

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=525>



**MEAP Edition  
Manning Early Access Program**

Copyright 2009 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

© Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Licensed to jeremy hulick <jeremy.hulick@gmail.com>

**Part I Introduction to Ext JS**

- 1. A framework apart
- 2. Back to the basics
- 3. Events, Components, and Containers

**Part II Ext components**

- 4. Panels, TabPanels, and windows
- 5. Organizing Components
- 6. Ext takes Form

**Part III DataStore components**

- 7. The venerable GridPanel
- 8. The EditorGridPanel
- 9. DataView
- 10. Charts and Graphs
- 11. Taking root with Trees
- 12. Toolbars and Menus

**Part IV Advanced Ext**

- 13. Drag and drop basics
- 14. Drag and drop with widgets
- 15. Plug-ins & extensions

**Part V Building applications**

- 16. Application development basics
- 17. Constructing an application

# 1

## *A framework apart*

Envision a scenario where we are tasked to develop an application with many of the typical UI widgets such as menus, tabs, data grids, dynamic forms and styled pop-up windows. We want something that allows us to programmatically control the position of widgets, which means it has to have layout controls. We also desire detailed and organized centralized documentation to ease our learning curve with the framework. Lastly, this application needs to look mature and go into beta phase as quickly as possible, which means we don't have lots of time to toy with HTML and CSS. Before entering the first line of code for the prototype, you need to decide on an approach to developing the front-end. What are your choices?

We do some recon on the common popular libraries on the market and quickly learn that all of them can manipulate the DOM but only two of them have a mature UI library, YUI and Ext JS.

At a first glance of YUI, we might get the sense that we need not look any further. We toy with the examples and notice that they look mature but not exactly professional quality, which means we need to modify CSS. No way. Next, we look at the documentation at <http://developer.yahoo.com/yui/docs>. It's centralized and extremely technically accurate, but it is far from user friendly. Look at all of the scrolling required to locate a method or class. There are even classes that are cut off because the left navigation pane is too small. What about Ext JS? Surely it has to better -- right?

### **1.1 An easier way to get the job done**

Out of the proverbial box, you're provided a rich set of widgets. These coupled with the Layout management tools give you full control to organize and manipulate the UI as requirements dictate. Most widgets are highly customizable, affording you the option to enable, disable features, override, and use custom extensions and plug-ins. One example of

a web application that takes full advantage of Ext JS is Conjoon. Below is a screenshot of Conjoon in action.

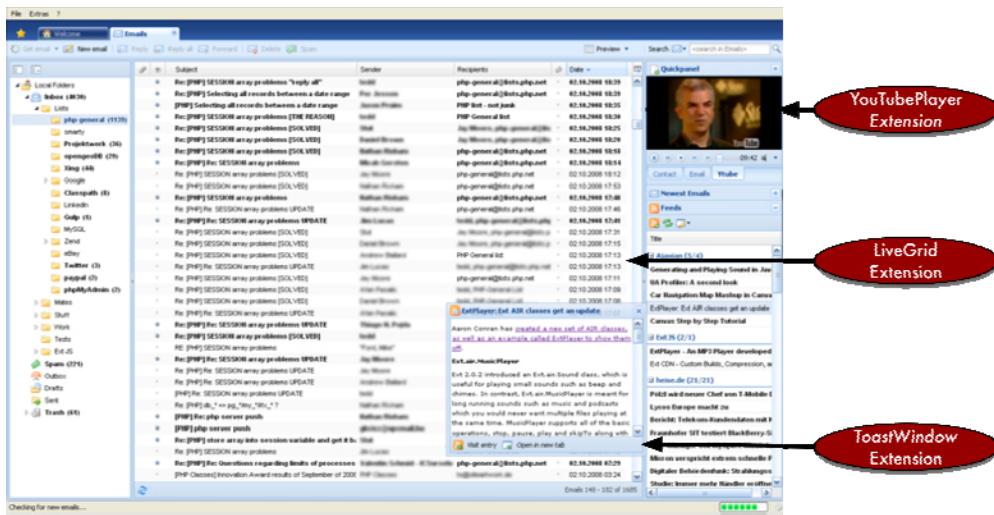


Figure 1.1 Conjoon is an open source personal information manager that is a great example of a web application that leverages the framework to manage a UI that leverages 100% of the browser's viewport. You can download it at <http://conjoon.org/>.

Conjoon is an open source Personal Information Manager and can be considered the epitome of web applications developed with Ext JS. It uses just about all of the framework's native UI widgets and demonstrates how well the framework can integrate with custom extensions such as the YouTubePlayer, LiveGrid and ToastWindow. You can get a copy of Conjoon by visiting <http://conjoon.org>. What if you're looking to just add some Ext flavor to your existing web application or site?

### 1.1.1 Integration with existing sites

Any combination of widgets can be embedded inside an existing web page and or site with relative ease, giving your users the best of both worlds. An example of a public facing site that contains Ext JS is located at the Dow Jones Indexes site, which you can visit via <http://djindexes.com>.

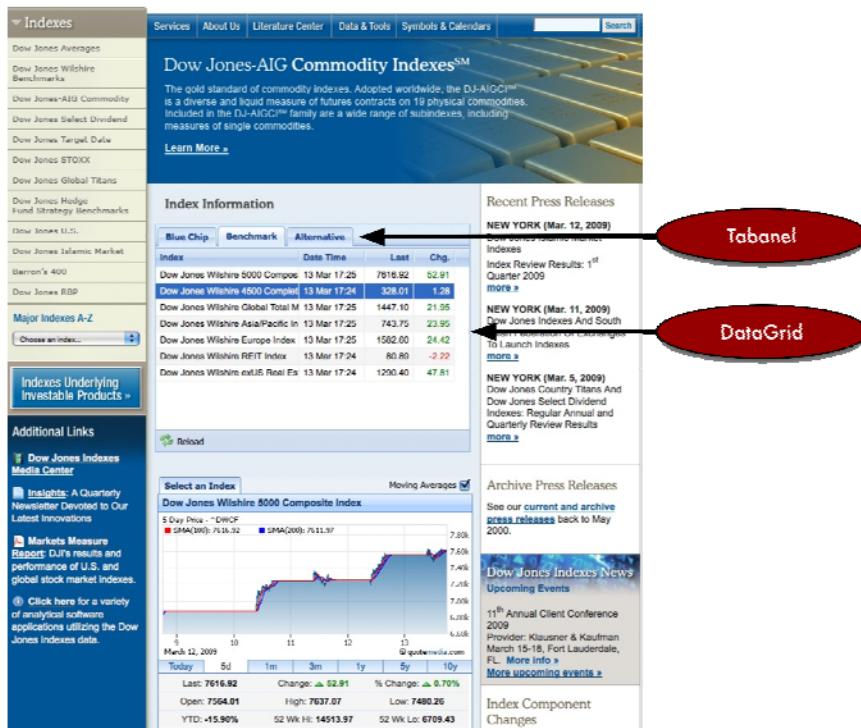


Figure 1.2 The Dow Jones indexes web site, <http://djindexes.com>, is one example of Ext JS embedded in a traditional Web 1.0 site.

This Dow Jones indexes web page gives its visitors rich interactive views of data by utilizing a few of the Ext widgets such as the TabPanel, DataGrid, and Window (not shown). Their visitors can easily customize the view of the stocks by selecting a row in the main data grid, which invokes an AJAX request to the server, resulting in an updated graph below the grid. The graph view can be modified as well by clicking on one of the time period buttons below it.

We now know that Ext JS can be leveraged both to build entire applications or can be integrated into existing ones. However, we still have not satisfied the requirement of API documentation. How does Ext solve this?

### 1.1.2 Rich API documentation

When opening the API documentation for the first time (Figure 1.3), you get a sense that this unlike any other to date. Unlike competing frameworks, the Ext API Documentation leverages its own framework to present a clean and easy to use documentation tool that uses AJAX to present us with the documentation.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

We'll explore all of the features of the API and talk about some of the components used in this documentation tool.

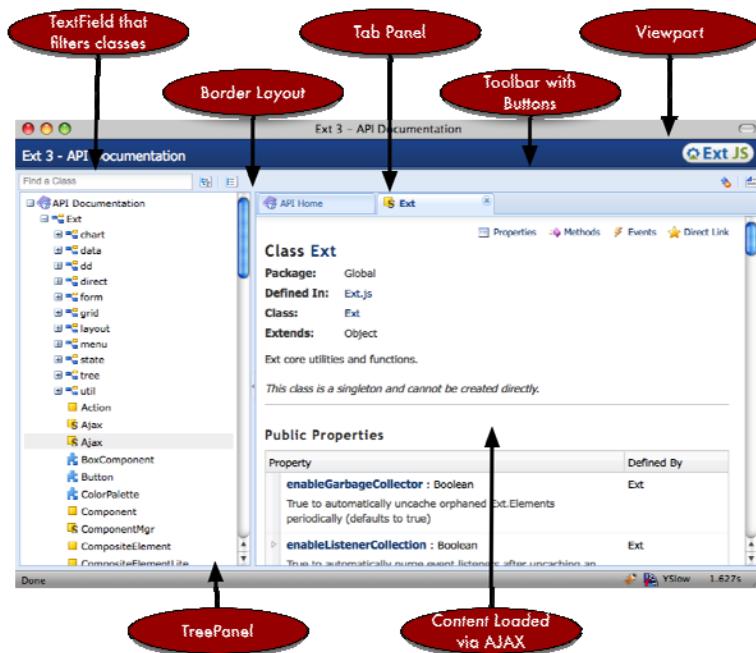


Figure 1.3 The Ext API documentation contains a wealth of information and is a great resource to learn more about components and widgets. It contains most of what you need to know about the API including constructor configuration options, methods, events, properties, component hierarchy and more.

The API documentation tool uses six of the most commonly used widgets, which include the Viewport, Tree and Tab Panels, Toolbar with an embedded TextField and Toolbar buttons and the Center, which contains the TabPanel. Before we move on, I'm sure you're wondering what all of these are and do. Let's take a moment to discuss them.

Looking from the outside in, the Viewport is a class that leverages all of the browser viewing space to provide an Ext managed canvas for UI widgets. It is known as the foundation from which an entire Ext JS application is built upon. The "BorderLayout" layout manager is used, which divides the Viewport (or any Container) up into three regions; North (top) for the page title, link and Toolbar, West (left) which contains the TreePanel and the Center region, which contains the TabPanel to display documentation.

The Toolbar is a class that provides a means to present commonly used UI components such as Buttons and menus, but can also contain, as in this case, form fields. I like to think

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

of the Toolbar as the common File>Edit\View menus that you see in popular Operating Systems and desktop applications. The TreePanel is a widget that displays hierarchical data visually in the form of a tree much like Windows Explorer displays your hard drive's folders. The Tab Panel provides a means to have multiple documents or components on the canvas but only allows for one to be active.

Using the API is a cinch. To view a document, click the class node on the tree. This will invoke an AJAX request to fetch the documentation for the desired class. Each document for the classes is an HTML fragments (not a full HTML page). With the TextField in the Toolbar, you can easily filter out the class tree with just a few strokes of the keyboard. If you're connected to the Internet, API documentation itself can be searched in the "API Home" tab.

So the documentation is great, but what about rapid application development? Can Ext accelerate our development cycles?

### **1.1.3 Rapid development with prebuilt widgets**

With Ext JS can help you jump from conception to prototype because it offers many of the UI elements that you would require already built and ready for integration. Having these UI widgets already prebuilt means that much time is saved from having to engineer them. In many cases, the UI controls are highly customizable and can be modified or customized to your application needs.

### **1.1.4 It works with Prototype, jQuery, YUI and inside of Air**

Even though we have discussed how Ext JS stands out from Prototype, jQuery, and YUI, Ext JS can easily be configured to leverage these frameworks as a base, which means if you're using these other libraries, you don't have to give them up to enjoy Ext JS UI goodness.

Even though we won't cover development of Ext JS applications in Air, it is worth noting that the framework has a complete set of utility classes to help with the integration of Air. These utility classes include items like Sound management, a video player panel access to the desktop clipboard. Even though we won't go into Air in this book, we will cover many of the very important parts of the framework from which you'll need to know when developing with Ext in Air.

Before we talk any more about Ext JS, we should set the playing field and discuss what types of skills are necessary to utilize the framework.

## **1.2 What you need to know**

While being an expert in web application development is not required to develop with Ext JS, there are some core competencies that developers should have before attempting to write code with the framework.

The first of these skills is a basic understanding of Hyper Text Markup Language (HTML) and Cascading Style Sheets (CSS). It is important to have some experience with these technologies because Ext JS, like any other JavaScript UI library, uses HTML and CSS to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

build its UI controls and widgets. While its widgets may look like and mimic typical modern Operating System controls, it all boils down to HTML and CSS in the browser.

Because JavaScript is the ‘glue’ that ties AJAX together, a solid foundation in JavaScript programming is suggested. Again, you need not be an expert, but you should have a solid grasp on key concepts such as arrays, reference and scope. It is a plus if you are very well familiar with Object Oriented JavaScript fundamentals such as objects, classes, and prototypal inheritance. If you are extremely new to JavaScript you’re in luck. JavaScript has existed since nearly the dawn of the Internet. An excellent place to start is W3Schools.com, which offers a lot of free online tutorials and even has sandboxes for you to play with JavaScript online. You can visit them here:

<http://w3schools.com/JS/>

If you are required to develop code for the server side, you’re going to need a server side solution for Ext JS to interact with as well as a way to store data. Most developers choose databases, but some choose direct file system storage as well as well. Naturally, the range of solutions available is quite large. For this book, we won’t focus on a specific language. Instead, we’re going to use online resources at <http://extjsinaction.com>, where I’ve done the server side work for us, this way all you have to focus on is learning the Ext JS.

We’ll begin our exploration of Ext JS with a bird’s eye view of the framework, where we’ll learn about the categories of functionality.

### 1.3 A bird’s eye view of the framework

Ext JS is a framework that not only provides UI widgets, but also contains a host of other features. These can be divided up into six major areas of purpose; Core, UI Components, Data Services, Drag and Drop and general utilities. Figure 1.4 illustrates the six areas of purpose.

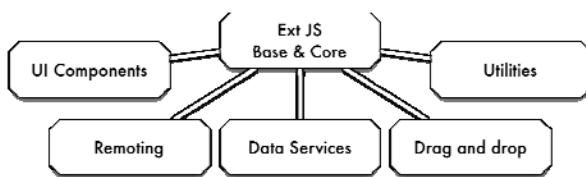


Figure 1.4 The six areas of purpose for Ext Classes; Core, UI, Remoting, Data Services, Drag and Drop and Utilities.

The first feature set is the Ext JS Base and Core, which comprises of many of the basic features such as AJAX Communication, DOM Manipulation and event management. Everything else is dependent on the Core of the framework, but the Core is not dependant on

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

anything else. The UI Components contains all of the Widgets, gadgets and gizmos that interface with the user.

Web Remoting is a means for JavaScript to easily remotely execute method calls that are defined and exposed on the server, which is commonly known as a Remote Procedure Call or RPC. It is extremely convenient for development environments where you would like to expose your server side methods to the client and not worry about all of the fuss of AJAX method management.

The Data Services section takes care of all of your data needs, which includes fetching, parsing and loading information into Stores. With the Ext Data Services classes, you can read Array, XML and JSON (JavaScript Object Notation), which is a data format that is quickly becoming the standard for client to server communication. Stores typically feed UI components.

Drag and Drop is like a mini framework inside of Ext, where you can apply Drag and Drop capabilities to an Ext Component or any HTML element on the page. It includes all of the necessary members to manage the entire gamut of all Drag and Drop operations. Drag and Drop is a particularly complex topic which we will discuss in chapters 13 and 14.

The Utilities section is composed of cool utility classes that help you get some of your routine tasks easier. An example would be `util.Format`, which allows you to format or transform data really easily. Another neat utility is the CSS singleton, which allows you to create, update, swap and remove style sheets as well as request the browser to update its rule cache.

Now that we have a general understanding of the framework's major areas of functionality, let's take some time to look at some of the more commonly used UI widgets that Ext has to offer.

### **1.3.1 Containers and Layouts at a glance**

Even though we will cover these topics in greater detail later in this book, I think we should spend a little bit of time talking about Containers and Layouts. The term Container and Layout are used extensively throughout this book and I want to make sure you have at least a basic understanding of them before we continue. Afterwards, we'll begin our view into visual components of the UI library.

Containers are widgets that can manage one or more child items. A child item is generally any widget or component that is managed by a Container or parent, thus the parent-child paradigm. We've already seen this in action in the API. The `TabPanel` is a Container that manages one or more child items, which are presented as tabs. Please remember this term, as we will be using it a lot when we start to learn more about how to use the UI portion of the framework.

Layouts work with Containers to visually organize the child items on a canvas, which is commonly known as a Container's content body. Ext JS has a total of 12 layouts from which to choose, which we will go into great detail in chapter 5 and learn the ins and outs of each

layout. Now that we have a high level understanding Containers and Layouts, let's start to look at some containers in action.

In the following illustration, we see two descendants of Container, Panel and Window each engaged in parent-child relationships.

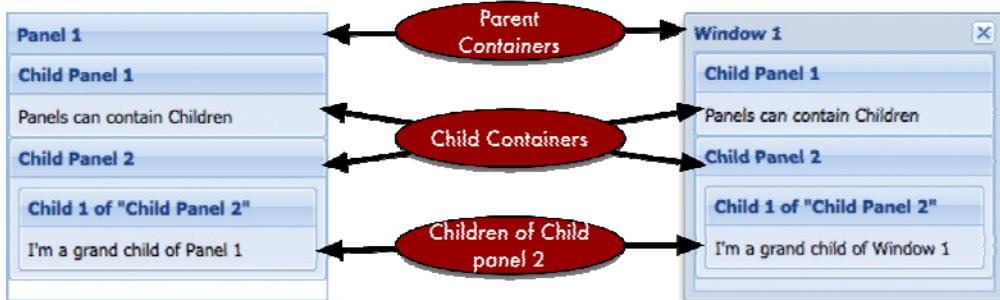


Figure 1.5 Here, we see two parent Containers, Panel (left) and Window(right), managing child items, which includes nested children.

In Figure 1.5, we see a Panel (left) and a Window (right) managing two child items each. "Child Panel 1" of each of the parent containers simply *contain* HTML, while the children with the title "Child Panel 2" manage one child panel each using the no-frills ContainerLayout, which is the base for all other layouts. This parent-child relationship is the crux of all of the UI management of Ext JS and will be reinforced and referenced repeatedly throughout this book.

We just learned that Containers manage child items and use layouts to visually organize them. Being that we now have these important concepts down, we will move on to see and discuss other Containers in action.

### 1.3.2 More Containers in Action

We just saw Panel and Window being used, when we learned about Containers. Here are some other commonly used descendants of Container.

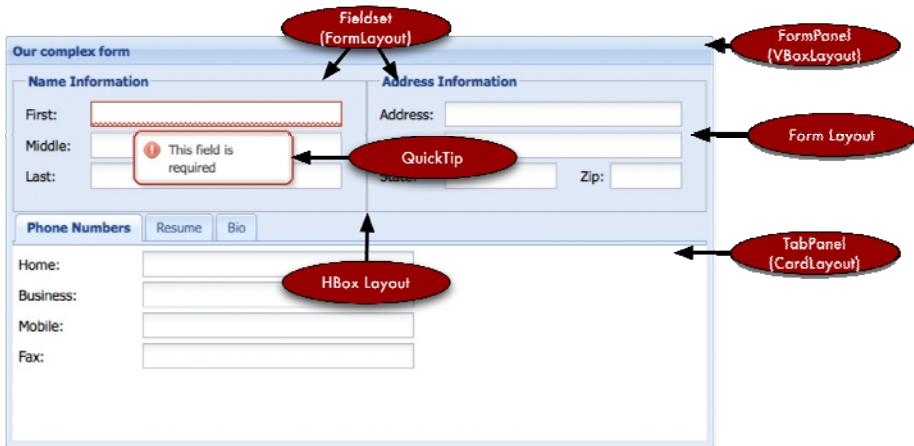


Figure 1.6 Commonly used descendants of Container, the FormPanel, TabPanel, FieldSet and QuickTip and the layouts used to compose this UI Panel. We will build this in Chapter 6, where we learn about forms.

In Figure 1.6, we see The FormPanel, TabPanel, FieldSet and QuickTip widgets. The FormPanel essentially works with the BasicForm class to wrap fields and other child items with a form element.

Looking at this from a parent-child perspective, the FormPanel is being used to manage three child items, two instances of FieldSet and one instance of TabPanel. Fieldsets are generally used to display fields in a form much like a typical fieldset tag does in general HTML. These two fieldsets are managing child items, which are text fields. The TabPanel here is managing three direct children (tabs), which its first tab ("Phone Numbers") is managing many children, which are text fields. Lastly, the QuickTip, which is used to display helpful text when the mouse is hovering over an element, is being displayed, but is not a child of any Ext Component.

We will actually spend some time building this complex UI in Chapter 6, where we learn more about form panels. For now, let's move on to learn about what data presentation widgets the framework has to offer.

### 1.3.3 Grids, DataView and ListView

We've already learned that the Data Services portion of the framework is responsible for the loading and parsing of data. The main consumers of the data for which the data Store manages, are the GridPanel, DataView and its descendant, the ListView. Below is a screenshot of the Ext GridPanel in action.



Figure 1.7 The Data Grid as seen in the “Buffered Grid” Example in the Ext SDK.

The `GridPanel` is a descendant of `Panel`, and presents data in a table-like format, but its functionality extends far beyond that of a traditional table, offering sortable, resizable and movable column headers and row or cell selection models. It can be customized to look and feel as you desire and can be coupled with a `PagingToolbar` to allow large data sets to be shown in pages. It also has descendants like the `EditorGridPanel`, which allow you to create a grid where users can edit data on the grid itself, leveraging any of the Ext form data input widgets.

The Grid is great for displaying data, but is somewhat computationally expensive. This is because of the many DOM elements that each row contains. Ext offers some lighter-weight ways to display data from a store, which include the `DataView` and its descendant, the `ListView` as seen below.

The screenshot displays two examples side-by-side. On the left is a `DataView` component showing a grid of thumbnail images with file names below them. On the right is a `ListView` component showing a table with columns for "File", "Last Modified", and "Size", listing various image files and their details.

File	Last Modified	Size
dance_fever.jpg	03-17 12:10 pm	2 KB
gangster_zack.jpg	03-17 12:10 pm	2.1 KB
kids_hug.jpg	03-17 12:10 pm	2.4 KB
kids_hug2.jpg	03-17 12:10 pm	2.4 KB
sara_pink.jpg	03-17 12:10 pm	2.1 KB
sara_pumpkin.jpg	03-17 12:10 pm	2.5 KB
sara_smile.jpg	03-17 12:10 pm	2.4 KB
up_to_something.jpg	03-17 12:10 pm	2.1 KB
zack.jpg	03-17 12:10 pm	2.8 KB
zack_dress.jpg	03-17 12:10 pm	2.6 KB
zack_hat.jpg	03-17 12:10 pm	2.3 KB
zack_sink.jpg	03-17 12:10 pm	2.2 KB
zacks_grill.jpg	03-17 12:10 pm	2.8 KB

Figure 1.8 The `DataView` (left) and `ListView` (right) as seen in the Ext SDK Examples.

The `DataView` is a class that consumes data from a Store and “paints” it on screen using what are called Templates and provides a simple selection model. An Ext Template is a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

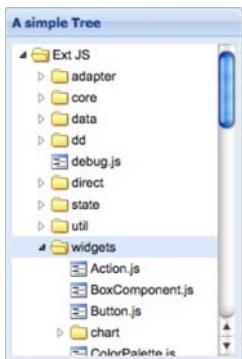
DOM utility that allows you to create a *template*, with placeholders for data elements, which can be filled in by individual records in a store and stamped out on the DOM. In Figure 1.8, the DataView (left) is displaying data from a store, which includes references to images. It uses a pre-defined template, which contains image tags, where the individual records are leveraged to fill in the location of the images. The Template then stamps out an image tag for each individual record, resulting in a nice collage of photos. The DataView can be used to display anything in a data store.

The ListView, as pictured in Figure 1.8 (right), is displaying data from a store in a grid-like fashion, but is a subclass of DataView. It is a great way to display data in a tabular format without the weight of the GridPanel. That is, if you're not looking to use some of the GridPanel features like sortable and resizable columns.

Grids and DataViews are essential tools for painting data on screen, but they do have one major limitation. They can only show lists of records. That is, they cannot display hierarchical data. This is where the TreePanel comes to the rescue.

### 1.3.4 Make like a TreePanel and leaf

The TreePanel widget is an exception to the list of UI widgets that consume data, where it does not consume data from a data Store. Instead, it consumes hierarchical data via the usage of the `data.Tree` class. Below is an example of an Ext TreePanel widget.



1.9 An Ext JS Tree, which is an example from the Ext SDK.

In Figure 1.9, the TreePanel is being used to display the parent-child data inside of the directory of an installation of the framework. The TreePanel can leverage what is known as a TreeLoader to fetch data remotely via AJAX or can be configured to use data stored on the browser. It can also be configured to use Drag and Drop and has its own selection model.

We've already seen `TextFields` in a form when we discussed Containers a short while ago. Next, we'll look at some of the other input fields that the framework has to offer.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

### 1.3.5 Form Input fields

Ext has a palette of eight input fields from which to use. They range from simple `TextField`s, as we've seen before, to complex fields such as the `ComboBox` and the `HTML Editor`. Below is an illustration of the available Ext form field widgets that come out of the box.

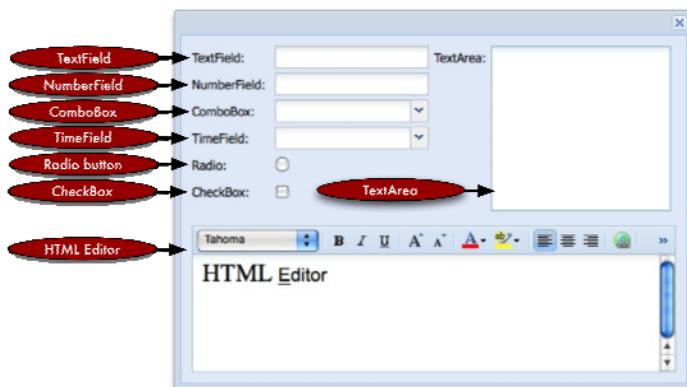


Figure 1.10 All of the out of the box form elements displayed in an encapsulating Window.

As we can see in Figure 1.10, some of the form input fields look like stylized versions of their native HTML counterparts. The similarities end here however. With the Ext Form Fields, there are much more than meets the eye!

Each of the Ext fields (minus the `HTML Editor`) includes a suite of utilities to perform actions like get and set values, mark the field as invalid, reset and perform validations against the field. You can apply custom validation to the field via regex or custom validation methods, allowing you to have complete control over the data being entered into the form. The fields can validate data as it's being entered, providing live feedback to the user.

The `TextField` and `TextArea` can be considered extensions of their generic HTML counterparts. The `NumberField`, however, is a descendant of the `TextField` and is a convenience class, which utilizes regular expressions to ensure users can only enter numbers. With the `NumberField`, you can configure decimal precision as well as specify the range of the value entered. The `ComboBox` and `TimeField` require a little extra time relative to the other fields, so we're going to skip these two for now and jump back to them in a bit.

Like the `TextField`, the `Radio` and `Checkbox` input fields are extensions of the plain old `Radio` and `Checkbox`, but include all of the Ext Element management goodness have convenience classes to assist with the creation of `Checkbox` and `Radio` groups with automatic layout management. Below is an example of `Checkbox` and `Radio` groups.



Figure 1.11 An example of the Checkbox and Radio Group convenience classes in action with automatic layouts.

Figure 1.11 is just a small sample of how the Ext Checkbox and RadioGroups can be configured with complex layouts. To see the entire set of examples, you can visit <http://extjs.com/deploy/dev/examples/form/check-radio.html>.

The HTML Editor is a WYSIWIG (What You See Is What You Get) is like the TextArea on steroids. The HTML editor leverages existing browser HTML editing capabilities and can be considered somewhat of a black sheep when it comes to fields. Because of its inherent complexities (using IFrames, etc), it does not have a lot of the abilities like validation and cannot be marked as invalid. There is much more to discuss about this field, which we're going to save for Chapter 6. But for now, let's circle back to the ComboBox and its descendant the TimeField.

The ComboBox is easily the most complex and configurable form input field. It can mimic traditional option dropdown boxes or can be configured to use remote data sets via the data Store. It can be configured to auto-complete text, known as "type-ahead" in Ext, that is entered by the user and perform remote or local filtering of data. It can also be configured to use your own instance of an Ext Template to display a custom list in the dropdown area, known as the ListView. The following figure is an example of a custom ComboBox in action.



Figure 1.12 A Custom ComboBox, which includes an integrated Paging Toolbar, as seen in the downloadable Ext Examples.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

In Figure 1.12, a custom combo box is being leveraged to search the Ext forums. The ComboBox here shows information like the post title, date, author and a snippet of the post in the list box. Because some of the data set ranges are so large, it is configured to use a paging toolbar, allowing users to page through the resulting data. Because the ComboBox is so configurable, we could also include image references to the resulting data set, which can be applied to the resulting rendered data.

Here we are, on the last stop of our UI Tour. Here, we're going to take a peek at some of the other UI components that work anywhere.

### 1.3.6 Other widgets

Here, we find a bunch of UI controls that kind of stand out, where they are not real major components, but play supporting roles in a grander scheme of a UI. Let's look at the illustration below and discuss what these items are and do.

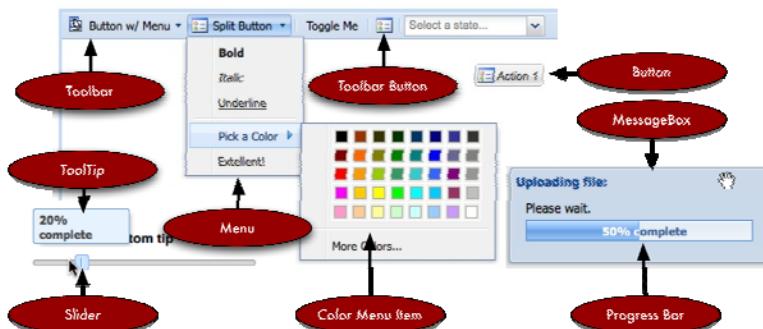


Figure 1.13 A collection of miscellaneous UI widgets and controls.

The Toolbar is a widget, which allows you to place just about any widget that fits in it. Generally developers will place menus and buttons. In discussing the custom ComboBox, We've seen the Toolbar's descendant, the PagingToolbar. Panels and any just about descendant thereof can use these toolbars on the top or bottom of their content body. The button widget is essentially a stylized version of the generic HTML button, which can include icons as well as text.

Menus can be displayed by a click of a button on the toolbar or shown on demand at any X and Y coordinate on screen. While they typically contain menu items, such as the items shown and the Color Menu Item, they can contain widgets such as ComboBoxes.

The MessageBox is a utility class, which allows us to easily provide feedback to the user without having to craft up an instance of Ext.Window. In this illustration, it's using an animated ProgressBar to display the status of an upload to the user.

Lastly, the Slider is a widget that leverages drag and drop to allow users to change a value by repositioning the knob. They can be styled with images and CSS to create your

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

own custom look. The movement of the knob can be restricted so it only moves in increments. In this example, the slider has a ToolTip above the knob, displaying the value of the slider as the user moves it. In addition to the default horizontal orientation, Sliders can be configured to be vertical.

We've recently learned how Ext can help us get the job done through a large palette of widgets. We learned that we could elect to use Ext to build an application without touching an ounce of HTML or we can integrate it with existing sites. We also performed a top-down view of the framework, which included a UI tour. All of the stuff discussed thus far was already in existence for Ext 2.0. Let's take a moment to discuss what's new in Ext 3.0.

## 1.4 New Ext 3.0 Goodies

Ext 2.0 introduced some radical changes, which made upgrading from 1.0 was rather difficult. This was mainly because of the introduction to the more modern layout manager and new robust component hierarchy, which broke most of the user developed Ext 1.x code. Thankfully, due to the great engineering of Ext 2.0, the migration from Ext 2.0 to 3.0 is much easier. While the additions to Ext 3.0 are not as drastic, there is excitement about this latest release and is certainly worthwhile discussing some of the additions.

### 1.4.1 Ext does Remoting with Direct

Web Remoting is a means for JavaScript to easily execute method calls that are defined on the server side. It is extremely convenient for development environments where you would like to expose your server side methods to the client and not have to worry about all of the muck with AJAX connection handling. Ext.Direct takes care of this for us by managing AJAX requests and acts as a bridge between the client side JavaScript and any server side language.

This functionality has great advantages to it, which include method management in a single location as well as unification of methods. Having this technology inside of the framework will ensure consistency across the consumers such as the data classes. Speaking of which, see how the addition of Ext.Direct brings new classes to the data classes.

### 1.4.2 Data Class

The Ext.data class is the nerve center for all data handling in the framework. The data classes manage every aspect of data management including fetching, reading and parsing data to create Records, which are loaded into a Store. With the addition of Direct, Ext has added additional convenience Data classes, DirectProxy and DirectStore, to facilitate ease of integration with your web remoting needs.

Now that we've reviewed some of the behind-the-scenes changes and additions to the framework, take a gander at some of the UI widgets that have been added.

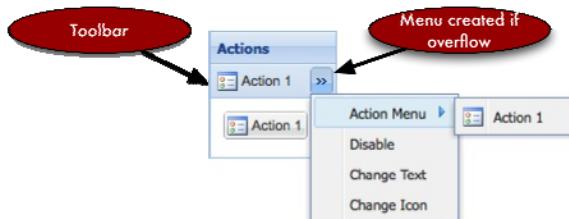
### 1.4.3 Meet the new Layouts

A total of five new layouts make their way into the Ext 3.0 framework, which include Menu, Toolbar, VBox and HBox. The MenuLayout is an abstraction of the way menu

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

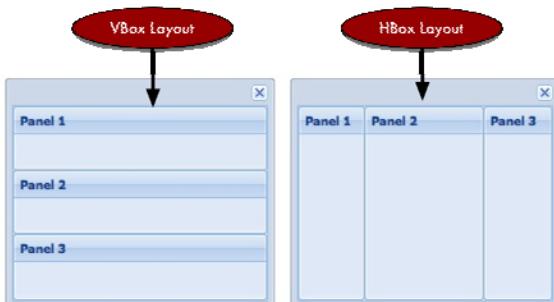
items were organized with in the past but now provide a cleaner way to manage menu items and is not to be used directly. Likewise, the ToolbarLayout adds important features to the Toolbars like overflow management as illustrated below.



1.14 The new ToolbarLayout is responsible for detecting the toolbar's size and creating menu stubs if menu items were to overflow.

As we see in Figure 1.14, the toolbar layout will detect the overflow of toolbar items in a toolbar and will automatically create a menu, which lists and contains the rest of your items. The changes to the MenuLayout help support this effort. Just like the Menu layout, the ToolbarLayout is not to be used directly by the end developer.

The last two layouts, the VBox and HBox, are great additions to the end-developer usable layouts. The HBox layout allows you to divide up a Container's content body into horizontal slices, while the VBox layout does the same, except vertically as illustrated below.



1.15 An example of the VBox and HBox layouts in action.

Many experienced ext developers would think that the HBox looks like the column layout in practice. While it does provide similar function it extends way beyond the capabilities, where it will vertically and horizontally stretch children based on weights, which is known as `flex`. These two layouts usher in a whole new era of layout capabilities within the framework.

In addition to the layout changes, the `ColumnModel`, a supporting class for the `GridPanel` has undergone some fundamental changes. Let's learn about some of these changes and why they are going to help us out in our development efforts.

#### **1.4.4 Grid ColumnModel Enhancements**

The Grid `ColumnModel` is a class that models how the columns are organized, sized and displayed for the `GridPanel` widget. Prior to Ext 3.0, individual columns were generally configured a list of configuration objects in an array, which is consumed by `ColumnModel`. For each column in the `ColumnModel`, you could enhance or modify the way the data is displayed by creating a custom renderer, which is a method that is called for each data point for that column and returns the desired formatted data or HTML. This means that if you wanted, lets say, a date to be formatted or displayed a certain way, you had to configure, which many people found themselves doing a lot. In this release, the `ColumnModel` changed somewhat to make our jobs that much easier.

The individual `Column` has been abstracted from the `ColumnModel` and an entirely new class was created called the `grid.Column`. From here, many convenient `Column` subclasses have been created, which includes `NumberColumn`, `BooleanColumn`, `TemplateColumn` and `DateColumn` each of which allow you to display your data, as you desire. To display formatted Dates, you could use the `DateColumn` and simply specify a format from which the dates are to be displayed. The `Template Column` is another welcomed change as it allows you to leverage `XTemplates` and display them in a `GridPanel`, which are convenience methods to create and stamp out HTML fragments based on data. To use any of these `Column` subclasses, no custom renderers are required, though you can if you wish.

A lot of applications require data to be displayed in tabular format. While the `GridPanel` is a great solution, it is computationally expensive for generic data displays that require little or no user interaction. This is where `ListView`, an extension of one of `DataView`, comes to the rescue.

#### **1.4.5 ListView, like GridPanel on a diet**

With this new addition to the framework, you can now display more data in a grid-like format without sacrificing performance. Below is an illustration of the `ListView` in action. While it looks similar to the `GridPanel`, in order to achieve better performance, we sacrifice features such as these include drag and drop column reordering as well as keyboard navigation. This is mainly because the `ListView` does not have any of the elaborate feature-rich supporting classes, such as the `ColumnModel` we discussed just a moment ago.

Simple ListView (1 item selected)		
File	Last Modified	Size
dance_fever.jpg	03-17 12:10 pm	2 KB
gangster_zack.jpg	03-17 12:10 pm	2.1 KB
kids_hug.jpg	03-17 12:10 pm	2.4 KB
kids_hug2.jpg	03-17 12:10 pm	2.4 KB
sara_pink.jpg	03-17 12:10 pm	2.1 KB
sara_pumpkin.jpg	03-17 12:10 pm	2.5 KB
sara_smile.jpg	03-17 12:10 pm	2.4 KB
up_to_something.jpg	03-17 12:10 pm	2.1 KB
zack.jpg	03-17 12:10 pm	2.8 KB
zack_dress.jpg	03-17 12:10 pm	2.6 KB

Figure 1.16 The new Ext.ListView class, which is like a very light weight DataGrid.

Using the ListView to display your data will ensure that you have faster response from DOM manipulation, but remember that it does not have all of the features of the GridPanel. Choosing which one to use will depend on your application requirements.

Ext has always been excellent at displaying textual data on screen, but lacked a graphical means of data representation. Let's take a quick look at how this has changed with Ext 3.0.

#### 1.4.6 Charts come to Ext

One of things that was missing in the 2.0 version of Ext JS was charts. Thankfully the development team listened to the community and has introduced them for version 3.0. These are a great addition, which adhere to the Ext Layout models.



Figure 1.17 These charts now bring rich graphical views of trend data to the framework. It is important to note that this new widget requires Flash.

Use of these charts, however, requires Adobe Flash to be installed for the browser you're using, which can be downloaded at <http://get.adobe.com/flashplayer/>. In addition to the Line and Column charts shown above, the framework has Bar, Pie and Cartesian charts available for your data visualization needs.

We now have all just about all of the things we need to lay down some code. Before we start our path to becoming Ext JS ninjas, we must first download and setup the framework and have a discussion about development.

## 1.5 Downloading and Configuring

Even though downloading Ext is a simple process, configuring a page to include Ext JS is not as simple as referencing a single file in the HTML. In addition to configuration, we'll learn about the folder hierarchy and what they are and do.

The first thing we need to do is get the darn source code. Visit the following link:

<http://extjs.com/products/extjs/download.php>

The downloaded file will be an `ext-3.0.zip`, which weighs in over 6MB in size. We'll explain why this is so large in a bit. Now, extract the file in a place where you serve JavaScript. In order to leverage AJAX, you're going to need a web server. I typically use Apache, which is cross-platform, but IIS for Windows will do. Let's take a peek at what just got extracted.

### 1.5.1 Looking at the SDK contents

If you're like me, you probably checked the size of the files extracted from the downloaded SDK zip file. If your jaw dropped, feel free to pick it back up. Yes, over 30MB is rather large for a JavaScript framework. The reason it's that big will be revealed in just a bit. For now, look at what got extracted.

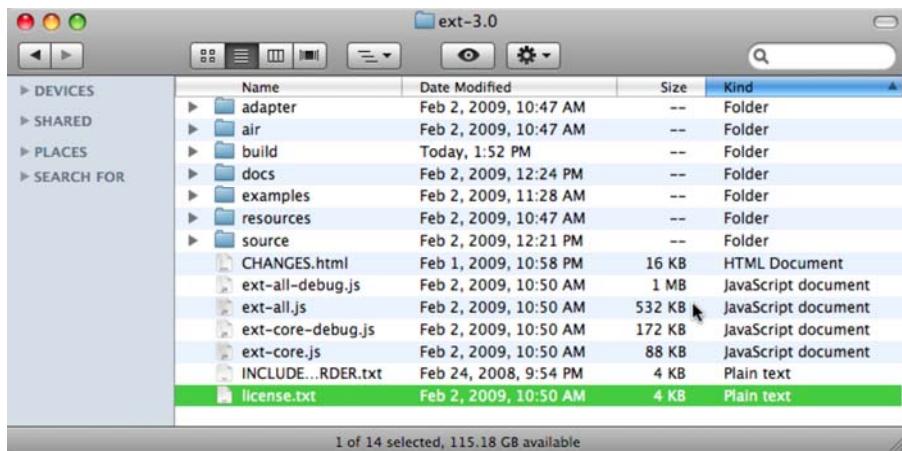


Figure 1.18 A view of the Ext JS SDK contents.

Looking at the contents of the SDK, we see a lot of stuff. The reason there are so many folders and files is because the downloadable package contains a few copies of the entire code base and CSS. It is this way because you get the freedom to build or use Ext JS anyway you see fit. Table 1.1 explains what each of the folders are and what they do.

**Table 1.1 Ext Js installation folders**

Name	Description
adapter	Contains the ext-base.js, which is the base ext library, which is used for an all-Ext JS setup. It also contains necessary adapters and supported versions of Prototype, jQuery or YUI libraries if you want to use any of those as a base.
air	Contains all of the required libraries to integrate Ext JS with Adobe Air.
build	Has each of the files that comprises of the Ext JS framework with all of the unnecessary whitespace removed. It's essentially a deflated version of the source directory.
docs	This is where the full API documentation is located.
examples	All of the example source code, which is great source to learn by example (no pun intended).
resources	Contains all of the necessary images and CSS required to use the UI widgets. It contains all of the CSS files broken down by widget and a ext-all.css, which is a concatenation of all of the framework's CSS.
source	Here is where the entire framework sits, which includes all of the comments.
ext-all.js	This is a minified version of the framework, which is intended to be used for production applications.
ext-core.js	If you just wanted to use the core library, which means none of the UI controls, ext-core.js is what you'd set your page up with.
*debug.js	Anything with 'debug' in the name means that the comments are stripped to reduce file space, but the necessary indentation is remained intact. When we develop, we're going to use ext-all-debug.js.

While there are quite a few files and folders in the distribution, you only need a few to get the framework running in your browser. Now is a good time to talk about how to set up Ext JS for use.

### 1.5.2 Setting up Ext for the first time

In order to get Ext JS running in your browser, you need to include at least two required JavaScript files and at least one CSS File.

```
<link rel="stylesheet" type="text/css"
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

    href="extjs/resources/css/ext-all.css" />
<script type="text/javascript" src="extjs/adapter/ext/ext-base.js">
</script>
<script type="text/javascript" src="extjs/ext-all-debug.js">
</script>

```

Above, we're linking the three core files for an all-Ext configuration. The first thing we do is link to the `ext-all.css` file, which is all of the CSS for the framework in one file. Next, we include `ext-base.js`, which is the underlying base of the framework. Lastly, we include `ext-all-debug.js`, which we'll use for development. When setting up your initial page, be sure to replace `extjs` in your path with wherever you plan to reference the framework on your development web server.

What if you wanted to use any of the other base frameworks? How do we include those?

### 1.5.3 Configuring Ext JS for use with others

In order for Ext JS to work with the previously mentioned frameworks, an *adapter* must be loaded after the external base framework. The adapter essentially maps `ext-base` methods to the external library of choice, which is absolutely crucial. The following patterns can be used to use any of the other three base frameworks in addition to Ext JS.

First up, prototype:

```

<link rel="stylesheet" type="text/css"
      href="extjs/resources/css/ext-all.css" />

<script type="text/javascript"
       src="extjs/adapter/prototype/prototype.js">
</script>

<script type="text/javascript"
       src="extjs/adapter/prototype/scriptaculous.js?load=effects.js">
</script>

<script type="text/javascript"
       src="extjs/adapter/prototype/ext-prototype-adapter.js">
</script>

<script type="text/javascript"
       src="extjs/ext-all-debug.js"></script>

```

As you can see, this is just like the generic Ext JS setup with two additional JS files. The `prototype` and `scriptaculous` libraries take the place of `ext-base` and the `ext-prototype-adapter.js` maps the external library methods to Ext. Note, that we are still loading `ext-all-debug.js`. We'll continue to do so for the other two cases.

Next is jQuery:

```

<link rel="stylesheet" type="text/css"
      href="extjs/resources/css/ext-all.css" />

<script type="text/javascript"
       src="extjs/adapter/jQuery/jQuery.js">

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
</script>

<script type="text/javascript"
       src="extjs/adapter/jQuery/ext-jquery-adapter.js">
</script>

<script type="text/javascript"
       src="extjs/ext-all-debug.js"></script>
```

As you can see, it's similar to the prototype setup. The YUI configuration will look similar, with the difference being that we're loading different base library and adapter files.

Lastly, YUI:

```
<link rel="stylesheet" type="text/css"
      href="extjs/resources/css/ext-all.css" />

<script type="text/javascript"
       src="extjs/adapter/yui/yui-utilities.js">
</script>

<script type="text/javascript"
       src="extjs/adapter/yui/ext-yui-adapter.js">
</script>

<script type="text/javascript"
       src="extjs/ext-all-debug.js"></script>
```

And there you have it, the recipes for an Ext-all setup and all of the other three supported base JS Libraries. Moving forward, we'll be using the Ext-all configuration, but you are free to use whichever base library you wish. Before we move on to coding, we need to talk about one final crucial step to setting up Ext, and that's configuring the reference for s.gif.

#### **1.5.4 Configuring BLANK\_IMAGE\_URL**

The configuration of the `Ext.BLANK_IMAGE_URL` is one of those steps that developers often overlook and may lead to issues with the way the UI renders for your application. The `BLANK_IMAGE_URL` property specifies a location for the 1x1 pixel clear `s.gif` graphic, which is an essential piece of the UI portion of the framework and is used to create items like icons. Out of the box, `BLANK_IMAGE_URL` points to `http://extjs.com/s.gif`. For most users, that is OK, but if you're in an area where `extjs.com` is not accessible, this will become an issue. This also becomes a problem if you are using SSL, where `s` is being requested via HTTP instead of HTTPS, which will invoke security warnings in browsers. In order to prevent these issues, simply set `Ext.BLANK_IMAGE_URL` to `s.gif` locally on your web server as such:

```
Ext.BLANK_IMAGE_URL = '/you.path.to.extjs/resources/images/default/s.gif';
```

It is recommended that you set this parameter immediately after the inclusion of the Ext JS files or immediately before your application code is parsed. I'll show you an example of where to place it when we take Ext JS for a test drive.

If you're like me, you're probably itching to start using Ext JS. Let's dive in.

## 1.6 Take it for a test drive

For this exercise, we're going to create an Ext JS Window and we'll use AJAX to request an HTML file for presentation in the content body of the Window. We'll start by creating the main HTML file from which we'll source all of our JavaScript files.

### Listing 1.1 Creating our helloWorld.html

```
<link rel="stylesheet" type="text/css"
      href="/extjs/resources/css/ext-all.css" />                                <!-- 1 -->
<script type="text/javascript"
       src="/extjs/adapter/ext/ext-base.js"></script>                            <!-- 2 -->
<script type="text/javascript"
       src="/extjs/ext-all-debug.js"></script>
<script type="text/javascript">
    Ext.BLANK_IMAGE_URL = '/extjs/resources/images/default/s.gif';
</script>
<script type="text/javascript" src='helloWorld.js'></script>                  <!-- 4 -->
1) Including ext-all.css
2) Ensure ext-base.js and ext-all-debug.js are loaded
3) A JavaScript block to configure BLANK_IMAGE_URL
4) The inclusion of our soon to be created helloWorld.js file.
```

In Listing 1.1, I've included the HTML markup for a typical Ext-only setup, which includes the concatenated CSS file, ext-all.css **{1}** and the two required JavaScript files, ext-base.js and ext-all-debug.js **{2}**. Next, we set create a JavaScript block **{3}**, where we set the ever-important Ext.BLANK\_IMAGE\_URL property. Lastly, we include our soon to be created helloWorld.js file **{4}**.

If you have not noticed it, we're using /extjs as the absolute path to our framework code. Be sure to change it if your path is different. Next, we'll create our helloWorld.js file, which will contain our main JavaScript code.

### Listing 1.2 Creating helloWorld.js

```
function buildWindow() {
    var win = new Ext.Window({
        id      : 'myWindow',
        title   : 'My first Ext JS Window',                                     // 1
    });
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        width      : 300,
        height     : 150,
        layout     : 'fit',
        autoLoad   : {
            url      : 'sayHi.html',
            scripts  : true
        }
    });
    win.show(); // 3
}

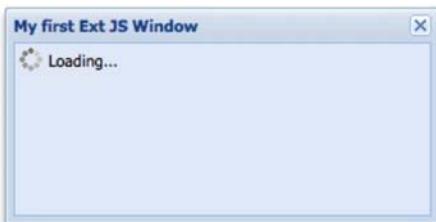
Ext.onReady(buildWindow); // 4

```

- 1) Instantiation of a new instance of Ext.Window.**
- 2) Specifying an autoLoad configuration object**
- 3) Calling upon our window to show().**
- 4) Passing buildWindow to Ext.onReady.**

In listing 1.2, we create the function `buildWindow`, which will be passed to `Ext.onReady` for later execution. Inside of this `buildWindow`, we create a new instance of `Ext.Window`, and setup a reference to it called `win` **{1}**. We pass a single configuration object to `Ext.Window`, which has all of the properties required to configure the instance we're instantiating.

In the configuration object, we specify an `id` of '`mywindow`', which will be used in the future to look up the window using the `Ext.getCmp` convenience method. We then specify a title of the window, which will appear as blue text on the top most portion of the window, which is known as the title bar. Next, we specify height and width of the window. We then go on to set the `layout` as '`fit`', which ensures that what ever is in the `ContentBody` of our `Window` is stretched to the dimensions of the `ContentBody`. We then move on to specify an `autoLoad` configuration object **{2}**, which will instruct the window to automatically fetch an HTML fragment (specified via the `url` property) and execute an JavaScript if found (specified via `scripts : true`). This ends the configuration object for our instance of `Ext.Window`. Next, we call on `win.show` **{3}**, which renders our window. This is where we find the conclusion of the `buildWindow` Method. The last thing we do is call `Ext.onReady` **{4}** and pass the our method, `buildWindow`, as a reference. This ensures that `buildWindow` is executed at just the right time, which is when the DOM is fully built and before any images are fetched. Let's see how our window renders. Go ahead and request `helloWorld.html` in your browser.



© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Figure 1.19 Our first Ext JS window attempting to load content via AJAX.

If you coded everything properly, you'd see a window that looks very similar to the one in Figure 1.19 with a spinning icon next to the 'Loading...' text, which is known as a loading indicator. Why do we see this? Well, this is because we have not created sayHi.html, which we referenced in the url property of the autoLoad configuration object. Essentially, we instructed Ext to load something that wasn't on the web server. Next, we'll construct sayHi.html, where we'll create an HTML fragment, which will include some JavaScript.

### **Listing 1.3 Creating sayHi.html**

```
<div>Hello from the <b>world</b> of AJAX!</div>           <!-- 1 -->  
<script type='text/javascript'>  
    function highlightWindow() {  
        var win      = Ext.getCmp('myWindow');  
        var winBody = win.body;  
        winBody.highlight();  
    }  
  
    highlightWindow.defer(1000);           /* 2 */  
</script>                                /* 3 */
```

- 1) The "Hello world" DIV tag.
- 2) A method to highlight the body of the window.
- 3) Delay the execution of highlightWindow by one second.

In Listing 1.3, we are creating an HTML fragment file, sayHi.html. It contains a div **{1}** from which we have our hello world message. After that, we have a script tag with some JavaScript, which will get executed after this fragment is loaded by the browser. In our code, we create a new function called highlightWindow **{2}**, which is going to be executed after a delay of one second. Inside that function, we perform a highlight effect on the content body of the window. The execution of highlightWindow is delayed by one second. Here is how this method works.

We first start by creating a reference to the Window we created in our `helloWorld.js` file by using a utility method called `Ext.getCmp`, which looks up an Ext Component by id. When we created our window, we assigned it an id of 'myWindow', which is what we're passing to `Ext.getCmp`. The reason this works is because all Components (widgets) get registered with the `ComponentMgr` upon instantiation. `Ext.getCmp` is a way to retrieve a reference by id from any context within your application.

After we get the reference of our Window, we create a reference, `winBody`, to its content body via the `body` property. We then call its `highlight` method, which will perform the highlight (fade from yellow to white) operation on the element. This is where we conclude the highlight Window method.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

The last thing we do in this JavaScript block is to call `highlightWindow.defer` and pass a value of 1000, which defers the execution of `highlightWindow` by one thousand milliseconds (or one second).

If you've never heard of `defer` in the JavaScript language, that's because we're using an Ext-introduced method. Ext leverages JavaScript's extensibility to add convenience methods to important core language classes, such as `Array`, `Date`, `Function`, `Number`, and `String`. This means every instance of any of those classes, have the new convenience methods. In this case, we're using `defer`, which is an extension of `Function`. If you're an old-timer, you're probably asking "why not just use `setTimeout`"? The first reason is because of ease of use. Call `.defer` on any method and pass the length of time to defer its operation. That's it. It also allows us to control the scope from which the deferred method is being executed and pass custom parameters, which `setTimeout` lacks.

Here we've ended our HTML fragment, which can now be fetched by our Window. Refresh `helloWorld.html` and you should see something like what we have below.

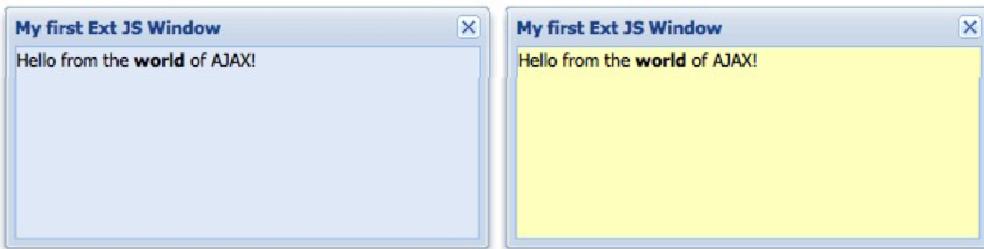


Figure 1.20 Our Ext JS Window loading the our HTML fragment (left) and the highlight effect performed on the Window's content body(right).

If you did everything correctly, your results should be exactly like the one in figure 1.20, where we see the content body being populated with the HTML fragment (left), and exactly one second later, the content body of the window highlights yellow (right). Pretty cool, huh? I suggest taking some time to modify the example and using the API to do things like changing the color of the highlight effect. Here's a hint, look under Ext -> Fx for the list of Effects and their parameters.

## 1.7 Summary

In this introduction to Ext JS, we learned how it can be used to build robust web applications or can be integrated into existing websites. We also learned how it measures up when stacked against other popular frameworks on the market and how it is the only UI based framework to contain UI-centric support classes such as the Component, Container and Layout models. Remember that Ext JS can "ride" on top of jQuery, prototype and YUI.

We explored many of the core UI widgets that the framework provides and learned that the number of prebuilt widgets helps rapid application development efforts. In doing that, we talked about some of the changes that Ext JS 3.0 has brought forth, such as Flash charts.

Lastly, we discussed where to download and how to setup the framework with each individual base framework. We created a “Hello world” example of how to use an Ext JS window to retrieve an HTML fragment via AJAX with just a few simple lines of JavaScript.

In the chapters to follow, we will explore how Ext works from the inside out. This knowledge will empower you to make the best decisions when building well-constructed UIs and better enable you to leverage the framework effectively. This will be a fun journey.

# 2

## *Back to the basics*

When working on applications, I often think metaphorically, which helps me develop parallels for concepts in my mind. I like to compare of the timing of an application's launch to that of the space shuttle's launch, where timing can mean the difference between a successful launch and inevitable frustration. Knowing when to initialize your JavaScript is one of the most critical things to know when dealing with anything that manipulates the DOM. We'll learn how to launch our JavaScript using the Ext to ensure our application code initializes at the right time on each browser. We can then begin our discussions on using Ext.Element to manipulate the DOM.

As we all know, DOM manipulation is one of the tasks that web developers are required to code for most of the time. Whether it's the addition or the removal of elements, I'm sure you've felt the pain of performing these tasks with the out of the box JavaScript methods. After all, DHTML has been at the very center of dynamic web pages for ages now.

We will look at the heart of Ext, known as the Ext.Element class, which is a robust cross-browser DOM element management suite. We'll learn to use Ext.Element to add and remove nodes from the DOM and see how it makes this task easier.

After we get familiar with the Ext.Element class, we'll learn how to use Templates to literally stamp out HTML fragments into the DOM. We'll also dive deep into the use of the XTemplate, which descend from Template and learn how to use it to easily loop through data and inject behavior modification logic while we're at it. This is going to be a fun chapter. Before we can begin coding, we must learn the proper way of launching our code.

### **2.1 Starting off the right way**

Since the early days, when most developers wanted to initialize their JavaScript, they typically would add an onLoad attribute to the body tag of the html page that is loading:

```
<body onLoad='initMyApp() ;'>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

While this method of invoking JavaScript works, it's not ideal for AJAX enabled Web 2.0 sites or applications because the `onLoad` code is generally fired at different times for different browsers. For instance, some browsers fire this when the DOM is ready and all content has been loaded and rendered by the browser. For modern Web 2.0, this is not Kosher as the code generally wants to start managing and manipulating DOM elements when the DOM is ready but before any images are loaded. This is where the right balance of timing and performance can be achieved. I like to call this the "sweet spot" in the page loading cycle.

Just like many things the world of browser development, each browser generally has its own way of knowing when its DOM nodes can be manipulated.

### **2.1.1 Fire only when ready!**

There are native browser solutions for detecting that the DOM is ready, but like a lot of things, they are not implemented uniformly across each browser. For instance, Firefox and Opera fire the `DOMContentLoaded` event. Internet Explorer requires a script tag to be placed in the document with a `defer` attribute, which fires when its DOM is ready. Safari fires no event, but sets the `document.readyState` property to 'complete', so a loop must be executed to check for that property and fire off a custom event to tell our code that the DOM is ready. Boy, what a mess!

### **2.1.2 Let Ext JS pull the trigger**

Luckily, we have `Ext.onReady`, which solves the timing issues and serves as the base from which to launch your application specific code. Ext JS achieves Cross Browser Compatibility (CBC) with by detecting which browser the code is executing on and manages the detection of the DOM ready state, executing your code at just the right time.

`Ext.onReady` is actually a reference to `Ext.EventManager.onDocumentReady` and accepts three parameters: the method to invoke, the scope from which to call the method and any options to pass to the method. The second parameter, `scope`, is used when you're calling an initialization method that is requires execution within a specific scope.

All of your Ext-based JavaScript code can be anywhere below (after) the inclusion of Ext JS script. This is very important because JavaScript files are requested and loaded synchronously. Trying to call any Ext methods before Ext is actually defined in the namespace will cause an exception and your code will fail to launch. Here is a simple example of using `Ext.onReady` to fire up an Ext Messagebox alert window.

```
Ext.onReady(function() {
    Ext.MessageBox.alert('Hello', 'The DOM is ready!');
});
```

In the preceding example, we pass, what is known as, an "anonymous" method to `Ext.onReady` as the only parameter, which will be executed when the DOM is ready to be manipulated. Our anonymous method contains a line of code to invoke an Ext Messagebox as seen in the figure below.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>



Figure 2.1 the result of our Ext.onReady call, an Ext.MessageBox window.

In short, an anonymous method is any method, which has no variable reference or label. Ext.onReady registers our anonymous method, which is to be executed when the internal docReadyEvent event is fired. In short, an event is like a message that something has occurred. A listener is a method that is registered to be executed or “called” when that event occurs or “fires”.

Ext fires this docReadyEvent event when it finds *exactly* the right time (remember the sweet spot) in the page loading cycle to execute our anonymous method and any other registered “listeners”. If the concept of events sounds a bit confusing, don’t be alarmed. Event management is a complex topic and we’ll cover it later, in chapter 3.

I cannot stress the importance of using Ext.onReady enough. All of our example code (and eventually your application) code *has to* be launched this way. Moving forward, if Ext.onReady is not explicitly detailed in the examples, please assume that you must launch the code with it and wrap the example code in the following manner:

```
Ext.onReady(function() {  
    // ... Some code here ...  
});
```

Now that we are comfortable with using Ext.onReady to launch our code, we should spend some time learning about the Ext.Element class, which is known as the heart of the framework. This is one of those essential topics that is used everywhere in the framework where DOM manipulation occurs.

## 2.2 The Ext.Element

All JavaScript based web applications revolve around a nucleus, which is the HTML Element. JavaScript’s access to the DOM nodes gives us the power and flexibility to perform any action against the DOM we wish. These could include adding, deleting, styling or changing the contents of any node in the document. The traditional method to reference a DOM node by ID is:

```
var myDiv = document.getElementById('someDivId');
```

The getElementById method works very well to allow you to perform some basic tasks such as changing the innerHTML, style or assign a CSS class. But, what if you wanted to do more with the node, such as manage its events, apply a style on mouse click or replace a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

single CSS class? You would have to manage all of your own code and constantly update to make sure your code is fully cross browser compatible. I honestly cannot think of another thing that I wouldn't want to spend my time on than this. Thankfully Ext takes care of this for us.

### 2.2.1 The heart of the framework

Let's turn our heads to the `Ext.Element` class, which is known to many in the Ext JS community, as the 'heart of Ext JS' as it plays a role in each and every UI widget in the framework and can be generally accessed by the `getEl()` method or the `e1` property.

The `Ext.Element` class is a full DOM element management suite, which includes a treasure chest of utilities, enabling the framework to literally work its "magic" on the DOM and provide the robust UI that we have come to enjoy. This toolset and all of its power is available to us, the end developers.

Due to its design, its capabilities are not relegated to simple management to DOM elements, but rather perform complex tasks such as manage dimensions, alignments and coordinates with relative ease. You can also easily update an element via AJAX, manage child nodes, animate, full event management and so much more.

### 2.2.2 Using `Ext.Element` for the first time

Use of the `Ext.Element` is extremely easy and makes some of the hardest tasks simple. In order to go on with exercising `Ext.Element`, we need to setup a base page. Setup a page where you include the Ext JavaScript and CSS, just like we discussed in chapter 1. Next, include the following CSS and HTML.

```
<style type="text/css">
    .myDiv {
        border   : 1px solid #AAAAAA;
        width    : 200px;
        height   : 35px;
        cursor   : pointer;
        padding  : 2px 2px 2px 2px;
        margin   : 2px 2px 2px 2px;
    }
</style>

<div id='div1' class='myDiv'> </div>
```

What we're doing here is setting the stage for our examples by ensuring our target div tags have specific dimensions and a border so we can clearly see it on the page. We include one div with the id of 'div1', which we'll use as a target. If you setup your page correctly, the stylized div should be clearly visible as illustrated blow.

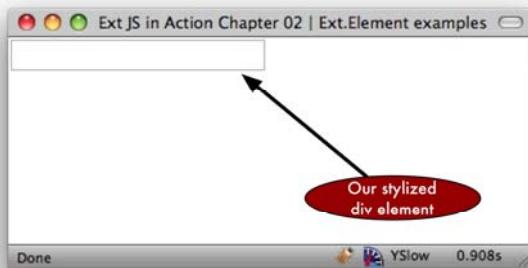


Figure 2.2 Our base page with our stylized div ready for some Ext element action.

In Figure 2.2, we see our generic HTML box, which we'll use to exercise the fundamental Ext.Element methods.

#### NOTE

All of our Ext.element example code will reference the same base page which we've just setup. If you're interested in watching changes to the DOM occur live, I suggest using the multiline Firebug console inside of Firefox with these examples. If you are unfamiliar with Firefox and FireBug, please see appendix XX for instructions. Conversely, these examples can be placed inside generic script blocks. Just be sure to remember to use Ext.onReady().

According to the CSS any div with the class "myDiv" is set to 35 pixels high and 200 pixels wide and looks a bit odd. Let's make that perfectly square by setting the height to 200 pixels.

```
var myDiv1 = Ext.get('div1');
myDiv1.setHeight(200);
```

The execution of the previous two lines is pretty important. The first line uses Ext.get, which we pass the string 'div1' and returns an instance of Ext.Element referenced by the variable myDiv1. Essentially, Ext.get uses document.getElementById and wraps it with the Ext element management methods.

We leverage our newly referenced instance of Ext.Element, myDiv1, and call its setHeight method, passing it an inter value of 200, which grows the box to 200 pixels tall. Conversely, you can use its setWidth method to change the width of the element, but we'll skip that and jump to something more fun.

"OK, it now looks like a perfectly square size. Whoop-dee-doo," you're saying? Well, let's change dimensions again, instead we'll use `setSize`. Let's make the width and height 350 pixels. We'll leverage the already created reference, `myDiv1`.

```
myDiv1.setSize(350, 350, {duration: 1, easing:'bounceOut'});
```

What happens when you execute the line of code above? Did it animate and have a bouncing effect? Cool - huh? I did this so you wouldn't fall asleep on me. OK, I'm just teasing.

Essentially, the `setSize` method is the composite `setHeight` and `setWidth`. For this method, we passed the target width, height and an object with two properties, duration and easing. The third property, if defined, will make `setSize` animate the size transition of the element. Don't care of animations, simply omit the third argument, the box will change its size within an instant, much like when we set the height.

Setting dimensions is just a single facet of the many sides of element management with the `Element` class. Some of the `Ext.Element`'s greatest power comes from its ease of use for full CRUD of elements, which is Create Update and Delete.

### 2.2.3 Creating Child Nodes

One of the great powers of JavaScript is to manipulate the DOM, which includes the creation of DOM nodes. There are many methods that JavaScript provides natively that provide you this power. ExtJS conveniently wraps a lot of these methods with the `Ext.Element` class. Let's have some fun creating child nodes.

To create a child node, we'll use `Element`'s `createChild` method:

```
var myDiv1 = Ext.get('div1');
myDiv1.createChild('Child from a string');
```

The code above adds a string node to the `innerHTML` of our target div. What if we wanted to create an element? Easy as pie.

```
myDiv1.createChild('<div>Element from a string</div>');
```

The usage of `createChild` above will append a child div with the string of "Element from a string" to the `innerHTML` of `div1`. Personally, I really don't like to append children this way because I find the string representation of elements to be messy. Ext helps me with this problem by accepting a configuration object instead of a string.

```
myDiv1.createChild({
  tag : 'div',
  html : 'Child from a config object'
});
```

Here, we're creating a child element by using a configuration object. We specify the tag property as 'div', and the 'html' as a string. This technically does the same thing as the prior `createChild` implementation but can be considered cleaner and self-documenting. What if we wanted to inject nested tags? With the configuration object approach, we can easily get this done.

```
myDiv1.createChild({
    tag      : 'div',
    id       : 'nestedDiv',
    style    : 'border: 1px dashed; padding: 5px;',
    children : [
        {
            tag : 'div',
            html : '...a nested div',
            style : 'color: #EE0000; border: 1px solid'
        }
    );
});
```

In the above code, we're creating one last child, which has an ID, a bit of styling applied and a child element, which is a div with some more styling. The following illustrates what the changes to the div look like.

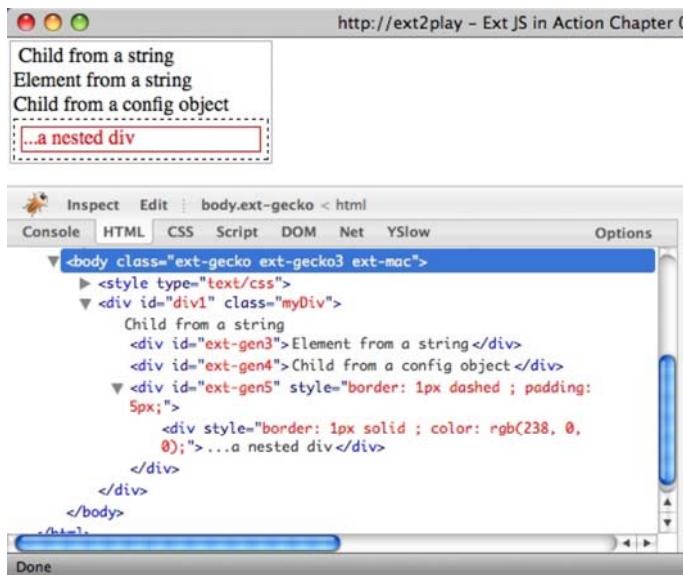


Figure 2.3 A composite of our element additions using `myDiv1.createChild()`.

In the preceding illustration, we see all of the additions to myDiv1 including the live DOM view from FireBug, showing us that we added a string node and three child divs, one of which has its own child div.

If you want to inject a child at the top of the list, you would use the convenience method insertFirst.

For instance:

```
myDiv1.insertFirst({
    tag : 'div',
    html : 'Child inserted as node 0 of myDiv1'
});
```

Element.insertFirst will always insert a new element at position zero, even no child elements exist in the DOM structure.

If you want to target the insertion of a child node at a specific index, the createChild method can take care of that task. All you need to do is pass it the reference of where to inject the newly created node.

For instance:

```
myDiv1.createChild({
    tag : 'div',
    id : 'removeMeLater',
    html : 'Child inserted as node 2 of myDiv1'
}, myDiv1.dom.childNodes[3]);
```

In the code above, we're passing two arguments to createChild. The first of which is the configuration object representation of the newly created DOM element and the second is the actual DOM reference of the target childNode from which createChild will use as the target to *inject* the newly created node. Please keep in mind the id that we've set for this newly created item, we're going to use this in a bit.

Notice how we're using myDiv1.dom.childNodes? Ext.Element gives us the opportunity to leverage all of the generic browser element management goodness by means of the dom property.

#### NOTE

The Element.dom property is the exact same reference as what is returned by document.getElementById().

The following illustration shows what our inserted nodes look like both in the page view and the DOM hierarchy using the firebug DOM inspection tool.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

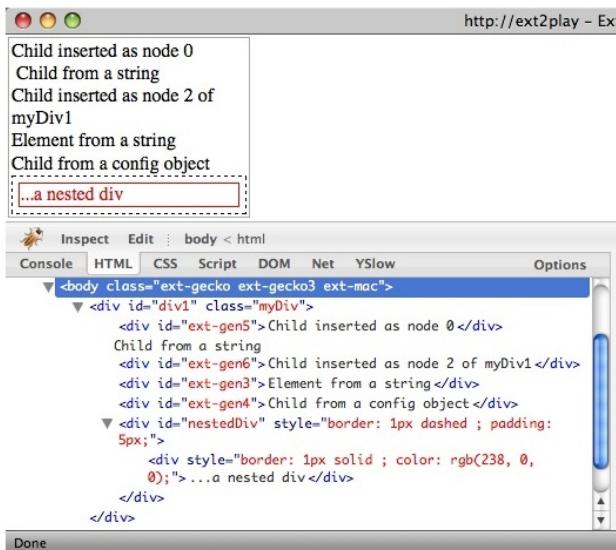


Figure 2.4 The results of our targeted DOM element insertions with `createChild()` using an index and usage of `insertFirst()`.

As you can see in the figure above, the node insertions functioned, as we wanted them to. We used `insertFirst` to inject a new node at the top of the list and `createChild` to inject a node *above* `childNode 3`. Remember to always count child nodes starting with the number 0 instead of 1.

Adding is something that we do often as web developers. After all, this is part of what DHTML is all about. But, on the flip side, removing is equally as important to know. Let's learn how we can remove some of the child elements using `Ext.Element`.

### 2.2.3 Removing Child Nodes

Removing nodes could be considered much easier than adding. All you need to do is locate the node with `Ext` and call its `remove` method. To exercise removal of child nodes, we are going to start with a clean and controlled slate. Please create a new page with the following HTML:

```
<div id='div1' class='myDiv'>
    <div id='child1'>Child 1</div>
    <div class='child2'>Child 2</div>
    <div class='child3'>Child 3</div>
    <div id='child4' class='sameClass'>
        <div id='nestedChild1' class='sameClass'>Nest Child 1</div>
    </div>
    <div>Child 5</div>
</div>
```

Examining the HTML above, we have a parent div with the ID of 'div1'. It has five direct descendants. The first of which has the id of 'child1'. The second and third children have no IDs, but have CSS classes of 'child2' and 'child3'. The fourth child element has an ID of 'child4' and a CSS class of 'sameClass'. Likewise, it has a direct child with an ID of "nestedChild1" and the exact same css class as its parent. The last child of div1 has no ID or CSS class. The reason we have all this stuff going on is because we're going to start using what is known as "CSS selectors" as well as targeting directly on the IDs of the elements.

In the examples where we added child nodes, we always referenced the parent div (id='div1') by wrapping it in an Ext.Element class and leveraging its create methods. To remove child nodes the approach is completely different as we need to target the node specifically that is to be removed. Using the new DOM structure above, we're going to exercise a few ways of doing this.

The first approach we'll examine is removing a child node for an already wrapped DOM element. That is, we'll create an instance of Ext.Element wrapping div1, then use it to find its first child node using what is known as a CSS selector.

```
var myDiv1 = Ext.get('div1');
var firstChild = myDiv1.down('div:first-child');
firstChild.remove();
```

In the preceding example, we create a reference to div1 using Ext.get. We then create another reference, firstChild, to the first child using the Element.down method and pass a pseudo-class selector, which basically causes ext to query the DOM tree within the context of div1 for the first child, which is a div and wrap it within an instance of Ext.Element.

The Element.down method essentially queries the first-level DOM nodes for any given Ext.Element. It so happens that the element that is found is the one with the div id of "child1". We then call firstChild.remove, which actually removes that node from the DOM.

Here is how you could remove the last child from the list using selectors.

```
var myDiv1 = Ext.get('div1');
var lastChild = myDiv1.down('div:last-child');
lastChild.remove();
```

The Example above works similarly to the one prior. The biggest difference being that we use the selector 'div:last-child', which locates the last childNode for div1 and wraps it in an instance of Ext.Element. After that, we call lastChild.remove and poof, it's gone.

#### NOTE

CSS selectors are an extremely powerful way of querying the DOM for items. Ext JS supports the CSS3 selector specification. If you're new to CSS selectors, I highly advise

visiting the following W3C page which has a plethora of information.  
<http://www.w3.org/TR/2005/WD-css3-selectors-20051215/#selectors>

OK, but what about if we want to simply target an element by an ID? Well, we can just use Ext.get to do our dirty work for us. This time, we'll create no reference and just use what is known as *chaining* to take care of the job.

```
Ext.get('child4').remove();
```

Executing the code above removes the child node with the id of 'child4' and its child node. Always remember that removing a node with children will remove its child nodes.

#### NOTE

If you would like to read more about chaining, Dustin Diaz, an industry leading developer, has an excellent article written in his site at <http://www.dustindiaz.com/javascript-chaining/>

The last thing we'll look at is using Ext.Element to perform an AJAX request to load remote HTML fragments from the server and inject it into the DOM.

#### 2.2.4 Using AJAX with Ext.Element

The Ext.Element class has the ability to perform an AJAX call to retrieve remote HTML fragments and inject those fragments into its innerHTML. To exercise this, we'll need to first write an HTML snippet to load:

```
<div>
    Hello there! This is an HTML fragment.
    <script type="text/javascript">
        Ext.getBody().highlight();
    </script>
</div>
```

In the above HTML fragment, we have a simple div with an embedded script tag, which performs an Ext.getBody call and uses chaining to execute the results of that call to execute its highlight method. Ext.getBody is a convenience method to get a reference to the document.body wrapped by Ext.Element. Save this file as htmlFragment.html.

Next, we'll perform the load of this snippet.

```
Ext.getBody().load({
    url : 'htmlFragment.html',
    scripts : true
});
```

In the above snippet, we use call the load method of the result of the Ext.getBody call, and pass a configuration object, which specifies the url to fetch, which is our 'htmlFragment.html' file and scripts set to true. What happens when execute this code?

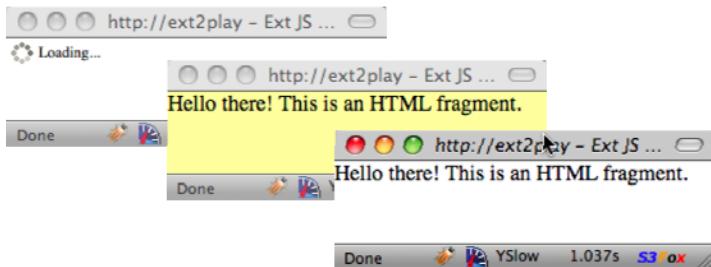


Figure 2.5 Loading an HTML fragment into the document body.

When executing the preceding code snippet, we see that the document body performs an AJAX request to retrieve our 'htmlFragment.html' file. While the file is being retrieved, it shows a loading indicator. Once the request is complete, the HTML fragment is injected into the DOM. We then see the entire body element highlighted yellow, which is an indication that our JavaScript was executed. We can now see that using the Ext.Element.load utility method is a great convenience over having to manually code an Ext.Ajax.request call.

And there you have it. Adding and removing elements to the DOM is a cinch when using Ext.Element. Ext has another way to make adding elements even simpler, especially if you have repeatable DOM structures to be placed in the DOM. These are the Template and XTemplate utility classes.

## 2.3 Using Templates and XTemplates

The Ext.Template class is a powerful core utility that allows you to create an entire DOM hierarchy with slots that can be filled in by data later. After you define a template, you can then use it to *stamp out* one or more of the pre-defined DOM structures with your data filling in the slots. Mastering templates will help you master UI widgets that use templates, such as the GridPanel, DataView and ComboBox.

### 2.3.1 Exercising Templates

We'll start out first by creating an extremely simple template and then we'll move on to create one that is much more complex.

```
var myTpl = new Ext.Template("<div>Hello {0}</div>");

myTpl.append(document.body, ['Marjan']);
myTpl.append(document.body, ['Michael']);
myTpl.append(document.body, ['Sebastian']);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

In the above example, we create an instance of `Ext.Template` and pass it a string representation of a div with a slot, which is marked in curly braces and store a reference in the variable `myTpl`. We then call `myTpl.append` and pass it a target element, which is `document.body` and data to fill in the slots, which in this case happens to be a single element array that contains a first name.

We do this three consecutive times, which results in three divs being appended to the DOM with each different first name filling in a slot. Below is an illustration of the result from our append calls.

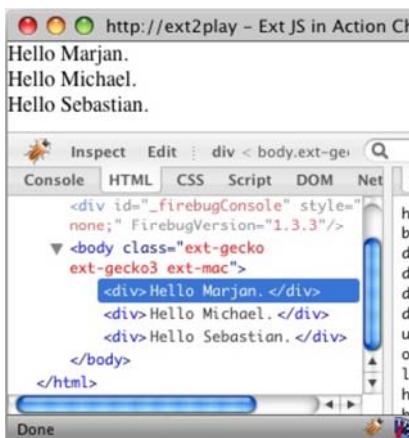


Figure 2.6 Using our first template to append nodes to the DOM with the exploded view in Firebug.

As we can see in figure 2.9, three divs were appended to the document body, each with different names. The benefits of using Templates should now be clear. We set the template once and apply it to the DOM with different values.

In the prior example, the slots were integers in curly braces and we passed in single item arrays. Templates can also map object key-values from plain objects. Here is an example of how to create a template that uses such syntax.

### **Listing 2.1 Creating a complex template**

```
var myTpl = new Ext.Template(                                     // 1
    '<div style="background-color: {color}; margin: 10;">',
    '<b> Name :</b> {name}<br />',
    '<b> Age :</b> {age}<br />',
    '<b> DOB :</b> {dob}<br />',
    '</div>'
);
myTpl.compile();                                              // 2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
myTpl.append(document.body, { // 3
    color : "#E9E9FF",
    name  : 'John Smith',
    age   : 20,
    dob   : '10/20/89'
});

myTpl.append(document.body, {
    color : "#FFE9E9",
    name  : 'Naomi White',
    age   : 25,
    dob   : '03/17/84'
})
{1}Creating our complex template
{2}Compiling the template for faster speed
{3}Appending our template to the document body
```

When creating our complex Template**{1}**, the first thing you'll probably notice is that we pass in quite a few arguments. We do this because when creating our template, it's much easier to view our pseudo HTML in a tab delimit format rather than a long string. The Ext developers were keen to this idea, so they programmed the Template constructor to read all of the *arguments* being passed, no matter how many.

In the Template pseudo HTML, we have slots for four data points. The first is `color`, which will be used to stylize the background of the element. The three other data points are `name`, `age` and `dob`, which will be directly visible when the template is appended.

We then move on to `compile{2}` our template, which speeds up the Template by eliminating regular expression overhead. Even though for these two operations, we *technically* don't need to compile it as we wouldn't see the speed benefits, but for larger applications, where many templates are stamped out, compiling has a very clear benefit. To be safe, I always compile templates after instantiating them

Lastly, we perform two append calls, where we pass in the reference element and a data object. Instead of passing an array like we did when in our first exploration of Templates, we pass in a data object, which has keys that match the template slots.

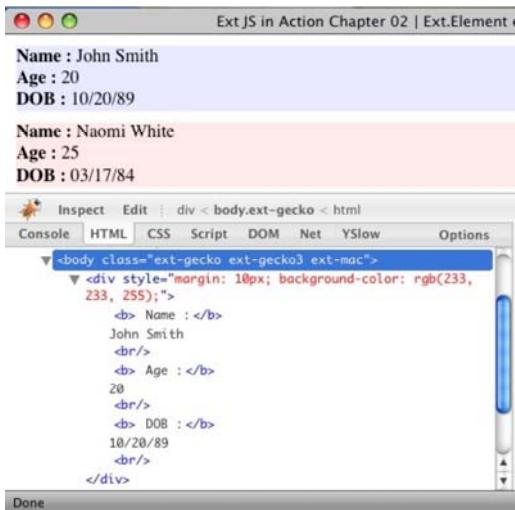


Figure 2.7 The result of our complex template with a DOM view in Firebug.

In using the Template, we were able to get two differently styled elements in the DOM with our being displayed. What if we had an *array* of objects? For instance, what if an Ajax request returned an array of data objects and we needed to apply a template for each data object? One approach could be looping through the array, which is easily done with a generic for loop or the more sexy `Ext.each` utility method. I say “nay” for that approach however. I would use `XTemplate` instead, which makes our code much cleaner.

### 2.3.2 Looping with XTemplates

`XTemplate`s can *technically* be used for single data objects, but can make life much easier when you have to deal with looping through data to stamp out HTML fragments on screen. It extends `Templates` and offers much more functionality. We'll start our exploration by creating an array of data objects and then create an `XTemplate`, which we'll use to stamp out HTML fragments.

#### Listing 2.1 Using an XTemplate to loop through data

```
var tplData = [
    {
        color : "#FFE9E9",
        name : 'Naomi White',
        age : 25,
        dob : '03/17/84',
        cars : ['Jetta', 'Pilot', 'S2000']
    },
    {
        color : "#E9E9FF",
        name : 'John Smith',
        age : 20,
    }
] // 1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
dob : '10/20/89',
cars : ['Civic', 'Accord', 'Pilot']
};

var myTpl = new Ext.XTemplate(
    '<tpl for=".">' , // 2
        '<div style="background-color: {color}; margin: 10;">',
            '<b> Name :</b> {name}<br />',
            '<b> Age :</b> {age}<br />',
            '<b> DOB :</b> {dob}<br />',
        '</div>',
    '</tpl>' // 3
);

myTpl.compile();

myTpl.append(document.body, tplData);
{1} Creating data to be used in by our XTemplate
{2} Instantiating an instance of XTemplate
{3} Instruct the XTemplate to "auto-fill" from an array
```

We start Listing 2.XX by first setting up an array of data objects **{1}**, which are like the data objects we used in our last Template exploration with the addition of a `cars` array, which we'll use in our next example.

Next, we instantiate an instance of `XTemplate`, which looks very much like our last Template configuration, except we encapsulate the `div` container with a custom `tpl` element with the attribute `for`, which contains the value `".`. The `tpl` tag is like a logic or behavior *modifier* for the template and has two operators, which are `for` and `if`, which alters the way the `XTemplate` generates the HTML fragments. In this case, the value of `".` simply instructs the `XTemplate` to loop through the root of the array for which it is passed and construct the fragment based on the pseudo HTML encapsulated inside the `tpl` element. When you look at the rendered HTML, there will be no `tpl` tags rendered to the DOM. The results of our efforts are...

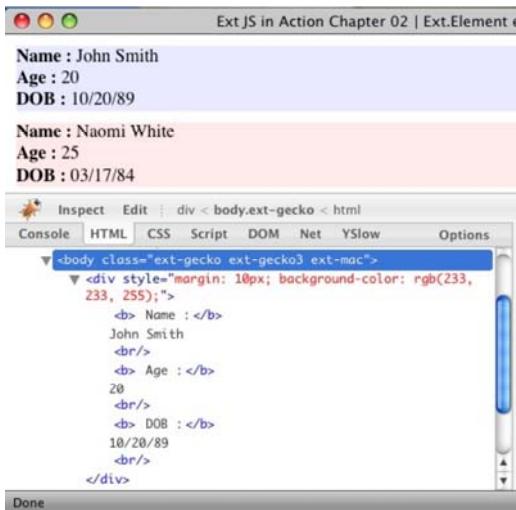


Figure 2.8 The results of our XTemplate usage with an exploded DOM view from Firebug.

...absolutely identical Template example. Remember, the advantage of using XTemplates in this case, is not having to write code to loop through the array of objects. We let the framework do the dirty work for us. The capabilities of XTemplates extend far beyond that of merely looping through arrays, which increases its usability exponentially.

### 2.3.3 Advanced XTemplate usage

You can configure them to loop through arrays within arrays and even have conditional logic. The next example will flex some XTemplate muscle and demonstrate many of these advanced concepts. Some of the syntax you're about to see will be foreign to you. Do not get discouraged. I will explain every bit.

#### Listing 2.2 Advanced XTemplate usage

```
var myTpl = new Ext.XTemplate(
    '<tpl for=".">' ,
        '<div style="background-color: {color}; margin: 10;">',
            '<b> Name :</b> {name}<br />',
            '<b> Age :</b> {age}<br />',
            '<b> DOB :</b> {dob}<br />',
            '<b> Cars : </b>',
            '<tpl for="cars">' ,
                '{.}' ,
                '<tpl if="this.isPilot(values)">' ,
                    '<b> (same car)</b>' ,
                '</tpl>' ,
                '{[ (xindex < xcount) ? " , " : "" ]}' ,
            '</tpl>' ,
            '<br />',
        '</div>' ,
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

'</tpl>',
{
  isPilot : function(car) {
    return car == 'Pilot';
  }
};

myTpl.compile();

myTpl.append(document.body, tplData);

{1} XTemplate loops through the cars data
{2} XTemplate displays all data in the array
{3} Execute the this.isPilot method
{4} Arbitrarily JS code, testing for the end of the array
{5} An object containing a local member, isPilot

```

This usage of XTemplate exercised quite a few advanced concepts. The first of which is looping within a loop**{1}**. Rememberm, the `for` attribute instructs the xtemplate to loop through a list of values. In this case, the `for` attribute has the value of 'cars', which differs from the value that is set for the first `for` attribute, "..". This will instruct the XTemplate to loop through this block of pseudo HTML for each individual car. Remember that `cars` is an array of strings.

Inside of this loop, we have a string with "`{ . }`"**{2}**, which instructs XTemplate to place the value of the array at the current index of the loop. In simple terms, the name of a car will be rendered at this position.

Next, we see a `tpl` behavior modifier with an `if` attribute**{3}**, which executes `this.isPilot` and passes values. The `this.isPilot` method is actually something that is generated at the very end of our XTemplate**{5}**. We'll speak more about this in a bit. Essentially the `if` attribute is more like an *if condition*, where the XTemplate will generate HTML fragments *if* the condition is met. In this case, `this.isPilot` must return true for the fragment that is encapsulated inside of this `tpl` flag to be generated.

The `values` property is an internal reference of the values for the array we're looping through. Because we're looping through an array of strings, it references a single string, which is the name of a car.

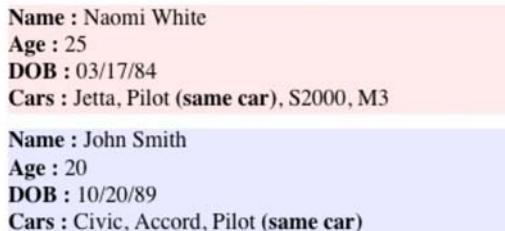
In the next line, we are arbitrarily executing JavaScript code**{4}**. Anything encapsulated in curly braces and brackets ("`{[ ... JS code ... ]}`"), will be interpreted as generic JavaScript and has access to some local variables that are provided by XTemplate and can change with each iteration of the loop. In this case, we're simply checking to see if the current index (`xindex`) is less than the amount of items in the array (`xcount`) and returning either a comma with a space or an empty string. Performing this test inline will ensure that commas are placed exactly between the names of cars.

The last item of interest is the object which contains our `isPilot` method**{5}**. Including an object (or reference to an object) with a set of members with the passing arguments to the XTemplate constructor will result in those members being applied directly to the instance of XTemplate itself. This is why we called `this.isPilot` directly in the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

if condition of one of our `tpl` behavior modifier pseudo elements. The all of these member methods are called within the scope of the instance of the XTemplate for which they are being passed. This concept is extremely powerful, but can be dangerous, as you can override an existing XTemplate member, so please try to make your methods or properties as unique. Our `isPilot` method simply uses JavaScript shorthand to test if the passed string, `car`, is equal to "Pilot" and will return `true` if it is, else `false`. Here the results of our advanced XTemplate exercise.



The screenshot shows two separate XTemplate renderings. The first rendering has a pink background and contains the following data:  
**Name :** Naomi White  
**Age :** 25  
**DOB :** 03/17/84  
**Cars :** Jetta, Pilot (**same car**), S2000, M3

The second rendering has a light blue background and contains the following data:  
**Name :** John Smith  
**Age :** 20  
**DOB :** 10/20/89  
**Cars :** Civic, Accord, Pilot (**same car**)

Figure 2.9 The results from our advance XTemplate exercise

In looking at the results from our advance XTemplate exercise, we see that all of our behavior injections worked as planned. We have all of our cars listed, which includes proper comma placement. We can tell that our arbitrary JavaScript injection worked because the string "(**same car**)" is placed to the right of the "Pilot" name.

As we can see, Templates and XTemplates have myriad of benefits over generic DOM injections using `Ext.Element` to stamp out HTML fragments with data. I encourage you to look over the `Template` and `XTemplate` API pages for more details and examples on how to use these utilities. Our next exposure to Templates will be when we learn how to create a custom `ComboBox`.

## 2.4 Summary

In this chapter we discussed how JavaScript application logic was launched in the olden days with the `onLoad` handler of `body` element. Remember that browsers typically have their own way of publishing when the DOM is ready for manipulation, which causes a code management nightmare. In exercising `Ext.onReady`, we learned that it takes care of launching our application code just at the right time for each browser, so we can worry about the important stuff - application logic.

We then took an in depth look the `Ext.Element` class, which wraps and provides end-to-end management for DOM nodes. We exercised a few of the management utilities for DOM Nodes where we added and removed elements. All UI widgets use the `Ext.Element`, making it one of the most used components of the core framework. Each widget's element can be access via the (public) `getEl` method or the (private) `e1` property, but only after it has been rendered.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Lastly, we learned about the `Template` class to inject HTML fragments into the DOM. We also jumped into advanced techniques with the `XTemplates` and learned how to embed behavioral modifying logic into the template definition itself, producing results depending on the data that it was provided.

Looking forward, we're going to start focusing on the UI side of the framework, where we'll jump right into the core concepts and models that drive the framework.

# 3

## *Events, Components and Containers*

I recall my early days with the framework when I started learning by toying with the examples and reading the API documentation. I spent many hours on some of the most important core UI concepts, such as adding user interaction, the reusability of widgets and how widget contain or control another. For instance, how does one make the click of an anchor tag display an Ext Window? Sure there is a generic JavaScript way of attaching an event handler, but I wanted to use Ext JS. Likewise, I needed to know how to get widgets to communicate with other. An example would be how to reload a `GridPanel` when a row of another `GridPanel` is clicked. Also, how do we add and remove items dynamically from a `Panel`? Or how can I find a particular field within a form panel based on the type field?

This chapter is designed to cover these core concepts, which are essential to building rich and interactive user interfaces with the framework. Being that we just covered `Ext.Element` in Chapter 2, we will leverage what we learned to setup native click handlers on DOM elements and learn how events flow in the DOM. We'll also touch upon how widget events work from registration to firing the event.

We will also explore the deep caverns of the fundamental UI building block, the Component, and learn how it serves as the central model for all UI widgets by implementing a template for standard behaviors known as the Component Lifecycle.

Lastly, we'll take some time discussing the Container class, where we'll get an in-depth understanding of how widgets can manage child items. We'll learn how to add and remove items dynamically to widgets like the `Panel`, which can be used as a building block for dynamically updating UIs.

## 3.1 Managing Events with Observable

For me, one of the most fun tasks for developing web applications is coding to manage events. Events can be thought of as a 'signal' that is sent by some source when something occurs that could require action. Understanding events is one of the key core concepts from which you must be very familiar with, as this will aid you in developing user interfaces, which provide truly rich user interaction. For instance, what if you wanted to display a context menu on a `GridPanel` when a user clicked on a row? You would setup an event handler for the `rowcontextmenu` event, which will create and display the context menu.

Likewise, knowing how Ext components communicate with events is equally as important. This section will give you the fundamental knowledge that you'll need.

### 3.1.1 Taking a step back

Though we may not have realized it, but we use an event driven operating system every day. All modern User Interfaces are driven by events, which on a high level, generally come from inputs such as mouse or keyboard, but can be synthesized by software. Events that are sent are dispatched or 'fired'. Methods that take actions on these events are 'listeners' and are sometimes called a 'handler'.

Just like modern Operating Systems, the browser too has an event model where it fires events because of user input. This powerful model allows us to leverage that input and perform complex tasks, such as refreshing a grid or applying a filter. Just about every interaction that is performed with the browser fires events that can be leveraged. These are known as DOM based events.

### 3.1.2 DOM Based events

Recall that events can come from user input or synthesized by software. We need to first explore DOM based events, which are initiated by user input before we can have fun with software-based events. A lot of old school web developers attached listeners directly on the HTML element via the `onclick` property:

```
<div id="myDiv" onclick="alert(this.id + ' was clicked');">Click me</div>
```

This method of adding event handlers was standardized by Netscape (remember them?) many years ago and is considered to be ancient practice by most modern JavaScript developers. This is due to the fact that it adds a dependency for embedded JavaScript in HTML and leads to a code management nightmare. I would suggest avoiding this method at all cost.

The way events are managed across browsers is added to our ever grow list of cross-browser incompatibilities. Luckily for us, Ext JS takes care of that and presents a unified interface for us to leverage:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
var el = Ext.get('myDiv');
el.on('click', doSomething);
```

Here, we use the native method, `Ext.get`, which allows Ext to wrap its element management class, `Ext.Element`, around a referenced element. Embedded with each instance of `Ext.Element` is the usage of the Ext event management engine, `Ext.util.Observable`. It is important to remember that all event management with Ext stems from `Ext.util.Observable`, which serves as a base for all DOM elements and components that need to manage events.

Next, we attach a listener by calling the `el.on` and pass the native event to be handled and a method to perform an action on the event. Ext takes event listener registration a step further by allowing you to pass a reference to the scope from which the event handler is to be called and any parameters:

```
var el = Ext.get('myDiv');
el.on('click', doSomething, scopeRef, [opt1, opt2]);
```

It's important to note that the default scope is always the object from which the handler is being defined. If scope were not explicitly passed, the method `doSomething` would be called within the scope of the object `el`, which is an instantiation of `Ext.Element`.

#### NOTE

The concept of scope can be confusing for some. If you are unsure about what it is, the following URL has a great tutorial that covers scope in great detail. [http://www.digital-web.com/articles/scope\\_in\\_javascript/](http://www.digital-web.com/articles/scope_in_javascript/)

Now that have learned and exercised simple event handling on an Element, we'll take a quick glance at how events flow in the DOM.

#### 3.1.3 Event flow in the DOM

In the early days of the Internet, Netscape and Microsoft had two completely separate approaches with regard to event flow direction. In the Netscape model, events flowed downward from the document body to the source, which is known as *event capture*. The Microsoft model, known as bubbling, was exactly the inverse of capture, where the event is generated from the node from which the user action is performed and *bubbles* up to the document object, which performs a default action based on what type of node was clicked. Thankfully, the W3C model combined these two models and is what we use now.

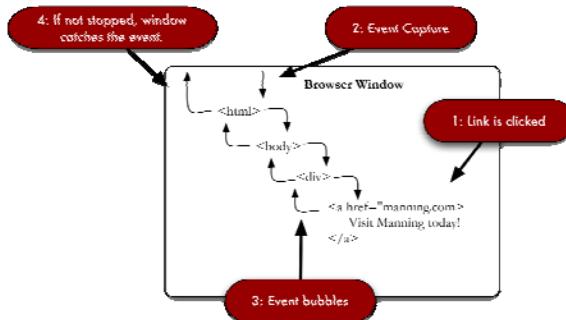


Figure 3.1 The W3C Event model, where events first flow downward (capture) and then return up the tree (bubbling).

As we can see in Figure 3.1, the user clicking on the anchor tag generates a dom-level click event. This causes the event capture phase to occur, which cascades the event down the DOM tree, where it eventually lands at the “target” node, the anchor tag. If the event is not stopped, it will bubble upwards back to the browser window, where it will cause the `window.location` to change.

#### NOTE

If you are new to DOM events and want to learn more about the inner workings, Peter-Paul Koch has an excellent article on his site, which, he explains events in browsers in much greater detail than what is covered in this book. Here is the direct link: [http://www.quirksmode.org/js/events\\_order.html](http://www.quirksmode.org/js/events_order.html).

Most of the time you’re developing event handlers on the DOM, you’re going to have to worry about bubbling. To demonstrate why this is important, we’ll need to setup some HTML and attach event handlers to the node. We’re going to use the Firebug console to echo out messages. If you don’t wish to use firebug, the generic JavaScript `alert` box will work.

Suppose you have the following HTML and you want to apply separate click listeners to both the `div` and the `anchor` tags.

```

<div id="myDiv">
  MyDiv Text
  <a href="#" id="myHref">
    My Href
  </a>
</div>

```

We will register a click handler to the outer most element, `myDiv` and will use chaining so we don’t have to set a static reference to the result of the `Ext.get` method call. We will also

pass an anonymous function as the second parameter instead of passing a reference to a function:

```
Ext.get('myDiv').on('click', function(eventObj, elRef) {
    console.log('myDiv click Handler, source element ID: ' + elRef.id);
});
```

After rendering the page expand firebug and click on the *anchor tag*. You'll see a firebug console message indicating that you clicked the myHref tag. Hold the phone... we only assigned a listener to the anchor's parent element, 'myDiv'. How could this be?

This is event bubbling in action. Remember that the click event generated from the anchor tag bubbled upwards to the container div, where we attached the click handler. That bubbled event triggered the event handler, causing its execution, thus the console message.

Next, attach a click handler to the anchor tag:

```
Ext.get('myHref').on('click', function(eventObj, elRef) {
    console.log('myHref click handler, source element ID: ' + elRef.id);
});
```

Refresh your page and click on the anchor again, you'll see two events fired:



Figure 3.2 The results of our click event handlers in the firebug console.

Having both event listeners fired could be troublesome and considered a waste of resources, especially if only one is needed. In order to stop this from happening, we need to stop the bubbling (propagation) of the click event in the anchor. Only then can we have separation from the anchor and the div element click event handlers.

### 3.1.4 Burst the bubble

To prevent both event handlers from triggering, we'll have to modify the anchor click event handler to include a `stopEvent` call on the instance of `Ext.EventObject` (`eventObj`) passed to the handler:

```
Ext.get('myHref').on('click', function(eventObj, elRef) {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
        eventObj.stopEvent();
        console.log('myHref click handler, source element ID: ' + elRef.id);
    })
}
```

Refresh the page and click the anchor again. This time, you'll only see one firebug console message per click, indicating that you've clicked on the anchor tag. Likewise, clicking on the div element will produce a message stating that you clicked on the div.

The eventObj.stopEvent method call stops the event bubbling in its tracks. This is important to remember because there are other interactions for which you may want to cancel event propagation, such as contextmenu, where you would want to show your own context menu instead of the browser's default menu.

We'll modify our example to listen to listen to the contextmenu event to perform this task:

```
Ext.get('myDiv').on('contextmenu', function(eventObj, elRef) {
    console.log('myDiv contextmenu Handler, source element ID: ' + elRef.id);
});

Ext.get('myHref').on('contextmenu', function(eventObj, elRef) {
    eventObj.stopEvent();
    console.log('myHref contextmenu Handler, source element ID: ' + elRef.id);

    if (!this.ctxMenu) {
        this.ctxMenu = new Ext.menu.Menu({
            items : [
                {text : "This is"},
                {text : "our custom"},
                {text : "context menu"}
            ]
        });
    }
    this.ctxMenu.show(elRef);
});
```

In this example, we change the registered event from 'click' to 'contextmenu' for the both div and anchor elements. For the div event handler, we simply echo out a console message, providing an indication that the handler was triggered. For the anchor event handler, we stop the event propagation, log the event trigger in the firebug console, create a new Ext Menu and display it below the anchor element.



Figure 3.3 Displaying our custom context menu by means of a “contextmenu” event handler.

If you right click on the div, you'll see the default browser context menu. However, if you right click on the anchor (Figure 3.3), you'll see your own custom menu because we halted the propagation of the contextmenu event, preventing it from bubbling back up to the browser.

As we can see, the registration of event handlers on DOM nodes is pretty much a simple task. Knowing if you have to stop an event from bubbling up can be tricky. Certainly the contextmenu event is the only event that comes to mind that always needs to be stopped to prevent default browser behavior.

Next, we'll focus our attention on software driven events, which is another key concept of the framework as inter-component events are used extensively in the framework.

### 3.1.5 Software driven events

Almost every Ext widget and component has custom events that it fires when it deems necessary. For instance, when a widget is done rendering in the DOM, it fires an `Ext.util.Observable`, which gives the widgets the ability to exhibit similar behavior to complex UI environments such as your desktop UI. Events from widgets and components can include DOM based events that are bubbled up, such as `click` and `keyUp` or Ext-internal events such as `beforerender` and `datachanged`.

The API is a great place to read about events for a particular descendant of `Observable`. The last section of each API page is dedicated to Public Events from which other components can register listeners and take action.

### 3.1.6 Registration of Events and Event Listeners

Before an Ext-based event can be fired, it must be added to the list of events for that instance of `Observable`. This is typically done with the `addEvent` method, which stems from the `Observable` class. Let's instantiate a new instance of `Observable` and add a custom event.

```
var myObservable = new Ext.util.Observable();
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
myObservable.addEvents('sayHello');  
myObservable.addEvents('sayGoodbye');
```

If you want to register more than one event, you can pass one event label per argument:

```
myObservable.addEvents('sayhello', 'saygoodbye');
```

Alternatively, you can pass in a configuration object that has a list of event labels and if they should be enabled by default:

```
myObservable.addEvents({  
    'sayHello' : true,  
    'sayGoodbye' : true  
})
```

Now that we have our events registered, we need to register an event handler. This should look familiar to you.

```
myObservable.on('sayHello', function() {  
    console.log('Hello stranger');  
});
```

Here, we are providing an event handler for our custom event, 'sayHello'. Notice how this is exactly how we registered events on the DOM. The handler will simply log a console message in firebug as an indication that it was executed. Being that we have our custom event defined and we have a listener registered, we need to fire the event so that we may watch the event handler get called:

```
myObservable.fireEvent('sayHello');
```

We can see that firing the event causes the handler to execute, thus resulting in a Firebug console message being displayed.

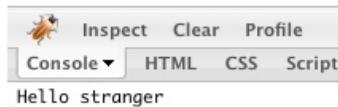


Figure 3.4 A Firebug console message being displayed, indicating that the registered handler for the sayHello event was triggered.

A lot of events from the framework pass parameters when firing, which among the parameters is a reference to the component firing the event. Create an event handler for a

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

custom 'sayGoodbye' event so that it accepts two parameters, `firstName` and `lastName`:

```
var sayGoodbyeFn = function(firstName, lastName) {
    console.log('Goodbye ' + firstName + ' ' + lastName + '! ');
}

myObservable.on('sayGoodbye', sayGoodbyeFn);
```

Here, we define a method named `sayGoodbyeFn`, which accepts the two parameters. We then call `myObservable.on`, which is shorthand for `myObservable.addListener`, to register the event handler `sayGoodbyeFn`. Next, fire the 'sayGoodbye' event and pass a first and last name.

```
myObservable.fireEvent('sayGoodbye', 'John', 'Smith');
```

When calling `fireEvent`, the only required parameter is the first one, which is the event name. Parameters passed on thereafter are relayed to the event handler. The result of the 'sayGoodbye' event being fired with the `firstName` and `lastName` parameters passed should appear in your firebug console.



Figure 3.5 A Firebug Console message displays as the result of the firing of the `sayGoodbye` event.

Remember that if you register an event handler without specifying a scope, it will be called within the scope of the component that is firing the event. As we continue our journey into the Ext JS world, we will revisit events again as we will use them to 'wire up' widgets together.

Just as the registration of event handlers are important to get tasks complete, deregistration of event handlers are equally as important to ensure proper cleanup when an event handler is no longer required. Fortunately, the method call to do so is extremely simple:

```
myObservable.removeListener('sayGoodbye', sayGoodbyeFn);
```

Here, we call `myObservable.removeListener`, passing in the event from which the listener is to be deregistered and the listener method to deregister. The shorthand for `removeListener` is `un` (opposite of `on`) and is commonly used in the framework and application code. Here is what the preceding code looks like in shorthand.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
myObservable.un('sayGoodbye', sayGoodbyeFn);
```

Managing events and event listeners is as straightforward as described. Always remember to deregister event handlers when they are no longer needed. This will help reduce the memory footprint of your application and help ensure that no exceptions occur due to an event handler being triggered when it should not be.

We're now familiar with registering events to DOM elements and have glanced at software-driven events. We can now start to gravitate towards the UI portion of the framework. Before we dive into configuring and constructing widgets, we need to look at the Component model, which serves as the base model for all UI widgets. Having a solid grasp on the how the component model works in Ext 3.0 will allow you to better utilize the UI portion of framework, especially when managing child items of a Container.

### **3.2 The Component Model**

The Ext Component model provides a centralized model that provides many of the essential component related tasks, which includes a set of rules dictating how the component instantiates, renders and is destroyed, which is known as the Component lifecycle.

All UI widgets are a descendant of `Ext.Component`, which means that all of the widgets conform to the rules dictated by the model. Figure 3.2 partially depicts how many items actually subclass `Component`, directly or indirectly.

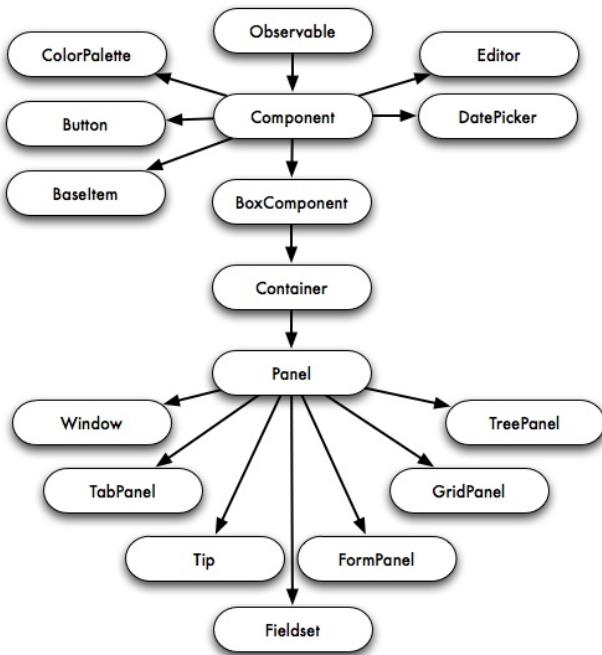


Figure 3.6 This illustration of the Ext class hierarchy focuses on some of the common descendants of Ext.Component and depicts how widely used the Component model is used in the framework.

Knowing that each and every UI widget is going to behave introduces stability and predictability into the framework, which I enjoy. The Component model also supports direct instantiation of classes or deferred instantiation, which is known as XTypes. Knowing which to use when can enhance the responsiveness of your application.

### 3.2.1 XTypes and Component Manager

Ext 2.0 introduced a radical new concept known as an XType, which allows for lazy instantiation of components, which can speed up complex user interfaces and can clean up our code quite a bit.

In Short, an XType is nothing more than a plain JavaScript object, which generally contains an `xtype` property with a string value denoting which class the XType is for. Here is a quick example of an XType in action:

```

var myPanel = {
    xtype      : 'panel',
    height    : 100,
    width     : 100,
    html      : 'Hello!'
}

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

In the configuration object above, `myPanel` is XType configuration object that would be used to configure an `Ext.Panel` widget. This works because just about every widget is registered to the `Ext.ComponentMgr` class with a unique string key and a reference to that class, which is then referred to as an XType. At the tail end of each Ext UI widget class, you'll find the registration of that widget in the `Ext.ComponentMgr`.

Registration of a component is simple:

```
Ext.reg('myCustomComponent', myApp.customClass);
```

The act of registering a component to the `ComponentMgr` appends or replaces the new Component to the internal reference map of the `ComponentMgr` singleton. Once registration is complete, you can specify your custom component as an XType:

```
new Ext.Panel({
    ...
    items : {
        xtype : 'myCustomComponent',
        ...
    }
});
```

When a visual component, that can contain children, is initialized, it looks to see if it has `this.items` and will inspect `this.items` for XType configuration objects. If any are found, it will attempt to create an instance of that Component using the `ComponentMgr.create`. If the `xtype` property is not defined in the configuration object, the said visual component will use its `defaultType` property when calling `ComponentMgr.create`.

I realize that this may sound a tad confusing at first. I think we can better understand this concept if we exercise it. To do this, we'll create a window with an accordion layout that includes two children, one of which will not contain an `xtype` property. First, let's create our configuration objects for two of the children:

```
var panel1 = {
    xtype : 'panel',
    title : 'Plain Panel',
    html : 'Panel with an xtype specified'
}

var panel2 = {
    title : 'Plain Panel 2',
    html : 'Panel with <b>no</b> xtype specified'
}
```

Notice that `panel1` has an explicit `xtype` value of '`panel`', which in turn will get used to create an instance of `Ext.Panel`. Objects `panel1` and `panel2` are very similar, but have two distinct differences. Object `panel1` has an `xtype` specified while `panel2` does not.

Next, we'll create our window, which will use these xtypes:

```
new Ext.Window({
    ...
    ...
})
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

width      : 200,
height     : 150,
title      : 'Accordion window',
layout     : 'accordion',
border     : false,
layoutConfig : {
    animate : true
},
items : [
    panel1,
    panel2
]
}).show();

```

In our new instantiation of `Ext.Window`, we pass items, which is an array of references to the two configuration objects we created earlier. The rendered window should appear as illustrated in Figure 3.7. Clicking on a collapsed panel will expand and collapse any other expanded panels and clicking on an expanded panel will collapse it.



Figure 3.7 The results of our XType exercise; an Ext Window, which has two child panels derived from XType configuration objects.

One of the lesser-known advantages of using XTypes is developing somewhat cleaner code. Because you can use plain object notation, you can specify all of your XType child items inline, resulting in cleaner and more streamlined code. Here is the previous example reformatted for to include all of its children inline:

```

new Ext.Window({
    width      : 200,
    height     : 150,
    title      : 'Accordion window',
    layout     : 'accordion',
    border     : false,
    layoutConfig : {
        animate : true
    },
    items : [
        {
            xtype : 'panel',
            title : 'Plain Panel',
            html  : 'Panel with an xtype specified'
        },
        {
            title : 'Plain Panel 2',
            html  : 'Panel with <b>no</b> xtype specified'
        }
    ]
});

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        }
    ]).show();
});
```

As you can see above, we've included all of the child configuration items inline with Window configuration object. The performance enhancements with using XTypes cannot be seen with such a plain example. The biggest XType-based performance gains come from rather large applications, where there is a rather large number of Components to be instantiated.

Components also contain another performance-enhancing feature, which is known as lazy rendering. That is a component is only rendered when necessary.

### 3.2.2 Component rendering

The Ext.Component class supports both direct and lazy (on-demand) render models. Direct rendering can happen when a descendant of Component is instantiated with either renderTo or applyTo attribute, where renderTo points to a reference from which the component renders itself in to and applyTo references an element that has HTML that is structured in such a way that allows the component to create its own child elements based on the referenced HTML. You typically would use these parameters when if you wanted a component to be rendered upon instantiation. For instance:

```
var myPanel = new Ext.Panel({
    renderTo : document.body,
    height   : 50,
    width    : 150,
    title    : 'Lazy rendered Panel',
    frame    : true
});
```

The end result of the code above would be the immediate render of the Ext.Panel, which sometimes is favorable, and other times not. The times where it is not favorable can be when you want to *defer* rendering to another time in code execution or the Component is a child of another.

If you want to defer the rendering of the Component, simply omit the renderTo and applyTo attributes, and call the Components render method when you (or your code) deem it necessary.

```
var myPanel = new Ext.Panel({
    height : 50,
    width  : 150,
    title  : 'Lazy rendered Panel',
    frame  : true
});
// ... some business logic...
myPanel.render(document.body);
```

In the above example, we instantiate an instance of Ext.Panel and create a reference to it, myPanel. After some hypothetic application logic, we call myPanel.render and pass a reference to document.body, which renders the panel to the document body.

You could also pass an ID of an element to the render method:

```
myPanel.render('someDivId');
```

When passing an element ID to the render method, Component will use that ID with Ext.get to manage that element, which gets stored in its local el property. If this rings a bell, you may recall the conversation we had in the last chapter, when we were discussing Ext.Element, where we learned you can access a widget's el property or use its *accessor* method, getEl to get obtain the reference.

There is one major exception to this rule however. You *never* specify applyTo or renderTo when the Component is a child of another. There is a parent-child relationship that Components that *contain* other Components have, which is known as the Container model. If a Component is a child of another component, it is specified in the items attribute of the configuration object, its parent will manage the call to its render method when required. This is known as lazy or deferred rendering.

We'll investigate Containers later in this chapter, where we'll learn more about the parent-child relationship Components can have. But first, need to understand the component life cycle, which details how the components are created, rendered and eventually destroyed. Learning how each of phase works will better prepare you for building robust and dynamics interfaces and can assist troubleshooting issues.

### 3.3 The Component Life Cycle

Ext Components, just like everything in the real world, have a life cycle where they are created, used and destroyed. This lifecycle is broken up into three major phases: initialization, render and destruction as displayed in figure 3.8.



Figure 3.8 The Ext Component lifecycle always starts with initialization and always ends with destruction. The component need not enter the render phase to be destroyed.

To better utilize the framework, we must understand how the lifecycle works in a finer detail. This is especially important if you will be building extensions, plugins or composite

components. Quite a bit of steps take place at each portion of a lifecycle phase, which is controlled by the base class, `Ext.Component`.

### 3.3.1 Initialization

The initialization phase is when the component is born. All of the necessary configuration settings, event registration and pre-render processes take place in this phase as illustrated in Figure 3.9.

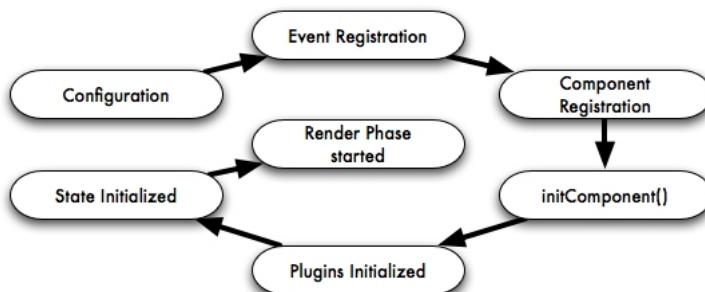


Figure 3.9 The initialization phase of the component lifecycle executes important steps such as event and component registration as well as the calling the `initComponent` method. It's important to remember that a component can be instantiated but may not be rendered.

Let's explore each step of the initialization phase:

1. The configuration is applied: When instantiating an instance of a Component, you pass a configuration object, which contains all of the necessary parameters and references to allow the component to do what it's designed to do. This is done within the first few lines of the `Ext.Component` base class.
2. Registration of the base Component events: Per the Component model, each descendant of `Ext.Component` has, by default, a set of core events that are fired from the base class. These are fired before and after some behaviors occur: enable/disable, show, hide, render, destroy, state restore, state save. The before events are fired and tested for a successful return of a registered event handler and will cancel the behavior before any real action has taken place. For instance, when `myPanel.show` is called, it fires the `beforeshow` event, which will execute any methods registered for that event. If the `beforeshow` event handler returns false, `myPanel` does not show.
3. ComponentMgr registration: Each component that is instantiated is registered with the `ComponentMgr` class with a unique Ext-generated string ID. You can choose to override the Ext-generated ID by passing an `id` parameter in the configuration object passed to a constructor. The main caveat is that if a registration request occurs with

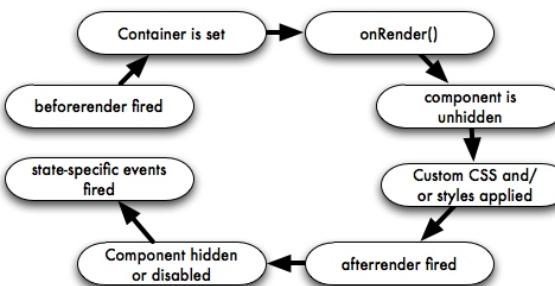
a non-unique registration ID, the newest registration will override the previous one. Be careful to use unique IDs if you plan on using your own ID scheme.

4. initComponent is executed: The `initComponent` method is where a lot of work occurs for descendants of Component like registration of descendant-specific events, references to data stores and creation of child components. `initComponent` is used as a supplement to the constructor, thus is used as the main point to extending Component or any descendant thereof. We will elaborate on extending with `initComponent` later on.
5. Plugins are initialized: If plugins are passed in the configuration object to the constructor, their `init` method is called, with the parent component passed as a reference. It is important to remember that the plugins are called upon in the order in which they are referenced.
6. State is initialized: If the component is state-aware, it will register its state with the global `StateManager` class. Many Ext widgets are state-aware.
7. Component is rendered: If the `renderTo` or `applyTo` parameters are passed into the constructor, the render phase begins at this time, else the component then lies dormant, awaiting its `render` method to be called.

This phase of a Component's life is usually the fastest as all of the work is done in JavaScript. It is particularly important to remember that the component does not have to be rendered to be destroyed.

### 3.3.2 Render

The render phase is the one where you get visual feedback that a component has been successfully initialized. If the initialization phase fails for whatever reason, the component may not render correctly or at all. For complex components, this is where a lot of CPU cycles get eaten up, where the browser is required to paint the screen and computations take place to allow all of the items for the component to be properly laid out and sized. Figure 3.8 illustrates the steps of the render phase.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Figure 3.10 The render phase of the a component's life can utilize a lot of CPU as it requires elements to be added to the DOM and calculations to be performed to properly size and manage them.

If `renderTo` or `applyTo` are not specified a call to the `render` method must be made, which triggers this phase. If the component is not a child of another Ext component, then your code must call the `render` method, passing a reference of the DOM element:

```
someComponent.render('someDivId');
```

If the component is a child of another component, then its `render` method will be called by the parent component. Let's explore the different steps of the render phase:

1. `beforerender` is fired: The component fires `beforerender` event and checks the return of any of the registered event handlers. If a registered event handler returns false, the component halts the rendering behavior. Recall that step two of the initialization phase registers core events for descendants of `Component` and that "before" events can halt execution behaviors.
2. The container is set: A component needs a place to live, and that place is known as its container. Essentially, if you specify a `renderTo` reference to an element, the component adds a single child `div` element to the referenced element, known as its container and renders the component inside that newly appended child. If an `applyTo` element is specified, the element referenced in the `applyTo` parameter becomes the components container, and the component only appends items to the referenced element that are required to render itself. The DOM element referenced in `applyTo` will then be fully managed by the component. You generally pass neither when the component is a child of another component, in which the container is the parent component. It is important to note, that you should only pass `renderTo` or `applyTo`, not both. We will explore `renderTo` and `applyTo` later on, when we learn more about widgets.
3. `onRender` is executed: This is a crucial step for subclasses of `Component`, where all of the DOM elements are inserted to get the component rendered and painted on screen. Each subclass is expected to call its `superclass.onRender` first when extending `Ext.Component` or any subclass thereafter, which ensures that the `Ext.Component` base class can insert the core DOM elements needed to render a component.
4. The component is "unhidden": Many components are rendered hidden using the default Ext CSS class like '`x-hidden`'. If the `autoShow` property is set, any Ext CSS classes that are used for hiding components are removed. It is important to note that this step does not fire the `show` event, thus any listeners for that event will not be fired.

5. Custom CSS classes or styles are applied: Custom CSS classes and styles can be specified upon component instantiation by means of the `cls` and `style` parameters. If these parameters are set, they are applied to the container for the component. It is suggested that you use `cls` instead of `style`, as CSS inheritance rules can be applied to the components children.
6. The `render` event is fired: At this point, all necessary elements have been injected into the DOM and styles applied. The `render` event is fired, triggering any registered event handlers for this event.
7. `afterRender` is executed: The `afterRender` event is a crucial post-render method that is automatically called by the `render` method within the `Component` base class and can be used to set sizes for the `Container` or perform any other post-render functions. All subclasses of `Component` are expected to call their `superclass.afterRender` method.
8. The component is hidden and/or disabled: If either `hidden` or `disabled` is specified as true in the configuration object, the `hide` or `disable` methods are called, which both fire their respective “before<action>” event, which is cancelable. If both are true, and the “before<action>” registered event handlers do not return false, the component is both hidden and disabled.
9. State-specific events are fired: If the component is state-aware, it will initialize its state-specific events with its `Observable` and register the `this.saveEvent` internal method as the handler for each of those state events.
10. Once the render phase is complete, unless the component is disabled or hidden, it's ready for user interaction. It stays alive until its `destroy` method is called, in which it then starts its destruction phase.

The render phase is generally where a component spends most of its life until it meets its demise with the destruction phase.

### **3.3.3 Destruction**

Just like in real life, the death of a component is a crucial phase in its life. Destruction of a component performs critical tasks, such as removing itself and any children from the DOM tree, deregistration of the component from `ComponentMgr` and deregistration of event listeners as depicted in Figure 3.9.

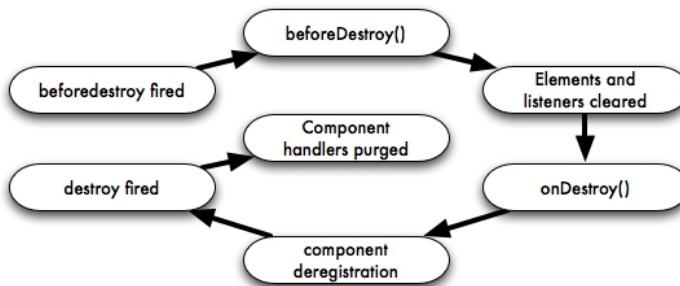


Figure 3.11 The destruction portion of a component's life is equally as important as its initialization as event listeners and DOM elements must be deregistered and removed, reducing over-all memory usage.

The component's destroy method could be called by a parent Container, or by your code. Here are the steps in this final phase of a Component's life:

1. `beforedestroy` is fired: This, like many “`before<action>`” events, is a cancelable event, preventing the component's destruction if its event handler returns `false`.
2. `beforeDestroy` is called: This method is first to be called within the Component's `destroy` method and is the perfect opportunity to remove any non-component items, such as toolbars or buttons. Any subclass to Component is expected to call its `superclass.beforeDestroy`.
3. Element and Element listeners purged: If a component has been rendered, any handlers registered to its Element are removed and the Element is removed from the DOM.
4. `onDestroy` is called: While the Component class itself does not perform any actions within the `onDestroy` method, subclasses are expected to use this to perform any post-destruction actions, such as removal of data stores. The Container class, which subclasses Component indirectly, manages the destruction of all registered children by in its `onDestroy` method, alleviating this task from the end developer.
5. Component is unregistered from ComponentMgr: The reference for this component in the ComponentMgr class is removed.
6. The `destroy` event is fired: Any registered event handlers are triggered by this event, which signals that the component is no longer in the DOM.
7. Component's event handlers are purged: All event handlers are deregistered from the Component.

And there you have it, an in-depth look at the Component lifecycle, which is one of the features of the Ext framework that makes it so powerful and successful.

Be sure to not dismiss the destruction portion of a component's lifecycle if you plan on developing your own custom components. Many developers have gotten into trouble where they've ignored this crucial step and have code that has left artifacts such as data stores which continuously poll web servers, or event listeners that are expecting an Element to be in the DOM, were not cleaned properly and cause exceptions and the halt of execution of a crucial branch of logic.

Next, we'll look at the Container class, which is a descendant of Component and gives Components the ability to manage other components in a parent-child relationship.

### 3.4 Containers

The Container model is a behind-the-curtains class that provides a foundation for components to manage their child items and is often overlooked by developers. This class provides a suite of utilities, which includes add, insert and remove methods along with some child query, bubble and cascade utility methods. These methods are used by most of the descendants, which include Panel, Viewport and Window.

In order to learn how these tools work, we need to build a Container with some child items for us to use. The following listing is rather long and involved, but stay with me on this. The reward is just around the corner.

#### **Listing 3.1 Building our first container**

```

var panel1 = { //1
    html : 'I am Panel1',
    id   : 'panel1',
    frame: true,
    height: 100
}

var panel2 = {
    html : '<b>I am Panel2</b>',
    id   : 'panel2',
    frame: true
}

var myWin = new Ext.Window({ // 2
    id     : 'myWin',
    height: 400,
    width : 400,
    items  : [
        panel1,
        panel2
    ]
});
myWin.show();

{1} The first and second child panels
{2} The last child, a form panel

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Let's take a gander at what we're doing in Listing 3.1. The first thing we do is create two vanilla panels **{1}** and create `myWin` **{2}**, an instance of `Ext.Window`, which *contains* the previously defined. The rendered UI should look like the one in Figure 3.1.



Figure 3.12 The rendered container UI from Listing 3.1.

We left some room at the bottom of `myWin`, which will come in handy when we add items. Each container stores references to its children via an *items* property, which can be accessed via `someContainer.items` and is an instance of `Ext.util.MixedCollection`.

The `MixedCollection` is a utility that allows the framework to store and index a *mixed collection* of data, which includes strings, arrays and objects and provides a nice collection of handy utility methods. I like to think of it as the `Array` on steroids.

Now that we've rendered our Container, let's start to exercise the addition of children to a container.

### 3.4.1 Learning to tame children

Normally, taming children involves a lot of yelling and timeouts. Unfortunately, these methods don't really work with Ext component, so we must learn to use the tools that they provide for us. Mastering these utility methods will enable you to dynamically update your UI, which is in the spirit of AJAX web pages.

Adding components is a simple task, in which we're provided two methods; `add` and `insert`. The `add` method only *appends* a child to the container's hierarchy, while `insert` allows you to inject an item into the container at a particular index.

Let's add to the Container that we created in Listing 3.1. For this, we'll use our handy FireBug JavaScript console:

```
Ext.getCmp('myWin').add({
    title : 'Appended Panel',
    id    : 'addedPanel1',
    html   : 'Hello there!'
});
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Running the preceding code adds the item to the Container. Wait a cotton-picking second! Why didn't the new panel appear? Why does the container still look like Figure 3.1? It did not show up because we didn't call the `doLayout` method for the Container. Let's see what happens when we call `doLayout`:

```
Ext.getCmp('myWin').doLayout();
```

Ah-ha! It showed up, but why? Well, the `doLayout` method forces the recalculation of Containers and its children and will render any un-rendered children. The only reason you would not need to call it is if the Container is not rendered. We went down this path of pain so that we may learn the valuable lesson of calling `doLayout` when we add items to a container at run time.

Appending children is handy, but sometimes we need to be able to insert items at a specific index. Using the `insert` method and calling `doLayout` afterwards easily accomplishes this task:

```
Ext.getCmp('myWin').insert(1, {
    title : 'Inserted Panel',
    id    : 'insertedPanel',
    html   : 'It is cool here!'
});
Ext.getCmp('myWin').doLayout();
```

We simply insert a new Panel at index 1, which is right under `Panel1`. Because we called the `doLayout` method immediately after we did an insertion, we see the newly inserted panel in the Window instantaneously. The changes should look like Figure 3.13:



Figure 3.13 The rendered results of our dynamically added and inserted child panels.

As you can see, adding and inserting children is a cinch. Removing items is just as easy as adding them and accepts two arguments. The first of which is a reference to the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

component or the component ID from which you want to be removed. The second parameter, however, specifies whether or not the destroy method should be called for that component, which gives you incredible flexibility, allowing you to move components from one container to another if you so desired. Here is how we would remove one of the child panels that we recently added using our handy Firebug console:

```
var panel = Ext.getCmp('addedPanel');
Ext.getCmp('myWin').remove(panel);
```

After you execute this code, you'll notice that the panel immediately disappeared. This is because we didn't specify the second parameter, which is, by default `true`. You can override this default parameter by setting `autoDestroy` to `false` on a Parent Container. Also, you don't need to call the parent's `doLayout` method, as the removed Component's `destroy` method is called, initiating its destruction phase and deleting its DOM element.

If you wanted to move a child to a different container, you simply specify `false` as the `remove`'s second parameter, and then add or insert it into the parent like this:

```
var panel = Ext.getCmp('insertedPanel');
Ext.getCmp('myWin').remove(panel, false);
Ext.getCmp('otherParent').add(panel);
Ext.getCmp('otherParent').doLayout();
```

The preceding code snippet assumes that we already have another Parent container instantiated with the id of 'otherParent'. We simply create a reference to our previously inserted panel, and perform a non-destructive removal from its parent. Next, we add it to its new parent and call its `doLayout` method to actually perform the DOM-level move operation of the child's element into the new parent's content body element.

The utilities offered by the `Container` class extend beyond the addition and removal of child items. They provide you the ability to descend deep into the Container's hierarchy to search for child components, which become useful if you want to gather a list of child items of a specific type or that meet special criteria and perform an operation on them.

### 3.4.2 Querying the container hierarchy

Of all of the query utility methods, the easiest is `findByType`, which is used to descend into the container hierarchy to find items of a specific XType and returns a list of items that it finds. For instance, if you wanted to find all of the text input fields for a given container:

```
var fields = Ext.getCmp('myForm').findByType('field');
```

Executing the preceding code against a Container with the Component ID of 'myForm' will result in a list of all input fields at any level in its hierarchy. The `findByType` leverages the Container's `findBy` method, which we can use as well. We're going to explore how `findBy` works. Please stay with me on this, as it may not make sense at first.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

The `findBy` method accepts two parameters, a custom method which we use to test for our search criteria and the scope from which to call our custom method. For each of the custom containers, our custom method gets called and is passed the reference of the child Component from which it's being called at. If the custom search criteria are met, the custom method *needs to* return true, which instructs `findBy` to add that recently referenced Component to a list. Once all components are exhausted, `findBy` returns the list, which contains any components that met our criteria.

OK, with that out of the way, let's explore this concept through code. For arguments sake, let's say we wanted to find all child items that are hidden. We could use `findBy` in the following way to do so:

```
var findHidden = function(comp) {
    if (!comp.isVisible()) {
        return true;
    }
}
var panels = Ext.getCmp('myContainer').findBy(findHidden);
```

In our rather simplistic `findHidden` query method, we are testing on if the component is *not* visible. If the component's `isVisible` method returns anything but true, our `findHidden` method returns true. We then call `findBy` on the Container with the ID of 'myContainer' and pass it our custom `findHidden` method with the results being stored in the `panels` reference.

By now, you have the core-knowledge necessary to manage child items. Let's shift focus on to flexing some Ext-UI muscle by exploring some of the commonly used descendants of containers. We'll see how we can use Ext to create a UI using all of the browser's available viewing space.

### 3.4.3 The Viewport Container

The `Viewport` class is the foundation from which all web applications that depend solely on Ext are built by managing 100% of the browser's - you guessed it - viewport or display area. Weighing in at just a tad over 20 lines, this class is extremely lightweight and efficient. Being that it is a direct descendant of the `Container` class, all of the child management and layout usage is available to you. To leverage the `viewport`, you can use the following example code:

```
new Ext.Viewport({
    layout : 'border',
    items : [
        {
            height : 75,
            region : 'north',
            title : 'Does Santa live here?'
        },
        {
            width : 150,
            region : 'west',
            title : 'The west region rules'
        }
    ]
});
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
        },
        {
            region : 'center',
            title  : 'No, this region rules!'
        }
    ]
});
```

The rendered Viewport from the code above utilizes the entire browser's viewport and display three panels organized by the "border layout". If you resize the browser window, you'll notice that the center panel is resized automatically, which demonstrates how the Viewport listens and responds to the browser's window resize event.

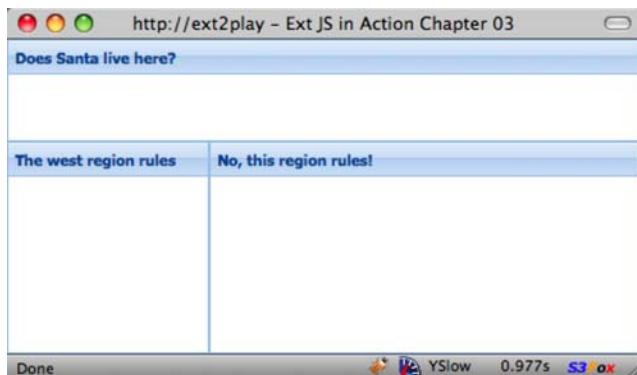


Figure 3.14 Our first Viewport, which takes up 100% of the browser's available viewing space.

The Viewport class provides the foundation for all ext-based applications that leverage the framework as a complete web-based UI solution for their RIAs.

Many developers run into a brick wall when they attempt to create more than one Viewport in a fully Managed Ext JS page to display more than one "screen". To get around this, you can use the card layout with the viewport and "flip" through different application screens, which are resized to "fit" the viewport. We'll dive into layouts in chapter 5, where you'll get to understand key terms like "fit" and "flip" in the context of layouts.

We've covered quite a few important core topics, which help us manage child items in a Container. We also learned how to use the Viewport to help us manage all of the browsers viewing space to layout out Ext UI widgets.

One of the most commonly used UI widgets to display content is the Panel. Let's explore this ubiquitous widget.

### 3.5 Summary

In this chapter we took a deep look at three fundamental areas of Ext JS. In doing so, we learned about managing DOM-level, where we exercised the registration of event listeners for DOM elements and learned how to stop them from bubbling up the DOM hierarchy. We also glanced at component-level events and learned how to register and fire that event.

We took a real in-depth look at the Component Models, which gives the framework a unified method of managing instances of components. The component lifecycle is one of the most important concepts for the UI portion of the framework, which is why we covered it before we went too deep into learning about the many widgets.

Lastly, we explored the world of Containers and learned how they are used to manage child components. In doing so, we also learned about the Viewport and how it is the foundation for web applications that are based entirely on Ext JS.

We now have the foundation that will help propel us forward, as we'll start to exercise the framework's UI machinery.

# 04

## *Panels, TabPanels, and Windows*

When developers start to experiment or build applications with Ext, they often start by copying examples from the downloadable SDK. While this approach is good for learning how a particular layout was accomplished, it falls short in explaining how the stuff actually works, which leads to those throbbing forehead arteries. In this chapter, we'll explain some of the core topics, which are some of the building blocks to developing a successful UI deployment.

In this chapter, we'll going to dive into how the Panel works, and explore the areas where it can display content and UI widgets. We'll then explore Windows and the MessageBox, which float above all other content in the page. Towards the end, we'll dive into using Tab Panels, and explore some of the usability issues that may occur when using this widget.

Upon completion of this chapter, you will have the ability to manage the full CRUD (CReate, Update and Delete) lifecycle for Containers and their child items, which you will depend on as you develop your applications.

### **4.1 The Panel**

The Panel, a direct descendant of Container, is considered another workhorse of the framework as it what many developers use to present your UI widgets. A fully loaded panel is divided into six areas for content as seen in figure 4.1. Recall that Panel is also a descendant of Component, which means that it follows the Component lifecycle. Moving forward, we will use the term *Container* to describe any descendent of Container. This is because I want to reinforce the notion that the UI widget in context is a descendant of Container.

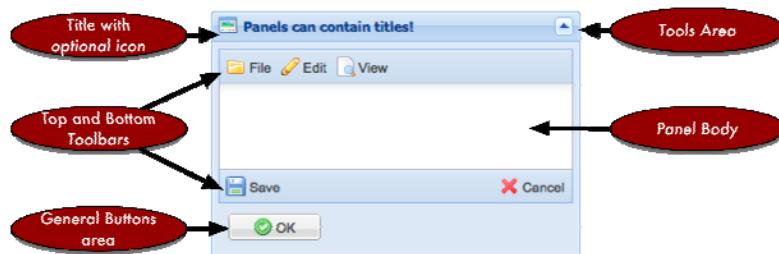


Figure 4.1 An example of a fully loaded Panel, which has a title bar with an Icon and tools, top and bottom toolbars, and a button bar on the bottom.

The title bar is a very busy place for a panel that offers both visual and interactive content for the end user. Like Microsoft Windows, an icon can be placed at the top left of the panel, offering your users a visual queue as to what type of panel they are focused on. In addition to the Icon, a title can be displayed on the panel as well.

On the right most area of the title bar is a section for 'tools', which is where miniature icons can be displayed, which will invoke a handler when clicked. Ext provides many icons for tools, which include many common user related functions like help, print and save. To view all of the available tools, visit the Panel API.

Of the six content areas, the Panel body is arguably the most important, which is where the main content or child items are housed. As dictated by the Container class, a layout must be specified upon instantiation. If a layout is not specified, the Container layout is used by default. One important attribute about layouts is that they cannot be swapped for another layout dynamically.

Let's build a complex panel with a top and bottom toolbars, with two buttons each.

#### 4.1.1 Building a complex panel

Being that we're going to have toolbar buttons, we should have a method to be called when they are clicked.

```
var myBtnHandler = function(btn) {
    Ext.MessageBox.alert('You Clicked', btn.text);
}
```

This method will be called when a button on any toolbar is clicked. The toolbar buttons will call handlers passing themselves as a reference, which is what we call `btn`. Next, let's define our toolbars:

##### **Listing 4.1 Building toolbars for use in a Panel**

```
var myBtnHandler = function(btn) { // 1
    Ext.MessageBox.alert('You Clicked', btn.text);
}
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var fileBtn = new Ext.Button({ // 2
    text : 'File',
    handler : myBtnHandler
});

var editBtn = new Ext.Button({ // 3
    text : 'Edit',
    handler : myBtnHandler
});

var tbFill = new Ext.Toolbar.Fill(); // 4

var myTopToolbar = new Ext.Toolbar({ // 5
    items : [
        fileBtn,
        tbFill,
        editBtn
    ]
});

var myBottomToolbar = [ // 6
{
    text : 'Save',
    handler : myBtnHandler
},
{
    text : 'Cancel',
    handler : myBtnHandler
},
'->',
'<b>Items open: 1</b>',
];
{Annotation 1} The button click handler method
{Annotation 2} The 'File' button
{Annotation 3} The 'Edit' button
{Annotation 4} The "greedy" toolbar fill,
{Annotation 5} The top toolbar instantiation
{Annotation 6} The bottom toolbar array configuration.

```

In the preceding code example, we do quite a lot and display two different ways of defining a toolbar and its child components. Firstly, we define myBtnHandler **{#1}**. By default, each button's handler is called with two arguments, the button itself, and the browser event wrapped in an Ext.Event object. We use the passed button reference ("btn") and pass that text on over to Ext.MessageBox.alert to provide the visual confirmation that a button was clicked.

Next, we instantiate the File **{#2}**, Edit **{#3}** buttons and the "greedy" toolbar spacer **{#4}**, which will push all toolbar items after it to the right. We assign myTopToolbar to a new instance of Ext.Toolbar **{#5}**, referencing the previously created buttons and spacer as elements in the new toolbar's items array.

That was a lot of work for just a relatively simple toolbar. We did it this way to "feel the pain" of doing things the "old way" and better appreciate how much time (and end developer code) the Ext shortcuts and XTypes really save. The myBottomToolbar **{#6}** reference is a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

simple array of objects and strings, which Ext translates into the appropriate objects when its parent container deems it necessary to do so. To get references to the top toolbar, you can use `myPanel.getTopToolbar()` and inversely, to get a reference to the bottom; `myPanel.getBottomToolbar()`. You would use these two methods to add and remove items dynamically to either toolbar. We'll cover toolbars in much greater detail later. Next, let's create our panel body:

```
var myPanel = new Ext.Panel({
    width      : 200,
    height     : 150,
    title      : 'Ext Panels rock!',
    collapsible: true,
    renderTo   : Ext.getBody(),
    tbar       : myTopToolbar,
    bbar       : myBottomToolbar,
    html       : 'My first Toolbar Panel!'
});
```

We've created panels before, so just about everything here should look familiar except for the `tbar` and `bbar` properties, which reference the newly created toolbars. Also, there is a `collapsible` attribute, which when set to `true`, the panel creates a toggle button on the top right of the title bar. Rendered, the panel should look like the one in figure 4.2. Remember, clicking on any of the toolbar buttons will result in an `Ext.MessageBox` displaying the button's text, giving you visual confirmation that the click handler was called.

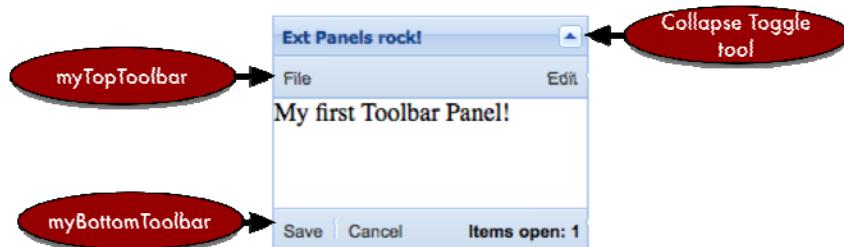


Figure 4.2 The rendered results of Listing 4.1, where we create a complex-collapsible panel with a top and bottom toolbar that have each contain buttons.

Toolbars are great places to put content, buttons or menus that are outside of the Panel body. There are two areas from which we still need to explore, buttons and tools. To do this, we'll add to the `myPanel` example, but we're going to do it using the Ext shortcuts with XTypes inline with all of the other configuration options.

#### **Listing 4.2 Adding buttons and tools to our existing panel**

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var myPanel = new Ext.Panel({
    ...
    buttons : [
        {
            text : 'Press me!',
            handler : myBtnHandler
        }
    ],
    tools : [
        {
            id : 'gear',
            handler : function(evt, toolEl, panel) {
                var toolClassNames = toolEl.dom.className.split(' ');
                var toolClass = toolClassNames[1];
                var toolId = toolClass.split('-')[2];

                Ext.MessageBox.alert('You Clicked', 'Tool ' + toolId);
            }
        },
        {
            id : 'help',
            handler : function() {
                Ext.MessageBox.alert('You Clicked', 'The help tool');
            }
        }
    ]
});

{1} Original properties from the previous example
{2} The buttons Array begins
{3} The 'Press me!' panel button
{4} The tools array begins
{5} The 'gear' tool and inline click handler
{6} The 'help' tool and inline click handler

```

In Listing 4.2, we added to the previous set of config options **{#1}**, and included two shortcut arrays, one for buttons and the other for tools. Because we specified a buttons array **{#2}**, when the Panel renders, it will create a footer div, a new instance of Ext.Toolbar with a special CSS class 'x-panel-fbar' and render it to the newly created footer div. The 'Press Me!' button **{#3}** will be rendered in the newly created footer toolbar, and when clicked will invoke our previously defined myBtnHandler method.

If you look at the myBottomToolbar shortcut array in Listing 4.1 and the buttons shortcut array in Listing 4.3 you'll see some similarities. This is because all of the panel toolbars (tbar, bbar and buttons) can be defined using the exact same shortcut syntax because they all will get translated into instances of Ext.Toolbar and rendered to their appropriate position in the panel.

We also specified a tools array **{#4}** configuration object, which is somewhat different than the way we define the toolbars. Here, to set the icon for the tool, you must specify the id of the tool, such as 'gear' **{#5}** or 'help' **{#6}**. For every tool that is specified in the array, an icon will be created in the tools. Panel will assign a 'click' event handler to each tool, which will invoke the handler specified in that tools' configuration object. The rendered version of the newly modified myPanel should look like the one in figure 4.3.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

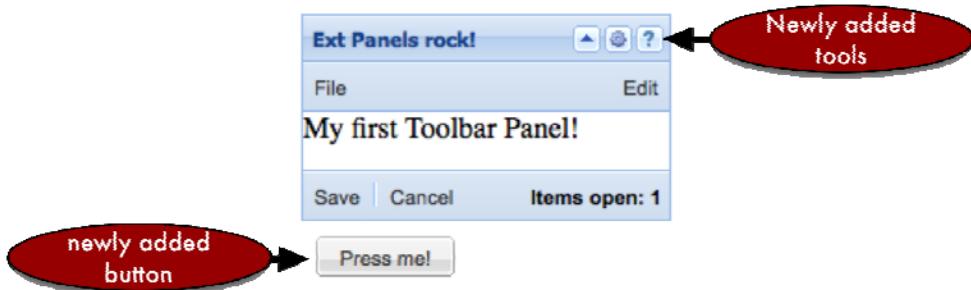


Figure 4.3 The rendered results from Listing 4.2, which add a button in the button bar as well as tools to the title bar.

The preceding example is meant to display all of the items that can be utilized on a panel, but is not the best example of elegant and efficient User interface design. While it may be tempting to load up your panels with buttons and toolbars, you must be careful not to overload a panel with too much on-screen "gadgetry", which could overwhelm your user and take up valuable screen real-estate.

Now that we have some experience with the Panel class, let's look at one of its close descendants, the Window, which you can use to float content above everything else on the screen and can be used to replace the traditionally lame browser-based popup.

## 4.2 Popping up Windows

The Window UI widget builds upon the Panel, providing you the ability to float UI components above all of the other content on the page. With Windows, you can provide a modal dialog, which masks the entire page, forcing the user to focus on the dialog and prevents any mouse-based interaction with anything else on the page. Figure 4.4 is a perfect example of how we can leverage this class to focus the user's attention and request input.

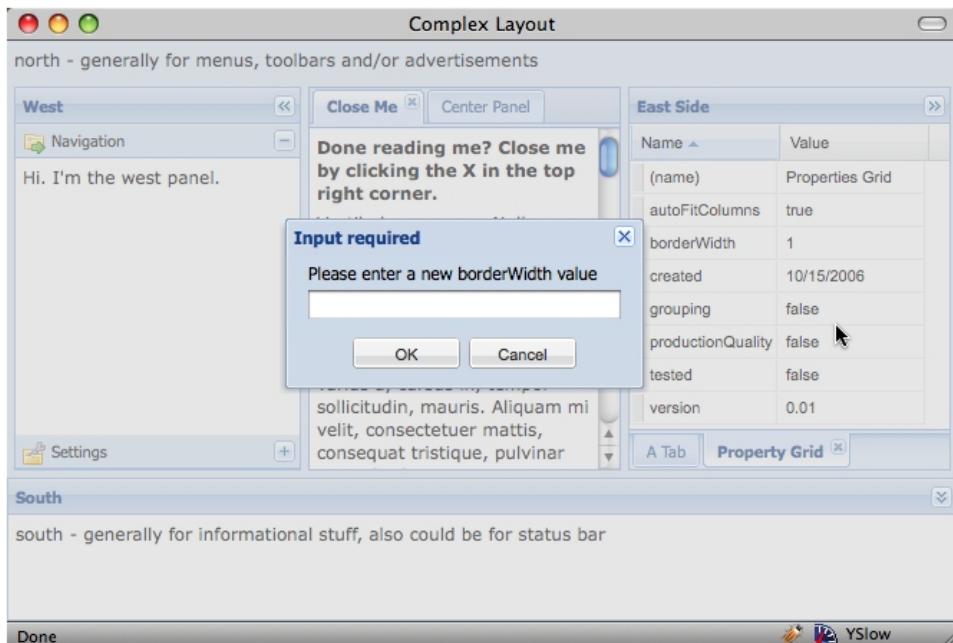


Figure 4.4 An Ext Modal Window, which masks the browser's viewport.

Working with the Windows class is a lot like working with the Panel class, except you have to consider issues like whether or not you want to disable resizing or want the Window to be constrained within the boundaries of the browser's viewport. Let's look into how we can build a window. For this, we'll need a vanilla Ext Page with no widgets loaded.

### **Listing 4.3 Building an animated window**

```

var win;
var newWindow = function(btn) {
    if (!win) {
        win = new Ext.Window({
            animateTarget : btn_el,
            html          : 'My first vanilla Window',
            closeAction   : 'hide',
            id            : 'myWin',
            height        : 200,
            width         : 300,
            constrain     : true
        });
    }
    win.show();
}
new Ext.Button({
    renderTo : Ext.getBody(),
    // 5
}

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
text      : 'Open my Window',
style     : 'margin: 100px',
handler   : newWindow
}) ;

{1} A handler that creates a new window
{2} Instantiate a new Window
{3} Prevent the automatic destruction upon close
{4} Constrain the window to the browser's viewport
{5} Create a button that launches the window
```

In listing 4.3, we do things a little differently in order for us to see the animation for our Window's close and hide method calls. The first thing we do is create a global variable, win, for which we'll reference the soon to be created window. We create a method, newWindow{1} that will be the handler for our future button and is responsible for creating the new Window{2}.

Lets take a quick moment examine some of the configuration options for our Window. One of the ways we can instruct the Window to animate upon show and hide method calls is to specify an animateEl property, which is a reference to some element in the DOM or the element ID. If you don't specify the element in the configuration options, you can specify it when you call the show or hide methods, which take the exact same arguments. In this case, we're the launching button's element. Another important configuration option is closeAction{3}, which defaults to 'close' and destroys the Window when the close tool is clicked. We don't want that in this instance, so we set it to 'hide', which instruct the close tool to call the hide method instead of close. We also set the constrain{4} parameter to true, which instructs the Window's drag and drop handlers to prevent the window from being moved from outside of the browser's viewport.

Lastly, we create a button{5} that, when clicked, will call our newWindow method, resulting in the window animating from the button's element. Clicking on the (x) close tool will result in the window hiding. The rendered results will look like figure 4.5.



Figure 4.5 The rendered results from Listing 4.3, where we create a window that animates from the button's element upon click.

Because we don't destroy the window when the close tool is pressed, you can show and hide the window as many times as you wish, which is ideal for windows that you plan on reusing.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Whenever you deem that it is necessary to destroy the window, you can call its destroy or close method. Now that we have experienced in creating a reusable Window, we can now begin exploring other configuration options to further alter the behavior of the Window.

#### 4.2.1 Further Window configuration exploration

There are times when we need to make a Window behave to meet requirements of our application. In this section, we'll learn about some of the commonly used configuration options.

Some time we need to produce a window that is modal and is very rigid. To do this, we need to set a few configuration options.

##### **Listing 4.4 Creating a rigid modal Window**

```
var win = new Ext.Window({
    height      : 75,
    width       : 200,
    modal       : true,                                     // 1
    title       : 'This is one rigid window',
    html        : 'Try to move or resize me. I dare you.',
    plain       : true,
    border      : false,
    resizable   : false,                                    // 2
    draggable   : false,                                   // 3
    closable    : false,                                   // 4
    buttonAlign : 'center',
    buttons     : [
        {
            text   : 'I give up!',
            handler: function() {
                win.close();
            }
        }
    ]
})
win.show();
{1} Ensuring the page is masked
{2} prevent resizing from occurring
{3} disabling window movement
{4} preventing window closure
```

In Listing 4.4, we create an extremely strict modal Window. To do this, we had to set quite a few options. The first of which is modal`{1}`, which instructs the window to mask the rest of the page with a transparent div. Next, we set resizable`{2}` to false, which prevents the window from being resized via mouse actions. To prevent the window from being moved around the page, we set draggable`{3}` to false. I only wanted a single center button to close the window, so closable`{4}` is set to false, which hides the close tool. Lastly, set some cosmetic parameters, plain, border and buttonAlign. Setting plain to true will make the content body background transparent. When coupled with setting the border to false, the window appears to be one unified cell. Being that I wanted to have the single button

centered, we specify the `buttonAlign` property as such. The rendered example should look like Figure 4.5.

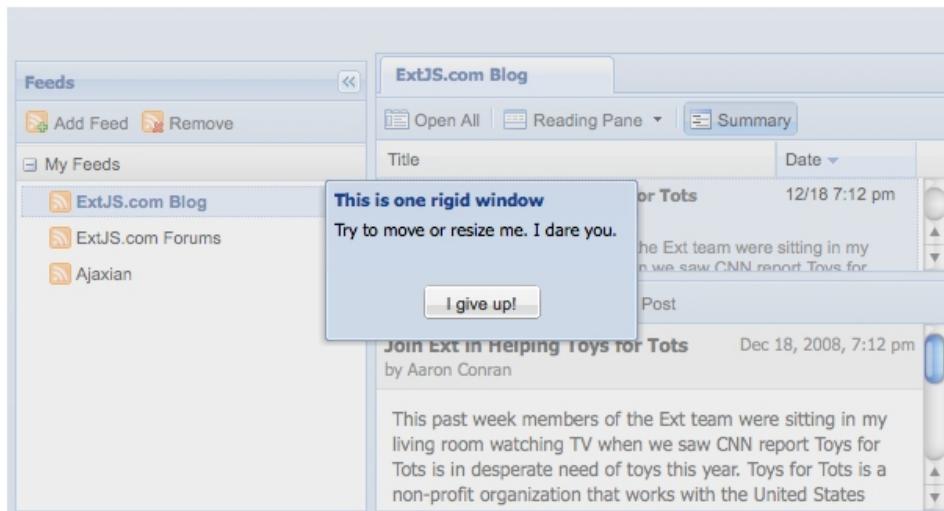


Figure 4.5 Our first strict modal Window rendered in the Ext SDK feed viewer example.

There are other times when we want to relax the restrictions on the window. For instance, there are situations where we need a window to be `resizable`, but should not be resized less than specific dimensions. For this, you allow `resize` (`resizable`) and specify `minWidth` and `minHeight` parameters. Unfortunately, there is no easy way to set boundaries to how large a window can grow.

While there are many reasons to create our own windows, there are times where we need something quick and dirty to, for instance, display a message, or prompt for user data. The `Window` class has a stepchild known as the `MessageBox` to fill this need.

#### 4.2.2 Replacing alert and prompt with MessageBox

The `MessageBox` class is a reusable, yet versatile, Singleton class that gives us the ability to replace some of the common browser based dialogs such as `alert` and `prompt` with a simple method call. The biggest thing to know about the `MessageBox` class is that it *does not* stop JavaScript execution like traditional alerts or prompts, which I consider an advantage. While the user is digesting or entering information, your code can perform AJAX queries or even manipulate the UI. If specified, the `MessageBox` will execute a callback method when the Window is dismissed.

Before we start to use the `MessageBox` class, let's create a callback method. We'll need this later on.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
var myCallback = function(btn, text) {
    console.info('You pressed ' + btn);
    if (text) {
        console.info('You entered : ' + text)
    }
}
```

Our `myCallback` method will leverage Firebug's console to echo out the button pressed and the text you entered if any. The `MessageBox` will only pass two parameters to the callback method, the button ID and any entered text. Now that we have our callback, lets launch an alert dialog.

```
var msg = 'Your document was saved successfully';
var title = 'Save status:';
Ext.MessageBox.alert(title, msg);
```

Here, we call the `MessageBox.alert` method, which will generate a window, which will look like Figure 4.9 (left) and will dismiss when the OK button is pressed. If you wanted `myCallback` to get executed upon dismissal, add it as the third parameter. Now that we have looked at alerts, lets see how we can request user input with the `MessageBox.prompt` method.

```
var msg = 'Please enter your email address.';
var title = 'Input Required';
Ext.MessageBox.prompt(title, msg, myCallback);
```

We call the `MessageBox.prompt` method, which we pass the reference of our callback method and will look like Figure 4.6. Enter some text and press the Cancel button. In the FireBug console, you'll see the button ID pressed and the text entered.



Figure 4.6 The `MessageBox`'s alert (left) and prompt (right) modal dialog Windows.

And there you have it, alert and prompt at a glance. I find these very handy, as I don't have to create my own singleton to provide these UI widgets. Be sure to remember them when you're looking to implement a Window class to meet a requirement.

I have to confess a little secret. Ready? The alert and prompt methods are actually shortcut methods for the much larger and highly configurable `MessageBox.show` method. Here is an example of how we can use the `show` method to display an icon with a multi-line textarea input box.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

### 4.2.3 Advanced MessageBox techniques

The MessageBox.show method provides an interface to display the MessageBox using any combination of the 24 available options. Unlike the previously explored shortcut methods, show accepts the typical configuration object as a parameter. Let's display a multi-line textarea input box along with an icon.

```
Ext.Msg.show({
    title      : 'Input required:',
    msg        : 'Please tell us a little about yourself',
    width      : 300,
    buttons    : Ext.MessageBox.OKCANCEL,
    multiline  : true,
    fn         : myCallback,
    icon       : Ext.MessageBox.INFO
});
```

When the preceding example is rendered, it will display a modal dialog like the one in Figure 4.7 (left). Next, let's see how we create an alert box that contains an icon and three buttons.

```
Ext.Msg.show({
    title      : 'Hold on there cowboy!',
    msg        : 'Are you sure you want to reboot the internet?',
    width      : 300,
    buttons    : Ext.MessageBox.YESNOCANCEL,
    fn         : myCallback,
    icon       : Ext.MessageBox.ERROR
})
```

The preceding code example will display our tri-button modal alert dialog window, like in Figure 4.7 (right).



Figure 4.7 A multi-line input box with an icon (left) and a tri-button icon alert box (right).

While everything in our two custom MessageBox examples should be self-explanatory, I think it's important to highlight some two of the configuration options, which we pass references to MessageBox public properties.

The buttons parameter is used as a guide for the Singleton to know which buttons to display. While we pass a reference to an already existing property,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Ext.MessageBox.OKCANCEL, you can display no buttons by setting buttons to an empty Object, such as “{}”. Else, you can customize which buttons you want to display. To display yes and cancel, pass “{ yes : true, cancel : true}” and so on. The Singleton already has a set of predefined popular combinations, which are CANCEL, OK, OKCANCEL, YESNO and YESNOCANCEL.

The icon parameter works in the same way as the button parameter, except it is a reference to a string. The MessageBox class has three predefined values, which are INFO, QUESTION and WARNING. These are references to strings that are CSS classes. If you wish to display your own icon, create your own CSS class and pass the name of your custom CSS class as the icon property. Here is an example of a custom CSS class.

```
.icon-add {
    background-image: url(/path/to/add.png) !important;
}
```

Now that we have our feet wet with some advanced MessageBox techniques, we can now explore how we can leverage the MessageBox to display an animated dialog, which you can use to offer the user live and updated information regarding on a particular process.

#### 4.2.4 Showing an animated wait MessageBox

When we need to stop a particular workflow, we need to display some sort of modal dialog box, which can be as simple and *boring* as a modal dialog with a ‘please wait’ message. I prefer to introduce some spice into the application and provide an animated “wait” dialog box. With the MessageBox class, we can create a seemingly effortless and infinitely looping progress bar:

```
Ext.MessageBox.wait("We're doing something...", 'Hold on...');
```

Which will produce a wait box like in Figure 4.11. If the syntax seems a little strange, it is because the first parameter is the message body text, with the second parameter being the title. It’s exactly opposite of the alert or prompt calls. Lets say you wanted to display text in the body of the animating progress bar itself, you could pass a third parameter with a single text property, such as {text: ‘loading your items’}. Figure 4.11 also shows what it would be like if we added progress bar text to our dummy wait dialog. If the syntax



Figure 4.8 Two simple animate MessageBox wait dialog where the progress bar is looping infinitely at a predetermined fixed interval(left) and a similar message box with text in the progress bar (right).

While this may seem cool at first, it's not very interactive as the text is static and we're not controlling the progress bar status. We can customize the wait dialog box by using the handy show method and passing in some parameters. Using this method, we now have the leeway to update the progress bar's advancement as we see fit. In order to create an auto-updating wait box, we need to create a rather involved loop, so please stay with me on this.

#### **Listing 4.5 Building a dynamically updating progressbar**

```
Ext.MessageBox.show({
    title      : 'Hold on there cowboy!',
    msg        : "We're doing something...",
    progressText: 'Initializing...!',
    width      : 300,
    progress    : true,                                // 1
    closable   : false
});

var updateFn = function(num){
    return function(){
        if(num == 6){                                // 2
            Ext.MessageBox.updateProgress(100, 'All Items saved!'); // 3
            Ext.MessageBox.hide.defer(1500, Ext.MessageBox);          // 4
        }
        else{
            var i = num/6;
            var pct = Math.round(100 * i);
            Ext.MessageBox.updateProgress(i, pct + '% completed'); // 5
        }
    };
};

for (var i = 1; i < 7; i++){                         // 6
    setTimeout(updateFn(i), i * 500);
}

{1} Instructing MessageBox to show a progress bar
{2} An updater method to update our message box's progress text
{3} Updating the progress bar's percentage and text
{4} Defering the execution of the MessageBox's dismissal
{5} Updating the progress if we have not hit our limit
{6} A looping time half second timeout
```

In Listing 4.5, we show a MessageBox, with the option of progress`{1}` set to true, which will show our progress bar. Next, we define a rather involving updater function, aptly named `updateFn{2}` which is what is called at a predefined interval. In that function, if the number passed equal to our limit of 6, we update the progress bar 100% width and the completion text`{3}`. We also defer the dismissal of the message box by one and a half seconds`{4}`. Else, if we will calculate a percentage completed and update the progress bar width and text accordingly`{5}`. Lastly, we create our loop that calls `setTimeout{6}` six consecutive times, which delays our calls of `updateFn` by the iteration times one half second. The results of this rather lengthy example will look like Figure 4.9, below.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>



Figure 4.9 Our automatically updating wait MessageBox (left) with the final update(right) before automatic dismissal.

And there you have it, with some effort; we can dynamically update our users with a status of operations that are taking place before they can move further.

In this section, we learned how to create both flexible and extremely rigid Windows to get the user's attention. We also explored a few different ways of using one of Ext's super singletons, the Ext MessageBox class. Let's now shift focus to the Tab Panel class, which provides a means to allow a UI to contain many screens but only display them one at a time.

### 4.3 Components can live in Tab Panels too

The Ext.TabPanel class builds upon Panel to add to create a robust Tabbed Interface., which gives the user the ability to select any screen or UI control associated with a particular tab. Tabs within the tab panel can be un-closable, closable, disabled and even hidden as illustrated in Figure 4.10.

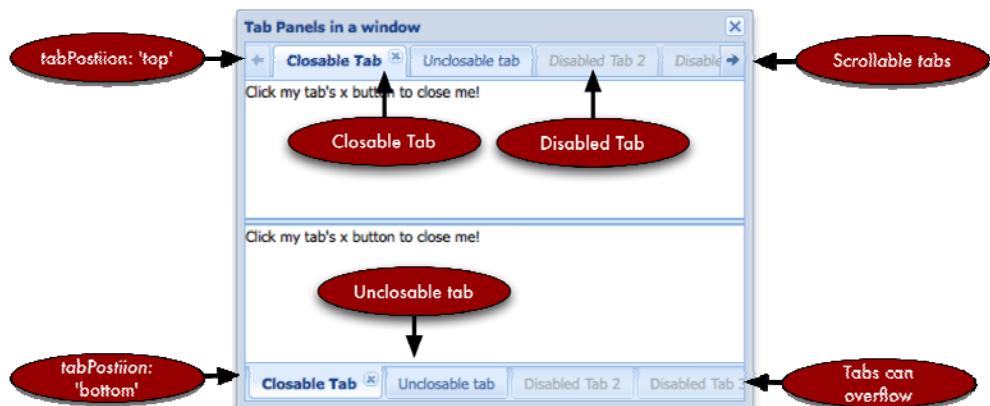


Figure 4.10 Exploring top and bottom positioned tabs.

Unlike other tab interfaces, the Ext Tab Panel only supports a top or bottom tab strip configuration. This is mainly due to the fact that most modern browsers do not support CSS version 4.0, where vertical text is possible. While configuring a Tab Panel may seem straight forward by looking at the API, there are two, that if not understood could cause office bleeping to occur.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

### 4.3.1 Remember these two options

It is amazing that just two of these options could cause developers pain and fill up the office expletive jars, providing free lunch or coffee for fellow team members. Exploring these may help keep those jars empty, and your wallets fuller. Before we build our Tab Panel, I think it's important to lay these out first, so we can get on to the fun!

One of the reasons that the Card Layout is so darn fast is because it makes use of a common technique called lazy or deferred rendering for its child components. This is controlled by the `deferredRender` parameter, which is set true by default. Deferred render means that only cards that get activated are actually rendered. It is fairly common that tab panels have multiple children that have complex UI controls, such as the one in Figure 4.14, which can require a significant amount of CPU time to render. Deferring the render of each child until it is activated accelerates the tab panel's initial rendering and gives the user a better responding widget.

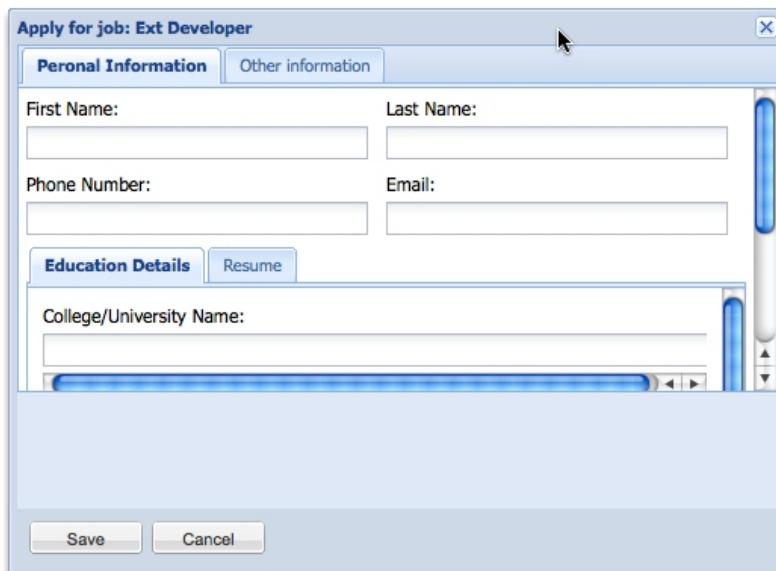
The screenshot shows a window titled "Apply for job: Ext Developer". It contains a Tab Panel with two tabs: "Personal Information" (selected) and "Other information". Under "Personal Information", there are fields for "First Name" and "Last Name" (both empty). Below them are fields for "Phone Number" and "Email" (both empty). At the bottom of this section are tabs for "Education Details" and "Resume". The "Education Details" tab is selected, showing fields for "College/University Name" (empty), "Address" (empty), and "City" (empty). At the very bottom of the window are "Save" and "Cancel" buttons.

Figure 4.11 A Tab Panel with children that have complex layout.

There is one major disadvantage to allowing `deferredRender` to be true, and it has to do with the way form panels work. The form's `setValues` method does not apply values across any of the un-rendered member fields, which means that if you plan on populating forms with tabs, be sure to set `deferredRender` to `false`, otherwise you might be adding to the jar!

Another configuration option that fellow developers often overlook is `layoutOnTabChange`, which forces a `doLayout` method call on child items when that tab is activated. This is

important because on deeply nested layouts, sometimes the parent's resize event may not cascade down properly, forcing a recalculation on child items that are supposed to conform to the parent's content body. If your UI starts to look funky, like the one in Figure 4.13, I suggest setting this configuration option to true and the issue will be solved.



4.12 A child panel whose layout has not been properly recalculated after a parent's resize.

I would only suggest setting `layoutOnTabChange` to true only when you have problems however. The `doLayout` method forces calculations that in extremely nested layouts, can require a considerable amount of CPU time, causing your web app to jitter or stutter. Now that we've covered some of the Tab Panel basics, let's move on to build our first tab panel.

### 4.3.2 Building our first Tab Panel

The Tab Panel is a direct descendant of Panel and makes clever use of the Card Layout. Tab Panel's main job is managing tabs in the tab strip. This is because child management is performed by the Container class and the layout management is performed by the Card Layout. Let's build out our very first Tab Panel.

#### Listing 4.6 Exploring a Tab Panel for the first time.

```
var disabledTab = {  
    title : 'Disabled tab',  
    id : 'disabledTab',  
    html : 'Peekaboo!',  
    disabled : true,  
} // 1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        closable : true
    }

var closableTab = { // 2
    title : 'I am closable',
    html : 'Please close when done reading.',
    closable : true
}

var disabledTab = { // 3
    title : 'Disabled tab',
    id : 'disabledTab',
    html : 'Peekaboo!',
    disabled : true,
    closable : true
}

var tabPanel = new Ext.TabPanel({ // 4
    activeTab : 0,
    id : 'myTPanel',
    enableTabScroll : true,
    items : [
        simpleTab,
        closableTab,
        disabledTab,
    ]
});

new Ext.Window({ // 5
    height : 300,
    width : 400,
    layout : 'fit',
    items : tabPanel
}).show();

{1} A simple, static tab
{2} A simple, closable tab
{3} A closable, yet disabled tab
{4} Our actual Tab Panel
{5} The container for our tab panel.

```

While we could have defined all of the items in the above code in a single large object, I thought it would be best to break it up so things are readily apparent. The first three variables define our Tab Panel's children in generic object form, with the assumption that the defaultType (XType) for the TabPanel class is 'panel'. The first child is a simple and non-closable tab**{1}**. One thing to note here is that all tabs are non-closable by default. This is why our second tab**{2}**, has closable set to true. Next, we have a closable *and* disabled tab.

We then go on to instantiate our TabPanel**{3}**. We set the activeTab parameter to 0. We do this because we want the very first tab to be activated after the Tab Panel is rendered. You can specify any index number in the tab panel's items mixed collection. Because the mixed collection is an array, the first item always starts with 0. We also set enableTabScroll to true, which instructs the TabPanel class to scroll our tabstrip *if* the sum of the tab widths exceeds that of the viewable tab strip. Lastly, our Tab Panel's items array has our three tabs specified.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Next, we create a Container for our TabPanel, an instance of Ext.Window. We specify a fit layout for the window and set the tabPanel reference as its single item. The rendered code should look like the one in Figure 4.13 below.

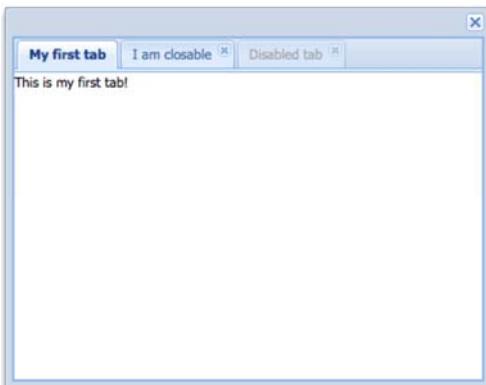


Figure 4.13 Our first TabPanel rendered inside of a Window.

Now that we have our first TabPanel rendered, we can start to have fun with it. I'm sure that you've probably closed "I am closable" tab, which is OK. If you have not done so, feel free to explore the rendered UI control and close out the only closable tab when you are comfortable doing so, which will leave only two tabs available, "My first tab" and "Disabled tab".

### 4.3.3 Tab management methods you should know

Because the TabPanel class is a descendant of Container, all of the common child management methods are available to utilize. These include add, remove and insert. There are a few methods, however, that you will need to know in order to take full advantage of the TabPanel.

The first of which is setActiveTab, which activate a tab, as if the user selected the item on the tab strip and accepts either the index of the tab, or the actual component ID.

```
var tPanel = Ext.getCmp('myTPanel');

tPanel.add({
    title : 'New tab',
    id   : 'myNewTab'
});

tPanel.setActiveTab('myNewTab');
```

Executing the prior code will result in a new tab with the title of "New closable tab", which gets activated automatically. Calling setActiveTab after an add operation is akin to calling

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

doLayout on a generic container. We also have the capability to enable and disable tabs at runtime, but requires a different approach than simply calling a method on the TabPanel

The TabPanel does not have enable or disable methods, so in order to enable or disable a child, we need to call those methods of the child items themselves. We can leverage Listing 4.1 to enable our disabled tab.

```
Ext.getCmp('disabledTab').enable();
```

Yes, that's all there is to it. The tab strip item (tab UI control) now reflects that the item is no longer disabled. This happens because the TabPanel actually subscribes to the child item's, you guessed it, enable and disable events to manage the associated tab strip items.

In addition to enabling and disabling tabs, you can also hide them. To hide a tab, however, the TabPanel does have a utility method, which is hideTabStripItem. This method accepts a single parameter, but three possible data values, which are the tab index number, tab component ID or a reference to the actual component instance itself. In our case, we'll use the ID since that's a known.

```
Ext.getCmp('myTPanel').hideTabStripItem('disabledTab');
```

And the inverse of which is unhideTabStripItem:

```
Ext.getCmp('myTPanel').unhideTabStripItem('disabledTab');
```

There you have it, managing tab items. While there are many advantages to using the TabPanel in your web application we should explore some of the usability problems that you may encounter. After all, we need to keep that swear jar as empty as possible.

#### 4.3.4 Working with caveats and drawbacks

While the TabPanel opened new doors for UI control, it does have some limitations that we should explore. Two of which are related to the size of the tabs and the width of the bounding area for width the TabPanel is being displayed. If the sum of the width of the tabs is greater than the viewport, the tabs can either be pushed off screen. This can happen by the tab widths being too large or by the total number of tabs exceeding the allowable viewing space. When this issue occurs, the usability of the TDI is somewhat reduced.

To offer some relief of these shortcomings, the TabPanel can be configured to resize tabs automatically or even scroll them if they go beyond the viewport. While these are good at easing the problem, they do not solve them.

In order to fully understand these issues, we should explore them as best as possible. Let's start out with a TabPanel inside of a Viewport.

#### Listing 4.7 Exploring scrollable tabs

```
Ext.QuickTips.init();
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

new Ext.Viewport({
    layout : 'fit',
    title  : 'Exercising scrollable tabs',
    items  : [
        {
            xtype      : 'tabpanel',
            activeTab : 0,
            id        : 'myTPanel',
            enableTabScroll : true,
            items     : [
                {
                    title : 'our first tab'
                }
            ]
        }
    );
}); // 1

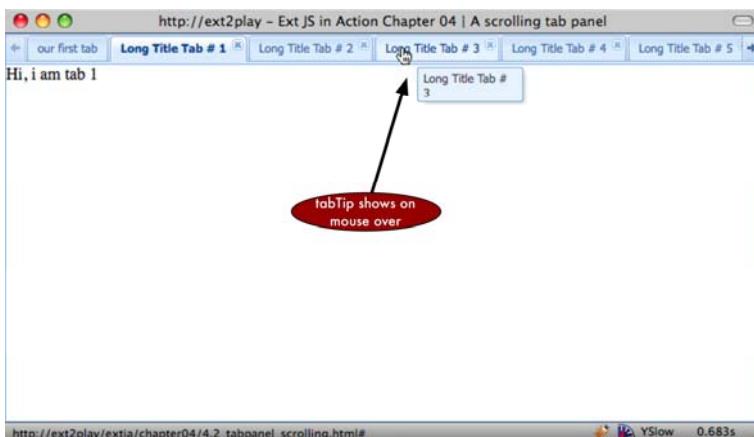
(function (num) {
    for (var i = 1; i <= 30; i++) {
        var title = 'Long Title Tab #' + i;
        Ext.getCmp('myTPanel').add({
            title   : title,
            html    : 'Hi, i am tab ' + i,
            tabTip : title,
            closable : true
        });
    }
}).defer(500); // 2

```

**{1} An Ext Viewport with an embedded TabPanel**

**{2} An deferred anonymous function execution**

In Listing 4.7, we create a viewport with our TabPanel{1} which contains a single child. Next, we create an anonymous function{2} and defer its execution by  $\frac{1}{2}$  seconds. We specify 20 as the number of dynamic tabs to create dynamically in the for loop. For each new tab that we create, we include the tab number in the tab title, html and tabTip. The rendered code should look like Figure 4.14.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Figure 4.14 A TabPanel with a scrolling tab strip, which includes mouse over tooltips for the dynamic tabs.

Now that we have our TabPanel rendered, scroll over to find “Long Title Tab 14”. Took a while - huh? Even with an extra-wide display, the tabs will still scroll. One way to remedy this situation is to perhaps set a minimum tab width. Let’s modify our example by adding the following configuration parameters to the TabPanel XType configuration:

```
resizeTabs : true,
minTabWidth : 75,
```

Refreshing the newly modified TabPanel in Figure 4.15 (bottom) results in tabs that are either unusable or hard to use. Specifying `resizeTabs` as true instructs the TabPanel to reduce the width of a tab as much as it needs to in order to display the tabs without scrolling. Auto sizing a tab works if the tab title does not get truncated or hidden. This is where the diminishing usability of TabPanels becomes apparent. If the tab title is not completely visible, the user must activate each tab in order to find the correct one. Else, if the tab tooltips are enabled, the user must over each tab in order to locate the one they wish to activate. As you can see, the tooltips can enhance the speed of the tab search but does not remedy the issue completely.

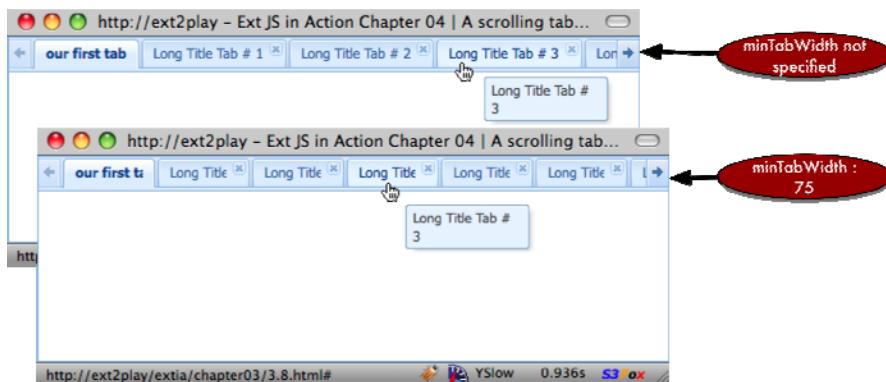


Figure 4.15 A TabPanel that has no minimum tab width (top) specified and a TabPanel (bottom) that has a minimum tab width of 75 specified.

No matter which route you choose for implementing the TabPanel, always keep in mind that too many tabs could lead to trouble by either reducing usability or even reducing performance of the web application.

In exploring the TabPanel, we learned how we could create tabs that could be static, controlled, disable or even hidden and programmatically control them. We also learned of the two of the configuration options, `deferredRender` and `layoutOnTabChange`, that

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

could cause some of our hair to jump ship. We exercised some of the common tab management methods and discussed some of the caveats for using this UI control.

While this chapter was relatively lengthy, I hope that by now you feel very comfortable with the material covered.

## **4.4 Summary**

We covered a lot of material about the Swiss Army Knife of UI display widgets, the Panel, which is enough to make just about any developer's head spin. In exploring the Panel class, we saw how it provides a plethora of options to display user interactive content including toolbars, buttons, title bar icons and miniature tools.

We used the Window class as a general container and mastered the art of adding and removing children dynamically providing us the ability to dynamically and drastically change an entire UI or a single widget or control.

In exercising the Window Class and its cousin, the MessageBox, we learned how we could replace the generic alert and prompt dialog boxes to get the users attention to display or request user input. We also had some fun fooling with the animated wait MessageBox.

Lastly, we examined the TabPanels, learning how to dynamically manage tab items as well as a few of the usability pitfalls that the UI control brings.

In the next chapter, we explore the many Ext Layout schemes, where you'll learn the common uses and pitfalls of these controls.

# 05

## *Organizing Components*

When building an application, many developers often struggle on how to organize their UI and which tools to use to get the job done. In this chapter, you'll gain the necessary experience to be able to make these decisions in a further educated manner. We're going to explore all of the numerous layout models and try to identify some of the best practices and common issues that you will face.

### **5.1 Laying it all out**

The Layout management schemes are what are responsible for visual organization of widgets on screen. They include simple layout schemes such as 'Fit', where a single child item of a Container will be sized to fit the Container's body or complex layouts such as border layout, which splits up a Container's content box into five manageable slices or "regions".

When exploring some of the layouts, we'll hit upon examples that are verbose, thus lengthy and can serve as a great springboard or starting point for your layout endeavors. We'll start our journey with taking a look at the Container Layout, which is the nucleus of the entire layout hierarchy.

### **5.2 The simple Container layout**

As you may be able to recall, the Container layout is the *default* layout for any instance of Container and simply places items on the screen, one on top of. I like to think of it as the Lincoln Logs of the Ext layouts.

Though the Container layout does not explicitly resize child items, a child's width may conform to the Container's content body if it is not constrained. It also serves as the base class for all other layouts, providing a much of the base functionality for the descendant layouts. Figure 5.1 illustrates the Ext.layout class hierarchy.

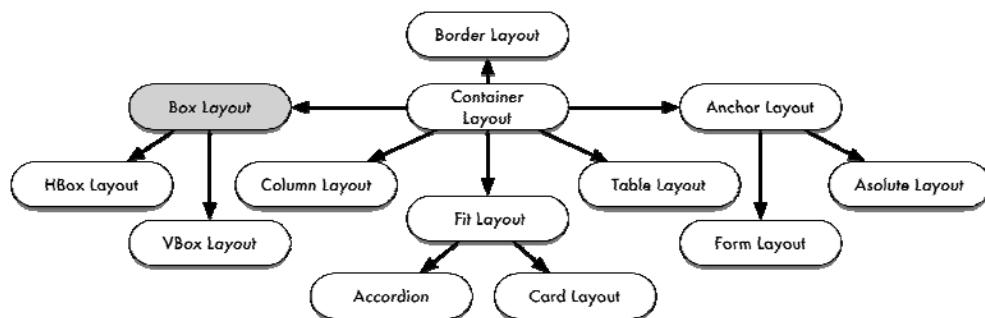


Figure 5.1 The layout class hierarchy, where layouts are descendants of the Container layout

Implementing a container layout is extremely simple, requiring you to just add and remove child items. In order to see this, we need to setup somewhat of a dynamic example, using quite a few components.

### **Listing 5.1 Leveraging the container layout**

```

var childPnl1 = { // 1
  frame : true,
  height : 50,
  html   : 'My First Child Panel',
  title  : 'First children are fun'
}

var childPnl2 = { // 2
  width : 150,
  html   : 'Second child',
  title  : 'Second children have all the fun!'
}

var myWin = new Ext.Window({ // 3
  height   : 300,
  width    : 300,
  title    : 'A window with a container layout',
  autoScroll: true, // 4
  items    : [ // 5
    childPnl1,
    childPnl2
  ],
  tbar : [ // 6
    {
      text   : 'Add child',
      handler: function() {
        var numItems = myWin.items.getCount() + 1;
        myWin.add({
          title   : 'Child number ' + numItems,
          height  : 60,
          frame   : true,
          collapsible: true,
          collapsed: true,
          html    : 'Yay, another child!'
        });
        myWin.doLayout();
      }
    }
  ]
}
  
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        }
    ]
});
```

- {1} The first child item, a Panel
- {2} The third child item, another Panel
- {3} The Window which contains the three child items
- {4} Set autoScroll to true, to allow the content body to scroll automatically
- {5} The three child items are referenced
- {6} The toolbar with the add item button

In Listing 5.1, we do are doing quite a lot to exercise the container layout. This because I want you to be able to see how the items stack and do not resize.

The first thing we do is instantiate object references using XTypes for the two child items that will be managed by a Window; childPnl1{#1} and childPnl2{#2}. These three child items are static.

Next, we begin our myWin{#3} reference, which is an instance of Ext Window. We also set the autoScroll property{#4} to true. This tells the Container to add the CSS attributes overflow-x and overflow-y to auto, which instructs the browser to show the scroll bars only when it needs to.

Notice how we set the child "items"{#5} property to an array. The items property for any container can be an instance of an array to list multiple children or an object reference for a single child. The window contains a toolbar{#6} that has a single button that, when pressed adds a dynamic item to the window. The rendered window should look like the one in figure 5.2.



Figure 5.2 The results of our first implementation of the container layout.

While the container layout does not provide much to manage the size of child items, it is not completely useless. It's lightweight relative to its descendants, which makes it ideal if you

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

want to simply display child items that have fixed dimensions. There are times, however, that you will want to have the child items dynamically resize to the container's content body. This is where the Anchor Layout can be very useful.

### 5.3 The Anchor layout

The Anchor layout is similar to the Container layout, where it stacks child items one on top of another, except it adds dynamic sizing to the mix using an *anchor* parameter specified on each child. This anchor parameter is used to calculate the size of the child item relative to the parent's content body size and is specified as a percentage, an offset, which is an integer. The anchor parameter is a string, with using the following format:

```
anchor : "width, height" // or "width height"
```

Let's take our first stab at implementing an anchor layout using percentages.

#### **Listing 5.2 The Anchor layout using percentages**

```
var myWin = new Ext.Window({
    height      : 300,                                     // 1
    width       : 300,
    layout     : 'anchor',
    border     : false,
    anchorSize : '400',
    items      : [
        {
            title  : 'Panel1',
            anchor : '100%, 25%',                           // 3
            frame   : true
        },
        {
            title  : 'Panel2',
            anchor : '0, 50%',                             // 4
            frame   : true
        },
        {
            title  : 'Panel3',
            anchor : '50%, 25%',                           // 5
            frame   : true
        }
    ]
});                                                      
myWin.show();                                          

{1} The parent container, myWin
{2} Layout set to 'anchor'
{3} Panel1's anchor parameters, 100% width, 25% of parent's height.
{4} Panel2's anchor parameters, full width, 50% of parent's height.
{5} Panel3's anchor parameters, full width, 25% of parent's height.
```

In Listing 5.2, we instantiate a `myWin`{#1}, an instance of `Ext.Window`, specifying the layout as `'anchor'`{#2}. The first of the child items, Panel 1 has its anchor parameters{#3} specified as 100% of the parent's width, and 25% of the parent's height. Panel 2 has its

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

anchor parameters{ #4} specified a little differently, where the width parameter is 0, which is shorthand for 100%. We set Panel2's height to 50%. Panel3's anchor parameters{ #5} are set to 50% relative width and 25% relative height. The rendered item should look like Figure 5.5.

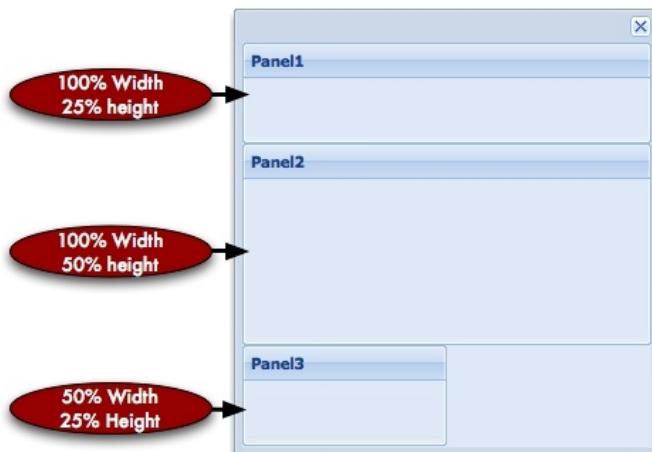


Figure 5.3 The rendered results of our first implementation of the Anchor layout.

Relative sizing with percentages is great, but we also have the option to specify an offset, which allows us to achieve greater flexibility with the Anchor layout.

Offsets are calculated as the content body dimension + *offset*. Generally, offsets are specified as a negative number to keep the child item in view. Lets put on our Algebra hats on for a second and remember that adding a negative integer is exact same as subtracting an absolute integer. Specifying a positive offset would make the child's dimensions greater than the content body's, thus requiring a scroll bar.

Lets explore offsets by using the previous example by modifying only the child item XTypes from Listing 5.7:

```
items : [
  {
    title      : 'Panel1',
    anchor    : '-50, -150',
    frame     : true
  },
  {
    title      : 'Panel2',
    anchor    : '-10, -150',
    frame     : true
  }
]
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

The rendered panel from the preceding layout modification should look like figure 5.3. We reduced the number of child items to only two to easily explain how offsets work and how they can cause you a lot of pain.

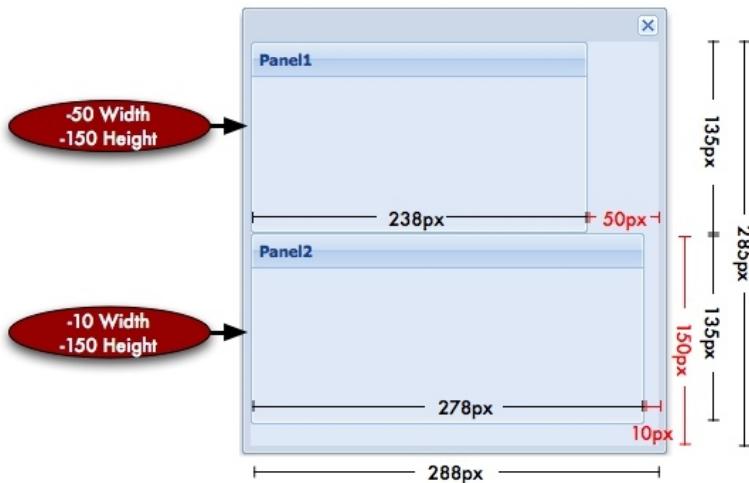


Figure 5.4 Using offsets with an Anchor layout with sizing calculations.

It is very important to dissect what's going on, which will require us to do a little math.

Through inspecting the DOM with Firebug, I learned that the window's content body is 285 pixels high and 288 pixels wide. Using simple math, we can easily determine what the dimensions of Panel1 and Panel2 should be.

$$\begin{aligned}\text{Panel1 Width} &= 288\text{px} - 50\text{px} = 238\text{px} \\ \text{Panel1 Height} &= 285\text{px} - 150\text{px} = 135\text{px}\end{aligned}$$

$$\begin{aligned}\text{Panel2 Width} &= 288\text{px} - 10\text{px} = 278\text{px} \\ \text{Panel2 Height} &= 285\text{px} - 150\text{px} = 135\text{px}\end{aligned}$$

We can easily see that both child panels fit perfectly within the Window. If we add the height of both of the panels, we see that they easily fit with a total of only 270px. But what happens if you resize the window vertically? Notice any strangeness? Increasing the Window's height by any more than 15 pixels results in Panel2' being pushed off screen, and scroll bars appearing in the windowBody.

Recall that with this layout, the child dimensions are relative to the parent's content body minus a *constant*, which is the offset. To combat this problem, we can mix anchor

offsets with fix dimensions. To explore this concept, we'll only need to modify Panel2's anchor parameters and add a fix height:

```
{
    title      : 'Panel2',
    height     : '150',
    anchor     : '-10',
    frame      : true
}
```

This modification makes Panel2's height a fixed 150 pixels. The newly rendered window can now be resized to virtually any size and Panel1 will grow to Window content body minus 150 pixels, which leaves just enough vertical room for Panel2 to stay on screen. One neat thing about this is that Panel2 still has the relative width.

Anchors are used for a multitude of layout tasks. A sibling of the Anchor layout, the form layout, is leveraged by the Ext.form.FormPanel class by default, but can be used by any Container or descendant that can contain other child items such as Panel or Window.

## 5.4 The Form layout

The form layout is just like the anchor layout, except wraps each child element in a div with the class 'x-form-item', which makes each item stack vertically like an outline. It adds a 'label' element in front of each of the child items, using the element's 'for' attribute, which when clicked, focuses on the child item.

### **Listing 5.3 The form Layout**

```
var myWin = new Ext.Window({
    height      : 180,
    width       : 200,
    bodyStyle   : 'padding: 5px',
    layout      : 'form',           // 1
    labelWidth  : 50,              // 2
    defaultType : 'field',         // 3
    items       : [
        {
            fieldLabel : 'Name',          // 4
            width      : 110
        },
        {
            fieldLabel : 'Age',           // 5
            width      : 25
        },
        {
            xtype      : 'combo',          // 6
            fieldLabel : 'Location',
            width     : 120,
            store     : [ 'Here', 'There', 'Anywhere' ]
        },
        xtype      : 'textarea',          // 7
    ]
});
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
        fieldLabel : 'Bio'  
    },  
    {  
        xtype      : 'panel',  
        fieldLabel : '',  
        labelSeparator : '',  
        frame      : true,  
        title      : 'Instructions',  
        html       : 'Please fill in the form',  
        height     : 55  
    }  
];  
});  
  
myWin.show();  
  
{1} Setting the window's layout to 'form'  
{2} Setting the default label width to 50px  
{3} Setting the default XType to 'field'  
{4} First Child Item, Textfield, constant width  
{5} Second Child Item, Textfield  
{6} Third Child Item, a combo box  
{7} Fourth child Item, a text area  
{8} Fifth Child item, a panel with instructions
```

There is a heck of a lot that we're doing here to achieve a fairly complex form layout. Like all of the other layouts, we set the Window's layout{#1} to 'form'. We set a layout-specific attribute, labelWidth{#2}, to 50 pixels. Remember the label element we discussed earlier? This attribute sets the width of that element. Next, we specify the default XType by setting the 'defaultType'{#3} attribute to 'field', which is used for the first{#4} and second{#5} child items, which automatically creates an instance of Ext.form.Field. The third child item{#6} is an xtype definition of a static combination autocomplete and drop down box, known as a combo box or Ext.form.ComboBox. The fourth child item is a simple xtype for a text area, while the last child item{#7} is a fairly complex XType object, specifying a panel.

In order to keep the field label element, but show no text, we set the fieldLabel's property to a string, containing a single space character. We also remove the label separator character, which is a colon (":") by default, by setting it as an empty string. The rendered code should look like Figure 5.5.



Figure 5.5 Using the form layout.

Remember that it is an ancestor to the Anchor layout, which makes it very powerful for dynamically resizing child items. While the layout in figure 5.9 works, it is static and could be improved. What if we wanted the Name, Location and Bio fields to dynamically size with its parent? Remember those anchor parameters? Lets use offsets to better our use of the form layout.

#### **Listing 5.4 Using offsets with the form layout**

```
{
    fieldLabel : 'Name',
    anchor     : '-4' // 1
},
{
    fieldLabel : 'Age',
    width     : 25
},
{
    xtype     : 'combo',
    fieldLabel: 'Location',
    anchor   : '-4',
    store    : [ 'Here', 'There', 'Anywhere' ]
},
{
    xtype     : 'textarea',
    fieldLabel: 'Bio',
    anchor   : '-4, -134' // 2
},
{
    xtype     : 'panel',
    fieldLabel: '',
    labelSeparator: '',
    frame    : true,
    title   : 'Instructions',
    html    : 'Please fill in the form',
    anchor   : '-4'
}
```

In the preceding code, we are adding anchor parameters to the child items originally defined in Listing 5.5. The rendered changes should look like figure 5.6. When you resize the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

example window, you'll see how well the child items resize and conform to their parent container.



Figure 5.6 Using offsets to create a much fuller looking form.

Always try to remember that the form layout is a direct descendant of the Anchor layout, this way you will not forget to set proper anchor parameters for dynamically resizable forms.

There are times where we need complete control over the positioning of the widget layout. The Absolute Layout is perfect for this requirement.

## 5.5 The Absolute layout

Next to the Container layout, the Absolute layout is by far one of the simplest to use. It fixes the position of a child by setting the CSS "position" attribute of the child's element to "absolute" and sets the top and left attributes to the x and y parameters that you set on the child items. Many designers place HTML elements as a "position:absolute" with CSS, but Ext leverages JavaScript's DOM manipulation mechanisms to set attributes to the elements themselves, with out having to muck with CSS.

Let's create a window with an absolute layout:

### **Listing 5.5 An absolute layout in action**

```
var myWin = new Ext.Window({
    height      : 300,
    width       : 300,
    layout      : 'absolute', // 1
    autoScroll  : true,
    border      : false,
    items       : [
        {
            title : 'Panel1',
            x     : 50, // 2
            y     : 50,
            height: 100,
            width : 100,
            html   : 'x: 50, y:50'
        }
    ]
});
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        frame  : true
    },
    {
        title  : 'Panel2',
        x      : 90,
        y      : 120,
        height : 75,
        width  : 77,
        html   : 'x: 90, y: 120',
        frame  : true
    }
]);
});

myWin.show();
{#1} Setting the window's layout to 'absolute'
{#2} Panel1's X and Y coordinates
{#3} Panel2's X and Y coordinates

```

By now, most of the code in here should look very familiar to you, except for a few new parameters. The first noticeable

change should be the Window's layout**{#1}** being set to 'anchor'. We've attached two children to this Window. Being that we are using the absolute layout, we need to specify the X and Y coordinates.

The first child, Panel1, has its X**{#1}** (CSS left attribute) set to 50 pixels and Y (CSS top attribute) coordinate set to 50. The second child, Panel2, has its X**{#2}** and Y parameters set to 90 pixels and 120 pixels. The rendered code should look like Figure 5.7.

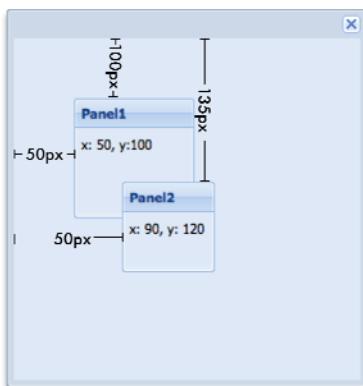


Figure 5.7 The results of our anchor layout implementation.

One of the apparent attributes about this example is that Panel 2 overlaps Panel 1. Panel 2 is on top is because of its placement in the DOM tree. Panel2's element is below Panel1's element and being that Panel2's CSS 'position' attribute is set to 'absolute' as well, it is going to show 'above' Panel1. Always keep the risk of overlapping in mind when you implement

this layout. Also, being that the position of the child items are fixed, which does not make the anchor layout an ideal solution for parents that resize.

If you have one child item and want it to resize with its parent, the fit layout is the best solution.

## 5.6 Making components ‘fit’

The ‘fit’ layout forces a Container’s single child to “fit” to its body and is, by far, the simplest of the layouts to use.

### Listing 5.6 The fit layout

```
var myWin = new Ext.Window({
    height      : 200,
    width       : 200,
    layout      : 'fit',                                // 1
    border      : false,
    items       : [
        {
            title : 'Panel1',
            html   : 'I fit in my parent!',
            frame  : true                                // 2
        }
    ]
});                                                     // 3
myWin.show();
```

{1} The window’s layout set to ‘fit’  
{2} The single child widget

In the preceding example, we set the Window’s layout to ‘fit’{1} and instantiate a single child, an instance of Ext.Panel{2}. The child’s XType is assumed by the Window’s “defaultType” attribute, which is automatically set to ‘panel’ by the Window’s prototype. The rendered panels should look like figure 5.8.



Figure 5.8 Using the fit layout for the first time

The fit layout is a great solution for a seamless look when a Container has one child. Often, however, we have multiple widgets being housed in a container. All other layout management schemes are generally used to manage multiple children. One of the best looking layouts is the Accordion layout, which allows you to vertically stack items, which can be collapsed, showing the users one item at a time.

## 5.7 The Accordion layout

The Accordion layout, a direct descendant of the fit layout, is useful when you want to display multiple Panels vertically stacked, where only a single item can be expanded or contracted.

### Listing 5.7 The Accordion layout

```
var myWin = new Ext.Window({
    height      : 200,                                // 1
    width       : 300,
    border     : false,
    title      : 'A Window with an accordion layout',
    layout     : 'accordion',                         // 2
    layoutConfig : {
        animate : true
    },
    items : [
        {
            xtype      : 'form',                      // 4
            title      : 'General info',
            bodyStyle   : 'padding: 5px',
            defaultType : 'field',
            labelWidth  : 50,
            items       : [
                {
                    fieldLabel : 'Name',
                    anchor    : '-10',
                },
                {
                    xtype      : 'numberfield',
                    fieldLabel : 'Age',
                    width     : 30
                },
                {
                    xtype      : 'combo',
                    fieldLabel : 'Location',
                    anchor    : '-10',
                    store     : [ 'Here', 'There', 'Anywhere' ]
                }
            ]
        },
        {
            xtype  : 'panel',                           // 5
            title : 'Bio',
            layout : 'fit',
            items : {
                xtype : 'textarea',
                value : 'Tell us about yourself'
            }
        }
    ],
    // 6
});
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        title : 'Instructions',
        html   : 'Please enter information.',
        tools  : [
            {id : 'gear'}, {id:'help'}           // 7
        ]
    }
});

myWin.show();

{1} myWin, an instance of Ext.Window
{2} specifying an Accordion layout
{3} specifying the Accordion layout's layoutConfig
{4} the first child item, a form panel
{5} the second child item, a panel, which contains a textarea
{6} the last child item, a vanilla panel, which has some tools

```

Listing 5.7 is quite large so we can demonstrate the usefulness of the Accordion layout. The first thing we do is instantiate a Window, `myWin{1}`, which has its layout set to `'accordion'{2}`. A new configuration option you have not seen thus far is `layoutConfig{3}`. Some layout schemes have specific configuration options, which you can define as a configuration option for a Component's constructor.

These `layoutConfig` parameters can change the way a layout behaves or functions. In this case, we set the `layoutConfig` for the accordion layout, specifying `animate:true`, which instructs the accordion layout to animate the collapse and expansion of a child item. Another behavior changing configuration option is `activeOnTop`, which if set to true, will move the active item to the top of the stack. When working with a layout for the first time, I suggest consulting the API for all the options available to you.

Next, we start to define child items, which leverage some of the knowledge we've gained thus far. The first child, is a form panel`{4}`, which uses the anchor parameters we learned about earlier in this chapter. Next, we specify a panel`{5}` that has its layout set to fit and contains a child text area. Lastly, we define the last child item as a vanilla panel with some tools. The rendered code should look like Figure 5.9.

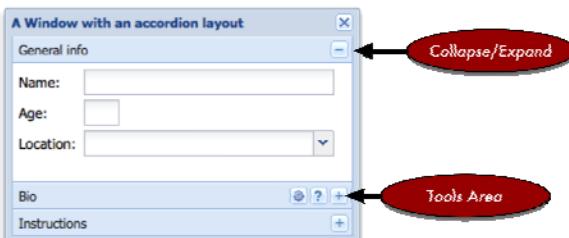


Figure 5.9 The Accordion layout is an excellent way to present the user with multiple items with a single visible component.

One important to note that the accordion layout can only function well with Panels and two of its descendants, GridPanel and TreePanel. This is because the Panel (and the two specified subclasses) has what is required for the accordion to function properly. If you desire anything else inside of an accordion such as a tab panel, simply wrap a panel around it and add that panel as a child of the Container that has the accordion layout.

While the accordion layout is a good solution for having more than one panel on screen, it has its limitations. For instance, what if you needed to have 10 Components in a particular Container? The sum of the heights of the title bars for each item would take up a lot of valuable screen space. The card layout is perfect for this requirement, allowing you to show and hide child components or “flip” through them.

## 5.8 The Card layout

A direct descendant of the Fit Layout, the Card Layout ensures that its children conform to the size of the Container. Unlike the Fit layout however, the Card layout can have multiple children under its control. This tool gives us the flexibility to create components that mimic wizard interfaces.

Except for the initial active item, the Card layout leaves all of the flipping up to the end developer with its publicly exposed setActiveItem method. In order to create a wizard-like interface, we need to create a method which we can control the card flipping:

```
var handleNav = function(btn) {
    var activeItem = myWin.layout.activeItem;
    var index = myWin.items.indexOf(activeItem);
    var numItems = myWin.items.getCount() - 1;
    var indicatorEl = Ext.getCmp('indicator').el;

    if (btn.text == 'Forward' && index < numItems) {
        myWin.layout.setActiveItem(index + 1);
    }
    else if (btn.text == 'Back' && index > 0) {
        myWin.layout.setActiveItem(index - 1);
    }

    indicatorEl.update((index + 1) + ' of ' + (numItems + 1));
}
```

In the preceding code, we control the card flipping by determining the active item's index and setting the active item based on if the Forward or Back button is pressed. We then update the indicator text on the bottom toolbar. Next, lets implement our Card layout. This particular code example is rather long and involving, so please stick with me.

### **Listing 5.8 The Card layout in action**

```
var myWin = new Ext.Window({
    height : 200,
    width : 300,
    border : false,
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

title      : 'A Window with a Card layout',
layout     : 'card',
activeItem : 0,
items      : [
    {
        xtype      : 'form',
        title      : 'General info',
        bodyStyle   : 'padding: 5px',
        defaultType : 'field',
        labelWidth  : 50,
        items       : [
            {
                fieldLabel : 'Name',
                anchor     : '-10',
            },
            {
                xtype      : 'numberfield',
                fieldLabel : 'Age',
                width     : 30
            },
            {
                xtype      : 'combo',
                fieldLabel : 'Location',
                anchor    : '-10',
                store     : [ 'Here', 'There', 'Anywhere' ]
            }
        ]
    },
    {
        xtype  : 'panel',
        autoEl : {},
        title : 'Bio',
        layout : 'fit',
        items : [
            {
                xtype : 'textarea',
                value : 'Tell us about yourself'
            }
        ],
        title : 'Congratulations',
        html  : 'Thank you for filling out our form!'
    }
],
bbar : [
    {
        text      : 'Back',
        handler   : handleNav
    },
    '-',
    {
        text      : 'Forward',
        handler   : handleNav
    },
    '->',
    {
        xtype  : 'box',
        id    : 'indicator',
        style : 'margin-right: 5px',
        autoEl : {
            tag  : 'div',
            html : '1 of 3'
        }
    }
]
);
});

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
myWin.show();  
{1} Setting the layout to 'card'  
{2} Set the Container's active item to 0  
{3} The back button and forward buttons  
{4} The BoxComponent with the id of 'indicator'
```

In Listing 5.8, we detail the creation of a Window, which leverages the Card layout. While most of this should be familiar to you, I feel that we should point out a few things. The first obvious item should be the `layout{1}` property, which is set to 'card'. Next, is the `activeItem{2}`, which the Container passes to the layout at render time. We set this to 0, which tells the layout to call the child Component's render method when the Container renders.

Next, we define the bottom toolbar, which contains the Back and Forward{3} buttons which call our previous defined `handleNav` method and our Box Component{4} that we use to display the index of the current active item. The rendered Container should look like the one in Figure 5.10.



Figure 5.10 Our first card layout implementation with a fully interactive navigation toolbar.

Pressing the Back or Forward buttons will invoke the `handleNav` method, which will take care of the card "flipping" and update the indicator Box Component. Remember that with the Card layout, the logic of the active item switching is completely up to the end developer to create and manage.

In addition to the previously discussed layouts, Ext offers a few more schemes. The Column layout is one of the favorite schemes among UI developers for organizing UI columns that can span the entire width of the parent Container.

## 5.9 The Column layout

Organizing components into columns allows you to display multiple components in a Container side by side. Like the Anchor layout, the Column layout allows you to set the absolute or relative width of the child Components. There are some things to look out for when using this layout. We'll highlight these in just a little bit, but first, let's construct a column layout window.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

**Listing 5.9 Exploring the column layout**

```

var myWin = new Ext.Window({
    height      : 200,
    width       : 400,
    autoScroll  : true,                                     // 1
    id          : 'myWin',
    title       : 'A Window with a Card layout',
    layout      : 'column',                                // 2
    defaults    : {
        frame : true
    },
    items       : [
        {
            title     : 'Col 1',
            id        : 'col1',
            columnWidth: .3                               // 3
        },
        {
            title     : 'Col 2',
            html      : "20% relative width",
            columnWidth: .2
        },
        {
            title : 'Col 3',
            html  : "100px fixed width",
            width : 100                                    // 4
        },
        {
            title     : 'Col 4',
            frame    : true,
            html    : "50% relative width",
            columnWidth: .5                               // 5
        }
    ]
});

myWin.show();
    {1} Automatically scroll the container
    {2} Set the layout to 'column'
    {3} Set a relative column width attribute, 30%
    {4} Set a fixed width of 100 pixels
    {5} Another relative width.

```

In a nutshell, the column layout is really easy to use. Declare child items, specify relative or absolute widths or a combination of both, like we do here. In Listing 5.9, we set the autoScroll{1} property of the Container to true, which ensures that scrollbars will appear if the composite of the child component dimensions grows beyond that of the Container's. Next, we set the layout property to 'column'{2}. We then go ahead and declare four child Components. The first of which has its relative width set to 30% via the columnWidth{3} attribute. Also, the second child has its relative width set to 20%. We mix things up a bit by setting an absolute width for the third child to 100 pixels{4}. Lastly, we set a relative width{5} for the last child to 50%. The rendered example should look like Figure 5.11.

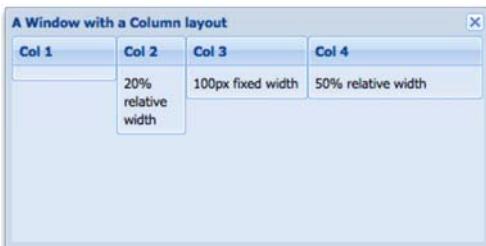


Figure 5.11 Our first column layout, which uses relative column widths with a fixed width entity.

If we tally up the relative widths, we see that they total up to 100%. How can that be? Three Components, taking 100% width *and* a fixed width Component? To understand how this is possible, we need to dissect how the Column layout actually sets the sizes of all of the child components. Lets put on our mathematical thinking caps for a moment.

The meat of the Column layout its `onLayout` method, which calculates the dimensions of the Container's body, which in this case, is 388 pixels. It then goes through all of its direct children to determine the amount of available space to give to any of the children with relative widths.

To do this, it first subtracts the width of each of the absolute-width child components from the known width of the containers body. In our example, we have one child with an absolute width of 100 pixels. The column layout calculates the difference between 388 and 100, which equals 288 (pixels).

Now that the Column layout knows exactly how much horizontal space it has left, it can set the size of each of the child components based on the percentage. It now goes through each of the children and sizes them based on the known *available* horizontal width of the container's body. It does this by multiplying the percentage (decimal) by the available width. Once complete, the sum of the widths of relatively sized Components turns out to be just about 288pixels.

Now that we understand the width calculations for this layout, lets change our focus to the height of the child items. Notice how the height of the child components does not equal the height of the container body. This is because the column layout does not manage the height of the child components. This causes an issue with child items that may grow beyond the height of its Container's body height. This is precisely why we set `autoScroll` to true for the Window. We can exercise this theory by adding an extra large child to the 'Col 1' Component by entering the following code inside of Firebug's JavaScript input console. Make sure you have a virgin copy of Listing 5.14 running in your browser.

```
Ext.getCmp('col1').add({
    height : 250,
    title  : 'New Panel',
    frame   : true
});
Ext.getCmp('col1').doLayout();
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

You should now see a panel embedded into the 'Col 1' panel with its height exceeding that of the window's Body. Notice how scroll bars appear in the window. If we did not set autoScroll to true, our UI would look cut off and might have its usability reduced or halted. You can scroll vertically and horizontally. The reason you can scroll vertically is because Col1's over-all height is greater than that of the Window's body. That is acceptable. The horizontal scrolling is the problem in this case. Recall that the Column layout only calculated 288 pixels to properly size the three columns with relative widths. Being that the vertical scrollbar is now visible, the physical amount of space from which the columns can be displayed is reduced by the width of the vertical scrollbar. To fix this issue, you must call doLayout on the Parent Container: Ext.getCmp('myWin').doLayout(), which will force a recalculation of the available horizontal space and resize the relatively sized columns so the Container's body need not scroll horizontally. Remembering to call the parent's doLayout method when adding a component to any of the direct children will help your UIs looking great.

As you can see, the Column layout is great for organizing your child components in columns. With this layout, however, there are two limitations. All child items are always left justified and their heights are unmanaged by the parent Container. Ext provides us with the HBox layout that overcomes the limitations of the Column layout and extend far beyond its capabilities.

## 5.10 HBox and VBox layouts

New to version 3.0 are the HBox layout's behavior is very similar to the Column model, where it displays items in columns, but allows for much greater flexibility. For instance, you can change the alignment of the child items both vertically and horizontally. Another great feature of this layout scheme is the ability to allow the columns or rows to stretch to their parent's dimensions if required. Lets dive into examining the HBox layout, where we'll create a Container with three child panels for us to manipulate.

### **Listing 5.10 HBox layout, exploring the packing configuration**

```
new Ext.Window({
    layout      : 'hbox',                                     // 1
    height      : 300,
    width       : 300,
    title       : 'A Container with an HBox layout',
    layoutConfig : {
        pack : 'start'
    },
    defaults   : {
        frame : true
    },
    items      : [
        {
            title  : 'Panel 1',
            height : 100
        },
        {
            title  : 'Panel 2',
        }
    ]
});
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

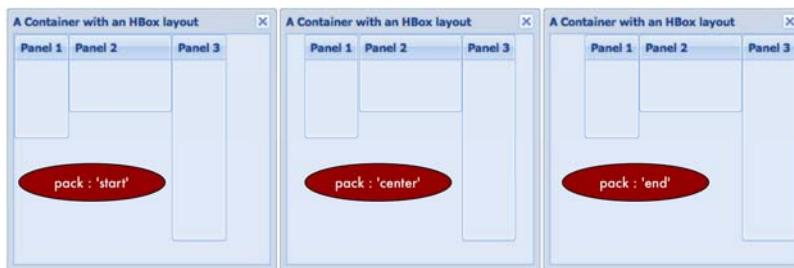
```

        height : 75,
        width  : 100
    },
    {
        title  : 'Panel 3',
        height : 200
    }
])
).show();

```

**{1} Setting the layout to hbox****{2} Specifying the layout configuration.**

For Listing 5.10, we set the layout to `hbox{1}` and specify the `layoutConfig{2}` configuration object. We created the three child panels with irregular shapes, allowing for us to properly exercise the different layout configuration parameters for which we can specify two, *pack* and *align*. Where *pack* means “vertical alignment” and *align* means “horizontal alignment”. Being able to understand the translated meanings for these two parameters is important as they are flipped for the HBox’s cousin, the VBox Layout. The *pack* parameter accepts three possible values; start, center and end. In this context, I like to think of them as left, center, and right. Modifying that parameter in Listing 5.15 will result in one of the rendered windows in Figure 5.12. The default value for the *pack* attribute is ‘start’.



**Figure 5.12** Different results with the three pack options.

The align parameter accepts four possible values: ‘top’, ‘middle’, ‘stretch’ and ‘stretchmax’. Remember that with the HBox, the align property specifies vertical alignment. Remember that the default parameter for align is ‘top’. In order to change how the child panels are vertically aligned, we need to override the default, by specifying it in the layoutConfig object for the Container. Figure 5.13 illustrates how we can change the way the children are sized and arranged based on a few different combinations.

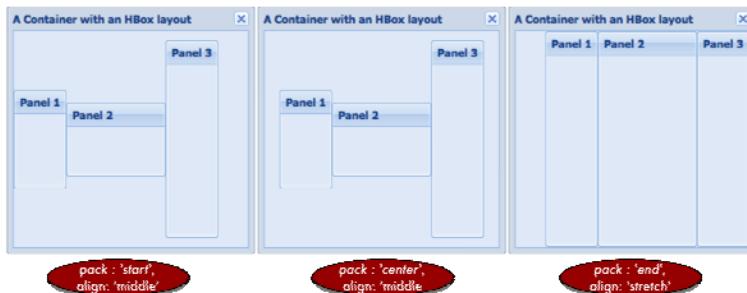


Figure 5.13 The 'stretch' alignment will always override any height values specified by the child items.

Specifying a value of 'stretch' for the align attribute instructs the HBox layout to resize the child items to the height of the Container's body, which overcomes one limitation of the column layout.

The last configuration parameter that we must explore is "flex", which is similar to the columnWidth parameter for the columnLayout and gets specified on the child items. Unlike the columnWidth parameter, the flex parameter is interpreted as a weight or a priority instead of a percentage of the columns. Lets say for instance, you would like each of the columns to have equal widths. Simply set each column's flex to the same value, and they will all have equal widths. If you wanted to have two of the columns expand to a total of one half of the width of the parent's container and the third to expand to the other half, make sure that the flex value for each of the first two columns is exactly half of the third column. For instance:

```
items : [
  {
    title  : 'Panel 1',
    flex   : 1
  },
  {
    title  : 'Panel 2',
    flex   : 1
  },
  {
    title  : 'Panel 3',
    flex   : 2
  }
]
```

Stacking items vertically is also possible with the VBox Layout, which follows exactly the same syntax as the HBox Layout. To use the VBox layout, simply modify Listing 5.15 and change the layout to 'vbox' and refresh the page. Next, you can apply the flex parameters described above to make each of the panels relative in height to the parent Container. I like to think of VBox as the container layout on steroids.

Contrasting the VBox Layout against HBox Layout, there is one parameter change. Recall that the align parameter for the HBox accepts a value of "top". For the VBox Layout, however, we specify, "left" instead of "top".

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Now that we have mastered HBox and VBox layouts, we will switch gears and the Table Layout, where you can position child Components, just like a traditional HTML table.

## 5.11 The Table layout

The table layout gives you complete control over how you want to visually organize your components. Many of us are used to building HTML tables the traditional way, where we actually write the HTML code. Building a table of Ext Components however, is different as we specify the content of the table cells in a single dimension array, which can get a little confusing. I'm sure that once you're done with these exercises, you'll be an expert in this layout. Lets create a 3 x 3 table layout:

### Listing 5.11 A vanilla table layout

```
var myWin = new Ext.Window({
    height      : 300,
    width       : 300,
    border      : false,
    autoScroll  : true,
    title       : 'A Window with a Table layout',
    layout      : 'table',                                // 1
    layoutConfig: {
        columns: 3                                     // 2
    },
    defaults   : {
        height : 50,                                    // 3
        width  : 50
    },
    items      : [
        {
            html : '1'
        },
        {
            html : '2'
        },
        {
            html : '3'
        },
        {
            html : '4'
        },
        {
            html : '5'
        },
        {
            html : '6'
        },
        {
            html : '7'
        },
        {
            html : '8'
        },
        {
            html : '9'
        }
    ]
});
myWin.show();
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

- {1} Specifying the layout as table
- {2} Set the number of columns for the table to 3
- {3} Set default size for each box to 50x50

The code in Listing 5.11 creates a Window Container that has nine boxes stacked in a 3x3 formation like in Figure 5.15. By now, most of this should seem very familiar to you, but I want to make sure we highlight a few items. The most obvious of which should be the layout parameter {1} being set to 'table'. Next, we set a layoutConfig{2} object, which sets the number of columns. Always remember to set this property when using this layout. Lastly, we're setting the defaults{3} for all of the child items to 50 pixels wide by 50 pixels high.

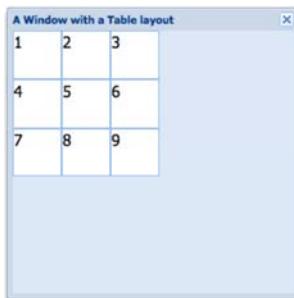


Figure 5.14 The results of our first simple table layout.

Often we need sections of the table to span multiple rows or multiple columns. To accomplish this, we specify either the rowspan or colspan parameters explicitly on the child items. Lets modify our table so the child items can span multiple rows or columns

### **Listing 5.12 Exploring rowspan and colspan**

```
items : [
    {
        html      : '1',
        colspan   : 3,                                     // 1
        width    : 150
    },
    {
        html      : '2',
        rowspan  : 2,                                     // 2
        height   : 100
    },
    {
        html : '3'
    },
    {
        html      : '4',
        rowspan  : 2,                                     // 3
        height   : 100
    },
    {
        html : '5'
    },
    html : '6'
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

},
{
    html : '7'
},
{
    html : '8'
},
{
    html : '9',
    colspan : 3,
    width : 150
}
]
{1} Set colspan to 3 and width to the total width to 150 pixels
{2} Set rowspan to 2 and height to 100 pixels
{3} Set rowspan to 2 and height to 100 pixels
{4} Set colspan to 3 and width to 150 pixels

```

In Listing 5.12, we reuse the existing Container code from Listing 5.14 and replace the child items array. We set the colspan attribute for the first panel**{1}** to 3, and manually set its width to fit the total known width of the table, which is 150 pixels. Remember that we have 3 columns of default 50x50 child containers. Next, we set the rowspan of the second child**{2}** item to 2 and its height to the total of two rows, which is 100 pixels. We do the exact same thing for Panel 4**{3}**. The last change involves panel 9, which has the exact same attributes as panel 1**{4}**. The rendered change should look just like Figure 5.15.

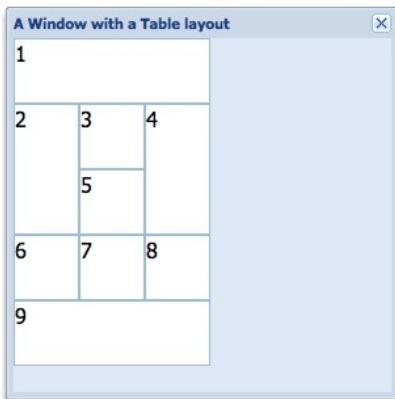


Figure 5.15 When using the table layout, you could specify rowspan and colspan for a particular Component, which will make it occupy more than one cell in the table.

When using the Table layout, you should remember a few things. First of which is to determine the total number of columns that will be used and specify it in the layoutConfig parameter. Also, if you're going to have Components span rows and/or columns be sure to

set their dimensions accordingly, otherwise the components laid out in the table will not seem to be aligned correctly.

The Table Layout is extremely versatile and can be used to create any type of box-based layout that your imagination conjures up with the main limitation being that there is no parent-child size management.

Moving to our last stop on the Ext Layout journey, we reach the ever-so-popular Border Layout, where you can divide any container into five collapsible regions that manage their children's size.

## 5.12 The Border layout

The Border Layout made its début in 2006, back when Ext was merely more than an extension to the YUI Library and has matured into an extremely flexible and easy to use layout that provides full control over its sub-parts or "regions". These regions are aptly named by polar coordinates: north, south, east, west and center. Figure 5.16 illustrates what a border layout implementation from the Ext SDK.

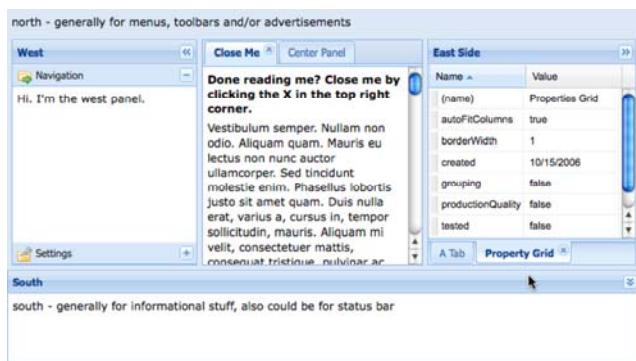


Figure 5.16 The border layout is what attracts many new developers to the Ext Framework and is widely used in many applications to divide the screen into task-specific functional areas.

Each region of a border layout is actually managed by the `BorderLayout.Region` class, which is what provides all of the UI and programmatic controls to that specific division of the layout scheme. Depending on the configuration options provided, the region can be resized or collapsed by the user. There are also options to limit the resize of the region or prevent it from being resized altogether.

To explore the Border Layout and the Region class, we will use the Viewport class, which we discussed in 5.1.6, earlier in this chapter.

### **Listing 5.13 Flexing the Border Layout**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

new Ext.Viewport({
    layout : 'border',
    defaults : {
        frame : true,
        split : true
    },
    items : [
        {
            title : 'North Panel', // 1
            region : 'north',
            height : 100,
            minHeight : 100,
            maxHeight : 150,
            collapsible : true
        },
        {
            title : 'South Panel', // 2
            region : 'south',
            height : 75,
            split : false,
            margins : {
                top : 5
            }
        },
        {
            title : 'East Panel', // 3
            region : 'east',
            width : 100,
            minWidth : 75,
            maxWidth : 150,
            collapsible : true
        },
        {
            title : 'West Panel', // 4
            region : 'west',
            collapsible : true,
            collapseMode : 'mini'
        },
        {
            title : 'Center Panel', // 5
            region : 'center'
        }
    ]
});
{1} Setting split to true in the defaults object, which is a BorderLayout.Region specific parameter
{2} The static "north" region panel.
{3} The resizable "south" region panel, which is also collapsible
{4} The resizable and collapsible East panel
{5} The mini-collapse west panel
{6} The no-frills Center panel.

```

In Listing 5.13, we accomplish quite a lot using the viewport a bit in just a few lines of code. We set the layout to “border”{1} and set split to true in the defaults configuration object. Being that there is a lot going on here at one time, feel free to reference Figure 5.17, which depicts what the rendered code will look like.

While all regions are technically divided, the split parameter instructs the Border Layout to render a five pixel high (or wide) divider between the center and regions. This divider is used as the resize handles for the regions. In order to work this magic, the border layout

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

employs the BorderLayout.SplitRegion class, which creates an absolutely position invisible div that intercepts the click and drag action of the user. When the drag action occurs, a proxy div appears, which is a direct sibling of the split bar handle div, allowing users to preview exactly how wide or high they are about to resize a region to.

Next, we begin to instantiate child items, which have BorderLayout.Region specific parameters. In order to review many of them, we make each region's behavior different from the other.

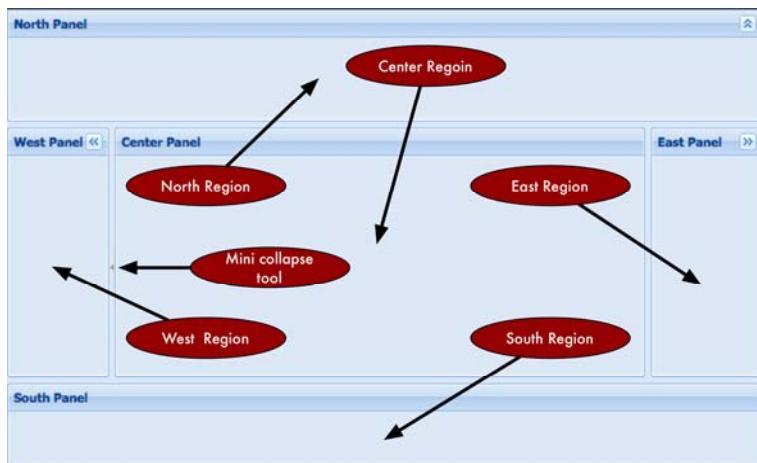


Figure 5.17 The Border Layout's versatility and ease of use makes it one of the most widely used in Ext-based RIAs.

For the first child{2}, we set the region property to 'north' to ensure that it is at the top of the border layout. We play a little game with the BoxComponent-specific parameter, height and the Region-specific parameters minHeight and maxHeight. By specifying a height of 100, we're instructing the Region to render the panel with an initial height of 100 pixels. The minHeight instructs the Region to not allow the split bar to be dragged beyond the coordinates that would make the northern region the minimal height of 100. The same is true for the maxHeight parameter, except it applies to expanding the region's height. We also specify the Panel-specific parameter of collapsible as true, which instructs the region to allow it to be collapsed to a mere 30 pixels high.

Defining the south region, the Viewport's second child{2}, we play some games to make prevent it from being resized, but work to keep the layouts 5 pixel split between the regions. Setting the split parameter to false, we instruct the region to not allow it to be resized. Doing so also instructs the Region to omit the 5 pixel split bar, which would make the layout somewhat visually incomplete. In order to achieve a façade-split bar, we specify a Region-specific 'margins' parameter, which specifies that we want the south region to have a

5 pixel buffer between itself and anything above it. One word of caution about this however; while the layout will now look complete, end-users may try to resize it, possibly causing frustration on their end.

The third child{3} is defined as the east region. This region is similarly configured to the north panel except it has sizing constraints that are a bit more flexible. Where the northern region starts its life out at its minimum size, the eastern region starts its life between its minWidth and maxWidth. Specifying size parameters like these allow the UI to present a region in a default or suggested size, but allow the panel to be resized beyond its original size.

The western region{4} is has a special Region-specific parameter, collapseMode, set to the string 'mini'. Setting this parameter that way instructs Ext to collapse a panel down to a mere five pixels, providing more visual space for the center region. Figure 5.18 illustrates how small. By allowing the split parameter to stay as true (remember our defaults object) and not specifying minimum or maximum size parameters, the western region can be resized as far as the browser will physically allow.

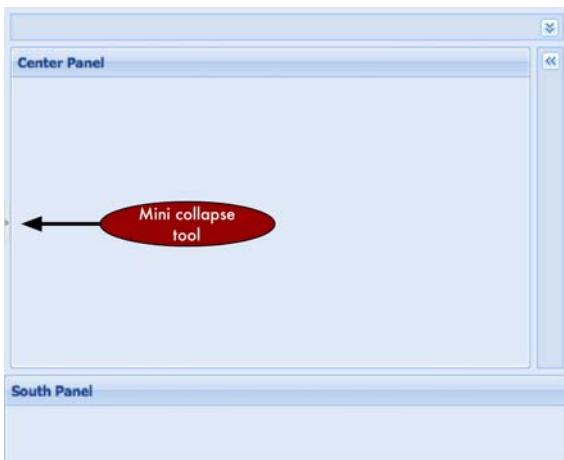


Figure 5.18 Our Border layout where two of the regions, north and east, are collapsed in regular mode and the west panel is collapsed in miniature mode.

The last region is the center region{5}, which is the *only* required region for the border layout. While our center region seems a bit bare, it is very special indeed. The center region is generally the canvas in which developers place the bulk of their RIA UI components and its size is dependent on its sibling region's dimensions.

For all of its strengths, the Border Layout has one huge disadvantage, which is that once a child in a region is defined/created it cannot be changed. Because the BorderLayout.Region is a base class and does not extend container, it does not have the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

power to replace a child once its instantiated. The fix for this is extremely simple. For each region where you wish to replace components, simply specify a Container as a region. Lets exercise this by replacing the center region section for listing 5.19:

```
{
    xtype : 'container',
    region : 'center',
    layout : 'fit',
    id : 'centerRegion',
    autoEl : {},
    items : {
        title : 'Center Region',
        id : 'centerPanel',
        html : 'I am disposable',
        frame : true
    }
}
```

Remember that the viewport can only be created once, so a refresh of page where the example code is required. The refreshed viewport should look nearly identical to Figure 5.18, except the center region now has HTML showing that it's disposable. In the example above, we simply define the container XType with the layout of fit and an ID that we can leverage with FireBug's JavaScript console.

Recalling our prior discussion and exercises over adding and removing child Components to and from a Container, can you remember how to get a reference to a Component via its ID and remove a child? If you can, excellent work! If you can't it's OK, I've already worked it out for us. But be sure to review the prior sections as they are extremely important to managing the EXT UI. Lets take a swipe at replacing the center region's child component:

#### **Listing 5.14 Replacing a Component in the center region**

```
var centerPanel = Ext.getCmp('centerPanel');
var centerRegion = Ext.getCmp('centerRegion');

centerRegion.remove(centerPanel, true);
centerRegion.add({
    xtype : 'form',
    frame : true,
    bodyStyle : 'padding: 5px',
    defaultType : 'field',
    title : 'Please enter some information',
    defaults : {
        anchor : '-10'
    },
    items : [
        {
            fieldLabel : 'First Name'
        },
        {
            fieldLabel : 'Last Name'
        },
        {
            xtype : 'textarea',
            fieldLabel : 'Bio'
        }
    ]
});
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
centerRegion.doLayout();
```

Listing 5.14 leverages everything we've learned thus far regarding Components, Containers and Layouts, providing us with the flexibility to replace the center region's child, a panel with a form panel with relative ease. You can use this pattern in any of the regions to replace items at will.

## 5.13 Summary

We took a lot of time to explore them many and versatile Ext Layout schemes. In doing so, we learned some of the strengths, weaknesses and pitfalls. Remember that while many layouts can do similar things, each has its place in a UI. Finding the correct layout to display components may not be immediately apparent and will take some practice if you're new to UI design altogether.

If you are exiting this chapter and are not 100% comfortable with the material, I would like to suggest moving forward and returning back to it after some time has past and the material has had some time to sink in.

Now that we have much of the core topics behind us, put your seatbelt on because we're going to be in for a wild ride, where we really start to learn more about and use Ext's UI widgets starting off with Form Panels.

# 06

## *Ext takes Form*

Developing and designing forms is a common task for web developers. Ext builds upon the basic HTML input fields to both add features of the developer and enhance the user experience. For instance, let's say a user is required to enter HTML into a form. Using out of the box text area input field, the user would have to write the HTML by hand. This is not required with the HTML Editor, where you get a full WYSIWYG input field, allowing the user to input and manipulate richly formatted HTML easily. In this chapter, we'll look into the FormPanel and learn about many of the Ext form input classes. We'll also see how we can leverage what we know about layouts and the Container model to build a complex form and use that implementation to submit and load the data via AJAX.

### **6.1 FormPanel at a glance**

With the Ext FormPanel you can submit data using AJAX, provide live feedback to users if a field deemed invalid and perform pseudo AJAX file uploads. Because the FormPanel is a descendant of the Container class, you can easily add and remove input fields to create a truly dynamic form. An added benefit is the ability to leverage other layouts or components, such as the tab panel with the card layout, to create robust forms that take considerably less screen space than traditionally laid out single-page forms. Because the FormPanel is a direct descendant of Panel, you get all of Panel's features, which include top and bottom toolbars and the button footer bar (fbar). Before we start to work with the FormPanel, we should take a glimpse at the underlying form element controller, known as the BasicForm.

#### **6.1.1 Configuring the underlying form controller**

The FormPanel leverages the BasicForm class, which is what serves as the controller for the FormPanel's form DOM element. While configuring your FormPanel, you can specify

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

properties for the BasicForm via the initialConfig property, which is defined as an object. Here is a sample FormPanel with an initialConfig object:

```
var x = new Ext.Form.FormPanel({
    ...
    initialConfig : {
        method      : 'GET',
        fileUpload   : true,
        standardSubmit : false,
        baseParams   : {
            foo : 'bar',
            sna : 'fu'
        }
    },
    ...
});
```

As illustrated above, some common attributes that developers usually set are method, baseParams, fileUpload and standardSubmit. In order to override the default HTTP POST method, you can specify the *method* property. One special attribute is fileUpload, which specifies if the form is going to perform a file upload function. If the form is to perform a standard submit operation (non-AJAX), you set the standardSubmit to true. The baseParams serves as an object, which provides a means to send the same parameters for each submission and can be used to replace the typical hidden fields.

Next, we'll begin our exploration of the various input fields that Ext provides. Once we're comfortable with them, we'll circle back to the FormPanel class and learn how we can submit and load data via AJAX.

## 6.2 The TextField

The Ext TextField features to the existing HTML input field such as basic validations, a custom validation method, automatic resizing and keyboard filtering. To utilize some of the more powerful features such as keyboard filters (masks) and automatic character stripping, you will need to know a little about Regular Expressions. If you're new to Regular Expressions, there is a plethora of information on the Internet.

We're going to explore quite a few features of the text field at once. Please stay with me, as some of example code can be pretty lengthy.

Because the TextField class is a descendant of Component, we could renderTo or applyTo an element on the page. Instead we're going to build them as children of a form panel, which will provide us a better presentation. To start, we'll create our items array, which will contain the XType definitions of the different text fields.

### **Listing 6.1 Our text fields**

```
Ext.QuickTips.init();

var fpItems = [
    {
        fieldLabel : 'Alpha only', // 1
        ...
    }
];
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        allowBlank : false, // 2
        emptyText  : 'This field is empty!', // 3
        maskRe     : /[a-z]/i // 4
    },
    {
        fieldLabel : 'Simple 3 to 7 Chars',
        allowBlank : false,
        minLength : 3, // 5
        maxLength : 7
    },
    {
        fieldLabel : 'Special Chars Only',
        stripCharsRe : /[a-zA-Z0-9]/ig // 6
    },
    {
        fieldLabel : 'Web Only with VType',
        vtype      : 'urlOnly' // 7
    }
];
{1} The field label is what is generally what is used to identify all fields in a form
{2} Setting allowBlank to false, ensures that Ext performs the basic blank validation on the field
{3} "Empty text" shows up in a form as helper when the field is empty
{4} Ensure only alpha characters are entered here
{6} Set the minimum and maximum number of characters
{6} A regular expression to strip out any non-special character
{7} Make use of the custom vType we will create later.

```

In the preceding code example we work a lot of angles to demonstrate the capabilities of the simple text field. We create four text fields in the fItems array. One of the redundant attributes that each child has is fieldLabel{#1}, which describes to the form layout (remember the form panel uses the form layout by default) what text to place in the label element for the field element.

For the first child, we ensure that the field cannot be blank by specifying allowBlank{#2} as false, which ensures we use one of Ext's basic field validations. We also set a string value for emptyText{#3}, which displays helper text and can be used as a default value. One important thing to be aware of is that it actually gets sent as the field's value during its form submission. Next, we set maskRe{#4}, a regular expression mask, to filter keystrokes that resolve to anything other than alpha characters. The second text field is built so it cannot be left blank and must contain from 3 to 7 characters to be valid. We do this by setting the minLength{#6} and maxLength parameters. The third textfield can be blank, but has automatic alphanumeric character striping. We enable automatic stripping by specifying a valid regular expression for the stripCharsRe{#6} property. For the last child item, we're going to veer off course for a bit to explore VTYPES.

Our last child item is a plain text field that makes use of a custom vtype{#7}, which we'll build out in a little. A VType is a custom validation method that is called automatically by the form field by a field losing focus or some time after the field is modified. To create your own VType, you can use a regular expression OR a custom function. The anatomy of a VType is simple and can contain up to three ingredients. The validation method is the only required item with the input mask regular expression and invalid text string as optional.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

The name of the VType is the validation method while the mask and text properties are a concatenation of the name with "Mask" or "Text". Let's create our custom VType:

```
var myValidFn = function(v) {
    var myRegex = /https?:\/\/([-\\w\\.]+)(:\\d+)?(\/([\\w/\\_.]*([?\\S+]?))?)?/;
    return myRegex.test(v);
}

Ext.apply(Ext.form.VTypes, {
    urlOnly : myValidFn,
    urlOnlyText : 'Must a valid web url'
});
```

OK, don't run away! The regular expression is scary, I know. It does serve a purpose though, and you'll see what I mean in a little bit. Our validation method, myValidFn, contains our monster regular expression and returns the result of using its test method, where we pass v, which is the value of the text field at the time the VType's validation method is called. Next, we apply an object to the Ext.form.VTypes singleton, which contains urlOnly - the reference to our validation method. Our VType is now known to Ext.form.VTypes as 'urlOnly', and is why we set the vtype property as such on the last textfield. We also set the urlOnlyText property for the vtype as a string with our custom error message. Now that we have explored VTypes, let's go on ahead to build the form from which our text fields will live:

### **Listing 6.2 Building the FormPanel for our text fields**

```
var fp = new Ext.form.FormPanel({
    renderTo : Ext.getBody(),
    width : 400,
    height : 160,
    title : 'Exercising textfields',
    frame : true,
    bodyStyle : 'padding: 6px',
    labelWidth : 126,
    defaultType : 'textfield', // 1
    defaults : {
        msgTarget : 'side', // 2
        anchor : '-20'
    },
    items : fpItems
});
```

- {1} Overriding the default XType to textfield
- {2} Specifying default target for the validation message.

Because we've already gone over the form layout, most of the code construct in Listing 6.2 should be very familiar to you. But, let's review a few key items relating to the form layout and Component model. We override the default Component XType by setting the defaultType{#1} property to 'textfield', which, if you can recall, will ensure our objects are resolved into text fields. We also setup some defaults{#2}, which ensure our error

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

message target is to the right side of the field and our anchor property is set. Lastly, we reference the FormPanel's items to the fpItems variable that we created earlier that contains the four text fields. The rendered FormPanel should look like figure 6.1.

Figure 6.1 The rendered results of our FormPanel which contains our four text fields.

Notice how in Figure 6.1 we have a little extra space to the right of the text fields. This is because we wanted to ensure that validation error messages are displayed to the right of the fields. This is why we set msgTarget to 'side' for our defaults object in our FormPanel definition. We can invoke validation one of two ways: Focus and blur (lose focus) of a field or invoking a form-wide isValid method call: fp.getForm().isValid(). Here is what the fields look like after validation has occurred:

Figure 6.2 "Side" validation error messages.

Each field can have its own 'msgTarget' property, which can be any of five possible attributes:

qtip	Displays an Ext quick tip on a mouse hover
title	Shows the error in the default browser "title" tooltip
under	Positions the error message below the field
side	Renders an exclamation icon to the right of the field
[element id]	Adds the text of the error message as the innerHTML of the target element

It is important to note that the msgTarget property only effects how the error message is displayed when the field is inside a form layout. If the text field is rendered to some

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

arbitrary element somewhere on the page (i.e. using renderTo or applyTo), the msgTarget will only be set to *title*. I encourage you to spend some time experimenting with the different msgTarget values, this way when it comes down to building your first real-world form, you'll have a good understanding of the way they work. Let's see how we can create password and file upload fields using the TextField.

### 6.2.1 Password and File select fields

To create a password field in HTML, you set its *type* attribute to 'password'. Likewise, for a file input field, you set *type* to 'file'. In Ext, to generate these

```
var fpItems = [
{
    fieldLabel : 'Password',
    allowBlank : false,
    inputType : 'password'
},
{
    fieldLabel : 'File',
    allowBlank : false,
    inputType : 'file'
];

```

Here is a rendered version of the password and file input fields in a form panel:

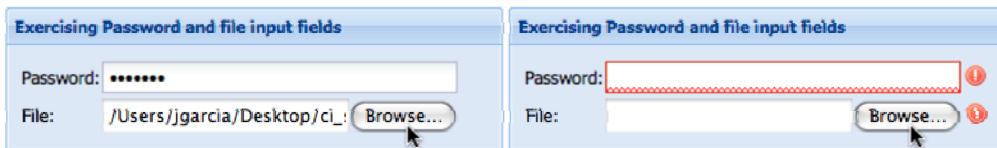


Figure 6.3 Our password and file upload fields with data filled in (left) and an example of the side validation error icons (right).

When using file upload fields, remember to configure the underlying form element with *fileUpload* : true, otherwise your files will never get submitted. Also, if you haven't noticed it, the file upload field in Figure 6.3 (right) does not have a red bounding box around it. This is because of the browser's security model preventing styling of the upload field.

### 6.2.2 Building a TextArea

The TextArea extends TextField and is a multiline input field. Constructing a TextArea is just like constructing a TextField, except you have to keep the component's height into consideration. Here is an example TextArea with a fixed height but a relative width.

```
{
    xtype      : 'textarea',
    fieldLabel : 'My TextArea',
    height     : 100
};
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

name      : 'myTextArea',
anchor   : '100%',
height   : 100
}

```

It's as easy as that. Let's take a quick look at how we can leverage the NumberField, which is another descendant of TextField.

### 6.2.3 The convenient NumberField

Sometimes requirements dictate that we place an input field that only allows numbers to be entered. We could do this with the text field and apply our own validation, but why reinvent the wheel? The NumberField does pretty much all of the validation for us for integer and floating numbers. Lets create a NumberField that accepts floating point numbers with the precision to thousandths and only allow a specific value.

```

{
    xtype      : 'numberfield',
    fieldLabel : 'Numbers only',
    allowBlank : false,
    emptyText  : 'This field is empty!',
    decimalPrecision : 3,
    minValue   : 0.001,
    maxValue   : 2
}

```

In the above example, we create our NumberField configuration object. In order to apply our requirements, we specify decimalPrecision, minValue and maxValue properties. This ensures that any floating number written with greater precision than 3 is rounded up. Likewise the minValue and maxValue properties are applied to ensure that valid range is 0.001 to 2. Any number outside of this range is considered invalid and Ext will mark the field as such. The NumberField looks exactly like the text field when rendered. There are a few more properties that can assist with the configuration of the NumberField. Please see the API documentation at

<http://extjs.com/docs/?class=Ext.form.NumberField> for further details.

Now that we have looked at the TextField and two of its subclasses, the TextArea and NumberField, lets look at its distant cousin, the ComboBox.

## 6.3 TypeAhead with the ComboBox

The cleverly named ComboBox input field is like a Swiss Army Knife of all text input fields. It is a *combination* of a general text input field and a general dropdown box to give you a very flexible and highly configurable combination input field. The ComboBox has the ability for automatic text completion (known as "type ahead") in the text input area and coupled with a *remote* data store, can work with the server side to filter results. If the combo box is performing a remote request against a large dataset, you can enable result paging via setting the pageSize property. The following is an illustration of the anatomy of a remote loading and paging ComboBox.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

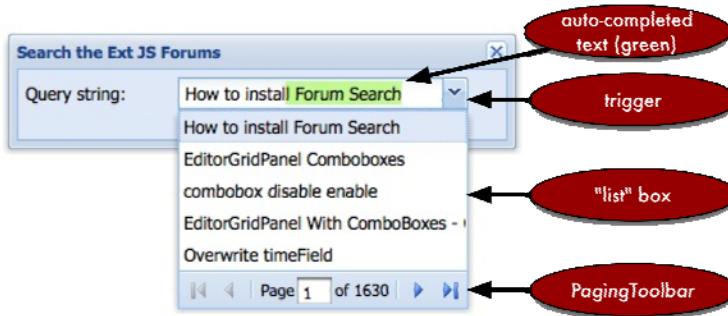


Figure 6.4 An example UI of a remote-loading and paging ComboBox with type ahead

Before we look at how the ComboBox works, we should explore how to construct one. Being that you're familiar with how to lay out child items, I think this is an excellent opportunity to leverage your new newly gained experience. So moving forward, when we discuss items that don't contain children, such as fields, I will leave it up to you to build a container. As a hint, you can use the Form Panel in listing 6.2.

### 6.3.1 Building a 'local' ComboBox

Creating a TextField is extremely simple compared to building a combo box. This is because the ComboBox has a direct dependency on a class called the DataStore, which is the main tool to manage data in the framework. We will just scratch the surface of this supporting class here and will go into much further detail in chapter 6. Lets move on to build our first combo box using an XType configuration object:

#### Lisiting 6.3 Building our first ComboBox

```
var mySimpleStore = new Ext.data.ArrayStore({                                     // 1
    data : [
        ['Jack Slocum'], ['Abe Elias'], ['Aaron Conran'], ['Evan Trimboli']
    ],
    fields : ['name']
});

var combo = {
    xtype      : 'combo',
    fieldLabel : 'Select a name',
    store      : mySimpleStore,                                         // 2
    displayField: 'name',                                              // 3
    typeAhead  : true,
    mode       : 'local'                                               // 4
}
```

- {#1} Building our first ArrayStore
- {#2} Specifying the store in the combo
- {#3} Setting the display field

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

#### {#4} Ensure there are no remote data AJAX requests

In listing 6.3, we construct a simple store that reads array data, known as an `ArrayStore{1}`, which is a preconfigured extension of the `Ext.data.Store` class, which makes it easy for us to create a store that digests array data. We populate the consumable array data and set it as the `data` property for the configuration object. Next, we specify the `fields` property as an array of data points from which the `DataStore` will read and organize Records. Being that we only have one data point per array in our array, we specify only a single point and give it a name of 'name'. Again, we'll go into much greater detail on the `DataStore` further on, where we'll learn the entire gamut from Records to connection Proxies.

We specify our combo as a simple POJSO, setting the `xtype` property as 'combo' to ensure that its parent container will call the correct class. We specify the reference of our previously created simple store as the `store{2}` property. Remember the `fields` property we set for the store? Well, the `displayField{3}` is directly tied to the fields of the data store that the combo box is using. Being that we have a single field, we will specify our `displayField` with that single field, which is 'name'. Lastly, we set the `mode{4}` to 'local', which ensures that the `DataStore` does not attempt to fetch data remotely. This attribute is extremely important to remember because the default value for `mode` is 'remote', which ensures that all data is fetched via remote requests. Not remembering to set it to remote will cause some pain. Here is what the combo box looks like rendered:

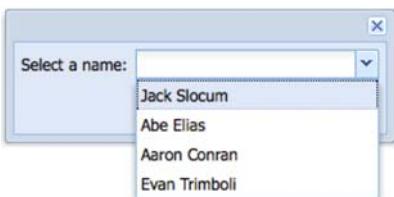


Figure 6.5 An example rendering of our ComboBox from Listing 6.3 inside of a Window.

To exercise the filtering and type ahead features, you can immediately start to type inside the text input field. Now our record set only contains four records, but we can begin to see how this stuff works. Entering a simple 'a' into the text field will filter the list and display only two names in the list box. At the same time, the ComboBox will type ahead the rest of the first match, which will show up as 'be Elias'. Likewise, entering in 'aa', will result in the store filtering in all but a single record and the type ahead will fill in the rest of the text, 'ron Coran'. There you have it, a nice recipe for a local ComboBox.

Using a local ComboBox is great if you have a minimal amount of static data. It does have its advantages and disadvantages however. Its main advantage being that the data does not have to be fetched remotely. This, however ends up being a major disadvantage when there is an extreme amount of data to parse through, which would make the UI slow

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

down, sputter or even grind to a halt showing that dreaded "This script is taking too long" error box. This is where the remote loading ComboBox can be called to service.

### 6.3.2 Implementing a 'remote' ComboBox

Using a remote ComboBox is somewhat more complicated than a static implementation. This is because you have a server side code to manage, which will include some type of server side store like a database. To keep our focus on the ComboBox, we'll use the pre-constructed PHP code at <http://tdg-i.com/dataQuery.php> on my site, which contains randomly generated names and addresses. Let's get on to implementing our remote ComboBox.

#### **Listing 6.4 Implementing a remote loading combo box**

```
var remoteJsonStore = new Ext.data.JsonStore({
    root           : 'records',                                // 1
    baseParams     : {
        column : 'fullName'
    },
    fields         : [                                         // 2
        {
            name   : 'name',
            mapping : 'fullName'
        },
        {
            name   : 'id',
            mapping : 'id'
        }
    ],
    proxy : new Ext.data.ScriptTagProxy({                      // 3
        url : 'http://tdg-i.com/dataQuery.php'
    })
});
```

```
var combo = {                                                 // 4
    xtype          : 'combo',
    fieldLabel     : 'Search by name',                         // 5
    forceSelection : true,
    displayField   : 'name',                                  // 6
    valueField     : 'id',                                    // 7
    hiddenName     : 'customerId',                           // 8
    loadingText    : 'Querying....',                          // 9
    minChars       : 1,                                      // 10
    triggerAction  : 'name',                                 // 11
    store          : remoteJsonStore
};
```

- {1} Specifying the root property for our records
- {2} Including base parameters on every remote request
- {3} Specifying the store fields, ensuring that the mapping is accurate
- {4} Allow Ext to request data from any domain
- {6} Ensuring that an item must be selected from the field
- {6} ensure the 'name' data field being displayed in the text field
- {7} send the 'id' data field when the form is submitted
- {8} specify a name for the hidden field who's value is set by the valueField
- {9} Specify custom loading text

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

- {10} Send a request as soon as a single character is entered
- {11} Always request all data when the trigger is pressed

In Listing 6.4, we change the data store type to a `JsonStore{1}`, which is a pre-configured extension of the `Ext.data.Store` class to allow us to easily create a store that can consume JSON data. For the store, we specify a `baseParams{2}` property, which ensures that base parameters are sent out with each request. For this instance, we only have one parameter, `column`, which is set to '`fulllname`' and specifies which column in the database the PHP code is to query from. We then specify `fields{3}`, which is now an array containing a single object and we translate the inbound '`fullName`' property to '`name`' with the `name` and `mapping` attributes. We also create a mapping for the ID for each record, which we'll use for submission. We could have set `fields` to an array of strings, like we did in our local `ArrayStore`, but that makes the mapping order dependant. If you specify `name` and `mapping`, the order of the properties in each record will not matter, which I prefer. Lastly for the store, we specify a `proxy{4}`, property where we create a new instance of `ScriptTagProxy`, a tool that is used to request data from across domains. We instruct the `ScriptTagProxy` to load data from a specific URL via the `url` property.

In creating our combo box, we are specifying `forceSelection{6}` to true, which is useful to use remote filtering (and `typeAhead` for that matter), but keeps users from entering arbitrary data. Next, we set the `displayField{6}` to '`name`', which shows the name data point in the text field and we specify the `valueField{7}` as '`id`', which ensures that the ID is used to send data when the combo's data is being requested for submission. The `hiddenName{8}` property is much overlooked but very important. Because we're displaying the name of the person, but submitting the ID, we need an element in the DOM to store that value. Because we specified `valueField` above, a hidden input field is being created to store the field data for the record that is being selected. To have control over that name, we specify `hiddenName` as '`customerId`'.

We also customize the list box's loading text by specifying a `loadingText{9}` string. The `minChars{10}` property defines the minimum number of characters that need to be entered into the text field before the combo executes a data store load and we override the default value of 4. Lastly, we specify `triggerAction{11}` as '`'all'`', which instructs the `Combo` to perform a data store load querying for all of the data. An example of our newly constructed `Combo` can be seen below.



Figure 6.6 An example rendition of our remote-loading combo from Listing 6.4

Exercise the rendered results, and you'll see how remote filtering can be a joy for a user to work with. Let's take a look at how the data coming back from the server is formatted

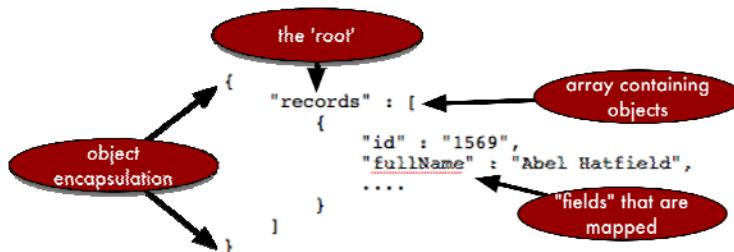


Figure 6.7 An exploded view of a slice of the served up JSON.

In examining a snippet of the resulting JSON in Figure 6.7, you can see the root that we specified in our remote combo's JSON store and the fullName field we mapped to. The root contains an array of Objects, which the DataStore will translate and pluck out any of the properties we map as "fields". Notice how the id is the first property in the record and fullName is the second. Because we used name and mapping in our store's fields array, our store will ignore id and all other properties in the records.

Following the format in Figure 6.7 when implementing your server side code will help ensure that your JSON is properly formatted. If you're unsure, you can use a free online tool at <http://jsonlint.com>, where you can paste your JSON and have it parsed and verified.

When exercising the example code in Listing 6.4, you might notice that when you click on the trigger, the UI's spinner stops for a brief moment. This is because all of the 2000 records in the database are being sent to the browser, parsed, and DOM manipulation is taking place to clear the list box and create a node. The transfer and parsing of the data is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

relatively quick for this large data set. DOM manipulation, however, is one of the main reasons for JavaScript slowing down and is why you would see the spinner animation stop. The amount of resources required to inject the 2000 DOM elements is intense enough for the browser to halt all animation and focus its attention on the task at hand. Not to mention bombarding the user with that many records may present a usability issue. To mitigate these issues, we should enable paging.

To do this, our server side code needs to be aware of these changes, which is the hardest part of this conversion. Luckily, the PHP code that we're using already has the code in place necessary to adapt to the changes we're going to make. The first change is adding the following property to your JSON store:

```
totalProperty : 'totalCount'
```

Next, we need to enable paging in our combo box. This can be done by simply adding a pageSize property to our combo box:

```
pageSize : 20
```

That's it! Ext is now ready to enable pagination to our combo box. Refresh the code in your browser and either click the trigger or enter a few characters into the text field and you'll see the results of your changes.

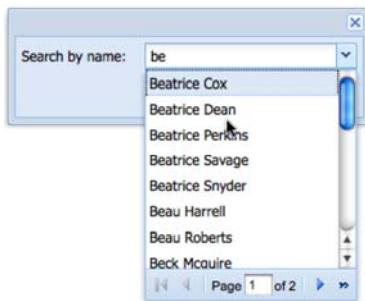


Figure 6.8 Adding pagination to our remote Combo Box.

Thus far, we've explored the UI of the ComboBox and implemented both local and remote versions of using both the Array and JSON Stores. Although we've covered a lot of the ComboBox, we have just been using it as an enhanced version of a drop down box and have not learned how we can customize the resulting data's appearance. In order to understand why we will be changing some things, such as the template and itemSelector, we will need to take a quick glance at the innards of the Combo.

### 6.3.3 The ComboBox deconstructed

At the nucleus of the ComboBox lay two helper classes. We've touched on the DataStore, which provides the data fetching and loading, but we have not really discussed the DataView, which is the Component responsible for displaying the result data in the list box as well as providing the events necessary to allow users to select the data. DataViews work by binding to DataStores by subscribing to events such as 'beforeload', 'datachanged' and 'clear'. They leverage the XTemplate, which actually provides the DOM Manipulation to stamp out the HTML based on the HTML template you provide. Now that we have taken a quick look at the components of a Combo box, let's move forward in creating our custom combo.

### 6.3.4 Customizing our ComboBox

When we enabled pagination in our ComboBox, we only saw names. But what if we wanted to see the full address along with the names that we're searching? Our data store needs to know of the fields. In modifying listing 6.4, we'll need to add the mappings for address, city, state and zip. I'll wait here while you finish that up.

Ready? Ok, before we can create a template, we need create some CSS that we'll need:

```
.combo-result-item {
    padding: 2px;
    border: 1px solid #FFFFFF;
}

.combo-name {
    font-weight: bold;
    font-size: 11px;
    background-color: #FFFF99;
}

.combo-full-address {
    font-size: 11px;
    color: #666666;
}
```

In the preceding CSS, we create a class for each of the divs in our template. Now we now need to create a new template so our list box can display the data that we wish. Enter the following code before you create your combo:

```
var tpl = new Ext.XTemplate(
    '<tpl for="."><><div class="combo-result-item">',
        '<div class="combo-name">{name}</div>',
        '<div class="combo-full-address">{address}</div>',
        '<div class="combo-full-address">{city} {state} {zip}</div>',
    '</div></tpl>'
);
```

We won't go too in depth into the XTemplate because it deserves its own section. It is important to note that any string encapsulated in curly braces ("{}") is directly mapped to the record. Notice how we have all of our data points except for 'id', which we don't need to

show and are just using for submission. The last change we need to make is to the combo itself. We need to reference the newly created template and specify an itemSelector:

```
tpl : tpl,
itemSelector : 'div.combo-result-item'
```

It's worth noting that the string for the itemSelector property is part of a pseudo sub-language called Selectors, which are patterns for which a query against the DOM can match. In this case, the Ext.DomQuery class is being used to select the div with the class 'combo-result-item' when any of its children are clicked. Your changes are now ready to be tested. If you did things correctly, your results should look something similar to Figure 6.9.

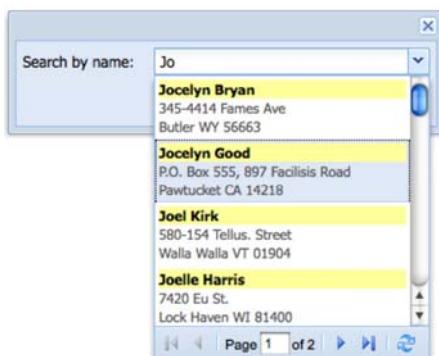


Figure 6.9 An example rendition of our customized combo box.

What we did to customize our combo box is the tip of the iceberg! Because you have complete control of the way the list box is being rendered, you can even include images or QuickTips in the list box.

In this section, we learned how to create a local and remote ComboBox's. We also learned about the ArraStore and JsonStore data store classes. We had some fun adding pagination to our remote implementation, dissected the ComboBox and customized the list box. The ComboBox has a descendant, the TimeField, which assists with creating ComboBox to select times from specific ranges. Let's see how we can create a TimeField.

### 6.3.5 Finding the time

The TimeField is another convenience class that allows us to easily add a time selection field to a form. To build a generic TimeField, you can simply create a configuration object with the xtype set to 'timefield' and you'll get a ComboBox that has selectable items from 12:00 AM to 11:46 PM. Here is an example of how to do that:

```
{
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        xtype      : 'timefield',
        fieldLabel : "Please select time",
        anchor     : '100%'
    }
}

```

Here is an example of how the field above would render:

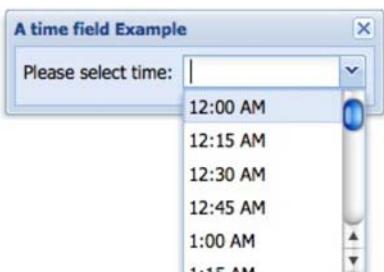


Figure 6.10 Our rendered generic TimeField.

The TimeField is configurable however, where you can set the range of time, increments and even the format. Let's modify our TimeField by adding the following properties, which will allow us to use Military time, set an increment of 30 minutes and only allow from 9AM to 6PM:

```

...
minValue  : '09:00',
maxValue  : '18:00',
increment : 30,
format    : 'H:i'

```

In the above property list, we set the minValue and maxValue properties which sets the range of time that we want our TimeField to have. We also set the increment property to 30 and format to 'H:i' or 24 Hours and two digit minutes. The format property must be valid per the Date.parseDate method. The full API documentation should be consulted if you intend on using a custom format. Here is the direct API link: <http://extjs.com/docs/?class=Date&member=parseDate>

Now that we've seen how The ComboBox and its descendant, the TimeField works, lets now take a look at the HTML Editor.

## 6.4 WYSIWYG?

The Ext HTML editor is known as a WYSIWYG or What You See Is What You Get editor. It is a great way to allow users to enter rich HTML formatted text without having to push them to master HTML and CSS. It allows you to configure the buttons on the toolbar to prevent certain interactions by the user. Let's move on to building our first HTML editor.

### 6.4.1 Constructing our first HTML editor

Just like the TextField, constructing a generic HTML editor is really simple:

```
var htmlEditor = {
    xtype      : 'htmleditor',
    fieldLabel : "Enter in any text",
    anchor     : '100% 100%'
}
```

Our HTML Editor rendered to a form will look like the following:

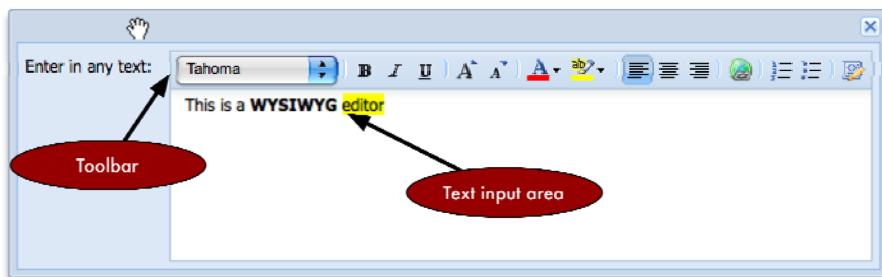


Figure 6.11 Our first HTML Editor in an Ext Window.

We discussed how the HTML editor's toolbar could be configured to prevent some items from being displayed. This is easily done by setting the enable<someTool> properties to false. For instance, if you wanted to disable the font size and selection menu items, you would set the following properties as false:

```
enableFontSize : false,
enableFont    : false
```

And that's all there is to it. After making the changes, refresh your page. You'll no longer see the text dropdown menu and the icons to change font sizes. To see a full list of the available options, be sure to visit the API. The HTML Editor is a great tool, but it, like many things has some limitations.

### 6.4.2 Dealing with lack of validation

The single biggest limitation to the HTML Editor is that it has no basic validation and no way to mark the field as invalid. When developing a form using the field, you will have to create your own custom validation methods. A simple validateValue method could be created as such:

```
var htmlEditor = {
    xtype      : 'htmleditor',
    fieldLabel : "Enter in any text",
    anchor     : '100% 100%',
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        allowBlank      : false,
        validateValue  : function() {
            var val = this.getRawValue();
            return (this.allowBlank ||
                (val.length > 0 && val != '<br>')) ? true : false;
        }
    }
}

```

While the validateValue method above will return false if the message box is empty or contains a simple line break element, it will not mark the field as such. We'll talk about how to test the form for validity before form submissions a little later in this chapter. For now, we'll switch gears and look at the date field.

## 6.5 Selecting a date

The DateField is a fun little form widget that is chock full of UI goodness that allows a user to either enter a date via an input field or select one via the leveraged DatePicker widget. Let's build out a DateField:

```

var dateField = {
    xtype      : 'datefield',
    fieldLabel : "Please select a date",
    anchor     : '100%'
}

```

Yes, it's that easy. Let's look at the how the DateField renders.



Figure 6.12 The DateField the DatePicker exposed (left) and the DatePicker's month and year selection tool (right).

This widget can be configured to prevent days from being selected by setting a disabledDate property, which is an array of strings that match the *format* property. The format property

defaults to 'm/d/Y' or 01/01/2001. Here are some recipes for disabling dates using the default format:

```
[ "01/16/2000", "01/31/2009" ] disables these two exact dates
[ "01/16" ] disables this date every year
[ "01.../2009" ] disables every day in January for 2009
[ "^\d{2}" ] disables every month of January
```

Now that we're comfortable with the DateField, lets move on to explore the Checkbox and Radio fields and learn how we can use the CheckboxGroup and RadioGroup classes to create clusters of fields.

## 6.6 CheckBoxes and Radios

The Ext CheckBox field wraps Ext element management around the original HTML Checkbox field, which includes layout controls as well. Like with the HTML Checkbox, you can specify the value for the checkbox, overriding the default Boolean value. Let's create some checkboxes, where we use custom values.

### **Listing 6.5 Building Checkboxes**

```
var checkboxes = [
    {
        xtype      : 'checkbox',
        fieldLabel : "Which do you own",
        boxLabel   : 'Cat',
        inputValue : 'cat'                                // 1
    },
    {
        xtype      : 'checkbox',
        fieldLabel : "",
        labelSeparator : '|',
        boxLabel   : 'Dog',
        inputValue : 'dog'                                // 2
    },
    {
        xtype      : 'checkbox',
        fieldLabel : "",
        labelSeparator : '|',
        boxLabel   : 'Fish',
        inputValue : 'fish'
    },
    {
        xtype      : 'checkbox',
        fieldLabel : "",
        labelSeparator : '|',
        boxLabel   : 'Bird',
        inputValue : 'bird'
    }
];
```

- {1} Specifying text that will reside to the right of the field
- {2} Overriding the default input value

The code in Listing 6.6 builds out four Checkboxes, where we override the default inputValue for each node. The boxLabel{1} property creates a field label to the right of the input field

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

and the `inputValue{2}`, of course, overrides the default Boolean value. An example rendering of the code above is as follows:

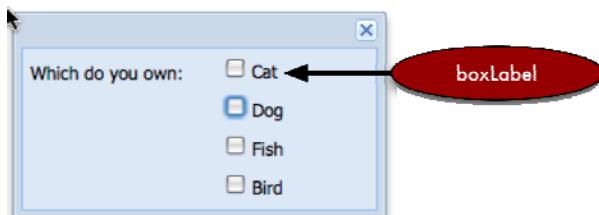


Figure 6.13 Our first four checkboxes.

While the above will work for a lot of forms, for some large forms, it is a waste of screen space. Let's use the CheckboxGroup to automatically layout our checkboxes.

#### **Listing 6.6 Using a checkbox group**

```
var checkboxes = {
    xtype      : 'checkboxgroup',
    fieldLabel : "Which do you own",
    anchor     : '100%',
    items       :
    [
        {
            boxLabel   : 'Cat',
            inputValue : 'cat'
        },
        {
            boxLabel   : 'Dog',
            inputValue : 'dog'
        },
        {
            boxLabel   : 'Fish',
            inputValue : 'fish'
        },
        {
            boxLabel   : 'Bird',
            inputValue : 'bird'
        }
    ]
}
```

Using the CheckboxGroup in this way will lay out your checkboxes in a single horizontal line as seen in figure 6.14. Specifying the number of columns is as simple as setting the `columns` attribute to the number of desired columns.



© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Figure 6.14 Two implementations of the CheckboxGroup. Single horizontal line (left) and a two column layout (right).

Your Implementation of the CheckboxGroup will depend on you needs and requirements. Implementing the Radio and RadioGroup classes is nearly identical to the Checkbox and CheckboxGroup classes. The biggest difference is that you can *group* radios by giving them the same name, which only allows one item to be selected at a time. Let's build a group of radios.

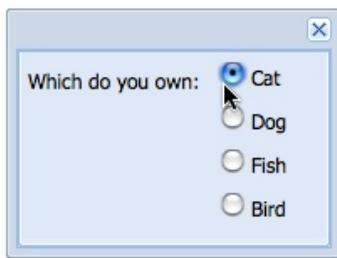


Figure 6.15 A single column of radios.

Because the RadioGroup class extends the CheckboxGroup class, the implementation is identical, so we'll save you from going over the same material. Now that we've gone over the Checkbox and Radio classes and their respective Group classes, we'll begin to tie these together by taking a more in-depth look at the form panel, where we'll learn to perform form-wide checks and complex form layouts.

## 6.7 Complex form layouts

Like the other components, the FormPanel class can leverage any layout that is available from the framework to create exquisitely laid out forms. To assist with the grouping fields, the FormPanel has a cousin called the Fieldset. Before we actually build our componentry, lets take a sneak peak of what we're going to achieve.

Figure 6.16 A sneak peek of the Complex form panel we're going to build

In constructing our complex form, we're going to have to construct two FieldSets one for the name information and another for the address information. In addition to the FieldSets, we're going to setup a TabPanel that has a place for text fields and two HTML editors. In this task, we're going to leverage all of what we've learned thus far, so buckle your seatbelts; we're going to go over quite a bit of code.

Now that we know what we're going to be constructing, let's start by building out the FieldSet that will contain the TextFields for the name information.

### **Listing 6.7 Constructing two FieldSets**

```
var fieldset1 = {
    xtype      : 'fieldset', // 1
    title      : 'Name',
    flex       : 1,
    border     : false,
    labelWidth : 60,
    defaultType: 'field',
    defaults   : {
        anchor   : '-10',
        allowBlank: false
    },
    items     : [
        {
            fieldLabel : 'First',
            name      : 'firstName'
        },
        {
            fieldLabel : 'Middle',
            name      : 'middle'
        },
        {
            fieldLabel : 'Last',
            name      : 'lastName'
        }
    ]
};
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        name      : 'firstName'
    }
]
}

```

- {1} Setting the xtype property to 'fieldset'  
{2} Removing the border from the fieldset

In constructing our first fieldset{1} XType, the parameters may look like that of a Panel or Container. This is because the FieldSet class actually extends Panel and adds some functionality for the collapse methods to allow you to include fields in a form or not, which we do not exercise here. The reason we're using the Fieldset in this instance is because it's giving us that neat little title up top and we're getting exposure to this Component.

We're going to skip rendering this first FieldSet because we're going to use it in a form panel a little later on. Let's go on to build the second FieldSet, which will contain the address information. This one is rather large, so please stick with me on this.

### **Listing 6.8 Building our second fieldset.**

```

var fieldset2 = Ext.apply({}, {
    flex      : 1,
    title    : 'Address Information',
    items     : [
        {
            fieldLabel : 'Address',
            name      : 'address'
        },
        {
            fieldLabel : 'Street',
            name      : 'street'
        },
        {
            xtype      : 'container', // 1
            border    : false,
            layout    : 'column',
            anchor   : '100%',
            items     : [
                {
                    xtype      : 'container', // 2
                    layout    : 'form',
                    width    : 200,
                    items     : [
                        {
                            xtype      : 'textfield', // 3
                            fieldLabel : 'State',
                            name      : 'state',
                            anchor   : '-20'
                        }
                    ]
                },
                {
                    xtype      : 'container', // 4
                    layout    : 'form',
                    columnWidth: 1,
                    labelWidth: 30,
                    items     : [
                }
            ]
        },
        {
            xtype      : 'container', // 5
            layout    : 'form',
            columnWidth: 1,
            labelWidth: 30,
            items     : [
        }
    ]
},
{
    xtype      : 'container',
    layout    : 'form',
    columnWidth: 1,
    labelWidth: 30,
    items     : [
}
]
}

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        xtype      : 'textfield', // 6
        fieldLabel : 'Zip',
        anchor    : '-10',
        name      : 'zip'

    }
}

]
}

],
fieldset1);
{1} Leverage Ext.apply to use some of our properties from our first FieldSet
{2} Column-layout Containers for our state and zip fields
{3} A form-layout Container for our State text field
{4} The actual state text field
{6} Another form-layout Container for the Zip Code TextField
{6} The Zip Code TextField

```

In Listing 6.8, we leverage `Ext.apply{1}` to copy many of the properties from `fieldset1` and apply them to `fieldset2`. This utility method is commonly used to copy or override properties from one object or another. We'll talk more about this method when we look into Ext's toolbox-o-methods. To accomplish the desired layout of having the State and Zip Code fields side-by-side, we had to create quite a bit of nesting. The child`{2}` of our second fieldset is actually a Container, which has its layout set to column. The first child of that Container a form-layout Container`{3}` which contains our State TextField`{4}`. The second child`{6}` of our column-layout Container is another form-layout Container, which contains our Zip Code TextField`{6}`.

You might be wondering why there are so many nested containers and perhaps why the code to get this done is so darn long. The Container nesting is required to use different layouts within other layouts. This might not make sense immediately. I think the picture will be clearer to you when we actually render the form. For now, lets move on to building a place for these two FieldSets to live.

In order to achieve the side-by-side look of the form, we're going to need to create a container for it that is setup to leverage the hbox layout. In order to have equal widths in the hbox layout, we've set both of our FieldSets to stretch property to 1. Let's build a home for the two FieldSets:

```

var fieldsetContainer = {
    xtype      : 'container',
    layout      : 'hbox',
    height     : 120,
    layoutConfig : {
        align : 'stretch'
    },
    items   : [
        fieldset1,
        fieldset2
    ]
}

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

In the preceding code block, we create a Container that has a fixed height but has no width set. This is because this Container's width will be automatically set via the vbox layout, which our future FormPanel will use.

Now that we have that done, we're going to move on to building a tab panel, which will have three tabs, one with phone number form elements and the other two being html editors. This will use the bottom half of the FormPanels' available height. We're going to configure all of the tabs in one shot so this will be pretty lengthy. Please bear with me on this one.

### **Listing 6.9 Building atabpanel with form items**

```
var tabs = [
    {
        xtype      : 'container',
        title     : 'Phone Numbers',
        layout    : 'form',
        bodyStyle : 'padding:6px 6px 0',
        defaults  : {
            xtype : 'textfield',
            width : 230
        },
        items: [
            {
                fieldLabel : 'Home',
                name       : 'home'
            },
            {
                fieldLabel : 'Business',
                name       : 'business'
            },
            {
                fieldLabel : 'Mobile',
                name       : 'mobile'
            },
            {
                fieldLabel : 'Fax',
                name       : 'fax'
            }
        ]
    },
    {
        title  : 'Resume',
        xtype   : 'htmleditor',
        name   : 'resume'                                // 2
    },
    {
        title  : 'Bio',
        xtype   : 'htmleditor',
        name   : 'bio'
    }
];
}

{1} The Container that has four Text Fields
{2} The two HTML Editors as Tabs
```

In Listing 6.9 we wrote a lot of code to construct the an array that comprises of the three tabs that will serve as children to our future TabPanel. The first tab{1} is a Container that leverages the Form Layout and has four text fields. The second{2} and third tabs are HTML editors that will be used to enter Resume and a short biography. Let's move on to building our tab panel:

```
var tabPanel = {
    xtype           : 'tabpanel',
    activeTab      : 0,
    deferredRender : false,
    layoutOnTabChange: true,
    border          : false,
    flex             : 1,
    plain            : true,
    items            : tabs
}
```

In the preceding code block, we configure a TabPanel object that contains our tabs. We're setting deferredRender to false because we want to ensure that the tabs are actually built and in the DOM when we get around to loading our data. We also set layoutOnTabChange to true to ensure that the doLayout method for the tab we're activating is called, which ensures that the tab is properly sized.

Our next task will be to construct the form panel itself, which is relatively trivial compared to all of its child items.

### **Listing 6.10 Piecing it all together**

```
var myFormPanel = new Ext.form.FormPanel({
    renderTo      : Ext.getBody(),
    width         : 700,
    title         : 'Our complex form',
    height        : 360,
    frame          : true,
    id            : 'myFormPanel',
    layout         : 'vbox',
    layoutConfig   : {
        align : 'stretch'
    },
    items          : [
        fieldsetContainer,
        tabPanel
    ]
});
```

Here, we're finally getting to create our FormPanel. We set renderTo so we can ensure the formPanel is automatically rendered. In order to have the fieldsetContainer and the TabPanel properly sized, we're using the vbox layout with layoutConfig's align property set to stretch. We only specified a height for the fieldsetContainer. We do this because other than the height of the fieldsetContainer we're letting the vbox do its job in managing the size of the child items of the FormPanel. Lets take a look at what this beast of a form renders to.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

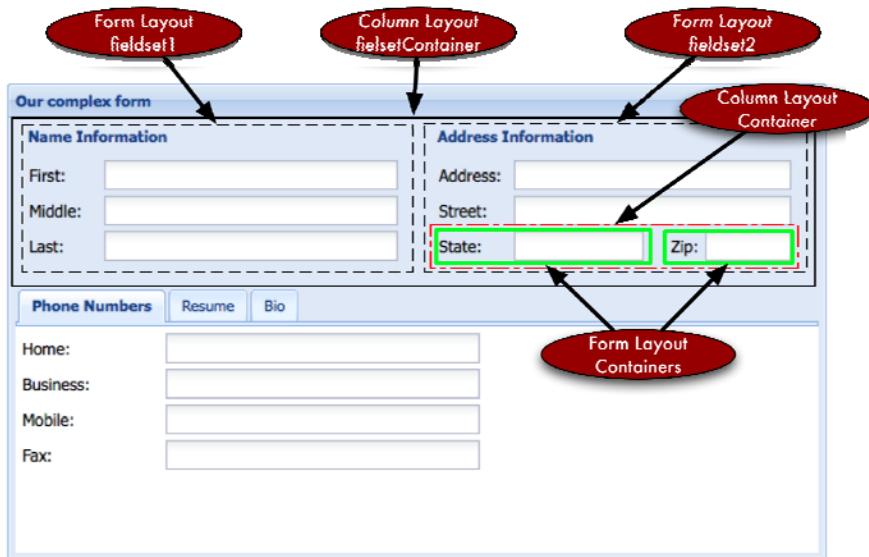


Figure 6.17 The results of our first complex layout form with the different containers used to comprise the complex layouts.

In the preceding Figure, I've highlighted the different Containers that comprise the first half of the form, which includes our fieldsetContainer, two FieldSets and their child components. In using this many containers, we're ensuring complete control on how the UI is laid. It's common practice to have these long code batches to create a UI with this type of complexity. In exercising our newly built FormPanel, you can flip through the three different tabs and reveal the HTML editors underneath.

By now you've seen how combining the usage of multiple components and layouts can result in something that is both usable and space saving. We now must focus our attention to learning to use for form for data submission and loading.

## 6.8 Data Submission and Loading

Submitting data via the basic form submit method is one of the most common areas new developers get tripped up on. This is because for so many years, we were used to submitting a form and expecting a page refresh. With Ext, the form submission requires a little bit of know-how. Likewise, loading a form with data can be a little confusing for some, so we'll explore the few ways you can do that as well.

### 6.8.1 Submitting the good old way

As we said before, submitting our form the good old way is extremely simple, but we need to configure the FormPanel's underlying form element with the standardSubmit property set to true. To actually perform the submission you simply call:

```
Ext.getCmp('myFormPanel').getForm().submit();
```

This will actually call the generic DOM form submit method, which will submit the form the old fashioned way. If you are going to use the FormPanel in this way, I would still suggest going over submitting via AJAX, which will highlight some of the features that you can't use when using the older form submission technique.

### 6.8.2 Submitting via AJAX

To submit a form, we must access the FormPanel's BasicForm component. To do this, we use the accessor method getForm or FormPanel.getForm(). From there, we have access to the BasicForm's submit method, which we'll use to actually send data via AJAX.

#### **Listing 6.11 Submitting our form**

```
var onSuccessOrFail = function(form, action) {
    var formPanel = Ext.getCmp('myFormPanel');
    formPanel.el.unmask(); // 1

    var result = action.result;
    if (result.success) { // 2
        Ext.MessageBox.alert('Success',action.result.msg);
    }
    else {
        Ext.MessageBox.alert('Failure',action.result.msg);
    }
}

var submitHandler = function() {
    var formPanel = Ext.getCmp('myFormPanel');
    formPanel.el.mask('Please wait', 'x-mask-loading');

    formPanel.getForm().submit({ // 3
        url      : 'success.true.php',
        success : onSuccessOrFail,
        failure : onSuccessOrFail
    });
}

{1} Unmask our form panel
{2} Display a message based on the status of the returning JSON
{3} Perform the actual form submission
```

In Listing 6.12, we create a success and failure handler called onSuccessOrFail, which will be called if the form submission attempt succeeds or fails. It will display an alert MessageBox{2} depending on the status of the returning JSON from the web server. We then move on to create the submission handler method named submitHandler, which

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

actually performs the form submission{3}. While we specify the url on the submit call, we could have specified it at the BasicForm or FormPanel level, but we specify it here because I wanted to point out that the target URL could be changed at runtime. Also, if you are providing any type of wait message, like we do here, you should have success and failure handlers.

At minimum, the returning JSON should contain a 'success' Boolean with the value of true. Our success handler is expecting a msg property as well, which should contain a string with a message to return back to the user:

```
{success: true, msg : 'Thank you for your submission.'}
```

Likewise, if your server side code deems that the submission was unsuccessful for any reason, the server should return a JSON object with the success property set to false. If you want to perform server side validation, which can return errors, your return JSON could include an errors object as well. Here is an example of a failure message with attached errors.

```
{
    success : false,
    msg      : 'This is an example error message',
    errors   : {
        firstName : 'Cannot contain "!" characters.',
        lastName  : 'Must not be blank.'
    }
}
```

If the returning JSON contains an errors object, the fields that are identified by that name will be marked invalid. Here is our form with the JSON code above served to it.

Figure 6.18 The results from our server side errors object using the standard QuickTip error msg.

In this section, we learned how to submit our form using the standard submit methods as well as the AJAX way. We also saw how we could leverage the errors object to provide server side validation with UI level error notification. Next, we'll look at loading data into our form using the load and setValues methods.

### 6.8.3 Loading data into our Form

The use cycle of just about every form includes saving and loading data. With Ext, we have a few ways to load data, but we must have data to load, so we'll dive right into creating some data to load. Let's create some mock data and save it in a file called data.php.

```
var x = {
    success : true,
    data : {
        firstName : "Jack",
        lastName : "Slocum",
        middle : "",
        address : "1 Ext JS Corporate Way",
        city : "Orlando",
        state : "Florida",
        zip : "32801",
        home : "123 346 8832",
        business : "832 932 3828",
        mobile : "",
        fax : "",
        resume : "Skills:<br><ul><li>Java Developer</li><li>Ext JS Senior
Core developer</li></ul>",
        bio : " Jack is a stand-up kind of guy.<br>"
    }
}
```

Just like form submission, the root JSON object must contain a success property with the value of true, which will trigger the setValues call. Also, the values for the form need to be in an object, whose reference property is data. Likewise, it's great practice to keep your form element names inline with the data properties to load. This will ensure that the right fields get filled in with the correct data. For the form to actually load the data via AJAX, you can call the BasicForm's load method, whose syntax is just like submit:

```
var formPanel = Ext.getCmp('myFormPanel');

formPanel.el.mask('Please wait', 'x-mask-loading');
formPanel.getForm().load({
    url : 'data.php',
    success : function() {
        formPanel.el.unmask();
    }
});
```

Executing the code above will result in our form panel performing an XHR and ultimately the form being filled in with the values as illustrated in below.

The screenshot shows a complex Ext JS form titled "Our complex form". It includes sections for "Name Information" and "Address Information". Under "Name Information", the "First" field is populated with "Jack", the "Middle" field is empty, and the "Last" field is "Slocum". Under "Address Information", the "Address" field is "1 Ext JS Corporate Way", the "City" field is "Orlando", the "State" field is "Florida", and the "Zip" field is "32801". Below these sections are three tabs: "Phone Numbers", "Resume", and "Bio". A toolbar is present with various icons for font style (Tahoma, B, I, U), alignment (A<sup>+</sup>, A<sup>-</sup>), and other form-related functions. A "Skills:" section contains a list of bullet points: "Java Developer" and "Ext JS Senior Core developer". At the bottom of the form are two buttons: "Submit" and "Load".

Figure 6.19 The results of loading our data via XHR.

If you have the data on hand, lets say from another component such as a DataGrid, you can set the values via `myFormPanel.getForm().setValues(dataObj)`. Using this, `dataObj` would contain only the proper mapping to element names. Likewise, if you have an instance of `Ext.data.Record`, you could use the form's `loadRecord` method to set the form's values.

Loading data can be as simple as that. Remember that if the server side wants to deny data loading, you can set the `success` value to false, which will trigger the `failure` method as referenced in the `load`'s configuration object.

## 6.9 Summary

In focusing on the `FormPanel` class, we've covered quite a few topics including many of the commonly used fields. We even got a chance to take an in-depth look at the `ComboBox` field, where we got our first exposure to of its helper classes, the `DataStore` and the `DataView`. Using that experience, we saw how we can customize the `ComboBox`'s resulting list box. We also took some time to build a relatively complex layout form and used our new tool to submit and load data.

Moving forward, we're going to take an in depth view at the data `GridPanel`, where we'll learn about its inner components and see how we can customize the look and feel of a grid. We'll also see how we can leverage the `EditorGridPanel` class to edit data inline. Along the way, we'll learn more about the `DataStore`. Be sure to get some candy, this is going to be a fun ride!

# 7

## *The venerable GridPanel*

Since the very early days of the Ext JS, the GridPanel has been the centerpiece of the framework, which can display data like a table but is much more robust. In many respects, I believe this still holds true to this day and is arguably one of its more complicated widgets, as it has a dependency on five directly supporting classes.

In this chapter, you're going to learn a lot about the GridPanel and the class that feeds it data, the Data Store. We'll start by constructing GridPanel that feeds from a Store that reads local in-memory Array data. At each step of the process, we're going to learn more about the both the DataStore and GridPanel and their supporting classes.

After we become more familiar with the data Store and GridPanel, we're going to move on to building a remote loading data Store that can parse JSON that will feed a paging Toolbar.

### **7.1 Introducing GridPanel**

At a first glance, the GridPanel may look like a glorified HTML table, which have been used for ages to display data. If you take a moment to look at one of the Ext JS Grid Examples, you'll come to the realization that this is no ordinary HTML table. You can see one example implementation of the GridPanel online, which uses an array store at: <http://extjs.com/deploy/dev/examples/grid/array-grid.html>. If you're not online, that's OK. I've included a snapshot of it below in Figure 7.1.

In the "Array Grid" Example (below), you can see that the features provided by this widget extend beyond those of a typical HTML table. These include column management features such as sorting, resizing, reordering, showing and hiding. Mouse events are also

tracked, out of the box, to allow you to highlight a row by hovering over it and even select it by clicking on it.

Array Grid				
Company	Price	Change	% Change	Last Updated
3m Co	\$71.72	0.02	0.03%	09/01/2009
Alcoa Inc	\$29.01	0.42	1.47%	09/01/2009
Altria Group Inc	\$83.81	0.28	0.34%	09/01/2009
American Express Company	\$52.55	0.01	0.02%	09/01/2009
American International Group, Inc.	\$64.13	0.31	0.49%	09/01/2009
AT&T Inc.	\$31.61	-0.48	-1.54%	09/01/2009
Boeing Co.	\$75.43	0.53	0.71%	09/01/2009
General Electric Company	\$34.14	-0.08	-0.23%	09/01/2009
General Motors Corporation	\$30.27	1.09	3.74%	09/01/2009
Hewlett-Packard Co.	\$36.53	-0.03	-0.08%	09/01/2009

Figure 7.1 The “Array Grid” Example, found in the examples folder of the downloadable SDK.

The example also demonstrates how the GridPanel’s “view” (known as the GridView) can be customized with what are known as “custom renderers”, which are applied to the “Change” and “% Change” columns. These custom renderers color the text based on negative and positive values.

This example merely skims the surface when it comes to how the GridPanel can be configured or extended. In order to fully understand more about the GridPanel and why it is so extensible, we need to know more about its supporting classes.

### 7.1.1 Looking under the hood

The key supporting classes that drive the GridPanel are the ColumnModel, GridView, SelectionModel and a DataStore. Lets take a quick glance at an implementation of a grid panel and see how each class plays a role in making the GridPanel work.

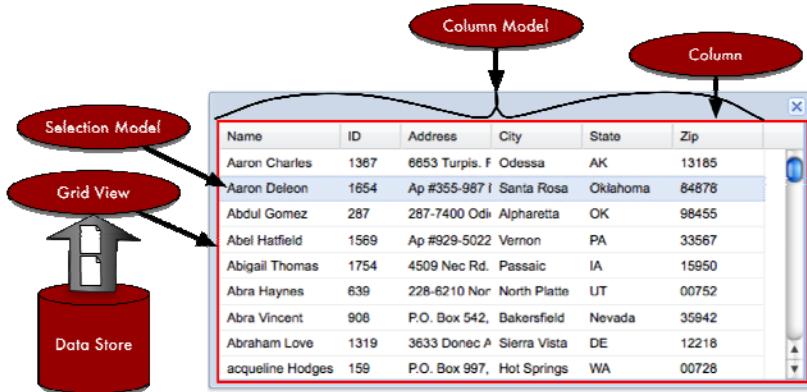


Figure 7.2 The GridPanel's five supporting classes, the DataStore, GridView, ColumnModel, Column and selection model.

In Figure 7.2, we can see a GridPanel and its five supporting classes highlighted. Starting from the very beginning, the data source, we have the DataStore class. DataStores work by leveraging a Reader, which is used to "map" data points from a data source and populate the data store. They can be used to read Array, XML or JSON data via the Array, XML and JSON readers. When the reader parses data, it is organized into Records, which are organized and stored inside the DataStore.

This should be a little familiar to you, as you leveraged it with creating combo boxes. As we learned earlier, DataStores can get their data from either local or remote sources. Just like the ComboBox, the DataStore feeds a View. In this case, it's the GridView.

The GridView is the class is the Actual UI component of the Grid View. It is responsible for reading the data and controlling the painting of data on screen. It leverages the ColumnModel to control the way the data is presented on screen.

The ColumnModel is the UI controller for each individual column. It is what provides the functions for Columns, such as resize, sort, etc. In order to do its job, it has to leverage one or more instances of Column.

Columns are classes that actually map the data fields from each individual record for placement on screen. They do this by means of a dataIndex property, which is set for each column and is responsible for displaying the data it obtains from the field its mapped to.

Lastly, the Selection Model is a supporting classes that work with a View to allow users to select one ore more items on screen. Out of the box, Ext supports Row, Cell and Checkbox Selection models.

OK, we have a nice head-start on GridPanels and their supporting classes. Before we actually go ahead and construct our first grid, we should learn more about the DataStore class, which many widgets in the framework depend on for data.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

## 7.2 The DataStore at a glance

As we learned just a bit ago, the DataStore is the class that provides the data for the grid panel. The data store actually feeds quite a few widgets throughout the framework wherever data is needed. To put this into plain view, here is an illustration enumerating the classes that depend on the DataStore.

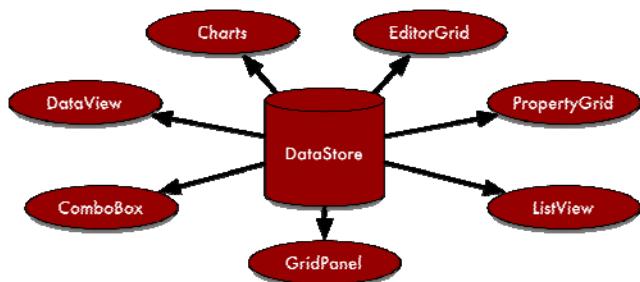


Figure 7.3 The DataStore and the classes it feeds data to. This illustration does not depict class hierarchy.

As we can see in Figure 7.3, the DataStore supports quite a few widgets, which include the DataView, ListView, ComboBox, Charts, the GridPanel and all of its descendants. The only exception to this pattern is the TreePanel. The reason for this is the DataStore contains a list of records, where TreePanels require hierarchical data.

Just as a quick warning, this may be one of those "dry" areas that you might not think is important - but hold on one second. Remember all of those classes that the DataStore feeds data to? Being proficient area of the framework better enable you to easily use any of those consumer widgets.

### 7.2.1 How DataStores work.

When we got our first real exposure to the DataStore, we learned how to use descendants of the DataStore, ArrayStore and JsonStore, to read Array and JSON data. These descendants are convenience classes - or preconfigured versions of the actual DataStore, which take care of things for us like attaching the correct reader for data consumption. We used these convenience methods because they make life easier for us in the earlier chapters, but there is a lot that is going on under the hood that is not immediately exposed and is important to know. We'll start by looking at exactly how the data flows from a data source to the store. We'll begin with a simple flow illustration.

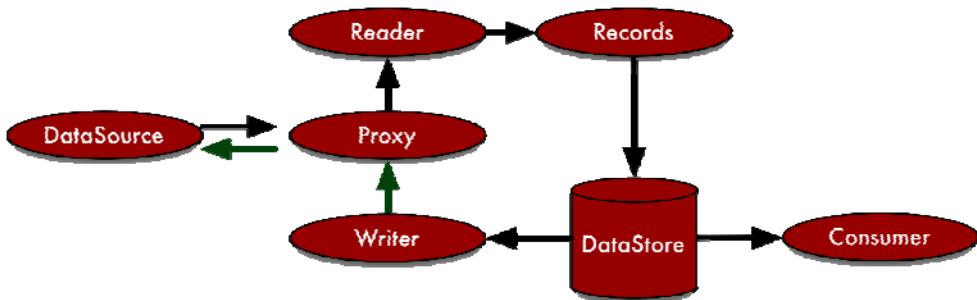


Figure 7.4 The data flow from a data source to a DataStore consumer.

As we can see in Illustration 7.4, the data always starts from a **DataProxy**. The **DataProxy** classes facilitate the retrieval of unformatted data objects from a multitude of sources and contain their own event model for communication for subscribed classes such as the **DataReader**. In the framework, there is an abstract class aptly called **DataProxy**, which serves as a base class for the descendant classes, which are responsible for retrieving data from specific sources as illustrated below.

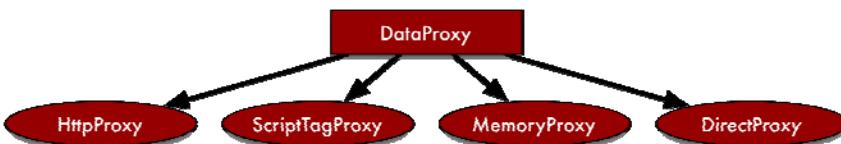


Figure 7.5 The **DataProxy** and its four descendants. Each are responsible for retrieving data from a specific data source.

The most commonly used proxy is the HTTP proxy, which leverages the browser's XHR object to perform generic AJAX requests. The HTTP proxy is limited, however, to the same domain because of what is known as the "same origin policy". This policy basically dictates that XHR requests via XHR cannot be performed outside of the domain from which a specific page is being loaded. This policy was meant to tighten security with XHRs, but has been construed as more of an annoyance than a security measure. The Ext developers were quick to come up with a work-around for this "feature", which is where the **ScriptTagProxy** (STP) comes into the picture.

The STP cleverly leverages the script tag to retrieve data from another domain and works very well, but requires that the requesting domain return JavaScript instead of generic data snippets. This is important to know because you can't just use the STP against any third party website to retrieve data. The STP *requires* the return data to be wrapped in a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

global method call, passing the data in as the *only* parameter. We'll learn more about the STP in just a little bit as we'll be using it to leverage extjsinaction.com to retrieve data from our examples.

The MemoryProxy is a class that offers Ext the ability to load data from a memory object. While you can load data directly to an instance of DataStore via its loadData method, usage of the MemoryProxy can be helpful in certain situations. One example is the task of reloading the data store. If you use DataStore.loadData, you need to pass in the reference to the data, which is to be parsed by the reader and loaded into the store. Using the memory proxy makes things simple, as you only need to call the DataStore.reload method and let Ext take care of the dirty work.

The DirectProxy is new to Ext JS 3.0 and allows the DataStore to interact with the Ext.direct remoting providers allowing for data retrievals via Remote Procedure Calls (RPC). We will not be covering usage of Direct as there is a direct dependency on server side language to provide the remoting methods.

#### **NOTE**

If you're interested in learning more about Ext.direct, I suggest visiting <http://extjs.com/products/extjs/direct.php> for details on specific server side implementations.

After a proxy fetches the raw data, a Reader then 'reads' or parses it. A Reader is a class that takes the raw-unformatted data objects and abstracts the data points known as 'dataIndexes' and arranges them into name data pairs, or generic objects. The following illustrates how this mapping works.

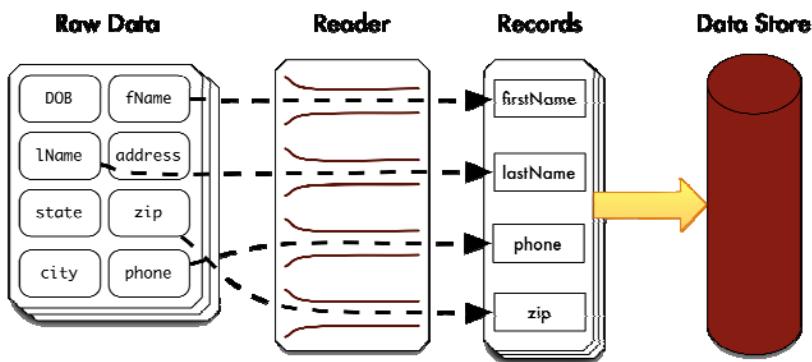


Figure 7.6 A Reader maps raw or unformatted data so that they can be inserted into Records, which then get spooled into a Data Store

As you can see in Figure 7.6, the raw and unformatted data is organized and fed into records that the Reader then creates. These Records are then spooled into the Data Store and are now ready to be consumed by a widget.

Ext provides readers for the three common data types. Array, XML and JSON. As the reader is chewing on records, it creates a Record for each row of data, which is to be inserted into the DataStore.

A Record is a fully Ext-managed JavaScript object. Much like Ext manages Element, the Record has getter and setter methods and a full event model for which the DataStore is bound. This management of data adds usability and some cool automation to the framework.

For example, changing a value of a record in a store that is bound to a consumer, like the GridPanel, will result in the UI being updated when the record is committed. We'll learn much more about management of records next chapter, when we learn about editable grids. After the Records are loaded into the DataStore, the bound consumer refreshes its view and the load cycle then completes.

Now that we have some fundamental knowledge of the DataStore and their supporting classes, we can begin to build our first GridPanel.

## 7.3 Building a simple GridPanel

When implementing GridPanels, I typically start by configuring the DataStore. The reason for this is because the configuration of the ColumnModel is directly related to the configuration of the DataStore. This is where we'll start too.

### 7.3.1 Setting up an Array DataStore

In the following Example, we're going to create a complete end-to-end data Store that reads data already present in memory. This means that we're going to instantiate instances of all of the supporting classes from the Proxy to the Store. This exercise will help us see the working parts being configured and instantiated. Afterwards, we'll learn how to use some of the pre-configured data Store convenience classes to make constructing certain types of stores easier with much less code.

#### **Listing 7.1 Creating a data Store that loads local array data.**

```
var arrayData = [ // 1
    ['Jay Garcia',      'MD'],
    ['Aaron Baker',     'VA'],
    ['Susan Smith',     'DC'],
    ['Mary Stein',       'DE'],
    ['Bryan Shanley',   'NJ'],
    ['Nyri Selgado',    'CA']
];
var nameRecord = Ext.data.Record.create([ // 2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

    {
      name : 'name', mapping : 1 },
      name : 'state', mapping : 2 }
  );

var arrayReader = new Ext.data.ArrayReader({}, nameRecord);           // 3
var memoryProxy = new Ext.data.MemoryProxy(arrayData);                // 4
var store = new Ext.data.Store({
  reader : arrayReader,
  proxy   : memoryProxy
}); 

{1} Creating local array data
{2} Using Ext.data.Record.create to create a constructor for an Ext.data.Record
{3} Instantiating an ArrayReader
{4} Constructing a new MemoryProxy
{5} Building our Store

```

In the above listing, we implement the full gamut of Data Store configuration. We first start by creating an array of arrays, which is referenced by the variable `arrayData`{1}. Please pay close attention to the format the array data is in, as this is the expected format for the `ArrayReader` class. The reason the data is an array of arrays is because each child array contained within the parent array is treated as a singular record.

Next, we create an instance of `data.MemoryProxy`, which is what will load our unformatted data from memory and is referenced by the variable `memoryProxy`{2}. We pass in the reference `arrayData` as the only argument.

Next, we create an instance of `data.Record`{3} and reference it in the variable `nameRecord`, which will be used as the template to map our array data points to create actual records. We pass an array of object literals{4} to the `Record.create` method, which is known as the ‘fields’ and details each field name and its mapping. Each of these object literals are configuration objects for the `Ext.data.Field` class, which is the smallest unit of data managed data within a Record. In this case, we map the field ‘`personName`’ to the first data point in each array record and the ‘`state`’ field to the second data point.

#### **NOTE**

Notice how we’re not calling `new Ext.data.Record()`. This is because `data.Record` is a special class that is able to create constructors by using its `create` method, which returns a new record *constructor*. Understanding how `data.Record.create` works is essential to performing additions to a Data Store.

We then move on to create an instance of `ArrayReader`{5}, which is what’s responsible for sorting out the data retrieved by the proxy and creating new instances of the record constructor we just created. From a 30 thousand foot view, the `ArrayReader` reads each Record, it creates a new instance of `nameRecord` by calling `new nameRecord`, passing the parsed data over to it, which is then loaded to the store.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Lastly, we create our DataStore for which we pass the reader and proxy we created, which completes the creation of our array data store. This completes our end-to-end example of how to create a store that reads array data. With this pattern, you can change the type of data the store is able to load. To do this, you swap out the ArrayReader with either a JsonReader or an XmlReader. Likewise, if you wanted to change the data source, you can swap out the MemoryProxy for another such as the HttpProxy, ScriptTagProxy or DirectProxy.

Recall that I mentioned something a bit earlier about convenience classes to make our lives a little easier. If we recreate the Store above using the ArrayStore convenience class, this is what our code would look like using our arrayData from above.

```
var store = new Ext.data.ArrayStore({
    data : arrayData,
    fields : ['personName', 'state']
});
```

Just as we see in the above example, we use shortcut notation for the fields to create an instance of `Ext.data.ArrayStore`. We achieve this by and passing a reference of the data, which is our `arrayData` and a list of fields, which provide the mapping. Notice how the `fields` property is a simple list of strings? This is a completely valid configuration of field mappings because Ext is smart enough to create the name and index mapping based on the string values passed in this manner. In fact, you could have a mixture of objects and strings in a fields configuration array. For instance, the following configuration is completely valid:

```
fields : [ 'fullName', { name : 'state', mapping : 2 } ]
```

Having this flexibility is something that can be really cool to leverage. Just know that having a mixture of field configurations like this can make the code a bit hard to read.

Using this convenience class saved us the step of having to create a proxy, record template and reader to configure the store. Usage of the `JsonStore` and `XML Store` are equally as simple, which we'll learn more about later. Moving forward, we'll be using the convenience classes to save us time.

For now we'll move on to creating the `ColumnModel`, which defines the vertical slices of data that our `GridPanel` will display along with our `GridView` component.

### 7.3.2 Completing our first GridPanel

As we discussed before, the `ColumnModel` has a direct dependency on the Data Store's configuration. This dependency has to do with a direct relationship between the data field records and the column. Just like the data fields map to a specific data point in the raw inbound data, columns map to the record field `names`.

To finish our `GridPanel` construction, we need to create a `ColumnModel`, `GridView`, `SelectionModel` and then we can configure the `GridPanel` itself.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

### Listing 7.2 Creating an ArrayStore

```

var colModel = new Ext.grid.ColumnModel([
    {
        header      : 'Full Name',
        sortable   : true,
        dataIndex  : 'fullName'                                // 1
    },
    {
        header      : 'State',
        dataIndex  : 'state'
    }
]);                                                       // 2

var gridView = new Ext.grid.GridView();                      // 3
var selModel = new Ext.grid.RowSelectionModel({
    singleSelect : true
})                                                       // 4

var grid = new Ext.grid.GridPanel({
    title       : 'Our first grid',
    renderTo   : Ext.getBody(),
    autoHeight : true,
    width      : 250,
    store       : store,                                     // 5
    view        : gridView,
    colModel    : colModel,
    selModel    : selModel
});                                                       // 6

{1} Creating our ColumnModel
{2} Mapping the dataIndexes to the columns themselves
{3} Instantiating a new GridView
{4} Creating a new single-selection RowSelectionModel
{5} Instantiating our Grid
{6} Referencing our Store, GridView, ColumnModel and SelectionModel

```

In the above Listing, we configure all of the supporting classes before constructing the GridPanel itself. The first thing we do is create a reference for a newly instantiated instance of a ColumnModel, for which we pass in an array of configuration objects. Each of these configuration objects are used to instantiate instances of Ext.grid.Column (or any subclasses thereof), which is the smallest managed unit of the ColumnModel. These configuration objects **{2}** detail the text that is to be populated in the column *header* and which Record field the column maps to, which is specified by the dataIndex property. This is where we see the direct dependency on the configuration of the Store's fields and the ColumnModel's columns. Also, notice that we set sortable to true for the "Full Name" column and not the "State" column. This will enable sorting on just that one column.

We then move on to create an instance of Ext.grid.GridView**{3}**, which is responsible for managing each individual row for the grid. It binds key event listeners to the Data Store, which it requires to do its job. For instance, when the Data Store performs a load, it fires the "datachanged" event. The GridView listens for that event and will perform a full refresh. Likewise, when a record is updated, the Data Store fires an "update" event, for which the

GridView will only update a single row. We'll see the update event in action later in the next chapter, when we learn how to leverage the EditableGrid.

Next, we create an instance of Ext.grid.RowSelectionModel{4} and pass a configuration object that instructs the selection model to only allow single selection of rows to occur. There are two things to know this step. The first is that by default, the GridPanel always instantiates an instance of RowSelectionModel and uses it as the default selection model if we do not specify one. But we did create one because by default the RowSelectionModel actually allows for multiple selections. You can elect to use the CellSelectionModel in place of the RowSelectionModel. The CellSelectionModel does not allow for multiple selections of items, however.

After we instantiate our selection model, we move on to configure our GridPanel{5}. GridPanel extends Panel, so all of the Panel-specific configuration items apply. The only difference is you *never* pass a layout to the grid panel as it will get ignored. After we set the Panel-specific properties, we set our GridPanel-specific properties. This includes configuring the references for the data Store, ColumnModel, GridView and Selection Model. Loading the page will generate a grid panel that looks like the following illustration.

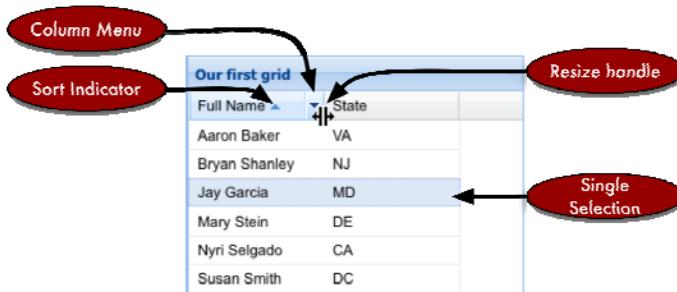


Figure 7.7 Our first grid rendered on screen demonstrating the single-select configured RowSelectionModel and the sortable “Full Name” column.

As we can see in Figure 7.7, the data is not in the same order that we specified. This is because before I took the snapshot, I performed a single click on the “Full Name” column, which invoked the click handler for that column. The click handler checks to see if this column is sortable (which it is) and invoked a DataStore sort method call passing in the data field (“dataIndex”), which is “fullName”. The sort method call then sorts all of the records in the store based on the field that was just passed. It first sorts in an ascending order, then toggles to descending thereafter. A click of the “State” column will result in no sorting because we didn't specify “sort : true” like we did for the “Full Name” column.

To exercise some of the other features of the ColumnModel, you can drag and drop the columns to reorder them, resize them by dragging the resize handle or click the column menu icon, which appears whenever the mouse hovers over a particular column.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

To exercise the Selection Model, simply select a row by clicking on it. Once you've done that, you can use the keyboard to navigate rows by pressing the up and down arrow keys. To exercise the multi-select RowSelectionModel, you can simply modify the SelectionModel by removing the "singleSelect: true" property, which defaults to false. Reloading the page will allow you to select many items by using typical Operating System multi select gestures such as shift click or control click.

Creating our first grid was a cinch. Wasn't it? Obviously there is much more to GridPanels than just displaying data and sorting it. Features like pagination and setting up event handlers for gestures like right mouse clicks are used frequently. These advanced usages are exactly where we're heading next.

## 7.4 Advanced GridPanel construction

In the prior section, we built a GridPanel that used static in-memory data. We instantiated every instance of the supporting classes, which helped us get some exposure to them. Like many of the components in the framework, the GridPanel and its supporting classes have alternate configuration patterns. In building our advanced grid panel, we'll explore some of these alternate patterns in a couple of the supporting classes.

### 7.4.1 What we're building.

The GridPanel we're going to constructing will leverage some advanced concepts, the first of which is using a remote data Store to query against a large data set of randomly generated data, which gives us the opportunity to use a paging toolbar. We will learn how to construct custom renders for two of these columns. One of which will simply apply color to the ID column and the other will be more advanced, concatenating the address data into one column. After we build this grid panel, we're going to circle around and setup a rowdblclick handler as well as get introduced to context menus as we learn to use the GridPanel's rowcontextmenu event. Put on your propeller hat if you have one, we'll be spending the rest of this chapter on this task and will be covering a lot of material.

### 7.4.2 Creating the store using shortcuts

When creating our store, we're going to learn some of the common shortcuts, which will save you time. If you need to customize the configuration beyond what's covered here, you can mix and match shortcuts with long-hand versions of the configuration.

#### **Listing 7.3 Creating an ArrayStore**

```
var recordFields = [
    { name : 'id',           mapping : 'id'          },
    { name : 'firstname',   mapping : 'firstname'   },
    { name : 'lastname',    mapping : 'lastname'    },
    { name : 'street',      mapping : 'street'      },
    { name : 'city',         mapping : 'city'        },
    { name : 'state',        mapping : 'state'        },
    { name : 'zip',          mapping : 'zip'          },
    { name : 'country',     mapping : 'country'     }
];
```

//1

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var remoteJsonStore = new Ext.data.JsonStore({
    fields      : recordFields,
    url         : 'http://tdg-i.com/dataQuery.php',
    totalProperty : 'totalCount',
    root        : 'records',
    id          : 'ourRemoteStore',
    autoLoad    : false,
    remoteSort  : true
}) ;

{1} Creating a list of fields mapped to raw data points.
{2} A shortcut configuration of a remote JSON Data Store

```

In the above code, we are configuring a remote JsonStore using some shortcuts. The first thing we do is create a reference, `recordFields{2}`, which is an array of field configuration objects. In this array, we're mapping a lot of data fields, some of which we'll specify in the column model.

If the field labels map the data point labels, you wanted to further simplify the mappings, you could just specify an array of string values.

```

var recordFields = [
    'id','firstname','lastname','street','city','state','zip','country'
];

```

You could also specify a mixture of objects and strings for the list of fields. When I build applications, I always configure objects instead of strings is because I like to think of the code as self-documenting. Also, if the data point on the backend needs to change, all you need to modify is the mapping attribute compared to having to modify the mapping *and* column model if you were to use just strings.

We then move on to configure our `JsonStore{2}`, which will fetch data remotely. When configuring this store, we set the `fields` property to the reference of the `recordFields` array we just created. Ext uses this `fields` configuration array and use it to automatically create the `data.Record` that it will use to fill the store.

We then move on to pass a `url` property, which is one of the shortcuts we're using. Because we pass this property, the `Store` class will use it to instantiate a `Proxy` to fetch data. Also, being that this is a remote URL, an instance of `ScriptTagProxy` will be used. Remember that the `ScriptTagProxy` requires that the data get passed as the first parameter to the `callback` method that it automatically produces. The following figure illustrates the format that the server must respond with.

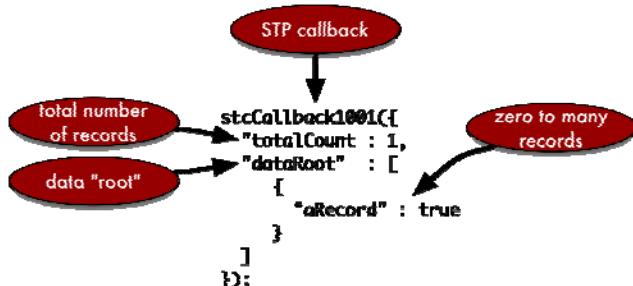


Figure 7.8 The format that a remote server must respond within

In the following illustration, we see that the server returns a method call to `stcCallback1001`. The callback method name the server responds with is passed to the server in the request via a `callback` property during each request. The number will increment for each STP request.

The `totalCount` property is an optional value, which specifies how many records are available for viewing. In configuring our remote `JsonStore`, we specified the `totalProperty` configuration property as `totalCount`. This property will be leveraged by the `PagingToolbar` to calculate how many *pages* of data are available.

The most important property is the `data "root"`, which is the property that contains our array of data. We specified the `root` configuration property as `records` in the remote `JsonStore` configuration.

We then instruct the store not to automatically fetch the data from the data source. We will need to specially craft the first request so we do not fetch all of the records in the database for the query that we're performing.

We also set a static id, `"ourRemoteStore"`, for the `Store`, which we'll use later to get a reference of the store from the `Ext.StoreMgr`, which is to `DataStores` what the `ComponentMgr` is to `Components`. That is, each instance of `DataStore` can have a unique ID assigned to it or will assign one to itself and is registered to the `StoreMgr` singleton upon instantiation. Likewise, deregistration of the store occurs when a store is destroyed.

#### NOTE

We could configure the `JsonStore` using the `XType "jsonstore"`, but because we're binding it to the `GridPanel` *and* the `PagingToolbar`, we must use an actual instance of `Ext.data.Store`.

Lastly, we enable remote sorting by specifying `remoteSort` as the Boolean value of `true`. Because we're paging, sorting locally would cause your UI to behave abnormally as the data sorting and page count would mismatch.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Now that we've got that out of the way, we can move on to configure our advanced ColumnModel.

### 7.4.3 Building a ColumnModel with custom renderers

The ColumnModel we constructed for our first GridPanel was pretty boring. All it did was map the column to the record data field. This new ColumnModel, however, will leverage two custom renderers, one of which will allow us to leverage the Address data Fields to build composite and stylized cells.

#### Listing 7.4 Creating two custom renderers

```
var colorTextBlue = function(id) {
    return '<span style="color: #0000FF;">' + id + '</span>';
}

var stylizeAddress = function(street, column, record) {
    var city = record.get('city');
    var state = record.get('state');
    var zip = record.get('zip');

    return String.format('{0}<br />{1} {2}, {3}', street, city, state, zip );
}
```

In the listing above, we construct two custom renderers (methods) that will be used by two different columns. The first method, `colorTextBlue`, returns a concatenated string that consists of a span tag that wraps the `id` argument being passed to it. The span tag has a CSS style property that will result in blue text.

The second custom renderer, `stylizeAddress` is a much more complex method that will create a composite view of all of the address data available to us minus the country. All custom renderers are called with six arguments. We're using the first and third in this case. The first is the field value that the column is bound to. The second is the column metadata, which we're not using. The third is a reference to the actual data Record, which we'll use heavily.

In this method, we create references to the city and state values of the record by using its `get` method, passing in the field for which we want to retrieve data. This gives us all the references we need to construct our composite data value.

The last thing we do in this method is return the result of the `String.format` method call, which is one of the lesser-known power tools that Ext offers. The first argument is a string that contains integers wrapped in curly braces, which get filled in by the subsequent values passed to the method. Using this method is a nice alternative to the string concatenation we performed above.

Excellent. Our custom renderers are set and we can now proceed to constructing our column configuration. This listing is going to be rather long because we're configuring five columns, which requires quite a bit of configuration parameters. Please stick with me on this. Once you start to see the pattern, reading through this will be rather easy.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

### Listing 7.5 Configuring our advanced column model

```

var columnModel = [
  {
    header      : 'ID',
    dataIndex  : 'id',
    sortable   : true,
    width      : 50,
    resizable  : false,
    hidden     : true,
    renderer   : colorTextBlue
  },
  {
    header      : 'Last Name',
    dataIndex  : 'lastname',
    sortable   : true,
    hideable   : false,
    width      : 75
  },
  {
    header      : 'First Name',
    dataIndex  : 'firstname',
    sortable   : true,
    hideable   : false,
    width      : 75
  },
  {
    header      : 'Address',
    dataIndex  : 'street',
    sortable   : false,
    id         : 'addressCol',
    renderer   : stylizeAddress
  },
  {
    header      : 'Country',
    dataIndex  : 'country',
    sortable   : true,
    width      : 150
  }
];
{1} Hide the ID Column
{2} Bind the colorTextBlue custom renderer to the "ID" column
{3} Bind the stylizeAddress custom renderer to the "Address" column

```

Configuring this column model is much like the configuring the column model for our previous grid. The biggest difference being that instead of instantiating an instance of `Ext.grid.ColumnModel`, we're using the shortcut method by creating an array of objects, which will be translated to a list of `Ext.grid.Columns`. However, we do some things different. For instance, for the "ID" column, is `hidden{1}` and bound to the `colorTextBlue{2}` custom renderer.

We also set both the `hideable` property for the "Last Name" and "First Name" columns to false, which will prevent them from being hidden via the Columns menu. We'll get a chance to see this in action after we render the `GridPanel`.

The "Address" column is a bit special because disable sorting. This is because we are binding the column to the "street" field but are using the `stylizeAddress` custom renderer to provide cells based on a composite of other fields in the record, such as city, state and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

zip. We do, however, enable sorting on each individual column. This column also has an `id` property set to "addressCol" and no width property. This is configured this way because we're going to configure the `GridPanel` to automatically expand this column so that it takes all of the available width after all of the statically sized columns are rendered.

Now that we have constructed the array of `Column` configuration objects, we can move on to piece together our paging `GridPanel`.

#### 7.4.4 Configuring our advanced `GridPanel`

We now have just about all of the pieces required to configure our paging `GridPanel`. In order to do this, however, we will need to first configure the paging toolbar, which will be used as the bottom toolbar or `bbar` in the `GridPanel`.

##### **Listing 7.6 Configuring our advanced column model**

```
var pagingToolbar = { //1
    xtype      : 'paging',
    store      : remoteJsonStore,
    pageSize   : 50,
    displayInfo : true
}

var grid = { //2
    xtype          : 'grid',
    columns        : columnModel,
    store          : remoteJsonStore,
    loadMask       : true,
    bbar           : pagingToolbar,
    autoExpandColumn: 'addressCol'
}
{1} Configuring the PagingToolbar using the "paging" XType
{2} Configuring the GridPanel using the "grid" XType
```

In the previous listing, we use the `XTypes` as a shortcut to configure both the `PagingToolbar` and the `GridPanel`.

For the `PagingToolbar` configuration, we bind the `remoteJsonStore` we configured earlier and set the `pageSize` property to 50. This will enable the `PagingToolbar` to bind to the data Store allowing it to control requests. The `pageSize` property will be sent to the remote server as the `limit` property, and will ensure that the Store receives bundles 50 (or less) records per request. The `PagingToolbar` will leverage this `limit` property along with the servers returning `totalCount` property to calculate how many "pages" there are for the data set. The last configuration property, `displayInfo`, instructs the `p` to display a small block of text, which contains text that displays the current page position and how many records (remember `totalCount`) are available to be flipped through. I'll point this out when we render the `GridPanel`.

We then move on to configure a `GridPanel` `XType` configuration object. In this configuration, we bind the earlier created configuration variables `columnModel`, `remoteJsonStore` and `pagingToolbar`. Because we set the `columns` property, Ext will

automatically generate an instance of `Ext.grid.ColumnModel` based on the array of configuration objects in the `columnModel` variable.

The `loadMask` property is set to true, which will instruct the `GridPanel` to create an instance of `Ext.LoadMask` and bind it to the `bwrap` (body wrap) element, which is the tag that ultimately *wraps* or contains all of the elements below the `titlebar` of a Panel. These elements include the top toolbar, content body and bottom toolbar and `fbar`, which is the bottom button footer bar. The `LoadMask` class binds to various events that the Store publishes to show and hide itself based on situation the store is in. For instance, when the Store initiates a request it will mask the `bwrap` element and when the request completes, it will *unmask* that element.

We then set the `bbar` property to our `pagingToolbar` XType configuration Object, which will render an instance of the `PagingToolbar` widget with that configuration data as the bottom toolbar in the `GridPanel`.

Lastly, we set the `autoExpandColumn` property to the string of 'addressCol', which is the ID of our Address Column, ensuring that this column will be dynamically resized based on all of the available viewport width minus the other fixed width columns.

Our `GridPanel` is now configured and ready to be placed in a Container and rendered. We could render this `GridPanel` to the document body element, but I would like to place it as a child of an instance of `Ext.Window`, this way we can easily resize the `GridPanel` and see how features like the automatic sizing of the Address column works.

#### 7.4.5 Configuring a Container for our GridPanel

We'll now move on to create the Container for our advanced `GridPanel`. Once we render the Container, we're going to initiate the first query for the remote data store we created just a while ago.

##### **Listing 7.7 Placing our GridPanel inside a Window**

```

new Ext.Window({
    height : 350,
    width  : 550,
    border : false,
    layout  : 'fit',
    items   : grid
}).show(); // 1

Ext.StoreMgr.get('ourRemoteStore').load({
    params : {
        start : 0,
        limit : 50
    }
});
{1} Rendering our GridPanel inside of a Window
{2} Using the Ext Store Manager class to cause our store to perform its initial request

```

In Listing 7.7, we perform two tasks. The first of which is the creation of `Ext.Window{1}`, which uses the fit layout and has our `GridPanel` as its only item. Instead of creating a reference to the instance of `Ext.Window` and then calling the `reference.show` method, we use chaining to call the `show` method directly from the result of the constructor call.

Lastly, we use the `Ext.StoreMgr.get` method, passing it our remote store ID string, and again use chaining to call the result's `load` method. We pass an object, which contains a `params` property, which itself is an object specifying start and limit properties.

The `start` property is instructed by the server as to which Record or row number to begin the query. It will then read the `start` plus the `limit` to return a "page" of data. We have to call this `load` method because the `PagingToolbar` does not initiate the first store load request on its own. We have to kind of nudge it a little to get it started.

Our rendered grid panel should look like the one in the following illustration.



Figure 7.9 the results of our advanced paging GridPanel implementation.

As you can see from the fruit of our labor, our `GridPanel`'s `Address` Column displays a composite of the address fields in one neat column that is dynamically sized and cannot be sorted, while all of the other columns start life with a fixed size and can be sorted.

A quick look at the communication from the first request via firebug will show us the parameters sent to the server. The following figure illustrates those parameters.



Figure 7.10 A list of parameters sent to the remote server to request paged data.

We already covered the callback, limit and start parameters a short while ago when we learned about the paging toolbar. What we see new here is the `_dc` and `xaction` parameters.

The `_dc` parameter is what's known as a "Cache Buster" parameter that is unique for every request and contains the timestamp for which the request was made in the UNIX epoch format, which is the number of seconds since the beginning of *Computer* time or 12 AM on January 1, 1970. Because the value for the requests are unique, the request bypasses proxies, thus prevents them from intercepting the request and returning cached data.

The `xaction` parameter is used by `Ext.direct` to instruct the controller on which `action` to execute, which in this case, happens to be the `load` action. The `xaction` parameter is sent with every request generated by stores and can safely be ignored if needed.

I'm not sure if you have detected this already, we have not seen our ID column in action. This is because we configured it as a hidden column. In order to enable it, we can simply leverage the Column menu and check off the id column.

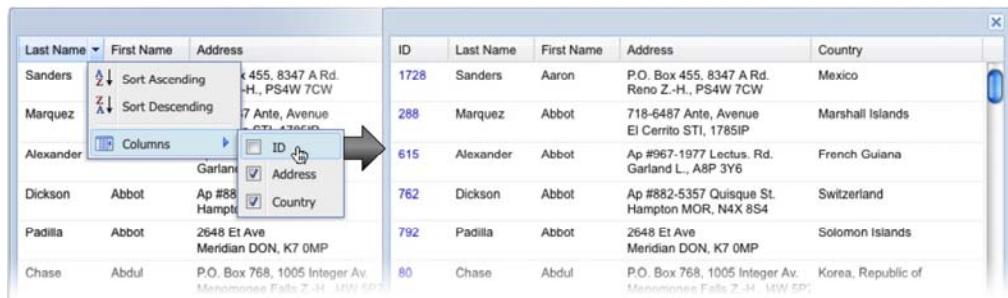


Figure 7.11 Enabling the ID column via the Columns menu.

After checking the ID column in the "Columns" menu, you'll see it appear in the in the `GridView`. In this menu, you can also specify the direction for which a column is to be sorted. One thing you may notice right away is by looking at the Columns menu is that the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

menu options for the "First Name" and "Last Name" columns are missing. This is because we set the hideable flag to false, which prevents their respective menu option from being rendered. The Column menu is also a great way to sort a column directly by the order that you desire.

Cool, we have our GridPanel constructed. We can now configure some event handlers for the GridPanel that will allow us to interact more with it.

#### 7.4.6 Applying event handlers for interaction

In order to create row-based user interaction, you need to bind event handlers to events that are published by the GridPanel. Here, we'll learn how to leverage the `rowdblclick` event to pop up a dialog when a double click gesture is detected on a row. Likewise, we'll listen for a `contextmenu` (right click) event to create and show a single item context menu using the mouse coordinates.

We'll begin by creating a method to format a message for the Ext alert dialog and then move on to create the specific event handlers. We'll insert this code anywhere *before* our GridPanel configuration.

#### Listing 7.8 Creating event handlers for our data grid

```

var doMsgBoxAlert = function(record) { // 1
    var firstName = record.get('firstname');
    var lastName = record.get('lastname');

    var msg = String.format('The record you chose:<br /> {0}, {1}', // 2
        lastName, firstName);

    Ext.MessageBox.alert('', msg);
}

var doRowDblClick = function(thisGrid, rowIndex) {
    var record = thisGrid.getStore().getAt(rowIndex); // 3
    doMsgBoxAlert(record);
}

var doRowCtxMenu = function(thisGrid, rowIndex, evtObj) { // 4
    evtObj.stopEvent();

    thisGrid.getSelectionModel().selectRow(rowIndex);

    if (!thisGrid.rowContextMenu) { // 5
        thisGrid.rowContextMenu = new Ext.menu.Menu({
            items : [
                {text : 'View Record',
                 handler : function() {
                     var record = thisGrid.getStore().getAt(rowIndex);
                     doMsgBoxAlert(record);
                 }
            }
        });
    }

    thisGrid.rowContextMenu.showAt(evtObj.getXY());
}
{1} Create a utility method to generate an Ext alert dialog

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

- {2} The rowdoubleclick event handler
- {3} The rowcontextmenu event handler
- {4} Prevent event, hiding the browser's context menu
- {5} Creating a static instance of an Ext Menu for the GridPanel

In the above listing, we create three methods. The first of which, `doMsgBoxAlert{1}`, is a utility method that accepts a record as its only argument. It leverages the `record.get` method to extract the first and last name fields and uses them to display an Ext alert dialog that contains a message with those two properties.

Next, we create the first handler, `doRowDblClick{2}`, which is configured to accept two of the parameters that the event publishes. The first is the reference to the grid, `thisGrid` and the second is the index of the row, `rowIndex`, for which the event occurred. This handler uses the `rowIndex` that the event occurred on to locate the reference of the row's source Record. It does this by calling `thisGrid.getStore`, which is the `GridPanel`'s store accessor method. Then we use chaining to tack on another method call, `getAt`, to the result of `getStore` and we pass `rowIndex` to it. This is what gets us the actual record reference. This handler then calls `doMsgBoxAlert`, passing that record reference, which will result in the alert dialog being displayed for the row that was double clicked.

The last method, `doRowCtxMenu{3}`, is much more complicated, as it does a lot more than just call `doMsgBoxAlert` and has some complex JavaScript code. If we look at the parameter list of the method, we can see that the list is the same as the other event handler with the addition of `evtObj`, which is an instance of `Ext.EventObject`. Knowing this is important because we need to prevent the browser's own context menu from displaying. This is why calls `evtObj.stopEvent{4}` as the first task. Calling `stopEvent` stop the native browser context menu from showing.

We then move on to select the record based on the `rowIndex` argument. Doing so will provide the necessary feedback to the user. We then test to see if `thisGrid` does not have `rowCtxMenu` property, which on the first execution of this method will be `true` and the interpreter will dive into this branch of code. We do this because we only want to create the menu once if it never existed. If we didn't have this fork in logic, we would be creating menus every time the context menu was called, which would be wasteful.

We then assign the `rowCtxMenu` property `{5}` to `thisGrid` as the result of a new instance of `Ext.menu.Menu`, which has one item, which is written in typical XType shorthand. The first property of the single menu item is the text that will be displayed when the menu item is shown. The other is a handler method that is defined inline and causes `doMsgBoxAlert` to be called with the referenced record.

The last bit of code calls upon the newly created `rowCtxMenu`'s `showAt` method in which requires the X and Y coordinates to display the menu. We do this by directly passing the results of the `evtObj.getXY()` to the `showAt` method. The `EventObject.getXY` will return the exact coordinates that the event was generated at.

OK. Our event handlers are now armed and ready to be called upon. Before we can use them in the grid, we need to configure them as listeners.

### **Listing 7.9 Attaching our event handlers to our grid**

```
var grid = {
    xtype          : 'grid',
    columns        : columnModel,
    store          : remoteJsonStore,
    loadMask       : true,
    bbar           : pagingToolbar,
    autoExpandColumn: 'addressCol',
    stripeRows     : true,
    listeners       : {
        rowdblclick   : doRowDoubleClick,
        rowcontextmenu: doRowCtxMenu
    }
}
{1} Attaching the event handlers to our grid
```

To configure the event handlers to the grid, we simply add a `listeners`{1} configuration object, with the event to handler mapping and that's it. Refresh the page and generate some double click and right click gestures on the grid. What happens?

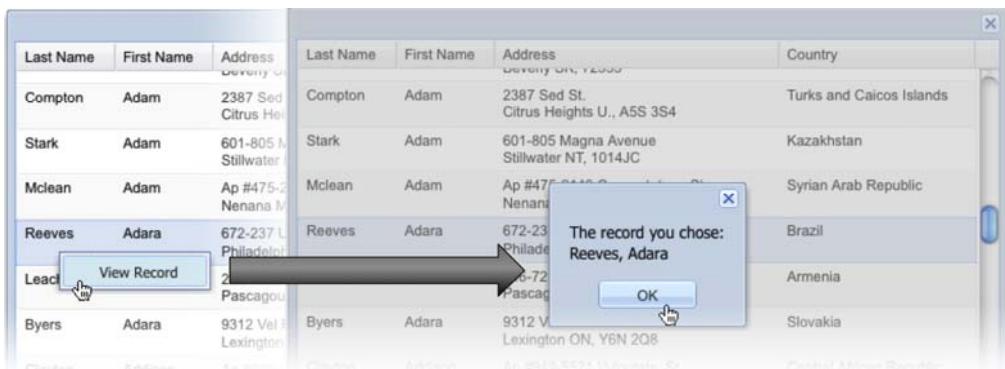


Figure 7.12 The results of our context menu handler addition to our advanced grid.

Now double clicking on any record will cause the Ext alert dialog to appear. Likewise, right clicking a row will cause our custom context menu to appear. If you click on the 'View Record' menu item, the Ext alert dialog will then appear.

Adding user interaction to a grid can be as simple as that. One key to effective development of UI interactions is not to only instantiate and only render widgets once and when needed, as we did with the context menu. While this technique works to prevent duplicate items, it falls short of cleanup. Remember the destruction portion of the

Component lifecycle? We can attach a quick method to destroy the context menu when the grid panel is destroyed by adding a destroy handler method to list of listeners.

```
listeners : {
    rowdblclick : doRowDblClick,
    rowcontextmenu : doRowCtxMenu,
    destroy : function(thisGrid) {
        if (thisGrid.rowCtxMenu) {
            thisGrid.rowCtxMenu.destroy();
        }
    }
}
```

In the above code snippet, we add the destroy event handler inline instead of creating a separate referenced method for it. The destroy event always passes the component for which is publishing the event, which we labeled thisGrid. In that method, we test for the existence of the rowCtxMenu variable. If this item exists, we call its destroy method.

Context menu cleanup is one of those topics that developers often miss and can lead to lots of useless leftover DOM node garbage, which chews up memory and can contribute to over-all application performance degradation over time. So if you're attaching context menus to any component, always be sure to register a destroy event handler for that component that destroys any existing context menus.

## Summary

In this chapter, we learned quite a bit about the `GridPanel` and the data Store classes. We started by constructing a local data-feeding `GridPanel` and learned about the both the supporting classes for both the data Store and `GridPanel`.

While building our first `GridPanel`, got to see how the data Store uses proxies to read data, a reader to parse it and spools up Records, which are Ext-managed data objects. We also learned how the `GridView` knows when to render data from the Store by listening to events.

When we constructed our remote-loading `GridPanel`, we learned about some of the shortcuts that can be used to configure the `GridPanel` and many of its supporting classes. we learned more about the `ColumnModel` and how it can have hidden columns or columns that cannot be hidden. While doing this, we configured the JSON reading data Store that allows for remote sorting as well.

Lastly, we added grid interactions to the `GridPanel`, where mouse double click and right click gestures were captured and resulted in the UI responding. In doing this, we got a quick glance at menus and learned the importance of cleanup of menu items after their parent component is destroyed.

Many of the concepts that we learned in this chapter will carry forward when we learn how to use the `EditorGridPanel` and its descendant, the `PropertyGrid`.

# 8

## *EditorGridPanel*

In the last chapter, we learned about the `GridPanel` and how it can be used to display data. We integrated a `PagingToolbar` to our complex `GridPanel` to allow us to flip through large paginated data.

In this chapter, we'll build upon our previous work to construct a working `EditorGridPanel`, which will allow you to modify data inline much like you can in popular desktop spreadsheet applications like Microsoft Excel. We'll also learn how to integrate context menus and toolbar buttons with Ajax requests for CRUD operations.

We'll start out by creating our first `EditorGridPanel`, where you'll get an introduction to what an `EditorGridPanel` is and how edits are even possible. We'll discuss ins and outs of setting up UI widgets for interaction to support insert and delete CRUD operations as well as how to get modified records from the store or even reject changes. Building up an `EditorGridPanel` without saving data is useless, so we're going to take advantage of this opportunity to learn how we can code for CRUD operations.

Afterwards we're going to integrate `Ext.data.DataWriter` and learn how it can save us time by doing a lot of the heavy lifting and reducing the code that we need to write.

This is going to be one of the most intense chapters yet.

### **8.1 A close look at the `EditorGridPanel`**

The `EditorGridPanel` panel is a class that builds upon the `GridPanel` class that works with the `ColumnModel` to allow the use of Ext form fields to edit data on the fly without the use of a separate `FormPanel`. It uses the Cell selection model by default, and has the an internal event model to detect user input such as clicks and keyboard keystrokes to trigger the cell selection and even the rendering or repositioning of the editor fields. It can leverage

the `Ext.data.DataWriter` class to automatically save data after it's been edited, which we'll explore later in this chapter.

Excited yet? Before we break ground, we should discuss what we're going to be constructing.

We're going to expand upon the complex `GridPanel` we constructed in the last chapter, but with some changes to allow us to edit data. The first obvious change is the use of the `EditorGridPanel` instead of the `GridPanel` widget.

Another change we're going to perform is the split up of the composite Address Column into separate columns for each `dataIndex`, which allows us to easily edit the separate address information instead of having to craft a complex editor. Figure 8.1 illustrates what our `EditorGridPanel` panel will look like.

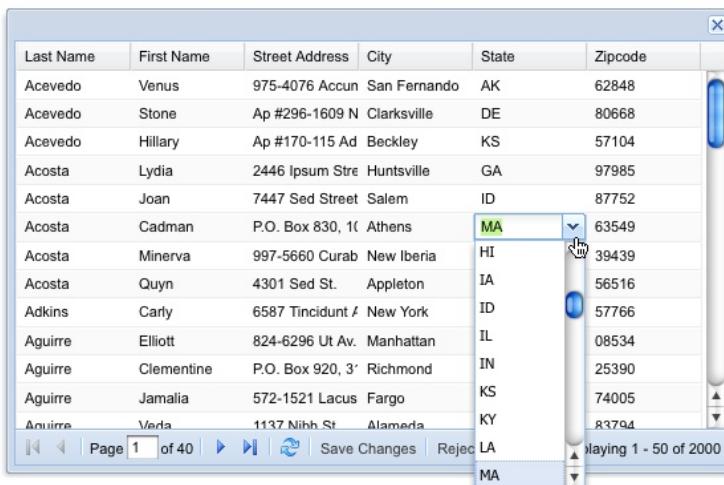


Figure 8.1 A quick view of what we're going to be constructing.

In building this `EditorGridPanel`, we're going to leverage the necessary UI components to construct something that could be considered a mini application, as it will have simulated CRUD (Create, Update and Delete) capabilities for records. The reason it is simulated is because we cannot make Ajax requests to a different domain, and in order to keep server-side code out of the picture, we're going to need to create some static responses.

To enable CURD, we'll add two buttons to the `PagingToolbar` that will allow us to save or reject changes. Likewise, we'll create a usable context menu to add or delete rows based on which cell was right-clicked. This will absolutely be the most complex code thus far, and we'll perform this in phases.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

The first of which is getting the `EditorGridPanel` functioning. We'll then circle back and add a slice of each CRUD action one at a time.

## 8.2 Building our first `EditorGridPanel`

Because we're expanding upon the complex `GridPanel` we built last chapter, we're going to see some of the exact same code and patterns. The reason we're doing it this way is so the flow of the code is as smooth as possible. In most cases, there will be changes, so please take the time to read through every bit. I'll point out all of the pertinent changes.

Before we begin, we need to create two files that contain some JSON that will enable the simulation of server-side responses to calls we're going to be making for the CRUD actions. The first of which will be called `successTrue.js` and will contain the following:

```
{success : true}
```

The second one will be called `successFalse.js` and will contain:

```
{success: false}
```

Please save these in your project space on your development web server. Now that we have that out of the way, we can now begin to construct our store.

### **Listing 8.1 Creating our remote store**

```
var remoteProxy = new Ext.data.ScriptTagProxy({
    url : 'http://tdg-i.com/dataQuery.php'
});

var recordFields = [
    { name : 'id', mapping : 'id' },
    { name : 'firstname', mapping : 'firstname' },
    { name : 'lastname', mapping : 'lastname' },
    { name : 'street', mapping : 'street' },
    { name : 'city', mapping : 'city' },
    { name : 'state', mapping : 'state' },
    { name : 'zipcode', mapping : 'zip' },
    { name : 'newRecordId', mapping : 'newRecordId' }
];

var remoteJsonStore = new Ext.data.JsonStore({
    proxy : remoteProxy,
    storeId : 'ourRemoteStore',
    root : 'records',
    autoLoad : false,
    totalProperty : 'totalCount',
    remoteSort : true,
    fields : recordFields,
    idProperty : 'id'
});
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

We begin by creating our `ScriptTagProxy`, which will allow us to fetch data from a remote domain. We then move on to create a list of fields, which map to the data points in the raw data. Remember, these are used to create actual instances of `Ext.data.field`, which are the smallest part of an `Ext.data.Record`. Notice the last field is added to the list. Our fictitious server side controller will use this 'newRecordId' for insert operations. We'll discuss more about this later on when we talk about saving inserted records and see why only new records will have this property.

Lastly, we create our `remoteJsonStore`, which uses our `remoteProxy` and `recordFields`. We *could have* set `remoteJsonStore` as an XType configuration object, but we're going to need that reference later on when we create some of the handlers for our CRUD actions. This will keep things simple for us down the road, I promise. For the `JsonStore`, we also set the `idProperty` to 'id', which will ensure that the data Store tracks IDs, allowing us to monitor insert operations down the road.

Our next step is to create the field editors that will be used for the `ColumnModel` later on.

### **Listing 8.2 Creating our remote store**

```
var textFieldEditor = new Ext.form.TextField(); // 1
var comboEditor = {
    xtype : 'combo',
    triggerAction : 'all',
    displayField : 'state',
    valueField : 'state',
    store : {
        xtype : 'jsonstore',
        root : 'records',
        fields : ['state'],
        proxy : new Ext.data.ScriptTagProxy({
            url : 'http://tdg-i.com/getStates.php'
        })
    }
}
var numberFieldEditor = { // 3
    xtype : 'numberfield',
    minLength : 5,
    maxLength : 5
}
{1} Creating the TextField editor
{2} An xtype configuration for a ComboBox Editor
{3} A NumberField configuration object
```

When creating our editors, we use two techniques; direct and lazy instantiation. We directly instantiate an instance of `Ext.form.TextField`{1} because we are going to be using it multiple times, and it would be wasteful to use an XType configuration. Conversely, the `comboEditor`{2} and `numberFieldEditor`{3} use XType configurations because they will only be used a for a single column. The `comboEditor` is an XType configuration for the `Ext.Form.ComboBox` and has a nested XType configuration for an `Ext.data.JsonStore`,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

which uses a ScriptTagProxy so we can get a list of states from the remote database. Likewise, the numberFieldEditor is an XType configuration for the Ext.form.NumberField class. We're using this in the zip code column so we set two basic validation rules that dictate the minimum and maximum character lengths for the field. Because the editor is a NumberField, no alpha characters can be entered, and only values with five integers will be accepted.

We can now begin creating the ColumnModel, where we'll use the editors we just configured. As in our complex GridPanel, this listing will be relatively lengthy. The patterns should be obvious, however.

### **Listing 8.3 Creating ColumnModel**

```
var columnModel = [
    {
        header : 'Last Name',
        dataIndex : 'lastname',
        sortable : true,
        editor : textFieldEditor // 1
    },
    {
        header : 'First Name',
        dataIndex : 'firstname',
        sortable : true,
        editor : textFieldEditor
    },
    {
        header : 'Street Address',
        dataIndex : 'street',
        sortable : true,
        editor : textFieldEditor
    },
    {
        header : 'City',
        dataIndex : 'city',
        sortable : true,
        editor : textFieldEditor
    },
    {
        header : 'State',
        dataIndex : 'state',
        sortable : true,
        editor : comboBoxEditor // 2
    },
    {
        header : 'Zip Code',
        dataIndex : 'zipcode',
        sortable : true,
        editor : numberFieldEditor // 3
    }
];
{1} Using our TextField editor
{2} Specifying the ComboBox configuration object
{3} Setting the numberFieldEditor to the column
```

As we review the ColumnModel configuration array we see the already familiar properties such as header, dataIndex and sortable. We also see a new kid on the block, editor, which allows us to specify an editor for each of the Columns.

Notice that the textFieldEditor{1} is used in four of the six Column configuration objects. The main reason we did this is because of performance. Instead of using an XType configuration object and having one instance of Ext.form.TextField instantiated for each column, the single TextField class is merely rendered and positioned where it's needed, which saves memory and reduces DOM bloat. Consider this a performance saving technique. We'll see this in action when we get around to rendering our EditorGridPanel.

Lastly we have the comboEditor used for the state Column and the numberFieldEditor used for the Zip Code column. Remember that since these are only used once, using an XType configuration object is Okay.

Cool, we now have our Store, editors and ColumnModel configured. We can now move on to creating our PagingToolbar and EditorGridPanel.

#### **Listing 8.4 Creating The PagingToolbar and EditorGridPanel**

```
var pagingToolbar = {
    xtype      : 'paging',
    store      : remoteJsonStore,
    pageSize   : 50,
    displayInfo: true
}

var grid = {
    xtype      : 'editorgrid', // 1
    columns   : columnModel,
    id        : 'myEditorGrid',
    store     : remoteJsonStore,
    loadMask  : true,
    bbar      : pagingToolbar,
    stripeRows: true,
    viewConfig: {
        forceFit: true
    }
}

new Ext.Window({
    height  : 350,
    width   : 550,
    border  : false,
    layout   : 'fit',
    items    : grid
}).show();

remoteJsonStore.load({ // 2
    params : {
        start : 0,
        limit : 50
    }
});
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

**{1} Specifying the 'editorgrid' xtype property  
{2} Loading our store after**

In the code above, we create the rest of our `EditorGridPanel`, starting with the `PagingToolbar`, which uses our `remoteJsonStore` and has the `pageSize` set to 50 records. Next, we create our `EditorGridPanel`{1}, which has its `xtype` property set to 'editorgrid' and uses our `columnModel`, `remoteJsonStore` and `pagingToolbar`.

We then move on to create the container for our `EditorGridPanel`, which is an instance of `Ext.Window`, and has its layout set to 'fit'. We use chaining to show the Window immediately after its instantiated.

Lastly, we call `remoteJsonStore.load` and pass a configuration object that specifies the parameters to send to the server. This ensures we start at record 0 and limits the number of returning records to 50.

Cool, all of the pieces of our puzzle are together for this phase. We can now render our `EditorGridPanel` and begin to edit data.

Last Name	First Name	Street Address	City	State	Zip Code
Acevedo	Venus	975-4076 Accun	San Fernando	AK	62848
Acevedo	Stone	Ap #296-1609 N	Clarksville	DE	80668
Acevedo	Hillary	Ap #170-115 Ad	Beckley	KS	57104
Acosta	Lylia	2446 Ipsum Stre	Huntsville	GA	97985
Acosta	Joan	7447 Sed Street	Salem	ID	87752
Acosta	Cadman	P.O. Box 830, 1C	Athens	MA	63549
Acosta	Minerva	997-5660 Curab	New Iberia	LA	39439
Acosta	Quyn	4301 Sed St.	Appleton	WY	56516
Adkins	Carly	6587 Tincidunt /	New York	CT	57766
Aguirre	Elliott	824-6296 Ut Av.	Manhattan	CT	08534
Aguirre	Clementine	P.O. Box 920, 3'	Richmond	MS	25390
Aguirre	Jamalia	572-1521 Lacus	Fargo	RI	74005
Aguirre	Verda	1137 Nihil St.	Alameda	MA	83794

Figure 8.2 Our first `EditorGridPanel` in action.

We can see that our `EditorGridPanel` and `PagingToolbar` have rendered with data just waiting to be modified. Initially it seems like a normal `GridPanel`. But under the covers, lies a whole new level of functionality just waiting to be unleashed. We should take a quick moment to discuss how exactly you can use it.

### 8.3 Navigating our `EditorGridPanel`

You can use mouse or keyboard gestures to navigate through the cells and enter or leave editing mode.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

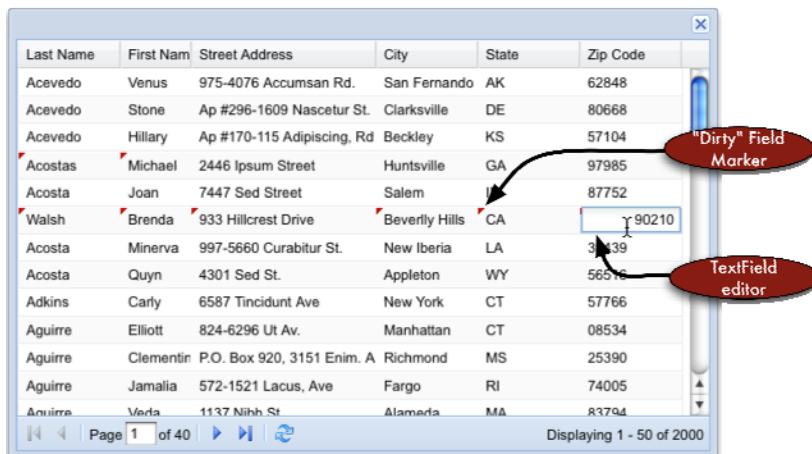
To initiate editing mode via the mouse, simply double click on a cell and the editor will appear, like in Figure 8.2. You can then modify the data and click or double click another cell or anywhere else on the page to cause the blur of the editor to occur. Simply repeat this process to update as many cells as you wish.

You can modify how many clicks it takes to edit a cell by adding a `clicksToEdit` property to the `EditorGridPanel` configuration object and specify an integer value. Some applications like editing via a single click of a cell, so we would need to simply set `clicksToEdit` to 1 and we're done.

Being a command line junkie, I feel that the keyboard navigation, offers you much more power than the mouse. If you're a power user of Excel or a similar spreadsheet application, you know what I'm talking about. To initiate keyboard navigation, I like to use the mouse to focus on the first cell I want to edit. This immediately places focus on exactly where I need it. I can use the Tab key or Shift + Tab key combination to move left or right. I can also use the arrow keys to focus any cell at will.

To enter "edit mode" using the keyboard, I hit the Enter key, which displays the editor for that cell. While in Edit mode, I can elect to stay in edit mode and can modify adjacent cells by hitting the Tab key to move one cell to the right or Shift + Tab to move one cell to the left.

To exit edit mode, I can elect to hit the Enter key again, or hit the Escape key. If the data you entered or modified validates properly, the record will be modified and the field will be marked as dirty. We can see quite a few fields being modified in the illustration below.



The screenshot shows a grid panel with columns for Last Name, First Name, Street Address, City, State, and Zip Code. A red oval labeled "Dirty" Field Marker points to a cell in the Zip Code column that has been edited. A red oval labeled TextField editor points to a floating text input box containing the value "90210". The grid has 15 rows of data. At the bottom, there are navigation buttons (back, forward, search) and a status bar indicating "Page 1 of 40" and "Displaying 1 - 50 of 2000".

Last Name	First Name	Street Address	City	State	Zip Code
Acevedo	Venus	975-4076 Accumsan Rd.	San Fernando	AK	62848
Acevedo	Stone	Ap #296-1609 Nascetur St.	Clarksville	DE	80668
Acevedo	Hillary	Ap #170-115 Adipiscing, Rd	Beckley	KS	57104
Acostas	Michael	2446 Ipsum Street	Huntsville	GA	97985
Acosta	Joan	7447 Sed Street	Salem		87752
Walsh	Brenda	933 Hillcrest Drive	Beverly Hills	CA	90210
Acosta	Minerva	997-5660 Curabitur St.	New Iberia	LA	33339
Acosta	Quyn	4301 Sed St.	Appleton	WY	56515
Adkins	Carly	6587 Tincidunt Ave	New York	CT	57766
Aguirre	Elliott	824-6296 Ut Av.	Manhattan	CT	08534
Aguirre	Clementin	P.O. Box 920, 3151 Enim. A	Richmond	MS	25390
Aguirre	Jamalia	572-1521 Lacus, Ave	Fargo	RI	74005
Aguirre	Verda	1137 Nihil St.	Alameda	MA	81794

Figure 8.3 Our first `EditorGridPanel` with an editor showing and dirty field markers.

When exiting an editor, depending if the field has a validator and the results of the validation, the data will be discarded. To test this, edit a Zip Code cell and enter more or less than five integers. Then exit edit mode by hitting Enter or Escape.

OK, we can now edit data, but the edits are useless unless we actually save our changes. This is where we enter the next building phase, adding the CRUD layers.

## 8.4 Getting the CRUD in

With `EditorGridPanel`, CRUD server requests can be either automatically or manually fired. Automatic requests take place whenever a record is modified or when modifications occur and preset timer expires in the client side logic, firing off a request to the server. To setup automatic CRUD, we can create our own logic to send the requests or we can do things the easy way and use the `Ext.data.DataWriter` class, which is exactly what we'll do later on in this chapter.

For now, we'll focus on manual CRUD, which is when the user invokes an action via UI, such as clicking on a menu Item or a Button somewhere. The reason we are focusing on manual CRUD is because even though `Ext.data.DataWriter` is helpful, it simply will not satisfy everyone's needs and going through this exercise will give you valuable exposure to what's going on with the `Store` and `Records` when data is modified. Also, because Create and Delete are part of CRUD actions, we will explore how to insert and delete records from the `Store`.

### 8.4.1 Adding Save and Reject logic

We will begin by creating the save and change rejection methods, which we'll tie into buttons that will live in our `PagingToolbar`. This code is quite complicated, so please bear with me. We're going to have an in-depth discussion about it, in which I'll walk you through just about every bit.

#### **Listing 8.5 Setting up our save and change rejection handlers**

```
var onSave = function() {
    var modified = remoteJsonStore.getModifiedRecords(); // 1
    if (modified.length > 0) {
        var recordsToSend = [];
        Ext.each(modified, function(record) { // 2
            recordsToSend.push(record.data);
        });
        var grid = Ext.getCmp('myEditorGrid');
        grid.el.mask('Updating', 'x-mask-loading'); // 3
        grid.stopEditing();
        recordsToSend = Ext.encode(recordsToSend); // 4
        Ext.Ajax.request({ // 5
            url : 'successTrue.js',
            params : {
                recordsToInsertUpdate : recordsToSend
            }
        });
    }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        },
        success : function(response) {
            grid.el.unmask();
            remoteJsonStore.commitChanges();
        }
    });
}

var onRejectChanges = function() {
    remoteJsonStore.rejectChanges();
}

{1} Getting a list of modified records
{2} Gathering the record data into a list to send
{3} Manually masking the grid's element
{4} Encoding the record JSON to be sent over the wire
{5} Sending the Ajax request

```

In the code above, we have two methods, `onSave`, which will be called when the “Save Changes” button is pressed. The other is called `onRejectChanges`, which will be called when the “Reject Changes” button is pressed.

`onSave` contains quite a few lines to achieve our goal of data updates by means of Ajax requests. It begins with retrieving a list of modified records by calling the `remoteJsonStore`'s `getModifiedRecords{1}` method, which returns a list of Record instances. In essence, whenever a Record Field is modified, it is marked as *dirty* and placed into the Store's modified list, which the `getModifiedRecords` returns. We test to see if the length of the returned array is greater than zero, which, of course, indicates that we have data to save. We then move on to create an empty array, `recordsToSend`, which we populate by looping through the modified records using `Ext.each`. When calling `Ext.each`, we pass two parameters, the modified records list and an anonymous method, which `Ext.each` will call for each item in the modified list. The anonymous method takes a single parameter, `record`, which changes with each item in the modified list. Inside the anonymous method, we push the `record.data` reference, which is an object that contains every data point in the record.

We then move on to mask the `EditorGridPanel`'s element via the `mask` method. We pass two properties to the `mask` method, the first is a message to display while the mask is visible and the second is CSS class that Ext JS uses to show a “spinner” loading graphic.

The `recordsToSend` reference is then overwritten with the result of the `Ext.encode` method call, for which we pass the original `recordsToSend` list. What this does is “stringify” our list of JSON objects so we can send it over the wire.

### NOTE

`Ext.encode` is a shortcut for `Ext.util.JSON.encode`. Its counterpart is `Ext.decode` or `Ext.util.JSON.decode`. The `Ext.util.JSON` class is a modified version of Douglas Crockford's JSON parser, but does not modify the Object prototype. Please see <http://www.json.org/js.html> for more details on Douglas Crockford's JSON parser.

Next, we actually perform our AJAX request, which is where the fun really begins. We pass `Ext.Ajax.request` a configuration object that has three properties to get the job done.

The first is the `url` for which we have our `successTrue.js` dummy response file. Naturally, there would be some business logic here on the server side to insert or update records. This dummy file assumes the role of a central controller for all of our CRUD operations.

The second property, `params`, is an object, which contains the `recordsToInsertUpdate` property with its value set to the “stringified” JSON, `recordsToSend`. Setting the `params` object ensures that the XHR being lobbed over at the server has parameters sent to it, which in this case will be one. The last property of the `Ajax.request` single parameter is `success`, which is a method to be called *if* the server returns successful status codes, such as 200 and so on.

Because this is a simulation, we don’t check the response back from the request. In here, you generally would have some type of business logic to do something based on the results returned. For now, we simply unmask the grid’s element and call the `remoteJsonStore`’s `commitChanges` method.

This is a very important step because it clears the dirty flag from the Records and Fields that were modified, which clears the dirty flag on the modified cells within the `EditorGridPanel` panel. Failing to do this after a successful submission will result fields not being cleared in the UI and the modified Records not being purged from the Store’s internal modified list.

The last method, `onRejectChanges` simply calls the `remoteJsonStore`’s `rejectChanges` method, which reverts the data back to the original values in the Fields and clears the dirty flag from the UI.

Cool, we have the supporting methods for save and rejection all setup. We can move on to modifying our `PagingToolbar` to include the two buttons that will call our methods above.

#### **Listing 8.6 Reconfiguring the PagingToolbar to include save and reject buttons**

```
var pagingToolbar = {
    xtype      : 'paging',
    store      : remoteJsonStore,
    pageSize   : 50,
    displayInfo: true,
    items      : [
        {
            text      : 'Save Changes',
            handler   : onSave
        },
        {
            text      : 'Reject Changes',
            handler   : onRejectChanges
        }
    ]
} // 1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

}

## {1} Adding a vertical line spacer for a cleaner UI.

In the listing above we reconfigure the PagingToolbar XType configuration object to include items{1}, which consists of five entities. The string entities that you see with the hyphens ("–") are shorthand's for the Ext.Toolbar.Separator, which will place a tiny vertical bar between toolbar child items. We're doing this because we want to show some separation between the buttons and the generic PagingToolbar navigational items.

Also in the list are generic objects, which are translated to instances of Ext.Toolbar.Button. Here, we have our "Save Changes" and "Reject" changes, which have their respective handlers set.

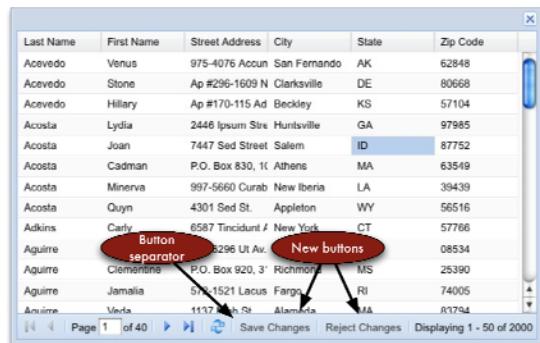


Figure 8.4 Our editor Grid Panel with our save and reject buttons added.

As we can see in the figure 8.4, the save and reject buttons are placed neatly inside of the PagingToolbar's center, where empty space normally resides and the buttons are separated by neat button separators. We can now begin to edit data and exercise our newly modified PagingToolbar functionality and newly created CRUD methods.

#### 8.4.2 Saving or Rejecting our changes

To use our save and reject buttons, we need to modify data first. Leveraging what we know about the using the EditorGridPanel, change some data and press the "Save Changes" Button. We should see the EditorGridPanel's element mask appear for a brief second and disappear after the save completes and the cells that are marked as "dirty" are marked "clean" or committed. The illustration below shows the masking in action.

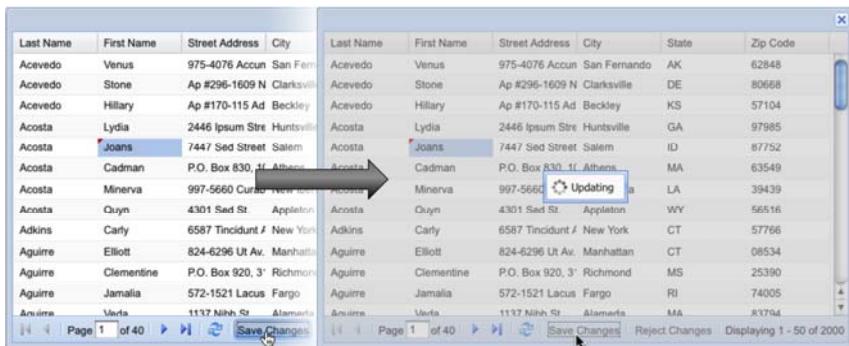


Figure 8.5 The load mask shows when save requests are being sent to the server

Remember that our `onSave` method retrieved a list of modified records and used the `Ext.encode` method to convert the raw list of JavaScript objects to “stringified” JSON. It then used that JSON and POSTed it as the `records` parameter sent by the Ajax request. Below are what the request and the POST parameters look like in FireBug’s XHR inspection view.



Figure 8.6 Inspecting our Ajax request post parameter in FireBug

As we can see in the FireBug request inspection view, the `records` parameter is an array of items. In the case above, it’s two JavaScript objects, which represent the actual data records. Whenever developing or debugging, always remember that you can inspect the POST and GET parameters being sent to the web server via firebug. Also, sometimes JSON blobs can get absolutely enormous and extremely hard to read. What I typically do is copy the JSON from firebug and paste it in the web form at <http://JSONLint.com>, which tabs and formats the data so it’s readable.

If we were to have actual code on the server side to handle this update request, we would read the `records` parameter and decode the JSON and test to see if its an instance of an Array. If so, we’d loop through each object inside of that array and search the database for the ID that is presented in the object instance. If the ID is in the database, we’d code the proper SQL update. Once the updates have occurred, we return a JSON object with `{success:true}` and an optional `msg` or `errors` property, which could contain a message from the server or a list of IDs for which the records could not be updated due to some business rules. We could then use the `success` or `failure` callback handlers,

that we send to the `Ajax.request` method, to inspect what's sent back from the server and perform commits or post error messages accordingly accordingly.

The last bit we need to discuss regarding saving modifications has to do with sorting and pagination. Remember that changing data in a sorted column throws off the sorting completely. What I typically do after a successful change in a sorted column is call the Store's `reload` method, which requests a new copy of the data set from the server and fixes sorting in the UI. Remember that we're simulating a successful server side save, which is why we don't reload the store in the Ajax request's success handler.

OK, we've saved our data and saw what it looks like going over the wire. We have yet to rejected changes though. Lets see what happens when we reject changes. To test this, simply modify data and press the "Reject Changes" button. What happened? Remember that the handler simply called the remote Store's `rejectChanges` method, which looks at each Record in its modified list and calls its `reject` method. This, in turn, clears the dirty flag both on the Record and the UI. That's it, no magic.

Now that we've seen what it takes to perform remote saves to modified records, we're going to add Create and Delete functionality our `EditableGrid`, which will complete our CRUD actions.

### 8.4.3 Adding Create and Delete

When configuring the UI for our save and reject functionality, we added buttons to the `PagingToolbar`. While we could add the Create and Delete functionality in the same way, its best we use a context menu because it's a much smoother flow to delete and add from a context menu. Think about it for a second. If you ever used a spreadsheet application, right clicking on a cell brings up a context menu that, among other things, has "Insert" and "Delete" menu items. We're going to introduce the same paradigm here.

As we did with the previously added functionality, we're going to develop the supporting methods before we construct and configure the UI components. We're going to ratchet up the complexity.

#### **Listing 8.7 Constructing our delete and new record methods**

```
var doDelete = function(rowToDelete) { // 1
    var grid = Ext.getCmp('myEditorGrid');
    var recordToDelete = grid.store.getAt(rowToDelete);

    if (recordToDelete.phantom) { // 2
        grid.store.remove(recordToDelete);
        return;
    }

    grid.el.mask('Updating', 'x-mask-loading');

    Ext.Ajax.request({ // 3
        url : 'successTrue.js',
        parameters : {
            rowToDelete : recordToDelete.id
        },
        success : function() {

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        grid.el.unmask();
        grid.store.remove(recordToDelete);
    });
}

var onDelete = function() { // 4
    var grid      = Ext.getCmp('myEditorGrid');
    var selected = grid.getSelectionModel().getSelectedCell(); // 5

    Ext.MessageBox.confirm(
        'Confirm delete',
        'Are you sure?',
        function(btn) {
            if (btn == 'yes') {
                doDelete(selected[0]);
            }
        }
    );
}

{1} The method responsible for delete operations
{2} Immediately delete records if they are phantoms
{3} Perform the Ajax request to delete real records
{4} The delete menu item handler
{5} Get confirmation from the user, then call the doAjaxReqForDelete
{6} The new record menu item handler

```

I'm hoping that you have not run away by now, but chances are that if you're reading this right now, you're still with me. Awesome. Some of this will look familiar to you from the save feature we added to our `EditorGridPanel` panel earlier.

The first method in our listing, `doDelete{1}`, is actually going to be called by the Delete menu item handler that we create just after it. Its only argument is `rowToDelete`, which is an integer indicating the index of the record that we're to delete. This is the method that is responsible for removing Records from the store. Here's how it works.

Basically what this method does is first get a reference to the `EditorGridPanel` panel via the `ComponentMgr`'s `get` method. It immediately then gets the Record that the Store is to remove using the `Store`'s `getAt` method and passes the `rowToDelete` argument. Then, it checks to see if the record is a phantom (new Record) or not.

If it is, the Record is immediately removed{2} from the Store and this method is aborted with a `return` call. When the record is removed from the Store, the `GridView` immediately shows the change by removing the Record's row in the DOM.

If the record is not a phantom, the grid's element is masked, preventing any further user interaction and provides feedback that something is taking place. `Ajax.request{3}` is then called to our server side simulation file, `successTrue.js` with the single parameter `rowToDelete`, which is the ID of the record in the database. The success handler of this Ajax request will unmask the element and remove the Record from the Store.

The `onDelete` handler<sup>{4}</sup> method is going to query the selected cell from the selection model and request a confirmation from the user. If the user presses the yes button, it will call the `doDelete` method. Here's how it works.

When `onDelete` is first called, it gets a reference to our `EditorGridPanel` via the `Ext.getCmp` method. It then gets the selected cell via calling the `EditorGridPanel`'s `SelectionModel.getSelecteteCell` method. What `getSelectedCell` returns is an array with two values, which are the coordinates of the cell; the row and column number.

A call to `Ext.MessageBox.confirm`<sup>{5}</sup> is made, passing in three arguments; title, message body and button handler, which is an anonymous method. The button handler determines if the button pressed was 'yes' and call our `doDelete` method, passing the first value of the cell coordinates, which is the row of the selected cell.

Before we move on to exercise delete, we should add the insert handler. This one is relatively small.

```
var onInsertRecord = function() { // 6
    var newRecord      = new remoteJsonStore.recordType({
        newRecordId : Ext.id()
    });
    var grid          = Ext.getCmp('myEditorGrid');
    var selectedCell  = grid.getSelectionModel().getSelectedCell();
    var selectedRowIndex = selectedCell[0];

    remoteJsonStore.insert(selectedRowIndex, newRecord);
    grid.startEditing(selectedRowIndex, 0);
}
```

The purpose of this method is to locate the row index that was right clicked and insert a phantom record at the index. Here's how it works.

The very first thing it does is create a `newRecord` via a call to `new remoteJsonStore.recordType`. It does this by passing an object with a single property, `newRecordId`, which is a unique value by virtue of the `Ext.id` utility method call. Having this unique `newRecordId` will aid the server side in inserting new records and returning a mapping for the client to register real ids for each of the new records. We'll discuss this more a little later, when we explore what the server side could be doing with the data we're submitting.

All data Stores have the default Record template accessible via the `recordType` property. Remember that to instantiate a new instance of a record, you must use the `new` keyword.

Next, we create a reference, `rowInsertIndex`, to the row of the newly selected cell. We do this because it ends up in easier to read code when we use it in the following two statements.

A call is then made to the `remoteJsonStore`'s `insert` method, which requires two parameters. The first is the index for which we wish to insert the Record and the second is a reference to an actual Record. This effectively inserts a record above the row that is right clicked, emulating one the spreadsheet features we discussed earlier.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Lastly, we want to initiate editing of that record immediately. We accomplish this by a call to the `EditorGridPanel`'s `startEditing` method, passing it the row for which we inserted the new record and 0, which means the first column.

This concludes the supporting methods for create and delete functions. We can now move on to create the context menu handler and reconfigure the grid to listen to the `cellcontextmenu` event.

#### **Listing 8.8 Setting up our context menu handler and reconfiguring the EditorGridPanel**

```
var doCellCtxMenu = function(editorGrid, rowIndex, cellIndex, evtObj) { // 1
    evtObj.stopEvent();

    if (!editorGrid.rowContextMenu) { // 2
        editorGrid.rowContextMenu = new Ext.menu.Menu({
            items : [
                {
                    text : 'Insert Record',
                    handler : onInsertRecord
                },
                {
                    text : 'Delete Record',
                    handler : onDelete
                }
            ]
        });
    }
    editorGrid.getSelectionModel().select(rowIndex, cellIndex); // 3
    editorGrid.rowContextMenu.showAt(evtObj.getXY());
}
```

{1} The cell context menu listener method  
{2} Creating the context menu object  
{3} Selecting the cell that was right clicked

Listing 8.8 contains `doCellCtxMenu`{1}, a method to handle the `cellcontextmenu` event from the `EditorGridPanel`, which is responsible for creating and showing the context menu for the insert and delete operations. Here is how it works.

`doCellCtxMenu` accepts four arguments, which are passed by the `cellcontextmenu` handler. They are `editorGrid`, a reference to the `EditorGridPanel` that fired the event, `rowIndex` and `cellIndex`, which are the coordinates of the cell that was right-clicked and `evtObj`, an instance of `Ext.EventObject`.

The first function that this method performs is preventing the right click event from bubbling upwards by calling the `evtObj.stopEvent`, preventing the browser from displaying its own context menu. If we did not prevent the event from bubbling, we would see the browser context menu on top of ours, which would just be silly and unusable.

`doCellCtxMenu` then tests{2} to see if the `EditorGridPanel` does not have a `rowContextMenu` property and creates an instance of `Ext.menu.Menu` and stores the reference as the `rowContextMenu` property on the `EditorGridPanel`. This effectively allows for the creation of a single Menu, which is more efficient than creating a new instance of

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

`Ext.menu.Menu` every time the event is fired and will last until the `EditorGridPanel` is destroyed, as we'll see later.

We pass a configuration object to the `Ext.menu.Menu` constructor, which has a single property, `items`, that is an array of configuration objects that get translated to instance of `Ext.menu.MenuItem`. The `MenuItem` configuration objects both reference the respective handlers to match the `Item` text.

The last two functions that this method performs is selecting the cell that was right clicked and showing the context menu at the correct X and Y coordinates on screen. It does this by calling the `select{3}` method of the `EditorGridPanel`'s `CellSelectionModel` and passing it the `rowIndex` and `cellIndex` coordinates. Lastly, we display the context menu using the coordinates where the right-click event occurred.

Before we move on to exercise our code, we're going to have to reconfigure the grid to register the context menu handler. Please add the following to your grid configuration object.

```
listeners : {
    cellcontextmenu : doCellCtxMenu
}
```

We now have everything we need to start exercising our new UI features. I want to see this thing in action.

#### 8.4.4 Exercising Create and Delete

At this point, we have our insert and delete handlers developed and ready to be used. We just finished creating the context menu handler and reconfigured our grid to call it when the `cellcontextmenu` event is fired.

We'll start our exploration by creating and inserting new Record.

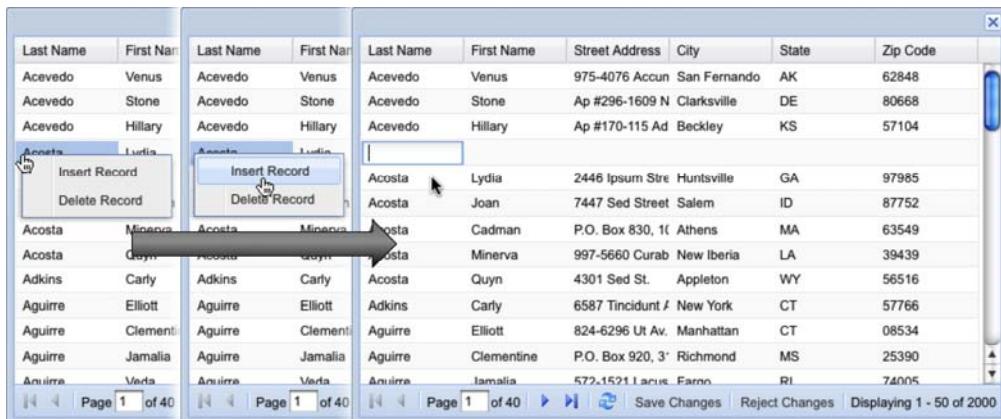


Figure 8.7 Adding a new record with our newly configured "Insert Record"

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

As illustrated in the figure above, we can display the context menu by right clicking on any cell, which calls the doCellContextMenu handler. This causes the selection the cell to occur and displays the custom Ext menu at the mouse's coordinates. Clicking on the "Insert Record" menu Item forces the call to the registered handler, onInsertRecord, which inserts a new record at the index of the selected cell and begins editing on the first column. Cool!

Now in order to save changes, we need to modify the newly inserted record, we need to click the "Save Changes" button that we created earlier.

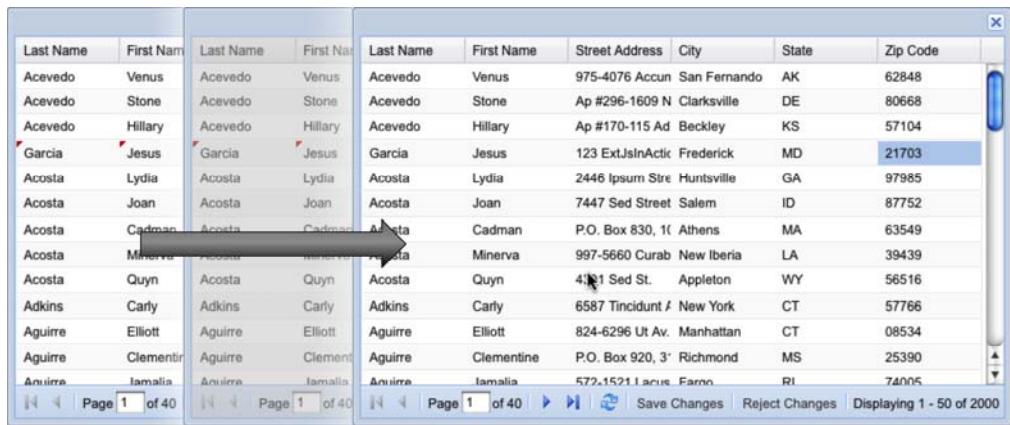


Figure 8.8 The UI transitions when saving our newly inserted record.

Clicking the "Save Changes" Button invokes the onSave handler, which performs the Ajax request to our mock server handler, the successTrue.js file. Here is what the JSON looks like being submitted in FireBug's XHR inspection tool.

```

POST http://ext2play/extia/chapter08/successTrue.js 200 OK 106ms
ext-base.js (line 1310)

records [{"newRecordId": "ext-gen85", "lastname": "Garcia", "firstname": "Jesus", "street": "123 ExtJSInAction Way", "city": "Frederick", "state": "MD", "zipcode": "21703"}]

```

Figure 8.9 Using FireBug to inspect the JSON being submitted for our newly inserted record.

As you can see, there is no id associated with this record, but there is a newRecordId property, which is a tell tale sign that it is a new record. This property is important because

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

the controller code on the server could use this to know that this is a new record to be inserted versus an update operation. Remember that our `onSave` handler is setup to handle both inserts and update operations, thus the theoretical controller needs to be able to handle both situations. Here is a quick overview on how things might work if we had actual server code to submit to.

The server would receive the `records` parameter and decode the JSON. It would then loop through the array and perform inserts on any of the records that have this `newRecordIdProperty` and return a list of database ids for the records that were newly inserted. Instead of a generic return of `{ success : true }`, it could return something much more intelligent, like a list of objects that map the `newRecordId` to the record's database id, which would look something like:

```
{
  success : true,
  records : [
    {
      newRecordId : 'ext-gen85',
      id          : 2213
    }
  ]
}
```

Our success handler can then search the store for the records that we just inserted and set the Record's to the database id, which effectively makes the record a non-phantom or a real record. The success handler code would look something like this.

```
success : function(response) {
  grid.el.unmask();
  remoteJsonStore.commitChanges();

  var result = Ext.decode(response.responseText);
  Ext.each(result.records, function(o) {
    var rIndex = remoteJsonStore.find('newRecordId', o.newRecordId);
    var record = remoteJsonStore.getAt(rIndex);

    record.set('id', o.id);
    delete record.data.newRecordId;
  });
}
```

In the above snippet, we use `Ext.decode` to decode the returning JSON, which is the `response.responseText` property. We then use `Ext.each` to loop through the resulting array of objects. For each object in the list, we get the record index by using the `remoteJsonStore`'s `find` method and pass two properties. The first is the field we're searching for, which is `newRecordId` and the other is the returned object's `newRecordId`. We then get the reference of the record by calling the `remoteJsonStore`'s `getAt` method, and pass the Record index that we just obtained. We then use the Record's `set` method to set the database id we just got back from the server and then move on to delete the record's `newRecordId` property. This ensures that any further modifications to that record will only

results in an update as it will pass its database id to the controller. Here is what an update to the recently inserted record would look like in FireBug.



Figure 8.10 A firebug view of an newly inserted record followed by a subsequent update.

In the above figure, we can see that the newRecordId being sent to the server for the insertion. The server then returned the database id as 9999, and our newly modified success handler set the record's database id and removed the newRecordId. Woah, that's a lot of material just for the creation of records. What about delete? Surely that is simpler, right?

Absolutely! Before we discuss the process of deleting records, we'll examine how the UI works.

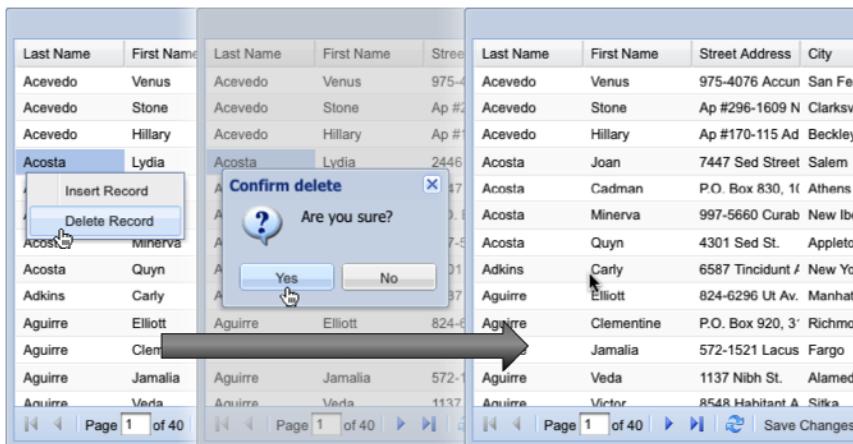


Figure 8.11 The UI workflow for deleting a record.

When we right click on a record, an `Ext.MessageBox` displays to confirm the delete operation. We click yes and an Ajax request is made to the controller to delete the records. Here is what the delete request looks like in FireBug's XHR inspection tool.

The screenshot shows a Firebug panel for a GET request to 'dataDelete.php?records=%5B%22241%22%'. The 'Params' tab is selected, displaying three parameters: '\_dc' with value '1244043974143', 'callback' with value 'stcCallback1002', and 'records' with value '[ "241", "883" ]'.

Figure 8.12 The controller request to delete a record as viewed in FireBug's XHR inspection tool.

This works because our `onDelete` handler called `MessageBox.confirm` and will call our `doDelete` method, which checks to see if this is a new record or not. Because we happened to request a deletion of a non-phantom record, an Ajax request was made to the central controller with one parameter, `rowToDelete`, which is the database id of the record. If the record was a phantom, the request would have never been made and the record immediately moved from the store.

We did a lot of work to get manual CRUD operations setup for our first `EditorGridPanel`. In doing so, we learned more about Stores, Records, how to detect changes and save them using Ajax requests. Along the way we got a chance to see a real-life case of an Ext confirmation `MessageBox` in action. Cool.

Now that we have learned the nuts and bolts of manual crud, we can switch tracks to learn how we can leverage `Ext.data.DataWriter` to manage CRUD operations easily and even automatically.

## 8.5 Using `Ext.data.DataWriter`

In our last example we learned how to code manual CRUD actions, which meant we had to code our own Ajax requests. But what if we wanted the editor grid to automatically save when we're editing? In order to do this without writer, we'd have to write an entire event model that fired off requests when a CRUD UI action took place. This would, of course, have to take into account exception handling, which means we'd have to code for the rollback changes. I can personally tell you that it's a lot of work. Luckily, we don't have to do all of this for easy and automated CRUD.

### 8.5.1 Enter `Ext.data.DataWriter`

Writer saves you time and effort by removing the requirement for you to have to code Ajax requests and exception handling, giving you more time to do more of the important stuff,

like building out the business logic for your application. Before we start coding our writer Implementation we should have a quick review how Writer fits into the picture.

Please recall our conversation about the operation of data Stores from the last chapter, where we learned about the flow of data from the source to the consumer. Remember that the Proxy is the intermediary connection class for reads as well as writes. If things still seem a little fuzzy, the figure below is there to help clear things up.

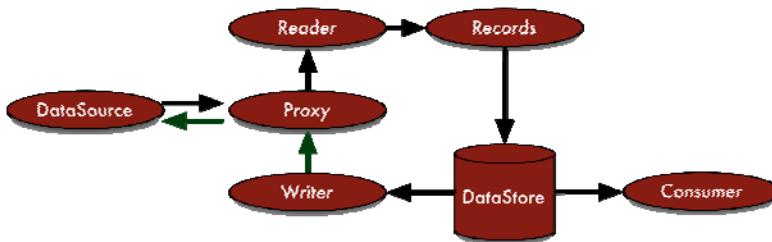


Figure 8.13 A depiction of the flow for data reads and writes when a data Store is used.

In order to user Writer, we will need to reconfigure our data Store and the supporting Proxy. Instead of configure a `url` property for the Proxy, we're going to create a configuration objects known as the `api`. The proxy `api` is a new concept for us and we'll discuss it more in detail in just a bit, when we review the example code.

We will then need to create an instance of Writer and plug it into our data Store as well as add some new configuration properties to the Store's configuration object itself, thus completing the reconfiguration of the Store.

To reconfigure the `EditorGridPanel` and the CRUD actions, we'll keep all of the UI changes but remove the supporting code for the Ajax requests. Being that we'll be familiar with most of this code, we'll be moving at a faster pace, but we'll slow down for the new material.

### 8.5.2 Adding DataWriter to our JsonStore

Now that we have an understanding of what we'll be doing, we can get our shovels out and start digging. Just as before, we'll begin by reconfiguring our Store.

#### **Listing 8.9 Reconfiguring data Store to use Writer**

```

var remoteProxy = new Ext.data.ScriptTagProxy({ // 1
    api : {
        read   : 'http://tdg-i.com/dataQuery.php',
        create : 'http://tdg-i.com/dataCreate.php',
        update : 'http://tdg-i.com/dataUpdate.php',
        destroy: 'http://tdg-i.com/dataDelete.php'
    }
}); 
  
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var recordFields = [
    { name : 'id',           mapping : 'id' },
    { name : 'firstname',    mapping : 'firstname' },
    { name : 'lastname',     mapping : 'lastname' },
    { name : 'street',       mapping : 'street' },
    { name : 'city',          mapping : 'city' },
    { name : 'state',         mapping : 'state' },
    { name : 'zipcode',       mapping : 'zip' }
];

var writer = new Ext.data.JsonWriter({
    writeAllFields : true
}); // 2

var remoteJsonStore = new Ext.data.JsonStore({
    proxy          : remoteProxy,
    storeId        : 'ourRemoteStore',
    root           : 'records',
    autoLoad       : false,
    totalProperty : 'totalCount',
    remoteSort    : true,
    fields         : recordFields,
    idProperty    : 'id',
    autoSave       : false,
    successProperty: 'success',
    writer         : writer,
    listeners      : {
        exception : function () {
            console.info(arguments);
        }
    }
}); // 3

{1} Configuring the api for the ScriptTagProxy
{2} Instantiating a new instance of JsonWriter
{3} Adding new properties to the Store configuration object

```

In listing 8.9, we kick things off by creating a new `ScriptTagProxy`{1} and pass a configuration object, as the property `api`, which is a configuration object denoting URLs for each of the CRUD actions, with `read` being the request to load data. Instead of using the dummy `successTrue.js` file as a single controller, we're going to actually use somewhat intelligent remote server side code, where a controller exists for each CRUD action. Writer requires intelligent responses, thus remote server side code was developed. Technically, we could use the exact same server side script for all of the CRUD actions, but I find it easier to create one for each action.

We then move on to create the list of fields, which get translated into \_\_\_\_? Correct! `Ext.data.Fields`. These Fields are the lowest supporting class for the \_\_\_\_? Yes! The `Ext.data.Record`. You're progressing in this framework nicely.

Next, we create a subclass of the `Ext.data.DataWriter`, known as `JsonWriter`{2}, which has the ability to save a request to modify a single or batch (list) of records. In the `JsonWriter` configuration object, we specify `writeAllFields` as `true`, which ensure that for each operation, `Writer` returns all of the properties, which is great for development and debugging. Naturally, you want to set this to `false` when in production, which will reduce overhead over the wire and at the server side and database stack.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

The last thing we do in this listing is reconfiguring the Store. Note that everything is exactly the same except for a few property additions to enable Writer integration and some debugging capabilities.

The first addition is `autoSave{1}`, which we set to `false`, but defaults to `true`. If left as `true`, the store would automatically fire requests for CRUD operations, which is not what we want just yet. I want to show you how *easy* it is to get invoke CRUD requests with Writer now in the picture.

Next, we add the `successProperty`, which is used as an indication that the operation was a success or failure and is actually consumed by the `JsonReader`, which the `JsonStore` automatically instantiates for us. Remember, this is just like when we submitted data with the `FormPanel`, where we required at a minimum a return of `{ success : true }` from the response from the web server. The same principle applies with the Store when using `DataWriter`. In our new `remoteJsonStore`, we're specifying the `successProperty` of '`success`', which is common and self-documenting.

The last change we made is adding a global exception event listener to our `JsonStore`, which is needed if you wanted something to occur upon any exception that the Store raises. Here, we simply spit all of the arguments to the Firebug console, which I use when developing with `Ext.data.DataWriter` because it provides a wealth of information that is hard to find anywhere else during debugging. I highly suggest doing the same. Trust me, it will save you time in the long run.

Cool! We've just recreated our `ScriptTagProxy` to work with our new instance of `Ext.data.JsonWriter` and reconfigured our Store to prevent `autoSaves`. Our next task is to modify the `PagingToolbar` and make the delete Context menu handler leaner

### **Listing 8.10 Reconfiguring the PagingToolbar and delete handler**

```
var pagingToolbar = {
    xtype      : 'paging',
    store      : remoteJsonStore,
    pageSize   : 50,
    displayInfo: true,
    items      : [
        '_',
        {
            text      : 'Save Changes',
            handler   : function () {
                remoteJsonStore.save(); // 1
            }
        },
        '_',
        {
            text      : 'Reject Changes',
            handler   : function () {
                remoteJsonStore.rejectChanges(); // 2
            }
        }
    ]
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var onDelete = function() { // 3
    var grid          = Ext.getCmp('myEditorGrid');
    var selected     = grid.getSelectionModel().getSelectedCell();
    var recordToDelete = grid.store.getAt(selected[0]);

    grid.store.remove(recordToDelete);
}

```

- {1} Saving our changes via Writer
- {2} Rejecting changes from our Store
- {3} A leaner onDelete context menu handler

In Listing 8.10, we reconfigure the PagingToolbar's save button handler{1} to simply call the save method of the data Store, which is what uses Writer to gather the data to save and eventually invoke a request to save changes. This effectively replaces the previous onSave handler, which is what was responsible for sending Create and Update Ajax requests. This is where we start to see some of the code savings we discussed earlier. We also added an inline method to reject{2} the Store's changes.

Next, we refactor the onDelete{3} context menu handler. We removed the typical confirmation dialog box to make it much leaner. We can choose to reject the changes from here, which means that the records that are deleted from the UI are rolled back via the rejectChanges handler we created above. We also removed the Ajax request code, thus we have more code savings.

Our changes to integrate Writer into our EditorGridPanel panel are now complete. The EditorGridPanel configuration code stays exactly the same as in Listing 8.8. Let's see Writer in action.

### 8.5.3 Using DataWriter

The change we made in the previous listings, where we Integrated writer were designed so we could use the exact same interaction to invoke a request, but see how we could reduce the amount of code when using Writer relative to creating our own Ajax requests and handlers.

We'll start by modifying some records and clicking the Save Changes button to inspect what Writer is sending to the server. Below is an illustration of the parameters being sent for an update request in Firebug's request inspection tool.



Figure 8.14 The result of our update request as view in Firebug's request inspection tool.

In Figure 8.14, we see the result of a request being sent to the update URL as configured in the Proxy API. Along with the usual `_dc` (cache buster) and `callback` parameters, we see `id`, which is a list of IDs affected and `records`, which is the list of records that were changed. We see the entire record contents here because we set the `writeAllFields` configuration parameter to true.

Our server side code then processes the list of records and returns the following JSON in return. Here is what the return looks like.



Figure 8.15 The server side response from our update request with Writer.

Notice that the `successProperty`, `success`, has a value of `true`, which is an indication that the server processed the data successfully. It also returned the list of records after it was processed. This is used to apply any changes that may be required by business logic. For instance, what if someone put a forbidden word as one of the values. The server could reject the change by quietly replacing that forbidden value with what was previously in that field. The UI would then update accordingly, thus completing an update cycle.

Next, we should exercise an insert and see what happens. This is where things get interesting. Insert a record via the context menu, add some values, and click the save button in the PagingToolbar. We see that a request was made to the `create` URL that we defined in the `api` configuration object for the `ScriptTagProxy`.

▼ GET dataCreate.php?records= 200 OK tdgi

Params Headers Response HTML

\_dc 1244040436412  
callback stcCallback1002  
records {"lastname": "Garcia", "firstname": "Jesus", "id": "ext-record-1"}

Figure 8.16 Inserting a record with Writer as viewed in the Firebug request inspection tool.

The biggest thing to note in the `records` parameter being sent to the server is the `id` property. Notice that the `id` fictitious, where its value is “ext-record-1”. Being that the create server side action is different from the update action, the `ID` parameter is ignored when the record is inserted into the database. The server then gets the ID for the newly inserted record and returns the database id in return as illustrated below.

▼ GET dataCreate.php?records= 200 OK tdgi 103 B

Params Headers Response HTML

stcCallback1002({success:true, records : {"lastname": "Garcia", "firstname": "Jesus", "id": "1244040436"}});

Figure 8.17 Inspecting results of a record insert using Writer with Firebug.

If an insert was successful, then the `id` value (along with all of the other values) returned to the browser will be applied to the recently inserted record. This ensures action requests for update and delete will submit the database ID of the record moving forward. This is where we can see the added value of using Writer, where we don’t have to manage this verification logic our selves.

In our test case, we inserted a single record. If you insert multiple records, they will be submitted to the server, upon save, in a list. It is important to note that the server **must** return the records back in the exact same order, or the database ID to record mapping or association will fail.

The delete action is the simplest, where the delete request sends a list of IDs to the host for processing. To exercise this, right click on a record, and choose delete from the context menu. Heck, do this for a few records. Notice how they are removed from the store? Now click save and inspect the request in Firebug.

▼ GET dataDelete.php?recor 200 OK tdgi

Params Headers Response HTML

\_dc 1244057155217  
callback stcCallback1002  
records ["241", "883"]

Figure 8.18 The JSON parameters for a delete action request.

For the delete action requests, we see that the records parameter simply is a list of IDs. Lets take a look at what the server side returns. Below is an illustration of the result from the server for the delete request.



```
▼ GET dataDelete.php?records=%5B%22241%22%3B 200 OK
Params Headers Response HTML
stcCallback1002({success:true, records :[{"id":"241"}, {"id":"883"}]});
```

Figure 8.19 The result of our delete action request using Writer.

The server side code takes the list of ids, removes the records from the database and returns the list of IDs for which the store is to permanently remove. And that's all there is to delete operations.

We've seen how we could use Writer for CRUD operations that were invoked manually, but required no special AJAX request code and handling of our own. The last topic for discussion is automated CRUD with Writer.

#### 8.5.4 An Automated Writing Store

To setup Writer for automated CRUD actions, all we need to do is set the autoSave property for the data Store configuration object to true. That's it. We already setup all of the hard stuff, if you want to call it that. Now all you need to do is exercise CRUD operations and the Store will automatically invoke requests based on your CRUD action.

The one thing to look out for with an automatic writing Store is that there is *no* undo for actions. That is, when data has been modified and successfully altered in the database, the rejectChanges method will have no effect. The database now reflects the latest changes. The same goes for delete and insert operations.

### 8.6 Summary

In this chapter we got our first exposure to the EditableGrid class and learned how it uses Ext.form.Fields to allow for editing of data on the fly. This gave us an opportunity to learn about the CellSelectionModel and some of its methods, such as getSelectedCell. We also learned how we could leverage keyboard and mouse gestures to navigate the EditorGridPanel to edit data relatively rapidly.

We learned how to manually code for CRUD operations with our own custom AJAX request logic and used a mock controller while doing so. We added two Menu items to the PagingToolbar and a context menu to the EditableGrid to allow us to Insert and delete records as well as reject changes. In doing this, we learned how to getModifiedRecords from the Store for submission the mock controller.

Lastly, we learned how we could reduce the amount of code we needed to generate by leveraging the `Ext.data.DataWriter` class for CRUD operations. We also discussed how to setup an automated Store with Writer.

In the next chapter, we're going to learn all about another flagship UI widget, the `Ext.tree.TreePanel`.

# 9

## DataView and ListView

Displaying lists of data in a web application is something that we've had to deliver at one time or another in our careers. Whether it's a list of books, servers on a particular subnet or a list of employees, the process is the same. Retrieve the data, format and display it. While this process is simple from a high-level, it's the burden of maintaining the under-the-hood JavaScript that has prevented us from being able to focus all of our attention on getting the task done. Throw in the ability to select multiple items and you find yourself spending more time on maintenance rather than further developing your application.

In this chapter, we'll learn that with the DataView you can achieve this goal easily, thus saving you time and allowing you to focus on the deliverables at hand. We'll begin by constructing a DataView and introduce a major supporting class, the XTemplate, along the way. We'll learn what it takes to properly configure the DataView for single or multiple record selections.

Aftwards, we'll learn about how to create a ListView to display Data in a tabular format, much like the GridPanel. We'll see what it takes to bind it to our implementation of the DataView to assist with the filtering of data from the DataView.

As icing on the cake, we'll learn how to complex two-way bindings between the DataView and FormPanel, to allow users to update data. What you'll learn in the final exercise will help you in binding the DataView and ListView to other widgets in the framework.

### 9.1 What is the DataView

The DataView class uses with data store and XTemplate to provide the ability to paint data on screen easily. It has all of the necessary plumbing to provide for the tracking of mouse and on its DOM structure and has a single or multiple node selection models.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Below is an illustration of a DataView in action and highlights how the DataStore and XTemplate support it.

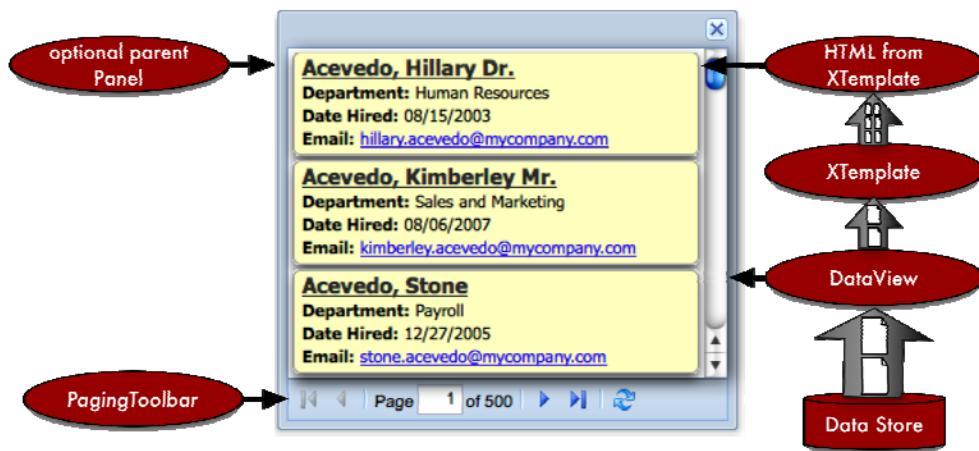


Figure 9.1 The various DataView and various supporting classes.

As illustrated above, the DataView consumes data from the Data store by means event bindings. It is because of these bindings that it is smart enough to know how to efficiently handle the DOM changes. For instance, if a single record is deleted from a store, the element whose index matches the record index is removed from the DOM. For added flexibility, pagination can also be applied to the DataView via binding a PagingToolbar to the DataView's bound store.

#### NOTE

Unlike the GridPanel and TreePanel, DataView does not extend from Panel, which means that it cannot be configured to use any of the Panel features, such as Toolbars. The good news is that the DataView does extend from BoxComponent, which means it can take place in layouts and can easily be wrapped by a Panel.

We talked about Templates earlier in this book and learned how we can stamp out HTML fragments really easily with that tool. The DataView uses the Template's more powerful cousin, the XTemplate, which adds the ability to have sub templates, inline code execution, conditional processing and much more.

OK, we now know what a DataView is and does. It's time to start building one.

## 9.2 Constructing a DataView

You've just been asked to develop a full screen mini-application to allow members of the HR team to quickly list employees in the company. They want the records to be styled and have a background similar to a manila folder. For each record, the view needs to be able to display the employee's full name, department, date hired and email address. The requestor stated that this view would be part of a larger app, which we'll learn more about once we've completed this view. The back end is complete and ready to serve up JSON, which means that all we need to do is write the front-end code.

To fulfill this initial requirement, we'll have to create DataView that will live in a ViewPort. But, before we can actually start writing the JavaScript code, we need to construct the CSS for our HTML. This is because when implementing DataViews, as end-developers, we are *required* to configure the CSS to style the contents of the widget. If we did not apply styling, the painted data would be unusable.

Here is what our DataView would look like if it was not styled, versus styled.

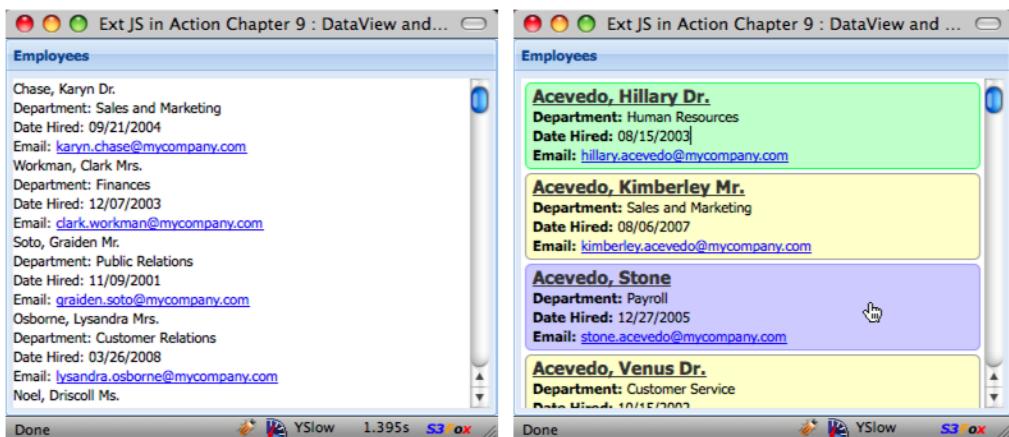


Figure 9.2 An un-styled (left) DataView compared to a styled (right) DataView.

It should be obvious that a styled DataView is both better for usability and the users eyes. Each implementation of the DataView has its own styles. Perhaps when you implement this widget on your projects, you could create a base CSS set of rules that all of these widgets can leverage, giving them unified look for the application.

OK, seeing what it will need to look like, we can configure the CSS before we begin work on the JavaScript for the widget and its supporting classes.

### **Listing 9.1 Setting the data view styles.**

```
<style type="text/css">
    .emplWrap {
        /* 1 */
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

border: 1px #999999 solid;
margin : 3px;
-moz-border-radius: 5px;
-webkit-border-radius: 5px;
background-color: #ffffcc;
padding-bottom: 3px;
}

.emplSelected {                                     /* 2 */
    border: 1px #66ff66 solid;
    background-color: #ccffcc;
    cursor: pointer;
}

.emplOver {                                       /* 3 */
    border: 1px #9999ff solid;
    background-color: #ccccff;
    cursor: pointer;
}

.emplName {                                       /* 4 */
    font-weight: bold;
    margin-left: 5px;
    font-size: 14px;
    text-decoration: underline;
    color: #333333;
}

.title {
    margin-left: 5px;
    font-weight: bold;
}

```

</style>

{1} Style all records to have a yellow color  
{2} Selected records are green  
{3} When moused over, the records turn blue  
{4} Make the employee name text larger than the rest

In the above listing, we configure all of the CSS that will be used by the DataView. The first rule, emplWRap{1}, will be used to style each record with a light gray border and a yellowish background. The next rule, emplSelected{2}, will be used color a record green when a user has selected it. The third rule, emplOver{3}, is going to be applied to a record when the mouse is hovering over it and will color it blue. The last two rules, emplName{4} and title are simply additional rules to style the inner contents of each rendered record.

Now that we have the CSS in place, we can begin with the construction of the store that will provide the data.

### 9.2.1 Building the Store and XTemplate

We are constructing the DataStore first because it will help us in knowing what the mapped fields are when we configure the XTemplate for the DataView.

#### Listing 9.2 Implementing a remote JsonStore

```

var employeeStoreProxy = new Ext.data.ScriptTagProxy({           // 1
    url : 'http://extjsinaction.com/examples/chapter09/getEmployees.php'
});

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var employeeDvStore = {
    xtype      : 'jsonstore',
    autoLoad   : true,
    storeId    : 'employeeDv',
    proxy      : employeeStoreProxy,
    fields     : [
        { name : "datehired",   mapping : "datehired" },
        { name : "department",  mapping : "department" },
        { name : "email",       mapping : "email" },
        { name : "firstname",   mapping : "firstname" },
        { name : "id",          mapping : "id" },
        { name : "lastname",    mapping : "lastname" },
        { name : "middle",      mapping : "middle" },
        { name : "title",       mapping : "title" }
    ]
};

{1} Create the remote proxy
{2} Configure a JsonStore XType object

```

We create the JsonStore for our DataView in Listing 9.2, which leverages a remote ScriptTagProxy{1} to perform the data request tasks. When creating the JsonStore{2} XType, we set autoLoad to true, which will automatically cause the proxy to fire the request after instantiation. We also define the field mappings inline for all of the required data points. This will help us in constructing the XTemplate.

Being that the Store is a non-visual component, there's not much to see at this point. We'll begin by constructing the XTemplate and have a brief discussion about it as it is important to understand how the data that will be passed to the XTemplate is applied to construct the DOM fragments.

### **Listing 9.3 Constructing the XTemplate**

```

var employeeDVTpl = new Ext.XTemplate(
    '<tpl for=".">' + // 1
        '<div class="emplWrap" id="employee_{id}">', // 2
            '<div class="emplName">{lastname}, {firstname} {title}</div>',
            '<div><span class="title">Department:</span> {department}</div>',
            '<div>',
                '<span class="title">Email:</span><a href="#">{email}</a>',
            '</div>',
        '</div>',
    '</tpl>'
);

```

```

{1} Specifying the template branch data point
{2} Begin the record displaying HTML fragment

```

In order for a DataView to render data on screen, it needs an Ext JS template. This is why we instantiate an instance of Ext.XTemplate first{1}. While we're not going to dive too deep into the usage of XTemplates, I do feel that it is important to discuss what the XTemplate is doing as this is our first exposure to this component and the `tpl` tag and the `for` attribute are commonly overlooked by many developers. I want to make sure you're not one of them.

The first (and last) tags that are specified in the XTemplate list of arguments are `tpl`, and help the XTemplate organize logical branches of HTML. In the first `tpl` tag, there is a `for` attribute, which specifies which data point the XTemplate is to use to fill in for that  
 ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

branch. To help us understand this, we'll quickly examine the data returned by the server and ponder how the template would look differently if the data were different.

Each of the records the XTemplate will use are constructed like the following plain JavaScript object:

```
{  
    "id" : "1",  
    "firstname" : "Karyn",  
    "lastname" : "Chase",  
    ...  
}
```

By setting the `for` attribute of the `tpl` tag with a single period `.`, the XTemplate knows that the branch below the `tpl` will use the root object to fill the data points as specified in the HTML fragment defined within.

Here's what that record looks like stamped out in the DOM with the XTemplate we've just defined.

```
▼ <div id="employee_1" class="emplWrap">  
  <div class="emplName">Chase, Karyn Dr. </div>  
  ▼ <div>  
    <span class="title">Department: </span>  
    Sales and Marketing  
  </div>  
  ▼ <div>  
    <span class="title">Date Hired: </span>  
    09/21/2004  
  </div>  
  ▼ <div>  
    <span class="title">Email: </span>  
    <a href="#">karyn.chase@mycompany.com</a>  
  </div>  
</div>
```

Figure 9.3 A rendered HTML Fragment as viewed in Firebug's DOM inspection panel.

The HTML fragment illustrated in Figure 9.2 shows the template filled out with the data for a single record. Notice that all of the data points that are defined in the XTemplate are replaced by the actual data from the *root* of the record.

OK, but what if the requirement was to require phone numbers to be displayed in a flexible fashion? That is only display what is in the record. For this, the data might be structured differently.

For instance, added to the object below is a phoneNumbers property, which is an array of objects containing the phone type and number.

```
{
    "id"          : "1",
    "firstname"   : "Karyn",
    "lastname"    : "Chase",
    ...
    "phoneNumbers": [
        {
            "type" : 'Mobile',
            "num"  : '555-123-4567'
        },
        {
            "type" : 'Office',
            "num"  : '555-765-4321'
        }
    ]
};
```

In this case the XTemplate to consume this data would look like the following:

```
var otherTemplate = new Ext.XTemplate(
    '<tpl for=".">' ,
        '<div class="emplWrap" id="employee {id}">',
            '<div class="emplName">{lastname}, {firstname} {title}</div>',
            '<div><span class="title">Department:</span> {department}</div>',
            '<div><span class="title">Date Hired:</span> {datehired}</div>',
            '<div>',
                '<span class="title">Email:</span> <a href="#">{email}</a>',
            '</div>',
            '<tpl for="phoneNumbers">',
                '<div><span class="title">{type}</span> {num}</div>',
            '</tpl>',
            '</div>',
        '</div>',
    '</tpl>'
);
```

Notice the second `tpl` tag towards the end of the argument list. This denotes another branch of the HTML fragment and the `for` attribute specifies that this branch is *for* the "phoneNumbers" property, which happens to be an array. For every object in that array, XTemplate will loop through and produce a copy of that branch of HTML.

Here's what that HTML fragment will look like:

```
▼ <div id="employee_1" class="emplWrap">
  <div class="emplName">Chase, Karyn Dr. </div>
  ▼ <div>
    <span class="title">Department:</span>
    Sales and Marketing
  </div>
  ▼ <div>
    <span class="title">Date Hired:</span>
    09/21/2004
  </div>
  ▼ <div>
    <span class="title">Email:</span>
    <a href="#">karyn.chase@mycompany.com</a>
  </div>
  ▼ <div>
    <span class="title">Mobile:</span>
    555-123-4567
  </div>
  ▼ <div>
    <span class="title">Office:</span>
    555-765-4321
  </div>
</div>
```

Figure 9.4 The Exploded HTML fragment that can be produced by the XTemplate with a `for loop.

Notice that there are two div elements for the phoneNumber objects and the values for the mapped properties are populated within. You now know how to flex some of the XTemplate's muscle and you're better prepared for a similar situation.

### LEARN MORE ABOUT XTEMPLATES

Though we will not be touching on the XTemplate's many capabilities, it's important to note that the Ext JS API has well-written examples on the XTemplate on the API documentation page. <http://www.extjs.com/deploy/dev/docs/?class=Ext.XTemplate>

OK, we have our XTemplate (employeeDvTpl) constructed and a basic understanding of what the `tpl` tag and `for` attribute arguments are and what they do. We can now move forward with constructing the DataView and then place it in a Viewport.

#### 9.2.1 Building the DataView and Viewport

Next, we'll construct the DataView that will display all of the employees in the company.

#### Listing 9.5 Constructing the DataView

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var employeeDv = new Ext.DataView({
    tpl         : employeeDvTpl,
    store       : employeeDvStore,
    singleSelect: true,                                // 1
    itemSelector: 'div.emplWrap',
    selectedClass: 'emplSelected',
    overClass   : 'emplOver',
    style       : 'overflow:auto; background-color: #FFFFFF;' // 2
});

{1} Instantiate a new DataView
{2} Enable single node selection.
{3} Scroll the auto flow

```

To construct the `DataView`{1}, we see that not much configuration has to be applied. In fact, most of the code to create a `DataView` actually occurs in the configuration or instantiation of the supporting data store and XTemplate. But, it is important to take a really good look at the configuration options that have been supplied.

Other than the usage of the template and store, we set `singleSelect`{2} to the Boolean value `true`. Setting this property as such will instruct the `DataView` to allow single or multiple (`multiSelect`) node selections when `DataView`'s element is clicked. This, of course, is important when you want a rendered record to be selectable. We set this property to `true` because I have a strong feeling that they are going to want to do something with this `DataView` in the future, like perhaps update records when an item is selected.

In order to help the `DataView` along with the selection of a node, when setting `singleSelect` or `multiSelect` properties to `true`, you *must* set the `itemSelector` property, which must be filled in with a proper CSS selector. This property helps the `DataView` hone in on the element that you want to be displayed as selected. It also helps with managing the visual cues for mouse over as well. The selector that we set will help the `DataView` highlight the entire record. In this case, we're using the `div` element with the CSS class `emplWrap` as the selector.

Next, the `selectedClass` and `overClass` are set for the CSS classes that we defined earlier and will be used to help give the user the visual cues that a record is selected or when the mouse hovers over a record.

In able to allow automatic scrolling of the `DataView`'s element, we need to manually set the CSS on the element to do so. I elected to include it inline to help you see the CSS being applied for automatic scrolling, you can just as easily replace the `style` with `cls`, whose value is a CSS class to enable automatic scrolling.

OK, this concludes the `DataView` construction. We need to give it a home. Because the client wants a full-screen view, we should place this in a `Viewport`, which is what we're going to do next.

### **Listing 9.6 Constructing the Viewport**

```

new Ext.Viewport({
    layout      : 'hbox',
    defaults    : {
        ...
    }
}); // 1

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        : 1
    },
layoutConfig : {
    align : 'stretch'
},
items      : [
{
    title  : 'Employees',
    frame  : true,
    layout : 'fit',
    items   : employeeDv
}
]
});
{1} Create the viewport
{2} Wrap the DataView in a Panel with a fit layout
{3} Including the employee dataview

```

Above, we instantiate an instance of Viewport to help render the DataView to take up the full browser view. We could instruct the Viewport to use the fit layout being that there is a single child element, but to have room for future expansion, we will use the HBox layout.

Also, notice that the first child item for the Viewport is a Panel{2}, whose layout is fit and has its frame property set to true. Wrapping the DataView{3} in a Panel will allow us to provide a nice blue frame around it, helping the users understand what they are looking at.

This concludes the construction of the DataView. Lets see what this thing looks like and behaves on screen.

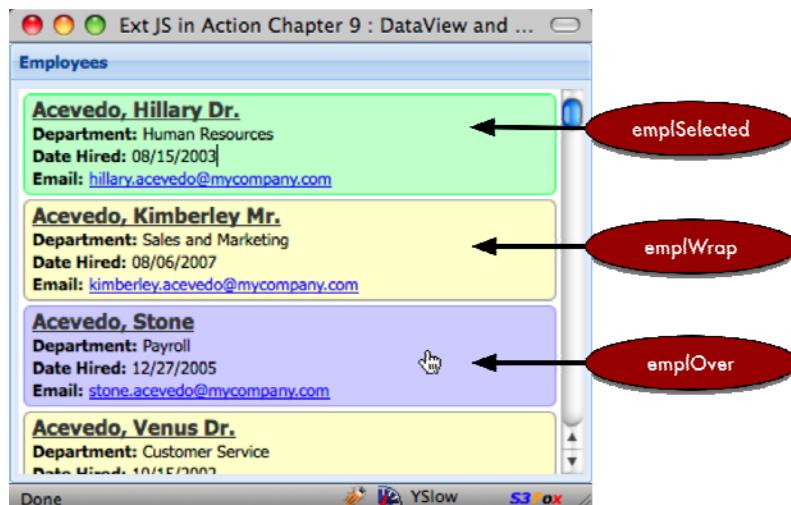


Figure 9.5 The rendered DataView on screen with the CSS classes in action.

Our DataView renders on screen, giving the users the ability to save view all of the employees in the company. We can see that the mouse over event is properly tracked and a single rendered record can be selected. Great!

But something is just not right. The UI seems somewhat slow. The initial render of the data is slow and when the browser is resized, they UI update lags significantly.

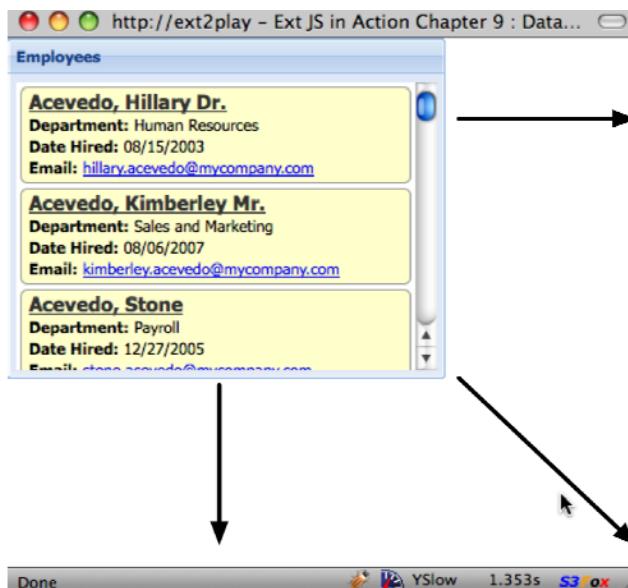


Figure 9.6 Because there are too many records on screen, the simple act of resizing the browser results in a visual delay of the Ext Viewport resize occurring.

If we look at the Store, we see 2,000 records on screen. Surely, this is impacting performance. Also, users would most likely complain if we bombard them with that many rows at a time. We need to come up with a way to increase performance. We need to take a moment to ponder the possible solutions.

A quick solution would be to add a PagingToolbar to the DataView's parent Panel and bind it to the DataView's store. Surely this would assist with the performance, but it would not really make the DataView very usable. I mean even if we split 2,000 records into 100 record-pages, the users would have to flip through 20 pages of data to get what they want. How can we increase performance and make it more usable?

What's that you say? Display the records by department? Yes! I love that idea. OK, but how? We could use a ComboBox, but the list of departments would not be static. Also, it would require two clicks to select a department. I think the users would want something that contained a static list of records, allowing them to filter the list by means of a single click to select a department.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

We could use another DataView, but I think it would be helpful to display the number of employees in each department, which means we need something that can easily manage the data in a column format. A GridPanel is a nice choice, but is overkill. It has features such as column drag and drop, which are not needed and add expense to the UI. I think this is a perfect job for the ListView.

### 9.3 Enter the ListView

We just constructed a DataView that renders data about all of our company's employees on screen at one time and its performance is less than ideal. We determined that we should use the ListView to display the departments in our company and how many employees each department contains. But what is a ListView?

The ListView widget is a subclass of the DataView class that provides a means of displaying data in a tabular format and has a few of features from the DataView and GridPanel. These features include resizable columns, selectable records, column sorting and the ability to have customized templates.

#### NOTE

Even though the ListView presents a GridPanel-like UI, it is not designed to be used in place of the GridPanel where GridPanel-specific features are required. For instance, the ListView does not have the ability for horizontal scrolling and no native support for editing of the data inline like the EditorGridPanel. The ListView also does not have the ability to drag and drop columns nor does it include a column menu.

The ListView also comes pre-packaged with a template and required CSS, which makes it a cinch to implement as we'll see. Lets construct the ListView that we'll need. After all of the configuration required to use the DataView, you're going to see the constructing a ListView is much easier. This code will have to be before the Viewport, as it will be rendered next to the DataView.

#### Listing 9.7 Constructing the ListView

```
var listViewStore = new Ext.data.ScriptTagProxy({                                     // 1
    url : "http://extjsinaction.com/examples/chapter09/getDepartments.php"
});

var departmentLvStore = {                                                 // 2
    xtype : 'jsonstore',
    autoLoad : true,
    storeId : 'departmentDv',
    proxy : listViewStore,
    fields : [
        { name : "department", mapping : "department" },
        { name : "numEmployees", mapping : "numEmployees" }
    ]
};
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var departmentLV = new Ext.ListView({
    store      : departmentLvStore,
    singleSelect : true,
    style       : 'background-color: #FFFFFF;',
    columns     : [
        {
            header   : 'Department Name',
            dataIndex : 'department'
        },
        {
            header   : '# Emp',
            dataIndex : 'numEmployees',
            width    : .20
        }
    ]
});
{1} Create the ScriptTagProxy
{2} Configure the JsonStore
{3} Create the ListView
{4} Configure the ListView's columns

```

To construct the ListView, we begin with the creation of ScriptTagProxy{1} to fetch and feed the data to the reader of the ListView-supporting JsonStore{2}. The JsonStore is only pulling the department name and number of employees per each record.

Next, we create the ListView{3} to display the departments. In addition to the previously created JsonStore, the configuration properties contain singleSelect, to enable selection. Because the ListView already ships with pre-built CSS rules and template, we need not specify the overCls, selectedCls and itemSelector properties. Thought if you did want to use a custom template for your own projects, you will most likely have to set those properties accordingly to match your template.

Next, we style the background of the ListView with a white background. By default, the ListView element contains no background styling and would appear blue when it renders in the wrapped Panel because we're going to set the Panel's frame property to true. Notice that we don't have to set the scroll CSS rule. This is because the ListView sets its element to automatically scroll for us.

Lastly, we set a columns{4} property, which is an array of objects that are used to configure the columns that will vertically organize the data. The second column is configured to display the number of employees for the departments. Notice that the width is .20, which is to express the percentage of the ListView width that column will take. Because there is no horizontal scrolling with this grid-like widget, the column sizing rules are a bit convoluted. Knowing these rules will help you understand why we've configured these columns as such and in further implementations of this Widget.

The first rule is that column widths are always expressed in percentage of the ListView's container. If no column widths are configured, the ListView will automatically size each column equally to fit the ListView's element. This means that for any columns that you wish to have some control over the widths, configure it. Else, if left blank, they will be automatically resized.

To translate these rules to our listing above: because the "Department Name" column has no defined width property, it will be stretched to the remaining width of the ListView after 20% has been carved out for the "# Emp" Column. This gives the "Department Name" column as much room as possible and makes the #Emp column small enough to display the data without being truncated.

OK, the ListView is now setup and ready to be rendered on screen. Let's wrap the ListView in a Panel and place it in a Viewport to the left of the DataView. To do this, we'll have to modify the Viewport's items array. Here's what the changes should look like:

```
items : [
    {
        title : 'All Departments',
        frame : true,
        layout : 'fit',
        items : departmentLV,
        flex : null,
        width : 210
    },
    {
        title : 'Employees',
        frame : true,
        layout : 'fit',
        items : employeeDv
    }
]
```

To add the ListView to the Viewport, we wrap it in a Panel. We injected it before the Employees DataView so it would be rendered to the left-most column and has a static width. Here's what the ListView looks like rendered next to the DataView.

All Departments		Employees
Department Name	# Emp	
Accounting	114	
Advertising	151	
Asset Management	106	
Customer Relations	124	
Customer Service	141	
Finances	176	
Human Resources	120	
Legal Department	144	
Media Relations	146	
Payroll	117	
Public Relations	127	
Quality Assurance	110	
Research and Development	125	

Acevedo, Hillary Dr.  
Department: Human Resources  
Date Hired: 08/15/2003  
Email: [hillary.acevedo@mycompany.com](mailto:hillary.acevedo@mycompany.com)

Acevedo, Kimberley Mr.  
Department: Sales and Marketing  
Date Hired: 08/06/2007  
Email: [kimberley.acevedo@mycompany.com](mailto:kimberley.acevedo@mycompany.com)

Acevedo, Stone  
Department: Payroll  
Date Hired: 12/27/2005  
Email: [stone.acevedo@mycompany.com](mailto:stone.acevedo@mycompany.com)

Acevedo, Venus Dr.  
Department: Customer Service  
Date Hired: 10/15/2002  
Email: [venus.acevedo@mycompany.com](mailto:venus.acevedo@mycompany.com)

Figure 9.7 The ListView (left) rendered with the DataView (right).

Cool! The ListView renders to the left of the DataView and department records can be selected. There are two problems with the current implementation of these two widgets. The first should be obvious - nothing happens when you select a record. Also, the Employees DataView is still loading the 2,000 records. In order to finish this up, we'll need to bind the DataView to the ListView and prevent it from loading all of those records automatically.

### 9.3.1 Binding the DataView to the ListView

We just solved the problem of displaying the departments and their total number of employees by implementing a ListView and rendering it to the left of the Employees DataView, but we did not bind them to solve the performance problem. To do this, we'll first need to prevent the DataView's store from automatically loading. This one is easy. Set the autoLoad property of the employeeDvStore as such:

```
autoLoad : true
```

Next comes the fun part. Configuring a click listener for the ListView to call upon the DataView's store to request the employees for the department that is selected. To do this, we'll need to add the following listeners configuration object to the ListView's config properties.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

listeners : {
    click : function(thisView, index) {
        var record = thisView.store.getAt(index);
        if (record) {
            Ext.StoreMgr.get('employeeDv').load({
                params : {
                    department : record.get('department')
                }
            });
        }
    }
}

```

By adding the above listener to the ListView, we ensure that the DataView is loaded with the records that represent the department that the users wish. Here's how it works.

The click event generated by the DataView (remember ListView is a descendant of DataView), passes the source view component and the index of the node that was clicked as located by the configured itemSelector. We attempt to create a reference of the record selected to instruct the employee DataView to load. If the record is defined, then we use the Ext.StoreMgr.get method to obtain a reference to the employee DataView store by its ID and call its load method. We pass in an object, which contains a params object. The params object contains the department name that was selected.

The load method call will instruct the employee DataView store to call upon the proxy to invoke a request for the employees for the selected department. Lets examine how this changes the behavior thus far.

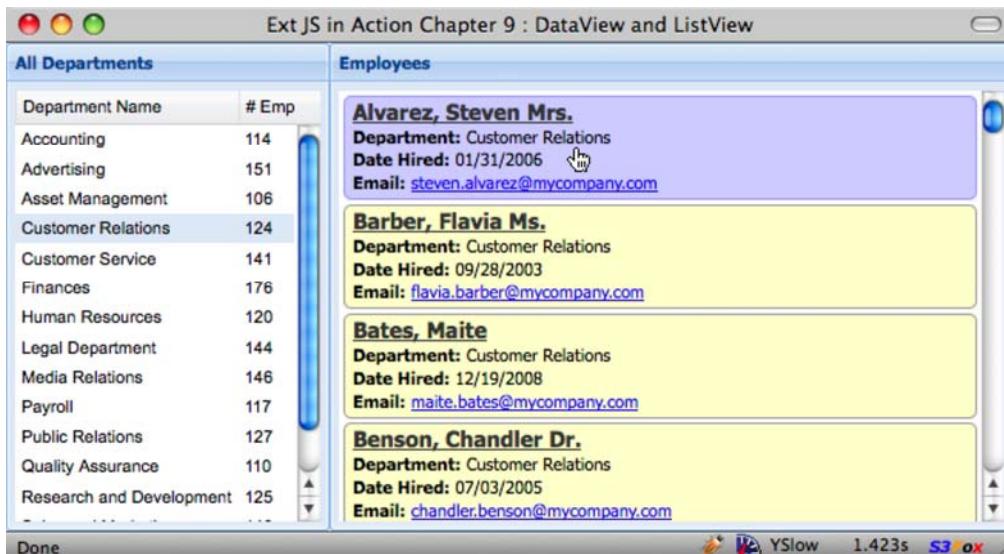


Figure 9.8 The ListView's (left) click event now filters the DataView (right).

Excellent. The ListView now filters the DataView as we've configured it to. We can see that the changes result in a much more responsive UI for the user.

I've sent this over to the users and they have responded expressing their appreciation but have one last request. They want some ability to modify the data for the employee selected. To do this we'll have to configure a FormPanel and add it to the Viewport. This gives us an excellent opportunity to learn how to bind a FormPanel to a DataView and observe the DataView's ability to efficiently update data on screen.

## 9.4 Bringing it all together

In order to allow the users to edit the employee records, we'll need to create a FormPanel for them. The FormPanel needs to have a button to allow them to commit the saved changes. In order to get all of this stuff to work together, we'll have to create a small event model. Here's how this stuff will work.

Here's an image of the proposed layout changes and proposed event handlers and their expected behavior.

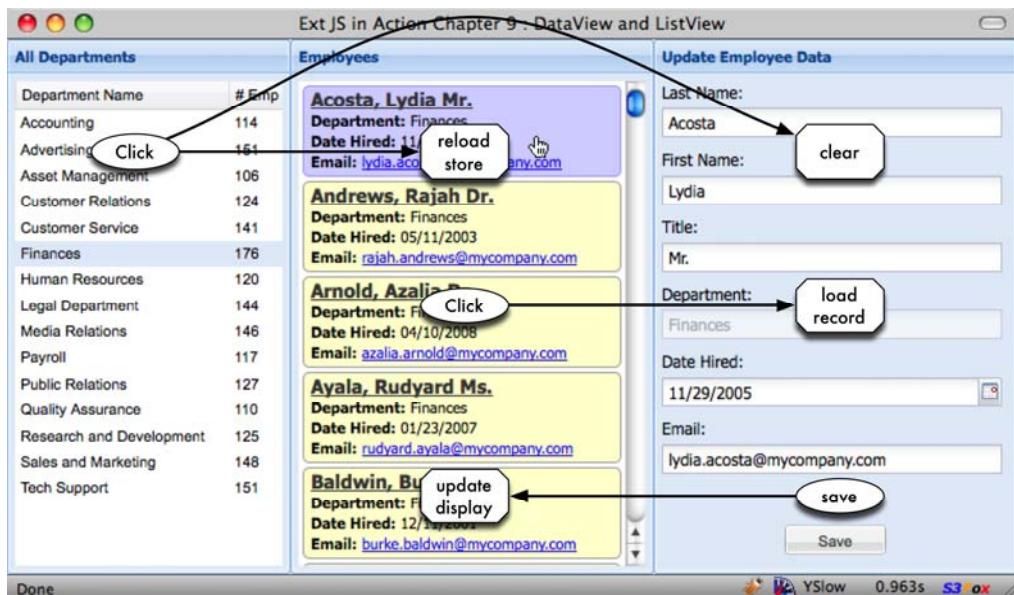


Figure 9.9 The addition of the Form panel and the required event model.

To populate the FormPanel with the selected data, the DataView will have to be reconfigured with click handler to get the job done. Also, the FormPanel's save button will be required to set the various properties on the record to update the DataView. Lastly, the Departments

ListView will have to clear the FormPanel's values when it is clicked to avoid an exception. It sounds like a lot of work, but it's really not. We have most of it done already.

Lets begin by creating the FormPanel and adding it to the Viewport. Then we'll circle back and work on the bindings.

#### 9.4.1 Configuring the FormPanel

The following listing will be lengthy, but it's mainly due to the number of fields that need to be in the form. The code below needs to be placed just before the instance of Viewport is created.

##### **Listing 9.8 Constructing the FormPanel**

```
var updateForm = {
    frame      : true,
    id         : 'updateform',
    labelWidth : 70,
    xtype      : 'form',
    defaultType: 'textfield',
    buttonAlign: 'center',
    title      : 'Update Employee Data',
    labelAlign  : 'top',
    defaults   : {
        anchor : "-5"
    },
    items      : [
        {
            name      : 'lastname',
            fieldLabel: 'Last Name'
        },
        {
            name      : 'firstname',
            fieldLabel: 'First Name'
        },
        {
            name      : 'title',
            fieldLabel: 'Title'
        },
        {
            name      : 'department',
            fieldLabel: 'Department',
            disabled  : true
        },
        {
            xtype     : 'datefield',
            name     : 'datehired',
            fieldLabel: 'Date Hired'
        },
        {
            name      : 'email',
            fieldLabel: 'Email'
        }
    ]
};
```

Above, we create the FormPanel with six fields. Notice that each field has a name attribute that matches the corresponding Record fieldname. This is important to know because if we

want to load the record, the field names must match the records that it will be configured to consumed.

The department field is disabled because the users have indicated that they don't want to use this tool to update departments. They want a drag and drop tool for that, which we'll be tasked later on to build.

Next, we'll place it in the Viewport. Here's what the modified items array will look like:

```
items : [
    {
        title : 'All Departments',
        frame : true,
        layout : 'fit',
        items : departmentLV,
        flex : null,
        width : 210
    },
    {
        title : 'Employees',
        frame : true,
        layout : 'fit',
        items : employeeDv,
        flex : 1
    },
    updateForm
]
```

Notice that all we're really doing is adding the updateForm reference to the end of the list, which will make it appear to the right of the employees DataView. Here's what it will look like rendered.

All Departments		Employees	Update Employee Data
Department Name	# Emp		Last Name:
Accounting	114	<b>Alford, Ross</b> Department: Media Relations Date Hired: 03/02/2008 Email: ross.alford@mycompany.com	First Name:
Advertising	151	<b>Ashley, Ferdinand Mrs.</b> Department: Media Relations Date Hired: 07/18/2007 Email: ferdinand.ashley@mycompany.com	Title:
Asset Management	106	<b>Ayers, Breanna Dr.</b> Department: Media Relations Date Hired: 06/01/2002 Email: breanna.ayers@mycompany.com	Department:
Customer Relations	124	<b>Bailey, Adam Dr.</b> Department: Media Relations Date Hired: 06/24/2008 Email: adam.bailey@mycompany.com	Date Hired:
Customer Service	141	<b>Banks, Hope</b> Department: Media Relations Date Hired: 04/07/2001 Email: hope.banks@mycompany.com	Email:
Finances	176		
Human Resources	120		
Legal Department	144		
Media Relations	146		
Payroll	117		
Public Relations	127		
Quality Assurance	110		
Research and Development	125		
Sales and Marketing	148		
Tech Support	151		

Done      YSlow 1.694s S3 ox

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Figure 9.10 The ListView (left), DataView (center) and FormPanel (right).

The FormPanel renders perfectly, which means we're ready to begin the bindings. This is where the fun begins.

#### 9.4.2 Applying the final bindings

We'll begin by binding the employees DataView to the FormPanel by means of a click handler, which will retrieve the selected record, and call upon the FormPanel's BasicForm to load the record. To assist with the task of binding the FormPanel to the DataView, we'll set a local reference to the selected record on the FormPanel itself.

To add the click handler to the employees DataView, we'll need to add the following listeners object to its configuration object.

```
listeners : {
    click : function(thisDv, index) {
        var record = thisDv.store.getAt(index);
        var formPanel = Ext.getCmp('updateform');
        formPanel.selectedRecord = record;
        formPanel.getForm().loadRecord(record);
    }
}
```

The above code snippet will work just like the click handler we assigned to the ListView just a bit earlier. The difference being that we're setting the record reference on the FormPanel and calling upon its BasicForm's loadRecord to set the values of the fields. Notice that the record reference that is being set is "selectedRecord". This will be key when we work on the round-trip binding.

We can test this by refreshing the page, selecting a department, then an employee. The record should load and the fields should be set with the data accordingly.

The screenshot shows a web application interface. On the left, there is a table titled "All Departments" with columns "Department Name" and "# Emp". The table lists various departments with their respective employee counts. In the center, there is a "Employees" section displaying four employee records in a list view. Each record contains the employee's name, department, date hired, and email. The first record, "Alford, Ross", is highlighted with a yellow background. To the right of the list view is a "Update Employee Data" form panel. This panel has fields for Last Name, First Name, Title, Department, Date Hired, and Email. The "Email" field contains "ross.alford@mycompany.com". The "Last Name" field also contains "ross.alford@mycompany.com", which appears to be a typo or a placeholder. The "First Name" field contains "Ross". The "Title" field is empty. The "Department" field contains "Media Relations". The "Date Hired" field contains "03/02/2008". The "Email" field contains "ross.alford@mycompany.com". There are arrows pointing from the highlighted record in the list view to the corresponding fields in the form panel.

Figure 9.11 Testing the DataView to FormPanel binding

Excellent! We have our one-way binding from the DataView to the FormPanel in place. In order to commit the changes, the users will need a save button. To do this, we'll have to add a buttons property to the FormPanel's configuration object with a single button, whose text is "save". The handler will be responsible for detecting leveraging the recently set FormPanel's selectedRecord and setting its value according to the changes made on the FormPanel.

Please add the following snippet to the FormPanel.

```
buttons : [
    {
        text : 'Save',
        handler : function() {
            var formPanel = Ext.getCmp('updateform');
            if (formPanel.selectedRecord) {
                var vals = formPanel.getForm().getValues();

                for (var valName in vals) {
                    formPanel.selectedRecord.set(valName, vals[valName]);
                }
                formPanel.selectedRecord.commit();
            }
        }
    }
]
```

When the newly configured button is pressed, it will use Ext.getCmp to get a reference to the FormPanel. If it contains a selectedRecord property, it will loop through each value retrieved from the form and set the record's corresponding property accordingly. Lastly, the record's

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

commit method is called, making the changes permanent and causes the DataView to only repaint the node that was modified.

All Departments		Employees	Update Employee Data
Department Name	# Emp		
Accounting	114		
Advertising	151		
Asset Management	106		
Customer Relations	124		
Customer Service	141		
Finances	176		
Human Resources	120		
Legal Department	144		
Media Relations	146		
Payroll	117		
Public Relations	127		
Quality Assurance	110		
Research and Development	125		
Sales and Marketing	148		
Tech Support	151		

http://ext2play - Ext JS in Action Chapter 9 : DataView and ListView

http://ext2play/extjs/chapter09/listview\_with\_dataview\_and\_form.html#

Figure 9.12 Testing the round trip binding of the FormPanel to the DataView.

As illustrated in Figure 9.10, clicking on “Media Relations”, then on “Alford Ross” will cause that employee’s record to populate the FormPanel. I added “Sr.” to the record, since the value was missing and clicked the newly added save button. What we can immediately see is that rendered record is updated based on the data from the FormPanel, completing the two-way binding between the DataView and FormPanel.

The last thing we need to do is code is the clearing of the FormPanel when the department ListView selection changes. This will help prevent exceptions that the user might cause if they attempt an update on a record that might not be there due to a department selection change.

Replace the previously developed click handler with the following code:

```
click : function(thisView, index) {
    var record = thisView.store.getAt(index);
    if (record) {
        Ext.StoreMgr.get('employeeDv').load({
            params : {
                department : record.get('department')
            }
        });
        var formPanel = Ext.getCmp('updateform');
        delete formPanel.selectedRecord;
    }
}
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
        formPanel.getForm().reset();  
    }  
}
```

By studying the changes, we can see that all we had to do was add three lines to create a reference to the FormPanel, delete the selectedRecord reference and then call the BasicForm's reset method, clearing the data off of the fields. We can observe this in action by selecting a department, employee, and finally selecting a different department. The fields will clear, signaling that it works.

This concludes the deliverable of binding DataView, ListView and FormPanel. When you encounter the need to implement a DataView or ListView for your application, remember that you can bind these two widgets to any other widgets by means of events. An example of this is a rowclick event of a GridPanel could trigger the data Store load of a DataView or ListView to provide a level of "drill-down" capability. Likewise, inverse binding can be applied as well.

## **Summary**

In this chapter we explored the two commonly used widgets, the DataView and its cousin the ListView. In doing so, we not only looked at how they work, but learned how to bind them together, applying advanced concepts that could be carried into any application.

We began by learning what a DataView is and what it takes to configure them. While learning about the DataView, we took some time to discuss a little bit about one of its supporting classes, the XTemplate. By example, we saw that we were required to create the CSS to style the painted data on screen.

Next, we learned about the ListView class, and the features it provides. We discussed the rules of the column-sizing model and learned how to bind the ListView to a DataView by means of a click event.

Lastly, we learned how to create two-way binding between a FormPanel and a DataView to allow our users to update the data for the employees. In doing so, we got to witness the efficiency of the DataView in updating only the DOM linked to the record that was updated.

Next, we'll tackle the use of charts and graphs and learn how to configure them to respond to click gestures.

# 10

## Charts

Many of the applications that I've developed in the past have had the need for data visualization. Whether it was the amount of space used for a particular disk volume or the value of a stock over time, the problems were exactly the same and ultimately amounted to a few major factors: What charting package will be used, how much is it going to cost to integrate and ultimately manage?

All of this, of course, introduces risk to the application and adds support costs, which dissatisfies customers. Thankfully, this is all now behind us. Ext JS 3.0 provides a charting package that handles the four most commonly used charts: line, bar, column and pie, thus mitigating the risk of using and supporting a third-party charting package and reducing support costs.

In this chapter, we're going to learn all about Ext JS charts. We'll begin by exploring the anatomy of charts and take some time discussing different classes that comprise the chart package. We'll also visually explore the different types of charts available to you.

Next, we'll take an exploratory approach when learning about the different charts in the framework. We'll begin by creating a basic LineChart and learn how to customize the colors used, create a custom tooltip renderer method and display legends.

Because the LineChart, ColumnChart and BarCharts are based off of the same basic chart, we'll use our newly found knowledge to convert the LineChart to create a ColumnChart and a BarChart by literally flipping the X and Y-axis configurations. We'll also learn how to create hybrid charts, such as ColumnCharts with embedded line series.

In the last section, we'll learn all about the PieChart and how it differs from the three other basic charts. We'll also learn how to add context to the PieChart tooltip by means of a custom tip renderer function, making it much more useful.

## 10.1 Defining the four charts

Charts provide the ability to visualize statistical data in a graphic format. There are three basic types of charts, Line, Bar and Pie. LineChart, BarChart, and ColumnChart are similar because they are based off the same *Cartesian* style chart, where points of data are plotted in a two-dimensional X-Y coordinate plane system.

It is for this reason that you can mix the bar and line series in the same chart. Likewise, you can mix line with the column in a chart. But Column and Bar series, however cannot exist in the same chart.

The PieChart is kind of the “black sheep” of the chart classes. This is because it is not based off of the Cartesian chart system, thus it can only leverage the PieSeries and is designed to display data representing percentages.

What makes Charts in the framework so cool is that you need not have any Flash experience to get this stuff to work for you. As we’ll see, everything will be done via JavaScript.

Let’s take a look at a Line chart in action and analyze it.

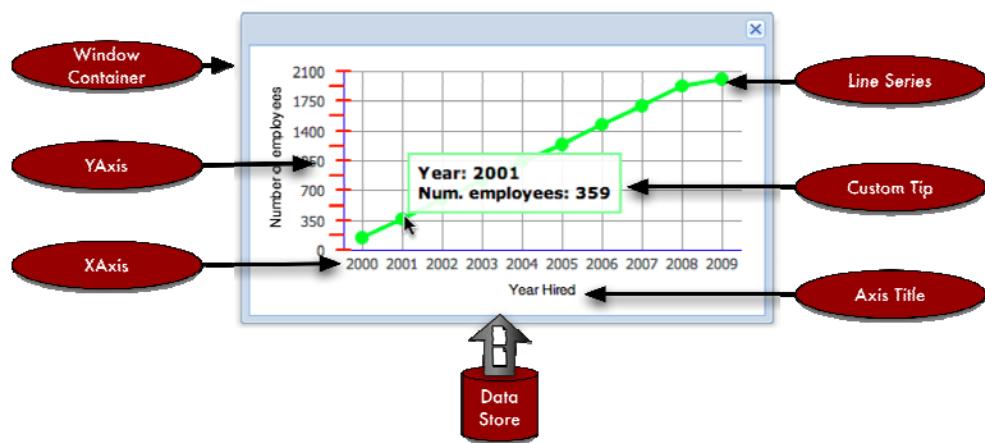


Figure 10.1 An example LineChart with a custom Tip.

ExtJS Charts are based off of the YUI Charts library, and are wired to consume data from a data Store, much like the GridPanel and DataView classes. Because charts follow the same pattern as every other data Store consumer class, providing data for this widget requires no additional learning. This binding also provides the benefit of automatic chart updates when data in the Store changes, which could be displayed with or without animation.

If desired, users have the ability to interact with the charts with the mouse by means of mouseover and click gestures. A mouseover gesture can provide useful information by means of a QuickTip, which can be customized as we see in Figure 10.1.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

The click gesture can provide powerful user interaction with your charts, allowing users to click a point in the LineSeries, bar in the BarSeries, column in the ColumnSeries or a slice of a PieSeries to make stuff happen in the UI, such as drilling down into that data point for a more fine-grain view of the data. This works by means of the Chart's itemclick event, which we'll explore and exercise when we start to build our first graphs.

Next, we'll look under the hood and peek at the components that make charts work.

## 10.2 Charts Deconstructed

Just like everything else UI, charts can be rendered to any div on the page or configured as a child item of any parent Container widget. This is because the chart classes are descendants of the BoxComponent class, which if you can recall, gives any component the ability to take part in a layout.

Here is a hierarchy diagram for the chart classes.

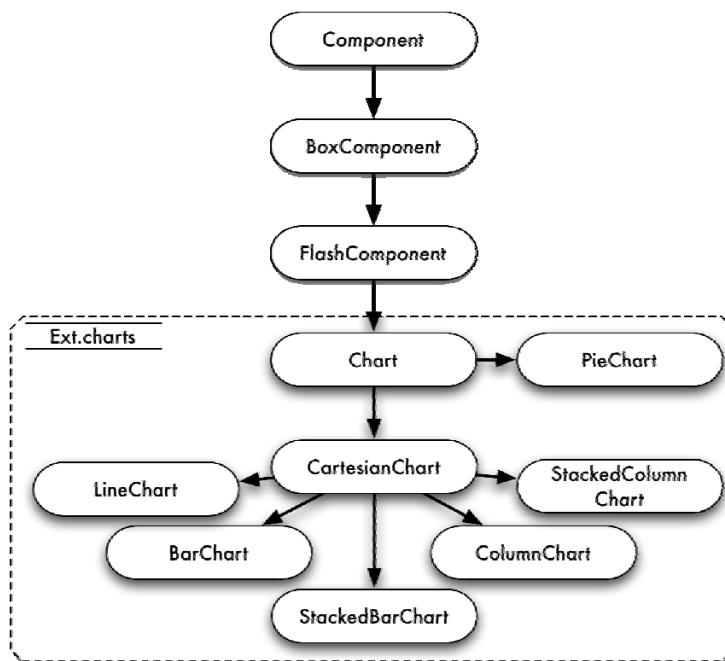


Figure 10.2 The Chart class hierarchy.

Charts are Adobe Flash-based and require special handling when compared to generic HTML objects. For instance, if Flash is to be displayed on a page, logic needs to be in place to detect whether or not the browser has Flash and if it's the correct version. If Flash is not installed or if the version is less than what is required, a link to install flash is rendered in the ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

area where the Flash content is to be, else the Flash widget is rendered on screen and all is well. Managing this type of logic, in addition to the cross-browser markup differences, can be quite a hassle to manage on your own.

Luckily, the FlashComponent takes care of this for us and serves as the base class for the Charts. The FlashComponent is a special class because it leverages the SWFObject to handle just about all Flash-specific issues and can be extended to allow Ext JS to manage any Flash-based content. It is for this reason that the Chart class subclasses FlashComponent.

If we focus on the Chart class, which exists in the Ext.charts namespace, we can see that it has two descendants, PieChart and CartesianChart. The Chart class is what wraps the YUI charts and is what binds the data Store and mouse gesture events with the framework. It is the most heavyweight class in the entire hierarchy. In fact, most of the work is done in the compiled YUI charts compiled SWF, which is beyond the scope of this book.

Looking at the descendants of CartesianChart, we can see three major categories of charts. LineChart, BarChart and ColumnChart. This is important to know when deciding how to implement them, as they are the three main Cartesian-chart options available.

The LineChart is used to plot dots of intersecting data on the X-Y plane and a line is drawn connecting those dots. The figure below illustrates a line chart implementation that contains two LineSeries.

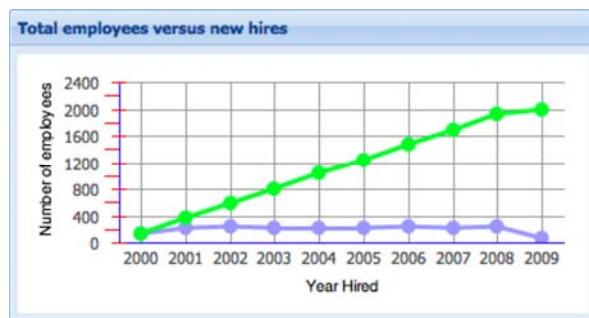


Figure 10.3 The LineChart with two LineSeries

The BarChart displays horizontal bars of data. Multiple BarSeries can be added to the chart, where for each category of data, will be flush against each other as illustrated in the figure below.

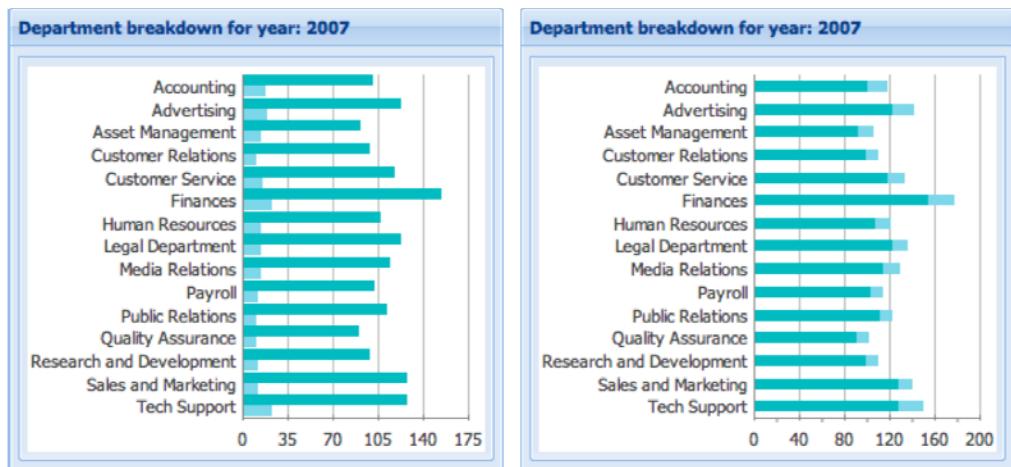


Figure 10.4 Bar (left) and StackedBar (right) chart examples

Sometimes data to be displayed is cumulative. For instance, the total number of employees for a given year can be calculated as the sum of the previously hired and newly hired for a given year. This data can must be displayed in a single horizontal line, but segmented with the StackedBarChart, as seen in Figure 10.4 (right). The data used for both of these charts is exactly the same.

The ColumnChart is used to plot the chart data in vertical bars. The columns can be stacked via the StackedColumnChart. Just like Figure 10.4, the data for the illustration below is the same.



Figure 10.5 The ColumnChart (left) and StackedColumnChart (right).

In contrast to the Cartesian-style charts, the PieChart is only used to display percentages of data. Below is an illustration of a PieChart in action.



Figure 10.6 A PieChart in action.

Just as with the Line, Column and Bar charts, the series for the PieChart (slices) can display a tip when a mouseover gesture occurs. They can also be clicked, which will generate the useful itemclick event.

We now have a high-level understanding of what charts are available to us and how they work, which means that we can begin the code exploration of the different charts. We'll kick it off with the construction of a LineChart, where we'll learn the basics of chart creation.

### 10.3 Building a LineChart

Our first chart will be a single-series LineChart that plots the total number of employees for a company from the year 2000 to 2009. Remember that this type of chart is based off of X and Y coordinates, which means that we'll have to configure the chart to utilize a data Store-mapped data field for each axis.

But, before we actually begin writing the code for this chart we'll have to take a look at the data the chart will consume. Relatively speaking, this is an easy task, since we're only dealing with two data points; year and total employees for each year.

The server side has been crafted for us and the JSON will look like this:

```
[
  {
    "newHires" : 135,
    "year"     : 2000,
    "total"    : 136,
    "prevHired": 1
  },
  ...
]
```

The JSON contains an array of objects representing records in the database. We can see the year and total attributes for each record. This means that we need to craft a JsonStore that will map these two fields. There are two other attributes per record, that report the number of new hires (newHires) and previously hired (prevHired) employees that we'll ignore for the time being.

Before we move on to construct our first chart, we need to configure the location of the "charts.swf" file, which defaults to the YAHOO! url:

```
http://yui.yahooapis.com/2.7.0/build/charts/assets/charts.swf
```

While leaving the default location of the above URL is generally OK, many would prefer not to depend on the availability of YAHOO to host the SWF content. Not to mention, if the application you're developing is going to be riding on HTTPS, the above URL would most likely cause the browser to display a security warning, which can give the false message that your application is less secure.

Also, if your application is within a secured intranet, the client's browser may not have access to outside web servers, thus rendering charts completely unusable. Configuring charts to require the SWF file from the local intranet server will help mitigate this issue.

To configure Ext JS to use a particular SWF for charts, set the CHART\_URL property on the Chart class as such:

```
Ext.chart.Chart.CHART_URL = '<path to extjs>/resources/charts.swf';
```

You want to place this line of code just after the Ext JS base libraries. This will ensure that the property is set before your code begins to get parsed by the JavaScript interpreter. This is very familiar to setting the BLANK\_IMAGE\_URL ('s.gif') configuration property earlier on in this book.

With that out of the way, we can move on to create the line chart that will plot this data on screen.

### **Listing 10.1 Constructing a basic line chart.**

```
var employeeStoreProxy = new Ext.data.ScriptTagProxy({
    url : 'http://extjsinaction.com/examples/chapter10/employeeData.json.js'
});

var remoteStore = {
    xtype : 'jsonstore',
    autoLoad : true,
    storeId : 'employeeDv',
    proxy : employeeStoreProxy,
    fields : [
        { name : "year", mapping : "year" },
        { name : "total", mapping : "total" }
    ]
};

var chart = {
    xtype : 'linechart', // 1
    store : remoteStore, // 2
    xField : 'year',
    yField : 'total'
};

new Ext.Window({
    width : 400,
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

height : 400,
layout : 'fit',
items : chart
}).show();
{1} Configure the LineChart XType
{2} Setting the X and Y axis fields

```

In Listing 10.1, we create the data Store that maps the year and total fields that are provided by the server side code and the Line chart that lives in a window. As you can see, creating a basic chart is extremely simple to do. Here's how it works.

When creating the LineChart XType configuration object{1}, we set the mandatory xtype property and required JSON data store. The next two properties, xField and yField are used to automatically create the X and Y-axis and map the data from the Store. That's pretty much it.

To display the chart, we create a Window with the fit layout, which hosts the LineChart. Here's what our first LineChart looks like in action.

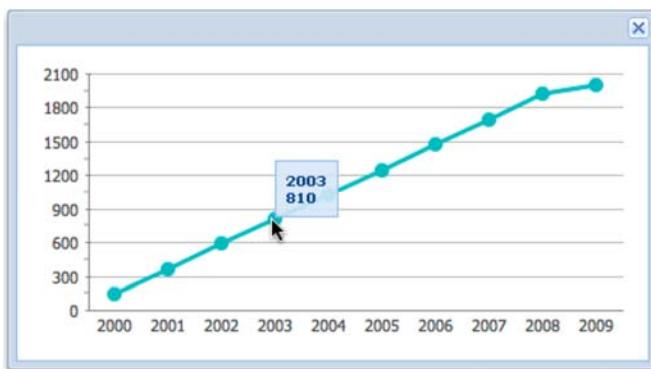


Figure 10.7 Our first LineChart in Action.

As illustrated in Figure 10.7, our line chart renders inside of an Ext.Window plotting the statistical data. If you hover the mouse cursor over one of the points in the line, you'll see the two values for the point in a tip. But, this presents a problem.

For this chart, the data in the tip is provides no context to the figures being presented, thus provides little value. Likewise the X and Y-axis have the same dilemma. It's obvious that the number on the X-axis represents years. However, you can't tell what the Y-axis represents. This is precisely a situation where you'd want to customize the chart.

### 10.3.1 Customizing the ToolTip

The first problem we'll tackle is providing context the tooltip. To do this, we'll have to add a tipRenderer to our LineChart configuration object as such:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
tipRenderer : function(chart, record, index, series){
    var yearInfo = "Year: " + record.data.year;
    var empInfo = 'Num. employees: ' + record.data.total;

    return yearInfo + '\n' + empInfo ;
}
```

In the tipRenderer method above, we create two strings, yearInfo and empInfo, which insert a meaningful label in front of the data. The method returns a concatenation of the yearInfo and empInfo with a new line character “\n” to introduce a line break between the two data fields. Why a new line character and not a standard “<br />” you ask?

The tipRenderer function works very similarly to the column renderer of the GridPanel, where the method is expected to return a string to be displayed on screen, however while the Column renderer allows HTML, the tipRenderer does not. This is because the tip is rendered inside of Flash, and does not support HTML. In order to introduce a line break between lines of data, the only way to do so is by means of the standard UNIX-style new-line string (“\n”).

Here's what the new tooltip will render:

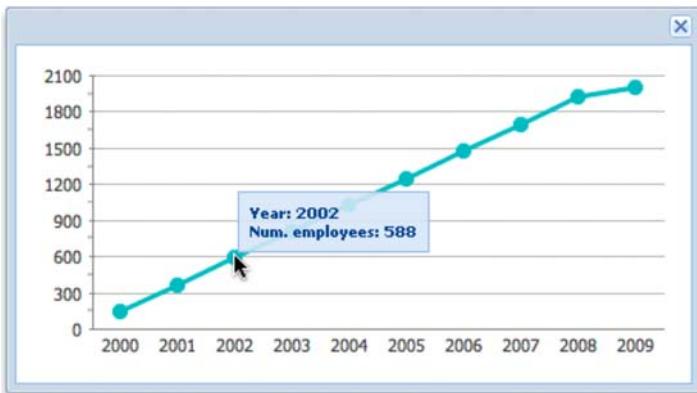


Figure 10.8 Our LineChart with our custom tooltip.

With the addition of the custom tipRenderer, mouse over events now provide context to the tips. Next, we'll work on customizing the X and Y-axis to add labels.

### 10.3.2 Adding titles to the X and Y-axis

To add labels to a Cartesian chart, you must manually configure and create axis. However, you must use the correct axis for the type of data you are displaying. There are three possible axis to use: CategoryAxis, NumericAxis and TimeAxis. Base on the line chart we've created thus far, which axis would you choose for each X and Y-axis?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

The values for each axis are numeric. The obvious choice is NumericAxis for both - right? Wrong. Here's why.

If we took a good look at how the Cartesian charts process data, we can tell that the charts display two types of values measurements and something that is being measured. The "thing" that is measured in each chart is known as the category.

Applying this to the line chart, we see that the measurement for our chart is the number of employees and the category is the year. The exact same logic can be applied to a chart that measures how many sales car manufacturers have made.

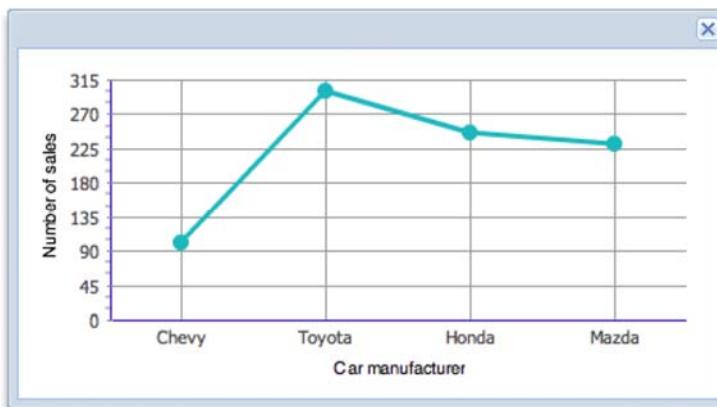


Figure 10.9 Categories of data are clearly defined in this car sales LineChart.

With this knowledge now in mind, the CategoryAxis is logically the only choice for the X-axis. This means we'll have to reconfigure the chart to utilize new X and Y axis.

### **Listing 10.2 Reconfiguring the chart to utilize new X and Y-Axis.**

```
var chart = {
    xtype      : 'linechart',
    store      : remoteStore,
    xField     : 'year',
    yField     : 'total',
    tipRenderer: function(chart, record, index, series){
        var yearInfo = "Year: " + record.data.year;
        var empInfo = 'Num. employees: ' + record.data.total;
        return yearInfo + '\n' + empInfo ;
    },
    xAxis: new Ext.chart.CategoryAxis({ // 1
        title : 'Year Hired' // 2
    }),
    yAxis: new Ext.chart.NumericAxis({ // 3
        title : 'Number of employees'
    })
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

- {1} Set the new X-axis.
- {2} Configure the X-axis title.
- {3} Use the NumericAxis for the Y-axis.

In the above code snippet, we add the `xAxis` and `yAxis` properties to the main chart configuration object. The `xAxis` is set to an instance of `CategoryAxis` while the `yAxis` property is set an instance of `NumericAxis`. Each new axis is configured with a `title` parameter.

Let's see what it looks like after we've added the newly overridden axis.

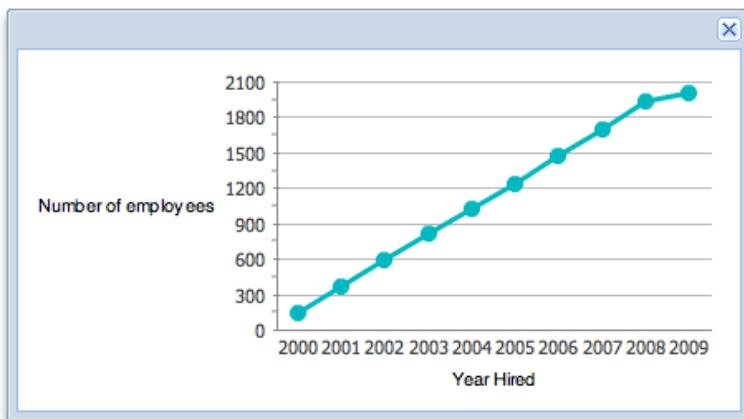


Figure 10.10 Adding Y Axis titles without the benefit of custom styling will result in unwanted wasted screen space.

If we look at the Y-axis label, we see that it's clearly wasting space. The only way to mitigate this issue this is rotate the text, which can only be done through custom styling of this widget.

Next, we'll learn how to stylize the chart body. Later on, we'll learn how to stylize the series as well.

### 10.3.3 Styling the Chart body

In contrast to the rest of the framework, JavaScript is the only vehicle to stylize charts. This means that you must configure the styles of the charts when you instantiate them. To me, this is one of the more lengthy tasks when developing charts as it requires a lot of testing and ultimately more code.

#### NOTE

There are quite a lot of style options available to choose from. Being that Ext JS uses YUI charts, naturally, the best source for documentation is going to be at YUI. Visit the

following URL to get the most comprehensive list of styles:  
<http://developer.yahoo.com/yui/charts/#basicstyles>

In order to rotate the title, we're going to have to include custom style configuration in our line chart's configuration object. This is done by means of setting the `extraStyle` property as such:

```
extraStyle : {
    yAxis: {
        titleRotation : -90,
    }
}
```

In the above code snippet, we create the `extraStyle` configuration object, which contains another configuration object for the Y-axis. Inside of the `yAxis` configuration object, the `titleRotation` property is set to -90 (degrees).

Let's see what the newly styled chart looks like.

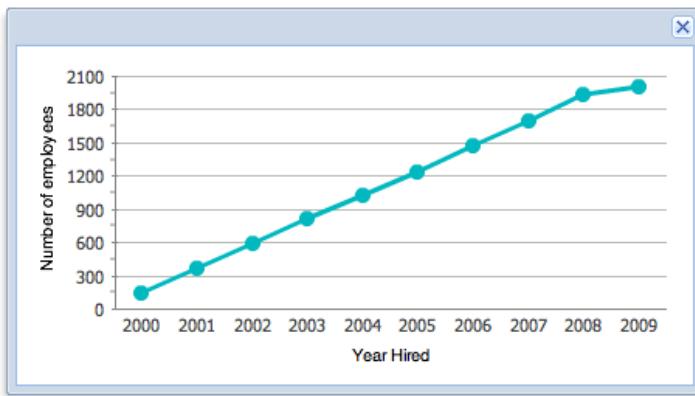


Figure 10.11 The `yAxis` is now rotated -90 degrees, which reduces wasted white space.

We can see that by simply adding the above `extraStyle` configuration property and its contents to our `LineChart` enable it to render with the Y-Axis title rotated, thus reaching our goal of reducing the amount of wasted space.

Thus far, we've created a single line chart and have provided some light styling to improve its readability. There are a few more measurement figures to add to the chart. To show these on the screen, we'll need to refactor the store and chart configuration completely.

## 10.4 Adding Multiple Series

When we looked at the data being provided by the server side, we saw multiple data points that are related to the number of employees for a given year. If we peek at the data again, we can see the other two figures are newHires and prevHired.

```
[  
  {  
    "newHires" : 135,  
    "year" : 2000,  
    "total" : 136,  
    "prevHired" : 1  
  },  
  ...  
]
```

To make use of this data, we'll need to reconfigure the data Store to map the fields. Luckily, all we have to do is add the two fields to the mappings list for the data Store.

```
fields : [  
  { name : "year", mapping : "year" },  
  { name : "total", mapping : "total" },  
  { name : "newHires", mapping : "newHires" },  
  { name : "prevHired", mapping : "prevHired" }  
]
```

Next, comes the fun part: refactoring the chart to add the extra series. While we're at it, we'll apply custom styles to make the much easier to read and more pleasing to look at. This is where we'll see that customizing charts requires quite a bit of code, and is easy once you understand it. It is for this reason that we'll break up the refactoring effort into smaller, more digestible chunks.

We'll begin with the configuration of the series.

### Listing 10.3 Configuring the series for our multi-series chart.

```
var series = [  
  {  
    yField : 'prevHired', // 1  
    displayName : 'Previously Hired', // 2  
    style : { // 3  
      fillColor : 0xFFFFAA,  
      borderColor : 0xAA3333,  
      lineColor : 0xAA3333 // 4  
    }  
  },  
  {  
    yField : 'total', // 5  
    displayName : 'Total', // 6  
    style : {  
      fillColor : 0xAAAFFF,  
      borderColor : 0x3333FF,  
      lineColor : 0x3333FF // 7  
    }  
  },  
  {  
    yField : 'newHires'  
  }  
];
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        yField      : 'newHires',
        displayName : 'New Hires',
        style       : {
            fillColor    : 0xAAFFAA,
            borderColor  : 0x33AA33,
            lineColor    : 0x33AA33
        }
    }
];
{1} Map the data point to the series
{2} Display name used for tips and legends
{3} Custom style for each series
{4} Set the point's fill color
{5} Specify the point's border color
{6} Configure the line color

```

In listing 10.3, we create an array of configuration objects that are used to configure the three different series. Here's what all of these properties do.

In each series-configuration object, we map the data points by means of the `yField{1}` property. Setting the `yField` property replaces the `yField` property in the main chart configuration object. It may be easier to think of it like mapping the `dataIndex` to `GridPanel` columns. We also set the `displayName{2}`, which is used in both the tooltip, and legend, which we'll learn about in just a bit.

Next, we set a custom `style{3}` configuration object for each series, which sets the color for the point's `fill{4}` and `border{5}` colors as well as the color for the `lines{6}`. Setting these styles will clearly distinguish each series, enhancing chart readability.

With the series in place, we can move on to construct a `tipRenderer` that is much more flexible and create the `extraStyle` configuration.

#### **Listing 10.4 Creating a flexible tipRenderer and extraStyle.**

```

var tipRenderer = function(chart, record, index, series){
    var yearInfo = "Year: " + record.get('year');
    var empInfo  = series.displayName + ': ' + record.get(series.yField); //1
    return yearInfo + '\n' + empInfo ;
};

var extraStyle = {
    xAxis : {
        majorGridLines : {                                         // 2
            color : 0x999999,
            size  : 1
        }
    },
    yAxis: {
        titleRotation : -90
    }
};
{1} Use the series displayName to generate a label
{2} Add vertical lines to the chart

```

In listing 10.4, we create the custom tipRenderer, which is much more flexible than the previous one. This is because this tipRenderer leverages the series' displayName{1} to create the custom label for the series. It also uses the series' yField property to pull the mapped data, resulting in a truly dynamic tipRenderer.

Next, we create a configuration object to contain the custom style parameters. Along with the yAxis's titleRotation, we add a xAxis configuration object, which contains majorGridLines. This configuration property instructs the chart to display a vertical line for each category data point, resulting in intersecting lines across the entire grid area.

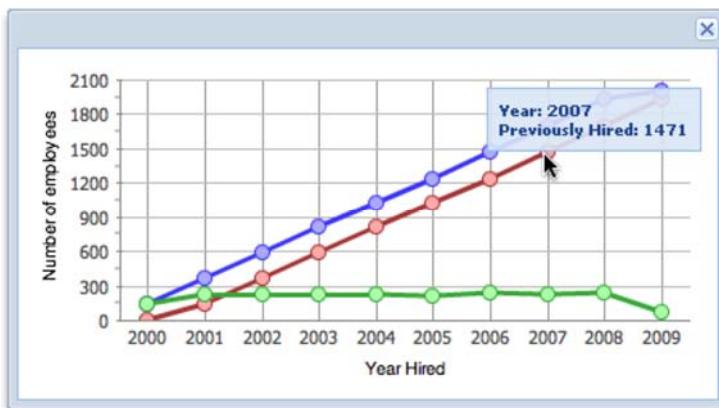
We now have all that we need to configure the chart. Lets do that next.

### **Listing 10.5 Refactoring the chart**

```
var chart = {
    xtype      : 'linechart',
    store      : remoteStore,
    xField     : 'year',
    tipRenderer: tipRenderer,
    extraStyle : extraStyle,
    series     : series,
    xAxis      : new Ext.chart.CategoryAxis({
        title : 'Year Hired'
    }),
    yAxis      : new Ext.chart.NumericAxis({
        title : 'Number of employees'
    })
};
```

To refactor the chart configuration object, we remove the yField property. Remember, that these are already present in each of the series configuration objects, thus is not needed in the chart configuration. We also set the tipRenderer, extraStyle and series properties to the variables that we created just a bit earlier.

Here's what the chart now looks like rendered inside of the Window we created earlier.



©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Figure 10.12 Our multi-line chart with much more custom styling.

When looking at our multi-series line chart we can see that each series is colored differently and the vertical lines are now drawn at each X-axis data point. But we're left with a problem. In order to tell what figure each line represents, we have to mouse over a point on the graph, which is not exactly user friendly.

The solution for this is to add a Legend to the chart.

#### 10.4.1 Adding legends

To add a legend to your chart, you need only add the style configuration object to the extraStyles configuration object as such:

```
legend : {
    display : "bottom",
    padding : 5,
    spacing : 2,
    font    : { color : 0x000000, family : "Arial", size   : 12 },
    border  : { size : 1, color : 0x999999 }
}
```

When setting the legend style configuration object, the property that controls whether the legend will display or not is `display`. This property defaults to "none", which prevents it from displaying. In addition to "bottom", it can be set to "top", "right" or "left".

Additionally, the `padding` property works much like the CSS `padding` style. The `spacing` property specifies how many pixels separate each of the series. To configure the font and border, we have to create separate configuration properties for each. Remember that all of the possible styles are detailed in the YUI documentation.

Here's what the addition of the legend looks like.



Figure 10.13 The addition of the Legend enhances readability for multi-series charts.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

We've spent a lot of time on the LineChart and learned how to customize it quite a bit with the inclusion of a custom tipRenderer, multiple series and applicable styles. What about the column and bar charts? How does the construction of each differ from the LineChart?

## 10.5 Constructing ColumnCharts

If you're comfortable with constructing LineCharts then you'll be happy to learn that relative to the LineChart, the construction of a ColumnChart is nearly identical. The biggest difference being that instead of setting the xtype to 'linechart', you set it to 'columnchart'.

Here is the LineChart refactored into a ColumnChart.

### **Listing 10.5 Creating a column chart**

```
var chart = {
    xtype      : 'columnchart',
    store      : remoteStore,
    xField     : 'year',
    tipRenderer: tipRenderer,
    extraStyle : extraStyle,
    series     : series,
    xAxis      : new Ext.chart.CategoryAxis({
        title : 'Year Hired'
    }),
    yAxis      : new Ext.chart.NumericAxis({
        title : 'Number of employees'
    })
};
```

It's that simple. Here's what the ColumnChart looks like rendered on screen.



Figure 10.14 A multi-series column chart.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

When looking at this chart, it's hard to see that the newly hired + previously hired employees equals the total employees. This is where the StackedColumnChart fits in.

### 10.5.1 Stacking Columns

We just learned that the conversion from a LineChart to a ColumnChart was relatively simple. But to make this chart much easier to read, we'll need to stack the columns. The conversion from a ColumnChart (or LineChart for that matter) is a bit more involving.

To make this conversion, we'll need to refactor the series array to remove the series representing the total number of employees, as it is unnecessary.

```
var series = [
    {
        yField      : 'prevHired',
        displayName : 'Previously Hired',
        style       : {
            fillColor   : 0xFFAAAA,
            borderColor : 0xAA3333,
            lineColor   : 0xAA3333
        }
    },
    {
        yField      : 'newHires',
        displayName : 'New Hires',
        style       : {
            fillColor   : 0xAFFAA,
            borderColor : 0x33AA33,
            lineColor   : 0x33AA33
        }
    }
];
```

Next we will need to refactor the chart configuration object a little bit.

```
var chart = {
    xtype      : 'stackedcolumnchart',
    store      : remoteStore,
    xField    : 'year',
    tipRenderer: tipRenderer,
    extraStyle: extraStyle,
    series     : series,
    xAxis      : new Ext.chart.CategoryAxis({
        title : 'Year Hired'
    }),
    yAxis      : new Ext.chart.NumericAxis({
        stackingEnabled : true,
        title          : 'Number of employees'
    })
};
```

In the newly refactored chart configuration object above, we set the xtype to "stackedcolumnchart". The only other change we made was the addition of the stackingEnabled property to the Y-NumericAxis. These are the only changes necessary to construct a StackedColumnChart.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Here's what it looks like painted on screen.



Figure 10.15 Our StackedColumnChart in action.

When analyzing the results of the StackedColumnChart, we can see that the previously and newly hired employee figures appear stacked, amounting the total number of employees for a given year. This is exactly what we wanted to do. But we introduced a problem. How does one find the total number of employees for a given year? Let us ponder the possibilities.

If left the "total" (StackedColumn) series in, then the sum of the columns would have doubled, which is undesirable. Surely there has to be a way to allow the users to see the total.

One way is to display the total could be via the tooltip. While this would work, it's always best to keep the tip within the context of the series being hovered over. We need something to tie the totals together.

A line series would work perfectly for this situation.

### 10.5.2 Mixing line with Columns

Thus far, we've exercised the ability to leverage multiple series in a chart, but have not looked into configuring a hybrid chart. That is, a chart with multiple types of series. It's much easier than you think.

To add a line series, simply add the following "LineSeries" style configuration object to the series array that we used to setup the StackedColumnChart.

```
{
    type      : 'line',
    yField   : 'total',
    displayName : 'Total',
    style    : {
        fillColor  : 0xAAAAFF,
        borderColor : 0x3333FF,
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        lineColor : 0x3333FF
    }
}

```

Notice that the configuration object for this newly added series is almost identical to that of the “total” series we removed when initially configuring the StackedBarChart. The item to key in on here is the `type` property, which is set to “line”.

Setting this property causes the framework to use a specific “type” of Series to display in a chart. It’s the XType equivalent for charts and saves us time from having to directly instantiate an instance of `chart.LineSeries`. To ensure that this line will render on top of the other series, it’s important that you add this LineSeries configuration as the last object of the series configuration objects array. How the series are rendered on screen is directly proportional to the order that they are placed in the `series` array. To put this in context, just think about how z-order with CSS and it will all make sense. The order of the series also affects how they appear in the legend as well.

Lets look at the newly configured hybrid Line and ColumnChart in action.



Figure 10.16 Our hybrid Line and ColumnChart in action.

After adding the `LineSeries` to the `StackedColumnChart`, we can easily access the “total” figure data by hovering the mouse cursor over the points rendered by the newly added `LineSeries`.

We’ve explored the depths of the `ColumnChart` and `StackedColumnChart` by means of converting the previously constructed `LineChart`. Along the way, we customized the `StackedColumnChart` by adding a `LineSeries`.

Next, look at creating a BarChart by converting our `StackedColumnChart`.

## 10.6 Constructing BarCharts

To construct a BarChart, we can use all of the exact same plumbing that we created before, with some tweaks. When thinking about BarCharts, you just have to think about swapping the LineChart or ColumnChart X and Y-Axis. Why?

Remember that the LineChart and ColumnChart use the category data on the X-axis, while the measurement figures are to be on the Y-axis. The BarChart expects the Category data to be on the Y-Axis and the measurement data to be on the X-Axis.

Using this logic, we can refactor the StackedColumnChart to a StackedBarChart quite easily. This means that all we need to do is refactor the series to leverage data on the X-axis, modify the chart configuration object a little bit.

### Listing 10.6 Configuring the series for the StackedBarChart

```
var series = [
    {
        xField      : 'prevHired',
        displayName : 'Previously Hired',
        style       : {
            fillColor   : 0xFFAAAA,
            borderColor : 0xAA3333,
            lineColor   : 0xAA3333
        }
    },
    {
        xField      : 'newHires',
        displayName : 'New Hires',
        style       : {
            fillColor   : 0xAFFFAA,
            borderColor : 0x33AA33,
            lineColor   : 0x33AA33
        }
    },
    {
        type       : 'line', // 2
        xField     : 'total',
        displayName: 'Total',
        style       : {
            fillColor   : 0xAAAAFF,
            borderColor : 0x3333FF,
            lineColor   : 0x3333FF
        }
    }
];
{1} Configure the data point for the X-axis
{2} Embed a line series into this bar chart
```

In the listing above, the series and styles remain the same. The difference is that instead of setting the `yField` property for the configuration objects, we're setting `xField{1}`. We also keep the `LineSeries{2}` configuration object in tact as well.

In order for the custom and dynamic tipRenderer to work, we'll need to modify it a little to read the `xField` property of the series configuration definitions.

```
var tipRenderer = function(chart, record, index, series){
    var yearInfo = "Year: " + record.get('year');
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var empInfo = series.displayName + ': ' + record.get(series.xField);
return yearInfo + '\n' + empInfo ;
};

```

In the above snippet, the tipRenderer method is nearly identical to the previous one with the replacement of “series.yField” with “series.xField”. Remember, we did this because of the series configuration changes we made.

Next, we’ll refactor the chart configuration object, completing the transformation.

### **Listing 10.7 Configuring the StackedBarChart**

```

var chart = {
    xtype      : 'stackedbarchart',                                // 1
    store      : remoteStore,
    yField     : 'year',
    tipRenderer: tipRenderer,
    extraStyle: extraStyle,
    series     : series,
    xAxis      : new Ext.chart.NumericAxis({                      // 2
        stackingEnabled: true,                                     // 3
        title         : 'Number of employees'
    }),
    yAxis      : new Ext.chart.CategoryAxis({                      // 4
        title         : 'Year Hired'
    })
};

{1} Set the type to "stackedbarchart"
{2} Configure the X-Axis
{3} Enable stacking on the X-Axis
{4} Configure the Y-axis

```

To configure the configuration object for a StackedBarChart, we first have to set the `xtype` property **{1}** to “stackedbarchart”. The next change is setting the `xAxis` property to **{2}** to leverage the `NumericAxis`, with `stackingEnabled`**{3}**. Lastly, we set the `yAxis` property to an instance of `CategoryAxis`, removing the previously defined `stackingEnabled` property.

That’s pretty much all that is needed to convert this chart over. Here’s what it looks like rendered in a browser.



Figure 10.17 The StackedBarChart in action.

Great, everything works as expected, including the dynamic tooltip. How would one convert this to a generic BarChart? The steps are as simple as unraveling the configuration parameters to enable bar stacking.

### 10.6.1 Configuring a BarChart

The steps to modify our StackedBarChart to a BarChart are short and simple. First, change the chart configuration object xtype property to 'barchart' as such:

```
xtype : 'barchart',
```

Next, remove the stackingEnabled property from the X (NumericAxis) axis configuration object:

```
xAxis : new Ext.chart.NumericAxis({  
    title : 'Number of employees'  
}),
```

And that's it. Here's what the BarChart looks like with the integrated LineSeries.



Figure 10.18 The BarChart with an integrated LineSeries.

With the changes we've made, we can see that the "newHires" and "prevHired" columns both sit side by side and the line. If you want to remove the LineSeries, for the 'total' measurement, all you need to do is remove the type attribute, and Ext JS will use that configuration object to create a BarSeries.

We've just seen the similarities between the LineChart, ColumnChart and BarChart. The PieChart, however, is completely different as it is not a descendant of the Cartesian line of charts.

Next, we'll explore this final chart type and learn how to apply some customizations.

## 10.6 A slice of PieChart

As with all charts, when considering the implementation of a PieChart, one must consider the data that the chart will consume and display. The PieChart is relatively simple, as it only really works with two pieces of information: the category and related numeric data. If thought of in simplistic terms, the category is the name for the slice of pie and the numeric data determines how large that slice will be relative to the other figures in the data set.

For the PieChart we're going to build, here's what the data will look like.

```
[  
  {  
    "total" : "42",  
    "range" : "20,000+'s"  
  }  
  ...  
]
```

We're going to build a PieChart that will display data describing the number of employees for a particular salary range. In the JSON data above, the "category" will be the salary range,

while the “data” will be the “total” property of each record. To put this in plain English: in the example record above, there are 42 employees within the “20,000’s” range.

Having this knowledge in hand, constructing a data Store to consume this is trivial.

### **Listing 10.8 Creating the data Store for the PieChart**

```
var remoteProxy = new Ext.data.ScriptTagProxy({
    url : 'http://extjsinaction.com/examples/chapter10/salaryRanges.php'
});

var pieStore = new Ext.data.JsonStore({
    autoLoad : true,
    proxy    : remoteProxy,
    id       : 'piestore',
    fields   : [
        { name : "total", mapping : "total" },
        { name : "range", mapping : "range" }
    ]
});
```

Next, we can construct the PieChart to consume the data. We’ll do so with some custom styling to depict the ranges with varying colors from green to red.

### **Listing 10.9 Creating the PieChart with a legend**

```
var pieChart = {
    xtype      : 'piechart',
    store      : pieStore,
    dataField  : 'total',                                // 1
    categoryField: 'range',
    series     : [
        {
            style : {
                colors : [0xB5FF6B, 0xFFFF6B, 0xFFB56B, 0xFF6B6B]           // 3
            }
        },
        extraStyle : {
            legend : {                                                 // 4
                display : "bottom",
                padding : 5,
                spacing : 2,
                font   : { color : 0x000000, family : "Arial", size   : 12 },
                border : { size : 1, color : 0x999999 }
            }
        }
    ],
    new Ext.Window({
        width  : 400,
        height : 250,
        layout : 'fit',
        items   : pieChart
    }).show();

    {1} Setting the data field
    {2} Setting the category field
    {3} Applying custom colors to the series
    {4} Add the extra styles to show the chart legend
};
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

To create the PieChart configuration, we first set the `xtype` property to 'piechart', which Ext JS uses to configure an instance of `Ext.chart.PieChart`. The next two properties, `dataField{1}` and `categoryField{2}` are absolutely crucial to the PieChart's operation. These are similar to the `xField` and `yField` properties of the basic Cartesian style chart we created earlier.

What makes these so important is the fact that you cannot configure custom series for the PieChart. Therefore, the only place to tell the chart what record fields to map for the category and data is in the root configuration object for the PieChart.

You're probably wondering what the `series` property is doing in the PieChart configuration object if you cannot configure series. It's there because the only means for which you can customize the colors used in the series is by means of a nested colors array`{3}` and is completely optional. If you configure a PieChart without custom colors, the framework will use its own custom palette of colors for the series.

In the last configuration area, we re-use the `extraStyle` configuration object we created earlier to display a legend on the bottom of the chart. The PieChart does not display labels on the series, which means that a legend is important for the users to decipher the data that the PieChart presents to them.

Finally, we place the PieChart inside of an `Ext.Window` to be rendered on a resizable canvas.

Here is what our PieChart looks like painted on screen.

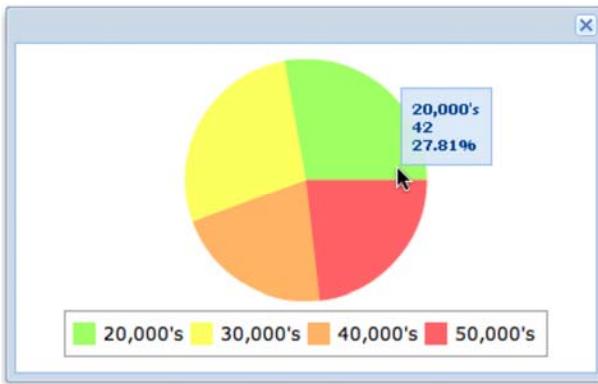


Figure 10.19 The PieChart rendered on Screen with custom colors and a generic tooltip.

While looking at the PieChart, we can see that the legend displays with the custom palette, ranging from green to red. If we hover the mouse cursor over the different series, we see the out of the box tip display, revealing the data that is used to draw the chart. But, just as in the other charts, sometimes it really does not provide much context to the data.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Of course, the only way to remedy this situation is to apply a custom tipRenderer, which is what we're going to embark on next.

### 10.6.1 Mixing line with Columns

When developing a custom tipRenderer for the PieChart, you are only presented with the data that the chart is used to draw the chart. That is, when a configured tipRenderer is called, the information for the category and data fields is all that is provided. This means that unless the percentage is provided by the web service for each record, you'll have to code the calculation manually.

Here is a quick recipe on how to do just that.

#### **Listing 10.9 Creating the PieChart**

```
var tipRenderer = function(chart, record, index, series) {
    var seriesData = record.data; // 1
    var total = 0;

    Ext.each(series.data, function(obj) { // 2
        total += parseInt(obj.total);
    });

    var slicePct = (seriesData.total/total) * 100; // 3
    slicePct = (' ' + slicePct.toFixed(2) + '%');

    var rangeMsg = 'Salary Range : ' + seriesData.range;
    var empMsg = 'Num Emp. : ' + seriesData.total + slicePct;

    return rangeMsg + '\n' + empMsg; // 4
};

{1} Get a reference to series data for the tooltip
{2} Calculate the sum of all of the data
{3} Calculate the percentage for the series
{4} Return the custom tip text
```

In the above tipRenderer method, we first create a reference of the series data by using the passed record's data object{1}. Next, we use Ext.each to loop through the series.data array to get the sum of all of the records{2} and calculate the percentage for the tip{3}. Lastly, we assemble the message for the tooltip{4}.

Next, we have to configure the PieChart to use the custom tipRenderer:

```
tipRenderer : tipRenderer,
```

With the custom tipRenderer now in place, lets take a look at what the new tooltip looks like on the chart.

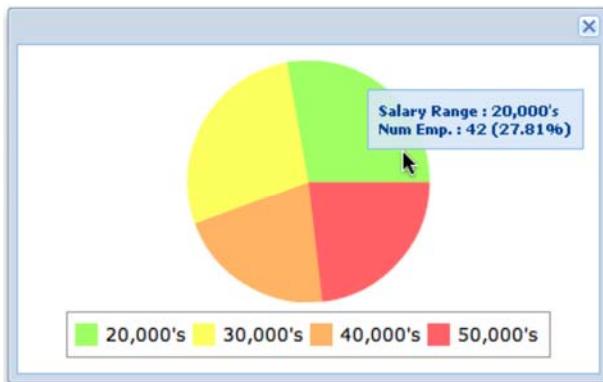


Figure 10.20 The PieChart with a custom tool tip.

As we can see, by taking some time to add a custom tipRenderer to a pieChart, we can add context to the information presented inside of the tooltip making it a lot more useful. One of the things to be careful when developing custom tipRenderers is not to display too much information in that little tip balloon. Bombarding the user with that much information could reduce the usability of the tooltip.

## 10.7 Summary

In this chapter, we took an in-depth look at the various charts provided by the framework, where we learned what it took to not only configure the different charts. Along the way, we learned the basics of applying basic styling and a custom tooltip.

We began by discussing the different charts available to us in the framework. We learned that the CartesianChart is the base class for the LineChart, ColumnChart and BarChart widgets. We also learned how the PieChart is different from the others, and discussed the details while examining the chart package class hierarchy.

Using an exploratory approach, we implemented each of the charts available to us in the framework. While doing so, we saw what it took to create a simple LineChart, customize it and then modify it to create a ColumnChart. By flipping the X and Y-axis and modifying the configuration slightly, we were able to convert the ColumnChart to a BarChart. We also learned how to create hybrid charts by adding a LineSeries to a ColumnChart and BarChart.

Lastly, we learned how to create PieCharts, further solidifying their difference from the other CartesianChart-based charts. Along the way, we learned how to add context and percentage data to the tooltip, making it more useful.

In the next chapter, we're going to learn how to leverage one of the more powerful UI widgets, the TreePanel, to display hierarchical data.



# 11

## *Taking root with Trees*

I can recall the first time I was tasked to create an application with what was known then as a “TreeView”. I had to allow for the navigation directory in a file system, which required that I allowed users to modify the names of the files easily. I was lucky that I had Ext JS in my toolbox to aid me in accomplishing my task. Using the framework not only made things easier, it sped up the development time of this task dramatically.

In this chapter, you’re going to learn about the Ext JS TreePanel, which is used to display hierarchical data, much like a typical filesystem. You’re going to learn how to setup both static and dynamic implementations of this widget. After getting comfortable with this component, we’ll work to setup CRUD operations by use of a dynamically updating context Menu and Ajax requests to send data. This is going to be a fun chapter.

### **11.1 What is a Tree(Panel)?**

While the term “tree” is generally used to describe a woody plant, in the UI world, it is meant to describe a widget or control that displays hierarchical data which generally begins at some central point, which is known as a *root*. And like the tree plant, trees in UIs have branches, which means that they contain other branches or leafs. Unlike the tree plant, computer trees only have one root.

In the computer world, this paradigm is ubiquitous and lives under our noses without much thought. Ever browse your computer’s hard disk? The directory structure is a tree structure. It has a root (any drive letter in Windows), branches (directories) and leafs (files). Trees are used in application UIs as well and are known by a few other monikers.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

In other UI libraries, other names for this type of widget include "TreeView", "Tree UI" or simply "Tree", while in Ext JS, it's known as the TreePanel. The reason it's called "TreePanel" is because it is a direct descendant of the Panel class. And much like the GridPanel, is not used to contain any children except what it was designed for. The reason TreePanel extends from Panel is simple - convenience. This gives us the flexibility to leverage all of the Panel's UI goodness, which include the top and bottom ToolBars and footer button bar.

Like the EditorGrid, TreePanels can be configured to allow the edit data, but does not have a DataWriter equivalent, which means that we must code our own Ajax requests for CRUD actions. We'll explore how to make a TreePanel editable and how to code for CRUD actions.

Unlike the GridPanel, however, the number of the supporting classes is actually quite small, which makes the configuration a TreePanel is *really* simple in contrast, as we'll see a little later on. On the flip side, many developers find that the server side code and related SQL to support them is much more challenging due to the relational nature of the data.

Lastly, the Ext.data classes do not apply to TreePanels, thus Proxys and Readers are out of the picture. This means that the data for the TreePanel must either come from memory or remotely, but is limited to the same domain. Before we get down to building our first TreePanel, we will discuss how a TreePanel works.

## 11.2 Looking under the roots

TreePanels work by loading data via a TreeLoader class, which either reads JSON-formatted data from memory, remotely from the web server or can be a mixture of both. Each Object in the JSON stream is converted to an instance of treeTreeNode, which is a descendant of the Ext.data.Node class. This data.Node class is the core of all of the TreePanel data logic and includes many of the utilities such as cascade, bubble and appendChild.

In order for the TreePanel to display the nodes visually, the TreeNode class uses the treeTreeNodeUI class. The root node gets some special attention, as it's the source of the entire structure and has its own RootTreeNodeUI class. If you want to customize the look and feel of nodes, you would extend this class.

Cool, we have a high level understanding of what a TreePanel is and how it works. We can start constructing our first TreePanel, which will load data from memory.

## 11.3 Planting our first TreePanel

As I mentioned before, coding a TreePanel, relative to the GridPanel, is pretty simple. We'll start out by constructing the TreePanel, which loads its data from memory. This will give us more insight into what we learned just some time ago.

### **Listing 11.1 Building a static TreePanel**

```
var rootNode = { // 1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

text      : 'Root Node',
expanded  : true,
children  : [
  {
    text : 'Child 1',
    leaf : true
  },
  {
    text : 'Child 2',
    leaf : true
  },
  {
    text      : 'Child 3',
    children : [
      {
        text      : 'Grand Child 1',
        children : [
          {
            text : 'Grand... you get the point',
            leaf : true
          }
        ]
      }
    ]
  }
]
}

var tree = { // 4
  xtype     : 'treepanel',
  id       : 'treepanel',
  autoScroll: true,
  root      : rootNode
}

new Ext.Window({
  height   : 200,
  width    : 200,
  layout   : 'fit',
  border   : false,
  title    : 'Our first tree',
  items    : tree
}).show();
{1} The JSON data for our tree nodes
{2} Including child nodes for this branch
{3} Specifying that a node is a leaf
{4} Configuring our treePanel

```

Yikes! Most of the code in listing 11.1 is the data to support the TreePanel. In walking through the `rootNode{1}` JSON, we see that the root node (object) has a `text` attribute. This is important because it is the `text` property is what is used by the `TreeNodeUI` to actually display the node's label. When coding the server-side code to support this widget, be sure to keep this property in mind. If you don't set it, the nodes may appear in the TreePanel, but will have no label.

We also see an `expanded` property, which is set to `true`. This ensures that, when rendered, the node is expanded immediately, thus displaying its contents. I set this here so you can see the root's `childNodes` immediately upon the rendering of the TreePanel.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

This parameter is optional. Leave it out or set it to false to have the node render initially collapsed.

A children property is set on the root node, which is an array of objects. When a node has a children array, the objects in that array will be converted to tree.TreeNode s and populated in the parent node's childNodes array. A similar paradigm can be found in the Container hierarchy, where a Container has children in its items MixedCollection.

If we walk through the rootNode's children, we see that the first and second child have no children property, but have a leaf{3} property, which is set to true. Setting a node's leaf property to true will ensure that this node will never contain other child nodes, thus is a leaf and not a branch. In this case, 'Child 1' and 'Child 2' are leaf nodes, while 'Child 3' is a branch because it does not have a leaf property set to true.

The 'Child 3' node contains one child node, which is a leaf because? Yes. It is a leaf because its leaf property is set to true. Wow you learn quickly. This node has a single child, which has a single child.

After configuring the supporting data, we move on to configure the TreePanel{4} using an XType configuration object. This is where we see the simplistic nature of the configuration of this widget. All of these properties should make sense to you but the root, which is what we use to configure the root node. In this case, the top most level object of the rootNode JSON will be treated as the TreePanel's root.

You can change your TreeNode icons by adding either an icon or iconCls properties to the node's configuration object, where icon specifies a direct location for an image and iconCls is the name of a CSS class for an icon style. However, iconCls property for the TreeNode works just like the Panel's iconCls configuration object and is the preferred method for changing the icon.

The last thing we do in this listing is create an instance of Ext.Window to display our TreePanel. Here is what our rendered TreePanel looks like.

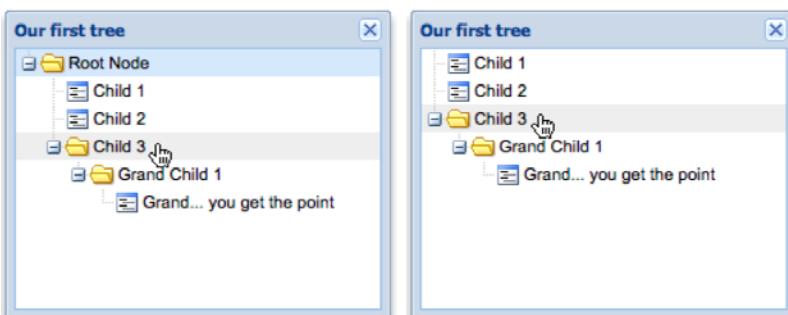


Figure 11.1 Our first (expanded) TreePanel with the root node visible (left) and the root node hidden (right).

After rendering our TreePanel, we can see the tree nodes displayed as we laid it out in the JSON. You can expand 'Child 3' and its child node to display the rest of the hierarchy. It's easy to exercise the selection model by clicking on a node. If you wanted to hide the root node, simply set `rootVisible` in the TreePanel configuration object to false, as depicted in Figure 11.1 (left).

And there you have it, a static TreePanel in action. Simple stuff, huh? Now that we have this out of the way, we're going to move on to creating a remote TreePanel.

## 11.4 Dynamic TreePanels Grow

Because our previous TreePanel is static, there was no need to directly create a TreeLoader. This changes with a remote-loading TreePanel. We're going to develop a TreePanel that will use the same data that we used for our GridPanels, where we displayed people. It just so happens that those people are employees for "My Company" and belong each to different departments. Here, we are going to configure the TreePanel to utilize the server-side component to list employees by department.

### **Listing 11.2 Building a static TreePanel**

```
var tree = {
    xtype      : 'treepanel',
    autoScroll : true,
    loader     : new Ext.tree.TreeLoader({
        url : 'getCompany.php'
    }),
    root       : {
        text      : 'My Company',
        id       : 'myCompany',
        expanded : true
    }
}

new Ext.Window({
    height   : 300,
    width    : 300,
    layout   : 'fit',
    border   : false,
    title    : 'Our first remote tree',
    items    : tree
}).show();

{1} Instantiating a TreeLoader for remote data calls
{2} Configuring a root node inline
{3} Setting the ID of the root node
```

As we can see in Listing 11.2, we configure a TreeLoader`{1}`, which has a configuration object passed with a `url` property set to 'getCompany.php'. When configuring your TreePanel, replace this PHP file with your controller of choice. Before you start coding your controller, however, please allow me to finish walking through the request and response cycle, which follows shortly after we look at the rendered version of this TreePanel implementation.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

The next thing we do when configuring this TreePanel is configure the `root{2}` inline. It is extremely important to notice that we added an `id{3}` property to this node. As we'll see, this property is going to be used to request the child data from the server. Also notice that we set `expanded` to `true`. This is going to ensure that the root node expands and *loads* its children as soon as it's rendered.

Lastly, we configure a bigger instance of `Ext.Window` to contain our TreePanel. Configuring the window a bit bigger for this demonstration will both increase the TreePanel's viewing space and eliminate horizontal scrolling due to long names. Here is what the rendered TreePanel looks like.

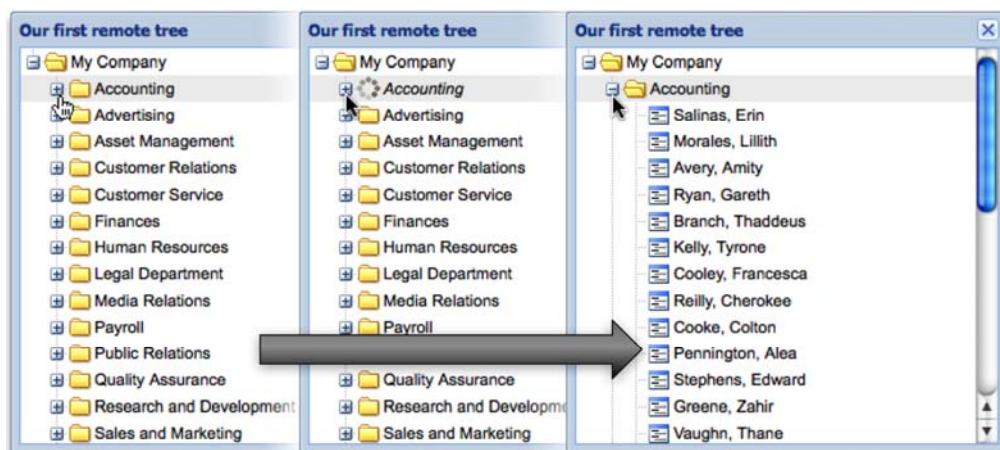
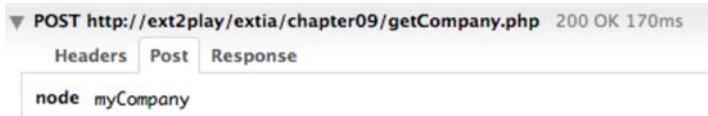


Figure 11.2 Our remote TreePanel displaying its ability to load data remotely.

After rendering the TreePanel, we see the root node (`My Company`) load immediately, as shown in Figure 11.2 (left), displaying all of the departments in "My Company". To view the employees in a particular department, click on the expand icon (+) or double click the label and you'll see the remote loading indicator appear in place of the folder icon as seen in the center of Figure 11.2. After the employee nodes are loaded successfully, they will appear below the department node.

We went through this pretty fast. Lets rewind time a bit and take a look at the requests being fired. We'll discuss what the server side controller is doing to support this implementation of the TreePanel.

We'll start by the load request fired off by the automatic expansion of the root node. Remember that we set the root's `expanded` property to `true` and that this expands a node when its rendered, thus either rendering the children if they are in memory or firing a load request.



### 11.3 The post parameter of the initial node request.

As we see in Figure 11.3, the first request to the `getCompany.php` controller was made with a single parameter, `node`, which has a value of `myCompany`. Can you remember where we set that value and which property it was set as? If you said “the `id` property root node”, you’re correct! When an asynchronously loading node is being expanded for the first time, the loader will use its `id` property to pass to the controller for the child data.

The controller will accept this parameter and query the database for all nodes associated with that `id` and return a list of objects illustrated below.

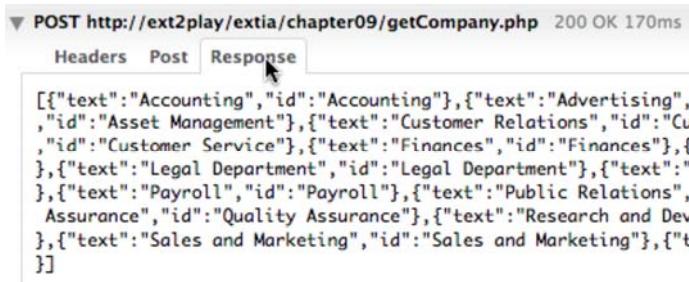


Figure 11.4 the results of the initial request to the `getCompany.php` controller.

In figure 11.4, we see an array of objects that define a list of departments. Each object has both a `text` and `id` property. How does the `text` property apply to the NodeUI? Correct! The `text` applies to the label of the NodeUI. Notice that the departments lack the `leaf` and `children` properties. Are these leaf or branch nodes? Correct. They are branch nodes. Because neither property is defined, they are treated as branch nodes. This means that when they are initially expanded, the `TreeLoader` will invoke an Ajax request, passing the department’s `ID` as the `node` parameter. The controller will accept the `node` parameter and return a list of employees for that department.

Using what we just learned, we can safely predict that when we expand the Accounting department that a request to the `getCompany.php` controller will be made with a single parameter, `node`, passed with a value of ‘`Accounting`’. Lets take a quick look at the results from the controller request.

```
[{"id": "19", "text": "Salinas, Erin", "leaf": true}, {"id": "28", "text": "Avery, Amity", "leaf": true}, {"id": "60", "text": "Ryc", "leaf": true}, {"id": "87", "text": "Kelly, Tyrc", "leaf": true}, {"id": "126", "text": "Reilly, Cherokee", "leaf": true}, {"id": "152", "text": "Pennington, Alea", "leaf": true}, {"id": "226", "text": "Greene, Zahir", "leaf": true}, {"id": "259", "text": "Bird, Kamal", "leaf": true}, {"id": true}]
```

Figure 11.5 the results from the Accounting department node request.

As we look at the JSON results, we see that a list of objects is returned, each with `id`, `text` and `leaf` properties. Remember, that it is because the `leaf` property is set, that the nodes appear as non-expanding leaf nodes.

Configuring a TreePanel for loading is just a small part of the job if you're tasked to build a UI that offers CRUD functionality for this type of widget. Next, we'll look at how to construct a TreePanel for these types of interactions.

## 11.5 CRUD on a TreePanel

To configure CRUD UI functionality, we're going to need to add much more code to the mix. After all, the TreePanel doesn't support these features natively. Here's what we're going to do.

To enable CRUD actions, we're going to modify our TreePanel by adding a `contextmenu` listener to it, which will call a method to select the node that was right-clicked and create an instance of `Ext.menu.Menu` to be displayed at the mouse cursor's X and Y coordinates. This will be very similar to how we coded the `EditorGridPanel`'s context menu handler in the last chapter.

We'll create three menu items, for Add, Edit and Delete. Because we can only add employees to a department, we're going to dynamically change text for, enable and disable the various menu items based on the type of node that was clicked; root, branch or leaf.

Each of the handlers will perform an Ajax requests to mock controllers for each CRUD action. Again, much of this will function like the CRUD for the `EditorGridPanel`, but adapted to the TreePanel to deal with nodes instead of rows.

Get ready. This will be the most complicated code yet. We'll start by creating the context menu handler and context menu factory method.

### 11.5.1 Adding Context Menus to our TreePanel

To add a context menu to the TreePanel, we're going to have to register a listener for the `contextmenu` event. This is super simple. Add a `listeners` configuration option to the TreePanel as such:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
listeners : {
    contextmenu : onContextMenu
}
```

Adding the previous code will ensure that the `onContextMenu` handler will be called when the `contextmenu` (or right-click) event is fired.

Cool. Our `TreePanel` is now setup to call the `onContextMenu` handler. Before we code it, we should generate a factory method to generate an instance of `Ext.menu.Menu` for us. This will help simplify `onContextMenu` quite a bit. You'll see what I mean after we are done with the factory method.

### **Listing 11.3 Configuring a context menu factory method**

```
var onConfirmDelete = Ext.emptyFn;
var onDelete      = Ext.emptyFn;
var onCompleteEdit = Ext.emptyFn;
var onEdit        = Ext.emptyFn;
var onCompleteAdd = Ext.emptyFn;
var onAddNode     = Ext.emptyFn;

var buildContextMenu = function() {
    return new Ext.menu.Menu({
        items: [
            {
                itemId : 'add',
                handler: onAdd
            },
            {
                itemId : 'edit',
                handler: onEdit,
                scope  : onEdit
            },
            {
                itemId : 'delete',
                handler: onDelete
            }
        ]
    });
}
```

In Listing 11.3, we first setup a bunch of placeholder methods that point to `Ext.emptyFn`, which is the same thing as instantiating a new instance of `Function`, but is easier on the eyes. We're adding them now so that when we circle back and fill in these methods in, you'll know exactly where to place them.

Next, we generate the `buildContextMenu` factory method, which returns an instance of `Ext.menu.Menu` and will be used by the `onContextMenu` handler that we're going to generate next. If you've never seen or heard of a factory method, from a high level, it is a method that constructs (hence the name *factory*) something and returns what it constructed. That's all there is to it.

Notice that each of the Menu items has no `text` property, but each have `itemId` specified. This is because the `onContextMenu` will actually dynamically set the `text` for each

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

menu item to provide feedback to the user that something may or may not be allowed. It will use the `itemId` property to locate a specific item in the Menu's items mixed collection.

The `itemId` configuration property is very similar to the `id` property of Components except it is local to a child Component's Container. This means that unlike the Component's `id` property, `itemId` is not registered with `ComponentMgr`, thus only the parent Component has the ability to look into its items `MixedCollection` to find a child component with a specific `itemId`.

Each `MenuItem` currently has a hard-coded handler to `Ext.emptyFn` as a placeholder so we can see our menu display in the UI without having to code the real handler. We'll go on to create each handler after we develop and review the `onContextMenu` handler, which is next. This listing is going to be large, so please don't run away.

#### **Listing 11.4 Configuring a context menu factory method**

```
var onContextMenu = function(node, evtObj) {
    node.select();
    evtObj.stopEvent();

    if (!this.ctxMenu) {
        this.ctxMenu = buildCtxMenu(); // 1
    }

    var ctxMenu      = this.ctxMenu;
    var addItem     = ctxMenu.getComponent('add');
    var editItem    = ctxMenu.getComponent('edit');
    var deleteItem  = ctxMenu.getComponent('delete');

    if (node.id == 'myCompany') { // 2
        addItem.setText('Add Department');
        editItem.setText(' Nope, not changing the name');
        deleteItem.setText('Can\'t delete a company, silly');

        addItem.enable();
        deleteItem.disable();
        editItem.disable();
    }
    else if (!node.leaf) {
        addItem.setText('Add Employee');
        deleteItem.setText('Delete Department');
        editItem.setText('Edit Department');

        addItem.enable();
        editItem.enable();
        deleteItem.enable();
    }
    else {
        addItem.setText('Can\'t Add Employee');
        editItem.setText('Edit Employee');
        deleteItem.setText('Delete Employee');

        addItem.disable();
        editItem.enable();
        deleteItem.enable();
    }

    ctxMenu.showAt(evtObj.getXY() );
}
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
{1} Using our context menu factory method  
{2} Configuring the menu each type of node
```

In listing 11.4 we construct our `onContextMenu` handler, which does quite a bit to enable our context menu to be quite dynamic. The first task that this handler accomplishes is selecting the node in the UI by firing the node's `select` method. We are selecting the node because we're going to need to query the `TreePanel` for the selected node further on, after Ajax calls are made.

It can do this because every time the `TreePanel`'s `contextmenu` event fires it passes two arguments: the node that the event occurred on and the instance of `EventObject` that was generated. If bells are ringing in your ears, it's probably because that this is very similar to the `GridPanel`'s `contextmenu` event, where the row that the event occurred and an instance of `EventObject` is passed to the handler.

Next we stop the browser's default context menu from showing up by calling `evtObj.stopEvent`. You'll see this pattern repeating anywhere you need to need to display your own context menu in placement of the browser's.

The handler then moves on to construct the context menu by calling the `buildContextMenu` factory method we created just a bit ago. It stores the reference locally as `this.ctxMenu` so it does not have to reconstruct a menu for each subsequent handler calls.

Next, we create local reference to the context menu, `ctxMenu` and each of the menu items. We're doing this for ease of readability further on, where we actually manage the menu items.

After we create the local references, we move on to an `if{2}` control block, where we detect the type of node and modify the menu items accordingly. This is the bulk of the code for this handler. Here is how this logic breaks down.

If the node that was right-clicked is the root (`node.id == 'myCompany'`), we configure the menu items to allow the addition of departments, but disallow the deletion and editing of the company text. We also disable those menu items so they cannot be clicked. After all, we don't want anyone to destroy an entire company by a single mouse click, do we?

Next, we detect to see if the node is not a leaf (department). We then move on to modify the text to allow the addition of employees and deletion of the entire department. Remember, the company needs to be able to downsize by removing entire department if need be. We also enable all menu items.

The code will encounter the else block if the node that was right-clicked is a leaf item. In this case, the text for the add item is modified and disabled to reflect the inability to add an employee to an employee, which would be just weird. The edit and delete menu item texts are modified and enabled.

Lastly, we show the context menu at the coordinates of the mouse by calling the `EventObject's getXY` method. Here is what the menu looks like customized for each node.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>



Figure 11.6 Displaying our dynamic context menu for the company (left), department (center) and employee (right) nodes.

As illustrated in Figure 11.6, our context menu displays and changes for each type of node that was right-clicked, which demonstrates how you can use the same menu to perform similar tasks with some modifications. If you wanted to not show or hide the menu items instead of enabling and disabling, simply swap the MenuItem enable calls for show and disable for hide. We can now begin to wire up handlers for our context menus. We'll start with the easiest, edit.

### 11.5.2 Wiring up the Edit logic

You probably noticed that clicking on a MenuItem resulted in nothing more than the menu disappearing. This is because we have our context menu set, but no real handlers for them to call. We'll start by creating the edit handler, which is by far the easiest to code.

This is where we'll instantiate an instance of tree.TreeEditor to allow for inline edits of node names. In order to get the TreeEditor to invoke an Ajax request to submit the modified node name, we're going to have to setup a listener for the TreeEditor's published complete event, which indicates that an edit has completed and the actual value of the node has been changed. We'll code for the complete handler first, then move on to create the onEdit handler, which we'll attach to the edit MenuItem.

Here is where we'll start filling in some of those placeholder methods.

#### **Listing 11.5 Configuring a context menu factory method**

```
var onCompleteEdit = function(treeEditor, newValue, oldValue) { // 1
    var treePanel = Ext.getCmp('treepanel')
    treePanel.el.mask('Saving...', 'x-mask-loading');

    var editNode = treeEditor.editNode;
    var editNodeId = editNode.attributes.id;
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

Ext.Ajax.request({
    url      : 'editNode.php',
    params   : {
        id       : editNodeId,
        newName : newValue
    },
    success  : function (response, opts) {
        treePanel.el.unmask();
        var responseJson = Ext.decode(response.responseText);
        if (responseJson.success !== true) { // 3
            editNode.setText(oldValue);
            Ext.Msg.alert('An error occurred with the server.');
        }
    },
    failure   : function (response, opts) { // 4
        treePanel.el.unmask();
        editNode.setText(oldValue);
        Ext.Msg.alert('An error occurred with the server.');
    }
});

var onEdit = function() { // 5
    var treePanel = Ext.getCmp('treepanel');
    var selectedNode = treePanel.getSelectionModel().getSelectedNode();

    if (!this.treeEditor) { // 6
        this.treeEditor = new Ext.tree.TreeEditor(treePanel, {}, {
            cancelOnEsc : true,
            completeOnEnter : true,
            selectOnFocus : true,
            allowBlank : false,
            listeners : {
                complete : onCompleteEdit
            }
        });
    }

    this.treeEditor.editNode = selectedNode;
    this.treeEditor.startEdit(selectedNode.ui.textNode);
}
}

{1} Create our TreeEditor complete event handler
{2} Invoke an Ajax request
{3} Revert the change if the server returns success : false
{4} If the request fails, revert the change
{5} Configure our edit MenuItem handler
{6} Create a tree editor if it does not exist.

```

In Listing 11.3, we create two methods that take care of the edit functionality. The first is the handler that will be called when the edit event of the `TreeEditor` is fired and is responsible for firing off an Ajax request to the server to save the modification. The second is the handler that is fired when the edit `MenuItem` is clicked. Here is how they work.

When the `TreeEditor` fires the `complete` event, it passes three parameters to the listeners; a reference to the `TreeEditor` that fired the event, and the new value and old value. Our `onCompleteEdit{1}` event handler uses all three parameters to do its job. This method first sets a local reference to the `TreePanel` that is being edited and masks its element. It then sets two more local references to the node being edited and its ID.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Next, it fires off an Ajax request<sup>{2}</sup> to save the data. Notice that in the parameters configuration object of the Ajax request, it's passing the id of the node and its new name. This would be important if you had to modify a value in a database where an ID identifies nodes.

If there were other application-specific values for that node that you were looking to send as parameters to the server, you can find them in the attributes property of the node. Anything properties that are part of the node to create the tree are tucked away for use in the attributes property. For instance, employee nodes could have employee-specific personal attributes, such as DOB or sex on each node. Obviously, this depends on the application you're developing for. Also, if you wanted to send an attribute from the node's parent, you can access it by the node's parentNode property and just tack it on to the params configuration object for the Ajax request.

In the request's success handler, the return JSON from the server is encoded and success<sup>{3}</sup> property tested. If the server returns a false success property the method reverts the value of the node. This is helpful if an invalid value is placed per business rules processed on the server. Likewise, if the request fails for some reason, the failure handler<sup>{4}</sup> is triggered and the node's text value will be reverted. Both handlers will unmask the TreePanel when they are triggered.

The second method in this listing, onEdit<sup>{5}</sup>, will be called when the edit MenuItem is clicked. It first creates a local reference for the TreePanel and the selected node. It then moves on to create an instance of TreeEditor<sup>{6}</sup> and sets a self (this) reference if it does not exist. Notice that it configures the onCompleteEdit listener for the complete event. Lastly, it will set the editNode property as the selectedNode on the TreeEditor and triggers the editing of that node by calling startEdit and passing the selectedNode's ui.textNode reference, which will tell the TreeEditor where to render and position itself.

Triggering the edit in the manner will ensure that the TreeEditor does not scroll the node to the top of the list if the TreePanel is scrollable, thus preventing this undesired effect from happening.

Cool, we have our edit logic all in place, which means we can see it in action! Refresh your page and right click on a node and click on the edit MenuItem. I'll follow along and show you what I see.

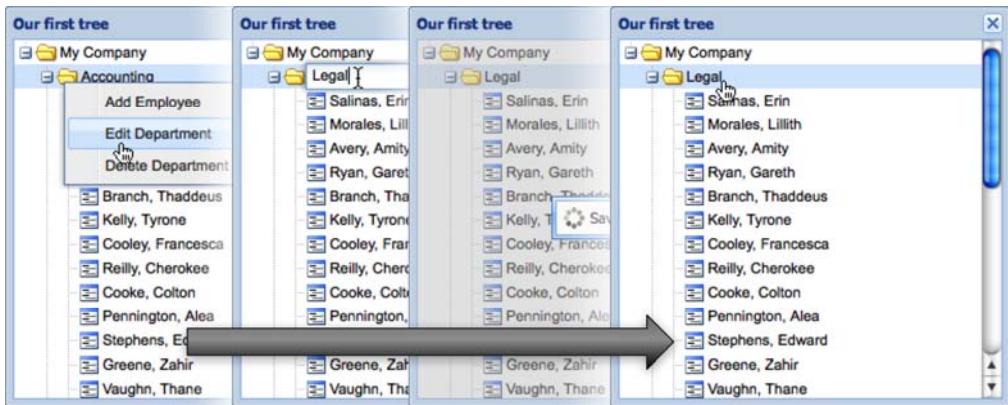


Figure 11.7 The results of editing a node in the TreePanel using the TreeEditor with an Ajax request to save the data.

In Figure 11.7, I modified the Accounting department name to Legal by right clicking on it, which selected the node. I then clicked on the edit MenuItem, which rendered the TreeEditor where the text of the node is. Remember that the TreeEditor knows where to render and position itself because we passed the `textNode` reference of the `TreeNode`'s ui. I then changed the name from "Accounting" to "Legal" and hit Enter on the keyboard. This caused the node value to change and the complete event to fire, thus triggering the `onComplete` method. Because the server accepted the value, the TreePanel's element was unmasked and the new value persisted in the UI. Remember that if the server returned `{ success : false }` or the request failed, the text value of the node would have been reverted.

This wraps up the easiest of the CRUD functionality for our TreePanel. Editing names this widget is quite a common task among web applications. Naturally, how you implement it is up to the needs of the business and requirements. Using the TreeEditor results in a cleaner application flow by saving you from having to use an input dialog box, such as `MessageBox.prompt`.

Next, we're going to ratchet up the level of difficulty by tackling the deletion of nodes. This will be far more complex than what we've done previously in this chapter.

### 11.5.3 Tackling delete

To setup the delete functionality of our TreePanel, we're going to have to create a handler for the delete MenuItem. Naturally, requirements usually dictate that a confirmation dialog is presented to the user, so we're going to have to code for user confirmation. To make things a bit easier, we'll use the out of the box `MessageBox.confirm` dialog. This means that we'll have to construct a callback method for the confirmation dialog. The dialog

callback is what will trigger the Ajax request and ultimately delete the node if the server returns a favorable result.

Now that we have an idea of what we need to do, we can get on with coding the handler methods.

### **Listing 11.6 Adding deletion functionality to our TrePanel**

```

var onConfirmDelete = function(btn) { // 1
    if (btn == 'yes') {
        var treePanel = Ext.getCmp('treepanel');
        treePanel.el.mask('Deleting...', 'x-mask-loading');

        var selNode = treePanel.getSelectionModel().getSelectedNode();

        Ext.Ajax.request({ // 2
            url : 'deleteNode.php',
            params : {
                id : selNode.id
            },
            success : function (response, opts) { // 3
                treePanel.el.unmask();
                var responseJson = Ext.decode(response.responseText);

                if (responseJson.success === true) { // 4
                    selNode.remove();
                } else {
                    Ext.Msg.alert('An error occurred with the server.');
                }
            }
        });
    }
}

var onDelete = function() { // 5
    var treePanel = Ext.getCmp('treepanel');
    var selNode = treePanel.getSelectionModel().getSelectedNode();

    if (selNode) {
        Ext.MessageBox.confirm( // 6
            'Are you sure?',
            'Please confirm the deletion of ' + selNode.attributes.text,
            onConfirmDelete
        )
    }
}

{1} The confirmation message box callback
{2} Invoking an Ajax request to delete the selected node
{3} The Ajax request success handler
{4} Remove the selected node if the server returns true
{5} The delete MenuItem handler
{6} Confirming the deletion of the selected node

```

In Listing 11.6, we create the two methods that take care of the delete operations of our CRUD functionality. The first method, `onConfirmDelete`{1}, is the handler for the confirmation dialog that we'll create a little later on. If the 'yes' button is clicked in the confirmation dialog, it will mask the TreePanel and invoke an Ajax request{2} to delete the selected node. Notice that we're only passing the ID of the node to the server.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

In our fictitious min-application, the server would take value of the node ID and perform a delete operation in the database or file system and return something like `{success:true}`, which fires the request's `success{3}` handler. This removes the TreePanel's element mask and updates the UI by removing the node`{4}` from the TreePanel by using the node's remove method.

To reduce complexity for this listing, I left out the Ajax request failure handler, which would post a failure message to the user. Naturally, when you develop your applications, you'll want to add it. In this case, it should remove the TreePanel's element mask and alert the user that a failure occurred.

The second method we create, `onDelete{5}`, is the handler for the delete MenuItem. When this method is called, it presents a confirmation dialog box by using the out of the box `MessageBox.confirm{6}` method and pass in three arguments; the title, message body and callback. This presents the user with a message and two options to proceed. Either button will trigger the callback, but remember that perform the deletion of the node the 'yes' button has to be clicked.

Please refresh your UI and delete a node. I'll do the same and we'll discuss how things transpire. Below is an illustration of what happened when I refreshed my UI and deleted the Accounting department node.

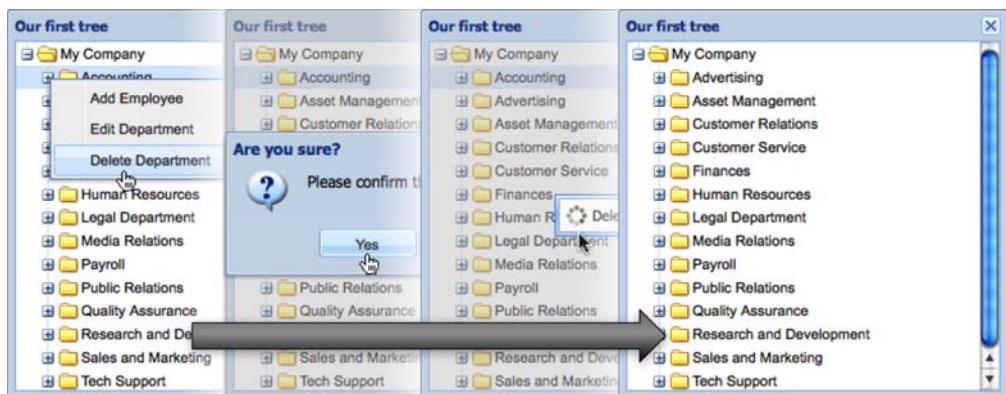


Figure 11.8 Deleting a node with a confirmation box and an Ajax request to the server.

After right clicking on the Accounting department node, our customized context menu appeared. I then clicked on the delete MenuItem, which triggered the `onDelete` handler. This immediately displayed the confirmation dialog box. I clicked on yes, which then caused the TreePanel's element to mask, giving me an indication that a request was being made to delete the node. When the server returned a favorable result, the mask was removed and the Accounting department disappeared.

In a real world application deleting a branch node would generally require the server to recursively gather a list of all of the child nodes and remove them from the database before removing the branch node itself. A clever way to do this could be to setup a trigger in the database to call a stored procedure to delete all associated child nodes when a delete operation is performed on a container node.

Deleting nodes from our TreePanel required a bit more effort because of the typically required confirmation dialog. Adding a node however is equally more difficult because the UI code needs to know what type of node is being added. Is it a branch or leaf node? Next, we'll see how to code for this type of branch in logic and have the UI react accordingly.

#### **11.5.4 Creating nodes for our TreePanel**

To create a node using the TreeEditor, we're going to have to do a lot of work, which makes this the hardest listing in this chapter but the results are going to be pretty sweet.

Because the TreeEditor needs to bind and display on top of a node, we will need to inject a node into the TreePanel and trigger an edit operation on the node. Once the edit of the new temporary node is complete, an Ajax request is fired off to the server with the name of the new node. If the server returns favorably, the ID is set to the node. This works very similarly to the way nodes are added to the EditorGrid we created last chapter.

Here is the code for the create node functionally.

#### **Listing 11.8 Addin create functionality to our TreePanel**

```

var onCompleteAdd = function(treeEditor, newValue, oldValue) { // 1
    var treePanel = Ext.getCmp('treepanel');

    if (newValue.length > 0) { // 2
        Ext.Ajax.request({
            url : 'createNode.php',
            params : {
                newName : newValue
            },
            success : function (response, opts) {
                treePanel.el.unmask();
                var responseJson = Ext.decode(response.responseText);

                if (responseJson.success !== true) {
                    Ext.Msg.alert('An error occurred with the server.');
                    treeEditor.editNode.remove(); // 3
                } else {
                    treeEditor.editNode.setId(responseJson.node.id);
                }
            }
        });
    } else {
        treeEditor.editNode.remove(); // 4
    }
}

var onAddNode = function() { // 5
    var treePanel = Ext.getCmp('treepanel');
    var selNode = treePanel.getSelectionModel().getSelectedNode();
}

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        if (! this.treeEditor) {
            this.treeEditor = new Ext.tree.TreeEditor(treePanel, {}, {
                cancelOnEsc : true,
                completeOnEnter : true,
                selectOnFocus : true,
                allowBlank : false,
                listeners : {
                    complete : onCompleteAdd
                }
            });
        }

        selNode.expand(null, null, function() { // 6
            var newNodeCfg = {
                text : '',
                id : 'tmpNode',
                leaf : (selNode.id != 'myCompany')
            }

            var newNode = selNode.insertBefore(newNodeCfg, selNode.firstChild);
            this.treeEditor.editNode = newNode;
            this.treeEditor.startEdit(newNode.ui.textNode);
        }, this);
    }

    {1} The TreeEditor complete event handler
    {2} Invoke an Ajax request if there is a new node name
    {3} Remove the temporary node if the server returns unfavorably
    {4} The temporary node is removed if the new node name is not present
    {5} The add MenuItem handler
    {6} Expand the selected node, insert a node and trigger an edit
}

```

We are doing quite a bit of work in listing 11.8 to get the create functionality to work smoothly. Just like the code for the edit operations, to reduce complexity for the listing, we omitted the Ajax request failure handler. Here's how all of this stuff works.

The first method in the listing is the TreeEditor complete event handler, `onCompleteAdd{1}`. This handler is arguably more complex than the edit TreeEditor complete event handler for a multitude of reasons. When this method is fired, if a new name has been entered for the node, an Ajax request will be fired`{2}`. If the server responds favorably, it will return the database id of the newly inserted node. The code will set the new node's id via its `setId` method. This is important because it will ensure that subsequent edits to this node will occur against the proper database record. If the server returns unfavorably the temporary node is removed from the TreePanel`{3}`. This also occurs if `onCompleteAdd` is called without a new name`{4}`. Having the UI coded this way ensures that phantom nodes cannot exist in the TreePanel. If you plan on using this pattern to create nodes, be sure to add a failure handler, which should alert the user of a failure and remove the temporary node.

The second method, `onAddNode{5}`, is the handler for our add MenuItem. This method does quite a bit of work as well to provide the user with a temporary node to add to the TreePanel. It does this by instantiating an instance of TreeEditor if it did not exist and binding the `onCompleteAdd` handler to its `complete` event. Next, it expands the

selected node. We expand{6} the selected node because the TreeEditor needs a home, which is going to be the soon-to-be-created node.

Notice that we're passing nulls as the first two parameters. We pass null because we don't need to set them. They are for deep expansion (to expand branches of branches) and to enable animation. We do pass the third and fourth parameters however, a callback method and scope for which it is to be called, which is the `onAddNode` method. The reason we use a callback is because the Ajax load is asynchronous and we need to make sure that the load is complete for a branch node and all of its children are rendered before we can inject a new node.

The callback method creates a node configuration object with an empty string for the text, an id of 'tmpNode' and uses JavaScript shorthand to set the `leaf` value if the selected node's `id` is not 'myCompany'. This ensures that if the department has been selected to have a node added to it, it will set the `leaf` property to `true`, meaning a person, else it's `false`, meaning a department node. It then uses the new node configuration to create a new `TreeNode` using the selected node's `insertBefore` method, passing the `newNode` configuration object and the selected node's `firstChild` reference to specify where to insert the new node. Lastly, an edit operation is triggered on the newly created node, giving the users the ability to add a name to it.

Wow, that was a long one! Lets refresh the UI and see our code in action.



Figure 11.9 Adding a new node to our TreePanel using the TreeEditor.

In Figure 11.9, we can see the how we can use the `TreeEditor` to add nodes to the `TreePanel` in a way that mimics operating system behavior.

When I right clicked on the Accounting department node, the dynamic context menu appeared as expected. I clicked on the `add` `MenuItem`, which triggered our `onAdd` handler. This caused the Accounting node to expand. After the child nodes were loaded, a new node was inserted and an edit operation was immediately triggered on that node using the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

`TreeEditor`. I typed in a new employee name and hit the Enter key on my keyboard. This caused the complete event to be fired by the `TreeEditor`, thus invoking the `onCompleteAdd` handler, which masked the `TreePanel` and performed an XHR. The server returned favorably so the new node stayed in the UI and the database ID of the newly inserted node was applied to it the `TreePanel`. You can also exercise this code to add department nodes to the tree and add employees to that.

This wraps up our coverage for create, which is our last CRUD operations. As you can see, with a little bit of work, we can create a `TreePanel` that allows for full CRUD for nodes complete with inline editing.

## ***Summary***

In this chapter, we covered quite a bit of code when learning about `TreePanels` and how to setup really cool CRUD interaction for nodes.

We started by learning about the `TreePanel` and discussed supporting classes like the `TreeLoader`, `TreeNode` and `TreeNodeUI`. Remember that the `TreePanel` is a direct descendant of `Panel` and can be configured as a child for any `Container`. We constructed a static `TreePanel`, where the nodes were read by memory and analyzed how the JSON should be formatted.

We then moved on to building a dynamic `TreePanel` that loaded data from a remote data source and spent lots of time enabling full CRUD operations. To enable CRUD, we learned how to dynamically modify, enable and disable the reusable context menu. We saw what it took to setup adding and editing with the `TreeEditor` class, which gives the `TreePanel` the ability to edit node names inline.

Until now, we've only scratched the surface with some of the "tools" of the framework, the Toolbar, menus and buttons. In the next chapter, we're going to dive deep and learn more about how they work and how we can better put them to work for our applications.

# 12

## *Menus, Buttons, and Toolbars*

In the chapters leading up to this one, we've configured and used Menus, Buttons and Toolbars, but we really never got to take a moment to really look at these widgets and learn more about them and what else they have to offer. One may ask the question "Why are these three lumped into one chapter?" The answer is simple. Their use cases are related in one way or another. For instance, a Button can be configured to display a menu and a Toolbar can contain that Button.

It is for this reason that we'll use this time to take an in-depth look at the `Ext.menu` and `Item` classes, where we'll learn about what can and how to display items in a menu, which include non-menu items.

Afterwards, we'll focus on the `Button` class and its cousin the `SplitButton`, where we'll learn about things such as how to change the inner button layout and how to attach a menu to it.

Soon after we become more familiar with the button class, we'll create clusters of buttons, known as `ButtonGroups`. We'll even take a moment to create a `ButtonGroup` that emulates one from MS Word 2007's famous ribbon toolbar.

Lastly, we'll tie all of this together in a Toolbar and discuss how to abstract like-functionality with the use of `Ext.Action` to save us time while developing for Buttons and Menus.

### **12.1 Menus at a glance**

For a moment, lets take a step back and ask our selves, "What is a menu?" In the simplest definition, a Menu is something that displays a list of items to choose from. Menus are all around us, including your local coffee shop, where you choose the blend of java to next indulge on. This, by the way, is my favorite kind of menu, as I am a self-proclaimed coffee addict.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

If we think about it, in the computer world, we use menus all the time. The most common are the typical File, Edit and View menus on your application toolbar, where we choose a menu item to Open a document or Copy something to the clipboard. Other ways we see menus are by means of what is commonly known as a Context menu, where a menu is displayed within the *context* of an item that has been either clicked or right clicked. This context menu only displays options only available for the item that the menu was summoned to display. Right clicking on anything on your computer will display a context menu. In Ext JS, we can use and display menus in the same way.

In Ext JS, Menus typically contain one or more `MenuItems` or descendants of `MenuItems`. While in 2.0, it was entirely possible to embed another widget inside of the menu, it was rather laborious to do so. In version 3.0 of ExtJS, accomplishing this task is so much easier because the `Menu` widget extends `Container` and uses the `MenuLayout`. This means that you get the power of the `Container` model managing your `Menu`, affording you the flexibility to manage the `Menu`'s children much like you would any `Container`.

As in the Desktop menu model, Ext JS Menus, can be displayed a number of ways. In the previous `GridPanel` and `TreePanel` chapters, we learned how to configure an instance of `Menu` to display and replace the browser's Context menu. While this is a common use case for this `Menu` widget, it's not the only one.

You have complete flexibility when developing with Menus. For instance, Menus can be attached to a button and displayed upon the click of that button. Or they can be anchored to and shown by any element on demand. How you employ the `Menu` will be based on the requirements of your applications, which means that you can employ Ext JS Menus much like menus are deployed in the desktop model.

OK, I'm itching to start writing some code, aren't you? Well, what are you waiting for?

### 12.1.1 Building a menu

Even though we've built and displayed a context menu in the past, we've only scratched the surface on how to use the `Menu` widget. In the coming sections, we're going to explore the different `Menu` items and how to use them.

We'll begin by creating a plain-Jane menu with a single menu item with a `newDepartment` handler that we'll be reusing.

#### Listing 12.1 Building our base menu

```
var genericHandler = function(menuItem) {                                // 1
    Ext.MessageBox.alert('', 'Your choice is ' + menuItem.text);
}

var newDepartment = {                                                 // 2
    text : 'newDepartment Item',
    handler : genericHandler
}

var menuItems = [                                                 // 3
    newDepartment
];
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var menu = new Ext.menu.Menu({
    items : menuItems,
    listeners : {
        'beforehide' : function() {
            return false;
        }
    }
});

menu.showAt([100,100]);
{1} A newDepartment handler to provide visual feedback
{2} Xtype configuration for a Ext.menu.Menu.Item
{3} A list of menu items for our menu
{4} Create an instance of Ext.menu.Menu to display
{5} Prevent the menu from hiding for ease of testing

```

In Listing 12.1 we do a whole heck of a lot to setup our test bed, which contains a newDepartment handler and a base menu with a single Item. We accomplish this with just a few lines of code. Here's how it works.

The first thing we do is create a genericHandler`{1}` method, which we'll use for just about every menu Item that we create and exercise. When clicked, menu Items call handlers with two arguments, the instance of menu Item that was clicked and the instance of Ext.EventObject that was generated due to the click action of the user. In our newDepartment handler, we only accept the menu Item argument and use its text property to display a MessageBox alert window to provide us with feedback that the handler was called.

Next, we create a configuration object for a menu Item, newDepartment`{2}`, which as a text property and a handler property, which is a reference to our genericHandler. Notice that we're not specifying an xtype property for this configuration object. This is because the defaultType property for the Menu widget is 'menuitem' out of the box.

We then move on to create an array referenced as menuItems`{3}`. We do this so we can abstract the list of menu Items that we're creating for our test Menu. As we continue on, we'll add items to this list.

An instance of Ext.menu.Menu`{4}` is created next, which has two properties. The first is items, which is the reference to the menuItems array of items. Next is a listeners configuration object, which contains a beforehide event listener that we place in here just for testing purposes. If you recall our Component model conversation from eons ago, you may remember that there are certain events that can be vetoed causing the cancelation of a specific behavior. By the listener explicitly returning false, we prevent the menu from hiding. This keeps the menu frozen on screen, which is perfect for testing.

Lastly, we show the menu by calling its showAt method, and pass single argument, which is an array of coordinates, top and left.

Lets see this thing in action.

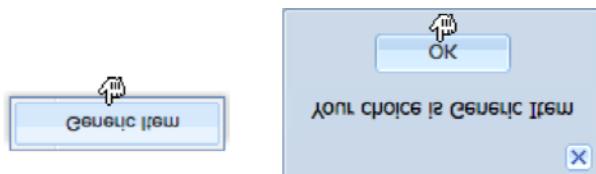


Figure 12.1 Our newDepartment Menu Item (left) and the Item handler (right) in action.

When we render our code, we see the Menu with the single menu Item (Figure 12.1, left) and when we click on it, the MessageBox appears (Figure 12.1, right), providing feedback that the handler was called. Cool! Something is missing however. The menu seems, well... a bit plain to me. What do you think about adding some icon flare to the menu? I like the idea too. Lets do it.

### 12.1.2 Obtaining and using Icons

Early in the book, we discussed how to add icons to widgets by means of the iconCls configuration property. With so much material covered between then and now, I think it's relatively safe to say that a quick refresher is in order.

Most developers who use Ext Js use 16x16 icons in their applications. The preferred method to specify icons is by means of CSS rules. Here is an example.

```
.icon-accept {
    background-image: url(icons/accept.png) !important;
}
```

The icon-accept CSS class above, simply contains a background-image rule with an !important directive. That's it, no magic. Per the Ext JS CSS (unwritten) standard, I typically prefix all CSS icon rules with "icon-". This ensures that there is no cross pollution with any other CSS namespace in the CSS domain. I would highly recommend doing the same.

By now you're probably wondering, where can I get these icons? Well, the most widely used icon set is the famfamfam silk icon set has over 1000 16x16 icons at your disposal and was developed by Mark James. It is licensed under the Creative Commons Attribution 2.5 License. To download it, visit <http://famfamfam.com/lab/icons/silk/>.

If that's not enough, then you can add over 460 icons from developer Damien Guard, known as the Silk Companion and contains many useful derivatives of the famfamfam Silk icons. To download this set, visit <http://damieng.com/creative/icons/silk-companion-1-icons>. The Silk Companion set is Licensed both under the Creative Commons Attribution 2.5 and 3.0 Licenses.

#### NOTE

To read learn more about the Creative Commons Attribution 2.5 and 3.0 Licenses, see <http://creativecommons.org/licenses/by/> and click on the 2.5 or 3.0 links.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

If you have been doing your math, you probably calculated over 1460 icons from which to choose from if you use both sets. This means 1460 CSS rules to write. Let me tell you that there is absolutely no way I would write those CSS rules by hand! How do we solve the problem of having to write all of those rules by hand?

### 12.1.3 Taming the icon madness

I solved this problem easily by downloading both icon sets, merging them into one directory and writing a quick and dirty BASH shell script to compile a CSS file that contains a rule for each icon.

To use this compilation, you will need to download the compiled set at <http://extjsinaction.com/icons/icons.zip>. When you extract the file you'll find two CSS files, icons.css and icons.ie6.css, along with the directory icons, which contains the icon images.

The first CSS file is a compilation for all of the icons that are in their native PNG (Portable Network Graphics) format, which works well for Mozilla-based browsers as well as Internet Explorer 7 and 8. PNGs, however, do not get rendered properly in Internet Explorer 6 without some extensive JavaScript hacking, which I refuse to do.

To solve that problem, I created GIF versions (via shell script, of course) of the PNG icons to make them compatible with IE6 and compiled a list of the gif files named icons.ie6.css

To use any one of these, you must include the required CSS in the head of your browser, as such:

```
<link rel="stylesheet" type="text/css" href="icons/icons.css" />
```

Next, we need to change the newDepartment configuration object to include the iconCls property.

```
var genericItem = {
    text : 'Generic Item',
    handler : genericHandler,
    iconCls : 'icon-accept'
}
```

Refresh the page and see the result of our change.

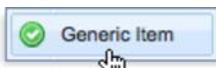


Figure 12.2 The results of adding an Icon to our menu Item via the iconCls configuration property.

As we can see in Figure 12.2, the accept.png icon image file now displays to the left of our menu Item as instructed in the iconCls configuration property. Remember that you can

use any of these icons in any widget that accepts the iconCls property, which includes Buttons, TreeNodes and Panels to just name a few.

We're going to be using this icon set for the rest of this chapter and every chapter moving forward. Please remember to include the icons when you generate new pages to work with.

Now that we have this out of the way, we can move on to the next typical Ext JS menu problem, which is adding submenus to our Menus.

### 12.1.4 Adding a submenu

Developing submenus for menus is something that is another pretty common task for Ext JS developers. They perform two useful functions at the same time. The first is the organization of like menu items and the second is a direct result of the grouping, which is cleanup of the parent menu.

Lets add a submenu to our Menu. You'll want to add this somewhere before the menuItems array we created earlier.

#### **Listing 12.2 Adding a submenu to our base menu**

```
var newDepartment = { // 1
    text : 'New Department',
    iconCls : 'icon-group add',
    menu : [ // 2
        {
            text : 'Management',
            iconCls : 'icon-user suit black',
            handler : genericHandler
        },
        {
            text : 'Accounting',
            iconCls : 'icon-user green',
            handler : genericHandler
        },
        {
            text : 'Sales',
            iconCls : 'icon-user brown',
            handler : genericHandler
        }
    ]
}
{1} The a menu Item configuration object with a submenu
{2} The menu shortcut list of configuration objects
```

In Listing 12.2, we create another newDepartment menu Item, newDepartment{1}, that contains the typical configuration properties such as text and iconCls. What's new is the menu property, which contains a list of menu Item configuration objects, each using our genericHandler. This *shortcut* method of adding a submenu a menu Item is common. In the spirit of Ext JS, there is more than one way to skin this cat. You could elect to set menu as an instance of Ext.menu.Menu or specify a Menu XType configuration object

Below is what the code would look like if we specified an xtype instead of a shortcut array of menu Items.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
var newDepartment = {
    text : 'New Department',
    iconCls : 'icon-group add',
    menu : {
        xtype : 'menu',
        /* menu specific properties here */
        items : [
            /* menu Items here */
        ]
    }
};
```

Choosing the implementation method is up to you. I suggest only creating a separate configuration object for a submenu if you need to set menu-specific properties. Obviously, if you have no menu-specific properties to apply to a submenu, just set the menu property to an array of menu items.

Next, we need to add our menu to the menuItems array.

```
var menuItems = [
    genericMenuItem,
    newDepartment
];
```

To see the fruits of our labor, we're going to have to refresh our page.

The screenshot shows a dropdown menu. The main menu item 'New Department' has a submenu with one item, 'genericMenuItem'. Both items have a tooltip 'bus with quicktime'.

Figure 12.3 The addition of a menu item with a submenu.

Cool, we can now see the “New Department” menu item displayed in our menu. Hovering over the MenuItem reveals the submenu. Clicking on any of menu items in the submenu will result in the MessageBox alert dialog displaying the text of the submenu.

While this is usable, it can be improved somewhat. Next, we'll explore adding a SeparatorItem to the main menu and a TextItem to the submenu, cleaning up the Menu UI a little bit.

### 12.1.5 Adding a Separator and TextItem

Much like the Toolbar separator widget is used to separate Toolbar items with a line, menu Separators are used to physically separate menu items. These are typically used to separate groups or clusters of like menu items from another. Because all of the menu items moving forward will contain submenus, we should separate them with a separator.

Modify the menuItems array as such:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
var menuItems = [
    genericMenuItem,
    '-',
    newDepartment
];
```

Simply add a string with a hyphen and to the menu's items list and it will get interpreted as menu.Separator. Lets take a quick look at the change to our UI.



Figure 12.3 Adding a menu separator (horizontal line) to our menu.

We can see now that there is a horizontal line separating the first and the second menu items. While the separator is generally used to group like items, it is much more powerful than a simple line.

Above, we used the shortcut to generate the separator. Technically, it extends menu.BaseItem, which means that you can make it clickable, and assign a handler to it. While it's *technically* possible to do this, in terms of UI design, I do not advocate using it in this manner.

Cool, we've added the separator, but I think we can stand to dress up the department submenu a bit. For this, we'll add a menu TextItem. To do this, we'll have to create a configuration object to add it to the department submenu.

### Listing 12.3 Adding a text item to the departments menu

```
{
    xtype : 'menutextitem', // 1
    text   : 'Choose One',
    style  : {
        'border'       : '1px solid #999999',
        'margin'       : '0px 0px 1px 0px',
        'display'      : 'block',
        'padding'      : '3px',
        'font-weight'  : 'bold',
        'font-size'    : '12px',
        'text-align'   : 'center'
        'background-color' : '#D6E3F2',
    }
},
```

In listing 12.3, we injected the above configuration object as the first element of the menu array for the newDepartment menu. Remember that the defaultType for menu is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

'menuitem', so we have to override this setting by setting the `xtype` property as '`menutextitem`{1}'. We also set the `style{2}` of this `TextItem` to ensure that it looks presentable. I'll show you in just a bit what the un-styled `TextItem` looks like.

To see the `TextItem` in action, we need to refresh our page.

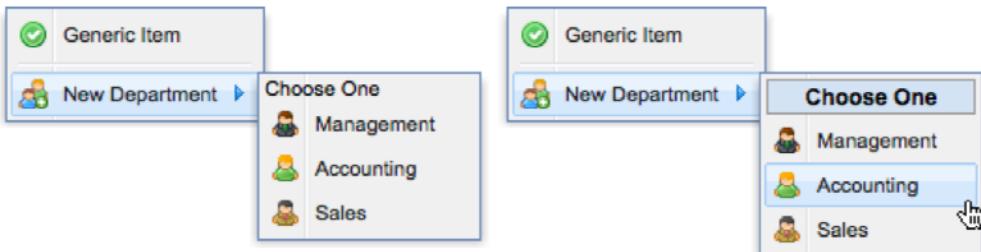


Figure 12.4 Choosing not to style a `TextItem` (left) will result in an incomplete look and feel for your menu. Adding a touch of style will clean up the menu.

In looking at Figure 12.4, it's easy to see the difference between the un-styled and styled `TextItem`. Obviously, the actual Style of the item is left up to you and requirements.

We've just covered how to add both the menu Separator and `TextItem`, both which aid in the visual styles of our Menu. There are two other types of menus that I'd like you to become familiar with, which are the `DateMenu` and `ColorMenu`.

### 12.1.6 Picking a color and choosing a Date

As developers, we're often tasked to allow for the selection of a date or color. Perhaps the start and end date of an appointment or the color of text to be displayed. Ext JS provides to widgets that allow us to give this functionality to our users via Menus. They are the `ColorMenu` and `DateMenu`, which are both descendants of `Ext.menu.Menu`.

This means that you can create a direct instance of either and anchor and display them like their super class. Unlike a traditional menu, they not manage any other child items other than what they are designed for.

Instead of creating instances of these directly, I want to add them to our test-bed menu. But, in order to do that, we're going to have to add menu Items that we can anchor them to. Also, their handlers pass somewhat different arguments than menu Items, so we'll need to create a generic handler specifically for these newly added menus.

Here comes that itch to code again. Lets get started.

#### **Listing 12.4 Using ColorMenu and DateMenu**

```
var colorAndDateHandler = function(picker, choice) { // 1
    Ext.MessageBox.alert('', 'Your choice is ' + choice);
}

var colorMenuItem = { // 2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

text      : 'Choose Color',
iconCls   : 'icon-color swatch',
menu      : {
    xtype   : 'colormenu',                                // 3
    handler : colorAndDateHandler
}
}

var dateMenuItem = {
    text      : 'Choose Date',
    iconCls   : 'icon-calendar',
    menu      : {
        xtype   : 'datemenu',                                // 4
        handler : colorAndDateHandler
    }
}

var menuItems = [
    genericMenuItem,
    '-',
    genericWithSubMenu,
    colorMenuItem,
    dateMenuItem
];
// 5

{1} The color and date handler
{2} The menu Item to display the ColorMenu
{3} The actual ColorMenu configuration Object
{4} The DateMenu configuration Object
{5} Adding the ColorMenu and DateMenu to the menu items array

```

In Listing 12.4, we create a common handler{2} for both the ColorMenu and DateMenu, which is very similar to the genericHandler that we created earlier, but accepts a second argument, choice. Here's how this stuff works.

When an item is chosen from either the ColorMenu or DateMenu, the handler is fired with two arguments. The first is the picker from which the item was selected and the first is the value of the item that was selected, which we aptly named choice. The reason this works the way it does is because the ColorMenu actually uses the ColorPallate widget and relays its select event. Likewise, the DateMenu uses the DatePicker widget in the exact same way. OK, that was a fun, but informative digression. Lets get back on track.

The next task Listing 12.4 accomplishes is create the configuration objects{2} for the menu Items that will be used to display the ColorMenu{3} and DateMenu{4}. Notice that we register the colorAndDateHandler with each both the ColorMenu and DateMenu. Also, we're choosing appropriate icons for these menu Items. Isn't candy is awesome?

Lastly, we append the newly created colorMenuItem and dateMenuItem references to the menuItems array. Lets refresh our UI and see the changes to our Menu.

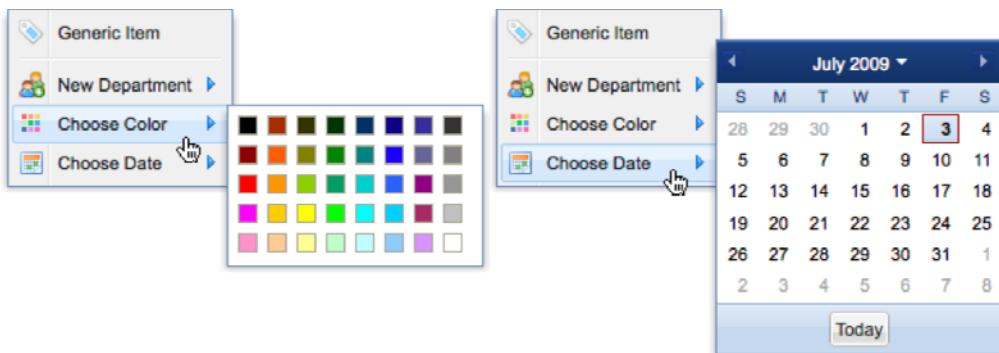


Figure 12.5 The rendered results of the ColorMenu(left) and DateMenu(right).

As illustrated in Figure 12.5, our DateMenu and ColorMenu can be displayed via hovering over the respective menu item. Clicking a color or a date in the respective Menu will trigger the handler, resulting a MessageBox alert displaying the value of the item selected.

The respective ColorPallete for the ColorMenu and DatePicker for the DateMenu can be customize by applying ColorPallete or DatePicker specific properties to their respective containers.

For instance, to show only the colors Red, Green and Blue on the ColorMenu, set the colors property with an array of color hex values as such:

```
colors : ['FF0000', '00FF00', '0000FF']
```

Setting this property will result in the ColorMenu displaying just three boxes, one for each of the colors we described in the array. To learn more about the ColorPallete or DatePicker specific properties to configure, check out their respective API documents.

We've just tackled using the ColorMenu and DateMenu widgets. We have yet one more menu topic to discuss, and that is menu CheckItems, which allows you to configure a menu item that works like a checkbox or radio group.

### 12.1.7 Put that menu Item in check

In forms, check boxes and radio groups allow users to select an item, where the selection is visibly persisted in the UI. In Microsoft Word for Apple OS X, for instance, I can choose how to view a document by clicking in the View menu and selecting a view to activate.

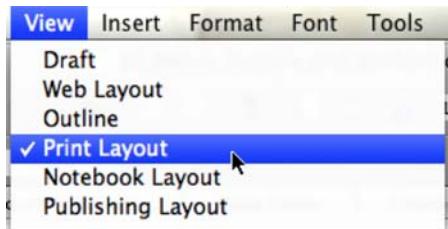


Figure 12.6 A check view menu in OS X, where the user selection is persisted.

In the figure above, I selected the “Print Layout” view and the selection is persisted for each and every subsequent viewing of this menu. The same paradigm could exist in Ext JS application. One such paradigm could exist where a user chooses between a `GridPanel` or `DataView` to display data from a single store.

First, I want to show you exactly how to leverage a single menu `CheckItem` and multiple `CheckItems` in a `checkGroup`. Afterwards, we’ll discuss a possible use for this widget in hopes that it will spark ideas in your mind on how to leverage it.

We’ll create a single check item and add it to our `menuItems` array.

### **Listing 12.5 Adding a single Checkitem**

```
var singleCheckItem = { // 1
    text      : 'Check me',
    checked   : false, // 2
    checkHandler : colorAndDateHandler // 3
}

var menuItems = [
    genericMenuItem,
    '-',
    genericWithSubMenu,
    colorMenuItem,
    dateMenuItem,
    '-',
    singleCheckItem // 4
];
{1} Creating a single CheckItem configuration
{2} Adding the 'checked' boolean value
{3} Specifying a checkHandler
{4} Adding a SeparatorItem and the CheckItem to the menuItems array
```

In Listing 12.5 we create a single `CheckItem`{1} using typical XType configuration. If we run down the list, we only see three properties, none of which are the actual xtype. This is because we can be extremely lazy with this widget and just specify whether the item is checked or not. Ext JS will conveniently handle the instantiation of the `menu.CheckItem` widget for us if the `checked`{2} property is a Boolean and is present.

The next property of importance is `checkHandler`{3}, which is similar to the typical menu item `handler` property, except it passes whether the `CheckItem` has been *checked*.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

or not, which is the second argument that is passed to this method. It's much like the arguments passed to the ColorMenu and DateMenu handlers. It is for this reason, we're reusing the `colorAndDateHandler`.

Lastly, we add a menu Separator and our `singleCheckItem` to the `menuItems` array. Here is what the addition of the `CheckItem` looks like in our menu.

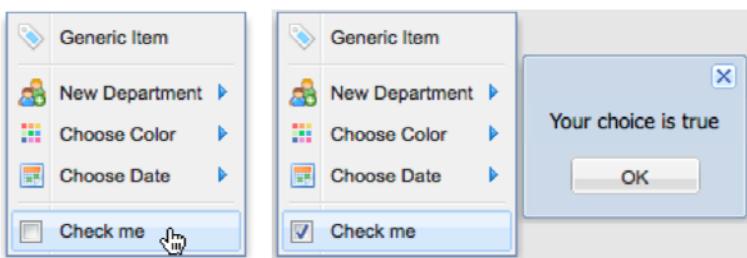


Figure 12.7 Clicking our `CheckItem`(left) makes a check icon appear and triggers the `checkHandler` (right).

Because we used the menu `Separator`, we can see the physical separation of our menu `CheckItem`. Because we set the `checked` value to false, the `CheckItem` appears with an icon that is unchecked. Clicking on it calls the `checkHandler`, which displays the `MessageBox` alert dialog.

#### NOTE

We didn't set the `iconCls` for the `CheckItem`, but you technically can. Setting the `iconCls`, however, means that you're responsible for programmatically changing the `iconCls` upon call of the `checkHandler`, which can be done by using the `Item.setIconClass` method. See the `Menu.Item` API documentation page for details.

The use and configuration of a single `CheckItem` is obviously pretty easy. Remember that they can be grouped, which allows users to select a single `CheckItem` out of that group, much like the Microsoft Word View menu we saw above. Next, we'll construct such a group.

#### **12.1.8 Select only one item at a time**

The configuration of a single `CheckItem` is relatively trivial to that of a group of `CheckItems`. This is mainly because you're configuring multiple `CheckItems` at the same time. Lets construct a cluster of grouped `CheckItems`.

#### **Listing 12.6 Adding a single Checkitem**

```
var setFlagColor = function(menuItem, checked) { // 1
    if (checked === true) {
        var color = menuItem.text.toLowerCase();
        var iconCls = 'icon-flag ' + color;
        Ext.getCmp('colorMenu').setIconClass(iconCls);
    }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        }
    }

var colorCheckMenuItem = {
    text      : 'Favorite Flag',                                // 2
    id       : 'colorMenu',
    iconCls  : 'icon-help',
    menu     : {
        defaults : {
            checked      : false,
            group        : 'colorChkGroup',
            checkHandler : setFlagColor
        },
        items   : [
            { text : 'Red' },
            { text : 'Green' },
            { text : 'Blue' }
        ]
    }
}

{1} A common checkHandler for each of the CheckItems
{2} The menu Item that will contain a menu of the CheckItems
{3} A Menu configuration object for a the group of CheckItems
{4} The list of grouped CheckItems

```

In listing 12.5, we create a new common `checkHandler`, `setFlagColor{1}`, for the grouped `CheckItems` we're going to create later. We do this because when `CheckItems` are grouped, they will each call their respective `checkHandler` when any member in the group has been clicked with each call passing the `checked` value of the respective `CheckItem`.

This shared `checkHandler` will test the value of the `checked` parameter. If it is true, it will call `setIconCls` on the parent menu Item of the `CheckItems` menu. This will effectively set the icon with a flag color that matches the selection of the grouped `CheckItem`. This is just a fun way to provide visual feedback that something was selected and flex some of the menu Item's muscle and flexibility.

We create the menu Item that, `colorCheckMenuItem{2}`, which has a question mark icon as default. Notice that the menu configuration is an object{3} instead of an array. This is precisely what we were talking about before, where you can use an Object to configure a menu instead of an array of menu Item configuration objects.

We do this because we want to set defaults on all menu items, which are checked, which is set to `false`, the common group that each `CheckItem` will belong to and the `checkHandler`. This means that the `items` array of the Menu configuration object can be simple objects with just a `text` property for each Item.

Now that we have that complete, please refresh the page and lets see this in action.

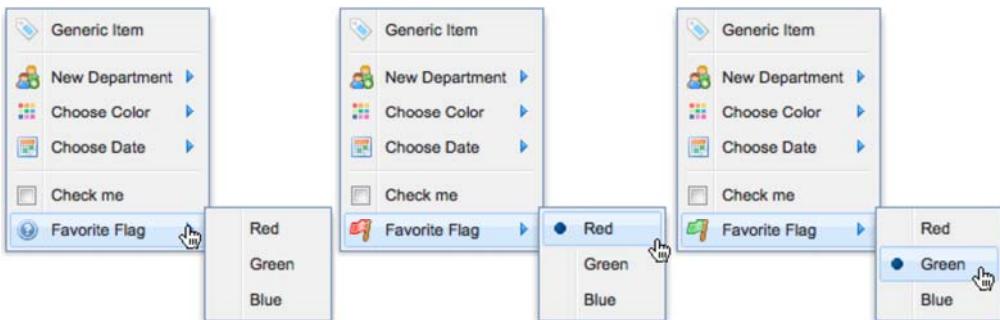


Figure 12.8 A group of CheckItems in action (left), where only one item can be selected (left and right).

Refreshing our page reveals our recently configured “Favorite Flag” menu Item in our test menu with a question icon. Hover over that Item and it will reveal the submenu with the three, grouped color CheckItems. Notice that none of the items bear the check mark. Remember, this is because all items are set to “checked : false” per the defaults configuration object. We could have easily set any one item as “checked : true”.

Click on a CheckItem and it will change the iconCls for the parent menu Item. Revealing the menu that contains the checked items again will result in a radio icon showing to the left of the selected item. Selecting a different Item will result in the relative changes occurring for that CheckItem.

This wraps up our long, but detailed view into the world of the Ext JS Menu where we learned about and exercised all of the menu Items along with the color and date menus. Next, we’ll dive into the world of Ext Buttons and Split Buttons, where we’ll learn how to configure them and associate menus to them.

## 12.2 Users know how to push your Buttons

In this section, we’ll learn how to implement Buttons and then move on to learn about the SplitButton, which is a descendant of the Button widget.

Buttons in Ext JS 2.0 were very rigid, where they did not scale very well and the icon position could not be configured. It was also particularly difficult to have a button take part in a layout as a managed child item.

Some of these limitations had to do with how the Button widget itself was constructed, where the Button class in 2.0 extended Component, which means that it is not meant to take place in a layout. Also, the Button Sprite images were not designed to allow the Button to scale much larger than its intended size. In 3.0 of the framework, this changes for the better.

The Buttons in this version of the framework extend BoxComponent, which is where all of the child size management methods are, meaning that buttons are now fully capable of

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

taking part in any layout. Likewise, the button sprites were updated to allow for extremely large buttons. You could have a single button take one half of the screen if you wanted to, though it would probably scare away your users.

As we transition to learning more about and constructing buttons, we'll learn that buttons have a lot in common with menu Items.

### 12.2.1 Building Buttons

We'll start out by creating a simple handler and a Button that will call it. Create a blank page that includes all of the required Ext JS files and the icon CSS file that we worked with just about ago.

From our work with menu Items earlier, a lot of this stuff will be very familiar to you, which means that we can increase our pace a little bit.

#### **Listing 12.7 Building a simple Button with a generic handler.**

```
var btnHandler = function(btn) { // 1
    btn.el.frame();
}

new Ext.Button({
    renderTo : Ext.getBody(),
    text     : 'Plain Button',
    iconCls  : 'icon-control power',
    handler   : btnHandler
});
{1} The generic button handler
{2} A generic button
```

In listing 12.7, we create generic button handler, `btnHandler{1}`, that will simply call the frame effect on the element of the button that calls it. Next, we create an instance of an Ext Button and render it to the document.body by means of the `Ext.getBody` method call. We also set the `text`, `iconCls` and `handler` properties, which are exactly like the menu Item configuration properties.

Load the page with this code and lets see the button in action.



Figure 12.9 Our button in action(left) and the frame effect(right), which is a visual indication that the handler was called.

As in figure 12.9, our Button renders on screen, seemingly just waiting to be pushed. Clicking on it forces the handler to be called, which applies the frame effect to the button's element, giving us visual indication that the handler fired properly. Simple, isn't it? Next, we'll attach a Menu to a Button, which is common practice in Ext JS applications.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

### 12.2.2 Attaching a menu to a Button

You attach a Menu to a Button exactly in the same way you attach a Menu to a menu Item, which is specifying either a list of menu Item configuration objects or a complete Menu configuration object. For this exercise, we're going to do the latter.

#### Listing 12.8 Attaching a menu to a Button

```

var setFlagColor = function(menuItem, checked) { // 1
    if (checked === true) {
        var color = menuItem.text.toLowerCase();
        var iconCls = 'icon-flag' + color;
        Ext.getCmp('flagButton').setIconClass(iconCls);
    }
}

new Ext.Button({
    renderTo : Ext.getBody(),
    text     : 'Favorite flag',
    iconCls  : 'icon-help',
    handler   : btnHandler,
    id       : 'flagButton',
    menu     : {
        defaults : {
            checked      : false,
            group       : 'colorChkGroup',
            checkHandler : setFlagColor
        },
        items : [
            { text : 'Red' },
            { text : 'Green' },
            { text : 'Blue' }
        ]
    }
}); // 2
{1} A modified version of the previous setFlagColor handler
{2} Associating the menu to the Button

```

In listing 12.8, we recreate the scenario for the previous CheckItems exercise, but modify it to fit the Button. Here's how it works.

We first create the checkHandler, setFlagColor`{1}`, which works exactly like the one we constructed earlier, except it changes the iconCls of the Button.

Next, we create another Button that is very similar to the first Button we created and is rendered to the document.body. The main difference between this Button and the other is that we configured`{2}` a Menu with three grouped CheckItems, all registered to call the setFlagColor handler.

Lets see what our code additions and changes bring forth.



Figure 12.10 Adding a menu to a Button, which contains a few menu CheckItems.

At first glance at our newly configured Button, we can see that a little black arrow now appears to the right of the text. This is the visual indication to the user that this button contains a menu to be revealed by a press of the button. To display the Menu, click on the button. What happens?

We see that the menu appears and that the button frame effect is shown as well. Wait. What? How can this be? Well, that's because we didn't remove the handler property, which means when the button is clicked the menu shows *and* the handler is called. In some cases, this is desirable behavior, but in most cases, it's not. If all this button is to do is display a menu, then simply remove the handler. However, if you want a handler to be called separately from the display of the associated Menu, this is where the SplitButton can be useful.

### 12.2.3 Do a Split(Button)

Use of a SplitButton is very similar to that of a Button. The main difference being that the SplitButton has a second handler that it can call when clicked, which is known as the arrowHandler.

To exercise the SplitButton, we're going to slightly modify Listing 12.8.

#### **Listing 12.9 Building a simple Button with a generic handler.**

```
new Ext.SplitButton({  
    renderTo : Ext.getBody(),  
    text : 'Favorite flag',  
    iconCls : 'icon-help',  
    handler : btnHandler,  
    id : 'flagButton',  
    menu : {  
        defaults : {  
            checked : false,  
            group : 'colorChkGroup',  
            checkHandler : setFlagColor  
        },  
        items : [  
            { text : 'Red' },  
            { text : 'Green' },  
            { text : 'Blue' }  
        ]  
    }  
}); // 1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

### {1} Using the SplitButton instead of Button

To exercise the SplitButton, we're using the same setFlagColor handler we created in listing 12.8, and all of the code is the same except we're instantiating an instance of SplitButton instead of Button. That's it. Lets render this on screen so we can examine the visual and functional differences.



Figure 12.10 Our SplitButton in action.

As illustrated in Figure 12.10, we see that there is a separator line that has been added to the button, which is the only visual difference between a regular Button and SplitButton. But, this visual difference actually depicts the functional difference between the Button and SplitButton.

This visual separator between the arrow and the rest of the button is an indication of two distinct zones for the SplitButton, where the Button only has one, which is why when we clicked on our button, the handler was called *and* the menu appeared.

Clicking on what I like to call the "Button zone" in the SplitButton will call its registered handler. Likewise, clicking on what I like to call its "Arrow zone" will display the menu *and* call an arrowHandler if it was registered.

As we said earlier, the 3.0 Button family is a lot more flexible in terms of layout and can scale really well relative to its 2.0 ancestor. Next, we're going to discuss how to modify the look of the button itself and learn about some of the configuration options available to allow you to do just that.

#### 12.2.4 Customizing the layout of your Button

Thus far, we've only configured Buttons and SplitButtons for use without changing altering the layout. With this version of the framework, you can elect to display the (menu indication) arrow at the bottom of the button if you wish. You would do this by configuring the button via the arrowAlign property, which can have the value of "right" or "bottom".

Likewise, the icons can be aligned at any side of the button, but left is the default. Below is a quick demonstration of each of the arrowAlign(ment) and iconAlign options.



Figure 12.11 Various implementations of the SplitButton with the possible icon and arrow alignments.

In the illustration above, we used 16x16 icons, which fit perfectly in just about every portion of the framework where custom icons can be used. You can use larger icons if you wish. Also, even though we see the SplitButton being used in Figure 12.11, remember that Buttons also can be configured in the same way.

The height the Buttons and SplitButtons can be conveniently configured via the scale configuration property. This also controls the height of the icon that can be displayed in the button. There are three possible values for this property, which are small (16px high and is the default), medium (24px high) and large (32px high). We'll see this configuration property in action when we work through ButtonGroups in just a bit.

A great demonstration of larger icons in Buttons can be found in the examples that are available in the SDK. They can be found by viewing the following page in your browser: <your\_ext\_dir>/examples/button/buttons.html.

You now have the necessary skills to employ Buttons and SplitButtons with any possible icon and arrow alignment. Now that we have those important skills, we can learn how to cluster buttons together in what's known as a ButtonGroup.

## 12.3 Grouping your Buttons

New to Ext 3.0 is what is known as the ButtonGroup widget, which is an extension of Panel and has one sole purpose, which is for the grouping similar or like functionality into clusters for the user. Ext JS ButtonGroups do a pretty good job at emulating the button groups found in Microsoft Word 2007 for Windows.

Being an extension of Panel means that just about any layout can be employed to control the way the ButtonGroup looks. Please be aware that just because most layouts will work within the confines of the ButtonGroup, please heed to my advice and stay within the confines well-known UI design patterns. Remember, we like to keep the users happy.

Here is a pop quiz. Ready?

What class does Button extend? Right! The Button widget extends BoxComponent. Next question. Why is this important? Right again! This is important because BoxComponent is what contains the component size management methods that allow a Component to take part in a layout. You're sharp!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Next, we'll build our first ButtonGroup using the default layout. If you're following along and want to display the icons, please remember to include the icon CSS file.

### Listing 12.10 Grouping Buttons with a ButtonGroup

```

new Ext.ButtonGroup({
    renderTo : Ext.getBody(),
    title   : 'Manage Emails',
    items   : [
        {
            text      : 'Paste as',                                // 1
            iconCls   : 'icon-clipboard',
            menu     : [
                {
                    text      : 'Plain Text',
                    iconCls   : 'icon-paste plain'
                },
                {
                    text      : 'Word',
                    iconCls   : 'icon-paste word'
                }
            ]
        },
        {
            text      : 'Copy',
            iconCls   : 'icon-page white copy'
        },
        {
            text      : 'Cut',
            iconCls   : 'icon-cut'
        },
        {
            text      : 'Clear',
            iconCls   : 'icon-erase'
        }
    ]
});
{1} Constructing a new ButtonGroup
{2} Use a SplitButton instead of Button

```

In listing 12.10, we construct a new instance of `ButtonGroup{1}` with four buttons, one of which has an attached `Menu{2}`. Like the Buttons that we created earlier, we render the `ButtonGroup` to the document body. That's all there is to it. Here is what our `ButtonGroup` looks like rendered on screen.

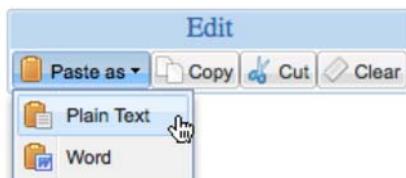


Figure 12.13 Our first ButtonGroup in action.

In Figure 12.13 we see our first ButtonGroup in action, which contains three buttons, the first of which is a SplitButton. By default, the ButtonGroup uses the TableLayout, which is why the buttons are arranged in a single row.

Next, we'll use our TableLayout ninja skills to rearrange these buttons so they look a little more like the edit button group in Word 2007 for Windows. If you're unfamiliar with the toolbars of MS Word 2007, below is a snapshot of what we're trying to achieve.

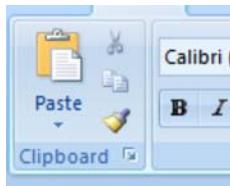


Figure 12.14 The Clipboard button group as found in MS Word 2007 for Windows.

OK, now that we know what we're going to replicate, let's get down to business.

### **Listing 12.11 Visually reorganizing our ButtonGroup**

```
new Ext.ButtonGroup({
    renderTo : Ext.getBody(), // 1
    title    : 'Clipboard',
    columns  : 2, // 2
    items    : [
        {
            text      : 'Paste',
            iconCls   : 'icon-clipboard 24x24',
            rowspan  : '3',
            scale    : 'large', // 3
            arrowAlign: 'bottom', // 4
            iconAlign: 'top'
        },
        {
            width    : 50,
            menu    : [
                {
                    text      : 'Plain Text',
                    iconCls   : 'icon-paste plain'
                },
                {
                    text      : 'Word',
                    iconCls   : 'icon-paste word'
                }
            ]
        },
        {
            iconCls : 'icon-cut'
        },
        {
            iconCls : 'icon-page white copy'
        },
        {
            iconCls : 'icon-paintbrush'
        }
    ]
});
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

- {1} Recreating the ButtonGroup
- {2} Specifing 2 columns, which is a TableLayout config
- {3} The paste button spans tree rows in the table
- {4} Scale Button and icon as large

In the above listing, we use all of our acquired knowledge thus far to leverage the default TableLayout that the ButtonGroup{1} uses to create an MS Word 2007-like button group. Here's how it works.

When recreating the ButtonGroup, we instruct the TableLayout to only render two columns{2}. This ensures that the Buttons within this group are going to be properly aligned.

We also modified how we created the first button in the set. We set a TableLayout only child configuration property, rowspan{3}, to 3, which ensures that this button spans the first three rows of the table. We also set the scale{4} to large, which ensures the modified famfamfam clipboard icon that I *stretched* to fit the 24x24 form factor so that it fits in the *large* Button nicely. Also notice that we set the arrowAlign to bottom, which helps us complete the look.

The rest of the three buttons have their text property removed, which means that their icon is the only visual representation of what the button does. Lets see what our changes render and compare it to what the MS Word 2007 button group looks like.

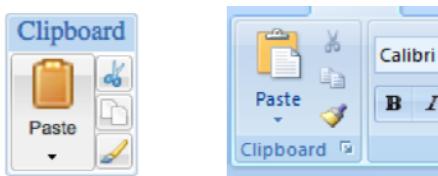


Figure 12.15 Our modified ButtonGroup(left) versus the Microsoft Word 2007 Button group (right).

OK, I know. Our latest's implementation of the Ext ButtonGroup is similar, but obviously it's not a 100% match. It's close though. And that's what is important to our users - right? The shallower the learning curve for the applications that we build for them the happier they'll be.

The point of this exercise wasn't to exactly duplicate the Word 2007 button group, but rather show that you can, if you choose to, configure the Ext ButtonGroup to organize the buttons it contains however your application requires.

As we just learned, ButtonGroups are a good tool to leverage to group similarly functioning buttons. The learning curve to using the ButtonGroup is relatively shallow because we already know how to use the various layouts, etc.

Now that we know how to use Menus, Buttons and ButtonGroups, we can shift gears and learn how to apply them to Toolbars, where they are most commonly used.

## 12.4 Toolbars

Toolbars are generally an area where you can layout a cluster of items for the user to interact with. In Ext 3.0, Toolbars can easily manage just about anything you want to put in there. This is because of the recent enhancements to the Toolbar class and the introduction of the ToolbarLayout.

Toolbars, just like any Component, can be renderedTo any element in the DOM, but are most commonly used in Panels and any descendant thereof. This is where we'll keep our focus for this Widget.

In the next example, we're going to create a Window that contains a Toolbar with quite a few items including a ComboBox as an exercise to show how easy it is to add Components other than Buttons to the Toolbar. The following Listing is going to be lengthy because of the amount of child items that we're going to create for the Toolbar.

### **Listing 12.12 Building a Toolbar within a Window**

```

var tbar = {
    items : [
        {
            text      : 'Add',
            iconCls   : 'icon-add'
        },
        '-',
        {
            text      : 'Update',
            iconCls   : 'icon-update'
        },
        '-',
        {
            text      : 'Delete',
            iconCls   : 'icon-delete'
        },
        '->',
        'Select one of these: ',
        {
            xtype     : 'combo',
            width    : 100,
            store    : [
                'Toolbars',
                'Are',
                'Awesome'
            ]
        }
    ];
};

new Ext.Window({
    width  : 500,
    height : 200,
    tbar   : tbar
}).show();

```

// 1  
// 2  
// 3  
// 4  
// 5  
// 6

{1} Creating the Toolbar configuration object  
{2} Shorthand for Toolbar separator  
{3} Shorthand for Toolbar greedy spacer  
{4} Shorthand for a Toolbar TextItem  
{5} A configuration object for a ComboBox

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

## {6} Adding the toolbar to the Window

In Listing 12.12, we create a Toolbar configuration object{1}. This object has a single property, `items`, which is a list of configuration objects and some strings. Notice that we don't have to specify an `xtype` for the toolbar configuration object. This is because when we associate it to the Window via the `tbar` or `bbar` properties, Ext will automatically assume that the configuration object is for a Toolbar for us.

The `items` array contains quite a few, well, items that we should discuss. The first, we've seen, which is a typical Button configuration with a `text` property and `iconCls`. The reason you don't see an `xtype` property for buttons is because the `defaultType` for toolbars is `button`.

After the first Button is defined, we see the shorthand for the `Toolbar.Separator`{2} class, which provides a vertical line dividing items in the Toolbar. While we could write out the configuration of the Separator, I typically find myself just using the shorthand. If you wanted to have granular control over the separator, you would need to write out the complete configuration object for it, including the `xtype` property, which is "`tbseparator`".

Moving on, we create two more buttons with another separator between them. After that, you'll find this funny looking string, '`->`', which is the shorthand for the `Toolbar.Fill`{3} widget. This little widget will instruct the Toolbar widget to place any item(s) defined thereafter in the right side of the toolbar. This effectively pushes the items over the right of the Toolbar.

### NOTE

Only one Fill item can be placed in the Toolbar. Think of it as an absolute divider, where between the left side of the Toolbar and the right. Once you place that divider, any items following after it will be placed to the right. Any other Fill definitions after the first will be ignored.

Next, we use the shorthand for the `Toolbar.TextItem`{4} widget. Just like the other shorthand strings, the `TextItem` can be configured via configuration object, which will give you greater control over it, which includes styling. I honestly only configure a `TextItem` via object if I need this control. Else, I just use the string shorthand.

After the `TextItem` configuration, we setup a quick and dirty `ComboBox`{5} configuration object. We do this to demonstrate how easy it is to embed items other than Buttons in the Toolbar.

Lastly, we render a new instance of Window and affix our Toolbar configuration object to the top Toolbar position by referencing via the `tbar`{6} property. Lets render it on screen and take a look at what it looks like.



Figure 12.15 Our Toolbar rendered inside of a Window.

When we look at our rendered Toolbar, we see that the three Buttons appear with Separators between them, all of which are left aligned. Right aligned are the TextItem and ComboBox.

The placement of Buttons in a Toolbar is relatively common practice, but it can quickly get overcrowded with items. Using ButtonGroups helps alleviate this situation. Adding ButtonGroups to a Toolbar is as simple as referencing it as a child item of the Toolbar.

The following illustration is a great example of ButtonGroups being used in the Ext Surf chat application.

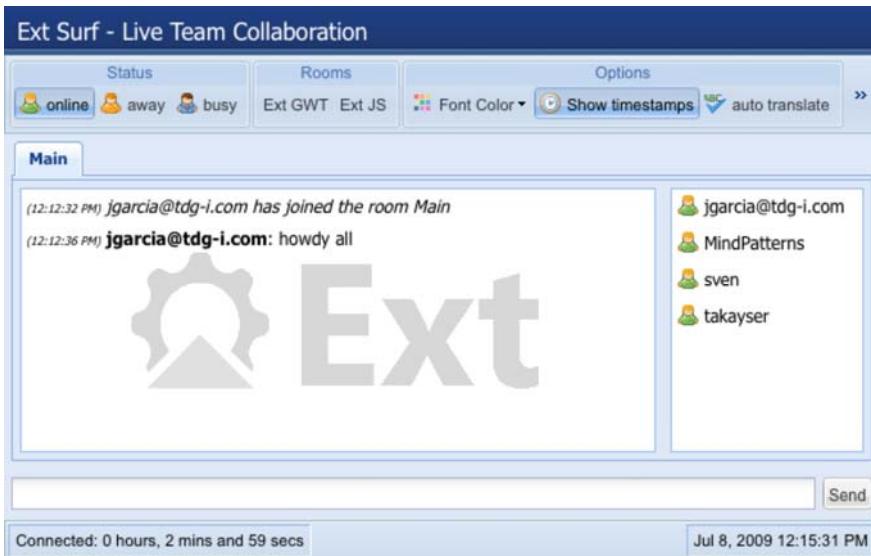


Figure 12.17 ButtonGroups used in Toolbar as seen in the Ext Surf live chat application.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

As we've learned, using Toolbars is relatively straightforward and we can add all types of stuff in the Toolbar to give our users the ability to effectively interact with our applications.

### THE DANGERS OF USING A DIFFERENT LAYOUT

Remember that the Toolbar uses the ToolbarLayout. This layout is what gives the Toolbar a lot of its power, which includes responsiveness to the ToolbarFill widget. We need to exercise some caution when thinking about using another layout with a Toolbar. While you *technically* can leverage another layout, you will lose features like the Toolbar Fill and automatic overflow protection, which are part of what make the Toolbar what it is.

OK. This wraps up our discussion with Toolbars. We can now focus on the last topic, Ext.Action, which ties together everything we've learned and exercised thus far.

## 12.5 Read, set, Ext.Action!

Often as developers, we're tasked to provide a means for the same functionality to be accessed by different methods. For instance, we may be asked to create an Edit menu that contains the typical cut, copy and paste menu items along with adding them directly to a toolbar.

With that request, you have two options. Either create instances of Button and menu Item each with similar configurations or abstract them to an Ext.Action. I vote the latter, as it's much easier.

Lets build out a couple of actions and we'll go into further detail about how this all works.

### Listing 12.13 Using Ext.Actions

```
var genericHandler = function(menuItem) {
    Ext.MessageBox.alert('', 'Your choice is ' + menuItem.text);                                // 1
}

var copyAction = new Ext.Action({
    text      : 'Copy',
    iconCls   : 'icon-page white copy',
    handler   : genericHandler
});                                                 // 2

var cutAction = new Ext.Action({
    text      : 'Cut',
    iconCls   : 'icon-cut',
    handler   : genericHandler
});

var pasteAction = new Ext.Action({
    text      : 'Paste',
    iconCls   : 'icon-paste plain',
    handler   : genericHandler
});
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var editMenuBtn = {
    text : 'Edit',
    menu : [
        cutAction,
        copyAction,
        pasteAction
    ]
}

new Ext.Window({
    width      : 300,
    height     : 200,
    tbar       : [
        editMenuBtn,
        '->',
        cutAction,
        copyAction,
        pasteAction
    ]
}).show();
{1} Creating a generic handler for the actions
{2} The creation of our first Ext.Action
{3} Adding the actions as menu items to a Menu
{4} Adding the actions as buttons to a Toolbar

```

In Listing 12.13 we do quite a lot to leverage `Ext.Action` to build out the hypothetically required UI. The first thing we do is recreate the `genericHandler{1}` that we used earlier in this chapter.

Next, we create three `Actions{2}` for the copy, cut and paste functionality that we desire. Notice that the properties set on each action are syntactically the same to those of the Button and menu items we've created in the past. This is part of the key to understanding what Actions are and how they work, which is why we're going to digress just a little.

Actions are essentially a class that do nothing more than encapsulate the common configuration of a Button or MenuItem. That's it. When you create an instance of a `Ext.Action` and inspect it in Firebug, you'll see a bunch of methods and properties. When we (or Ext JS) go to instantiate an instance of Button or MenuItem using Action, the properties and methods of the Action that we're using are applied to the Button or MenuItem during the constructor call of either class.

In other words `Ext.Action` is a configuration wrapper for either the Button or MenuItem, where many instances of each of those classes can share the Action configuration. Now what exactly Actions are, understanding the rest of this listing will be a breeze.

After we create the Actions, we create a Button`{3}` configuration object and set a menu property, which is an array of each of our actions. Next, we create and show an instance of `Ext.Window`, where we specify a tbar property, which is a list containing the Button with a menu, a Separator and the three actions. This is where we can see the usability of Actions.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Without rendering this on screen, we can predict what code will construct. Lets go ahead and display the page and see if we're right.

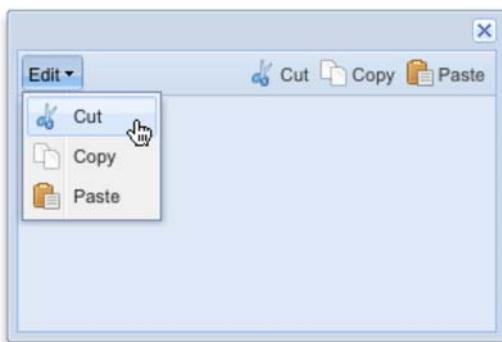


Figure 12.18 The results of Listing 12.13, where Actions are being used to configure Buttons and menu Items.

Ah-ha! The rendered UI from Listing 12.13 rendered just like we expected. We see three menu Items appear under the Edit Button when clicked and three Buttons that are right aligned in the Toolbar.

Unfortunately, Actions is one of those classes that is commonly overlooked by developers. I have to admit that I'm guilty of this myself. When developing Buttons and Menus for your UI, try to remember that if the functionality is the same for two or more Buttons or menu Items, use Actions. Trust me, it will save you time in the long run.

## 12.6 Summary

In this chapter we covered quite a lot of topics, which revolve around Menus, Buttons, Toolbars and Actions.

We started by learning the ins and outs of menus and all of the menu items that can be used. In doing so, we digressed a little to learn more about where to get free 16x16 icons that lots of web 2.0 applications are using. We also learned how to nest menus and even affix Color and Date menus to menu Items.

We also took time to learn about the Ext Button and SplitButton, where we learned how to configure how they look and feel by setting the iconAlign and arrowAlign properties. Next, we learned how to leverage the ButtonGroup to cluster similar buttons together and took the time to leverage the TableLayout to organize our buttons in a special way. Remember to set the scale property to medium or large if you're using an icon that is larger than 16x16.

Lastly, we began to tie together what we learned about menus and Buttons together when we discussed the Toolbar. We also took some time to learn about timesavings with the Ext.Action configuration wrapper class.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

In the next chapter, we're going to tackle the challenge of implementing Drag and Drop with DOM nodes.

# 13

## *Drag and Drop basics*

One of the greatest advantages of using a Graphical User Interface is the ability to easily move items around the screen with simple mouse gestures. This interaction is known as Drag and Drop. We use drag and drop just about every time we use a modern computer without giving it a second thought. If we take a step back and think about it, we can realize how this makes our lives a lot easier.

Want to delete a file? Simply click and drag the file icon and drop it on the trash or recycle bin icon. Easy, right? What if we didn't have drag and drop? How would we move a file from one location to another? Let's quickly ponder the possibilities. We first click the file to provide it focus. We could then use a keyboard key combination to "cut" the file. We then must then find and focus the destination folder window and use the keyboard key combination to "paste". Another option is click to select the file and press the delete key on your keyboard, but if you're right handed, this requires you to take your hand off your mouse and strike the delete key. In contrast, drag and drop is much simpler, isn't it? Now put yourself in the world of the RIA user. What if we could use drag and drop to simplify their experience?

Fortunately for us, Ext JS provides a means for us to do just that. In this chapter, we'll see that with a bit of elbow grease and determination, we can achieve the goal of adding drag and drop to our applications. We'll start off by learning to apply drag and drop to basic DOM elements, which will give us the foundation for applying these behaviors to widgets such as the `DataView`, `GridPanel` and `TreePanel`, which we'll learn about in the chapter following this.

## 13.1 Taking a good look at drag and drop

Before drag and drop can take place, the computer must decide what can and cannot be dragged and what can or cannot be dropped upon. For instance, Icons your desktop can generally be dragged around, but other items, such as the clock on your taskbar (Windows) or menu bar (OS X) cannot. This level of control is necessary to allow for the enforcement of certain workflows as we'll discuss in just a bit.

Now that the drag and drop registrations have taken place, we can now discuss what I like to call the drag and drop life cycle, which can be broken up into three major categories; start of drag, the actual drag operation and drop.

### 13.1.1 The drag and drop life cycle

Using the desktop paradigm, any icon on the desktop can be dragged around, but only a select few can be dropped on, which are generally disk or folder icons, the trash or recycle bin or icons for executables (applications). In Ext JS, the exact same registrations must occur for drag and drop to be possible. Any element that can take place in drag and drop must be initialized as such. For elements in the DOM to participate in drag and drop, they must, at the very least, be registered as drag items and as drop targets. Now that items are registered, drag and drop can take place.

"Drag" operations are initiated by the click and hold of a mouse button over a UI element followed by mouse movement while the mouse button is being held. The computer decides, based on if the registration we described above, if the item that is being clicked is "draggable". If it is not, then nothing happens. The user can click and attempt a "drag" operation with no results. But, if is an element that is allowed to drag, the UI generally creates a lightweight duplicate of that object, known as a "drag proxy" that is anchored to the movements of the mouse cursor. This gives the user the feeling that they are physically moving or "dragging" that item on screen.

During each "tick" or X Y coordinate change of the mouse cursor during the drag operation, the computer determines whether or not you can drop the item at that given position. If drop is possible then some sort of visual invitation of for a drop operation is displayed to the user. In the illustration below, we see a form of drop invitation when a file icon proxy is dragged over a folder icon on the desktop.

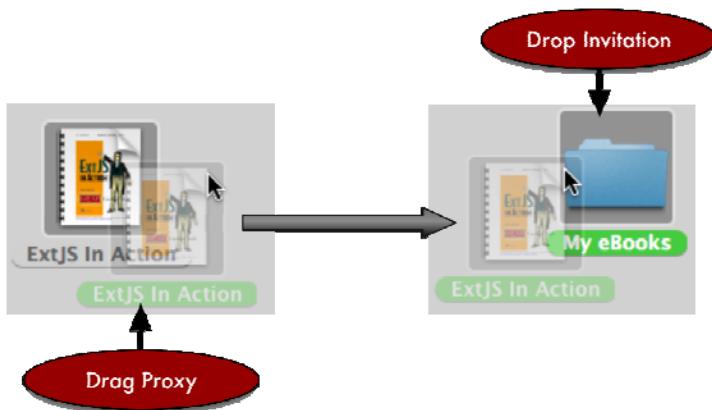


Figure 13.1 Desktop drag and drop interaction as seen in Mac OS X, where a drag proxy (left) is created upon an item drag event and drop invitation is being displayed (right).

The drag and drop lifecycle ends when a drop operation occurs. A drop operation is simply when the mouse button is released after a drag operation has occurred. It is at this time that the computer must decide what to do with the drop operation. Did the drop event occur on a valid drop target? If so, is it the drop target part of the same “drag and drop group” as the drag element? Does it copy or move the item that was dragged? This decision is generally left to the application logic to decide and where you’ll be doing most of your coding.

While it’s relatively easy to describe how drag and drop should behave, it is orders of magnitude more difficult to implement - but not impossible. In fact, one of the keys to being able to effectively implement drag and drop is a basic understanding of the class hierarchy and the jobs that each class performs. This holds true from implementing drag and drop from the basic DOM level, as we’ll see in just a little bit, all the way to implementing it on Ext JS UI widgets.

OK, throttle up. We’re going to climb up to 30,000 feet and take a birds eye view of the drag and drop class hierarchy.

### 13.1.2 A top-down view of the drag and drop classes

At first glance, the list of drag and drop classes can be a bit overwhelming. I know that when I first glanced at the list of classes in the API, I was taken back by the options. With fourteen classes, it actually can be considered a framework within a framework that provides functionality from the very basic, such as the ability to make any DOM element draggable, to more complex features such as the ability to drag and drop multiple nodes using what is called “Proxy”. The cool thing is that once you actually take a high level look at the classes, the supporting classes are not that difficult to organize and understand their roles.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

This is where we'll start our exploration. Below is what the class hierarchy looks like.

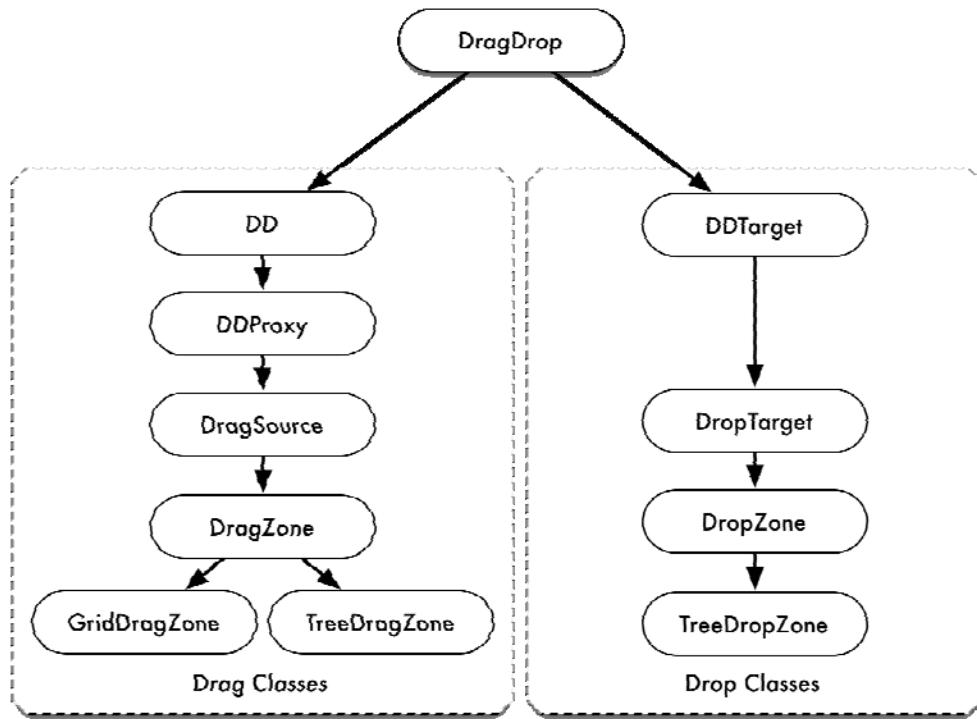


Figure 13.2 The drag and drop class hierarchy can be broken up into two major sections, drag (left) and drop (right).

In figure 13.2, we see the eleven drag and drop classes. All of the drag and drop functionality for the entire framework starts with the **DragDrop** class, which provides the base methods for everything drag and drop and are *meant* to be overridden. That is it provides only the basic tools to allow you to implement this behavior. It is up to write the code for the entire spectrum of the implementation.

This is fundamentally the first key in understanding how drag and drop works as this code design pattern is actually repeated throughout the drag and drop class hierarchy. This concept is extremely powerful because by having the basic tools to add this behavior to your application, you can easily ensure that drag and drop works for your application needs.

As we look down the chain of inheritance, we see that there is a split, which starts with **DD** (left) and **DDTarget** (right). **DD** is the base class for all *drag* operations, while **DDTarget** is the base class for all *drop* operations. Both provide the base functionality for their respective behaviors. Having this split in functionality allows you to focus your code for specific

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

behaviors. We'll see this in action when we look at implementing drag and drop with DOM nodes a little later on.

As we move down the chain, we see that Ext JS adds features progressively for the intended behavior. Below is a table enumerating the classes and a brief description of the task they are designed for.

Drag Classes		Drop Classes	
Name	Purpose	Name	Purpose
DD	A base drag implementation where an element can be dragged around and dropped anywhere. This is where most DOM-level drag implementations take place.	DDTarget	The basic to allow any element can participate in a drag and drop group, but cannot be dragged around, meaning it can only have items dropped on it.
DDProxy	A basic drag implementation where a lightweight copy of the drag element, known as a "drag proxy", is dragged around instead of the source element. It is common practice to use this class for drag operations where a drag proxy is desired.	DropTarget	A base class that provides the empty plumbing for an element to take action when a draggable element is dropped on to this one. It is left up to the developer to finish the implementation by overriding the notify* methods.
DragSource	Provides a base implementation for drag and drop using what is known as "status proxies" and is the base class for DragZone. This can be used directly, but it is more common to use the DragZone class (see below).	DropZone	A class that provides the means for multiple nodes to be dropped onto this element and works best with the DragZone class. There is a TreePanel specific implementation of this known as the TreeDropZone.
DragZone	This class allows for the drag of multiple dom elements at a time and is commonly used with the DataView or ListView widget. To provide drag and drop with the GridPanel and TreePanel, each have their own implementations of this class known as the GridDragZone and TreeDragZone.		

There you have it, each of the drag and drop classes ad what they are designed to do. It's very important to know that a drag item (DD class or subclasses) can be a drop target, but a drop target (DDTarget or subclasses) cannot be a drag item. It's important to know this because if you ever decide on having an element be a drag item **and** drop target, you must use one of the drag classes.

If you're implementing drag and drop with generic DOM nodes, where you need to allow for one drag node at a time, you would use the DD or DDProxy, as we'll see later on in this chapter. If you're looking to drag more than one element, you will want to use either the DragSource or DragZone classes. This is why the TreePanel and GridPanel have their own respective extensions or implementations of the DragZone class.

Likewise, if you're looking to drop a single node, the DDTTarget will be your drop class of choice. For multiple node drops, the DropTarget or DropZone are required because they have the necessary plumbing to interact with the DragSource, DragZone and its descendant classes.

Knowing what the classes are is just one piece of the puzzle. The next piece that you'll need to know is what methods are to be overridden, which is the biggest keys to successful deployment.

### 13.1.3 It's all in the overrides!

As we discussed earlier, the various drag and drop classes were designed to provide a base framework for various drag or drop behaviors and are only part of what is needed to really make drag and drop useful. Each of the drag and drop classes contain a set of abstract methods that are meant to be overridden by the you, the end developer.

While all of these methods are listed in the framework API for each drag drop class, it is a good idea that we at least briefly discuss a few of the more commonly used abstract methods that are to be overridden for the Ext.dd.DD class, this way you get a sense of what to look for in the API.

Method	Description
onDrag	Called when for each onMouseMove event while the element is being dragged. If you plan on doing something before an item is dragged, you may elect to override the b4Drag or startDrag methods.
onDragEnter	Called when a drag element first intersects another drag drop element within the same drag drop group. This is where you can code for drop invitation.
onDragOver	This method is called while a drag element is being dragged over
onDragOut	Called when a drag element 'leaves' the physical space of an associated drag or drop element.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

- onInvalidDrop This is called when a drag element is dropped on anything other than an associated drag or drop element. It's a great place to inject notification to the user that they dropped a drag element in the wrong place.
- onDragDrop This method is called when a drag element is dropped on another drag drop element within the same drag drop group.

Remember that Ext.dd.DD is the base class for all drag-specific elements and as you move down the chain of hierarchy, more features get added. The features are added by the subclasses progressively override these methods for you.

For instance, there are a few "b4" (before) methods that Ext.dd.DD provides that allow one to code a behavior before something happens, such as before the mouse down event fires (b4MouseDown) and before the drag of an element occurs (b4StartDrag). Ext.dd.DDProxy, the first subclass of Ext.dd.DD overrides these methods to create a dragable proxy just *before* the actual drag code begins to execute.

To figure out which methods you need to override to achieve a specific implementation, you'll need to consult the API for that specific drag or drop class. Because Ext JS has a more aggressive release cycle, obviously some methods may be added, renamed or removed, so looking at the API regularly will help you stay fresh with the current changes.

The last bit of drag and drop theory we need to discuss is the use of what is known as drag and drop groups and what they mean for implementing this behavior in your application.

### 13.1.4 Drag and drop always work in groups

Drag and drop elements are associated by what are known as groups, which is the basic constraint that governs whether or not a drag element can be dropped another. A group is basically a label that helps the drag and drop framework decide if a registered drag element should interact with another registered drag or drop element.

Drag or drop elements must be associated with at least one group, but can be associated with more than one. They are generally associated with a group upon instantiation and can be associated with more via the addToGroup method. Likewise, they can be unassociated via the removeFromGroup method.

This is the last piece of the puzzle to understanding the basics of drag and drop with Ext JS. It is time to start exercising and reinforcing what we've learned. We'll start out simple by implementing drag and drop with DOM elements.

## 13.2 Starting out simple

We'll begin our exploration with a plot that mimics a swimming pool setting, complete with locker rooms, a swimming pool and a hot tub. There are constraints that we must follow. For instance, men and women nodes can only be in their respective locker rooms. All can go

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

in the swimming pool, but only a few really like to go into the hot tub. Now that we understand what we must create, we can begin coding.

As we'll see, it is extremely simple to configure an element to be dragged around and about on screen. But before we can do that, we must create a workspace to manipulate. We'll do this by creating some CSS styles that govern how a specific set of DOM elements look and then apply drag logic to them. We'll keep things as simple as possible to focus on the subject matter.

### 13.2.1 Creating a small workspace

Lets create the markup to represent the locker rooms and people inside of them. This listing will is rather lengthy due to the HTML required to achieve the desired style and layout.

#### **Listing 13.1 Creating our simple drag and drop workspace**

```
<style type="text/css">
    body {
        padding: 10px;
    }

    .lockerRoom {
        width:          150px;
        border:         1px solid;
        padding:        10px;
        background-color: #ECECEC;
    }

    .lockerRoom div {
        border:         1px solid #FF0000;
        background-color: #FFFFFF;
        padding:        2px;
        margin:         5px;
        cursor:         move;
    }
</style>

<table>
    <tr>
        <td align='center'>
            Male Locker Room
        </td>
        <td align='center'>
            Female Locker Room
        </td>
    </tr>
    <tr>
        <td>
            <div id="maleLockerRoom" class="lockerRoom">
                <div>Jack</div>
                <div>Aaron</div>
                <div>Abe</div>
            </div>
        </td>
        <td>
            <div id="femaleLockers" class="lockerRoom">
                <div>Sara</div>
            </div>
        </td>
    </tr>
</table>
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        <div>Jill</div>
        <div>Betsy</div>
    </div>
</td>
</tr>
</table>

```

- {1} Configuring styles for the drag elements container
- {2} Ensuring the child nodes look different from their parent.
- {3} The HTML markup for our drag items.

In listing 13.1, we create the CSS styles and markup to set the stage for our exploration of basic DOM drag and drop. We begin by defining the CSS that will control the styles for the lockerRoom{1} element containers and their (people) child nodes{2}. Lastly, we setup the actual markup{3} to utilize the CSS.

Here is what our locker HTML looks like rendered.

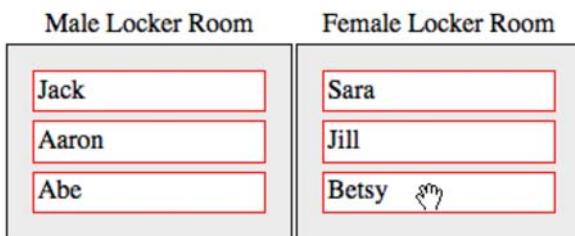


Figure 13.3 The locker room HTML rendered.

In figure 13.3, we see our HTML rendered on screen. If you notice, when you hover the mouse cursor over a child node of a locker room element, the arrow will change into a hand. This is due to the CSS styling that we configured above, and provides a nice means of inviting a drag operation.

Next, we'll configure the JavaScript to allow for these elements to be dragged around.

### 13.2.2 Configuring items to be “dragable”

In the listing below, we're going to configure the locker room child items to be "draggable". We're going to see exactly how easy it is to do this.

#### Listing 13.2 Creating our simple drag and drop workspace

```

var maleElements = Ext.get('maleLockerRoom').select('div'); // 1
Ext.each(maleElements.elements, function(el) {
    new Ext.dd.DD(el); // 2
});

var femaleElements = Ext.get('femaleLockerRoom').select('div');
Ext.each(femaleElements.elements, function(el) {

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
    new Ext.dd.DD(e1);  
});  
{1} Gather a list of child items  
{2} Make child items draggable
```

In Listing 13.2, we try to be dragged around screen. To accomplish this, we gather a list of child elements in the maleLockerRoom element by chaining a `select` call (DOM Query) to the results of the `Ext.get` call **{1}**. We then utilize `Ext.each` to loop through the list of child nodes and create a new instance of `Ext.dd.DD`, passing the element reference, which enables that element to be dragged around screen. We do the exact same thing for the elements in the femaleLockerRoom.

After refreshing the page, we can easily drag and drop the elements around the screen.

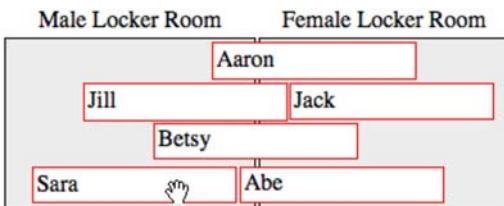


Figure 13.4 Drag enabled on the locker room elements without constraints.

As we can see in Figure 13.4, we can drag any of the child Divs around the screen, without constraints. Lets examine how `Ext.dd.DD` works and what it does to our DOM elements. To do this, we'll need to perform another refresh of the page and open up firebug's live HTML inspection tool. We'll focus in on Jack.

### 13.2.3 Analyzing the `Ext.dd.DD` DOM changes

Illustrated below is the HTML of our drag elements just after a page refresh along with the DOM inspection view in firebug.



Figure 13.5 Inspecting the DOM for the Jack element (highlighted in blue) before a drag operation takes place.

When looking at the Jack (highlighted in blue) element, the first thing that you might notice is that it is assigned a unique ID of "ext-gen3". Recall that in our markup, we didn't assign an ID to this element. If the element already had its own unique ID, Ext.dd.DD would use it. But, instead, in order to track this element by ID, it is assigned one by the Ext.dd.DD's superclass, Ext.dd.DragDrop.

#### **WARNING**

If the id of the ID is changed after it has been registered for drag, the drag configuration for that element will cease to function. If you plan on changing the ID for a particular element, it is best to call the destroy method on the instance of Ext.dd.DD for that element and create a new instance of Ext.dd.DD, passing the new element ID as the first parameter.

Another thing that we'll notice from looking at the HTML inspection is that there are no other attributes assigned to that element. Now, let's drag the element over a little and observe the changes.



Figure 13.6 Observing the changes that the drag operation makes on the Jack element.

In figure 13.6, I dragged the Jack element just a little. Ext.dd.DD, in turn, added a style attribute to the element, which changes the position, top and left CSS properties. This is important to know because usage of Ext.dd.DD will result in the change of positioning for the element on screen and is one of the key differences between using the Ext.dd.DD and the Ext.dd.DDProxy, which we'll explore later on.

The last observation that we will discuss is the ability for the dragged elements to be seemingly dropped anywhere. At first this may seem cool, and it is! But, it's hardly of any use. In order to make this useful, we will have to apply constraints.

To do this, we'll need to create some containers for us to be able to drop them on to. This is where we'll generate the pool and hot tub for these people to enjoy.

### 13.2.4 Adding the pool and hot tub drop targets

Just as before, we're going to have to add some CSS to stylize the HTML. Please insert the following CSS inside the style tags of your document. They will set the background color of the pool and hot tub blue and red, respectively.

```
.pool {
    background-color: #CCCCFF;
}

.hotTub {
    background-color: #FFCCCC;
}
```

OK, now that we have that said, we'll need to add the HTML to the document body. Append the following HTML markup below the locker room HTML Table.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

<table>
  <tr>
    <td align='center'>
      Pool
    </td>
    <td align='center'>
      Hot Tub
    </td>
  </tr>
  <tr>
    <td>
      <div id="pool" class="lockerRoom pool"/>
    </td>
    <td>
      <div id="hotTub" class="lockerRoom hotTub"/>
    </td>
  </tr>
</table>

```

The above HTML will give us the Elements we'll need to setup drop targets. Here is what the HTML now renders to.

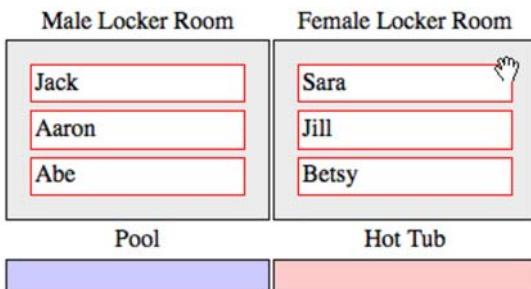


Figure 13.7 The Pool and Hot Tub html rendered on screen.

OK, we've added all of the HTML that we'll need to add for now. What we'll need to do next is setup the Pool and Hot Tub elements as what are known as DropTargets, which will enable them to participate in the drop portion of the drag and drop. We'll add this code just after the JavaScript in listing 13.2.

```

var poolDDTarget = new Ext.dd.DDTarget('pool', 'males');
var hotTubDDTarget = new Ext.dd.DDTarget('hotTub', 'females');

```

In the snippet above, we setup an instance of Ext.dd.DDTarget each for the 'pool' and 'hotTub' elements. The first parameter for the DDTarget constructor is the ID of the element (or DOM Reference) and a configuration parameter that is used to. The second parameter is the group for which the DDTarget is to participate in.

Now, refresh your page and drag and drop a male node to the pool node or a female node to the hot tub node. What happens when you drop the item on the drop target? Yes - ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

nothing. Why is that? Well, we setup drag items and the drop targets, which set the stage for complete drag and drop, but remember that it is left up to us to follow through with the rest of the implementation. We must develop the code for the drop invitation, valid and invalid drop. As we'll see, this is precisely where most of the drag and drop implementation code will take place and is what we're going to do next.

### 13.3 Finishing our drag and drop implementation

As we just saw, setting up an element to be dragged around and about is very simple. Likewise is setting up a drop target. But unless we connect the dots, we're left with a source and destination, but no way to get there.

In order to add the drop invitation and valid and invalid behaviors, we'll need to refactor how we configure the elements to be dragged around and about. We'll begin by adding one last CSS class, which we'll use to turn the drop target green for the drop invitation.

```
.dropZoneOver {
    background-color: #99FF99;
}
```

As you can see, the CSS above is really simple. Whatever element has this class will have a green background. Next, we'll work on refactoring the way we setup the male and female elements to be dragged around by the means setting up an overrides object that gets applied to each instance of Ext.dd.DD.

#### 13.3.1 Adding the drop invitation

To add the drop invitation, we're going to have to completely replace how we initialized the drop targets before. The below code is what we'll use and will set the stage for the valid and invalid drop behaviors.

#### Listing 13.3 Refactoring our implementation of Ext.dd.DD

```
var overrides = { // 1
    onDragEnter : function(evtObj, targetElId) { // 2
        var targetEl = Ext.get(targetElId);
        targetEl.addClass('dropZoneOver');
    },
    onDragOut : function(evtObj, targetElId) { // 3
        var targetEl = Ext.get(targetElId);
        targetEl.removeClass('dropZoneOver');
    },
    b4StartDrag : Ext.emptyFn,
    onInvalidDrop : Ext.emptyFn,
    onDragDrop : Ext.emptyFn,
    endDrag : Ext.emptyFn
};

var maleElements = Ext.get('maleLockerRoom').select('div');
Ext.each(maleElements.elements, function(el) {
    var dd = new Ext.dd.DD(el, 'males', { // 4
        isTarget : false // 5
    });
});
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

    Ext.apply(dd, overrides);                                // 6

}); var femaleElements = Ext.get('femaleLockerRoom').select('div');
Ext.each(femaleElements.elements, function(el) {
    var dd = new Ext.dd.DD(el, 'females', {
        isTarget : false
    });
    Ext.apply(dd, overrides);
});
{1} Creating the overrides object
{2} Coloring the DropTarget element green, inviting drop.
{3} Remove the drop invitation
{4} Setting up the male elements to be drag items
{5} These drag elements will not be drop targets.
{6} Applying the override methods to the DD instance.

```

In the code above, we create an object, `overrides`, which will be *applied* to the instances of `Ext.DD` that will be created. There are a total of five methods that we'll override to achieve the desired results, but for now, we're only going to override `onDragEnter{1}` and `onDragout{2}`.

Remember that `onDragEnter` is only going to be called when a drag element first intersects a drag or drop element with the same associated group. Our implementation of this method will add the '`dropzoneOver`' CSS class, which changes the background-color of the drop element to green and provides the drop invitation that we want.

Likewise, the `onDragOut` method will get called when the drag element first leaves a drag and drop object with the same associated group. We use this method to remove the invitation from the background of the drop element.

We then move on to stub three methods, `b4StartDrag`, `onInvalidDrop`, `onDragDrop`, and `endDrag` which we'll fill in later on. We don't do these now because I want you to be able to focus on the behaviors and constraints that we add in layers. But, just in case you're curious, we'll use `b4StartDrag` to get the original X and Y coordinates of the drag element. These coordinates will be used in the `onInvalidDrop` method, which will simply set a local property to indicate that this method was fired. The `onDragDrop` method will be used to actually move the drag node from its original container to the dropped container. Lastly, the `endDrag` method will reset the position of the drag element if the `invalidDrop` property is set to true.

In order to use this override object, we have to refactor how we're initializing the drag objects{4}. This is done for both the male and female elements alike.

We do this because we need to prevent the drag element from being a drop target itself, which is why we add a third argument to the `DD` constructor, which is meant to be a somewhat limited configuration object. You'll see what I mean by *limited* in just a bit. In that configuration parameter, we set `isTarget{5}` to false, which sets the controlling behavior for this drag item not to be a drop target.

Lastly, we *apply* the `overrides` object to the newly created instance of `Ext.DD`. Earlier, I said that the configuration object is only used to set a limited amount of properties. The

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

reason I said this is because the drag and drop code that exists in Ext JS today was written back in the early Ext JS 1.0 days before most constructors applied configuration properties to themselves. This is why we have to use Ext.apply to *inject* the override methods instead of setting them on the configuration object like we would for most constructors in the framework.

OK, we've just added the code for the invitation. Lets see what happens when we try to drag a male node over the pool or hot tub.

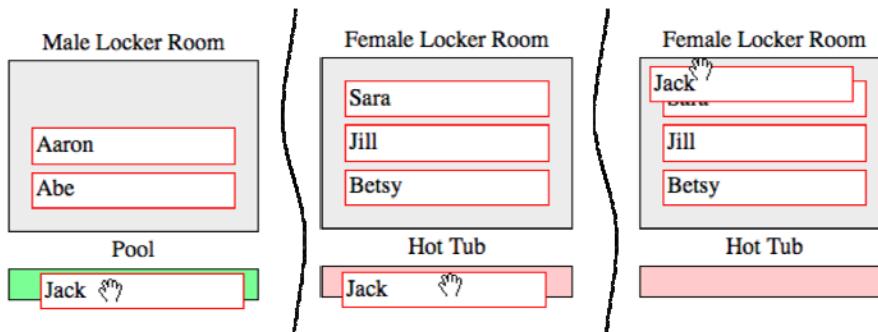


Figure 13.8 Conditional drop invitation for the male nodes.

As our knowledge of drag and drop and code dictates, a drag of a male node over a drop target (think onDragEnter) with the same associated group will result in a drop invitation, which is the background of the drop target turning green as seen in Figure 13.XX. When you drag the element out of the same drop target (think onDragOut), the background will return to its original state, thus removal of the drop invitation.

Conversely, a drag of a male element over any other drop target, such as the hot tub will result in no invitation. Why does this happen? Correct! The reason there is no drop invitation on the hot tub element is because the hot tub is not associated with the "males" drop group.

Another thing you will notice is that dragging a female element over the hot tub results in a drop invitation on the hot tub element but not the pool. That's because the hot tub is only associated with the "females".

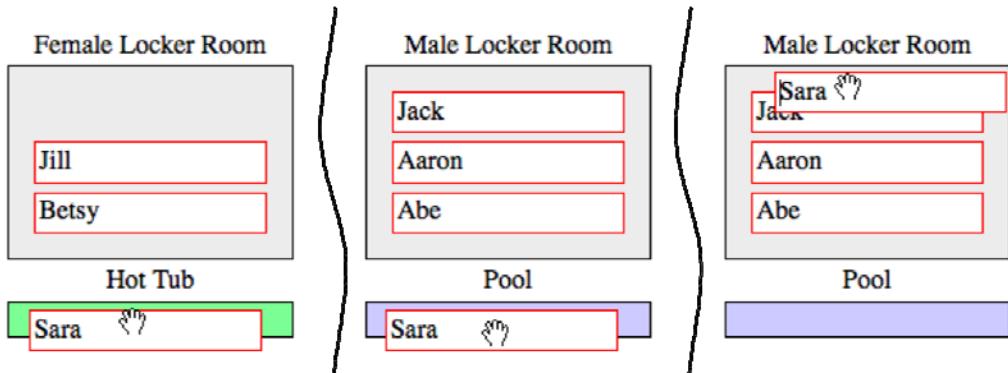


Figure 13.9 The conditional drop invitation for the female nodes

While this exercises the drop invitation very well, there is the issue of the pool and the hot tub needing to be able to receive both male and female nodes. To do this, we must register them with an additional group. To do this, we need to call the `addToManyGroups` method, passing in an alternate group. Here is what the pool and hot tub element DDTTarget registration looks like with the following `addToManyGroups` calls.

```
var poolDDTTarget = new Ext.dd.DDTTarget('pool', 'males');
poolDDTTarget.addToManyGroups('females');

var hotTubDDTTarget = new Ext.dd.DDTTarget('hotTub', 'females');
hotTubDDTTarget.addToManyGroups('males');
```

After injecting the following to our example, refresh the page. We can now see that the pool and the hot tub drop elements now invite the drop, but what happens when we drop a drag element on a valid drop target? Absolutely nothing. That's because we didn't code for the valid drop operation.

We'll do this next.

### 13.3.2 Adding valid drop

To add the "valid drop" behavior to our drag and drop implementation, we must replace the `onDragDrop` method in your overrides object the following.

#### **Listing 13.4 Adding valid drop to our overrides**

```
onDragDrop : function(evtObj, targetElId) {
    var dragEl = Ext.get(this.getEl());
    var dropEl = Ext.get(targetElId); // 1

    if (dragEl.dom.parentNode.id != targetElId) // 2
        dropEl.appendChild(dragEl);
    this.onDragOut(evtObj, targetElId); // 3 // 4
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
dragEl.dom.style.position = '';
}
else {
    this.onInvalidDrop(); // 5
}
{1} Create local references for drag and the drop targets
{2} Is the drag parent element not the same as the drop target
{3} Move the node from one container element to another.
{4} Clear the drop invitation
{5} Move the drag node back to where it was before drag
```

In our `onDragDrop` method, we setup the code for a successful or valid drop operation. To do this, we first need to create local references for the drag element, `el{1}`, and the drop element.

Next, we hit a conditional if statement`{2}`, where we test to see if the drag element's parent node's ID is the not the same as the drop target id and exercises the control where we don't wish to perform a drop operation on a drop target where the drag element is already a child. So if the drop target element is not the same as the drag element's parent, then we allow the drop operation to occur, else, we call the `onInvalidDrop` method, which we'll code for in just a bit.

The code to physically move the drag element from one parent container to another is very simple. Simply call the drop element's `appendChild{3}` method, passing in the drop element. Remember that even though Ext.dd.DD is allowing you to move the drag element on screen, it's only changing the X and Y coordinates. If we did not move the drag element to another parent node, it will still be a child of its original container element.

Next, we call the our `onDragOut` override`{4}`, which will clear the drop invitation. Notice that we're passing the `eventObj` and `targetElId` arguments along to the `onDragOut` method. This is so the `onDragOut` method can do its job as designed.

Lastly, we clear the element's `style.position` attribute. Recall that DD sets the position to "relative", which really isn't needed after the node has been moved from one parent container to another.

This ends the override of the `onDragDrop` method. Let's see what this does to our page.

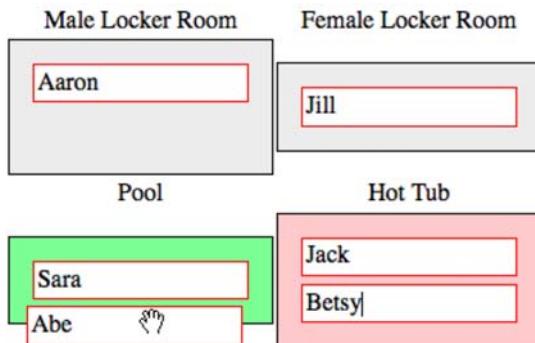


Figure 13.10 Male and female nodes can now be dropped onto the pool and hot tub drop targets.

As illustrated in Figure 13.10 we can successfully drop male and female elements to both the pool and hot tub drop elements, which successfully demonstrates our `onDragDrop` method in action.

While it's nice that the males and females can now be dropped into the pool or hot tub, we can't let them stay in there forever or they'll prune. We need to be able to pull them out and put them back into the locker room. What happens if you try to drag them over their respective locker room? No invitation. Why? Correct. It's because we have not registered the locker room elements as DDTTargets. Lets do that.

```
var mlrDDTarget = new Ext.dd.DDTTarget('maleLockerRoom', 'males');
var flrDDTarget = new Ext.dd.DDTTarget('femaleLockerRoom', 'females');
```

Adding the above code to the bottom of our drag drop implementation allows the male drag elements to be invited and dropped on every drop target except for the female locker room. Likewise, female drag elements can be dropped on any drop target except the male locker room. This follows the paradigm where most public places don't have coed locker rooms.

We now have the drop operations completely developed. The last piece to this implementation is the invalid drop behavior set, which is what we'll work on next.

### 13.3.3 Implementing invalid drop

By now we've implemented drop invitation and valid drop to our example set. I'm sure by now you've noticed that when you drop a node anywhere on screen other than a valid drop point the element stays stuck where it was dropped. This is because we need to setup the invalid drop behavior that will place the element back to its original position.

We're going to do this with some style using the Ext.fx class. The following code will replace the `b4StartDrag` and `onInvalidDrop` methods in the `overrides` object.

#### Listing 13.5 Cleaning up after invalid drop

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

b4StartDrag : function() { // 1
    var dragEl = Ext.get(this.getEl());
    this.originalXY = dragEl.getXY(); // 2
},
onInvalidDrop : function() { // 3
    this.invalidDrop = true;
},
endDrag : function() { // 4
    if (this.invalidDrop === true) {
        var dragEl = Ext.get(this.getEl());

        var animCfgObj = { // 5
            easing : 'elasticOut',
            duration : 1,
            callback : function() {
                dragEl.dom.style.position = '';
            } // 6
        };
        dragEl.moveTo(this.originalXY[0], this.originalXY[1], animCfgObj); // 7
        delete this.invalidDrop;
    }
},

```

- {1} Override the b4StartDrag method
- {2} Store the original X and Y coordinates for the drag element.
- {3} Set this.invalidDrop to true
- {4} Animate the return of the drag element if this is an invalid drop
- {5} Create the animation config
- {6} Reset the drag element's position
- {7} Perform the animation to reset the drag element's position

In Listing 13.5, we first override the `b4StartDrag`{1} method, which is called the moment a drag element is being dragged around. It is at this juncture in the time that we are able to store the drag element's original X and Y coordinates{2}, which will be used for the repair operation. To "repair" an invalid drop means to reset the position of the drag element or proxy (as we'll see in just a bit) to its original position before the drag operation took place.

Next, we override `onInvalidDrop`{3}, which is called when the drag item is dropped on anything other than a valid drop point, which is associated with the same group. In that method, all we do is set the local `invalidDrop` property to true, which is going to be used in the next method, `endDrag`.

Lastly, we override the `endDrag` method, which will actually perform the repair operation if the local `invalidDrop` property is set to true{4}. It also uses the local `originalXY` property set by the `b4StartDrag` method. Essentially, what this method does is create a configuration object for the animation{5}.

In the configuration object, we set the easing to 'elasticOut', which will give the element a nice springy or elastic end to the animation, and set the duration to one second. This ensures the animation is smooth and not jerky. We also create a callback method to reset

the drag element's `style.position` attribute<sup>{6}</sup>, which will ensure that the drag element fits exactly where it needs to go.

**NOTE**

If you wanted to forego the animation and just reset the position of the drag element, all that `onInvalidDrop` has to do is simply set the `style.position` to an empty string like so:  
`dragEl.dom.style.position = "";`

Next, the drag element's `moveTo` method is called, passing in the X and Y coordinates as the first and second parameters and the animation configuration object as the third. This actually invokes the animation on our drag element.

Lastly, we delete the local `invalidDrop` reference, as it is no longer needed. We will need to refresh the page to see these three override methods in action.

When we move drag an element and drop it anywhere other than a like-associated drop element, we see that it slides back to its original position with and has a cool springy effect when it gets to its target X and Y coordinates<sup>{7}</sup>.

We've just seen what it takes to implement drag and drop with the `Ext.dd.DD` and `Ext.DD.DDTarget` classes. Next, we'll see how to implement the `DDProxy` class, which is extremely similar. This is going to be fun.

### **13.4 Using the DDProxy**

The use of drag proxies in drag and drop implementation is extremely common and is something that is worth going over as the implementation is similar to `DD` but not quite the same. The reason for this is because the `DDProxy` essentially allows you to drag around a lightweight version of the drag element, which is known as the "drag proxy". Use of the `DDProxy` could result in huge performance savings if the drag element is extremely complex. Part of the performance savings comes from the fact that every instance of `DDProxy` uses the same proxy div element in the DOM. Remembering that the drag proxy is the element being moved around on screen will help you understand our implementation cod.

In this exercise, we will use the same HTML and CSS that we use before, and will give provide the pattern that you will need to use if you plan on using drag proxies in your drag and drop implementations.

The first thing we'll need to do is add one more CSS rule to our page, which will style the drag proxy with a yellow background.

```
.ddProxy {  
    background-color: #FFFF00;  
}
```

We're going to follow the exact same flow as we did with implementing the DD class. In doing this, we're going to see that implementing the DDProxy takes a bit more code than the DD class.

### 13.4.1 Implementing DDProxy and drop invitation

The DDProxy class is responsible for creating and managing the X and Y coordinates of the reusable proxy element, but it is up to us to actually style and fill it with content. We'll do this by means of overriding the startDrag method instead of the b4Drag method like we did with the DD implementation.

In the listing below, we're going to create the overrides object along with the creation of the instance of DDProxy. This listing is going to be rather long, but we're accomplishing quite a bit.

#### **Listing 13.6 Implementing drop invitation**

```

var overrides = {
    startDrag : function() { // 1
        var dragProxy = Ext.get(this.getDragEl());
        var dragEl = Ext.get(this.getEl());

        dragProxy.addClass('lockerRoomChildren');
        dragProxy.addClass('ddProxy'); // 2
        dragProxy.setOpacity(.70);
        dragProxy.update(dragEl.dom.innerHTML);
        dragProxy.setSize(dragEl.getSize());
        this.originalXY = dragEl.getXY();
    },

    onDragEnter : function(evtObj, targetElId) { // 3
        var targetEl = Ext.get(targetElId);
        targetEl.addClass('dropzoneOver');
    },

    onDragOut : function(evtObj, targetElId) {
        var targetEl = Ext.get(targetElId);
        targetEl.removeClass('dropzoneOver');
    },

    onInvalidDrop : function() {
        this.invalidDrop = true;
    },

    onDragDrop : Ext.emptyFn // 4
};

var maleElements = Ext.get('maleLockerRoom').select('div'); // 5
Ext.each(maleElements.elements, function(el) {
    var dd = new Ext.dd.DDProxy(el, 'males', {
        isTarget : false
    });
    Ext.apply(dd, overrides);
});

var femaleElements = Ext.get('femaleLockerRoom').select('div');
Ext.each(femaleElements.elements, function(el) {
    var dd = new Ext.dd.DDProxy(el, 'females', {
        isTarget : false
    });
});

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

    Ext.apply(dd, overrides);
});

{1} Override the startDrag method
{2} Stylize the drag proxy
{3} Add the drop invitation
{4} Add the onDragDrop stub
{5} Instantiate instances of DDProxy for each node

```

In listing 13.6, we accomplish the task of stylizing the proxy, adding drop invitation and instantiating the instances of Ext.dd.DDProxy for each of the elements. Here's how this all works.

The startDrag method **{1}** takes care of stylizing the drag element by means. It does this by first adding the lockerRoomChildren and ddProxy CSS classes **{2}** to the drag proxy element. Next, it sets the proxy's opacity to 70% and duplicates the html contents of the drag element. It then sets the size of the drag proxy to the size of the drag element. Lastly, the originalXY property is set, which will be used for an invalid drop repair operation down the road.

Next, we add the drop invitation by means of overriding the onDragEnter and onDragOut methods. This is exactly the same as the prior implementation. The onInvalidDrop override is exactly the same as before as well. The last override is a stub for the onDragDrop method, which we will fill out in just a bit.s

Before we actually can use the drop invitation, we'll have to setup the drop targets for the pool, hot tub and locker room elements.

```

var poolDDTarget = new Ext.dd.DDTTarget('pool', 'males');
poolDDTarget.addToGroup('females');

var hotTubDDTarget = new Ext.dd.DDTTarget('hotTub', 'females');
hotTubDDTarget.addToGroup('males');

var mlrDDTarget = new Ext.dd.DDTTarget('maleLockerRoom', 'males');
var flrDDTarget = new Ext.dd.DDTTarget('femaleLockerRoom', 'females');

```

Now that we have that setup, lets exercise the DDProxy implementation we've cooked up thus far. Refresh your page and drag a drag element around.

Below is what the drag proxy looks like in action.

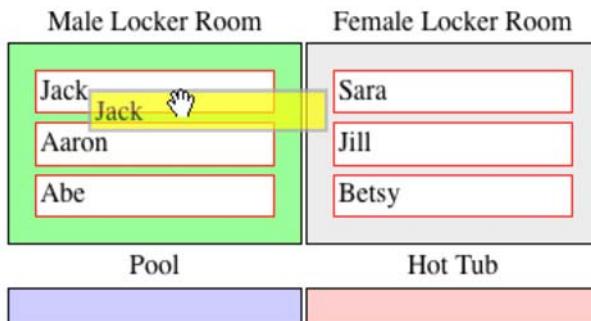


Figure 13.11 The DDProxy in action with one of our male drag elements.

As we can see in Figure 13.11, the performing a drag gesture on a dragable element produces the drag proxy, which is allowed to be dragged around while the drag element itself is stationary. We also see that the drop invitation works. What happens, when we drop the drag element on a valid or invalid drop target?

We see that in both cases, drag element is moved to the drag proxy's last known coordinates, which mimics the behavior of the DD class without the valid or invalid drop behavior constraints.

Lets add those next. This will be another long listing, but will wrap up our DDProxy implementation.

### **Listing 13.7 Adding the valid or invalid drop behaviors.**

```

onDragDrop : function(evtObj, targetElId) { // 1
    var dragEl = Ext.get(this.getEl());
    var dropEl = Ext.get(targetElId);

    if (dragEl.dom.parentNode.id != targetElId) {
        dropEl.appendChild(dragEl);
        this.onDragOut(evtObj, targetElId);
        dragEl.dom.style.position = '';
    }
    else {
        this.onInvalidDrop();
    }
},
b4EndDrag : Ext.emptyFn, // 2
endDrag : function() { // 3
    var dragProxy = Ext.get(this.getDragEl());

    if (this.invalidDrop === true) { // 4
        var dragEl = Ext.get(this.getEl());

        var animCfgObj = {
            easing : 'easeOut',
            duration : .25,
    
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        callback : function() {
            dragProxy.hide();
            dragEl.highlight();
        }
    };
    dragProxy.moveTo(this.originalXY[0], this.originalXY[1], animCfgObj);
}
else {
    dragProxy.hide(); // 5
}
delete this.invalidDrop;
}

{1} Override the onDragDrop method
{2} Prevent the proxy from hiding before drag ends
{3} Override the endDrag method
{4} Execute the repair animation if this was an invalid drop
{5} Hide the drag proxy if it was a valid drop
}

```

In listing 13.7 we finish up the rest of our DDPProxy implementation by adding the onDragDrop, b4EndDrag and endDrag overrides.

The onDragDrop{1} method is exactly the same as the DD implementation, where if the drop element is not the same as the drag element's parent, we allow the drop to occur, moving the node to the drop element. Else, we call the onInvalidDrop method, which sets the invalidDrop property to true.

The b4EndDrag{2} method is an intentional override using the Ext.emptyFn (empty function) reference. We do this because the DDPProxy's b4EndDrag method will actually hide the drag proxy before the endDrag method is called, which conflicts with the animation that we want to perform. And since it would be wasteful to allow the drag proxy to be hidden and then show it, we simply prevent it from hiding by overriding b4EndDrag with a function that does nothing.

Just like in the DD implementation earlier, the endDrag{3} method is tasked with doing the repair if the invalidDrop property is set to true{4}. But, instead of animating the drag element itself, it animates the drag proxy. The animation uses the easeOut easing to allow for a smoother animation finish. The callback will hide the drag proxy and then call the highlight effect method of the drag element, animating the background from yellow to white.

Lastly, if the endDrag was called with the invalidDrop property not set, it simply hides{5} the proxy element from view, completing our DDPProxy implementation.

As we've just seen, implementing the full gamut of drag and drop with generic DOM elements requires some work and understanding of basics of the drag and drop class hierarchy. The reward from that effort, of course, is a really cool way to drag and drop elements across the screen, adding that extra bit of functionality for our users.

## 13.5 Summary

In this chapter we explored the basics of drag and drop and implemented two types of drag behaviors with DOM nodes, which paves the way for you to learn to implement this behavior with the UI widgets.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Along the way, we explored the very basics of drag and drop and took some time discussing its behavior model. We took a good look at the drag and drop mini-framework and learned that the classes can be grouped into two major categories, drag and drop. While discussing the class hierarchy we learned what each class is designed for.

Afterwards, we learned what it takes to implement drag and drop with DOM nodes with constraints using drag groups. We discussed some of the common methods that need to be overridden to make this possible. We got to play with the cool "elastic" easing when developing the repair of the drag element.

In the last stretch, we took our basic drag and drop implementation and modified it to use drag proxies and learned how to properly code a repair operation, which differs from the repair for the DD repair.

Next, we'll exercise our basic knowledge of drag and drop and learn to implement drag and drop with three UI widgets, DataView, GridPanel and TreePanel, where you'll learn of the different vectors for implementing this behavior.

# 14

## *Drag and Drop with widgets*

When developing your projects, it's easy to produce context menus and buttons to perform certain actions, such as moving a record from one grid to another. But users always crave an easier way to perform those actions. Providing drag and drop behaviors to your applications will allow your users to accomplish these tasks much more effectively. The less clicking of a mouse a user has to do, the happier they ultimately are.

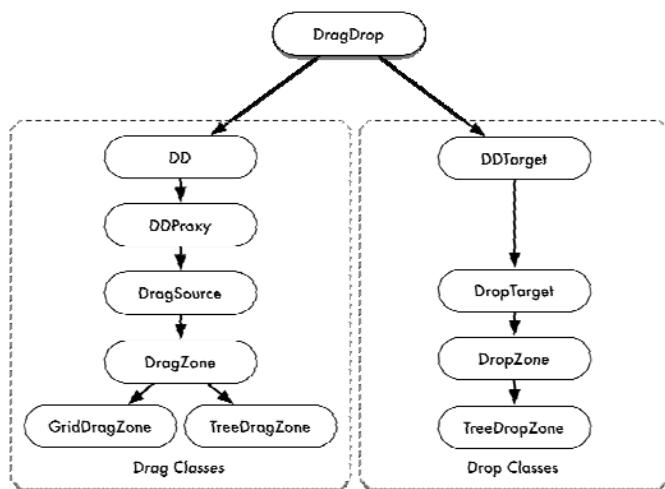
In this chapter we'll explore applying this behavior to the three widgets that drag and drop is commonly used with, which are the DataView, GridPanel and TreePanel. As we do that, we'll descend farther into the DragDrop class hierarchy and touch upon the DragZone and DropZone classes and their descendants.

We'll learn about each of the different implementation patterns that are required to get drag and drop working with each of these widgets and take some time discussing some of the hidden tricks and inner workings of the drag and drop classes.

Because the nature of this behavior is complex, this will be a very code-heavy chapter.

### **14.1 A quick review of the Drag and Drop classes**

Before we actually get down to implementing drag and drop with widgets, we have a quick review the DragDrop class hierarchy. This will give us a good picture of the road ahead and help us understand what classes we'll use and why.



**Figure 14.1** The DragDrop class hierarchy

In the last chapter, we covered use of the DD and DDProxy drag classes and the DDTTarget drop class. Recall that we utilized those classes because they were designed for single node drag and drop applications, which met our requirements at that time.

Each of the widgets we're going to use to employ drag and drop with each have selection models that allow for multiple node selections. Knowing this means that we're going to have to leverage the classes designed for handling that. The DragSource and DropTarget classes are technically designed to handle this type of drag and drop behavior. But, we're going to start with the DragZone and DropZone classes. Here's why.

While the DragSource has all of the necessary mechanics to handle multi-node drag operations, the DragZone class goes a few steps further and adds useful features like managing the scroll bar of a container if the container is scrollable. We'll see this in action in when we employ drag and drop with DataViews in just a bit.

The DropZone class adds features to the DropTarget class, such as being able to track the nodes that the drag element is currently hovering over, which makes it very easy to allow for precision drop operations of a drag node to specific indexes. We'll get a chance to exercise this when we learn how to employ drag and drop with GridPanels a little later on.

OK, we know now more about our starting point. Because the application of drag and drop with DataViews provides key and essential knowledge for the GridPanel and TreePanel implementation, we'll begin with it.

## 14.2 Drag and Drop with DataViews

We've been tasked to develop something that will allow managers to track employees who are on staff or on vacation using simple drag and drop gestures. We'll construct two DataViews, both of which are similar to the ones we constructed earlier. To use it, we'll make some slight modifications, which will include enabling multiple node selection.

Here's what the two DataViews will look like encapsulated in an instance of Ext.Window.



Figure 14.2 The outcome of our DataView construction.

Now that we know what we're going to be building, lets begin.

### 14.2.1 Constructing the DataViews

We'll start by creating the CSS required to style the elements within the DataView. The drag and drop related CSS will be included so we get it out of the way.

#### Listing 14.1 Setting up the CSS for our DataViews

```
<style type="text/css">
    .emplWrap {
        border: 1px #999999 solid; /* 1 */
        -moz-border-radius: 5px;
        -webkit-border-radius: 5px;
        margin : 3px;
        padding : 3px;
        background-color: #ffffcc;
    }

    .emplOver {
        border: 1px #9999ff solid; /* 2 */
        background-color: #ccccff;
        cursor: pointer;
    }

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        }

        .emplSelected {
            border: 1px #66ff66 solid;
            background-color: #ccffcc;
            cursor: pointer;
        }

        .emplName {
            font-weight: bold;
            margin-left: 5px;
            font-size: 14px;
            text-decoration: underline;
            color: #333333;
        }

        .emplAddress {
            margin-left: 20px;
        }
    
```

</style>

{1} Style the entire employee template div  
{2} Style the employee elements on mouse over  
{3} Configure the selected employee style

In the above CSS, we style how each employee div will look in the DataViews. An unselected employee element will have a yellow background{1}, similar to that of a manila folder. When the mouse hovers over an employee, it will use the emplOver{2} CSS class to style it blue. When selected, the employee will be colored green using the emplSelected{3} CSS class.

OK, we have the CSS in place for our future DataViews to use. Next, we'll configure the two stores that will be consumed by the different DataViews.

#### **Listing 14.2 Configuring the stores for the DataViews.**

```

var storeFields = [
    { name : "id",      mapping : "id" },
    { name : "department", mapping : "department" },
    { name : "email",     mapping : "email" },
    { name : "firstname", mapping : "firstname" },
    { name : "lastname",   mapping : "lastname" }
];

var onStaffStore = { // 1
    xtype : 'jsonstore',
    autoLoad : true,
    proxy : new Ext.data.ScriptTagProxy({
        url : 'http://extjsinaction.com/examples/chapter12/getEmployees.php'
    }),
    fields : storeFields,
    sortInfo : {
        field : 'lastname',
        direction : 'ASC'
    }
};

var onVactaionStore = { // 2
    xtype : 'jsonstore',
    fields : storeFields,
    autoLoad : false,
}

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        sortInfo : {
            field   : 'lastname',
            direction : 'ASC'
        }
    };
{1} Create a list of store fields
{2} Configure a remote JSON Store
{3} Create a configuration object for a local JSON Store

```

In listing 14.11, we create a list of fields{1} and two configuration objects for JsonStores. The first JsonStore{2} uses a ScriptTagProxy to fetch the list of employees, while the second JsonStore{3} will sit quietly, waiting for records to be inserted upon drop gesture.

Now that we have the data stores configured, we can create the DataViews.

### **Listing 14.3 Constructing the two DataViews.**

```

var dvTpl = new Ext.XTemplate( // 1
    '<tpl for=".">' ,
        '<div class="emplWrap" id="employee {id}">',
            '<div class="emplName">{lastname}, {firstname}</div>',
            '<div><span class="title">Department:</span> {department}</div>',
            '<div>',
                '<span class="title">Email:</span><a href="#">{email}</a>',
            '</div>',
        '</div>',
    '</tpl>'
);

var onStaffDV = new Ext.DataView({ // 2
    tpl      : dvTpl,
    store    : onStaffStore,
    loadingText : 'loading...',
    multiSelect : true,
    overClass : 'emplOver',
    selectedClass : 'emplSelected',
    itemSelector : 'div.emplWrap',
    emptyText : 'No employees on staff.',
    style     : 'overflow:auto; background-color: #FFFFFF;'
});

var onVactionDV = new Ext.DataView({ // 3
    tpl      : dvTpl,
    store    : onVactionStore,
    loadingText : 'loading...',
    multiSelect : true,
    overClass : 'emplOver',
    selectedClass : 'emplSelected',
    itemSelector : 'div.emplWrap',
    emptyText : 'No employees on vacation',
    style     : 'overflow:auto; background-color: #FFFFFF;'
});

{1} Construct the XTemplate used by both DataViews.
{2} Create the DataView for the employees that are on staff.
{3} Create the DataView for the employees that will be on vacation.

```

In the above listing, we configure and construct the two DataViews starting with a common XTemplate instance{1}. The onStaffDV{2} will consume the data from the onStaffStore

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

to load the list of employees currently on staff, while the `onVacationDV{3}` will use the unpopulated `onVacationStore`.

We could render the DataViews on screen, but I think they would look better inside of a Window and stand side by side with an HBox layout.

#### **Listing 14.4 Placing the DataViews inside of an Window.**

```
new Ext.Window({  
    layout      : 'hbox',  
    height     : 400,  
    width      : 550,  
    border     : false,  
    layoutConfig: { align : 'stretch' },  
    items       : [  
        {  
            title   : 'Employees on staff',  
            frame   : true,  
            layout   : 'fit',  
            items   : onStaffDV,           // 1  
            flex    : 1  
        },  
        {  
            title   : 'Employees on vacation',  
            frame   : true,  
            layout   : 'fit',  
            id      : "test",  
            items   : onVactionDV,         // 2  
            flex    : 1  
        }  
    ]  
}).show();  
{1} Instantiating a Window for the DataViews to live in  
{2} Place the DataViews inside of Panels
```

In Listing 14.4 we create an instance of `Ext.Window{1}` that utilizes the `HBoxLayout` to place two panels side by side with an equal width and their height stretched to the Window's body. The panel on the left will contain the on staff `DataView{2}` while the panel on the right will contain the `DataView` for vacationers.

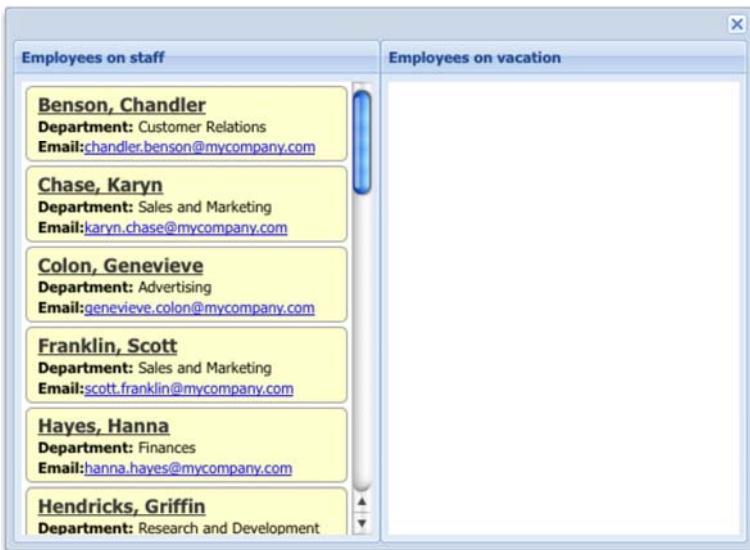


Figure 14.3 Our rendered DataViews inside of an Ext.Window.

OK we can see that our DataViews have rendered properly with the on staff employees appearing on the left and no one currently on vacation. With that, we've set the stage for us to apply drag and drop.

#### 14.2.2 Adding Drag gestures

The application of drag and drop with DataViews arguably requires the most effort when compared to the GridPanel and TreePanel. This is because unlike those widgets, the DataView class does not have its own DragZone implementation subclass for us to build upon, which means that we'll have to craft our own implementation of DragZone. Likewise, we'll have to develop and craft an implementation of DropZone to manage the drop gestures.

The DragZone class uses a special proxy known as a StatusProxy, which will use icons to indicate whether or not a successful drop is possible. Here is what they typically look like.



Figure 14.4 The StatusProxy in action, indicating that a drop is possible (left) or not (right).

The status proxy, by default is extremely lightweight and very efficient, but somewhat boring. While they provide useful information, they are far from fun to use. So we'll take

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

advantage of the ability to customize the StatusProxy look a little to spice up the drag gestures and make them much more enjoyable and informational. Another feature that DragZone adds is automated repair of an invalid drop scenarios, which reduces the amount of code that we need to generate to get this stuff working.

We'll begin by creating the overrides that will be applied to the instance of DragZone that we'll create afterwards. Because the DataViews *must* be rendered in order to have drag and drop applied, the following code shall be inserted below Listing 14.4.

### **Listing 14.5 Creating the DragZone overrides**

```
var dragZoneOverrides = {
    containerScroll : true, // 1
    scroll : false, // 2
    getDragData : function(evtObj) {
        var dataView = this.dataView; // 3
        var sourceEl = evtObj.getTarget(dataView.itemSelector, 10); // 4

        if (sourceEl) {
            var selectedNodes = dataView.getSelectedNodes();
            var dragDropEl = document.createElement('div');

            if (selectedNodes.length < 1) {
                selectedNodes.push(sourceEl);
            }

            Ext.each(selectedNodes, function(node) {
                dragDropEl.appendChild(node.cloneNode(true));
            });
            return {
                ddel : dragDropEl,
                repairXY : Ext.fly(sourceEl).getXY(),
                dragRecords : dataView.getSelectedRecords(),
                sourceDataView : dataView
            };
        }
    },
    getRepairXY: function() {
        return this.dragData.repairXY;
    }
};

{1} Automatically scroll the destination container
{2} Prevent the document.body from scrolling.
{3} Override the getDragData method
{4} Cache the element that the drag gesture was initiated with
{5} Create and return a drag data object
```

In the above listing, we create the override properties and methods that will be applied to the future instances of DragZone. Even though the amount of code is relatively short, there is a lot going on that we need to be aware of. Here's how all of this works.

Initially, we set two configuration properties that help manage scrolling when a drag operation is under way. The first is `containerScroll{1}`, which is set to true. Setting this property to true will instruct the DragZone to call `Ext.dd.ScrollManager.register`, which will

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

help manage scrolling of a DataView when scrolling operations are in effect. We'll look at this in detail when we look at the DataView after the application of DragZone.

The next property, scroll{2}, is set to false. Setting this to false prevents the document.body element from scrolling when the drag proxy is moved out of the browser's viewport. Keeping the browser canvas fixed during drag and drop operations will increase its effectiveness.

Next, we override getDragData{3}, which is an extremely important method to the multi-node drag and drop application. The purpose of getDragData is to construct what is known as the "drag data" object as we'll see returned towards the end of this method. It is important to note that the drag data object that will be generated and returned by the getDragData method will be cached on the instance of the dropZone and can be accessed via the this.dragData reference. We'll see this in action in the getRepairXY method later on.

In this method, we first set a reference to the element that the drag gesture was initiated with{4}, sourceEl. We will use it later on to update the StatusProxy if the number of selected nodes the DataView thinks it has is wrong. We also create a container element, dragDropEl that will be used to contain copies of the selected nodes during drag and placed in the StatusProxy.

#### **NOTE**

The presence of sourceEl is tested in order for the rest of the method to continue. getDragData is called upon the "mouse down" event of the element that is registered with the DragZone. This means that getDragData will be called even if the DataView element itself is clicked instead of a record element, which would cause the method to fail.

Next, we interrogate the number of items the DataView thinks it has selected during the time of the drag operation. If the number of selectedNodes{5} is less than 1, then we simply append the element that the drag gesture was started with. We do this because sometimes a drag gesture is initiated before the DataView can register an element as visually selected. This is a quick fix to this odd behavior.

We then use Ext.each{6} to loop through the selectedNodes list, appending it to the dragDropEl. This will help customize the StatusProxy and give the appearance that the user is dragging over a copy of the selected node(s).

In the last chunk of this override, we return an object that will be used on to update the StatusProxy and any drop operations. The only required property that is to be passed in this object is ddel, which is what will be placed inside of the StatusProxy.

For this implementation, we added a few other useful properties to the custom drag data object. First is repairXY, which is an array of the X and Y coordinates of the element that the drag gesture was initiated. This will be used later on to help the invalid drop repair operation.

Also included is dragRecords, which contains a list of instances of Ext.data.Record for each of the nodes selected and being dragged. Lastly, we set source DataView as the ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

reference of the DataView for which this DragZone is being used. Both dragRecords and sourceDataView properties will help the application of DropZone to remove the dropped records from the source DataView.

The last method in the list of overrides is the getRepairXY, which simply returns the locally cached data object's repairXY property and helps the repair operation know where to animate the StatusProxy on an invalid drop.

OK, we have our overrides set, it's now time to instantiate instances of DragZone and apply them to the DataViews.

#### **Listing 14.6 Applying DragZone to the DataViews**

```
var onStaffDragZoneCfg = Ext.apply({}, {
    ddGroup      : 'employeeDD',
    dataView     : onStaffDV
}, dragZoneOverrides); // 1

new Ext.dd.DragZone(onStaffDV.getEl(), onStaffDragZoneCfg); // 2

var vacationDragZoneCfg = Ext.apply({}, {
    ddGroup      : 'employeeDD',
    dataView     : onVactionDV
}, dragZoneOverrides);

new Ext.dd.DragZone(onVactionDV.getEl(), vacationDragZoneCfg);
{1} Create a custom copy of the dragZoneOverrides
{2} Instantiate a new instance of DragZone
```

In the above listing, we use Ext.apply to create a custom copy of the dragZoneOverrides object for the employment of DragZone targeted towards the on staff DataView**{1}**. The custom copy of the overrides will include a ddGroup property. Both DragZone implementations will share this. What makes each copy special is the dataView property, which references the DataView that is attached to the DragZone and is used by the getDragData method we created earlier. The same pattern is used to setup the DragZone for the vacation DataView.

One thing you may notice is that unlike the implementation of DDTarget in the last chapter, we don't *apply* the overrides to the instance of DragZone. This is because DragZone's superclass, DragSource, actually takes of that for us automatically, just like Ext.Component does.

Refreshing our project page will allow us to exercise drag operations. We can also see our customized StatusProxy in action.



Figure 14.5 DragZone with a custom DragProxy in action.

We can see that selecting and dragging one or more records in the on staff DataView reveals the StatusProxy with the copies of the selected nodes, which makes the drag operation nicer and much more fun to use.

We could also see the `getRepairXY` method in action by dropping the drag proxy anywhere on the page. The animation will make the drag proxy slide towards the XY coordinates of the element that the drag operation was initiated.

To exercise the ScrollMgr's scroll management capabilities, you will need to initiate a drag gesture and hover the mouse in what I like to call the "auto-scroll zones" as illustrated below.

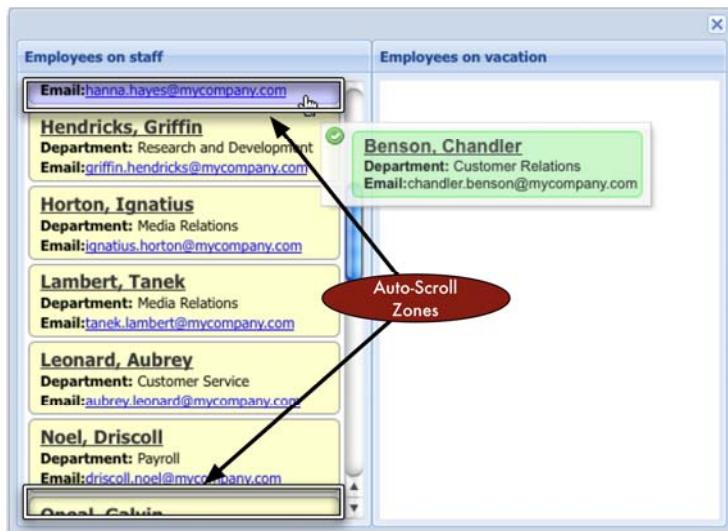


Figure 14.6 The areas where automatic scrolling will take place

As illustrated in figure 14.6, hovering the mouse in the zones highlighted by the boxes will result in the DataView automatically scrolling until it can no longer move.

You probably have already noticed that when we drag the nodes above the vacation DataView, the StatusProxy shows an icon indicating that the drop will not be successful. This is because we have not employed a DropZone, which is what we're going to do next.

### 14.2.3 Applying Drop

Just like our previous drag and drop applications, we must register a drop target of sorts for the drag classes to interact with. As we discussed before, we'll use the DropZone class. Following the pattern before this, we'll create an overrides object, which will handle the drop gestures and is much easier to implement relative to drag gestures.

#### **Listing 14.7 Creating the DropZone overrides**

```
var dropZoneOverrides = {
    onContainerOver : function() { // 1
        return this.dropAllowed;
    },
    onContainerDrop : function(dropZone, evtObj, dragData) {
        var dragRecords = dragData.dragRecords;
        var store = this.dataView.store;

        var dupFound = false;
        Ext.each(dragRecords, function(record) { // 2
            if (store.getAt(record.id)) {
                dupFound = true;
            }
        });
        if (!dupFound) {
            store.add(dragRecords);
        }
    }
};
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        var found = store.findBy(function(r) {
            return r.data.id === record.data.id;
        });

        if (found > -1 ) {
            dupFound = true;
        }
    });

    if (dupFound !== true) {
        Ext.each(dragRecords, function(record) {
            dragData.source DataView.store.remove(record); // 3
        });
        this.dataView.store.add(dragRecords); // 4
        this.dataView.store.sort('lastname', 'ASC'); // 5
    }
    return true; // 6
};

[1] Update the StatusProxy to indicate allowable drops
[2] Search for duplicate records
[3] Remove all records from the source DataView store
[4] Add the records to the destination DataView store
[5] Sort the records by last name, ascending
[6] Indicate successful drop, preventing repair animation

```

In the above listing, we create an override object with two methods, to enable drop gestures to successfully occur with the two DataViews. The first is the `onContainerOver{1}`, which is used to determine whether or not the drop should be allowed. In this application, there is no processing needed, but we need to, at the very least, return the `this.droppedAllowed` reference, which is a reference to the CSS class "x-dd-drop-ok" that provides the green check icon. If you wanted to use a custom icon, this is where you would return a custom CSS class.

The next method, `onContainerDrop` is where we will process the dropped nodes and will be called by the instance of `DragZone` when the mouse up event fires. Remember that `DragZone` won't interact with the `DropZone` if they are not both participating in the same drag drop group.

In this method, we use the `dragData` object that we created in our `DragZone` `getDragData` override. A local reference to the selected records (`dragRecords`) and the destination DataView's store (`store`) are created for later utilization.

Next, `onContainerDrop` will search for duplicate records`{2}`. This is useful if you're attempting a copy instead of a move. If no duplicates are found, `Ext.each` is used to loop through the drag records to remove them from the source DataView's store`{3}`. The records are then added`{4}` to the destination DataView's store and sorted`{5}` by last name in ascending order.

After all of the record management has taken place, the `onContainerDrop` returns the boolean value `true`. By returning `true`, convince the `DragZone` that the drop was successful, and it does not initiate a repair animation. Any other value will indicate that the drop was unsuccessful, thus a repair would occur.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Now that we have the overrides in place, it's time to apply them to the DataViews.

### **Listing 14.8 Creating the DropZone overrides**

```
var onStaffDropZoneCfg = Ext.apply({}, {
    ddGroup      : 'employeeDD',
    dataView     : onStaffDV
}, dropZoneOverrides);

new Ext.dd.DropZone(onStaffDV.ownerCt.el, onStaffDropZoneCfg);

var onVacationDropZoneCfg = Ext.apply({}, {
    ddGroup      : 'employeeDD',
    dataView     : onVactionDV
}, dropZoneOverrides);

new Ext.dd.DropZone(onVactionDV.ownerCt.el, onVacationDropZoneCfg);
```

In the above listing, we create custom copies of the dropZoneOverrides object for the implementation of DropZone for each of the DataViews and follows the exact same pattern that we used in listing 14.6, where we created instances of DragZone.

We can now see our end-to-end drag drop application in action. Lets refresh our page and attempt a drag operation from the on staff DataView to the vacation DataView.



Figure 14.7 The StatusProxy now shows that a drop gesture can occur on the drop zone.

Dragging nodes from the employee DataView to the vacation DataView produces a StatusProxy that contains a green checkmark a drop invitation. Dropping the nodes will invoke the onContainerDrop method, moving the records from left to right.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

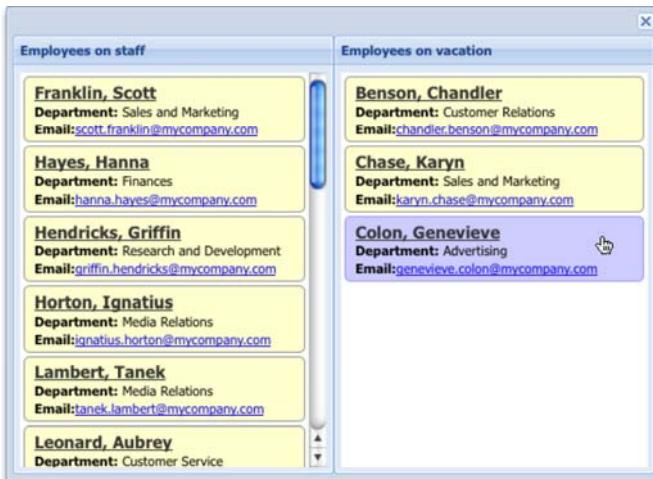


Figure 14.8 We've successfully dropped three records from the left DataView to the right.

There you have it, drag and drop from one DataView to another with a good looking StatusProxy. Because each DataView have their own attached instance of DragZone and DropZone, you can drag and drop items from one to the other and the records will automatically be sorted by last name.

We just finished learning how to apply drag and drop to two DataViews and learned that it we were responsible to employ the full end-to-end code for both gestures. Next, we'll dive into the world of Drag and Drop with GridPanels, where we'll learn that the implementation pattern changes compared to that of the DataView.

### 14.3 Drag and Drop with GridPanels

We've just been approached by management to create something that will allow them to track whether departments need computer upgrades or not. They want to be able to somehow flag the departments that require an upgrade and change the order of the departments that will get upgraded.

To get the job done, we'll use two GridPanels side by side, just like we did with the DataViews a short time ago. We'll put drag and drop into practice from GridPanel to GridPanel and allow for the reordering of departments in the list.

In this exercise, we'll learn that the application of drag and drop between two GridPanels is much simpler compared to that of the DataView. This is mainly due to the fact that GridPanels have their own DragZone subclass called GridDragZone, which takes care of half of the work for us. We are left to simply apply a DropZone and attach it to a good target element within the GridPanel.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

In addition to exploring how to setup a DropZone with the GridPanel, we're going to tackle one of the largest challenges that developers face, which is how to properly allow for drop gestures with the ability to select the index for which the item is to be dropped. This includes the ability for the GridPanel to allow for reordering self drag and drop gestures.

We'll start by constructing two GridPanels that will live in a Window. The Window will manage the GridPanel dimensions by means of the HBox layout and is the same paradigm as the exercise we completed just a short time ago.

### 14.3.1 Constructing the GridPanels

By now you should be very comfortable with creating GridPanels and configuring their supported classes. In order to keep things moving, we'll be picking up the pace a little.

#### **Listing 14.9 Creating the first GridPanel**

```
var remoteProxy = new Ext.data.ScriptTagProxy({ // 1
    url : 'http://extjsinaction.com/examples/chapter12/getPCStats.php'
});

var remoteJsonStore = { // 2
    xtype      : 'jsonstore',
    proxy      : remoteProxy,
    id         : 'ourRemoteStore',
    root       : '',
    autoLoad   : true,
    totalProperty : 'totalCount',
    fields     : [
        { name : 'department',      mapping : 'department' },
        { name : 'workstationCount', mapping : 'workstationCount' }
    ]
};

var depsComputersOK = new Ext.grid.GridPanel({ // 3
    title      : 'Departments with good computers',
    store      : remoteJsonStore,
    loadMask   : true,
    stripeRows : true,
    autoExpandColumn : 'department',
    columns    : [
        {
            header    : 'Department Name',
            dataIndex: 'department',
            id       : 'department'
        },
        {
            header    : '# PCs',
            dataIndex: 'workstationCount',
            width    : 40
        }
    ]
});
```

{1} Creating a remote ScriptTagProxy  
{2} Configuring the remote JSON store  
{3} Instantiating the first GridPanel.

In listing 14.9, we create a remote ScriptTagProxy`{1}`, which is utilized by the JsonStore`{2}` and will fetch the data from extjsinaction.com. Next, a GridPanel is instantiated`{3}`, which will be uses the previously created remoteJsonStore to display the departments.

Next, we'll create the second GridPanel, which will be used to list the departments in need of an upgrade.

#### **Listing 14.10 Creating the second GridPanel**

```
var needUpgradeStore = Ext.apply({}, { // 1
    proxy : null,
    autoLoad : false
}, remoteJsonStore);

var needUpgradeGrid = new Ext.grid.GridPanel({
    title : 'Departments that need upgrades',
    store : needUpgradeStore,
    loadMask : true,
    stripeRows : true,
    autoExpandColumn : 'department',
    columns : [
        {
            header : 'Department Name',
            dataIndex : 'department',
            id : 'department'
        },
        {
            header : '# PCs',
            dataIndex : 'workstationCount',
            width : 40
        }
    ]
});
```

`{1} Create a modified copy of the the remoteJsonStore`  
`{2} Configure the second GridPanel`

In the listing above, we use Ext.apply to create a modified copy of the remoteJsonStore, which overrides the proxy parameter and sets autoLoad to false. This allows us create a near duplicate of the remoteJsonStore but reuse most of the same properties, such as the fields.

Next, we create the second instance of GridPanel for the departments that need to be upgraded. These GridPanels need a home. Lets create an Ext Window to display these in.

#### **Listing 14.11 Giving the GridPanels a home.**

```
new Ext.Window({
    width : 500,
    height : 300,
    layout : 'hbox',
    border : false,
    defaults : {
        frame : true,
        flex : 1
    },
    layoutConfig : {
        align : 'stretch'
    },
});
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

    items      : [
        depsComputersOK,
        needUpgradeGrid
    ]
}) .show();

```

Above, we create an Ext Window, which leverages the HBox layout to manage the two GridPanels that we configured earlier. Lets take them out for a test drive.



Figure 14.9 The two department GridPanels side by side.

As illustrated in Figure 14.9, the two GridPanels render perfectly in the Window as configured. Our next task is to configure them to enabledrag and drop.

### 14.3.3 Enabling drag

To configure a GridPanel to enable drag gestures, all we need to do is add two properties to the both of the GridPanel configuration objects as such:

```

enableDragDrop : true,
ddGroup       : 'depGridDD',

```

The reason that we can easily enable drag gestures with GridPanel is because the GridView actually tests for the presence of the enableDragDrop (or enableDrag) in the GridPanel when it renders its UI. If the property is set, then it will create an instance of GridDragZone and use the grid's ddGroup (drag drop group) if present or a generic "GridDD" group.

When configuring drag and drop with GridPanels, I always like to specify a drag drop group specific to the GridPanels that are to interact with each other. If I didn't then every

single GridPanel with drag enabled would interact with one another, which could cause undesired effects and possibly lead to headaches down the road.

Lets refresh the GridPanel and see the drag gestures in action.

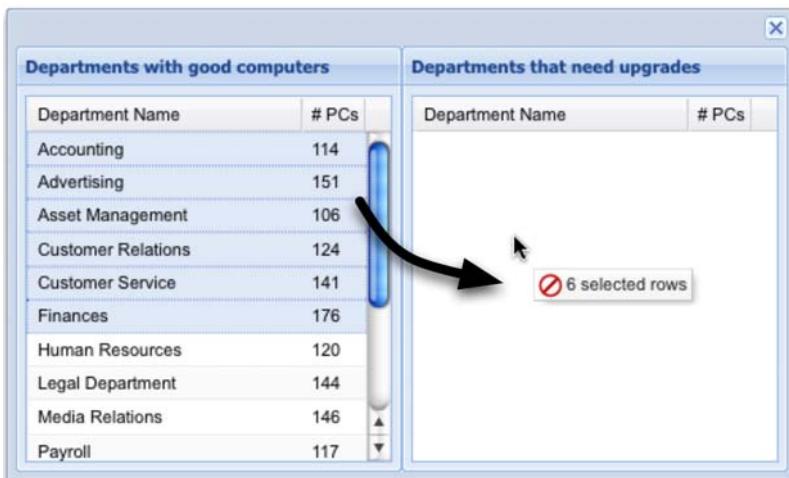


Figure 10.10 Drag gestures enabled in the GridPanel.

When attempting a drag gesture with the GridPanel on the left, we can see the StatusProxy appear with the number of Rows selected. This is how the GridDragZone uses the getDragData method, where it simply displays the number of selected rows for the ddel property of the drag data object. Sound familiar?

We see that the status proxy displays an unfavorable drop icon. This is because have not created a DropZone for the DragZone to interact with. Before we will do that, we'll add some CSS styles that the DropZone will use to provide better drop Invitation.

#### 14.3.4 Better drop invitation

While the StatusProxy provides enough information to inform the user that a successful drop is possible, it does not provide feedback as to what index the drop operation will take place, which is absolutely crucial to allow for the reordering of records.

We'll need some CSS rules to help us with this. Please add the following styles to the head of the page.

##### **Listing 14.12 Adding some CSS styles to provide better drop invitation information.**

```
<style type="text/css">
    .gridBodyNotifyOver {                                     /* 1 */
        border-color: #00cc33 !important;
    }
    .gridRowInsertBottomLine {                                /* 2 */
        border-bottom:1px dashed #00cc33;
    }
</style>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        }
        .gridRowInsertTopLine {                                     /* 3 */
            border-top:1px dashed #00cc33;
        }
    </style>
    {1} Change the GridPanel's body border to green
    {2} Add a dashed bottom border to a row
    {3} Add a dashed top border to a row
}

```

Adding the above CSS to the head of the document will enable us to provide much better drop invitation to the user, which will allow them to accurately pinpoint which GridPanel they are going to be dropping on and what row index they will inserting records in.

Now that we have that out of the way, we can begin the construction of the custom DropZone for our GridPanels.

#### 14.3.5 Adding drop

Just like the DataView drag and drop application, we can only instantiate instances of DropZone for the GridPanels after they are rendered. To simplify this process, we'll add the following code after the instantiation of Ext.Window.

We'll start by overriding the onContainerOver method, which will handle the necessary mouse movements to track what row the dragged records are to be inserted. The code to get the job done is pretty intense, but it the results are well worth the work.

#### Listing 14.13 Creating the Overrides Object.

```

var dropZoneOverrides = {
    ddGroup : 'depGridDD',
    onContainerOver : function(ddSrc, evtObj, ddData) {
        var destGrid = this.grid;
        var tgtEl = evtObj.getTarget();
        var tgtIndex = destGrid.getView().findRowIndex(tgtEl);           // 1
        this.clearDDStyles();

        if (typeof tgtIndex === 'number') {                                // 2
            var tgtRow = destGrid.getView().getRow(tgtIndex);
            var tgtRowEl = Ext.get(tgtRow);
            var tgtRowHeight = tgtRowEl.getHeight();
            var tgtRowTop = tgtRowEl.getY();
            var tgtRowCtr = tgtRowTop + Math.floor(tgtRowHeight / 2);
            var mouseY = evtObj.getXY()[1];

            if (mouseY >= tgtRowCtr) {                                     // 3
                this.point = 'below';
                tgtIndex++;
                tgtRowEl.addClass('gridRowInsertBottomLine');
                tgtRowEl.removeClass('gridRowInsertTopLine');
            } else if (mouseY < tgtRowCtr) {                               // 4
                this.point = 'above';
                tgtRowEl.addClass('gridRowInsertTopLine');
                tgtRowEl.removeClass('gridRowInsertBottomLine')
            }
            this.overRow = tgtRowEl;
        }
        else {
            tgtIndex = destGrid.store.getCount();                         // 5
        }
    }
}

```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        }
        this.tgtIndex = tgtIndex;
        destGrid.body.addClass('gridBodyNotifyOver'); // 6
        return this.dropAllowed;
    },
    notifyOut      : function() {},
    clearDDStyles : function() {},
    onContainerDrop : function() {}
};

{1} Get the index of the element that we're hovering over.
{2} Is this a row?
{3} Is the mouse Above the row?
{4} Or is it below the row?
{5} Append to the store if no row is being hovered over
{6} Highlight the GridPanel's body element with a green border

```

In the above listing, we create a dropZoneOverrides configuration object that contains the ddGroup property, onContainerOver implementation and three stub methods that we'll fill in later. For now, we'll focus on the onContainerOver method. Here's how it works.

When the DragZone detects that the drag gesture is hovering over a like-group participating DropZone, it calls the DropZone's notifyOver method, which calls onContainerOver. This occurs for each X and Y movement of the mouse while the mouse hovering over the DropZone element, which makes it perfect for detecting if the drop gesture will result in an append, insert or reorder operation.

In our onContainerOver implementation, we get the element that the mouse is hovering over (tgtDiv) and ask the GridView to find its index{1}. If findRowIndex returns a number, then we know that the mouse is hovering over a row that the GridView knows of, so the calculations to determine the insertion index can begin{2}.

To determine the exact coordinate of the target row, we must first get a reference to the row and wrap it with Ext.Element. From there, we use the helper methods to get the height and current coordinate. Next, we determine the center of the row, and compare it to the mouse's Y position.

If the mouse's Y position is greater or equal to the exact center of the target row's height, then we know that the record will be inserted *after* the target row {3}. We set a local this.point property, increase the targetIndex value by one and add the 'gridRowInsertBottomLine' CSS class to the target row, providing the proper drop invitation to the user.

If the mouse's Y position is less than the target row's center, then we know that the drop gesture will be an insert *above* the row {4}. We set the this.point property accordingly and add the 'gridRowInsertTopLine' CSS class to the row, which makes the top border green and dashed, providing the proper insertion index drop invitation.

When dragging over a the actual DropZone element, the target index is set to the destination GridPanel's record count{5}, which ensures that anything dropped is appended to the store.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

Next, the target index is cached locally and the destination GridPanel's body element has its border colored green, which assists with the drop invitation. Lastly, just like the DataView DropZone application, we return the this.dropAllowed reference to ensure the StatusProxy provides the correct drop invitation icon.

The last bit of this listing consists of three method stubs, notifyOut, clearDDStyles and onContainerDrop, which will handle the clearing of the added drop invitation and handle the drop gesture. Lets fill in those method stubs with some functionality.

#### **Listing 14.15 Creating the Overrides Object.**

```

notifyOut : function() { // 1
    this.clearDDStyles();
},
clearDDStyles : function() { // 2
    this.grid.body.removeClass('gridBodyNotifyOver');
    if (this.overRow) {
        this.overRow.removeClass('gridRowInsertBottomLine');
        this.overRow.removeClass('gridRowInsertTopLine');
    }
},
onContainerDrop : function(ddSrc, evtObj, ddData) { // 3
    var grid      = this.grid;
    var srcGrid   = ddSrc.view.grid;
    var destStore = grid.store;
    var tgtIndex  = this.tgtIndex;
    var records   = ddSrc.dragData.selections;

    this.clearDDStyles();

    var srcGridStore = srcGrid.store;
    Ext.each(records, srcGridStore.remove, srcGridStore); // 4

    if (tgtIndex > destStore.getCount()) {
        tgtIndex = destStore.getCount();
    }
    destStore.insert(tgtIndex, records); // 5

    return true;
}
{1} Clear drop invitation upon drag out
{2} Remove all drop invitation CSS
{3} Complete the drop logic
{4} Remove all records from the source grid store
{5} Insert the dropped records at the target index

```

In finalizing the dropZoneOverrides object, we add the clearing of the drop invitation that will appear on the destination GridPanel and rows (if applicable), and completed the code to move the records from the source GridPanel to the destination. Here's how all of this works.

The notifyOut**{1}** override is extremely simplistic and calls the clearDDStyles method is defined underneath it. Remember that notifyOut is called when the dragged object is pulled away from the DropZone, which means we need to remove drop invitation.

Next is the clearDDStyles**{2}** method, which is custom and not an template method that is required to get drag and drop working. The reason we added it is because the onContainerOver method will add styles to the target row and target GridPanel body, which

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

means we need a way to clear them and it is always best to move commonly reused code to a separate method. We just saw that this method is being called by `notifyOut`, and it will be called by `onContainerDrop` as well.

The last method, `onContainerDrop{3}` works very similarly to the employment of that method when we used DropZones for the DataView, where it's responsible for moving records from one store to another. Before it moves the records, it calls `clearDDStyles` to remove any added drop invitation CSS rules. Next, it removes the dropped records from the source GridPanel`{4}`. Finally, it inserts the dropped records into the destination GridPanel at the pre-determined target index and returns true to signal to the DragZone that the drop was successful.

OK, we have our `dropZoneOverrides` object complete. In order to complete this picture, we need to employ instances of DropZone for the GridPanels.

#### **Listing 14.16 Employing DropZone for the GridPanels.**

```
var leftGridDroptgtCfg = Ext.apply({}, dropZoneOverrides, {           // 1
    grid : depsComputersOK
});
new Ext.dd.DropZone(depsComputersOK.el, leftGridDroptgtCfg);           // 2

var needdUpgradesDZCfg = Ext.apply({}, dropZoneOverrides, {
    grid : needUpgradeGrid
});
new Ext.dd.DropZone(needUpgradeGrid.el, needdUpgradesDZCfg);
{1} Create a custom copy of the dropZoneOverrides object
{2} Instantiate a new instance of DropZone
```

In listing 14.16, we simply copy the `dropZoneOverrides` object`{1}`, customizing them for each of the GridPanels, then create instances of DropZone for each`{2}`. If it looks familiar, that's because we instantiated instances of DropZone for the DataViews in the same way.

Our drag and drop implementation with GridPanels is now properly configured and ready to rock. Lets refresh the page and look at how the drop invitations react, per the `onContainerOver` code.

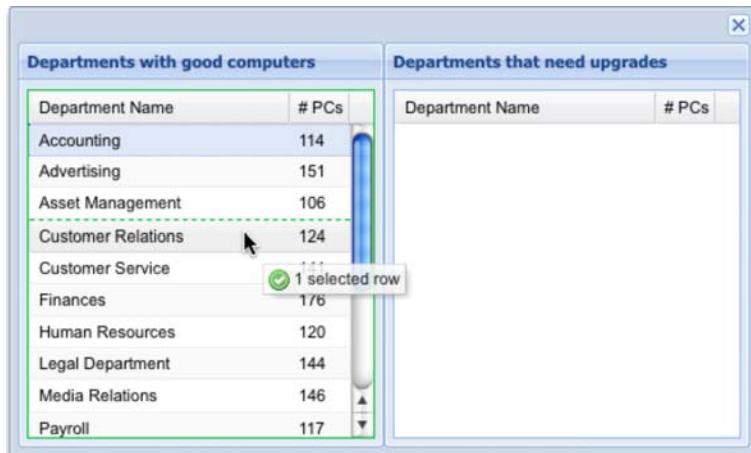


Figure 14.11 The drop invitation with mouse over row tracking drop index.

When we drag a record above a record in any of the two GridPanels, we see that the mouse movements are tracked properly and when the mouse hovers over the top half of a row, that row's top border turns green and appears dashed.

Like wise, hovering the row below the bottom half of a row will make its bottom border appear green and dashed, indicating that a drop of the dragged record will insert it below the row that we're currently hovering over.

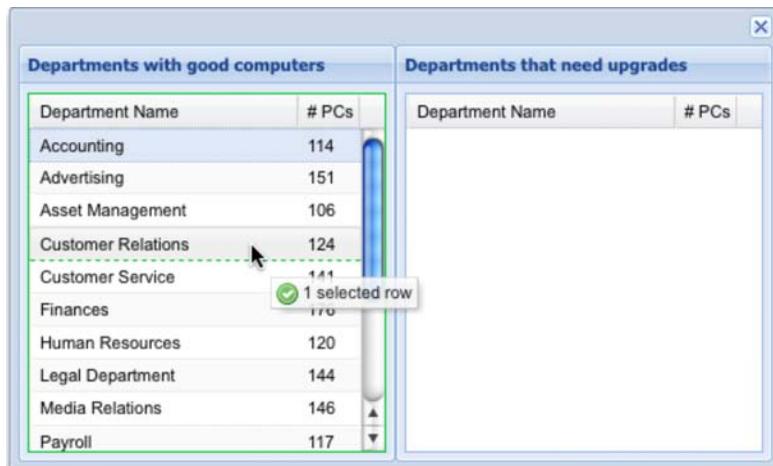


Figure 14.12 The below-row drop invitation.

And of course, dropping the record at the indicated position results in the record being moved to that position. Likewise dropping a record from one grid to another results in the record being moved as commanded, which concludes our GridPanel to GridPanel drag and drop. Notice that if a drop occurs on anything other than a row, the record is appended to the destination store.

We just learned that in order to make drag and drop between GridPanels possible, we did not have to instantiate a DragZone, but instead only needed to set enableDragDrop on the GridPanel's configuration object to true. Remember that setting that property instructs the GridView to instantiate an instance of GridDragZone when its UI renders. What we did have to implement, however was the DropZone and we did so with the ability to track the index of the record(s) to be dropped.

Next up is TreePanel to TreePanel drag and drop, where the implementation patterns changes somewhat again.

## 14.4 Drag and Drop with TreePanels

Our company just purchased another and our management needs a way to track how to absorb different employees from the purchased company's different departments. They requested that we develop something that will allow them to track the reassignment of employees using TreePanels and, of course, drag and drop.

The most important requirement is the ability to allow associates to be relocated to a specified set of similar departments. For instance, any associate from Accounting, Finances or Payroll can be reassigned to any of those departments. Likewise, associates from Customer Relations, Media Relations, Customer Service or Public Relations can be reassigned to any of those. Instead of building a valid drop matrix inside of JavaScript, the node list returned from the server will report a list of valid departments for each node. It will be up to us to somehow use that data to make the requirement a reality.

In this section, we'll not only learn how to enable drag and drop between trees, we'll tackle one of the most common challenges, which is to constrain the dropping of nodes. We'll begin by constructing the TreePanels and the Window that they'll live in.

### 14.4.1 Constructing the TreePanels

Just like with the DataView and GridPanel exercises before this, we will configure two TreePanels, both of which will be managed by an instance of Ext.Window utilizing the HBox Layout.

Being that we have built a few TreePanels already, we're going to move through this pretty fast.

#### Listing 14.17 Setting the stage for TreePanel Drag and Drop

```
Ext.QuickTips.init();  
var leftTree = {  
    xtype      : 'treepanel',  
}; // 1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

autoScroll : true,
title      : 'Their Company',
animate    : false,
loader     : new Ext.tree.TreeLoader({
    url : 'theirCompany.php'
}),
root       : {
    text      : 'Their Company',
    id       : 'theirCompany',
    expanded : true
}
};

var rightTree = { // 2
    xtype      : 'treepanel',
    title      : 'Our Company',
    autoScroll : true,
    animate    : false,
    loader     : new Ext.tree.TreeLoader({
        url : 'ourCompany.php'
}),
root       : {
    text      : 'Our Company',
    id       : 'ourCompany',
    expanded : true
}
};

new Ext.Window({ // 3
    height     : 350,
    width      : 450,
    layout     : 'hbox',
    border     : false,
    layoutConfig : {
        align : 'stretch'
    },
    defaults   : {
        flex : 1
    },
    items      : [
        leftTree,
        rightTree
    ]
}).show();
{1} Configuring the TreePanel for their company
{2} The TreePanel configuration for our company
{3} Construct the Window to house the TreePanels

```

In the listing above, we create two TreePanels and an Ext Window, which will contain them and manage their sizes using the HBox layout. The left TreePanel**{1}** will load a list of departments for the other company. Each department will have to be expanded to reveal the child items.

The right TreePanel**{2}** will load up a list of departments for our company, which lucky for us, align with the company being sold. For simplicity, we will not display the employees currently in our company's departments.

Lastly, the Ext Window is created to manage the two TreePanels side-by-side. Here are our TreePanels rendered on screen.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>



Figure 14.13 The two TreePanels rendered.

OK we have the two TreePanels rendered within the Ext Window. It's time to get the party started with drag and drop.

#### **14.4.2 Enabling Drag and Drop**

When exploring how to employ drag and drop with DataViews, we were required to implement both the `DragZone` and `DropZone` classes. When applying this feature to GridPanels, we learned that we were only required to implement `DropZone` as the `GridView` automatically creates `GridDragZone` if the `GridPanel` has the `enableDragDrop` property set.

With the TreePanel, drag and drop is much easier. All we have to do is set the following property on both of the TreePanel configuration objects:

```
enableDD : true
```

And the TreePanel takes care of instantiating the `TreeDragZone` and `TreeDropZone` classes for us. To specify the drag drop group that the TreePanel is to participate in, set the `ddGroup` configuration parameter as such:

```
ddGroup : 'myTreeDDGroup'
```

Here's what drag and drop with the two TreePanels looks like in action.

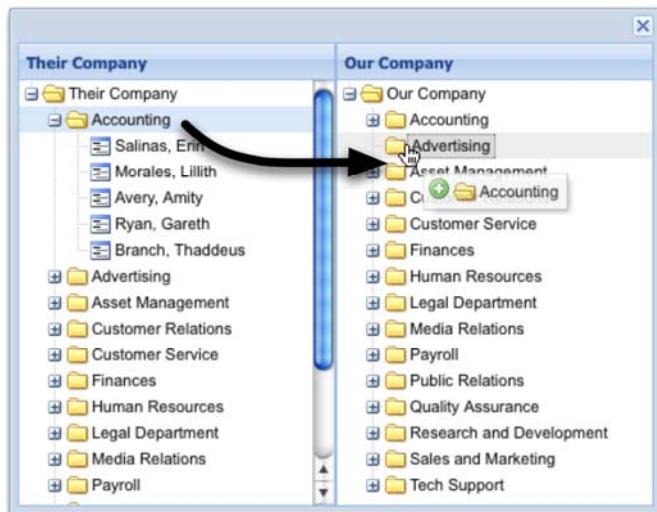


Figure 14.14 Drag and Drop enabled on TreePanels

As we can see, enabling drag and drop on TreePanels is very simple to do. But, wait a minute. Everything can be dragged and dropped, which can be useful for, lets say a File System management tool, but for what we need, it just won't cut it.

Out of the proverbial box, the `TreeNode` class allows for some constraints to be applied to the dragging and dropping of nodes when drag and drop is applied to their owner TreePanel. The two parameters that help control these behaviors are `allowDrag` and `allowDrop`, both of which are configured on each node of the tree and default to true even if the properties do not exist. We could have the returning JSON include these properties, but they are very rigid. With these two properties, you can either allow drag or not, or allow drop or not. Clearly this will not help us fulfill the requirement to allow any associate from a set group of departments to be dropped on those departments. For this, we'll need to craft up something much more flexible.

We're going to tackle this intense procedure next. Are you ready?

#### 14.4.4 Employing flexible constraints

Thus far, when implementing drag and drop, we created objects to override the required template methods, which gave us the control we desired for the drag and drop classes. With the TreePanel, we can apply template overrides by specifying `dragConfig` or `dropConfig` objects in the TreePanel's configuration object.

To have better control of what nodes can be dragged around, we will need to create a `dragConfig` object, which overrides the `onBeforeDrag` template method. We'll need to place this code *before* the TreePanel configurations.

```
var dragConfig = {
    onBeforeDrag : function(dragData, eventObj) {
        return dragData.node.attributes.leaf;
    }
};
```

In the above snippet, we create a dragConfig object that contains the onBeforeDrag override method. All it does is return the dragged node's leaf attribute. This will prevent all branch nodes from being dragged around screen. To apply the drag constraint, simply set the dragConfig object on both TreePanels as such:

```
dragConfig : dragConfig,
```

Refresh the page and you'll see that the department nodes can no longer be dragged around. This completes half of the requirement. As of now, any employee nodes can be dragged to any department, which means we need to work on the code to provide better drop point constraints.

This listing is going to be rather large. The reason being there are a lot of tests that need to take place in order to ensure that the desired drop operation is completely correct. Any slip up and nodes can be incorrectly dropped where they don't belong. Please don't be alarmed, we'll go into an in depth discussion about it.

Just like the dragConfig object creation, we'll need to place the following code *before* the TreePanel configuration.

### Listing 14.18 Applying better drop constraints.

```
var dropConfig = {
    isValidDropPoint : function(nodeData, pt, dd, eventObj, data) {

        var treeDropZone = Ext.tree.TreeDropZone;
        var isVldDrpPnt = treeDropZone.prototype.isValidDropPoint; // 1

        var drpNd          = data.node; // 2
        var drpNdPrntDept = drpNd.parentNode.attributes.text;
        var drpNdOwnerTreeId = drpNd.getOwnerTree().id;
        var validDropPoints = drpNd.attributes.validDropPoints || [];

        var tgtNd          = nodeData.node; // 3
        var tgtNdPrnt      = tgtNd.parentNode;
        var tgtNdOwnerTree = tgtNd.getOwnerTree();
        var tgtNdOwnerTreeId = tgtNdOwnerTree.id;

        var isSameTree = drpNdOwnerTreeId === tgtNdOwnerTreeId;

        if (!tgtNdPrnt || isSameTree) { // 4
            return false;
        }

        var tgtNdPrntDept = tgtNdPrnt.attributes.text; // 5
        var tgtNdTxt       = tgtNd.attributes.text;

        if (drpNdPrntDept === tgtNdPrntDept) { // 6
            return isVldDrpPnt.apply(tgtNdOwnerTree.dropZone, arguments);
        }
    }
};
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        }
        else if (Ext.isArray(validDropPoints)) { // 7
            var isVldDept = false;

            var drpPoint = tree.dropZone.getDropPoint(
                eventObj, nodeData, dd);

            Ext.each(validDropPoints, function(dpName) { // 8
                if (tgtNdTxt == dpName) {
                    isVldDept = dpName;
                }
            });

            if (isVldDept && drpPoint === 'append') { // 9
                return isVldDrpPnt.apply(tgtNdOwnerTree.dropZone, arguments);
            }
        }

        return false;
    };
}

{1} Create a reference to the TreeDropZone's isValidDropPoint method
{2} Gather drop node references
{3} Gather target node references
{4} Return false if the target node has no parent or this is the same tree
{5} Create more target node references
{6} Return true if the drop node parent is the same as the target
{7} If the node has a validDropPoints array, process it
{8} Loop through the valid drop points
{9} Allow drop if the target node is within the validDropPoints

```

In the listing above, create a dropConfig object and work really hard to constraint what nodes can be dropped where and we utilize the TreeDropZone.prototype.isValidDropPoint method finish the processing when we deem that the drop point is valid. Here's how all of this stuff works.

First, a reference to the TreeDropZone.prototype.isValidDropPoint is created**{1}**. We do this because even though we're this isValidDropPoint method will *override* the targeted TreeDropZone instance; we still need the processing that is provided by the original TreeDropZone isValidDropPoint method. We need it because it is responsible for appdending some important properties to the drag data object that the TreeDragZone creates. In order not to duplicate the code, we call it when we are comfortable with the current mouse over situation.

Next, quite a few drop node-related references are created**{2}**; such as the drop node's parent (department) text, the drop node's owner tree and its id. Similarly, we gather references for the target node**{3}**. These references will be the backbone to help determine whether or not the target node is a valid drop target.

Moving on, we reach our first test, where we determine if the drop target is a root node by detecting the absence of the target's parent node. If the target node has no parent, then it's a root node. We also try to see if the drop node's owner tree is the same as the target node's. If either test case is true, then we simply return false**{4}**.

Looking ahead, we create references for the target node's parent (department) text and the target node `text{5}`. We'll use both to determine if the drop node is a valid drop point based on the possible drop scenarios.

We then reach our first major test, where we look to see if the drop node's parent text matches that of the target node`{6}`. This test is useful in the scenario where you're dropping a node on another leaf element that exists within a similar parent. For example, we drag "Salinas, Erin" from Accounting from their company and drop Erin on another associate node that exists within Accounting in our company. If this condition is true we call upon the TreeDragZone's `isValidDropPoint` to properly process the drop and return the results, which ultimately results in a favorable possible drop condition.

If that test fails, we move on to determine if the target node is a branch node and matches any of the drop points in the `validDropPoints` array. We do this by testing the return of the utility method `Ext.isArray` to verify that the drop node's `validDropPoints` array is `valid{7}`.

If the test is favorable, then we move to get what is known as the drop "point". The TreeDropZone performs similar calculations to the ones we did when determining if the drag operation was occurring above or below a row, except it is concerned about nodes. Because drop nodes can be *appended* to a target branch as well as dropped *above* or *below* a target node, the TreeDropZone `getDropPoint` method returns any one of the values we just mentioned.

This is important to know because when the drag node is hovering over a branch element, the drop point could be "above" or "below", which means that the drop node could be dropped below or above a branch node. If we allowed this to occur, we would have associates that would be placed *outside* of a department and is unfavorable.

Moving on, we loop through the `validDropPoints` array and test to see if any of the drop points in the array match the target (department) node text. If so, then we return the value of the TreeDragZone's `isValidDropPoint` call to indicate that a successful drop is possible.

If none of these tests pass then a false value is returned and the TreeDragZone knows to update the StatusProxy indicating that a favorable drop is not possible.

OK, we've done a lot of work to get these constraints configured. In order to take advantage of this code, we're going to have to reconfigure the TreePanels with the `dropConfig` object as such:

```
dropConfig : dropConfig
```

Adding the above configuration parameter to your TreePanels will ensure that an instance of TreeDropZone is created and bound to the TreePanels. Lets refresh our page and see the constraints in action.

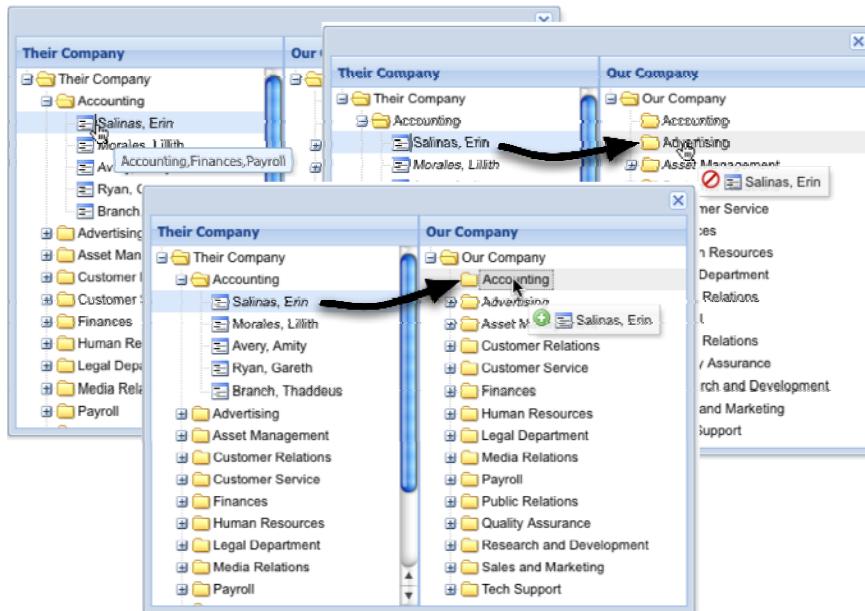


Figure 14.15 Testing our drop target constraint logic.

We can test our constraint logic by attempting to drag a department branch node. We'll see that it's impossible. This tells us that the `onBeforeDrag` override method is working as designed.

Next, to determine where the associate can be dragged, hover over it and you'll see a QuickTip appear with the values that are in the `validDropPoints` array. In hovering over Erin, we can see that she can be dropped on Accounting but not Advertising. When we drag Erin over Accounting on the right hand TreePanel, the StatusProxy displays a valid drop icon. However, if we hover her over advertising, we can see an invalid drop icon in the StatusProxy. As extra credit, we can try to hover Erin over Finances and Payroll and we'll see a valid drop icon there too.

The last test we can perform is by dropping an associate or two into a valid department. We can test the ability to drop leaf nodes above or below leaf nodes within a valid department as well.

There you have it. Drag and Drop with two tree panels with a complex, but somewhat flexible drop constraint system. I'm sure our managers will be pleased that we delivered what they were asking for.

## **Summary**

We spent a lot of time in this chapter looking at different ways to implement drag and drop with the three most commonly used widgets within the framework and we learned a lot while doing it.

We started with implementing this behavior with DataViews, where we learned that we were responsible for configuring both the DragZone and DropZone classes. While employing these this behavior, we developed a custom and more useful way to display the records being dragged around.

Next, we learned how to employ this behavior with two grid panels, and learned that we were mainly responsible for implementing a DropZone to handle the drops. In doing so, we tackled the hard problem of detecting the index of a node drop to allow records to be inserted any desired index.

Lastly, we explored drag and drop with TreePanels. In this implementation, we discovered that enabling this behavior with TreePanels is the simplest, and the application of somewhat complex constraints on the drop gestures was the most difficult task.

In the next chapter, we'll learn about plugins and Extensions and how they work. This will be where we start to use object-oriented techniques with JavaScript and will be a lot of fun.

# 15

## *Extensions and Plugins*

Every Ext JS developer faces the challenge where reusability becomes an issue. Often times, a component of an application is required to appear more than once within the application's usage lifetime. In much larger applications, reusability of components is paramount to performance as well as maintainability of the code itself. This is why we'll focus on the concept of reusability with the use of framework extensions and plugins.

In the first section of this chapter, we're going to learn the basics of extending (subclassing) with Ext JS. We'll begin by learning how to create subclasses with JavaScript, where we'll get to see what it takes to get the job done with the native language tools. This will give us the necessary foundation to refactor our newly created subclass to leverage `Ext.extend` implementing two different popular design patterns.

Once we've gotten familiar with creating basic subclasses, we'll focus our attention on extending Ext JS Components. This is where we'll get to have fun learning about the basics of framework extensions and actually solve a real-world problem by extending the `GridPanel` widget and see it in action.

When we finish the extension, we'll learn how extensions solve problems, but can create inheritance issues where similar functionality is desired across multiple widgets. Once we understand the basic limitations of extensions, we'll convert our extension into a plugin where its functionality can easily be shared across the `GridPanel` and any descendant thereof.

## 15.1 Inheritance with Ext JS

JavaScript provides all of the necessary tools for class inheritance, but there are a lot of manual steps that we, as developers have to take in order to achieve this. The net result is wordy code. Ext JS makes inheritance much easier with its `Ext.extend` utility method. To begin learning about inheritance, we're going to have to create a base class.

To help us along, envision we're working for an automobile dealership that is going to sell two types of car. First is the base car, which will serve as a foundation to construct the premium model. Instead of using 3D models to describe the two car models, we'll use JavaScript classes.

### NOTE:

If you are new to Object Oriented JavaScript or are feeling a bit rusty, the Mozilla foundation has an excellent article to bring you up to speed or polish your skills. It can be found at the following URL: [https://developer.mozilla.org/en/Introduction\\_to\\_Object-Oriented\\_JavaScript](https://developer.mozilla.org/en/Introduction_to_Object-Oriented_JavaScript).

We'll begin by constructing a class to describe the base car.

### Listing 15.1 Constructing our base class.

```
var BaseCar = function(config) { // 1
    this.octaneRequired = 86;

    this.shiftTo = function(gear) {
        this.gear = gear;
    };

    this.shiftTo('park');

};

BaseCar.prototype = { // 2
    engine : 'I4',
    turbo : false,
    wheels : 'basic',
    getEngine : function() {
        return this.engine;
    },
    drive : function() {
        return "Vrrrrooooooooom - I'm driving!";
    }
};

{1} Create the constructor
{2} Assign the prototype object
```

In the above listing, we create the `BaseCar` class constructor **{1}**, which when instantiated, sets the instance's local `this.octaneRequired` property, adds a `this.shiftTo` method and calls it, setting the local `this.gear` property to '`'park'`'. Next, we configure the `BaseCar`'s `prototype` object **{2}**, which contains three properties that describe the `BaseCar` and two methods.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

We could use the following code to instantiate an instance of BaseCar and inspect its contents with Firebug.

```
var mySlowCar = new BaseCar();
mySlowCar.drive();
console.log(mySlowCar.getEngine());

console.log('mySlowCar contents:');
console.dir(mySlowCar)
```

Here's what the output of the above code looks like in the Firebug multi-line editor and console.

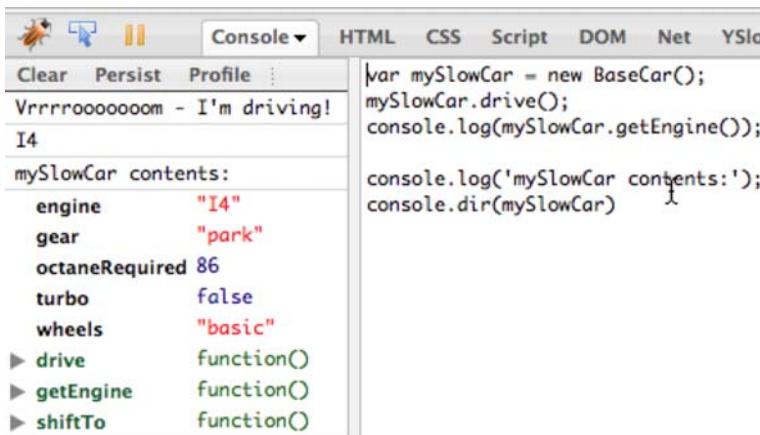


Figure 15.1 Instantiating an instance of BaseCar and exercising two of its methods;

With our BaseCar class set, we can now focus our attention at subclassing the BaseCar class. We'll do it the traditional way first. This will give us a better understanding to what's going on under the hood when we use Ext.extend later on.

### 15.1.1 Inheritance with JavaScript

Creating a subclass using native JavaScript is achievable with multiple steps. Rather than simply describing them, we'll actually walk through the steps together. The following Listing creates PremiumCar, a subclass of the BaseCar class.

#### **Listing 15.2 Creating a subclass the old-school way**

```
var PremiumCar = function() {
    PremiumCar.superclass.constructor.call(this);
    this.octaneRequired = 93;
};

PremiumCar.prototype = new BaseCar(); // 3
PremiumCar.superclass = BaseCar.prototype; // 4
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
PremiumCar.prototype.turbo = true;
PremiumCar.prototype.wheels = 'premium';

PremiumCar.prototype.drive = function() {
    this.shiftTo('drive');
    PremiumCar.superclass.drive.call(this);
};

PremiumCar.prototype.getEngine = function() {
    return 'Turbo ' + this.engine;
};

{1} Configure the subclass constructor
{2} Call the superclass constructor
{3} Set the subclass prototype
{4} Set the subclass' superclass reference
```

To create a subclass, we begin by creating a new constructor, which is assigned to the reference PremiumCar{1}. Within this constructor is a call to constructor method of the PremiumCar.superclass within the scope of the instance of PremiumCar{2} being created (this).

We do this, because unlike other Object Oriented languages, JavaScript subclasses do not natively call their superclass constructor{2}. Calling the superclass constructor gives it a chance to execute and perform any constructor-specific functions that the subclass might need. In our case, the shiftTo method is being added and called in the BaseCar constructor. Not calling the superclass constructor, would mean that our subclass would not get the benefits provided by the base class constructor.

Next, we set the prototype of the PremiumCar to the result of a new instance of BaseCar. Performing this step allows the PremiumCar.prototype to inherit all of the properties and methods from the BaseCar. This is known as inheritance through prototyping and is the most common and robust methods of creating class hierarchies in JavaScript.

In the next line, we set the PremiumCar's superclass reference to the prototype of the BaseCar class. We then can use this superclass reference to do things like create so-called extension methods, such as PremiumCar.prototype.drive. This method is known as an extension method because it *calls* the like-named method from the superclass prototype, but from the scope of the instance of the subclass it's attached to.

**TIP:**

All JavaScript Functions (JavaScript 1.3 and later) have two methods that force the scope execution, which are `call` and `apply`. To learn more about `call` and `apply` visit the following URL: <http://www.webreference.com/js/column26/apply.html>

With the subclass now created, we can test things out by instantiating an instance of PremiumCar with the following code entered into the Firebug editor.

```
var myFastCar = new PremiumCar();
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:  
<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
myFastCar.drive();

console.log('myFastCar contents:');
console.dir(myFastCar);
```

Here's what the output of would look like in the Firebug multi-line editor and console.

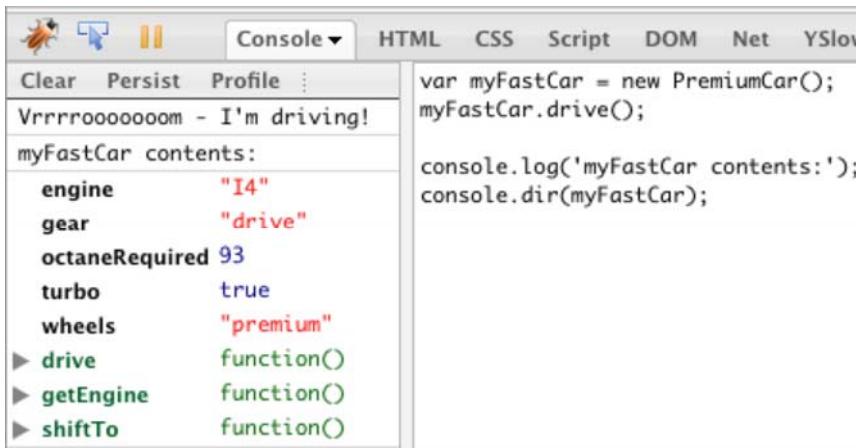


Figure 15.2 Our PremiumCar subclass in action.

We can see that our subclass performed as desired. From the `console.dir` output, we can see subclass constructor set the `octaneRequired` property to 93 and `drive` extension method even set the `gear` method as "drive".

From our exercise, we can see that we're responsible for all of the crucial steps in order to achieve prototypal inheritance with native JavaScript. First, we had to create the constructor of the subclass. Then we had to set the prototype of the subclass to a new instance of the base class. Next, for convenience, we set the subclass' superclass reference. Lastly, we added members to the prototype one by one.

We can see that quite a few steps need to be followed in order to create subclasses with the native language constructs. Next, we're going to see how the `Ext.extend` makes creating subclasses much easier.

### 15.1.2 Extending with Ext JS

There are two common patterns to using `Ext.extend`. The first is an older method, whose roots stem back from the 1.0 days of the framework and involves first creating the subclass constructor, then calling upon `Ext.extend` to finish the work. The second is a more modern method, stemming from the 2.0 days of the framework, and involves having `Ext.extend` do all of the work.

We'll begin exploration of the older pattern of `Ext.extend`. Knowing both patterns of will benefit you when reading extension code from other Ext JS developers, as the usage ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

between the two patterns vary from developer to developer. We'll be using the previously created `BaseCar` class for this exercise.

### **Listing 15.3 Creating our first extension**

```
var PremiumCar = function() { // 1
    PremiumCar.superclass.constructor.call(this); // 2
    this.octaneRequired = 93;
};

Ext.extend(PremiumCar, BaseCar, { // 3
    turbo : true,
    wheels : 'premium',
    getEngine : function() {
        return this.engine + ' Turbo';
    },
    drive : function() {
        this.shiftTo('drive');
        PremiumCar.superclass.drive.call(this);
    }
});
{1} Create the PremiumCar constructor
{2} Scoping the call of the superclass constructor
{3} Extending BaseCar
```

In the above listing, we create the `PremiumCar` class, which is an “extension” (subclass) of the `BaseCar` class using the `Ext.extend` tool. Here’s how this works.

First, we create a constructor for the `PremiumCar{1}`. This constructor is an exact duplicate of previously created `PremiumCar` constructor.

When thinking about extending classes, you must consider whether prototypal methods in the subclass will share the same name as prototypal methods in the base class. If they will share the same symbolic reference name, you must consider whether or not they will be extension methods or overrides.

An extension method is simply a method in a subclass that shares the same reference name as another method in a base class. What makes this an extension method is the fact that it includes the execution of the base class method within itself. The `PremiumCar` constructor is an *extension* of the `BaseCar` constructor because of the included execution of the `BaseCar` constructor method. The reason you would want to extend a method is to reduce code duplication, thus reusing the code in the base class method.

An override method is a method in a subclass that shares the same reference name as another method in a base class but does not execute the same-named method from the base class. You override a method if you wish to completely discard the code that is in the like-named method in the base class.

To complete the extension process, we call upon `Ext.extend{3}`, for which we are passing three parameters; subclass, base class and what is known in the community as an “overrides” object. Here’s how this works.

`Ext.extend` first sets the subclass’s prototype to the result of a new instance of the base class, which is like what we did when creating our subclass manually. `Ext.extend`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

then applies the symbolic references of the overrides object to the prototype of the subclass. The symbolic references in the overrides object take precedence over those in the base class's prototype. This is just like us manually adding the members when we created our subclass earlier.

We can see that it only took two steps to extend the `BaseCar` class: create the constructor for the subclass, and use `Ext.extend` to tie everything else together. It not only took less work on our part, but the amount of code we had to write was much less and easier to digest, which is one of the lesser-known benefits to using `Ext.extend` to subclass.

Now that we have our `PremiumCar` configured using `Ext.extend`, we can see it in action using Firebug. We can do so using the exact same code we used when exercising our manually created subclass:

```
var myFastCar = new PremiumCar();
myFastCar.drive();

console.log('myFastCar contents:');
console.dir(myFastCar);
```

Here's what it looks like in the Firebug console.

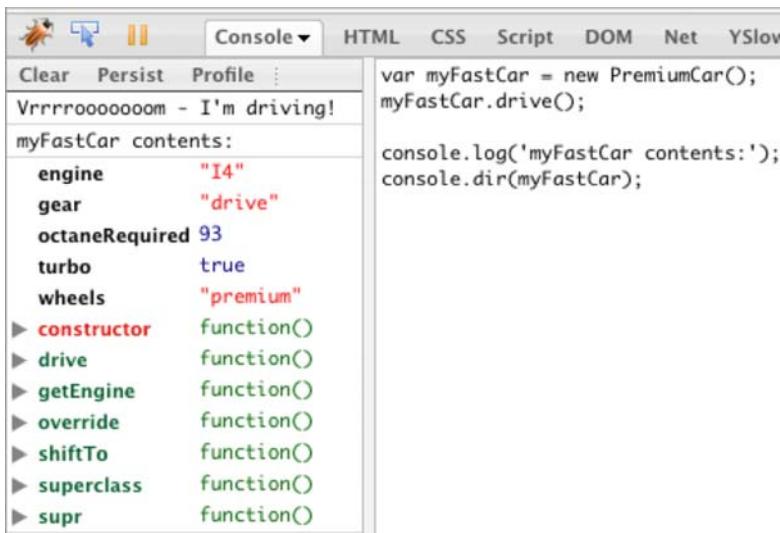


Figure 15.3 The results of the instantiation of the `PremiumCar` class.

Looking at the illustration above, we can see that `Ext.extend` adds some convenience references to the subclass. These can be useful from within the instance of the class. They are references to the `constructor`, and `superclass` methods. The `override` method is useful if you want to change members for the instance of the class that you've instantiated.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

We've just successfully extended a class using what is known as an old-fashioned method, which consists of creating a constructor then calling upon `Ext.extend` to "copy" the prototype from the base class to the subclass and apply the overrides object to the subclass's prototype. This pattern of extending works perfectly and is used by many, but there is a much more modern pattern that we'll be using, which allows us to reduce the extension to a single step and provide the constructor inside of the overrides object.

Here is the exact same extension with the modern `Ext.extend` implementation pattern:

#### **Listing 15.4 Using `Ext.extend` using the modern pattern.**

```
var PremiumCar = Ext.extend(BaseCar, {
    turbo      : true,
    wheels     : 'premium',
    constructor : function() { // 1
        PremiumCar.superclass.constructor.call(this);
        this.octaneRequired = 93;
    },
    drive      : function() {
        this.shiftTo('drive');
        PremiumCar.superclass.drive.call(this);
    },
    getEngine   : function() {
        return this.engine + ' Turbo';
    }
});
```

##### **{1} Including the constructor inside of the overrides object.**

In the above implementation of `Ext.extend`, we pass two arguments. The first is a reference to the base class and the second is the overrides object to be applied to the subclass's prototype. That's it. The biggest difference between the first implementation and this one is that we're including the `constructor{1}` method inside of the overrides object. Ext JS is smart enough to know to use this method to create a constructor for us. Lastly, notice that the `PremiumCar` reference is the result of the `Ext.extend` method call.

By now, you're probably wondering what makes this pattern better than the first? The single and most compelling reason is code readability. For many, it's easier digest to read code formatted in this way. It is for this reason that many developers have created classes from scratch by means of extending `Object`, instead of creating a constructor and then prototype object.

Here's how you create a simple class using this newer pattern.

```
var NewClass = Ext.extend(Object, {
    someProperty : 'Some property',
    constructor   : function() {
        NewClass.superclass.constructor.call(this);
    },
    aMethod      : function() {
        console.info("A method has executed.");
    }
})
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```
});
```

By now, understand how to use the utility method `Ext.extend` to create subclasses. We learned that `Ext.extend` provides a means for creating subclasses with fewer steps compared to the traditional JavaScript method. We also got a chance to explore the two most common `Ext.extend` patterns used by Ext JS developers. For the rest of this book, we'll be using the modern pattern.

Next, we're going to use our newly gain knowledge and learn how to extend Components.

## **15.2 Extending Ext JS Components**

Extensions to the framework are developed to introduce additional functionality to existing classes in the name of reusability. It is the concept of reusability that actually drives framework and when utilized properly can enhance the development of your applications.

Some developers create what are known as "preconfigured classes", which are constructed mainly as a means reduce the amount of application-level code by stuffing configuration parameters inside of the class itself. Having such extensions alleviates the application-level code from having to manage much of the configuration, thus only requiring the simple instantiation of such classes. This type of class is great, but only if you're expecting to stamp out more than one instance of this class.

Other extensions add features, such as utility methods or embed behavioral logic inside of the class itself. An example of this would be a `FormPanel` that automatically pops up a `MessageBox` whenever a save operation failure occurs. I often create extensions for applications just for this very reason, where the widget contains some built-in business logic.

My favorite kind of extension is what I like to call a "composite widget", which combines the use of one or more widgets together into one class. An example of this would be a `Window` that has an embedded `GridPanel`. Or a `FormPanel` that embeds a `TabPanel` to spread out its fields over multiple panels.

This is the type of extension that we're going to be focusing on today, where we'll merge a `GridPanel` with a `Menu`.

### **15.2.1 Thinking about what we're building**

When I work to construct an extension, I often take a step back and try to analyze the problem from all facets as I do with puzzles. I do this because I view the creation of extensions as a way to solve issues. Sometimes these problems can be extremely complex, such as the creation of a dynamic wizard-like widget that has a lot of workflow rules that must be controlled by the UI. Often times, it's the problem of reusability that I use extensions to solve. This is where we'll focus on for the rest of the chapter.

When thinking about common tasks during application development, I often wonder how such tasks can be made easier through an extension. One task that comes to mind is that

we, as end developers, must code for the destruction of loosely coupled widgets, such as Menus or Windows when their parent component is destroyed.

We experienced this when we explored the creation of a `GridPanel`, where we attached Menu and configured it to display when the grid's `contextmenu` event fires. Recall that we had to manually configure the Menu's to destruction upon the `GridPanel`'s destruction. If we extrapolate this task over the span of an application where many `GridPanels` coupled with Menus are to be rendered on screen, we can easily visualize the amount of code duplication required to make this work. Before we start coding, lets take a quick moment to analyze the problem and come up with the best possible solution.

In order to mitigate this code duplication risk, we're going to have to create an extension to the `GridPanel` that will automatically handle the instantiation and destruction of the Menu. But what other features can we add to this extension to make it more robust?

The first thing that comes to mind is the differences in the selection getter and setter methods for the `RowSelectionModel` and the `CellSelectionModel`. The `RowSelectionModel` has `selectRow` and `getSelected` while the `CellSelectionModel` has `selectCell` and `getSelectedCell`. It would be great if our extension should be able to handle this variation in the `GridPanel`'s selection models. Such a feature would reduce the amount of code in the application layer.

With a clear picture of the issues we're going to solve, we can begin the construction of our first Ext JS extension.

### **15.2.2 Extending `GridPanel`**

To extend the `GridPanel` class, we're going to leverage the `Ext.extend` method, using the modern pattern. To give you a bird's eye view of our extension, below is the template for the extension that we're going to create:

#### **Listing 15.5 The template for our `GridPanel` extension**

```
var CtxMenuGridPanel = Ext.extend(Ext.grid.GridPanel, {
    constructor : function() { // 1
        },
        onCellContextMenu : function(grid, rowIndex, cellIndex, evtObj) { // 2
            },
            getSelectedRecord : function() { // 3
                },
                onDestroy : function() { // 4
                    }
    });
    {1} The constructor extension
    {2} A custom contextmenu event handler
    {3} A convenience selection retrieval method
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

**{4} Extend to the `onDestroy` to do our cleanup.**

In the above listing, we have the template to our extension, where we have four methods that will be applied to the subclass' prototype. The first is the `constructor{1}`, which will be our vector to extend the `GridPanel`. Lets take a moment to analyze why we are extending via the `constructor` method. Because this is such an important topic, we're going to take a moment to discuss the differences in hopes that you can make an educated decision when you decide to create extensions in your projects.

Earlier in this book, when we discussed the `Component` lifecycle, we learned of the `initComponent` method, which is meant to augment the `constructor` and is meant as a place to for developers to extend components. We learned that `initComponent` is executed inside of the `Component` class' `constructor`, but only after a few crucial setup tasks for the `Component` has taken place. These tasks include the caching and application of the configuration object properties to the instance of the class, the setup of base events and the registration of the instance of `Component` with the `ComponentMgr` class.

Knowing this, we can make an educated decision on where to extend an Ext JS Component. To make this decision, I figure out if configured instances of the subclass will ever need to be cloned via the `cloneConfig` utility method. If yes, then extending via the `constructor` is the best choice. Else, extending via the `initComponent` method will work. If you are unsure, then default to extending via the `constructor`. Also, it's important to point out that all non-`Component` classes do not have an `initComponent` method, thus extending via `constructor` is the only option.

If we take a look at the rest of the extension template, we see three other methods in the overrides configuration object. The first, `onCellContextMenu{2}`, will be the method that will respond to the `cellcontextmenu` events and will ultimately display the menu. The reason we're choosing the `cellcontextmenu` event over the `rowcontextmenu` event is because with the `cellcontextmenu` event, we get the row and column coordinates for which the event occurred, which will help our extension understand how to select the row or cell. The `rowcontextmenu` event, only provides the row that the event occurred and is unusable when taking the `CellSelectionModel` into consideration.

**TIP:**

The arguments passed to `onCellContextMenu` utility method can be found in the `Ext.grid.GridPanel` API documentation, under the "cellcontextmenu" events section.

The next template method, `getSelectedRecord{3}`, is a utility for us to be able to get the selected record despite the cell or row selected. This method will take into account the selection model and leverage the correct selection model getter method. Lastly, the `onDestroy{2}` method will extend the `GridPanel`'s own `onDestroy` method. This is where we'll code the automatic destruction of the Menu.

Our template class is now set and ready to be filled in. We'll start with the constructor.

### **Listing 15.6 Adding the constructor to our extension**

```
constructor : function() {
    CtxMenuGridPanel.superclass.constructor.apply(this, arguments);           // 1

    if (this.menu) {
        if (!(this.menu instanceof Ext.menu.Menu)) {                         // 2
            this.menu = new Ext.menu.Menu(this.menu);
        }

        this.on({
            scope          : this,
            cellcontextmenu : this.onCellContextMenu
        });
    }
},
{1} Call the superclass constructor
{2} Intelligently create an instance of Menu
{3} Register a cellcontextmenu event handler
```

Our constructor extension method takes care of automatically creating the instance of the Menu widget for us - but does so in an intelligent manner. The very first task is executing the superclass (GridPanel) constructor`{1}` method within the scope of the instance of this subclass.

Next, in order to facilitate the automatic instantiation of the Menu widget, the method will need to understand if the local menu reference is present and if it is already an instance of `Ext.menu.Menu``{2}` or not. This simple test will allow for three different implementation possibilities for this subclass.

You can either pass a Menu configuration object:

```
new CtxMenuGridPanel({
    // ... (other configuration options)
    menu : {
        items : [
            { text : 'menu item 1' },
            { text : 'menu item 2' }
        ]
    }
});
```

Or an array of MenuItem configuration objects:

```
new CtxMenuGridPanel({
    // ... (other configuration options)
    menu : [
        { text : 'menu item 1' },
        { text : 'menu item 2' }
    ]
});
```

Or lastly an instance of `Ext.menu.Menu` as the menu configuration:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

var myMenu = new Ext.menu.Menu({
    items : [
        { text : 'menu item 1' },
        { text : 'menu item 2' }
    ]
});

new CtxMenuGridPanel({
    menu : myMenu
});

```

Having this type of flexibility to the implementation of the subclass, and plays to the framework's culture. Truth be told, part of this flexibility comes from the Menu widget itself, whose constructor will accept a configuration object or array of MenuItem configuration objects. Also notice that the local this.menu reference is overwritten if an instance of Ext.menu.Menu is created.

At the end of this constructor, we configure the local this.onCellContextMenu method as the cellcontextmenu event handler{3}. Notice that the scope for which the event handler is set to this, or the instance of our GridPanel subclass. Therefore, onCellContextMenu will need the local this.menu reference to manage the display of the Menu widget itself.

Next, we'll construct event handler method.

### **Listing 15.7 Constructing the onCellContextMenu event handler method**

```

onCellContextMenu : function(grid, rowIndex, cellIndex, evtObj) {
    evtObj.stopEvent();

    if (this.selModel instanceof Ext.grid.RowSelectionModel) {           // 1
        this.selModel.selectRow(rowIndex);
    } else if (this.selModel instanceof Ext.grid.CellSelectionModel) {
        this.selModel.select(rowIndex, cellIndex);
    }
    this.menu.showAt(evtObj.getXY());
},
{1} Selection model specific selection method calling

```

In order to properly select the cell or row being right-clicked on, the above event handler method needs to determine which selection model is being used by the GridPanel. We can determine this by using the generic JavaScript instanceof operator. If the selection model is a RowSelectionModel, then the selectRow method is used, else if it's a CellSelectionModel, the select method is used.

Having such logic in the contextmenu event handler brings another level of flexibility to this extension. We can also use the same logic to determine what selection "getter" method to use, which is what we're going to do next.

### **Listing 15.8 Creating the getSelectedRecord utility method**

```
getSelectedRecord : function() {
    if (this.selModel instanceof Ext.grid.RowSelectionModel) {
        return this.selModel.getSelected();
    }
    else if (this.selModel instanceof Ext.grid.CellSelectionModel) {
        var selectedCell = this.selModel.getSelectedCell();
        return this.store.getAt(selectedCell[0]);
    }
},
```

In the above `getSelectedRecord` utility method, the record related to the currently selected cell or row will be returned. This method is completely Again, we're using the `instanceof` operator to determine what type of selection model is being used and return result of the proper getter method call.

At this point, we have all but one method filled in. The last method, `onDestroy`, will handle the automatic cleanup for our extension, calling upon the destruction of the menu if it exists.

```
onDestroy : function() {
    if (this.menu && this.menu.destroy) {
        this.menu.destroy();
    }
    CtxMenuGridPanel.superclass.onDestroy.apply(this, arguments);
}
```

In the above extension method, we inspect for the presence of a local `this.menu` reference and check to see if that reference contains a `destroy` reference. If both of these conditions are true, then we execute its `destroy` method. Recall that the `destroy` method initiates the destruction phase of a Component's lifecycle, thus purging any DOM nodes that the loosely coupled menu might have created. Lastly, we execute the superclass `onDestroy` method within the scope of `this` instance of our subclass, thus completing the extension.

Like all of the Ext JS widgets, we should register our extension with `Ext.ComponentMgr` to allow for the lazy instantiation of our extension with XTypes. To perform this registration, we'll execute `Ext.reg` and pass the XType string to identify our subclass, and the actual reference to our subclass itself. These statements are placed at the very end of a class' creation.

```
Ext.reg('contextMenuGridPanel', CtxMenuGridPanel);
```

With our extension completed, we can next create an implementation of it and see it in action.

#### **15.2.3 Our extension in action**

When discussing the constructor to our `GridPanel` extension, we talked about the three different patterns for implementation where the `menu` reference for the configuration object ©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

can be set to an array of MenuItem configuration objects, an actual instance of Menu or a configuration object designed for an instance of Menu. For this implementation, we'll choose the first pattern, which is an array of MenuItem configuration objects. This will give us an opportunity to see an automatic instantiation of Menu as coded in our extension's constructor.

We'll start with the creation of the remote JsonStore.

### **Listing 15.9 Creating the remote JsonStore for our extension implementation.**

```
var remoteProxy = new Ext.data.ScriptTagProxy({
    url : 'http://extjsinaction.com/dataQuery.php'
});

var recordFields = ['firstname','lastname'];

var remoteJsonStore = new Ext.data.JsonStore({
    proxy      : remoteProxy,
    id         : 'ourRemoteStore',
    root       : 'records',
    autoLoad   : true,
    totalProperty : 'totalCount',
    remoteSort : true,
    fields     : recordFields
});

var columnModel = [
{
    header      : 'Last Name',
    dataIndex  : 'lastname'
},
{
    header      : 'First Name',
    dataIndex  : 'firstname'
}
];
```

Next, we'll create a generic handler for the MenuItem's and the implementation of our extension.

### **Listing 15.10 Implementing our extension.**

```
var onMenuItemClick = function(menuItem) {
    var ctxMenuGrid = Ext.getCmp('ctxMenuGrid');
    var selRecord = ctxMenuGrid.getSelectedRecord(); // 1
    var msg = String.format(
        '{0} : {1}, {2}',
        menuItem.text,
        selRecord.get('lastname'),
        selRecord.get('firstname')
    );

    Ext.MessageBox.alert('Feedback', msg);
};

var grid = {
    xtype      : 'contextMenuGridPanel', // 2
    ...
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

columns      : columnModel,
store        : remoteJsonStore,
loadMask     : true,
id           : 'ctxMenuGrid',
selModel     : new Ext.grid.CellSelectionModel(),
viewConfig   : { forceFit : true },
menu         : [
    {
        text      : 'Add Record',
        handler   : onMenuItemClick
    },
    {
        text      : 'Update Record',
        handler   : onMenuItemClick
    },
    {
        text      : 'Delete Record',
        handler   : onMenuItemClick
    }
],
},
{1} Using the extension's getSelectedRecord utility method.
{2} Configuring an XType object for our extension
{3} Inline array of MenuItem configuration objects.

```

In the above listing, we first create a generic handler for the `MenuItem`s that we're going to later configure. This will provide visual feedback that we've successfully clicked a `MenuItem`. Notice that it leverages the extensions' `getSelectedRecord`**{1}** utility method to gain a reference to the record selected by the right click event.

Next, we move on to create an `XType` configuration object for the extension by setting the generic object's `xtype` property to '`contextMenuGridPanel`', which is the string that we registered with `ComponentMgr`. All of the configuration options are common to the `GridPanel`. Notice that we're using the `CellSelectionModel` instead of the default `RowSelectionModel`. This gives us an opportunity to test both the `onCellContextMenu` event handler and the `getSelectedRecord` utility method to see if they can leverage the correct selection getter and setter methods.

The very last configuration item, `menu`, is an array of objects. Recall that our extension's constructor will create an instance of `Ext.menu.Menu` for us automatically if configured. This is the first time we get to really see how taking the time to generate such an extension can save time.

To render this on screen, lets wrap our extension in a `Window`.

```

new Ext.Window({
    height : 200,
    width  : 300,
    border : false,
    layout  : 'fit',
    items   : grid,
    center  : true
}).show();

```

To exercise our extension, all we need to do is right click on any one cell. This will invoke the display of the automatically instantiated Menu widget and select the cell that was right-clicked, exercising the `onCellContextMenu` event handler method. Next, click on any of the `MenuItem`s, which will invoke the generic `onMenuItemClick` handler. This will exercise the extension's `getSelectedRecord` utility method.

The contents of the selected record will be displayed in the `MessageBox` as coded in the `onMenuItemClick` handler method.

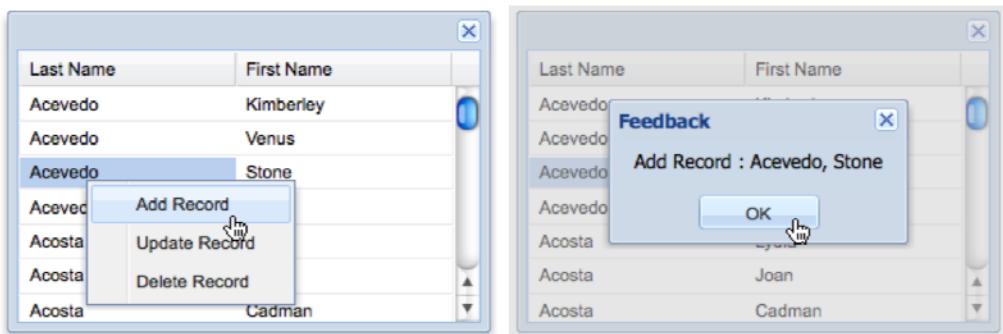


Figure 15.4 Our extension in action.

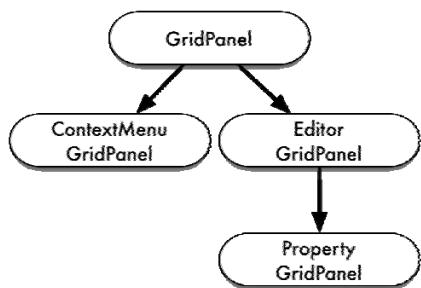
With this implementation of our extension, we didn't have to create an instance of `Menu` and setup an event handler to display the `Menu`. We also didn't have to configure the destruction of the `Menu` either. Our extension takes care of all of the dirty work for us simply only requiring the configuration an array of `MenuItem` configuration objects.

We've just seen our extension in action. Clearly, we solved a problem where code duplication would occur often in a large application. The extension solves this problem and automates a few of our actions for us. While our extension solved the problem of code duplication, it does have some limitations that may not be apparent, but are extremely important to understand.

### 15.2.5 Identifying the limitations of extensions

For a moment, put yourself into this situation: You're building application that requires the `GridPanel`, `EditorGridPanel` and `PropertyGridPanel`. You're required to attach context menus to each of these types of widgets. We already made an extension to `GridPanel` to make the task of attaching menus easier.

So here's the question: How do we get the same easy-menu functionality across the other types of grids? To put this into context, below is the `GridPanel` class hierarchy including our `ContextMenuGridPanel` extension.



15.5 The GridPanel class hierarchy with our ContextMenuGridPanel extension.

Looking at this diagram we see that the `EditorGridPanel` and `ContextMenuGridPanel` both extend `GridPanel`. How can we solve this problem. One solution would be to extend `EditorGridPanel` and `PropertyGridPanel` to carry over this functionality. That would make the class hierarchy look like this:

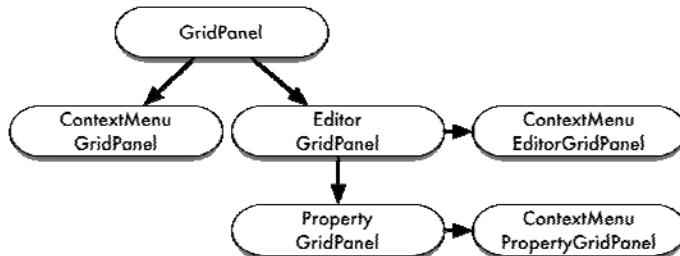


Figure 15.6 A proposed class hierarchy, where code duplication is possible.

To achieve the above solution, you could either elect to duplicate code, or perhaps even stitch together some type of cross-inheritance model. No matter what the solution, it's not going to be either elegant or useful.

The only real solution to this problem is a plugin.

### 15.3 Plugins to the rescue

Plugins solve this exact problem by allowing developers to distribute functionality across widgets without having to create extensions and were introduced in Ext JS version 2.0. Also what makes plugins powerful is the fact that you can have any number of them "attached" to a Component.

The basic anatomy of a plugin is simple and can be defined as a generic object with an `init` method:

```
var plugin = {
    init : function(parent) {
    }
}
```

If you can recall the initialization phase of the Component lifecycle, you will remember that at the end of the Component constructor execution any configured plugins are brought to life. The Component executes each plugin's `init` method and passes itself (`this`) as the only argument.

From a plugin's perspective, I like to think of the Component that a plugin is attached to as the plugin's parent. It is when a plugin's `init` method is executed that it is first aware of its parent Component and a crucial piece in time where the plugin can perform any work that may be required before a Component is rendered. An example of this is the attaching of event handlers.

Before we convert our extension to a plugin, I want to show you a much more flexible and powerful plugin design pattern.

### 15.3.1 A robust plugin design pattern

When I develop plugins, I use a more complex, but thorough, pattern compared to the simple example above. I do so because there are times where it is necessary to add methods to the parent Component itself. Also, it is our responsibility to code any cleanup actions, such as the destruction of any loosely coupled widgets.

Before we implement this pattern on a large scale, we should take a look at a skeleton and discuss it.

#### Listing 15.11 A thorough plugin design pattern

```
var plugin = Ext.extend(Object, {
    constructor : function(config) {
        config = config || {};
        Ext.apply(this, config); // 1
    },
    init : function(parent) {
        parent.on('destroy', this.onDestroy, this); // 2
        Ext.apply(parent, this.parentOverrides);
    },
    onDestroy : function() {
    },
    parentOverrides : { // 3
    }
});
Ext.preg('myPlugin', plugin);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

- {1} Apply any passed configuration to the instance of the plugin
- {2} Setup automatic cleanup when the parent is destroyed
- {3} Apply any overrides to the parent Component

As we can see, this plugin pattern is a bit more involving than the first, but is much more robust. We start out with the extension of Object, and have a full constructor, where the configuration properties passed are applied to the instance of the class{1}. This works *exactly* like most of the classes in the framework.

Next, is the init method, which automatically registers the local onDestroyMethod as a handler for the parent Component's destroy event. onDestroy is where you would want to place your plugin's cleanup code. Also, notice that we're applying the local parentOverrides{3} object to the parent Component. Having capability means that we can "extend on the fly", or add methods and properties to the parent Component. Of course not all plugins add members to the parent component, but this mechanism provides a means to do so if you need to.

**NOTE:**

All methods *applied* to the parent Component will execute with the scope of the parent Component.

In the last bit, we execute Ext.preg, which registers the plugin with Ext.ComponentMgr as what's known as a PType or plugin type and is exactly like XTypes. While this last step isn't completely necessary for a plugin to be used, it allows for lazy instantiation by the framework.

OK, we now know the basics of plugins. We can begin work to convert our extension to a plugin.

### 15.3.2 Developing a plugin

Our plugin needs to be able to do exactly what the extension was capable of. This includes the instantiation of a Menu and attaching the cellcontextmenu to the parent Component to be able to display the Menu. It also needs to manage the destruction of the Menu when the parent Component is destroyed. We can't leave out the getSelectedRecord utility method, which means we'll get to exercise the application of the parentOverrides object of our plugin design pattern.

A lot of the code we'll be assembling will come from the extension we created earlier. This means that we can move a bit faster, and focusing on how the plugin works. Below is the template for our plugin.

#### **Listing 15.12 The template for our plugin**

```
var GridCtxMenuPlugin = Ext.extend(Object, {  
    constructor : function(config) {
```

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

},
init : function(parent) {
},
onCellContextMenu : function(grid, rowIndex, cellIndex, evtObj) {
},
onDestroy : function() {
},
parentOverrides : {
    getSelectedRecord : function() {
}
}
);
Ext.preg('gridCtxMenuPlugin', GridCtxMenuPlugin);

```

In our plugin template above, we added the `onCellContextMenu` handler method, which just like in our extension, will handle the display of the menu and proper selection. We also added the `getSelectedRecord` utility method to the `parentOverrides` object. This will give us an opportunity to exercise adding a method to a parent Component on the fly.

OK, we have our template set. We can begin filling in the methods. We'll start with the constructor and the init methods.

### **Listing 15.13 Adding the constructor and init methods to our plugin.**

```

constructor : function(config) {
    config = config || {};
    Ext.apply(this, config);
},
init : function(parent) {
    this.parent = parent;
    if (parent instanceof Ext.grid.GridPanel) { // 1
        if (!(this.menu instanceof Ext.menu.Menu)) { // 2
            this.menu = new Ext.menu.Menu(this.menu);
        }
        parent.on({ // 3
            scope           : this,
            cellcontextmenu : this.onCellContextMenu,
            destroy         : this.onDestroy
        });
        Ext.apply(parent, this.parentOverrides);
    }
},
{1} Caching the parent reference
{2} Ensuring the plugin only works with GridPanels and descendants
{3} Attaching applicable event handlers

```

For the `constructor` method, we use the constructor from the pattern where we apply the passed configuration object to the instance of the plugin. This will be useful because when we implement this plugin, we'll be configuring the menu on the plugin itself.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

In the init method, we set a local `this.parent{1}` reference to the parent Component. This is useful for methods on the plugin itself that need to do something to or with the parent Component.

Next, there is an if block that tests to see if the parent Component is an instance of the `GridPanel` widget{2}. This type of test will only allow this plugin to work with `GridPanels` or any descendant thereof and is the type of control pattern to prevent the accidental use of plugins for widgets that they are not designed for.

Inside of this block, the creation of a Menu widget and registration of related event handlers will occur just like our extension. The difference being that a destroy handler is added to allow for the plugin to destroy the Menu widget.

OK, we have the first two methods filled in. Next, we'll fill in the last three.

#### **Listing 15.14 Adding the last three methods to our plugin.**

```
onCellContextMenu : function(grid, rowIndex, cellIndex, evtObj) {
    evtObj.stopEvent();
    if (grid.selModel instanceof Ext.grid.RowSelectionModel) {
        grid.selModel.selectRow(rowIndex);
    } else if (grid.selModel instanceof Ext.grid.CellSelectionModel) {
        grid.selModel.select(rowIndex, cellIndex);
    }
    this.menu.stopEvent(evtObj.getXY());
},
onDestroy : function() {
    if (this.menu && this.menu.destroy) {
        this.menu.destroy();
    }
},
parentOverrides : {
    getSelectedRecord : function() {
        if (this.selModel instanceof Ext.grid.RowSelectionModel) {
            return this.selModel.getSelected();
        } else if (this.selModel instanceof Ext.grid.CellSelectionModel) {
            var selectedCell = this.selModel.getSelectedCell();
            return this.store.getAt(selectedCell[0]);
        }
    }
}
}
```

The first of the last three methods, `onCellContextMenu` will be called within the scope of the instance of the plugin as registered in the `init` method. It works almost exactly like the similarly named method on our extension, except it references the parent Component via the first argument.

The `onDestory` method too will execute within the scope of the instance of the plugin and will destroy the menu accordingly. The `parentOverrides` object contains the `getSelectedRecord` utility method that is applied to the parent Component by the plugin's `init` method. Remember that this method will execute within the scope of the parent Component.

© Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

The construction of the plugin is now complete. It's time to put it into action.

### 15.3.3 Our plugin in action

To exercise our plugin, we're going to construct a `GridPanel` that will use a lot of the same code from the previous `GridPanel` implementation. The difference being that we'll configure the plugin, which will contain the Menu configuration. Replacing the menu configuration on the `GridPanel` will be the plugin.

We'll begin by creating the data Store. A lot of this is repeat code, so we'll be moving relatively fast.

#### **Listing 15.15 Constructing the data Store.**

```
var remoteProxy = new Ext.data.ScriptTagProxy({
    url : 'http://tdgi/dataQuery.php'
});

var recordFields = ['firstname','lastname'];

var remoteJsonStore = new Ext.data.JsonStore({
    proxy      : remoteProxy,
    id         : 'ourRemoteStore',
    root       : 'records',
    autoLoad   : true,
    totalProperty : 'totalCount',
    remoteSort  : true,
    fields     : recordFields
});
```

Next, we'll create the generic `MenuItem` handler and configure a plugin.

#### **Listing 15.16 MenuItem handler and plugin**

```
var onMenuItemClick = function(menuItem) {
    var ctxMenuGrid = Ext.getCmp('ctxMenuGrid');
    var selRecord = ctxMenuGrid.getSelectedRecord();
    var msg = String.format(
        '{0} : {1}, {2}',
        menuItem.text,
        selRecord.get('lastname'),
        selRecord.get('firstname')
    );
    Ext.MessageBox.alert('Feedback', msg);
};

var ctxMenuPlugin = {
    ptype : 'gridContextMenuPlugin',
    menu : [
        {
            text : 'Add Record',
            handler : onMenuItemClick
        },
        {
            text : 'Update Record',
        }
    ]
};
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

```

        handler : onMenuItemClick
    },
    {
        text : 'Delete Record',
        handler : onMenuItemClick
    }
]
};


```

Along with the creation of the generic MenuItem handler, we configure the plugin using a PType configuration object. Ext JS will use this to lazy-instantiate an instance of the plugin we created and registered with the string "gridCtxMenuPlugin". Again, this is just like XTypes for Components.

Lastly, we'll create the GridPanel that will use the plugin and display it on screen.

### **Listing 15.17 Configuring and showing the GridPanel**

```

var columnModel = [
    {
        header : 'Last Name',
        dataIndex : 'lastname'
    },
    {
        header : 'First Name',
        dataIndex : 'firstname'
    }
];

var grid = {
    xtype : 'grid',
    columns : columnModel,
    store : remoteJsonStore,
    loadMask : true,
    id : 'ctxMenuGrid',
    viewConfig : { forceFit : true },
    plugins : ctxMenuPlugin
};

new Ext.Window({
    height : 200,
    width : 300,
    border : false,
    layout : 'fit',
    items : grid,
    center : true
}).show();

```

In the above listing, we configure a GridPanel XType object that uses the ctxMenuPlugin that we configured earlier and display it via an Ext.Window. In this implementation, we only configured one plugin via the plugins reference. If we had more than one, we would configure an array of plugins such as `plugins : [ plugin1, plugin2, etc ]`.

Rendering this on screen, we can see that we have the exact same functionality as our GridPanel Extension.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>

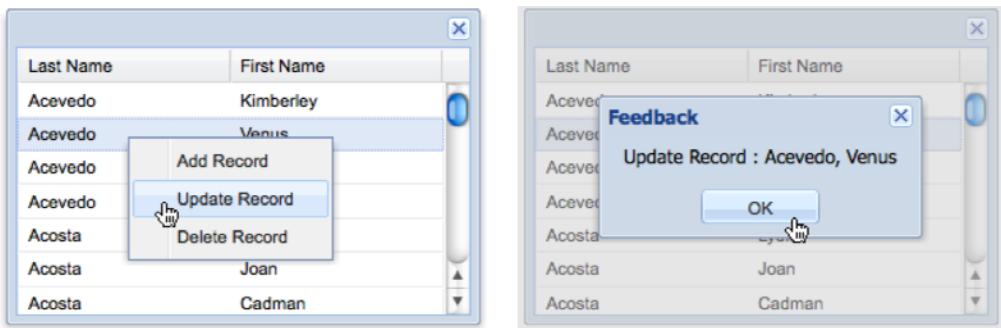


Figure 15.7 Our first Plugin in action.

If you want to read the source code for other plugins, you can look at the Ext JS SDK examples/ux directory, which has a few examples of plugins. Using the pattern that we just implemented, I contributed two plugins.

The first is known as the TabPanel scroller menu (`TabScrollerMenu.js`), which adds a menu to scrolling TabPanels, allowing users to select and focus a tab panel much easier than having to scroll. To see this plugin in action, simply navigate to the `<your ext js dir>/examples/tabs/tab-scroller-menu.html` url in your browser.

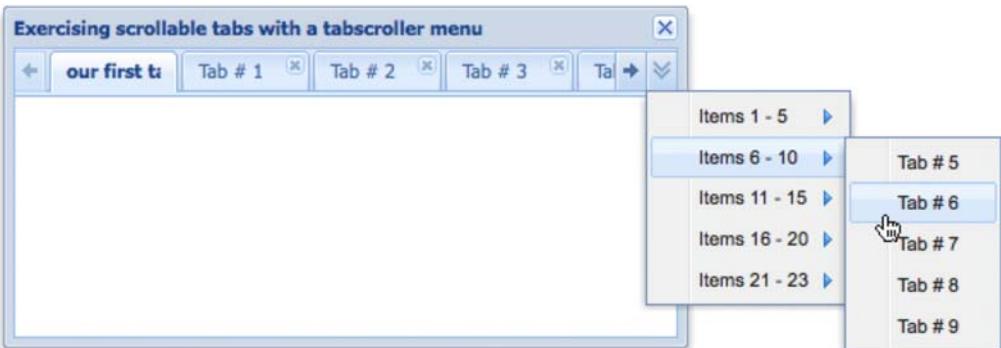


Figure 15.8 Thetabpanel scroller menu in action.

The second is known as the ProgressBar PagingToolbar (`ProgressBarPager.js`), which adds an animated progress bar to the PagingToolbar widget, making the paging toolbar much nicer to look at. To view this plugin in action, point your browser to `<your ext js dir>/examples/grid/progress-bar-pager.html`.

Sliding Pager				
Company	Price ▲	Change	% Change	Last Updated
Intel Corporation	\$19.88	0.31	1.58%	09/01/2009
Microsoft Corporation	\$25.84	0.14	0.54%	09/01/2009
Pfizer Inc	\$27.96	0.4	1.45%	09/01/2009
Alcoa Inc	\$29.01	0.42	1.47%	09/01/2009
General Motors Corporation	\$30.27	1.09	3.74%	09/01/2009
AT&T Inc.	\$31.61	-0.48	-1.54%	09/01/2009
General Electric Company	\$34.14	-0.08	-0.23%	09/01/2009
The Home Depot, Inc.	\$34.64	0.35	1.02%	09/01/2009
Verizon Communications	\$35.57	0.39	1.11%	09/01/2009
Hewlett-Packard Co.	\$36.53	-0.03	-0.08%	09/01/2009

Page 1 of 3 | | | | Displaying 1 - 10 of 29

15.9 A plugin that adds an animated ProgressBar to the PagingToolbar.

This concludes the exploration of plugins. By now, you have the necessary basics to create plugins that can enhance functionality in your projects. If you have an idea for a plugin and are not too sure if it's been done before, visit us at the Ext JS forums via the URL <http://extjs.com/forum>. There is an entire section dedicated to user extensions and plugins, where fellow community members have posted their work, some of which are completely for free use.

## Summary

In this chapter, we learned how to implement the prototypal inheritance model using the basic JavaScript tools. During that exercise we got to see how this inheritance model is constructed step by step. Using that foundational knowledge, we refactored our subclass using the `Ext.extend` utility method and implemented the two most popular extend patterns.

Next, we took all of our foundational knowledge and applied it to the extension of an Ext JS `GridPanel` and created a composite `GridPanel` and `Menu` component. This gave us an opportunity to discuss which vector to extend from, `constructor` or `initComponent`, and learned how extensions can assist with reusability in our applications.

Lastly, we analyzed how extensions can be limited when reusability needs to span across multiple widgets. We got to see this in action when we studied how the added features found in our newly created `GridPanel` extension could not be easily applied to descendants of the `GridPanel`, such as the `EditorGridPanel`. In order to mitigate this problem, we converted the code in our `GridPanel` extension and converted it to a plugin that could be applied to a `GridPanel` or any descendant thereof.

Next, we'll learn how to use all of the knowledge we've gathered thus far in this book and learn the trade secrets of building complex applications.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=525>