclass at that position (or null if none was found):

```
var userRecord = usersStore.getAt(0); // { Name: "John" ... }
```

Finding specific records

In order to find a specific record based on a field's value, we use the `findRecord` method which, in its simplest form, accepts a field name and value to match. This will return the first record that matches, or null if none was found:

```
var userRecord = usersStore.findRecord('Name', 'Karen');
```

By default, this will search all records and will look for the search value at the beginning of the record's value. It will be case insensitive and will allow partial matches. These options can be changed by passing the following parameters to the `findRecord` method:

- **Field Name (String):** The field to match against
- **Search Value (String/Number/Date):** The value to find
- **Start Index (Number):** The index at which to start searching
- **Any Match (Boolean):** True to match the search value anywhere (not just at the beginning)
- **Case Sensitive (Boolean):** True to match the case
- **Exact Match (Boolean):** True to match the whole value

If you would prefer to retrieve the index of the record, rather than the actual record instance, you can use the `find` method, which will return the record's position, or -1 if not found.

Complex searches

If your search criteria are more complex than a simple field match, you can use your own custom matching functions to introduce multiple criteria. For example, you may want to find a user by name and telephone number.

You can do this using the `findBy` method, passing it a function that will have one parameter, a `record` instance, and it should return true if it is deemed a match or false if it isn't. This function will be executed once for each record in the store or until it finds a match:

```
var userRecord = usersStore.findBy(function(record) {  
    return record.get('Name') === 'John' && record.get('TelNumber') ===  
    '0330 122 2800';  
});
```


Filtering a store

Stores can be filtered at any time with similar basic or complex queries. Doing so, will result in the store exposing a subset of its original dataset. The filters can be removed at any time to restore the full dataset once again.

We can filter a store by using the `addFilter` method, which accepts a single `Ext.util.Filter` instance or an array of `Ext.util.Filter` instances or configuration objects. These should specify the name of the field being filtered on and the value to compare against. The following example filters the store to only users named "Brian":

```
console.log(usersStore.getCount()); // 4
usersStore.addFilter({
    property: 'Name',
    value: 'Brian'
});
console.log(usersStore.getCount()); // 1
```

In a way similar to the `findBy` method, we can also filter with a function that allows more complex queries to be constructed. The following example shows the store being filtered by name and e-mail:

```
console.log(usersStore.getCount()); // 4
usersStore.filterBy(function(record) {
    return record.get('Name') === 'John' && record.get('Email') ===
    'john@swarmonline.com';
});
console.log(usersStore.getCount()); // 1
```

While a store is filtered, all query operations (`find`, `getAt`, and so on) are performed on the filtered dataset and won't search any filtered items. To return the store to its unfiltered state, simply call the `clearFilter` method:

```
console.log(usersStore.getCount()); // 1
usersStore.clearFilter();
console.log(usersStore.getCount()); // 4
```

Configuration-based filtering

All the preceding examples show the store being filtered programmatically. It is also possible to define a default filter when the store is configured, and which will be applied to all new records being added to the store. This can be a full `Ext.util.Filter` configuration object with a simple property/value combination or a more complex `filterFn`:

```
Ext.define('BizDash.store.Users', {
```

```

    extend: 'Ext.data.Store',
    model: 'BizDash.model.User',
    filters: [
        {
            property: 'Name',
            value: 'John'
        }
    ]
  });

```

Sorting a store

We can also change the sort order of a store's records by using the `sort` or `sortBy` methods. As with filtering and finding, these allow for simple sorting in one or more fields, or more complex sorting using a function. Note that if no sorting options are provided, the records will remain in the order they are added.

A simple sort on `Name` can be seen as follows:

```

console.log(usersStore.getAt(0).get('Name')); // John

usersStore.sort('Name', 'ASC');

console.log(usersStore.getAt(0).get('Name')); // Brian

console.log(usersStore.getAt(3).get('Name')); // Sarah

usersStore.sort('Name', 'DESC');

console.log(usersStore.getAt(0).get('Name')); // Sarah

console.log(usersStore.getAt(3).get('Name')); // Brian

```

To perform more complex sorting, you can provide an object that will configure an instance of the `Ext.util.Sorter` class. This example sorts by the reverse of each user's name (for example, `nhoJ`, `haraS`):

```

usersStore.sort({
    sorterFn: function(a, b) {
        var aName = a.get('Name').split('').reverse().join(''),
            bName = b.get('Name').split('').reverse().join('');
        return ((aName < bName) ? -1 : ((aName > bName) ? 1 : 0));
    },
    direction: 'ASC'
});

```

Configuration-based sorting

Once again, we can define a default sorter on a store that will be reapplied after each new record is added. This should be an `Ext.util.Sorter` configuration object:

```
Ext.define('BizDash.store.Users', {
    extend: 'Ext.data.Store',
    model: 'BizDash.model.User',
    sorters: [
        {
            property: 'Name',
            direction: 'DESC'
        }
    ]
});
```

Grouping

A store's records can also be grouped by specific fields or combination of fields, which can be very useful to display data in grouped grids. We use the `group` method as follows, passing in the `Role` field, as the field to group on and `ASC` to determine the sort direction of the groups:

```
usersStore.group('Role', 'ASC');
```

By calling this method, the store remains unchanged. This means all its records can be accessed as before, but it now gives us access to an `Ext.util.GroupCollection` instance, which is a collection of `Ext.util.Group` instances that holds a set of grouped records in each instance.

We can interrogate the grouped data using the `getGroups` method:

```
var groups = usersStore.getGroups();
console.log(groups.getCount()); // 3 groups: Administrator,
Supplier, Customer
console.log(groups.getAt(0).getGroupKey()); // Administrator
console.log(groups.getAt(1).getGroupKey()); // Customer
console.log(groups.getAt(2).getGroupKey()); // Supplier
console.log(groups.getAt(0).getCount()); // 2
console.log(groups.getAt(0).getAt(0).get('Name')); // John
console.log(groups.getAt(0).getAt(1).get('Name')); // Karen
```

In addition to accepting these parameters, we can also pass in an `Ext.util.Grouper` config object to define a more complex grouping setup. In the following example, we group by each user's email address domain (notice that we use the `clearGrouping` method to reset any existing groupings already in place):

```

usersStore.clearGrouping();
usersStore.group({
  groupFn: function(rec){
    var email = rec.get('Email'),
        emailSplit = email.split('@'),
        domain = emailSplit[1];
    return domain;
  },
  direction: 'DESC'
});
groups = usersStore.getGroups();
console.log(groups.getCount()); // 3 groups: hotmail.com,
//gmail.com, aol.com
console.log(groups.getAt(0).getGroupKey()); // hotmail.com
console.log(groups.getAt(1).getGroupKey()); // gmail.com
console.log(groups.getAt(2).getGroupKey()); // aol.com
console.log(groups.getAt(0).getCount()); // 1
console.log(groups.getAt(0).getAt(0).get('Name')); // Sarah

```

Configuration-based grouping

Grouping can also be done in the store's definition by using the `groupField` and `groupDirection` options together, or the `grouper` option on its own. The following snippets show the store grouped in `Role` using each combination:

```

Ext.define('BizDash.store.Users', {
  extend: 'Ext.data.Store',
  model: 'BizDash.model.User',
  grouper: [
    {
      property: 'Role',
      direction: 'ASC'
    }
  ]
});

```

And, here is the another one:

```

Ext.define('BizDash.store.Users', {
  extend: 'Ext.data.Store',
  model: 'BizDash.model.User',
  groupField: 'Role',
  groupDir: 'ASC'
});

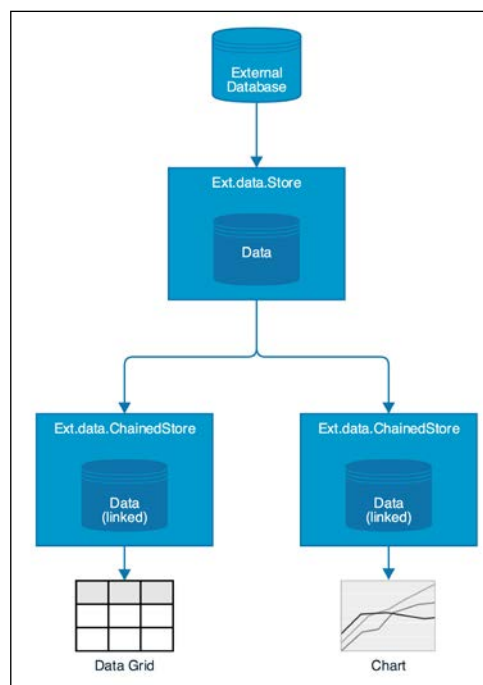
```

Chained stores

Chained stores are a new addition to Ext JS 5 and allow us to create different views of the same underlying data store without it affecting the base data or other chained stores.

Imagine that we want to display the data within our `usersStore` in separate grids for each role. We would want one grid attached to the `usersStore` with a filter of `Role=Administrator`, the next grid with `Role=Supplier`, and so on. Unfortunately, before Ext JS 5, this was not possible to do with just one store and so, we would have to create multiple stores with multiple copies of the same data and manage them all separately. Chained stores take this problem away and allow us to attach to an underlying base data store and apply any filters and sorters to the view of the data, without them affecting the base store or any other linked chained stores.

The following diagram explains this concept:



Key things to remember about chained stores are:

- The record instances are shared between all chained stores and the base store
- Any updates made to any of the records will be propagated through to all linked stores

Going back to our roles example, we can now solve the problem by defining three chained stores that can be bound to separate grids or data views.

We link them to the base data store using the `source` configuration option, which can accept a store instance or a store ID. We can then add any sorters or filters to the chained store, as we would to a regular store.

We create these in the folder `store/users/` to keep things organized:

```
// store/users/Admins.js
Ext.define('BizDash.store.users.Admins', {
    extend: 'Ext.data.ChainedStore',
    config: {
        source: 'Users',
        filters: [
            {
                property: 'Role',
                value: 'Administrator'
            }
        ]
    }
});

// store/users/Customers.js
Ext.define('BizDash.store.users.Customers', {
    extend: 'Ext.data.ChainedStore',
    config: {
        source: 'Users',
        filters: [
            {
                property: 'Role',
                value: 'Customer'
            }
        ]
    }
});

// store/users/Suppliers.js
Ext.define('BizDash.store.users.Suppliers', {
    extend: 'Ext.data.ChainedStore',
    config: {
        source: 'Users',
        filters: [
            {
                property: 'Role',
```

```
        value: 'Supplier'
      }
    ]
  }
});
```

We then add the new stores to the stores config within `Application.js` and create an instance of each and see the record counts in each:

```
var adminStore = Ext.create('BizDash.store.users.Admins');
var customerStore = Ext.create('BizDash.store.users.Customers');
var supplierStore = Ext.create('BizDash.store.users.Suppliers');
console.log(usersStore.getCount()); // 4
console.log(adminStore.getCount()); // 2
console.log(customerStore.getCount()); // 1
console.log(supplierStore.getCount()); // 1
```

TreeStores

The `Ext.data.TreeStore` class is a specialist store, extending from the regular `Ext.data.Store` class, which manages hierarchical data. This type of store must be used to bind to Tree Panels and other components, where data is required to have a hierarchical structure.

TreeStores are created in the same way as regular stores except for one main difference. The collection of the model that a `TreeStore` manages must extend from the `Ext.data.TreeModel` class, rather than the usual `Ext.data.Model`.

Ext.data.TreeModels

The reason that a `TreeStore`'s model must extend the `Ext.data.TreeModel` class is that each model instance must be decorated with additional properties and methods to allow the models' hierarchy to be managed correctly, and to allow the tree components to display them correctly. These extra properties and methods come from the `Ext.data.NodeInterface` class, whose members are all applied to each model instance in the `TreeStore`.

We will create a simple tree example by first defining the model that will be stored. Our data will represent our application's navigation structure and will form the basis of our menu:

```
Ext.define('BizDash.model.NavigationItem', {
  extend: 'Ext.data.TreeModel',
  fields: [
```

```

    {
      name: 'Label',
      type: 'string'
    },
    {
      name: 'Route',
      type: 'string'
    }
  ]
});

```

If you require this class within your `Application.js` file, create an instance of this model and inspect its contents. You will see that the model now has over 20 additional data fields. These fields are all used to describe each node for various purposes, for example, tracking its location, its state, and its appearance.

By default, these fields have their `persist` configs set to `false`, so they won't be included in any save operations initiated.

```

▼ navigationItem: constructor
  Ext.data.Model#persistenceProperty: (...)
  ▶ childNodes: Array[0]
  ▶ config: Object
  ▼ data: Object
    Label: "Home"
    Route: "/home"
    allowDrag: true
    allowDrop: true
    checked: null
    children: null
    cls: ""
    depth: 0
    expandable: true
    expanded: false
    href: ""
    hrefTarget: ""
    icon: ""
    iconCls: ""
    id: "NavigationItem-1"
    index: -1
    isFirst: false
    isLast: false
    leaf: false
    loaded: false
    loading: false
    parentId: null
    qshowDelay: 0
    qtip: ""
    qtitle: ""
    root: false
    text: ""
    visible: true
  ▶ __proto__: Object

```


If you look further, you will also see numerous extra methods added to the model. These methods can be used to manage the node and its children, traverse the tree structure, and interrogate its position in the hierarchy. Some of the more useful methods are detailed here:

- `appendChild`: This adds the specified node(s) as the last child of the current node
- `insertChild`: This inserts the new node at the specified position
- `removeChild`: This removes the specified node from the child collection
- `eachChild`: This executes a function on each of the child nodes
- `findChild`: This finds the first child that matches the given property/value given
- `isLeaf`: This determines if the current node is a leaf without any further children

Creating a TreeStore

We will create a simple TreeStore called `BizDash.store.Navigation`, which will contain a collection of `BizDash.model.NavigationItem` model instances. This store extends the base `Ext.data.TreeStore` class:

```
Ext.define('BizDash.store.Navigation', {  
    extend: 'Ext.data.TreeStore',  
    model: 'BizDash.model.NavigationItem'  
});
```

We include this store in the `Application.js` file's stores array, where it will be automatically loaded and instantiated, and will be accessible via a call to `Ext.getStore('Navigation')`.

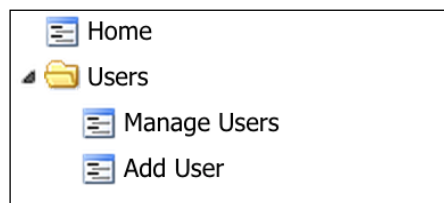
Populating a TreeStore

TreeStores must always have a *root node*, which is the root parent of all child nodes within the store. It is common for this node to always be hidden and never actually seen in tree views. To populate our TreeStore programmatically (that is, not from an external data source—we'll talk about that in the next section), we can either add the full hierarchy using the `setRoot` method, or add child nodes individually using the `appendChild` or `insertChild` methods.

We will start by using the `setRoot` method, which builds a hierarchy of `NavigationItem` instances expanding outward from the root:

```
var navigationStore = Ext.getStore('Navigation');
navigationStore.setRoot({
  Label : 'Root',
  children: [
    {
      Label: 'Home',
      Route: '/home'
    },
    {
      Label : 'Users',
      Route : '',
      children: [
        {
          Label: 'Manage Users',
          Route: '/manage-users',
          leaf : true
        },
        {
          Label: 'Add User',
          Route: '/add-user',
          leaf : true
        }
      ]
    }
  ]
});
```

This will produce a structure, as follows:



Note that our own fields (`Label` and `Route`) are combined with some of the `Ext . data . NodeInterface` fields (`children` and `leaf`); we do this to indicate the parent-child relationship between the nodes, and to indicate which nodes do not have any children, respectively.

Once a root node exists, we can also start adding nodes using the `appendChild` method, as shown in the following code snippet:

```
// append a node at the same level as 'Home' and 'Users'
navigationStore.getRoot().appendChild({
  Label: 'Orders',
  Route: '/orders'
});
// find the 'Users' node and append a node to it
navigationStore.getRoot().findChild('Label', 'Users', true).
appendChild({
  Label: 'Import Users',
  Route: '/import-users'
});
```

Getting data into your application

So far, we have only dealt with hard-coded data and not with real-world examples of loading and saving data from/to external sources, whether they are REST endpoints, local databases, or third party APIs. Ext JS supports a number of different ways of assisting with data loading and persistence, including AJAX and `LocalStorage`.

Ext.Ajax

Although only indirectly related to stores, we are first going to discuss AJAX requests in a more general sense and how we can perform them to make calls to a server backend.

`Ext.Ajax` is a singleton instance of the `Ext.data.Connection` class, which provides us with a very simple interface for making AJAX calls to a server and handling the response. We will primarily focus on the `request` method, which initiates this call and allows us to specify how it should be made.

Simple AJAX calls

We will start by demonstrating how to make a simple AJAX request to a static JSON file. We simply pass in a configuration object to the `request` method, telling the framework where to make the AJAX request to, in this case specifying the `url` property:

```
Ext.Ajax.request({
  url: 'user.json'
});
```

If we run this in the console of the Ext JS app, we should see the request being made in the **Network** tab. At the moment, we are ignoring the response given to us, so now we will include a callback function to process the received data.

As we're sure you know, AJAX calls are asynchronous, meaning the rest of the code will carry on while the request is made and will not wait for it to complete. This means we must handle the response in a callback function that will get executed when the response has been received. We do this by specifying the `success` property, whose function will be given two parameters: a response object and an options object.

In our success handler, we will decode the JSON string received and log the output to the console:

```
Ext.Ajax.request({
    url: 'user.json',
    success: function(response, options){
        var user = Ext.decode(response.responseText);
        console.log(user);
    }
});
```

Handling errors

Obviously we can't just rely on the fairytale case where all our AJAX requests complete successfully, so we must include some alternative should our request fail (that is, return a non-200 response code). We can do this by specifying a `failure` config and defining a function that will be executed should the request fail. The following example console logs the response status code when an error occurs:

```
Ext.Ajax.request({
    url: 'user.json',
    success: function(response, options){
        var user = Ext.decode(response.responseText);
        console.log(user);
    },
    failure: function(response, options){
        console.log('The request failed! Response Code: ' + response.status);
    }
});
```

You can try this out by modifying the `url` property to a non-existent one to force a 404 error.

Other useful configurations

You can pass a lot of configuration options to the `request` method. We run down a few of these as follows:

- `params`: This is an object whose key/value pairs will be sent along with the request
- `method`: This is the method by which the request is sent (defaults to `GET` when no `params` are present, or `POST` if they are)
- `callback`: The function defined here will be called following the request, regardless of the success or failure
- `timeout`: This defines the timeout length, in seconds, for the request
- `headers`: This defines the headers that are sent with the request

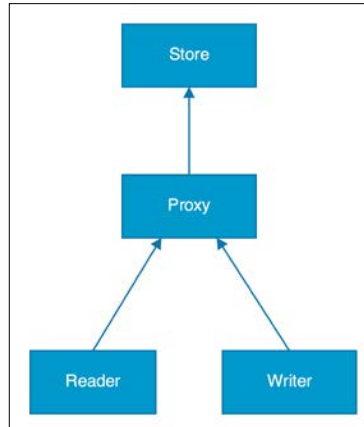
Proxies

Now that we have discussed performing simple AJAX requests, which could quite happily be used to populate stores, we will move on to explain proxies, which provide us with a simple mechanism for our stores to communicate with data sources.

A store or model can be configured with a proxy, which provides an abstracted layer above the specifics of each possible data source (be it an external server or `LocalStorage`). This abstraction allows us to provide a simple configuration and have the store take care of performing the intricacies of communicating with the source.

Proxies are linked to two other class types: `Reader` and `Writer`. The `Reader` class is responsible for interpreting the received data and parsing it correctly, so it can be turned into `Model` instances. The `Writer` class on the other hand, looks after collecting the data to be saved.

The following diagram shows how these classes are linked:



AJAX proxies

The most common type of proxy is an AJAX proxy (`Ext.data.proxy.Ajax`), which allows us to load and save data to server endpoints via an AJAX call. We will add an AJAX proxy to our users store to load the data from a simple JSON file:

```
Ext.define('BizDash.store.Users', {
    extend: 'Ext.data.Store',
    model: 'BizDash.model.User',
    proxy: {
        type: 'ajax',
        url: 'users.json',
        reader: {
            type: 'json',
            rootProperty: 'rows'
        }
    }
});
```

Let's break down this code. We start by adding a `proxy` config option with a `config` object. We specify the type of `proxy` we want (we'll talk about other types shortly) — in this case an AJAX proxy which equates to the `Ext.data.proxy.Ajax` class. We next specify the `url` to load the data from. Finally, we tell the proxy how to interpret the results by giving it a `reader` configuration. The type tells it we're going to be receiving JSON data and so we want to use the `Ext.data.reader.Json` class. The `rootProperty` tells the reader which property of the received JSON object to look in for the data records.

This setup will load data from the `users.json` file, which contains the following data:

```
{
  "success": true,
  "results": 4,
  "rows": [
    {
      "Name"      : "John",
      "Email"     : "john@gmail.com",
      "TelNumber": "0330 122 2800",
      "Role"      : "Administrator"
    },
    {
      "Name"      : "Sarah",
      "Email"     : "sarah@hotmail.com",
      "TelNumber": "0330 122 2800",
      "Role"      : "Customer"
    },
    {
      "Name"      : "Brian",
      "Email"     : "brian@aol.com",
      "TelNumber": "0330 122 2800",
      "Role"      : "Supplier"
    },
    {
      "Name"      : "Karen",
      "Email"     : "karen@gmail.com",
      "TelNumber": "0330 122 2800",
      "Role"      : "Administrator"
    }
  ]
}
```

We can now call the load method of the users store and it will make an AJAX request to the `users.json` file and populate itself with four records. The following code will load the store and log the loaded records once it has completed:

```
Ext.getStore('Users').load(function(records, operation, success){
    console.log(Ext.getStore('Users').getCount()); // 4
    console.log(records); // [ ...record instances... ]
});
```

LocalStorage proxies

Another type of proxy is the `localStorage` proxy, which allows us to load and save data to the LocalStorage within the browser. It is extremely simple to configure a store to communicate with the LocalStorage, and it follows the same pattern as the AJAX proxy, as seen here:

```
Ext.define('BizDash.store.Users', {
    extend: 'Ext.data.Store',
    model: 'BizDash.model.User',
    proxy: {
        type: 'localStorage',
        id : 'users'
    }
});
```

Again, we specify the proxy type, which in this case, will use the `Ext.data.proxy.LocalStorage` class. We also specify an `id` which will be used to identify items that belong to this store within the LocalStorage. This `id` must be unique across all proxies.

We will first demonstrate saving records, so we have some data to load in LocalStorage. We can save the store's records by calling the `sync` method after adding some records:

```
Ext.getStore('Users').add([
    {
        Name : 'John',
        Email : 'john@gmail.com',
        TelNumber: ' 0330 122 2800',
        Role : 'Administrator'
    },
    {
        Name : 'Sarah',
```



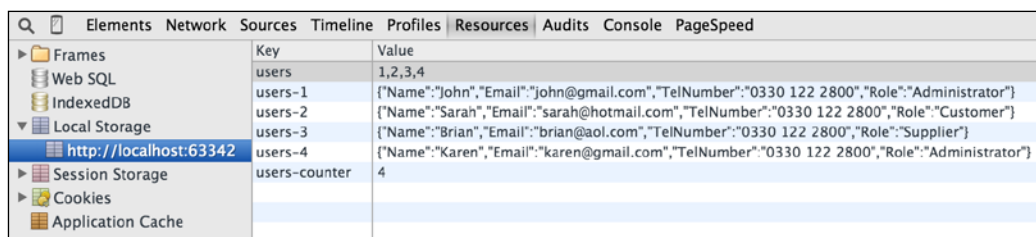
```
        Email : 'sarah@hotmail.com',
        TelNumber: ' 0330 122 2800',
        Role : 'Customer'
    },
    {
        Name : 'Brian',
        Email : 'brian@aol.com',
        TelNumber: ' 0330 122 2800',
        Role : 'Supplier'
    },
    {
        Name : 'Karen',
        Email : 'karen@gmail.com',
        TelNumber: ' 0330 122 2800',
        Role : 'Administrator'
    }
  ]);

Ext.getStore('Users').sync();
```

If you open up the **Resources** tab of Developer Tools and inspect the LocalStorage area, you will see the records with keys based on the `id` we provided.

There are three types of items stored:

- `<id>-counter`, which holds the number of records stored
- `<id>`, which contains a comma-separated list of the stored records' IDs
- Each record's JSON encoded data stored with a key in the format of `<id>-<Record ID>`

A screenshot of the Chrome DevTools 'Resources' tab. The left sidebar shows the 'Local Storage' section expanded, with a tree view showing 'users', 'users-1', 'users-2', 'users-3', 'users-4', and 'users-counter'. The main pane displays a table with 'Key' and 'Value' columns. The 'users' key has a value of '1,2,3,4'. The 'users-1' key has a value of '{"Name":"John","Email":"John@gmail.com","TelNumber":"0330 122 2800","Role":"Administrator"}'. The 'users-2' key has a value of '{"Name":"Sarah","Email":"sarah@hotmail.com","TelNumber":"0330 122 2800","Role":"Customer"}'. The 'users-3' key has a value of '{"Name":"Brian","Email":"brian@aol.com","TelNumber":"0330 122 2800","Role":"Supplier"}'. The 'users-4' key has a value of '{"Name":"Karen","Email":"karen@gmail.com","TelNumber":"0330 122 2800","Role":"Administrator"}'. The 'users-counter' key has a value of '4'.

Key	Value
users	1,2,3,4
users-1	{"Name":"John","Email":"John@gmail.com","TelNumber":"0330 122 2800","Role":"Administrator"}
users-2	{"Name":"Sarah","Email":"sarah@hotmail.com","TelNumber":"0330 122 2800","Role":"Customer"}
users-3	{"Name":"Brian","Email":"brian@aol.com","TelNumber":"0330 122 2800","Role":"Supplier"}
users-4	{"Name":"Karen","Email":"karen@gmail.com","TelNumber":"0330 122 2800","Role":"Administrator"}
users-counter	4

Now that we have records saved in LocalStorage, we can call the store's `load` method, in the same way as we did with the AJAX proxy, to retrieve these saved records and repopulate the store.

REST proxies

Many APIs are RESTful and so Ext JS has a proxy to make integration with them extremely easy. By defining a REST proxy, our CRUD requests will be executed to the endpoint with the correct method type. We can update our users store to use a REST proxy (the `Ext.data.writer.Rest` class), by changing the type to `rest`:

```
proxy: {
  type : 'rest',
  url  : 'users',
  reader: {
    type      : 'json',
    rootProperty: 'rows'
  }
}
```

Now that we have our store, by using a REST proxy, we can update one of the records and sync the store and see a `GET` request made to retrieve the records and a `PUT` request being made to update the data:

```
Ext.getStore('Users').getAt(0).set('Email', 'john@hotmail.com');
Ext.getStore('Users').sync();
```

Here is the screenshot of it:

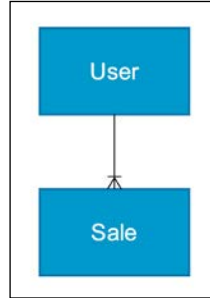


Data associations

Our application's data models will always have associations between them, which must be represented in our client-side applications, so that they can be manipulated easily and consistently. Ext JS offers the ability to model one-to-one, one-to-many, and many-to-many relationships between Model types.

One-to-many

One-to-many relationships exist when a single entity owns multiple entities of a different type. For example, an author *has many* books or a recipe *has many* ingredients. In our example project, we will model the association between users and sales – a user can have many sales and each sale only has one user.



Configuring a proxy and data source

We will start by defining a proxy and simple static data source to connect our models to so that they can be populated. We do this with the following code pointing to two simple JSON files:

```
Ext.define('BizDash.model.Sale', {
    extend: 'Ext.data.Model',
    ...
    proxy: {
        type: 'ajax',
        url: 'sale.json',
        reader: {
            type: 'json'
        }
    }
    ...
});

// sale.json
[
    {
        "id": 1,
        "userId": 1,
        "productId": 1,
```

```

        "Date" : "2014-08-04T14:41:17.220Z",
        "Quantity" : 1,
        "TotalCost": 9.99
    },
    {
        "id": 2,
        "userId": 1,
        "productId": 1,
        "Date" : "2014-08-03T14:41:17.220Z",
        "Quantity" : 2,
        "TotalCost": 19.98
    }
]

```

```

Ext.define('BizDash.model.User', {
    extend: 'Ext.data.Model',
    ...
    proxy: {
        type : 'ajax',
        url : 'user.json',
        reader: {
            type: 'json'
        }
    }
    ...
});

```

```

// user.json
{
    "id" : 99,
    "Name" : "Joe Bloggs",
    "Email" : "joe@gmail.com",
    "TelNumber": " 07777777777",
    "Role" : "Salesman"
}

```

Defining the association

Now that we have our infrastructure in place to allow data to be loaded into each model, we can define the relationship between these two entities. There are two ways this can be done: `hasMany` and `reference`.

hasMany config

The `hasMany` configuration option is the way these associations have always been created and allow us to define the models that would be related, and the names and data sources of each. This approach allows for more explicit control over the details of the association, making it more appropriate for more custom situations:

```
Ext.define('BizDash.model.User', {
    extend: 'Ext.data.Model',
    ...
    hasMany: [
        {
            model: 'BizDash.model.Sale',
            name: 'sales'
        }
    ],
    ...
});
```

The `hasMany` config accepts an array of association definitions and should include a `model` option, defining the name of the model that the association data represents, and a `name` option which will be used to access the associated data.

To access the associated data via the defined proxy, we use the user model's static `load` method to load the user with an ID of 1 and specify a callback function to execute when it is successfully loaded:

```
BizDash.model.User.load(1, {
    success: function(userRecord) {
    }
});
```

We can now load in the sale records that are associated with this user. We do this by calling the `sales` method (so called based on the name config we gave it in our `hasMany` configuration), which will return an `Ext.data.Store` instance that will contain our sale record instances. We then call that store's `load` method, which will hit the `sale.json` file and return the related sale records. Within a callback function, we can then see the data items loaded:

```
BizDash.model.User.load(1, {
    success: function(userRecord) {
        userRecord.sales().load(function() {
            console.log('User: ', userRecord.get('Name')); // Joe Bloggs
            console.log('Sales: ', userRecord.sales().getCount()); // 2
        });
    }
});
```

```

    });
  }
});

```

The reference config

The new way in Ext JS 5 to define associations is to use the `reference` config on a model's field, which will link the foreign key in a model to a related model type. This greatly simplifies the construction of associations and means both directions of the association can be accessed easily.

The following code shows the `reference` config added to the `BizDash.model.Sale` model, linking it to the `BizDash.model.User` model:

```

Ext.define('BizDash.model.Sale', {
  extend: 'Ext.data.Model',
  fields: [
    ...
    {
      name: 'userId',
      type: 'int',
      reference: 'BizDash.model.User'
    },
    ...
  ]
});

```

We can now use the same code to load the user and then its associated sale records:

```

BizDash.model.User.load(1, {
  success: function(userRecord) {
    userRecord.sales().load(function() {
      console.log('User: ', userRecord.get('Name')); // Joe Bloggs
      console.log('Sales: ', userRecord.sales().getCount()); // 2
    });
  }
});

```

This technique also makes it easy to gain access to a sale record associated to a user. We make use of the generated `getUser` method, which is added to the sale record because of the reference we created. Calling this method will load the relevant user model based on the foreign key (`userId`) defined in the sale:

```

var saleRecord = Ext.create('BizDash.model.Sale', {
  id : 1,
  userId : 1,

```

```
    productId: 1,  
    Date : new Date(),  
    Quantity : 1,  
    TotalCost: 9.99  
  });  
  saleRecord.getUser(function(userRecord) {  
    console.log(userRecord.get('Name')) // Joe Bloggs  
  });
```

Exploring requests

Each time we did a `load` or `getUser` call in our associations, an AJAX request was made to our server resources. In our case, we just had simple static JSON files, but in real life, we would have a proper server implementation which would return the correct data, based on the sale or user being requested. If we look at the **Network** tab of Developer Tools, we can see that the request to the `user.json` and `sale.json` files passed up parameters that we can then use to retrieve the correct data from our server database.

The following screenshot shows the user with ID of 1 loading. It passes the ID value as a query string parameter:



When loading the sales relating to this user, the server is given a slightly more complex JSON string with details of the filter that should be applied to retrieve the correct data. In our server-side code, we would parse this and use it in our database queries.



Many-to-many

Another relationship type that can be modeled is many-to-many. In our application, we want to define a many-to-many relationship between Products and Locations, meaning that one Product can be stored in multiple Locations and one Location can have multiple Products. For example, Product 1 could be located in Warehouse 1 and Warehouse 2, and Warehouse 1 could store Product 1 and Product 2.

Configuring a proxy and data source

As we did earlier, we will start by defining a proxy and simple static data source to connect our models to so that they can be populated. We do this with the following code pointing to two simple JSON files:

```
Ext.define('BizDash.model.Product', {
    extend: 'Ext.data.Model',
    ...
    proxy: {
        type: 'ajax',
        url: 'product.json',
        reader: {
            type: 'json'
        }
    }
    ...
});

// product.json
{
    "id" : 1,
    "Name" : "Product 1",
    "Description": "Product 1 Description",
    "Quantity" : 10,
    "Price" : 9.99
}

Ext.define('BizDash.model.Locations', {
    extend: 'Ext.data.Model',
    ...
    proxy: {
        type: 'ajax',
        url: 'location.json',
        reader: {
            type: 'json'
        }
    }
    ...
});
```



```
    }
    ...
  });

// location.json
[
  {
    "id" : 1,
    "Name" : "Location 1",
    "Row" : 20,
    "Shelf": 10
  },
  {
    "id" : 2,
    "Name" : "Location 2",
    "Row" : 11,
    "Shelf": 22
  }
]
```

Defining the association

The simplest way to define this relationship is to add the `manyToMany` config option to the models on both sides of the association (Product and Location) and by providing the name of the model that should be on the other side:

```
Ext.define('BizDash.model.Product', {
  extend: 'Ext.data.Model',
  ...
  manyToMany: [
    'Location'
  ]
  ...
});

Ext.define('BizDash.model.Location', {
  extend: 'Ext.data.Model',
  ...
  manyToMany: [
```

```
        'Product'
    ]
    ...
  });
```

Loading the associated data

Like the one-to-many association, by defining this link, Ext JS will generate new methods called `locations` and `products` in the `Product` and `Location` models respectively. These methods will return an `Ext.data.Store` instance and will contain our associated data:

```
BizDash.model.Product.load(1, {
    success: function(productRecord) {
        productRecord.locations().load(function() {
            console.log(productRecord.locations().getCount()); // 2
        });
    }
});
```

From a `Location` record, the related `Products` can be loaded in an identical way.

Saving data

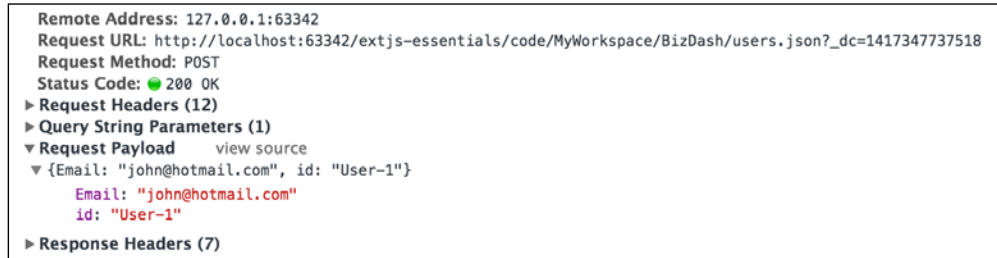
Now that we are able to load data successfully into our application, we can move on to persisting changes that we've made to this data within our client-side application. In most applications, when a change is made to a data store or record, we want to persist that change back to the data source, be it a server-side database or `LocalStorage` store. As mentioned previously, Ext JS provides us with the `Ext.data.writer.Writer` class (and its sub-classes) to manage the fields required to be written.

In its most simple and standard form, a store will save any changes made when the `sync` method is called, without any further additions to our proxy configuration.

If we continue with our users store example, we can update one of the `record` fields and then call the `sync` method:

```
Ext.getStore('Users').getAt(0).set('Email', 'john@hotmail.com');
Ext.getStore('Users').sync();
```

If we inspect the **Network** tab of our Developer Tools, we will see a `POST` request being made to our `users.json` file passing along the changed field and the record's ID:



Similarly, we can add or remove records and call the `sync` method and an equivalent `POST` request will be made:

```
Ext.getStore('Users').add({
    Name: 'Stuart',
    Email: 'stuart@gmail.com',
    Role: 'Customer',
    TelNumber: ' 0330 122 2800'
});
Ext.getStore('Users').sync();
```

Here is the screenshot of the request that is made:



The preceding request, after a record has been added, shows all fields being sent as `POST` parameters:

```
Ext.getStore('Users').removeAt(0)
Ext.getStore('Users').sync();
```

```
Remote Address: 127.0.0.1:63342
Request URL: http://localhost:63342/extjs-essentials/code/MyWorkspace/BizDash/users.json?_dc=1417348148761
Request Method: POST
Status Code: 200 OK
▶ Request Headers (12)
▶ Query String Parameters (1)
▼ Request Payload view source
  {id: "User-1"}
    id: "User-1"
▶ Response Headers (7)
```

When a record is removed, the `id` field is POSTed to the server.

CRUD endpoints

Obviously the preceding example is a little flawed, in that we are POSTing changes to a static JSON file. By default, Ext JS will send all changes to the URL specified in the proxy, regardless of whether they are adds, updates, or deletes. This is often undesirable when separate endpoints exist for each of these actions. We can easily configure our proxy to contact a different endpoint for each action using the `api` configuration property. Our proxy configuration could be rewritten as follows:

```
proxy: {
  type : 'ajax',
  url  : 'users.json',
  api  : {
    create : 'user-add.php',
    read   : 'users.json',
    update : 'user-update.php',
    destroy: 'user-delete.php'
  },
  reader: {
    type      : 'json',
    rootProperty: 'rows'
  }
}
```

If we rerun the add, update, and delete actions, we will see calls being made to each of the PHP files specified in our `api` configuration.

Data writers

Data writers allow us to control how a store's data is constructed for sending out to be saved and give us a variety of options to customize this process. By default, a proxy will use a JSON writer (`Ext.data.writer.Json`), which will result in the to-be-saved data being encoded as a JSON string. Alternatively, the XML writer (`Ext.data.writer.Writer`) could be used, which would transfer data as XML.

To define a writer, we use the `writer` config of the proxy and give it its own configuration object. The `type` property determines which writer class is used – either JSON or XML.

We can configure our users store to use a JSON writer, as follows:

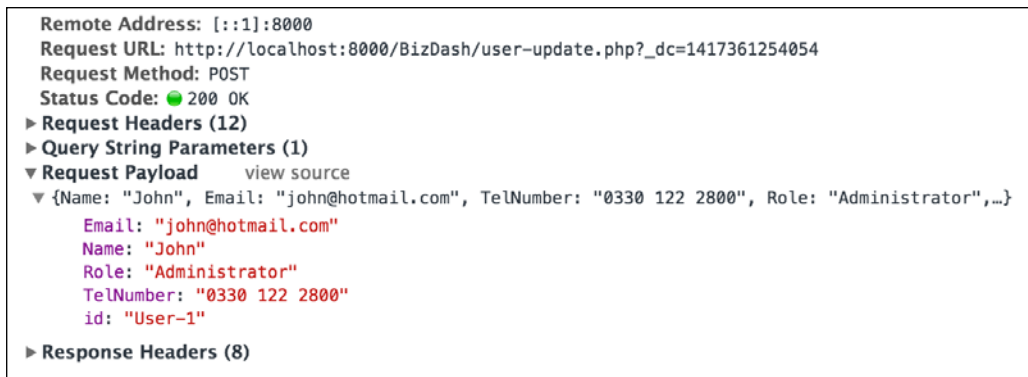
```
proxy: {  
  ...  
  writer: {  
    type: 'json'  
  }  
}
```

We'll now take a look at a couple of the configuration options the writer class offers to allow us to customize how and what data is sent to the server.

In our previous update example, we saw the edited field (in this case the `Email` field) being sent, along with the record's ID, to the server. Sometimes, we will want to send all the fields to the server regardless of whether they have been edited or not. We can do this easily with the `writeAllFields` config as demonstrated here:

```
proxy: {  
  ...  
  writer: {  
    type: 'json',  
    writeAllFields: true  
  }  
}
```

Here is the screenshot of this update request:



On occasion, you might want to do some additional processing on a dataset before it is sent up to the server. We can define this extra processing step by using the `transform` config, giving it a function to execute during the writing process. This function will accept the data object that will be sent and should return the processed data object. The following example shows how we can ensure that the given e-mail address is always in lowercase:

```
proxy: {  
  ...  
  writer: {  
    type : 'json',  
    writeAllFields: true,  
    transform : {  
      fn: function(data, request) {  
        data.Email = data.Email.toLowerCase();  
        return data;  
      }  
    }  
  }  
}
```

Summary

Throughout this chapter, we have explored the details of how to model our application's data structure with Ext JS. We have covered:

- Defining models
- Loading and saving data to server and LocalStorage
- Defining relationships between models through associations
- How to work with data within data stores

We will make use of all of these lessons throughout the rest of the book as our example application comes to life and we start to integrate data sources and data visualizations to it.

6

Combining UI Widgets into the Perfect Layout

Layouts are fundamental to the appearance and usability of your application. UI widgets can be combined, and arranged, in any number of different configurations to create simple and complex application layouts.

An Ext JS widget requires a layout to manage its sizing and positioning on the screen. The framework has a large number of different layouts, which provide simple configurations and flexibility to produce applications.

This chapter will cover the most common topics relating to layouts. These are as follows:

- How components fit together in layouts
- Examples and explanations of working with the most common layouts, namely:
 - The border layout
 - The fit layout
 - HBox and VBox layouts
- How to design layouts that are responsive to the user's screen size

Layouts and how they work

In Ext JS, containers have layouts that manage the sizing and positioning of their child components. Traditionally, a developer would apply a combination of CSS rules to the elements in the DOM to build the desired screen layout. Ext JS takes care of most of this for us, by allowing us to define a layout configuration in our container/component and configure the sizing and positioning with JavaScript.

By default, a container is configured with an Auto layout, which makes child components flow naturally at full width in much the same way DIVs do in a regular HTML page.

The following is a list of layouts that are provided by the framework. It is possible, and common, to combine a number of layouts together; this is so that child containers or components are positioned and sized appropriately. For example, a tab panel (card layout) has multiple child containers, of which each may have different layouts. Each of these layouts has different configuration options to control how your app renders. By default, the Ext JS UI widgets have component layouts configured, which you may need to be aware of.

- **Absolute:** Using X and Y coordinates, the absolute layout fixes the position of the container on screen. Overlapping is possible with this layout.
- **Accordion:** The accordion layout provides the ability to create an accordion stack of panels on screen.
- **Anchor:** This is a layout that enables anchoring of contained elements relative to the container's dimensions.
- **Border:** The border layout enables you to attach containers to the border of a central region giving you a north, south, east, and west region. This layout has built-in behavior for collapsing and resizing regions.
- **Card:** A card layout provides a stack of containers that can be moved back and forth. A card layout is ideal for a wizard-style component or tabbed components.
- **Center:** The contents of a center layout are centered within their container.
- **Column:** A column layout is ideal for presenting your interface in multiple columns.
- **Fit:** A very common layout, the fit layout stretches the widget to the size and position of the parent container. You can only have one item in a fit layout.

- **HBox:** Much like the column layout, this layout presents components horizontally. It has some useful configurations to stretch and position child components.
- **VBox:** Much like the HBox layout, this layout presents components vertically, one below the other. It has some useful configurations to stretch and position child components.
- **Table:** While tables are less popular with developers, a table layout can still be useful. The content in a table layout will be rendered as an HTML table.

How the layout system works

As we know, it's the layout that's responsible for the sizing and positioning of your container's children. Correctly rendering the screen requires that all child components have their sizes and positions calculated so that the DOM can be updated. The framework does this with the `updateLayout` method. This method recurses fully through all child components and calculates the appropriate positions and sizes.



For those familiar with previous versions of the framework, the `updateLayout` method replaced the `doLayout` method in Ext JS 4.1.

The framework automatically takes care of sizing and positioning by calling the `updateLayout` method. For example, when the browser window is adjusted or resized, or you add or remove components, the framework will do the necessary calculations to ensure your components appear correctly on screen.

There are some circumstances when it is beneficial to call the `updateLayout` manually. Laying out components can be a resource-intensive task, and if you know you are going to make a number of updates to components, then it may be worth thinking about batching the layout into one. For instance, adding three components one after another would trigger three calls to `updateLayout` (which recurses through all child components). By using a `suspendLayout` flag, we can prevent our application from updating the DOM until we're ready. When we are ready, it's a simple case of switching the `suspendLayout` flag off by setting it to false and then manually calling the `updateLayout` method of the container.

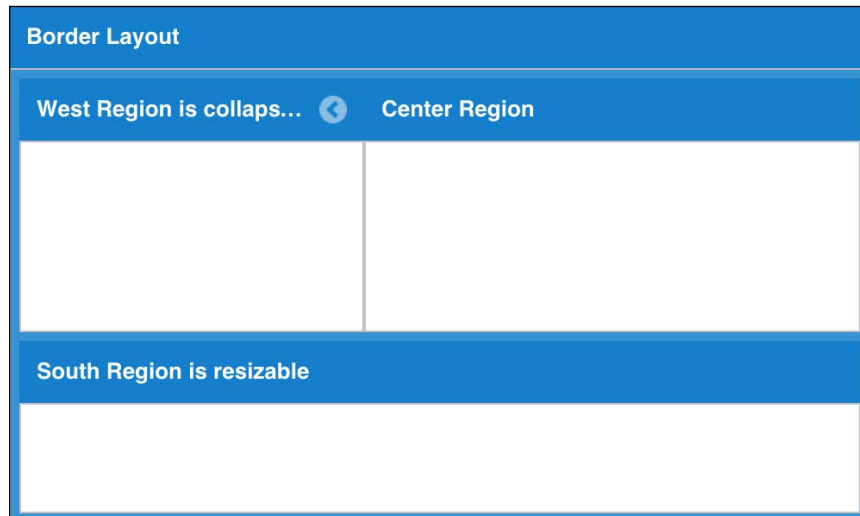
The component layout

A component also has a layout which defines how it sizes and positions its internal child items. Component layouts are configured using the `componentLayout` config option.

In most circumstances, you will not require the `componentLayout` configuration, unless you plan on writing custom components that have complex layout requirements.

Using the border layout

If you are looking to create a desktop style experience with your user interface, then the border layout is for you.



The border layout is an application-oriented layout, supporting multiple nested panels, the ability to collapse regions by clicking on the regions' header or collapse icon, and the resizing of regions by clicking and dragging the splitter bar between them.

One of the building blocks of our BizDash application will be a Viewport with a border layout. Here, we will learn how to create a simple border layout using the maximum number of regions configurable (north, south, east, west, and center). The west and east regions will be collapsible, with the east region loading pre-collapsed. Resizing will be demonstrated in the south and west regions. These four borders will surround the center region, which regardless of your configuration, is required for a border layout to work.

Starting with the Viewport

A Viewport renders itself to the document's body and automatically consumes the viewable area. It represents the entire viewable browser area and automatically uses 100 percent of the browser window's width and height (minus address bars, developer tools, and so on, of course). Our Viewport will have a border layout to manage the size and positioning of its child containers.

In our application, we set the `autoCreateViewport` property in `app.js` to `BizDash.view.main.Main`. This automatically sets our main view container to a Viewport for us.

```
Ext.application({
  name: 'BizDash',
  extend: 'BizDash.Application',
  autoCreateViewport: 'BizDash.view.main.Main'
});
```

Configuring the border layout

Our main view takes the following configuration:

```
Ext.define('BizDash.view.main.Main', {
  extend: 'Ext.container.Container',
  xtype: 'app-main',
  controller: 'main',
  viewModel: {
    type: 'main'
  },
  layout: {
    type: 'border'
  },
  items: [
    {
      region: 'north',
      margins: 5,
      height: 100,
    },
  ],
});
```

```
{
  title: 'West',
  xtype: 'panel',
  region: 'west',
  margins: '0 5 0 5',
  flex: 3,
  collapsible: true,
  split: true,
  titleCollapse: true,
  tbar: [{
    text: 'Button'
  }]
  bind: {
    title: '{name}'
  },
  html: 'This area is commonly used for navigation, for example,
  using a tree component.',
{
  title: 'Center',
  region: 'center',
  xtype: 'tabpanel',
  items:[{
    title: 'Tab 1',
    html: 'Content appropriate for the current
    navigation'
  }]
},
{
  title: 'East',
  region: 'east',
  margins: '0 5 0 5',
  width: 200,
  collapsible: true,
  collapsed: true
},
```

```

    {
      title: 'South',
      region: 'south',
      margins: '0 5 5 5',
      flex: .3,
      split: true
    }
  ]
});

```

The border layout, as the name suggests, creates a layout of components that border a central component. Therefore, a requirement of the border layout is that one item must be specified as the center.

The center region, which you must include for a border layout to work, automatically expands to consume the empty space left over from the other regions in your layout. It does this by having a pre-defined flex value of 1 for both height and width.

The north and south regions take a height or flex configuration. In our app, the north region has a fixed height of 100 px and the south region has a flex of 3. The south and center regions' heights are calculated based on the height remaining in the browser window. Here, the height of the south region is just under a third of the height of the center. The west and east regions, instead, take a width or flex configuration.

We add further functionality with `collapsed`, `collapsible`, `split`, and `titleCollapse` specified in the desired regions' configuration. They do the following:

- `collapsed`: If set to `true`, it means the region will start collapsed (the regions need to be `Ext.panel.Panel` to be collapsible)
- `collapsible`: If set to `true`, it allows the user to expand/collapse the panel by clicking on the toggle tool that's added to the header
- `titleCollapse`: If set to `true`, it makes the panel collapse no matter where the user clicks on the panel's header
- `split`: If set to `true`, it makes the region resizable by allowing the users to click and drag the dividing bar between regions.

Using the fit layout

The fit layout in Ext JS is ideal if you want a container to expand a component to fill its parent. The fit layout is easy to use and requires no configuration.

```

Ext.define('BizDash.view.main.Main', {
  extend: 'Ext.container.Container',
  xtype: 'app-main',

```

```
controller: 'main',
viewModel: {
  type: 'main'
},
layout: {
  type: 'border'
},
items: [{
  ...
},
{
  title: 'South',
  region: 'south',
  margins: '0 5 5 5',
  flex: .3,
  split: true,
  layout: 'fit',
  items: [{
    xtype: 'component',
    html: 'South Region'
  }]
}]
});
```

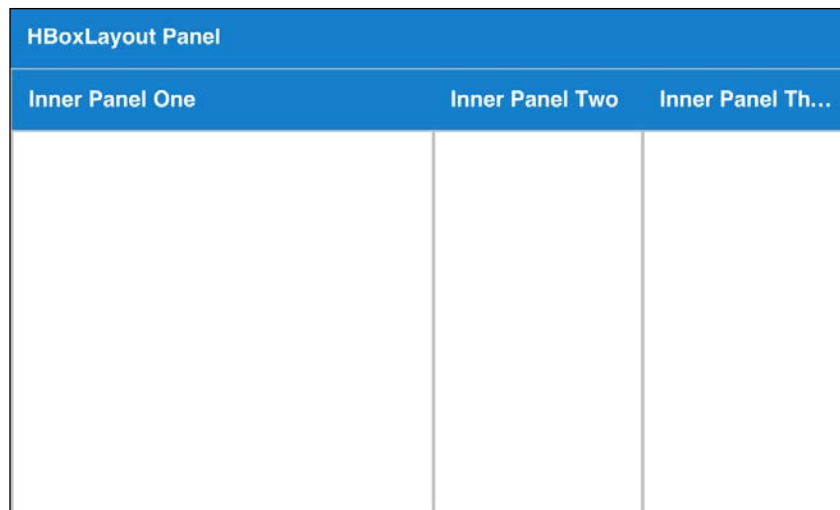
The main class has a border layout, but the south region requires a fit layout for its child panel.

The fit layout works by defining the `layout` config option as `fit` in the parent container. This tells Ext JS that the child item should expand to fill the entire space available from its parent.

It's worth noting that the fit layout will only work for the first child item of the parent container. If you have multiple items defined, the first will be displayed (as it will expand into the remaining space of its parent) and the others will not be visible.

Using the HBox layout

The HBox layout allows you to align components horizontally across a container in a manner similar to a column layout. However, it's a bit more advanced, as it lets you configure additional properties, such as the height of the columns as well.



Let's create an overview widget with three columns to show events, messages, and notes:

```
Ext.define('BizDash.view.dashboard.Overview', {
    extend: 'Ext.container.Container',
    xtype: 'app-overview',
    controller: 'overview',
    viewModel: {
        type: 'overview'
    },
    layout: {
        type: 'hbox',
        align: 'stretchmax'
    },
    items: [
        {
            flex: .3,
            title: 'Today\'s Events'
        },
        {
            flex: .3,
            title: 'Messages'
        },
        {
            width: 200,
            title: 'Notes'
        }
    ]
});
```


Defining an HBox layout ensures that Ext JS horizontally positions each child item giving the appearance of columns in our dashboard application.

We have configured `stretchmax` on the `align` config option, meaning that all child items will automatically be stretched to the height of the tallest child. The `align` config option controls how child items are vertically aligned in an HBox layout. Valid values are as follows:

- `begin`: This is the default value. All items in the HBox layout will be vertically aligned to the top of the container
- `middle`: All items will be vertically aligned to the middle (or center) of the container
- `stretch`: Each item will be vertically stretched to fit the height of the container
- `stretchmax`: This vertically stretches all items to the height of the largest item, creating a uniform look without having to individually define a height for each item

The VBox layout has a very similar config to this layout, but the options are slightly different, as they are designed to control the horizontal alignment.

By defining a height of 200 and configuring `align: 'stretchmax'`, all other panels will have their height stretched to 200 px.

Widths in HBox layouts

HBoxes can have their width defined in two different ways: fixed widths and flex widths.

Fixed width

Child containers can have their width fixed by defining a width configuration in the object. Containers with a fixed width will retain their width dimensions, even when the browser window is resized. Therefore, they are not fluid.

If, however, you require your layout to be more responsive to the resizing of windows and so on, then it's a `flex` config that's required.

Flex width

The `flex` config option relatively flexes the child items horizontally in their parent containers. For example, take a container with `flex: 1` and one with `flex: 3`. In these cases, 25 percent of the remaining parent space is given to the first container and 75 percent of the space is given to the other.

Flex values are calculated in the following way:

$$((\text{Container Width} - \text{Fixed Width of Child Components}) / \text{Sum of Flexes}) * \text{Flex Value}.$$

Packing items together

Another useful config option for the HBox layout is `pack`.

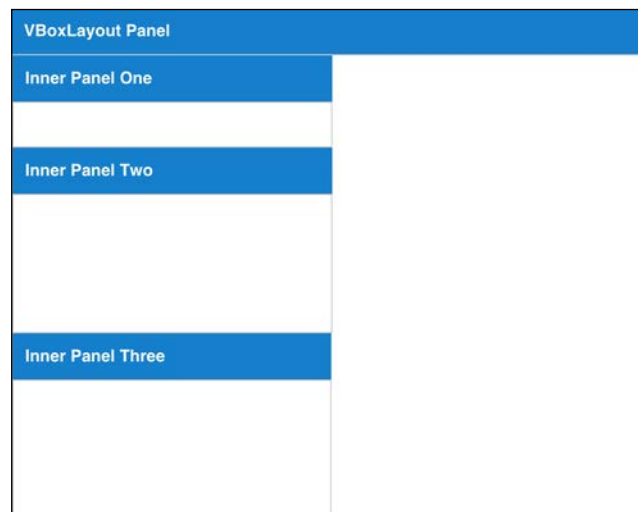
The `pack` config option controls how the child items are packed together. If the items do not stretch to the full width of the parent container, it's possible to align them to the left, middle, or right using this option. Valid values are as follows:

- `start`: This is the default value. It aligns all items to the left of the parent container.
- `center`: This aligns all items to the middle of the container.
- `end`: It aligns all items to the right of the container.

The VBox layout contains the same `pack` config, but it is designed to configure the packing of child items vertically.

Using the VBox layout

The VBox layout is very similar to the HBox layout. The only difference is that VBox allows you to align components vertically in a container. Just like the HBox layout, this layout is configured by setting fixed widths for its children using the `width` config, or by calculating the width automatically with the `flex` config.



Alignment and packing

The VBox layout has some useful configuration options that are described here.

align: String

The `align` config option controls how child items are horizontally aligned in a VBox layout. Valid values are:

- `begin`: This is the default value. All items in the VBox layout are horizontally aligned to the left of the container and use their `width` config to define how wide they are.
- `middle`: All items are horizontally aligned to the middle (or center) of the container.
- `stretch`: Each item is horizontally stretched to fit the width of the container.
- `stretchmax`: This horizontally stretches all items to the width of the largest item, creating a uniform look without having to individually define a width for each item.

pack: String

The `pack` config option controls how the child items are packed together. If the items do not stretch to the full height of the parent container (that is, have no flex values), it's possible to align them to the top, middle, or bottom using this option. Valid values are:

- `start`: This is the default value. It aligns all items to the top of the parent container.
- `center`: This aligns all items to the middle (or center) of the container.
- `end`: This aligns all items to the bottom of the container.

Responsive layouts

Long gone are the days when web browsers were only found in desktop PCs and laptops. Nowadays, they can be found in an ever-increasing range of hardware, such as phones, tablets, TVs, and cars, to name a few. Since the release of the iPhone in 2007, a new type of web, the mobile web, has grown in popularity, and the expectations of users have changed. Users expect to access a web application and have fantastic experience, no matter how large or small their screen is.

Until the advent of Ext JS 5, it was difficult to cater to the differing needs of users. Ext JS 5 has, however, adopted the popular move towards responsive design, giving developers a practical way of adapting the app layout, depending on the size and resolution of the screen presenting it.

Ext.mixin.Responsive and Ext.plugin.Responsive

There is a new responsive mixin and plugin that adds a `responsiveConfig` option.

In our border layout, we want to alter how the regions are presented, based on the screen orientation. Adding the `responsiveConfig` with the following *rules* to the main view tells the application to add the navigation panel to the north or west region based on the orientation of the screen:

```
Ext.define('BizDash.view.main.Main', {
    extend: 'Ext.container.Container',
    xtype: 'app-main',
    controller: 'main',
    viewModel: {
        type: 'main'
    },
    layout: {
        type: 'border'
    },
    items: [
        {
            xtype: 'panel',
            bind: {
                title: '{name}'
            },
            html: 'This area is commonly used for navigation, for example,
            using a tree component.',
            width: 250,
            split: true,
            tbar: [
                {
                    text: 'Button'
                }
            ],
            plugins: 'responsive',
            responsiveConfig: {
                landscape: {
```

```
        region: 'west'
      },
      portrait: {
        region: 'north'
      }
    },
    {
      region: 'center',
      xtype: 'tabpanel',
      items:[
        {
          title: 'Tab 1',
          html: 'Content appropriate for the current navigation'
        }
      ]
    }
  ]
});
```

In this example, the `responsiveConfig` has two rules defined: one for landscape and another for portrait. When the application satisfies the rule, the config defined in the object of that rule will be applied to the component:

```
responsiveConfig: {
  landscape: {
    region: 'west'
  },
  portrait: {
    region: 'north'
  }
}
```

If, for example, our screen orientation is landscape, then the framework will apply `region: 'west'` to the panel. This is the same as the following:

```
{
  xtype: 'panel',
  bind: {
    title: '{name}'
  },
  html: 'This area is commonly used for navigation, for example, using
a tree component.',
  width: 250,
  split: true,
  tbar: [
    {
```

```
        text: 'Button'
      }
    ],
    region: 'west'
  }
}
```

ResponsiveConfig rules

We must define rules that contain a condition, or multiple conditions, based on which the rules will be applied. These rules can be any valid JavaScript expression, but the following values are considered in scope:

- `landscape`: This returns true when the orientation is landscape or on a desktop device.
- `portrait`: This returns true when the orientation is portrait but is always false on desktop devices.
- `tall`: This returns true if the height is greater than the width.
- `wide`: This returns true if the width is greater than the height.
- `width`: This defines the width of the Viewport.
- `height`: This defines the height of the Viewport.

The width and height are particularly useful, as these give us a way to define our own values in a way similar to breakpoints in media queries.

The following example shows how to create a rule that will be applied when a portrait device is less than 400 pixels wide:

```
responsiveConfig: {
  'portrait && width < 400': {
    ...
  }
}
```

Summary

Throughout this chapter, we have explored in detail how to define the layout of our components on screen using Ext JS. We have covered:

- How the layout manager works
- An overview of layouts available to us

- Examples of how to use the border, fit and HBox/VBox layouts
- Designing a responsive layout to cope with different screen sizes and device types

We will make use of all of these lessons throughout the rest of the book, as our example application comes to life and we start to integrate our widgets into it.

In the next chapter, we will demonstrate how to create common UI widgets for our sample application. These widgets will make use of some of the layouts we have seen in this chapter.

7

Constructing Common UI Widgets

One of the biggest features that draws developers to Ext JS is the vast array of UI widgets available out of the box. The ease with which they can be integrated with each other and the attractive and consistent visuals each of them offers is also a big attraction. No other framework can compete on this front, and this is a huge reason Ext JS leads the field of large-scale web applications.

In this chapter, we will look at how UI widgets fit into the framework's structure, how they interact with each other, and how we can retrieve and reference them. We will then delve under the surface and investigate the lifecycle of a component and the stages it will go through during the lifetime of an application. Finally, we will add the first UI components to our BizDash application in the form of data grids, trees, data views, and forms.

Anatomy of a UI widget

Every UI element in Ext JS extends from the base component class `Ext . Component`. This class is responsible for rendering UI elements to the HTML document. They are generally sized and positioned by layouts used by their parent components and participate in the automatic component lifecycle process.

You can imagine an instance of `Ext . Component` as a single section of the user interface in a similar way that you might think of a DOM element when building traditional web interfaces.

Each subclass of `Ext . Component` builds upon this simple fact and is responsible for generating more complex HTML structures or combining multiple `Ext . Components` to create a more complex interface.

`Ext.Component` classes, however, can't contain other `Ext.Components`. To combine components, one must use the `Ext.container.Container` class, which itself extends from `Ext.Component`. This class allows multiple components to be rendered inside it and have their size and positioning managed by the framework's layout classes (see *Chapter 6, Combining UI Widgets into the Perfect Layout*, for more details).

Components and HTML

Creating and manipulating UIs using components requires a slightly different way of thinking than you may be used to when creating interactive websites with libraries such as jQuery.

The `Ext.Component` class provides a layer of abstraction from the underlying HTML and allows us to encapsulate additional logic to build and manipulate this HTML. This concept is different from the way other libraries allow you to manipulate UI elements and provides a hurdle for new developers to get over.

The `Ext.Component` class generates HTML for us, which we rarely need to interact with directly; instead, we manipulate the configuration and properties of the component. The following code and screenshot show the HTML generated by a simple `Ext.Component` instance:

```
var simpleComponent = Ext.create('Ext.Component', {  
    html      : 'Ext JS Essentials!',  
    renderTo: Ext.getBody()  
});
```

```
<div class="x-component x-component-default x-border-box" id="ext-comp-1010">  
Ext JS Essentials!</div>
```

As you can see, a simple `<DIV>` tag is created, which is given some CSS classes and an autogenerated ID, and has the HTML config displayed inside it.

This generated HTML is created and managed by the `Ext.dom.Element` class, which wraps a DOM element and its children, offering us numerous helper methods to interrogate and manipulate it. After it is rendered, each `Ext.Component` instance has the element instance stored in its `e1` property. You can then use this property to manipulate the underlying HTML that represents the component.

As mentioned earlier, the `e1` property won't be populated until the component has been rendered to the DOM. You should put logic dependent on altering the raw HTML of the component in an `afterrender` event listener or override the `afterRender` method.

The following example shows how you can manipulate the underlying HTML once the component has been rendered. It will set the background color of the element to red:

```
Ext.create('Ext.Component', {
    html      : 'Ext JS Essentials!',
    renderTo  : Ext.getBody(),
    listeners: {
        afterrender: function(comp) {
            comp.el.setStyle('background-color', 'red');
        }
    }
});
```

It is important to understand that digging into and updating the HTML and CSS that Ext JS creates for you is a dangerous game to play and can result in unexpected results when the framework tries to update things itself. There is usually a *framework way* to achieve the manipulations you want to include, which we recommend you use first.

We always advise new developers to try not to fight the framework too much when starting out. Instead, we encourage them to follow its conventions and patterns, rather than having to wrestle it to do things in the way they may have previously done when developing traditional websites and web apps.

The component lifecycle

When a component is created, it follows a lifecycle process that is important to understand, so as to have an awareness of the order in which things happen. By understanding this sequence of events, you will have a much better idea of where your logic will fit and ensure you have control over your components at the right points.

The creation lifecycle

The following process is followed when a new component is instantiated and rendered to the document by adding it to an existing container. When a component is shown explicitly (for example, without adding to a parent, such as a floating component) some additional steps are included. These have been denoted with a * in the following process.

constructor

First, the class' constructor function is executed, which triggers all of the other steps in turn. By overriding this function, we can add any setup code required for the component.

Config options processed

The next thing to be handled is the config options that are present in the class (see *Chapter 2, Mastering the Framework's Building Blocks*, for details). This involves each option's `apply` and `update` methods being called, if they exist, meaning the values are available via the getter from now onwards.

initComponent

The `initComponent` method is now called and is generally used to apply configurations to the class and perform any initialization logic.

render

Once added to a container, or when the `show` method is called, the component is rendered to the document.

boxready

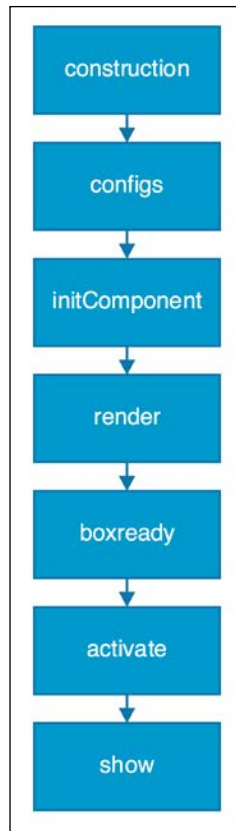
At this stage, the component is rendered and has been laid out by its parent's layout class, and is ready at its initial size. This event will only happen once on the component's first layout.

activate (*)

If the component is a floating item, then the `activate` event will fire, showing that the component is the active one on the screen. This will also fire when the component is brought back to focus, for example, in a **Tab** panel when a tab is selected.

show (*)

Similar to the previous step, the `show` event will fire when the component is finally visible on screen.



The destruction process

When we are removing a component from the Viewport and want to destroy it, it will follow a destruction sequence that we can use to ensure things are cleaned up sufficiently, so as to avoid memory leaks and so on. The framework takes care of the majority of this cleanup for us, but it is important that we tidy up any additional things we instantiate.

hide (*)

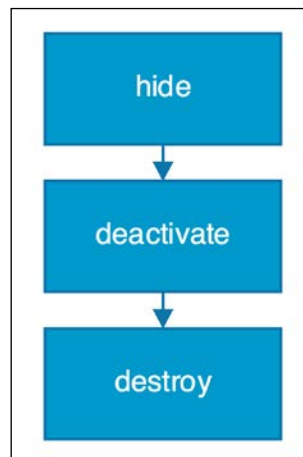
When a component is manually hidden (using the hide method), this event will fire and any additional hide logic can be included here.

deactivate (*)

Similar to the activate step, this is fired when the component becomes inactive. As with the activate step, this will happen when floating and nested components are hidden and are no longer the items under focus.

destroy

This is the final step in the teardown process and is implemented when the component and its internal properties and objects are cleaned up. At this stage, it is best to remove event handlers, destroy subclasses, and ensure any other references are released.



Component Queries

Ext JS boasts a powerful system to retrieve references to components called Component Queries. This is a CSS/XPath style query syntax that lets us target broad sets or specific components within our application. For example, within our controller, we may want to find a button with the text "Save" within a component of type MyForm.

In this section, we will demonstrate the Component Query syntax and how it can be used to select components. We will also go into details about how it can be used within `Ext.container.Container` classes to scope selections.

xtypes

Before we dive in, it is important to understand the concept of xtypes in Ext JS. An xtype is a shorthand name for an `Ext.Component` that allows us to identify its declarative component configuration objects. For example, we can create a new `Ext.Component` as a child of an `Ext.container.Container` using an xtype with the following code:

```
Ext.create('Ext.Container', {
    items: [
        {
            xtype: 'component',
            html : 'My Component!'
        }
    ]
});
```

Using xtypes allows you to lazily instantiate components when required, rather than having them all created upfront.

Common component xtypes include:

Classes	xtypes
<code>Ext.tab.Panel</code>	<code>tabpanel</code>
<code>Ext.container.Container</code>	<code>container</code>
<code>Ext.grid.Panel</code>	<code>gridpanel</code>
<code>Ext.Button</code>	<code>button</code>

xtypes form the basis of our Component Query syntax in the same way that element types (for example, `div`, `p`, `span`, and so on) do for CSS selectors. We will use these heavily in the following examples.

Sample component structure

We will use the following sample component structure—a panel with a child tab panel, form, and buttons—to perform our example queries on:

```
var panel = Ext.create('Ext.panel.Panel', {
    height : 500,
    width : 500,
    renderTo: Ext.getBody(),
    layout: {
        type : 'vbox',
        align: 'stretch'
```

```
    },
    items : [
        {
            xtype : 'tabpanel',
            itemId: 'mainTabPanel',
            flex : 1,
            items : [
                {
                    xtype : 'panel',
                    title : 'Users',
                    itemId: 'usersPanel',
                    layout: {
                        type : 'vbox',
                        align: 'stretch'
                    },
                    tbar : [
                        {
                            xtype : 'button',
                            text : 'Edit',
                            itemId: 'editButton'
                        }
                    ],
                    items : [
                        {
                            xtype : 'form',
                            border : 0,
                            items : [
                                {
                                    xtype : 'textfield',
                                    fieldLabel: 'Name',
                                    allowBlank: false
                                },
                                {
                                    xtype : 'textfield',
                                    fieldLabel: 'Email',
                                    allowBlank: false
                                }
                            ]
                        },
                        {
                            buttons: [
                                {
                                    xtype : 'button',
                                    text : 'Save',
                                    action: 'saveUser'
                                }
                            ]
                        }
                    ]
                }
            ]
        }
    ]
}
```

```
    ]
  },
  {
    xtype : 'grid',
    flex : 1,
    border : 0,
    columns: [
      {
        header : 'Name',
        dataIndex: 'Name',
        flex : 1
      },
      {
        header : 'Email',
        dataIndex: 'Email'
      }
    ],
    store : Ext.create('Ext.data.Store', {
      fields: [
        'Name',
        'Email'
      ],
      data : [
        {
          Name : 'Joe Bloggs',
          Email: 'joe@example.com'
        },
        {
          Name : 'Jane Doe',
          Email: 'jane@example.com'
        }
      ]
    })
  }
]
},
{
  xtype : 'component',
  itemId : 'footerComponent',
  html : 'Footer Information',
  extraOptions: {
    option1: 'test',
```



```
        option2: 'test'
      },
      height : 40
    }
  ]
}));
```

Queries with Ext.ComponentQuery

The `Ext.ComponentQuery` class is used to perform Component Queries, with the `query` method primarily used. This method accepts two parameters: a query string and an optional `Ext.container.Container` instance to use as the root of the selection (that is, only components below this one in the hierarchy will be returned). The method will return an array of components or an empty array if none are found.

We will work through a number of scenarios and use Component Queries to find a specific set of components.

Finding components based on xtype

As we have seen, we use `xtypes` like element types in CSS selectors. We can select all the `Ext.panel.Panel` instances using its `xtype`—`panel`:

```
var panels = Ext.ComponentQuery.query('panel');
```

We can also add the concept of hierarchy by including a second `xtype` separated by a space. The following code will select all `Ext.Button` instances that are descendants (at any level) of an `Ext.panel.Panel` class:

```
var buttons = Ext.ComponentQuery.query('panel buttons');
```

We could also use the `>` character to limit it to buttons that are direct descendants of a panel.

```
var directDescendantButtons = Ext.ComponentQuery.query('panel > button');
```

Finding components based on attributes

It is simple to select a component based on the value of a property. We use the XPath syntax to specify the attribute and the value. The following code will select buttons with an `action` attribute of `saveUser`:

```
var saveButtons =
  Ext.ComponentQuery.query('button[action="saveUser"]');
```

Finding components based on itemIds

ItemIds are commonly used to retrieve components, and they are specially optimized for performance within the `ComponentQuery` class. They should be unique only within their parent container and not globally unique like the `id` config. To select a component based on `itemId`, we prefix the `itemId` with a `#` symbol:

```
var usersPanel = Ext.ComponentQuery.query('#usersPanel');
```

Finding components based on member functions

It is also possible to identify matching components based on the result of a function of that component. For example, we can select all text fields whose values are valid (that is, when a call to the `isValid` method returns `true`):

```
var validFields = Ext.ComponentQuery.query('form >
textfield{isValid()}');
```

Scoped Component Queries

All of our previous examples will search the entire component tree to find matches, but often we may want to keep our searches local to a specific container and its descendants. This can help reduce the complexity of the query and improve the performance, as fewer components have to be processed.

`Ext.Containers` have three handy methods to do this: `up`, `down`, and `query`. We will take each of these in turn and explain their features.

up

This method accepts a selector and will traverse up the hierarchy to find a single matching parent component. This can be useful to find the grid panel that a button belongs to, so an action can be taken on it:

```
var grid = button.up('gridpanel');
```

down

This returns the first descendant component that matches the given selector:

```
var firstButton = grid.down('button');
```

query

The `query` method performs much like `Ext.ComponentQuery.query` but is automatically scoped to the current container. This means that it will search all descendant components of the current container and return all matching ones as an array.

```
var allButtons = grid.query('button');
```

Hierarchical data with trees

Now that we know and understand components, their lifecycle, and how to retrieve references to them, we will move on to more specific UI widgets.

The tree panel component allows us to display hierarchical data in a way that reflects the data's structure and relationships.

In our application, we are going to use a tree panel to represent our navigation structure to allow users to see how the different areas of the app are linked and structured.

Binding to a data source

Like all other data-bound components, tree panels must be bound to a data store—in this particular case it must be an `Ext.data.TreeStore` instance or subclass, as it takes advantage of the extra features added to this specialist store class.

We will make use of the `BizDash.store.NavigationTreeStore` we created in *Chapter 5, Modeling Data Structures for Your UI*, to bind to our tree panel.

Defining a tree panel

The tree panel is defined in the `Ext.tree.Panel` class (which has an xtype of `treepanel`), which we will extend to create a custom class called `BizDash.view.navigation.NavigationTree`:

```
Ext.define('BizDash.view.navigation.NavigationTree', {
    extend: 'Ext.tree.Panel',
    alias: 'widget.navigation-NavigationTree',
    store : 'Navigation',
    columns: [
```

```

    {
        xtype : 'treecolumn',
        text : 'Navigation',
        dataIndex: 'Label',
        flex : 1
    }
],
rootVisible: false,
useArrows : true
});

```

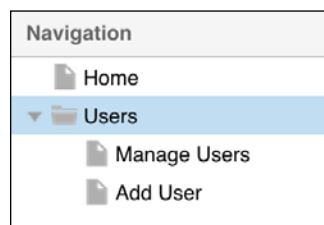
We configure the tree to be bound to our TreeStore by using its `storeId`, in this case, `Navigation`.

A tree panel is a subclass of the `Ext.panel.Table` class (similar to the `Ext.grid.Panel` class), which means it must have a columns configuration present. This tells the component what values to display as part of the tree. In a simple, *traditional* tree, we might only have one column showing the item and its children; however, we can define multiple columns and display additional fields in each row. This would be useful if we were displaying, for example, files and folders and wanted to have additional columns to display the file type and file size of each item.

In our example, we are only going to have one column, displaying the `Label` field. We do this by using the `treecolumn` xtype, which is responsible for rendering the tree's navigation elements. Without defining `treecolumn`, the component won't display correctly.

The `treecolumn` xtype's configuration allows us to define which of the attached data model's fields to use (`dataIndex`), the column's header text (`text`), and the fact that the column should fill the horizontal space (`flex`: see the *Using the VBox layout* and *Using the HBox layout* sections in *Chapter 6, Combining UI Widgets into the Perfect Layout*, for more details on this concept).

Additionally, we set the `rootVisible` to `false`, so the data's root is hidden, as it has no real meaning other than holding the rest of the data together. Finally, we set `useArrows` to `true`, so the items with children use an arrow instead of the `+/-` icon.



Displaying tabular data

The grid component is one of the biggest reasons Ext JS is selected by developers. Its performance, features, and flexibility make it a powerful feature. In this section, we will work through an example of creating a grid to display our product data.

Product data

Before we dive into creating a grid component, we must create a data store that will hold the product data that our grid will display. In *Chapter 5, Modeling Data Structures for Your UI*, we defined the models that represent our product data, but we didn't create a products store to hold a collection of product model instances. To do this, we create a new file called `Products.js` in the `store` folder of our project with the following class definition:

```
Ext.define('BizDash.store.Products', {
    extend: 'Ext.data.Store',
    model: 'BizDash.model.Product',
    autoLoad: true,
    proxy: {
        type: 'ajax',
        url: 'products.json',
        reader: {
            type: 'json',
            rootProperty: 'rows'
        }
    }
});
```

This is a simple store definition with the only new configuration being the `autoLoad: true` setting, which will perform a load using the defined proxy as soon as the store is instantiated. In order to have this store automatically instantiated, we include it in the stores config of the `Application.js` file.

Our `products.json` file contains some simple product data, as seen here:

```
{
    "success": true,
    "rows": [
        {
            "id": 1,
            "Name": "Product 1",
            "Description": "Product 1 Description",
            "Quantity": 1,
            "Price": 9.99
        }
    ]
}
```

```
    },
    {
      "id"          : 2,
      "Name"        : "Product 2",
      "Description": "Product 2 Description",
      "Quantity"    : 5,
      "Price"       : 2.99
    },
    {
      "id"          : 3,
      "Name"        : "Product 3",
      "Description": "Product 3 Description",
      "Quantity"    : 1000,
      "Price"       : 5.49
    }
  ]
}
```

Product grid

To create a grid, we use the `Ext.grid.Panel` component which extends from the `Ext.panel.Panel` class and so can be used in the same way as a simple panel (for example, it can have docked items, be given dimensions, and so on).

We will define our product grid in a file called `ProductGrid.js` inside a new folder called `product` under the project's view folder. We give it a basic configuration:

```
Ext.define('BizDash.view.product.ProductGrid', {
  extend: 'Ext.grid.Panel',
  xtype: 'product-ProductGrid',
  store: 'Products',
  columns: [
    {
      text: 'Name',
      dataIndex: 'Name'
    },
    {
      text: 'Description',
      dataIndex: 'Description',
      flex: 1
    },
    {
      text: 'Quantity',
      dataIndex: 'Quantity'
    }
  ]
});
```

```
    },  
    {  
        text: 'Price ',  
        dataIndex: 'Price'  
    }  
]  
});
```

You will recognize the `Ext.define` structure as the same as we have used throughout the book with the class name and parent class defined. The `xtype` config lets us define a string that can be used when lazily instantiating the product grid.

The grid component has only two required configurations which we have included in the previous example: a store containing the data to be displayed and an array of columns which define what data is shown.

The store can be an actual store reference or a `storeId` that will be used to look up the store instance—in this case the products store we defined earlier.

Our columns definition contains an array of configuration objects that will be used to instantiate `Ext.grid.column.Column` classes (or subclasses thereof, for example, `Date`, `Template`, `Number`, and so on). The `text` property will define the column's header text and the `dataIndex` is the field to map to within the store's records. In our example, we have opted to display all the products' fields.

Finally, we can include the new `ProductGrid` in our application and see it in action. We must first require the new component in the `views` array of `Application.js`:

```
...  
    views: [  
        'product.ProductGrid'  
    ]  
    ...
```

We can then use the `xtype` in the Main view's items collection, placing it in the center region:

```
...  
{  
    region: 'center',  
    xtype : 'product-ProductGrid'  
}  
...  

```

Reloading our application should show a grid with three rows of data:

Name	Description	Quantity	Price
Product 1	Product 1 Description	10	9.99
Product 2	Product 2 Description	77	2.99
Product 3	Product 3 Description	1000	5.49

Customizing column displays

So far, our grid simply displays the data values as they are held in the model, which is fine for simple values but some data would benefit from more advanced formatting.

There are multiple ways of achieving the same effect when it comes to formatting column values. We will cover the two main options: column renderers and template columns.

Column renderers

We will start by adding a bit of color to our grid by customizing the styling of the **Quantity** column when it starts to get low. We will make the figures red when they drop to 3 or below, and orange when between 7 and 3.

A `column renderer` is a function that allows us to manipulate the data value (without affecting the underlying stored value) before it is displayed. We add the `renderer` property to the column definition and give it a function with the following parameters:

- **value**: The value of the bound model field
- **metaData**: Additional attributes of the cell being rendered, for example, `tdCls`, `tdAttr`, and `tdStyle`
- **record**: The record for the current row
- **rowIndex**: The index of the current row
- **colIndex**: The index of the current column
- **store**: The store that is bound to the grid
- **view**: The grid view

The renderer function should return a string that will then be displayed in the cell. The following code shows our `renderer` function:

```
{
  text : 'Quantity',
  dataIndex: 'Quantity',
```



```
renderer : function(value, metaData, record, rowIndex, colIndex,
store, view) {
    var colour = 'black';
    if(value <= 3){
        colour = 'red';
    } else if(value > 3 && value <= 7){
        colour = 'orange';
    }
    return '' + value + '';
}
```

Name	Description	Quantity	Price
Product 1	Product 1 Description	1	9.99
Product 2	Product 2 Description	5	2.99
Product 3	Product 3 Description	1000	5.49

Template columns

As we mentioned earlier, there are various other column types that offer additional features. The template column allows us to define an `Ext.XTemplate` that is merged with the row's record. This is particularly useful if we want to include more complex HTML in the cell.

The next example shows how we can include the `StockValue` field besides the `Price`. We start by adding an `xtype: 'templatecolumn'` property to the column definition and then by defining a `tpl` string which will be converted into an `Ext.XTemplate` instance:

```
{
    xtype: 'templatecolumn',
    width: 200,
    text: 'Price ',
    dataIndex: 'Price',
    tpl: '£{Price} (£{StockValue})'
}
```

Name	Description	Quantity	Price
Product 1	Product 1 Description	1	£9.99 (£9.99)
Product 2	Product 2 Description	5	£2.99 (£14.95)
Product 3	Product 3 Description	1000	£5.49 (£5490)

Grid widgets

Rendering complex components in grids has long been a desire of Ext JS developers and has now been made much easier with the introduction of grid widgets. These are components that are rendered inside grid cells and bound to a model field. Examples of grid widgets are buttons, mini graphs, form fields, and so on.

We are going to add a simple button widget to our grid to allow each product's details to be viewed. We start by adding a new `widgetcolumn` to our grid's column array. The `widget` option defines the kind of widget displayed:

```
{
  xtype : 'widgetcolumn',
  width : 100,
  text  : 'Action',
  widget: {
    xtype : 'button',
    text  : 'Details'
  }
}
```

This will render a button on each row that we can then hook up to a click event to open up a product details view (see the next section for further details on doing this).

Name	Description	Quantity	Price	Action
Product 1	Product 1 Description	1	£9.99 (€9.99)	Details
Product 2	Product 2 Description	5	£2.99 (€14.95)	Details
Product 3	Product 3 Description	1000	£5.49 (€5490)	Details

We can hook into the widget's events using the `listeners` config as we would if we were using the component anywhere else in our application. The following snippet shows how to attach a simple `click` handler to the button:

```
{
  xtype : 'widgetcolumn',
  width : 100,
  text  : 'Action',
  widget: {
    xtype : 'button',
    text  : 'Details',
    listeners: {
      click: function(btn){
        var rec = btn.getWidgetRecord();
```

```
        console.log('Widget Button clicked! - ', rec.get('Name'));
    }
}
}
```

Inputting data with forms

A key aspect of web applications is forms. Being able to input data into our system is imperative, and as such, Ext JS has form components to suit all types of input that can be easily bound to our data models and related views.

In this section, we will expand on our product grid and allow users to edit products through a simple form.

Defining a form

We define our product form as we would any other view – by creating a file (in this case named `view/product/ProductForm.js`) and calling `Ext.define` within it. We extend the `Ext.form.Panel` class and give it an `xtype` of `product` – `ProductForm`:

```
Ext.define('BizDash.view.product.ProductForm', {
    extend: 'Ext.form.Panel',
    xtype: 'product-ProductForm'
});
```

Within our form, we want to have input fields for the product's name, description, quantity, and price. There are numerous specific form field types we can use to suit most data types: `text`, `number`, `textarea`, `combobox`, `time`, `file`, `date`, `html`, and more. Check out the `Ext.form.field.*` namespace for all the possibilities.

We will make use of the `text`, `textarea`, and `number` fields for our form. We configure each using their `xtypes` and `fieldLabel`. Additionally, we will give our `textarea` an explicit height and tell our `Price` field that it can have a `decimalPrecision` of 2:

```
Ext.define('BizDash.view.product.ProductForm', {
    extend: 'Ext.form.Panel',
    xtype: 'product-ProductForm',
    items: [
        {
            xtype: 'textfield',
            fieldLabel: 'Name'
        },
    ],
});
```

```

    {
      xtype: 'textarea',
      fieldLabel: 'Description',
      height: 100
    },
    {
      xtype: 'numberfield',
      fieldLabel: 'Quantity'
    },
    {
      xtype: 'numberfield',
      fieldLabel: 'Price',
      decimalPrecision: 2
    }
  ]
});

```

Now we have our input fields ready we will also add two buttons: Save and Cancel. We use the `bbar` config, which is a shortcut to adding a bottom docked toolbar, and define two buttons:

```

...
  items: [ ... ],
  bbar : [
    {
      xtype: 'button',
      text: 'Save'
    },
    {
      xtype: 'button',
      text: 'Cancel'
    }
  ]
...

```

Displaying our form

We want to open our form when a user clicks the **Details** button beside a product in the grid and we want it to be pre-populated with that product's information.

We start by creating a ViewController called `product.ProductGridController`, which will be attached to the `ProductGrid` view:

```

Ext.define('BizDash.view.product.ProductGridController', {
  extend: 'Ext.app.ViewController',

```

```
    alias: 'controller.ProductGrid'
  });
```

This class extends the `Ext.app.ViewController` class and we give it an alias of `ProductGrid`. We use this alias to tie it back to our `ProductGrid` view by adding `controller: 'ProductGrid'` to the `ProductGrid` class.

We will now use `ProductGridController` to listen for a click on the **Details** button and create and show our `ProductForm`.

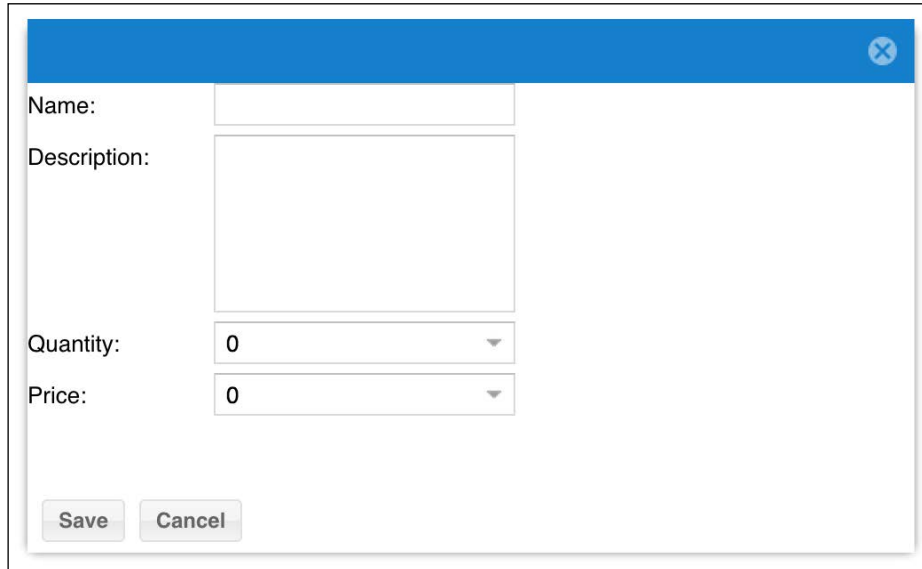
The click handler we added to the **Details** button in the previous section must be replaced with the name of the `ViewController` method that we want to execute when that event happens. We will call this `onDetailsClick`:

```
// ProductGrid.js
{
  xtype : 'widgetcolumn',
  width : 100,
  text : 'Action',
  widget: {
    xtype : 'button',
    text : 'Details',
    listeners: {
      click: 'onDetailsClick'
    }
  }
}
```

The `onDetailsClick` method of `ViewController` will instantiate a `ProductForm` instance, with some extra configuration to size and float it above the existing UI, and then show it:

```
onDetailsClick: function(btn) {
  var rec = btn.getWidgetRecord(),
  productForm = Ext.create('BizDash.view.product.ProductForm', {
    floating: true,
    modal : true,
    closable: true,
    center : true,
    width : 500,
    height : 300
  });
  productForm.show();
}
```

If we run our app, we will see the form display as a modal window above our grid.



Populating our form

At the moment, our product form is displayed but it isn't populated with the details of the product we clicked on. To do this, we must bind each of the form fields to the product model via a `ViewModel`.

We start by creating a `ProductFormModel` class with a very simple configuration. The class extends the `Ext.app.ViewModel` class and is given an alias of `ProductForm`.

We also define the `data` property, which holds the data to be bound, with a `rec` property which will be a reference to the product model instance we want to display:

```
Ext.define('BizDash.view.product.ProductFormModel', {
    extend: 'Ext.app.ViewModel',
    alias: 'viewmodel.ProductForm',
    data: {
        rec: null
    }
});
```

This class is then tied to our `ProductForm` class with its alias:

```
// ProductForm.js
viewModel: {
  type: 'ProductForm'
}
```

Now that these two are tied together, we can make use of the `bind` option of each form field to tell the framework to populate the field with a value from this `viewModel`. We can also do this with the form's `title` property, so it displays the product name:

```
...
  bind: {
    title: '{rec.Name}'
  },
  items: [
    {
      xtype      : 'textfield',
      fieldLabel: 'Name',
      bind       : '{rec.Name}'
    },
    {
      xtype      : 'textarea',
      fieldLabel: 'Description',
      height     : 100,
      bind       : '{rec.Description}'
    },
    {
      xtype      : 'numberfield',
      fieldLabel: 'Quantity',
      bind       : '{rec.Quantity}'
    },
    {
      xtype      : 'numberfield',
      fieldLabel: 'Price',
      decimalPrecision: 2,
      bind       : '{rec.Price}'
    }
  ]
}
...
```

The final piece of the puzzle is to give the `ViewModel` a reference to the correct product model instance. We do this in the `onDetailsClick` method of `ProductGridController`:

```
// ProductGridController.js
...
productForm.getViewModel().setData({
  rec: rec
});
...
```

Now, when we open the form, the fields will be prepopulated. You will also notice that when you edit the product name, for example, the grid and form title will automatically update in real time. Pretty clever.

Persisting updates

At the moment, our `Save` and `Cancel` buttons don't do anything, so let's hook them up.

We start by defining what happens when the buttons are clicked. We use the same pattern as we did with the product grid's `Details` button and assign the click event a method name which correlates to a method in the `ViewController`:

```
// ProductForm.js
bbar: [
  {
    xtype: 'button',
    text: 'Save',
    listeners: {
      click: 'onSave'
    }
  },
  {
    xtype: 'button',
    text: 'Cancel',
    listeners: {
      click: 'onCancel'
    }
  }
]
```


Now, we must create a `ViewController` called `ProductFormController`. Just like we did with `ProductGrid`, we have to tell the view that the `ViewController` exists by adding `controller: 'ProductFormController'`. The `ViewModel` and `ViewControllers` must also be *required* by the view, so they are available:

```
requires: [
  'BizDash.view.product.ProductFormController',
  'BizDash.view.product.ProductFormModel'
],
```

Inside the `ViewController`'s handlers, we use the `getView` method to retrieve a reference to the form, so we can act on it. The `onSave` method will retrieve a reference to the product record (via the `ViewModel`) and commit it (alternatively, if you have a proper backend setup, you would call the `save` method). The `onCancel` method calls the `reject` method on the product model, so any changes that have been made are reverted. Both methods will then close the form and destroy it.

```
//ProductFormController.js
onSave: function(btn) {
  var productModel = this.getView().getViewModel().getData().rec;
  productModel.commit();
  this.closeForm();
},
onCancel: function(btn) {
  var productModel = this.getView().getViewModel().getData().rec;
  productModel.reject();
  this.closeForm();
},
closeForm: function() {
  var productForm = this.getView();
  productForm.close();
  productForm.destroy();
}
```

Data-bound views

So far, we have seen how to display data sets in trees and grids, but what if we want a bit more flexibility in how things are displayed?

Data views give us this freedom and allow us to define custom HTML to be displayed for each item in the dataset but while retaining the powerful automatic binding setup. We can then apply our own custom CSS to this HTML to style it as we want.

We will work through an example where we create a data view for the users in our system.

Defining our users' data view

As always, we create a new view class—`BizDash.view.user.UsersView`—extending the base framework class `Ext.view.View`:

```
Ext.define('BizDash.view.user.UsersView', {
    extend: 'Ext.view.View',
    alias: 'widget.user-UsersView'
});
```

A data view requires three properties to function fully: a store, a `tpl`, and an `itemSelector`.

Store

In exactly the same way as the grid and tree we've looked at previously, this is the data source that the component will be bound to. Any changes made to this store or its records will automatically be reflected in the view.

In this instance, we will use the users store and will configure it using the `storeId`:

```
Ext.define('BizDash.view.user.UsersView', {
    extend: 'Ext.view.View',
    alias: 'widget.user-UsersView',
    store: 'Users'
});
```

Template

A data view is based on the concept of rendering a piece of HTML for each record in the bound data store. The `tpl` config defines `Ext.XTemplate` that will be merged with a record to produce that piece of HTML. Without this, nothing will be rendered to the screen.

We want to display each user's name, role, and photo. We define the data view with the following template string:

```
tpl: [
    '<tpl for=". ">',
    '    <div class="user-item">',
    '        ',
    '        <div class="name">{Name}</div>',
    '        <div class="role">{Role}</div>',
    '    </div>',
    '</tpl>'
].join('')
```

We start by using the `tpl` markup tag along with the `for` attribute. This tells the template to loop over the array it is given and merge the HTML within it with each record in turn. We mark data placeholders, whose names match the model's field names, using the `{{ . . }}` notation.

Item selector

Finally, for our view component to know how to identify a single record's HTML, we must give it a CSS selector that can identify a single node. This is used to allow events to be raised on individual items (for example, click, double-click, and so on).

We have given the wrapper div the `user-item` class, which we can use as the unique selector:

```
...
    itemSelector: 'user-item'
...
```

Styling the view

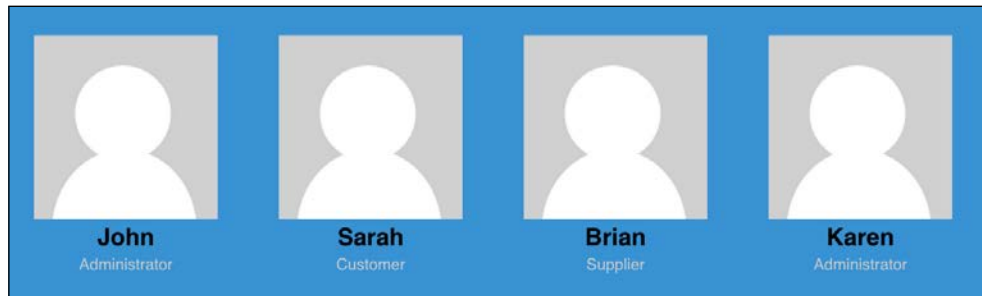
If you show our data view, it will look a bit poor, so we need to include some styling to make it pretty.

We do this by creating a `.scss` file called `UsersView.scss` within the `sass/src` folder and mirroring the view folders structure, so the SASS is automatically picked up. In this case, we create it in `sass/src/view/user/UsersView.scss`:

```
.user-item
{
    float: left;
    width: 120px;
    text-align: center;
    margin: 20px;
    img {
        width: 100%;
    }
    .name
    {
        font-size: 1.2em;
        font-weight: bold;
    }
    .role
```

```
{  
  font-size: 0.8em;  
  color: #CCC;  
}  
}
```

Following a `sencha app build` command and a refresh, you should see a nicer-styled data view, as follows:



Summary

In this chapter, we have learnt how Ext JS' components fit together and the lifecycle that they follow when created and destroyed.

We then moved on to utilizing some of the most popular and useful components. We have explored the details of the following components within our BizDash application:

- Trees
- Grids
- Forms
- Data views

In addition to explaining the main features and configuration options of these components, we linked them together within a simple MVVM architecture, taking advantage of two-way data binding and event listening.

In the next chapter, we will introduce Ext JS themes and how they can be customized to make your application unique.

8

Creating a Unique Look and Feel with SASS

Ext JS has come a long way since its beginning in terms of its design and the ease with which a new design can be applied to the framework.

Its original blue theme was something that made it very appealing to developers but has since become dated and overused. Ext JS 5 now includes six themes and a host of options to customize them and to create your own.

In this chapter, we will discuss how to:

- Apply different themes to your application
- Create your own custom theme
- Customize basic application visuals with SASS variables
- Create custom component UIs

Applying themes to your application

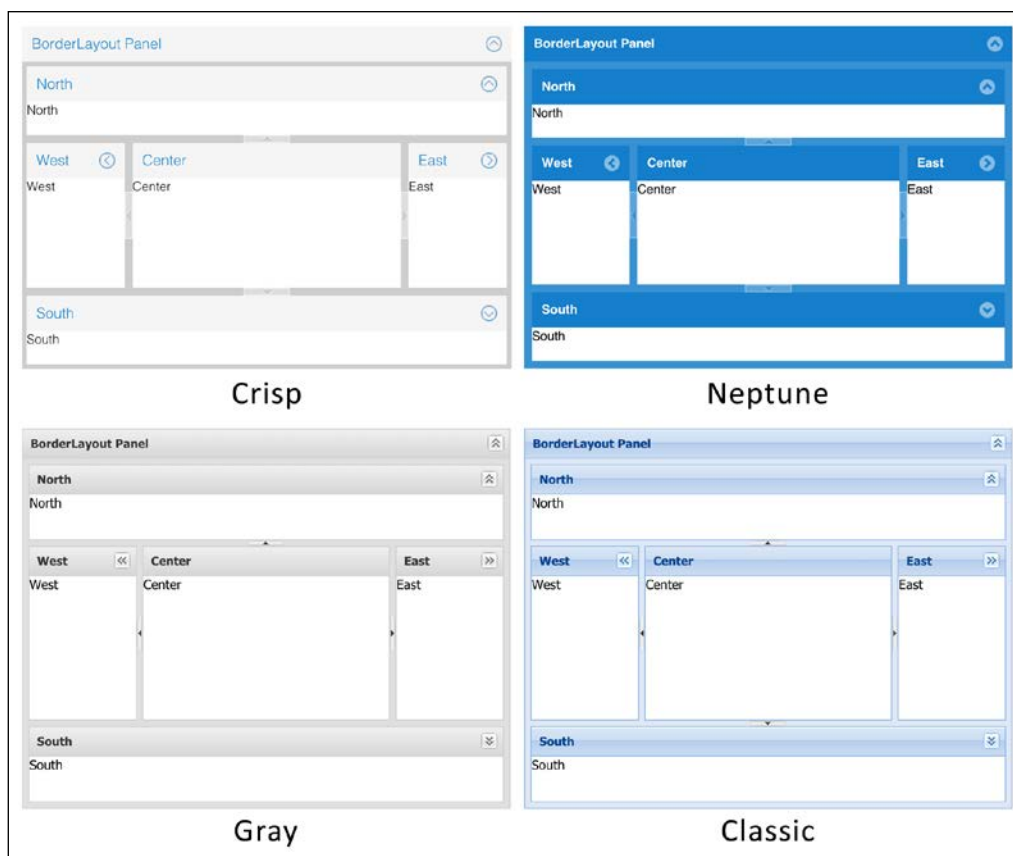
When you generate a new Ext JS application with Sencha Cmd, it will automatically use the Neptune theme. This is one of the newest themes and looks very clean and modern:

Navigation	Name	Description	Quantity	Price	Action
Home	Product 1	Product 1 Description	1	£9.99 (£9.99)	Details
Users	Product 2	Product 2 Description	5	£2.99 (£14.95)	Details
Manage Users	Product 3	Product 3 Description	1000	£5.49 (£5490)	Details
Add User					

As mentioned, there are a total of six themes packaged with Ext JS, as follows:

- Neptune
- Neptune Touch (touch-friendly version of Neptune)
- Crisp
- Crisp Touch (touch-friendly version of Crisp)
- Classic
- Gray

Have a look at these themes in the following screenshot:



If you navigate to the `ext` directory in your project and have a peek into the `packages` folder, you will see all of the theme packages that are available.

Configuring a new theme

You can change the theme your application uses easily, with only one configuration change. In your application's folder, open the `app.json` file. This file is used to configure different aspects of your application and how it is loaded and built.

You should see an item named `theme` near the top of the file. We will edit the value of this property and change it to `ext-theme-crisp`:

```
"theme": "ext-theme-crisp"
```

For this change to take effect, we must rebuild the application using the following command from the `BizDash` folder:

```
sencha app build
```

This regenerates the CSS file for the application and includes the new theme's styles in it.

Refreshing the application in the browser should now show it with the Crisp theme applied:

Navigation	Name	Description	Quantity	Price	Action
 Home	Product 1	Product 1 Description	1	£9.99 (€9.99)	Details
►  Users	Product 2	Product 2 Description	5	£2.99 (€14.95)	Details
	Product 3	Product 3 Description	1000	£5.49 (€54.90)	Details

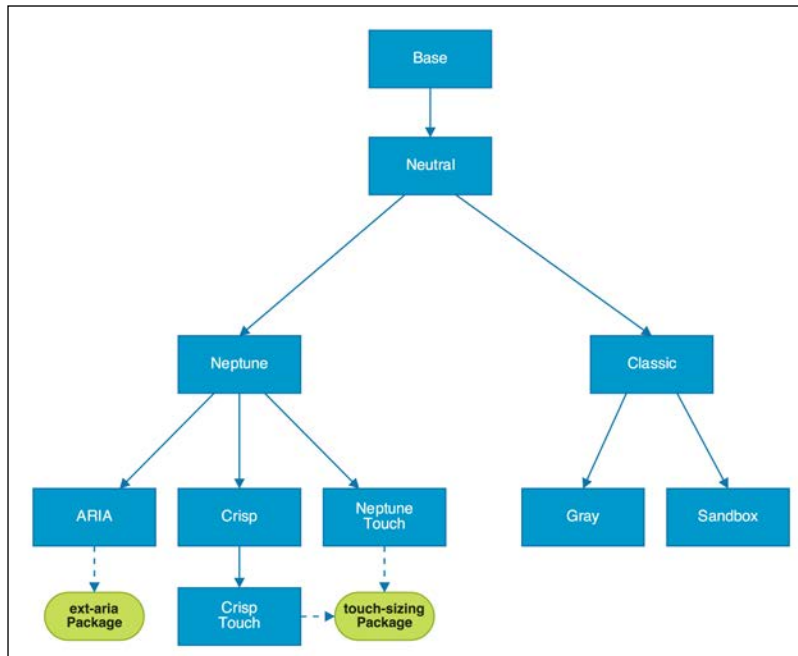
Creating a custom theme

Now that we know how to switch our application's theme, we will move on to creating our own custom theme.

Theme architecture

As you may have noticed from our foray into Ext JS' `packages` folder, themes are architected as packages in the same way that Sencha Core and Sencha Charts are. This means they are portable between applications and sit in isolation from our app's code.

Theme packages are also built with an inheritance hierarchy where they build upon common theme packages. The following diagram shows how each package relates to the others and how the basic styling is shared among each one:



Generating a theme package

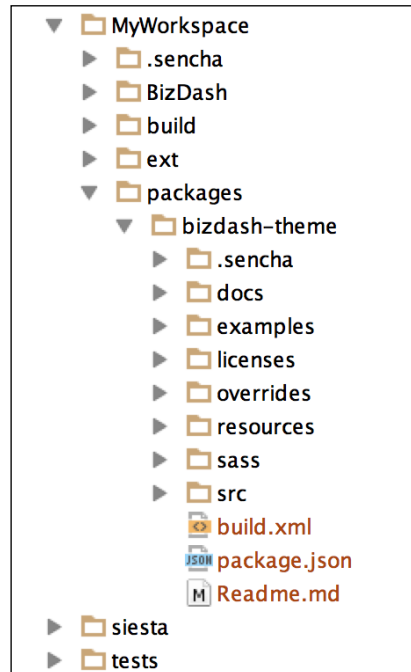
First, we must generate a blank theme package using Sencha Cmd. To do this, we open the terminal and navigate to our BizDash project folder.

Next, we run the following command to generate a basic new theme:

```
sencha generate theme bizdash-theme
```

We use the generate command, which you may remember from earlier in the book, but this time we tell it to generate a theme called bizdash-theme.

Once executed, we should see a new folder in our workspace's packages folder:



Anatomy of a theme

Within our new theme package, there should be all of the files and folders you expect in a regular code package. The main ones we are interested in dealing with are:

- `package.json`: This is where all the package's details and configurations are defined
- `sass/var`: This will be where we put all of our SASS variable overrides
- `sass/src`: This is where we define styling for individual components
- `sass/etc`: This is where any miscellaneous SASS files, which don't tie directly to a component file, can be placed

Cross-browser styling

One of the great benefits of Ext JS is its legacy browser support. So, how do our shiny new themes cope with these older browsers? Ext JS is very clever when rendering in different browsers and uses different styling for browsers that don't support CSS3 properties (such as gradients and rounded corners). These older browsers are given image sprites to display these design features, so the exact same design is replicated across all browsers.

During the Sencha Cmd build process, a sample page, which contains all of the framework's components, is rendered in a headless browser and a snapshot of it is taken. This snapshot is then sliced up into the required sprites and used as needed.

Theme inheritance

As we mentioned before, themes extend other themes and build upon the styling each level defines. By default, our new theme will extend the `ext-theme-classic` theme and will look identical to it.

We can change our new theme's base theme by opening the package's `package.json` file in our IDE. In this file, we can update the `extend` property with the name of the theme we want to extend. We can change it to extend the Crisp theme:

```
"extend": "ext-theme-crisp"
```

Applying the new theme

Now that we have a basic theme in place, we can apply it to our application in the same way that we applied one of the built-in themes. We modify our application's `app.json` file and change the included theme name to `bizdash-theme`:

```
"theme": "bizdash-theme"
```

After rebuilding the application and refreshing the browser, we will see our application displaying our new theme (although at the moment it will look just like the Crisp theme).

Basic theme customizations

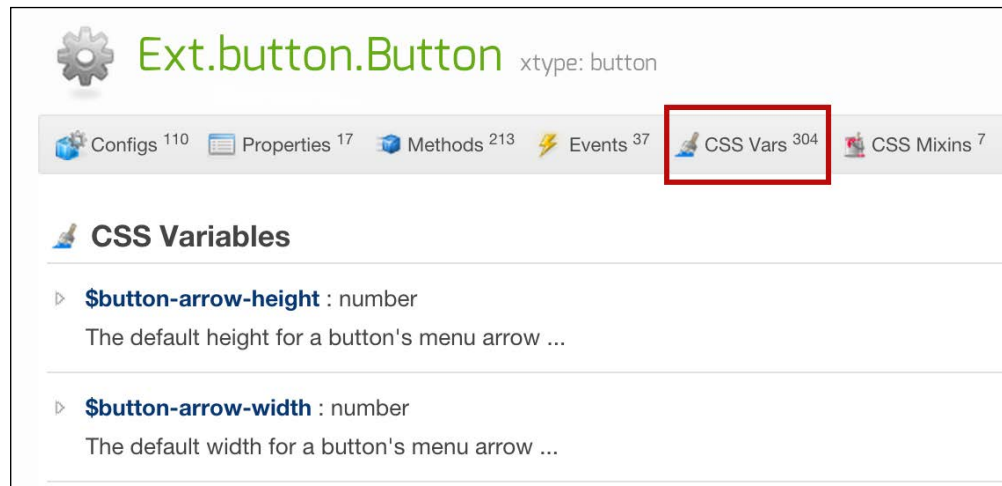
Now that we have a custom theme in place, we want to start making it our own and moving away from the defaults inherited from the base theme.

Theme variables

Ext JS themes are created with SASS and they place a lot of design control in the hands of SASS variables. By modifying these variables, we can make fundamental alterations to our theme very easily.

SASS is a CSS preprocessor that introduces a more terse and functional way of writing CSS. It introduces concepts such as variables, nested rules, selector inheritance, and mixin, which are extremely useful when writing CSS for large applications.

Each component in the framework has its own set of variables that will modify the look of that particular component. We can find a list of variables in the documentation alongside the methods, configs, and events that that component defines.



To define one of these variables in our theme, we must create a new SCSS file in the `sass/var` folder of our theme package. These SCSS files should match the structure of our JavaScript files. For example, defining variables for the `Ext.button.Button` class should be placed in a file named `sass/var/button/Button.scss`. This pattern should be followed when creating new styles in the `sass/src` folder.

We will now demonstrate how we can customize some areas of the UI with SASS variables.

Changing the main color


A common scenario is that we want to change the main color of our theme to match our corporate colors. This can be done easily with the `$base-color` variable, which can be assigned any valid HTML color code.

This variable forms part of the `Ext.Component` class, so it must be defined in the file named `sass/var/Component.scss`. To change the base color to red, we include the following code in the file:

```
$base-color: #FF0000 !default;
```

The `!default` suffix will allow the variable to be overridden in themes that extend this one.

If we rebuild our application and refresh the browser, we will see the effect this has on our application:

Navigation	Name	Description	Quantity	Price	Action
 Home ►  Users	Product 1	Product 1 Description	1	£9.99 (£3.99)	Details
	Product 2	Product 2 Description	5	£2.99 (£14.95)	Details
	Product 3	Product 3 Description	1000	£5.49 (£5490)	Details

Changing the font size

We can also quickly change the font size used throughout the application by including the `$font-size` variable with a new size value. We increase the font size to 16 px with the following code:



```
$font-size: 16px !default;
```

Changing a button's color

We have seen that changing the `$base-color` variable resulted in our buttons rendering with that new color. We can select a different color for the buttons by overriding the `$button-default-background-color` variable.

We create a new SCSS file named `sass/var/button/Button.scss` and add the following code to change the color:

```
$button-default-background-color: #0000FF !default;
```

Navigation	Name	Description	Quantity	Price	Action
 Home ►  Users	Product 1	Product 1 Description	1	£9.99 (£9.99)	Details
	Product 2	Product 2 Description	5	£2.99 (£14.950000000000000...	Details
	Product 3	Product 3 Description	1000	£5.49 (£5490)	Details

Custom component UIs

Ext JS components can be individually customized by providing a different `ui` configuration when they are created. This alters the CSS classes that are added to the components giving them a different look, in isolation from other components of that type. For example, you might want an action button to be green and a cancel button to be gray, but all the other buttons to be the default color.

Defining UIs

We define a UI by including a SASS mixin and configuring it with our required colors and settings. A SASS mixin is a set of style rules that are grouped together, so they can be reused in multiple places and customized by passing parameter values to them.

We will create the two UIs we mentioned earlier in this chapter for the `save` and `cancel` buttons of our `ProductForm`.

We start by creating a `Button.scss` file in a `sass/src/button` folder within our `bizdash-theme` package folder. This folder structure mirrors the `Ext.button.Button` component class and so will be picked up and compiled during a `sencha app build` process.

In this file, we include the `extjs-button-small-ui` mixin:

```
@include extjs-button-small-ui( );
```

We then define the name of the UI, which will be used to apply the styling to a button. We use the `$ui` parameter name and give it the name `action`:

```
@include extjs-button-small-ui(  
    $ui: 'action'  
);
```

Next, we define the button's background color, its text color, and its border color:

```
@include extjs-button-small-ui(  
    $ui: 'action',  
    $background-color: #008000,  
    $color: #FFFFFF,  
    $border-color: transparent  
);
```

We can repeat this for the `cancel` button and add a similar mixin to the `Button` scss file:

```
@include extjs-button-small-ui(  
    $ui: 'cancel',  
    $background-color: #EBEBEB,  
    $color: #000000,  
    $border-color: transparent  
);
```

Now that these are in place, we rebuild the application using the `sencha app build` command within the `BizDash` folder.

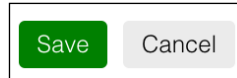
Applying UIs

We now have two UI styles ready to apply to buttons within our application. To apply these, we use the `ui` configuration option. When added, this option adds a new CSS class, which was generated by our mixins in the previous step, to the components:

In the `ProductForm` class our button configs become:

```
...  
    bbar : [  
        {  
            xtype      : 'button',  
            text       : 'Save',  
            ui          : 'action',  
            listeners: {  
                click: 'onSave'  
            }  
        },  
        {  
            xtype      : 'button',  
            text       : 'Cancel',  
            ui          : 'cancel',  
            listeners: {  
                click: 'onCancel'  
            }  
        }  
    ]  
    ...
```

Upon refreshing the application in our browser, we can see the new button styling in place. Inspecting the DOM shows the new CSS classes added to each of the buttons:



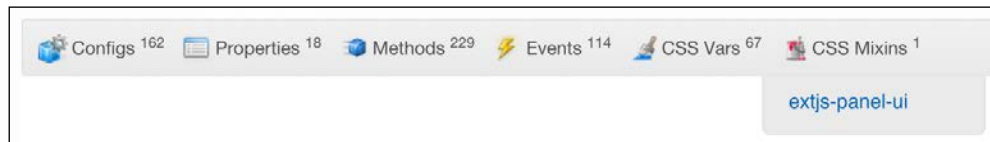
The following screenshot shows the new CSS classes:

```
> <a class="x-btn x-unselectable x-box-item x-toolbar-item x-btn-confirm-small" hidefocus="on" unselectable="on" id="button-1034" tabindex="0" componentid="button-1034" style="right: auto; left: 0px; top: 0px; margin: 0px;">
- </a>
> <a class="x-btn x-unselectable x-box-item x-toolbar-item x-btn-cancel-small" hidefocus="on" unselectable="on" id="button-1035" tabindex="0" componentid="button-1035" style="right: auto; left: 53px; top: 0px; margin: 0px;">
- </a>
```

Other UIs

Most components within the framework have mixins that allow us to define a different component style. We will now demonstrate how to create an alternative UI for an `Ext.Panel` component.

If you find the `Ext.Panel` in the `Ext JS` documentation, you will see one item under the **CSS Mixins** dropdown. We will use this to define our own UI, we first 'create' a `Panel.scss` file in a folder `sass/src/panel/Panel`.

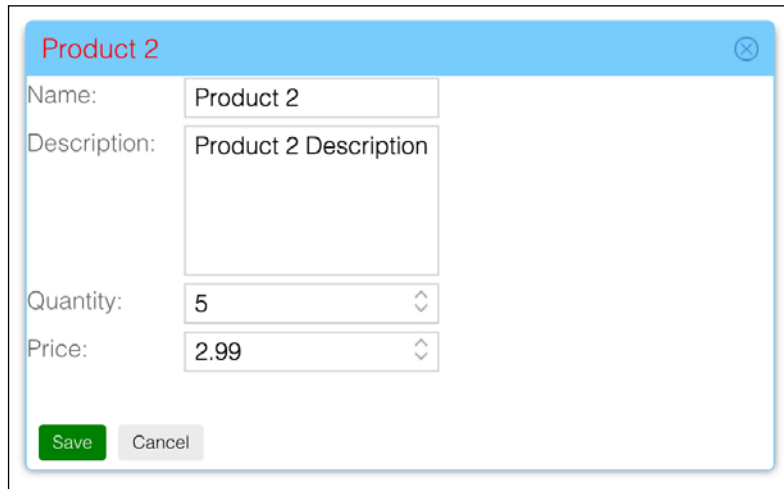


We include this in our `Panel.scss` file and start to configure the styling we want. You can see all of the available options in the documentation, but we will customize the header background color, border color, and border radius, as follows:

```
@include extjs-panel-ui(
  $ui: 'product',
  $ui-border-color: #78CCFC,
  $ui-border-radius: 5px,
  $ui-header-background-color: #78CCFC
);
```


This UI option can be added to the `ProductForm` in exactly the same way as we did with the buttons, using the `ui` config:

```
...   ui: 'product',  
...
```



Summary

In this chapter, we have looked at how we can customize the look and feel of our Ext JS applications through the use of custom themes. We have looked at how themes are constructed and how they inherit from each other.

We have also demonstrated how to create and customize a new theme through the use of global SASS variables and component mixins.

The next chapter will focus on visualizing data within our applications through the use of charts.

9

Visualizing Your Application's Data

This chapter will demonstrate how to visualize your data with charts and graphs. It will focus on the most popular types of graphs and how your data sources can be integrated with them. We will also discuss how to integrate visualizations into other components, such as grids.

We will cover the most common topics relating to charts. These are as follows:

- Understanding visualization technologies and how charting works in Ext JS
- Examples and explanations of how to work with common charts, namely:
 - Line
 - Bar
 - Pie
- Integrating these visualizations into other components, such as a grid

Anatomy of chart components

Ext JS provides a very flexible and feature-rich charting toolkit to visually represent data to users. To get the most out of the chart's package, it is worthwhile to have an understanding of the anatomy of the chart.

Series

In Ext JS, the series is essentially the chart type. The series is the most complicated part of the chart and handles how the elements are animated, shown, or hidden. In addition to this, the series handles how the data is labeled.

If, for example, the application requires a bar chart, then it's the Bar or Bar3D series classes you need to look out for.

Out of the box, Ext JS 5 offers the following charts:

- Bar (and a 3D variant)
- OHCL/Candlestick
- Area
- Pie (and a 3D variant)
- Line
- Gauge
- Scatter
- Radar
- Polar

Axes

As with any other chart, you'll typically be required to define axes for your chart components. The numeric and category axes are the most commonly used. The framework comes with a number of layouts and a method to segment data if you require something that's not necessarily out of the box.

Labels

Chart labels are simply a way to display the label for the data point. Ext JS enables us to customize how these are displayed and animated.

Interactions

Interactions simply refer to how the user interfaces with the chart. For example, using an interaction might allow the user to click and drag a part of the chart to zoom into the data.

Creating a line chart

Line charts are one of the most commonly used types of graphs and are most suited to representing trending data, which is often regularly updated and required to be analyzed in real time.

One of the great advantages of the Ext JS charting package is that we can work with the data in real time. It's possible to do this a number of ways:

- Simply poll periodically for updates
- Bind the store to a `WebSocket` and push updates from the server side

To continue the running theme of our BizDash business dashboard, we will create a line chart that shows how many people are currently visiting our website.

Creating the store and model

As we saw in *Chapter 5, Modeling Data Structures for Your UI*, we need to start by defining a model to represent an individual record being retrieved by the store.

Before we get into the details of creating the line chart, we need to create a store and a model. We don't yet have a `WebSiteVisitor` model in our application. So, let's create one:

```
// app/model/WebSiteVisitor.js
Ext.define('BizDash.model.WebSiteVisitor', {
    extend: 'Ext.data.Model',
    fields: [
        {
            name: 'Time',
            type: 'int'
        },
        {
            name: 'Visitors',
            type: 'int'
        }
    ]
});

// app/store/WebSiteVisitors.js
Ext.define('BizDash.store.WebSiteVisitors', {
    extend: 'Ext.data.Store',
    model: 'BizDash.model.WebSiteVisitor',
    proxy: {
        type: 'ajax',
        url: 'visitors.json',
        reader: {
            type: 'json',
            rootProperty: 'rows'
        }
    }
});
```

Polling the server for new data

It would be straightforward to have our store poll the server every 10 seconds and plot the number of visits at that moment in time with a `setInterval` or similar:

```
setInterval(function() {
  store.load({
    addRecords: true
  });
}, 10000);
```

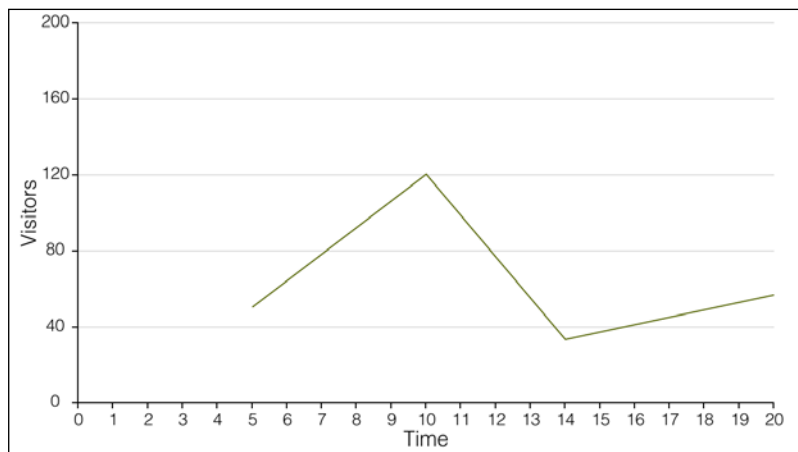
The `addRecords` configuration is passed into the store's `load` method. By setting this property to `true`, any new records retrieved by the load operation are appended to the existing dataset rather than replacing what is already there, which is the default behavior.

The configuration of the chart itself is surprisingly straightforward. We'll do this in a new view to remain in keeping with the rest of our application:

```
Ext.define('BizDash.view.chart.SiteVisits', {
  extend: 'Ext.chart.CartesianChart',
  xtype: 'chart-SiteVisits',
  config: {
    animate: true,
    store: 'WebSiteVisitors',
    series: [
      {
        type: 'line',
        smooth: false,
        axis: 'left',
        xField: 'Time',
        yField: 'Visitors'
      }
    ],
    axes: [
      {
        type: 'numeric',
        grid: true,
        position: 'left',
        fields: ['Visitors'],
        title: 'Visitors',
        minimum: 0,
```

```
        maximum : 200,  
        majorTickSteps: 5  
    },  
    {  
        type : 'numeric',  
        position : 'bottom',  
        fields : 'Time',  
        title : 'Time',  
        minimum : 0,  
        maximum : 20,  
        decimals : 0,  
        constrain : true,  
        majorTickSteps: 20  
    }  
    ]  
    }  
    });
```

Here is how it will look:



The line chart we have created is set up in a fairly standard way. The two numeric axes bind to the integer fields within the `WebSiteVisitor` model. The line itself is displayed by the `Ext.chart.series.Line` class where it is tied to the data via the `xField` and `yField` properties.

Presenting data in a bar chart

A bar chart is an incredibly useful way of presenting quantitative data to users. We will quickly demonstrate how to create a bar chart and have it load data asynchronously from the server.

As with the line chart, we're going to require a model and store to get things started:

```
// app/model/BarChart.js
Ext.define('BizDash.model.BarChart', {
    extend: 'Ext.data.Model',
    fields: [
        {
            name: 'name',
            type: 'string'
        },
        {
            name: 'value',
            type: 'int'
        }
    ]
});

// app/store/BarChart.js
Ext.define('BizDash.store.BarChart', {
    extend: 'Ext.data.Store',
    model: 'BizDash.model.BarChart',
    proxy: {
        type: 'ajax',
        url: 'path/to/your/datasource',
        reader: {
            type: 'json',
            rootProperty: 'data'
        }
    },
    autoLoad: true
});
```

In this contrived example, the data we return from the server must have a `name` and `value` property, as we have defined these as fields in our model.

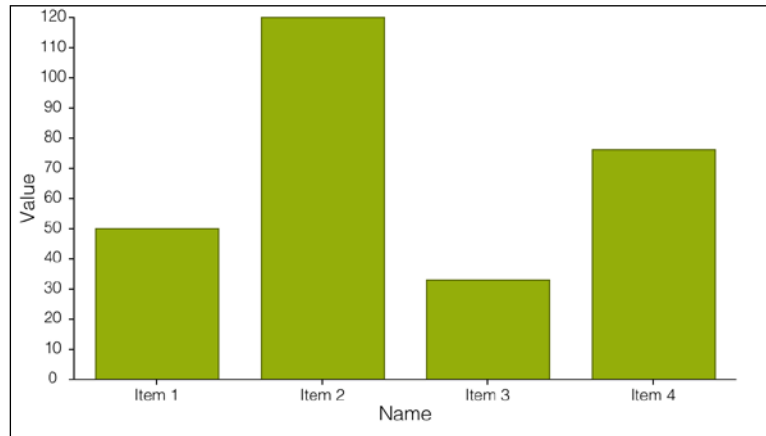
The store we defined in this example is used as the client-side cache for our data. By associating the `BarChart` model to the store, we ensure that the data is correctly represented in the chart component.

Now that the store is ready, we just need a component to bind it to. We will now create a new view called `BizDash.view.chart.BarChart` containing a Cartesian chart with a bar series. This gives us basic charting functionality to which we can add axes and series and bind our store to it.

The axes are used to define the boundaries of the chart and, in this instance, create the horizontal and vertical axes. The series handles the rendering of the data points across the chart. Here, we've used the `Ext.chart.series.Bar` class to create a simple bar chart. In the bar configuration, `xField` and `yField` must contain the same name fields we defined in our model.

```
Ext.define('BizDash.view.chart.BarChart', {
    extend: 'Ext.chart.Chart',
    xtype: 'chart-BarChart',
    config: {
        animate: true,
        store: 'BarChart',
        axes: [
            {
                type: 'numeric',
                position: 'left',
                fields: ['value'],
                title: 'Value'
            },
            {
                type: 'category',
                position: 'bottom',
                fields: ['name'],
                title: 'Name'
            }
        ],
        series: [
            {
                type: 'bar',
                axis: 'bottom',
                xField: 'name',
                yField: 'value'
            }
        ]
    }
});
```


This is how the graph will now look:



Creating a pie chart in Ext JS

The pie chart is a very common chart type and is excellent at representing proportional data, where each slice of the pie equates to the percentage that particular slice holds against the sum of the entire dataset.

In this section, we will demonstrate how to create a pie chart representing the distribution of products (by stock level) we have in our warehouse.

We already have a products store and model defined earlier in the book (*Chapter 5, Modeling Data Structures for Your UI*, and *Chapter 7, Constructing Common UI Widgets*). As a reminder, here's what we defined:

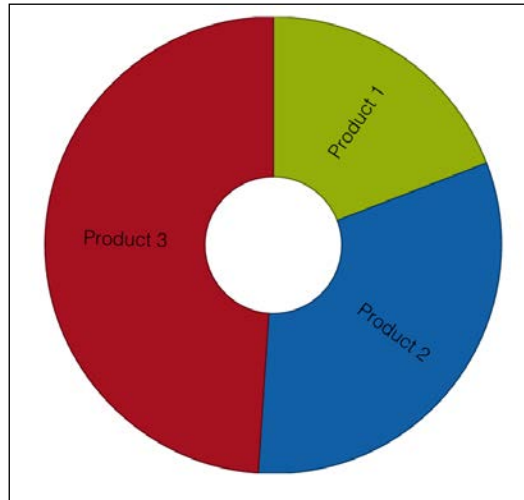
```
Ext.define('BizDash.model.Product', {
    extend: 'Ext.data.Model',
    fields: [
        {
            name: 'Name',
            type: 'string'
        },
        {
            name: 'Description',
            type: 'string'
        },
        {
            name: 'Quantity',
            type: 'int'
        }
    ],
});
```

```
        {
            name: 'Price',
            type: 'float'
        }
    ]
});
Ext.define('BizDash.store.Products', {
    extend: 'Ext.data.Store',
    model: 'BizDash.model.Product',
    autoLoad: true,
    proxy: {
        type: 'ajax',
        url: 'products.json',
        reader: {
            type: 'json',
            rootProperty: 'rows'
        }
    }
});
```

Now that we have the store and model organized, we need to create a new view for our pie chart:

```
Ext.define('BizDash.view.chart.StockLevelPie', {
    extend: 'Ext.chart.PolarChart',
    xtype: 'chart-StockLevelPie',
    config: {
        animate: true,
        store: 'Products',
        interactions: 'rotate',
        series: {
            type: 'pie',
            label: {
                field: 'Name',
                display: 'rotate'
            },
            xField: 'Quantity',
            donut: 30
        }
    }
});
```

Amazingly, it's as simple as that. You can see the output in the following image:



As with other charts that have polar coordinates, the `Ext.chart.PolarChart` class provides the infrastructure and canvas for our specific chart type to be rendered.

We use the pie series type to have the `Ext.chart.series.Pie` class process the records contained in the bound store and turn it into a series of sprites to form a chart. This series will convert each of the records in the store into a slice of the pie.

The most important configuration of this series type is `xField`. This tells the series which of our model fields holds the numeric value that will be used when calculating the size of each record's slice.

We add labels to the chart by using the `labels` property to configure the `Ext.chart.Label` mixin, which is applied to the `Ext.chart.series.Pie` class. These options allow us to configure how the labels are positioned and styled. By choosing the `rotate` value for the `display` property, the labels are positioned along the length of the slice. While discussing the look of the chart, it's perhaps worth highlighting that the `donut` config is the radius of the hole (as a percentage) that's cut out in the middle of the pie chart.

Integrating visualizations in grids

The grid widget is one of the best known features of Ext JS and the framework now supports the concept of widget columns. The widget column lets you embed a widget which, unlike a standard component, is a lightweight component that is quick to render and update. Widgets are suited perfectly to grids, where you may be displaying dozens of them at the same time.

Ext JS 5 introduces a number of lightweight widgets for grids, but the sparkline widget is without doubt the perfect widget to show a chart in a grid. It is an ultra-lightweight chart designed to represent a series of values with minimum real estate.

There are a number of different sparklines to work with:

- Bar
- Line
- Pie
- Bullet
- Range map
- TriState
- Box plot

To demonstrate a sparkline in action, we're going to go back to *Chapter 7, Constructing Common UI Widgets*, and make use of our product grid:

```
Ext.define('BizDash.view.product.ProductGrid', {
    extend: 'Ext.grid.Panel',
    xtype: 'product-ProductGrid',
    store: 'Products',
    columns: [
        {
            text: 'Name',
            dataIndex: 'Name'
        },
        {
            text: 'Description',
            dataIndex: 'Description',
            flex: 1
        },
        {
            text: 'Quantity',
            dataIndex: 'Quantity'
        },
        {
            text: 'Price ',
            dataIndex: 'Price'
        }
    ]
});
```

A sparkline graph must be attached to a single data field, which contains an array of points that will be plotted. To demonstrate this feature, we will add a `HistoricSales` field to our `Product` model which will hold the data that we will plot:

```
Ext.define('BizDash.model.Product', {
    extend: 'Ext.data.Model',
    ...
    fields: [
        ...
        {
            name: 'HistoricSales',
            type: 'auto',
            defaultValue: [4, 9, 12, 66, 9]
        }
    ]
});
```

This grid already has a store and model bound to it, so it's simply a case of adding a new `widgetcolumn` to the `columns` array as follows:

```
Ext.define('BizDash.view.product.ProductGrid', {
    extend: 'Ext.grid.Panel',
    columns: [
        ...
        {
            text : 'Historic Sales',
            xtype : 'widgetcolumn',
            dataIndex: 'HistoricSales',
            widget : {
                xtype: 'sparklineline'
            }
        }
    ]
});
```

The widget config allows us to specify the `xtype` of the sparkline that will be rendered in the column (possible values include `sparklinebar`, `sparklinepie`, `sparklinebullet`, and `sparklinediscrete`). The following screenshot shows how this line graph is rendered:

Name	Description	Quantity	Price	Historic Sales
Product 1	Product 1 Description	1	£9.99 (£9.99)	
Product 2	Product 2 Description	5	£2.99 (£14.95)	
Product 3	Product 3 Description	1000	£5.49 (£5490)	

Summary

Throughout this chapter, we have explored the details of adding charting visualizations to our Ext JS application. We have:

- Covered the basics on how charts are created
- Understood the components of a chart
- Built a line chart
- Built a bar chart
- Built a pie chart
- Integrated a visualization into a grid

The next chapter dives into testing your Ext JS application by teaching you how to write testable JavaScript and unit tests for your Ext JS application.

10

Guaranteeing Your Code's Quality with Unit and UI Testing

Being confident in our code's quality and functionality is of utmost importance when we come to add new features, refactor existing features, and ultimately, ship our product.

Without any process in place to automatically check our code, we will always be nervous about the impact our changes will have on the rest of the application. The time between bugs being introduced and them being discovered is extended, resulting in the cost of resolving them rising dramatically.

This chapter will focus on how we can ensure that our code is of a high standard and that it functions as it should at all times. We will explore the following topics:

- Best practices for writing testable JavaScript
- Development approaches that will put testing front and center
- Introduction to the Siesta testing framework
- How to write unit tests
- How to write UI tests
- How to integrate testing into the development workflow

Writing testable JavaScript

JavaScript has always been a language that has been difficult to test. This has largely been down to the unstructured nature in which it was used and the tendency to sprinkle it into pages to add small effects and pieces of functionality. This meant that isolating small sections of code to test them was extremely difficult and so, it tended to just not be done.

These days, JavaScript is used to write serious applications, and developers are building upon frameworks, such as Ext JS, to introduce a more rigid architecture to their projects. This structure makes testing units of functionality much easier and gives us much simpler scenarios to work with and construct tests around.

Although by writing an application using Ext JS you have already taken a big step towards more testable code, there are some good practices to follow to ensure things are even easier, which we will explore in this section.

Single responsibility

By ensuring that your classes and methods are only responsible for a single piece of functionality, means that testing that functionality is much easier as the inputs and outputs are much more clear-cut.

For example, having a method that, on a button click, will update three parts of your application in one big method will be complex to test. To test that single method, we would need to have all three areas available and check for the appropriate result in each.

If we extracted this into three separate methods, we could test each in isolation and concentrate on that function and its results.

Accessible code

If a function in our code base is **private** (that is, inaccessible to outside code via a closure, for example) then we will be unable to test it or mock it. This approach makes it impossible to test these private functions and very difficult to test methods that rely on them as they may need to be mocked.

It is therefore, sometimes necessary to make these private functions publicly accessible to aid testing.

Nested callbacks

Deeply nested callback functions are extremely difficult to test as they are all private and rely on lots of conditions to align to exercise them all. This relates closely to the previous point of making code accessible. Rather than nesting lots of callbacks, consider extracting them into member functions. The following example shows how the functions handling the `async` calls can't be tested:

```
Ext.define('MyClass', {
  doAction: function() {
    Ext.Ajax.request({
      url: 'action.php',
      success: function(response) {
        OtherClass.async(response.value, function(newValue) {
          AnotherClass.async(newValue, function() {
            // finished!
          });
        });
      }
    });
  }
});
```

This could be refactored to use member functions that can all be tested individually. In addition, the code is much cleaner and avoids lots of nested callbacks.

```
Ext.define('MyClass', {
  doAction: function() {
    Ext.Ajax.request({
      url: 'action.php',
      success: this.onActionSuccess
    });
  },
  onActionSuccess: function(response) {
    OtherClass.async(response.value, this.onOtherClassAsync);
  },
  onOtherClassAsync: function(newValue) {
    AnotherClass.async(newValue, this.onAnotherClassSync);
  },
  onAnotherClassSync: function() {
    // finished!
  }
});
```

Separate event handlers from actions

When attaching functionality to an event, we often carry out the action within the same function. The following example shows a button click handler that makes an AJAX request to delete a user, based on the selected row in a grid:

```
onDeleteButtonClick: function(btn) {
    var grid = btn.up('gridpanel'),
        selectedUserModel = grid.getSelection()[0];
    Ext.Ajax.request({
        url : 'deleteUser.php',
        params : {
            userID: selectedUserModel.get('userID')
        },
        success: this.onUserDeleteSuccess,
        failure: this.onUserDeleteFailure,
        scope : this
    });
}
```

The problem with this code is that to test deleting a user, we need to have a grid available and a row selected, which is cumbersome to set up. What would be better is to extract the action from the handler so that the action can be tested without the need for the setup code. It also has the added benefit of being able to be executed from a different context; for example, from a delete key handler.

```
onDeleteButtonClick: function(btn) {
    var grid = btn.up('gridpanel'),
        selectedUserModel = grid.getSelection()[0];
    this.doUserDelete(selectedUserModel.get('userID'));
},
doUserDelete: function(userID) {
    Ext.Ajax.request({
        url : 'deleteUser.php',
        params : {
            userID: userID
        },
        success: this.onUserDeleteSuccess,
        failure: this.onUserDeleteFailure,
        scope : this
    });
}
```

Testing frameworks

There are a large number of JavaScript testing frameworks which would all work well with Ext JS applications. They each offer different ways of doing things and a different feature set. We will discuss Jasmine (<http://jasmine.github.io/>) and Siesta (<http://www.bryntum.com/products/siesta/>) in this section, and move on to explain Siesta in detail in subsequent sections.

Jasmine

Jasmine is a simple, free, BDD-style framework which is widely used across the industry, including Sencha, to develop Ext JS and Sencha Touch.

This framework takes a very descriptive approach to outlining tests:

```
describe("Test Suite Title", function() {  
    var a;  
    it("Spec Description", function() {  
        a = MyClass.doSomething();  
        expect(a).toBe(true);  
    });  
});
```

Test suites and specs are just simple functions which should execute the test code and then assert whether the outcome was correct.

The `describe` method lets us define a test suite, which can include multiple specs (or indeed nested test suites). Specs are defined with the `it` method, which accepts a description of the spec and a function that forms the test. Within this function, our test is set up and executed, and its results analyzed to ensure the correct result was achieved.

Jasmine has a test harness page which executes all of the included test suites and displays the results. It can also be executed headlessly with a tool such as PhantomJS. This makes it perfect to integrate with your CI process.

Siesta

Siesta is a powerful testing framework which is focused on testing Ext JS and Sencha Touch applications. It allows us to write unit and UI tests for our applications, also boasting a powerful event recorder built in to record and test interactions with specific UI elements. Having a Sencha slant means it has helper methods to assist testing of common components and also supports Sencha style constructs, such as `Component Queries` and `Ext.Loader`.

This Sencha focus makes Siesta perfect for testing Ext JS applications and makes tasks much simpler because of its knowledge of the framework. An added bonus is that it is written with Ext JS and tested using itself, so developers are using their product in anger everyday giving this framework a high degree of developer focus.

Siesta allows you to include multiple specs in a single suite with each spec testing an area of your application:

```
StartTest(function(t) {
    t.diag("MyClass Test");
    var a = MyClass.doSomething();
    t.is(a, true, 'MyClass.doSomething returned true');
    t.done();
});
```

The function passed to `StartTest` contains all of the test logic, including the test steps and the assertions. The object `t` passed into this function gives us access to a large number of assertion methods which we will explore later.

We recommend using Siesta as your testing framework because of its close integration with Ext JS, which makes it extremely easy to implement tests quickly and effectively.

Writing unit tests

Siesta allows us to create unit test suites which allow us to exercise non-UI logic. We will write some simple tests for our `BizDash.config.Config` class to test its methods. A slightly cut down version of this class is shown here:

```
Ext.define('BizDash.config.Config', {
    extend: 'Ext.util.Observable',
    singleton: true,
    config: { version: '0.0.1-0' ... },
    ...
    getBuildNumber: function() {
        var versionSplit = this.getVersion().split('-');
        return versionSplit[1];
    },
    applyVersion: function(newVersion, oldVersion){
        return newVersion;
    },
    updateVersion: function(newVersion, oldVersion){
        if(this.hasListeners) {
```

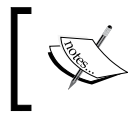
```

        this.fireEvent('versionchanged', newVersion, oldVersion);
    }
}
});

```

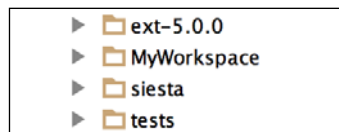
Testing project structure

We will start by adding the Siesta framework files to our project folder; these can be downloaded from the Siesta website (<http://www.bryntum.com/products/siesta/>).

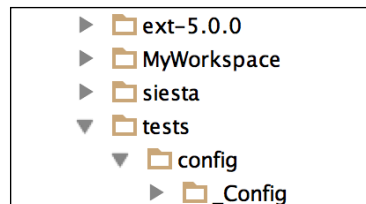


Siesta offers a Lite version, which can be used for free but is limited in its feature set. The standard license gives you access to extra features and support.

We then create a tests folder where we will keep our test suites. Our folder structure should look something like the following image:



Next, we will create a test suite for the `Config` class. We will mirror the app's folder structure and have a folder named the same as the class being tested, prefixed with an underscore. In this case, we end up with the following:



By following this convention, the tests are grouped by their class and so can be easily navigable. This is purely a suggested structure and your test suite can follow an entirely different pattern if you wish.

Creating the test harness

To run Siesta tests, we must have a test definition script (like the one we created in the previous section) and an HTML page that loads the Siesta framework and the definition script.

Our test definition should contain the following code and be within the `index.js` file:

```
var Harness = Siesta.Harness.Browser.ExtJS;
Harness.configure({
  title: 'Config Tests',
  preload: [
    '../..../MyWorkspace/build/testing/BizDash/resources/BizDash-all.
css',
    '../..../MyWorkspace/build/testing/BizDash/app.js' ]
});
Harness.start(
  {
    group: 'Config',
    items: [
    ]
  }
);
```

The first section configures Siesta with a test suite title and tells it to preload the built versions of our application (the ones that are created after running `sencha app build`).

Next, we tell the harness to start running our tests, which we will structure in groups. At the moment we have no tests to run, but when we do, we will reference the files in the `items` array.

Our HTML page is extremely simple and just loads in the Siesta and Ext JS frameworks and our test definition script.

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css" href="http://cdn.
sencha.io/ext/gpl/4.2.0/resources/css/ext-all.css">
  <link rel="stylesheet" type="text/css" href="../../siesta/
resources/css/siesta-all.css">

  <script type="text/javascript" src="http://cdn.sencha.io/ext/
gpl/4.2.0/ext-all.js"></script>
  <script type="text/javascript" src="../../siesta/siesta-all.
js"></script>
  <script type="text/javascript" src="index.js"></script>
</head>
```

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

Adding the first test

We will start by adding a test for the `getBuildNumber` method. Add a new file called `010_getBuildNumber.t.js`.

The naming convention we use here is not required but is used throughout the Siesta documentation. The numbering allows the tests to run in a specific order, with the name referring to the method or feature under test. The `.t.js` extension is used as a way of identifying a test file.

In this file, we call the `StartTest` method:

```
StartTest(function(t) {
  t.diag("Config.getBuildNumber");
  t.done();
});
```

Next, we call the `getBuildNumber` method and check that the result we get back is correct:

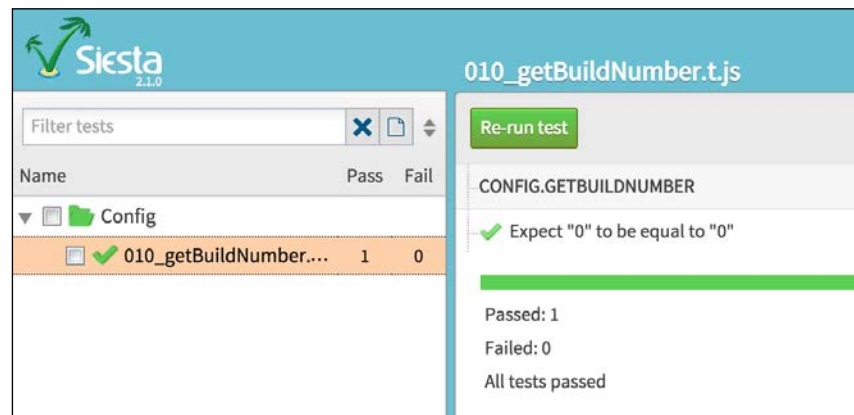
```
StartTest(function(t) {
  t.diag("Config.getBuildNumber");
  var buildNumber = BizDash.config.Config.getBuildNumber();
  t.expect(buildNumber).toEqual('0');
  t.done();
});
```

Finally, we must reference this new file in the `index.js` `items` array:

```
Harness.start(
{
  group: 'Config',
  items: [
    '010_getBuildNumber.t.js'
  ]
}
);
```


Executing tests

To run our tests, we simply open `index.html` in our browser and click the **Run All** button at the bottom left of the screen. Siesta will then execute each test listed and display the results as seen in the following screenshot:



Extending tests

We can add as many assertions and tests to this initial file as we wish, but it is often wise to put different test types in different files to keep things organized and to ensure a clean environment, unsullied by previous tests.

We will now add some new test files to test some edge cases. Each of these files should be added to the `index.js` file, so they are included in test runs.

Our next spec, `020_getBuildNumber_emptyString.t.js`, will test the output when the version number is set to an empty string. We use the `toBeUndefined` method to check the output:

```
StartTest(function(t) {
  t.diag("Config.getBuildNumber");
  BizDash.config.Config.setVersion('');
  var buildNumber = BizDash.config.Config.getBuildNumber();
  t.expect(buildNumber).toBeUndefined();
  t.done();
});
```

Finally, we will use our poor implementation of the `getBuildNumber` method to demonstrate how to test that a method should throw an exception when the version config is null. To do this, we use the `toThrow` assertion, passing to it the method to test:

```
StartTest(function(t) {  
  t.diag("Config.getBuildNumber");  
  BizDash.config.Config.setVersion(null);  
  t.expect(BizDash.config.Config.getBuildNumber).toThrow();  
  t.done();  
});
```

The framework will automatically call the method we pass into `expect` and pass the test if it throws an exception.

Testing UI interaction

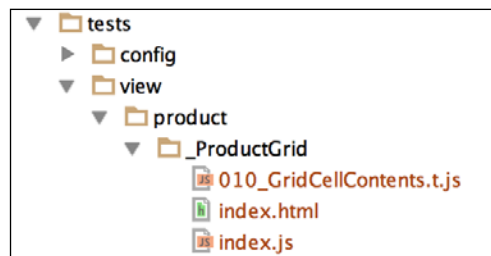
One of the big benefits of Siesta is its ability to test real UI interactions, allowing us to create automated, repeatable tests, based on the way the application is actually rendered and how a real user would interact with it.

In this section, we will test the product grid that we created in *Chapter 7, Constructing Common UI Widgets*. We will look at testing how it has been rendered and then move on to testing clicks on the grid row's **Details** button.

Testing cell contents

To kick off, we are going to test whether each cell of the grid renders the correct values based on the data in the store it is bound to. This will catch any issues with field names changing and grid column configurations.

We start by creating a new test suite, like we did in the previous section, and a new test file called `010_GridCellContents.t.js`. Our folder structure will look like the following:



Be careful to update all the paths in your `index.html` and `index.js` files to suit the new location.

In these test suites, we will include the entire `app.js` file, which means the application will be launched as it would be if you opened it in the browser. This means we can interact with the UI straightaway.

If you expand the DOM panel on the right-hand side of the Siesta interface, it will display the UI as it appears during the test. If you watch this area when tests are running, you will see the interactions happening in real time.

Setting up expected data

It is important that we know what data our grid is rendering, so we can check whether it is correct. For that reason, our first step is to populate our products' store with known test data:

```
StartTest(function(t) {
  t.diag("Product Grid Contents");
  var productsStore = Ext.getStore('Products');
  productsStore.removeAll();
  productsStore.add([
    {
      "id" : 1,
      "Name" : "Product 1",
      "Description": "Product 1 Description",
      "Quantity" : 1,
      "Price" : 9.99
    },
    {
      "id" : 2,
      "Name" : "Product 2",
      "Description": "Product 2 Description",
      "Quantity" : 5,
      "Price" : 2.99
    },
    {
      "id" : 3,
      "Name" : "Product 3",
      "Description": "Product 3 Description",
      "Quantity" : 1000,
```

```

        "Price" : 5.49
    }
  });
});

```

Checking cell contents

Now we can use the built-in `matchGridCellContent` method available to us via the `t` parameter. This method accepts a grid instance (or Component Query to find one), the cell row and column, the value expected, and a description of the test.

We can add a test to check whether the first column (containing the product's name) has the correct value, based on the data we added into our store:

```

// test Row 0, Cell 0
t.matchGridCellContent('product-ProductGrid', 0, 0, 'Product 1', 'Cell
0, 0 contents are correct');
// test Row 1, Cell 0
t.matchGridCellContent('product-ProductGrid', 1, 0, 'Product 2', 'Cell
1, 0 contents are correct');

```

Simulating clicks

Next, we will test the product editing process following a click on one of the grid row's **Details** button. We want to test after this interaction, whether:

- A new product form is created and shown
- The form is populated with the correct record details
- The grid is updated in real time after editing the Name field
- The form is hidden and the record committed after clicking on the **Save** button

We start by creating a new test file (`020_ProductEdit.t.js`) and including the same initial store population code as we did in the last section.

Now, we need to get a reference to the cell that holds the `Details` button. We do this using the `getCell` method, passing it a Component Query to find the grid and the row/cell indexes:

```

var cell = t.getCell('product-ProductGrid', 0, 4);

```

For this test, we are going to perform multiple actions in sequence which may be asynchronous. We could use callback methods to chain these items, but instead, we will use the `chain` method which makes this process much easier.

First, we will instigate a click on the cell and then confirm that a `ProductForm` component has been created. We do this using the `cqExists` (Component Query exists) method:

```
...
t.chain( function (next) {
  // click the button
  t.click(cell);
  // check ProductForm is created
  t.cqExists('product-ProductForm', 'Product Form is displayed');
  next();
}
...
);
```

Each function we pass to the `chain` method is passed a `next` method that should be called at the end, so that the next item in the chain is then executed.

We now want to check if the form has been populated with the correct values from the chosen record. First, we add a small pause of 10 milliseconds in the chain to allow the form time to populate, and then we call the `fieldHasValue` method for each field. We pass it in a Component Query to find the desired field, the value we expect it to have, and a description of the test:

```
...
{
  waitFor: 'Ms',
  args: 10
},
function (next) {
  t.fieldHasValue('product-ProductForm textfield[fieldLabel="Name"]',
    'Product 1', 'Name field has correct value');

  t.fieldHasValue('product-ProductForm textfield[fieldLabel="Description"]',
    'Product 1 Description', 'Description field has correct value');

  t.fieldHasValue('product-ProductForm textfield[fieldLabel="Quantity"]',
    '1', 'Quantity field has correct value');

  t.fieldHasValue('product-ProductForm textfield[fieldLabel="Price"]',
    '9.99', 'Price field has correct value');

  next();
}
...
```

The next step we want to test is what happens when one of the fields is updated. To do this, we add another step to our chain and update the `Name` field. We then check whether the grid has updated accordingly and that the model instance is now marked as dirty (that is, it has a change waiting to be committed):

```
function(next){
  var nameField = Ext.ComponentQuery.query('product-ProductForm
  textField[fieldLabel="Name"]')[0];

  nameField.setValue('Updated Product 1');

  t.matchGridCellContent('product-ProductGrid', 0, 0, 'Updated Product
  1', 'Cell 0, 0 contents are correct');

  t.expect(Ext.getStore('Products').getAt(0).dirty).toEqual(true);

  next();
}
```

Finally, we test the save process. We simulate a click on the `Save` button and test that the product form is closed and the changes to the record are committed. Finally, we call the `done` method to tell the harness that our tests are complete:

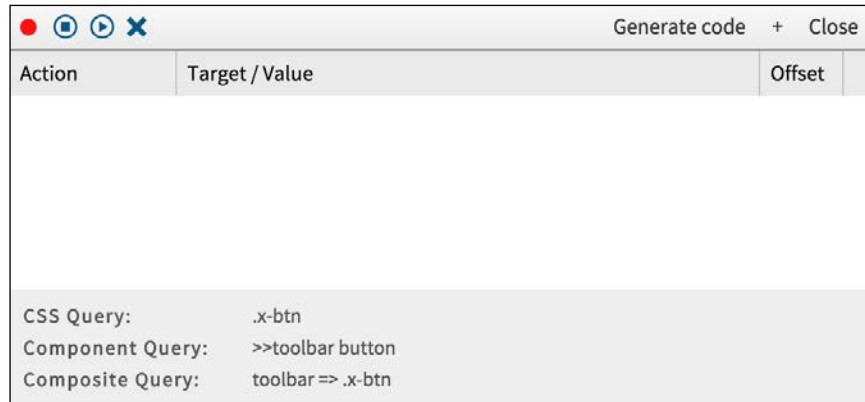
```
function(){
  t.click('>>button[text="Save"]');
  t.cqNotExists('product-ProductForm');
  t.expect(Ext.getStore('Products').getAt(0).dirty).toEqual(false);
  t.done();
}
```

The `>>` prefixed should be used to denote a Component Query string. If omitted, the query will be interpreted as a CSS style selector.

Event recorder

In the preceding example, we have performed very simple steps in our tests. However, hand coding all of the Component Queries for each button and the order of the interactions can get cumbersome. Siesta Standard features an event recorder that allows us to record a series of interactions and generate the code required to include these steps in a test. It is worth noting that this code is rarely used "as-is"; instead, it is usually modified and enhanced before being added to a test script.

To launch the event recorder, click the video camera icon in the test results window, which will show an empty list of steps, as seen in the following screenshot:

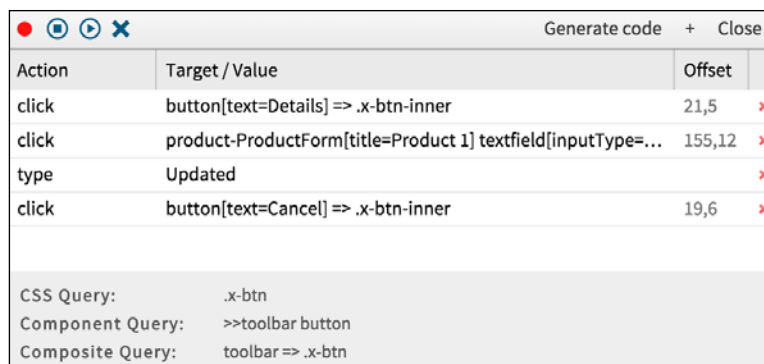


We will use the event recorder to test the cancel process of the product form. We start by running the `ProductEdit` test so that we have an interface to interact with and record against. We then click the record button (the small red circle) and start interacting with the interface in the DOM panel.

We will perform the following actions:

1. Click on the **Details** button.
2. Edit the **Name** field in the product form.
3. Click on the **Cancel** button.

Once we do these, we will click on the stop button and the steps will populate the grid.



You will see each action shown as a row with the action type, the target of the action (in this case the Component Query used to target the element or the text typed), and the offset of the action.

If you would like to alter the target used, you can click on the cell and choose a different query from the given list, or modify it manually.

Once you are happy with the steps, you can generate the code to include in the test spec. Simply click on the **Generate Code** button and the code will be displayed, which can be copied and pasted into our test file.

The output of the recorder generally uses the more condensed, declarative syntax for chain steps. Both are perfectly valid and result in the same outcome.

In most cases, this is enough to get us started, and we can add more steps to the chain which will check for the right outcome after each action.

Taking our example, we may construct the following test assertions around the recorded steps:

```
t.chain( { click : "button[text=Details] => .x-btn-button", offset :
[10, 13] },
function(next) {
  t.cqExists('product-ProductForm', 'Product Form is displayed');
  next();
},
{
  click : "product-ProductForm[title=Product 1]
textfield[inputType=text] => .x-form-text", offset : [127, 10] },
{
  action : "type", text : " Updated"
},
function(next) {
  t.matchGridCellContent('product-ProductGrid', 0, 0, 'Product 1
Updated', 'Cell 0, 0 contents are correct');
  t.expect(Ext.getStore('Products').getAt(0).dirty).toEqual(true);
  next();
},
{
  click : "button[text=Cancel] => .x-btn-inner", offset : [16, 1] },
function() {
  t.cqNotExists('product-ProductForm');
  t.expect(Ext.getStore('Products').getAt(0).dirty).toEqual(false);
  t.done();
}
);
```


Test automation and integration

So far, we have only run our Siesta unit tests in a browser using the Siesta interface, which is great for development. However, we will generally want to run these tests as part of an automated build process.

We can very easily run our test suites from a command line using PhantomJS or WebDriver. We will now demonstrate how to use WebDriver to run cross-browser tests from the command line.

WebDriver is a tool that allows cross-browser test automation and comes bundled with Siesta, making it extremely easy to get it working. In their simplest form, we can run the tests we have just created, which must be available from a server or localhost, in Google Chrome using the following code:

```
siesta/bin/webdriver  
http://localhost/tests/view/product/_ProductGrid/index.html --  
browser=chrome
```

By executing this code from our root project directory, an instance of Google Chrome will be launched, the tests run, and the outcome is printed in the Terminal/Command Prompt window.

```
Launching test suite, OS: MacOS, browser: Chrome 39.0.2171.99  
[PASS] 010_GridCellContents.t.js  
[PASS] 020_ProductEdit.t.js  
[PASS] 030_ProductEdit_Cancel.t.js  
17 passed, 0 failed assertions took 9.386s to complete
```

You can extend this to include FireFox, Safari, Chrome, and Internet Explorer by using the `--browser=*` option. This will open each browser in turn and execute the tests—perfect for cross-browser testing.

Test reports

It is also possible to generate a test report after each run that you can include in your build assets, for example. To do this, we include the `--report-format=json` and `--report-file=BuildReport.json` options at the end of our command, to save a JSON version of our test results.

Summary

This chapter has focused on how we can write better, more testable JavaScript code, and then moved on to how we can use testing frameworks to execute repeatable tests on this code.

We dove into the Siesta testing framework and demonstrated how to write simple unit tests as well as more complex UI tests that test our applications as if users were interacting with them.

Finally, we saw how our test suites can be run to test cross-browser effectiveness as part of a build process from the command line.

Index

A

AJAX proxies 75, 76

application

- anatomy 10
- creating, with Sencha Cmd 6
- generating 8
- implementing 9
- themes, applying to 139-141

application components

- controllers, generating 41
- generating 40
- models, generating 40
- views, generating 40, 41

application, offline

- advantages 50
- data, syncing 51, 52
- designing 50
- offline architecture 51
- options 51
- rating 50
- working 49

B

bar chart

- data, displaying in 156

border layout

- configuring 97
- layout of components, creating 99
- using 96
- Viewport 97

business logic

- about 48
- controllers 49
- ViewControllers 48, 49

C

cell contents

- checking 177
- expected data, setting up 176
- testing 175, 176

chained stores 66, 67

chart components

- anatomy 151
- axes 152
- interactions 152
- labels 152
- series 151, 152

classes

- configuration value, setting 25
- configuring 24
- defaults, overriding 27
- defining 13-15
- extending 21
- overriding 22
- platform-specific configs 27, 28

class members

- adding 19
- methods 19
- properties 19
- singletons 21
- statics 20
- statics, in subclasses 20

clicks

- simulating 177, 178

column displays

- column renderers 125
- customizing 125

component layouts

- absolute 94
- accordion 94

- anchor 94
- border 94
- card 94
- center 94
- column 94
- configuring 94
- fit 94
- HBox 95
- tables 95
- VBox 95
- component lifecycle**
 - about 111
 - creation lifecycle 111
 - destruction process 113
- Component Queries**
 - about 114
 - sample component structure 115
 - with Ext.ComponentQuery 118
 - xtypes 115
- components**
 - about 110, 111
 - finding, based on attributes 118
 - finding, based on itemIds 119
 - finding, based on member functions 119
 - finding, based on xtype 118
- component UIs**
 - about 147
 - applying 148
 - defining 147, 148
 - different component style, defining 149
- configuration options**
 - callback 74
 - headers 74
 - method 74
 - params 74
 - timeout 74
- content**
 - ResponsiveConfig rules 107
 - used, for navigation 106
- controller 44, 45, 49**
- creation lifecycle**
 - about 111
 - activate (*) 112
 - boxready 112
 - config options 112
 - constructor 112
 - initComponent 112
 - render 112
 - show (*) 112
- cross-class communication**
 - with events 49
- CRUD endpoints 89**
- custom theme**
 - anatomy 143
 - architecture 141, 142
 - creating 141
 - cross-browser styling 144
 - theme package, generating 142, 143

D

- data**
 - CRUD endpoints 89
 - defining 128
 - displaying 129, 130
 - displaying, in bar chart 156
 - inputting, with forms 128
 - obtaining, in application 72
 - populating 131-133
 - saving 87-89
 - updates, persisting 133
 - writers 89, 91
- data associations**
 - about 79
 - many-to-many relationships 85
 - one-to-many relationships 80
- data-bound views**
 - about 134
 - data view, defining 135
 - styling 136
- data view**
 - defining 135
 - item selector 136
 - store 135
 - template 135, 136
- deployment recommendations, application**
 - about 9
 - concatenation 10
 - GZip 10
 - image optimization 10
 - minification 10

destruction process

- about 113
- deactivate (*) 114
- destroy 114
- hide (*) 114

E

event handlers

- button handlers 31
- defining, in config objects 30
- on method 32, 33

event recorder 179-181

events

- event delegation 35
- firing 34
- listening, on elements 35

Ext.Ajax

- AJAX calls 72
- configuration options 74
- errors, handling 73

Ext JS

- about 2, 53
- applications, using 30
- defining 5
- development environment 5
- directory structure 45
- events 30
- Ext.Ajax 72
- features 53
- framework 5
- limitations 3
- naming convention 45
- pie chart, creating 158-160
- proxies, defining 74
- REST proxies 79

Ext JS 5, features

- about 4
- architectural improvements 4
- component enhancements 4
- responsive layouts 4
- touch support 4

Ext.Loader

- about 15
- and Sencha Cmd 18
- class definition process 16
- dependencies, defining 16, 17

- dependency root 18
- loader paths 17

F

fit layout

- using 99

folders, application 10

frameworks

- downloading 7
- Jasmine 169
- Siesta 169, 170
- testing 169

G

grids

- visualizations, integrating 160-162

grid widgets 127

H

hasMany configuration option 82

HBox layout

- pack config option 103
- using 100-102
- widths 102

hierarchical data, with trees

- about 120
- binding, to data source 120
- tree panel, defining 120, 121

HTML 110, 111

I

Integrated Development Environments (IDEs) 5

J

Jasmine

- about 169
- URL 169

K

keyboard events

- about 36
- KeyMap 36

L

layouts

working 94, 95

line chart

creating 152, 153
model, creating 153
server, polling for data 154, 155
store, creating 153

listener options 33

LocalStorage proxies 77, 78

M

many-to-many relationships

about 85
associated data, loading 87
association, defining 86
data source, configuring 85
proxy, configuring 85

models

custom data converters 58
custom field types 57
defining 54
fields 55
field validation 55, 56

mouse events 36

MVC

about 1, 39, 43
benefits 45, 46
Controller 44
drawbacks 45, 46
Model 43
View 43

MVVM

about 1, 39, 43, 46
managing 46
Model 46
View 47
ViewModel 47, 48

O

one-to-many relationships

about 80
association, defining 81
data source, configuring 80

proxy, configuring 80
reference config 83
requests, exploring 84

override class 23

overrides

targeting, to framework versions 24

P

packages 7

pack config option 103

pie chart

creating, in Ext JS 158-160

private 166

proxies

AJAX proxies 75, 76
defining 74
LocalStorage proxies 77, 78

R

Reader class 74

reference config 83

ResponsiveConfig rules 107

responsive layouts

about 104
Ext.mixin.Responsive 105
Ext.plugin.Responsive 105

REST proxies 79

S

sample component structure 115

scoped Component Queries

about 119
down 119
query 120
up 119

Sencha Cmd

about 40
and Ext.Loader 18
application components, generating 40
application, creating with 6
application metadata, refreshing 42
application, upgrading 42
automatic builds, triggering 42
installing 6, 7

Sencha website

URL 7

Siesta

about 169, 170

URL 169

skeleton application

bootstrapped launch process 11

browser API interaction 12

data, managing 12

event system 12

JavaScript to HTML 11

routing system 12

sparklines 161

stores

chained stores 66, 67

complex searches 61

configuration-based filtering 62

configuration-based grouping 65

configuration-based sorting 64

defining 59

filtering 62

grouping 64

record, retrieving 60

sorting 63

specific records, finding 61

stats, storing 60

TreeStores 68

working with 59

T

tabular data

column displays, customizing 125

displaying 122

grid widgets 127

product data 122

product grid 123, 124

template columns 126

testable JavaScript

accessible code 166

event handlers, separating from actions 168

nested callback functions 167

single responsibility 166

writing 166

test automation

about 182

and integration 182

test reports

generating 182

theme customizations

about 144

button's color, changing 146

font size, changing 146

main color, changing 145

theme variables 145

theme inheritance

about 144

new theme, applying 144

themes

applying, to application 139-141

configuring 141

touch events

about 37

event normalization 37

gestures 37

TreeStores

about 68

appendChild 70

creating 70

eachChild 70

Ext.data.TreeModels 68-70

findChild 70

insertChild 70

isLeaf 70

populating 70, 71

removeChild 70

U

UI interaction

cell contents, testing 175, 176

clicks, simulating 177, 178

event recorder 179-181

testing 175

UI widget

anatomy 109, 110

components 110, 111

HTML 110, 111

unit tests

executing 174

extending 174, 175

first test, adding 173

- project structure, testing 171
- test harness, creating 171, 172
- writing 170

updateLayout method 95

use cases 3

V

valid values 102

VBox layout

- align config option 104
- pack config option 104
- using 103

ViewControllers 48, 49

Viewport 97

visualizations

- integrating, in grids 160-162

W

watch command 42

widths, HBox layout

- fixed width 102
- flex width 102

workspace

- about 7
- generating 7, 8

Writer class 74

X

xtypes

- about 115
- component 115



About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

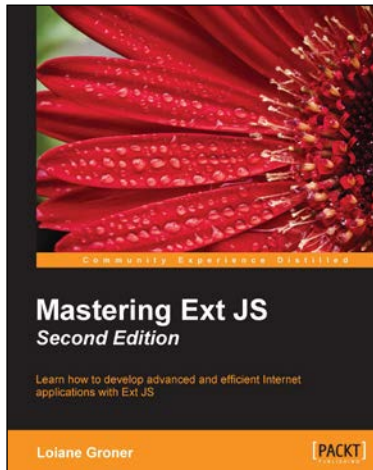
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



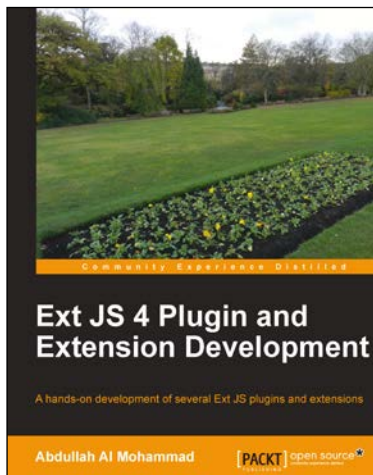
Mastering Ext JS Second Edition

ISBN: 978-1-78439-045-7

Paperback: 400 pages

Learn how to develop advanced and efficient Internet applications with Ext JS

1. Build a complete application with Ext JS from scratch to the production build.
2. Excellent tips and tricks to make your web applications stand out.
3. Written in an engaging and easy-to-follow conversational style, with practical examples covering the server side as well as MySQL.



Ext JS 4 Plugin and Extension Development

ISBN: 978-1-78216-372-5

Paperback: 116 pages

A hands-on development of several Ext JS plugins and extensions

1. Easy-to-follow examples on ExtJS plugins and extensions.
2. Step-by-step instructions on developing ExtJS plugins and extensions.
3. Provides a walkthrough of several useful ExtJS libraries and communities.

Please check www.PacktPub.com for information on our titles



JavaScript Mobile Application Development

ISBN: 978-1-78355-417-1

Paperback: 332 pages

Create neat cross-platform mobile apps using Apache Cordova and jQuery Mobile

1. Configure your Android, iOS, and Windows Phone 8 development environments.
2. Extend the power of Apache Cordova by creating your own Apache Cordova cross-platform mobile plugins.
3. Enhance the quality and the robustness of your Apache Cordova mobile application by unit testing its logic using Jasmine.



Mastering JavaScript Design Patterns

ISBN: 978-1-78398-798-6

Paperback: 290 pages

Discover how to use JavaScript design patterns to create powerful applications with reliable and maintainable code

1. Learn how to use tried and true software design methodologies to enhance your Javascript code.
2. Discover robust JavaScript implementations of classic as well as advanced design patterns.
3. Packed with easy-to-follow examples that can be used to create reusable code and extensible designs.

Please check www.PacktPub.com for information on our titles