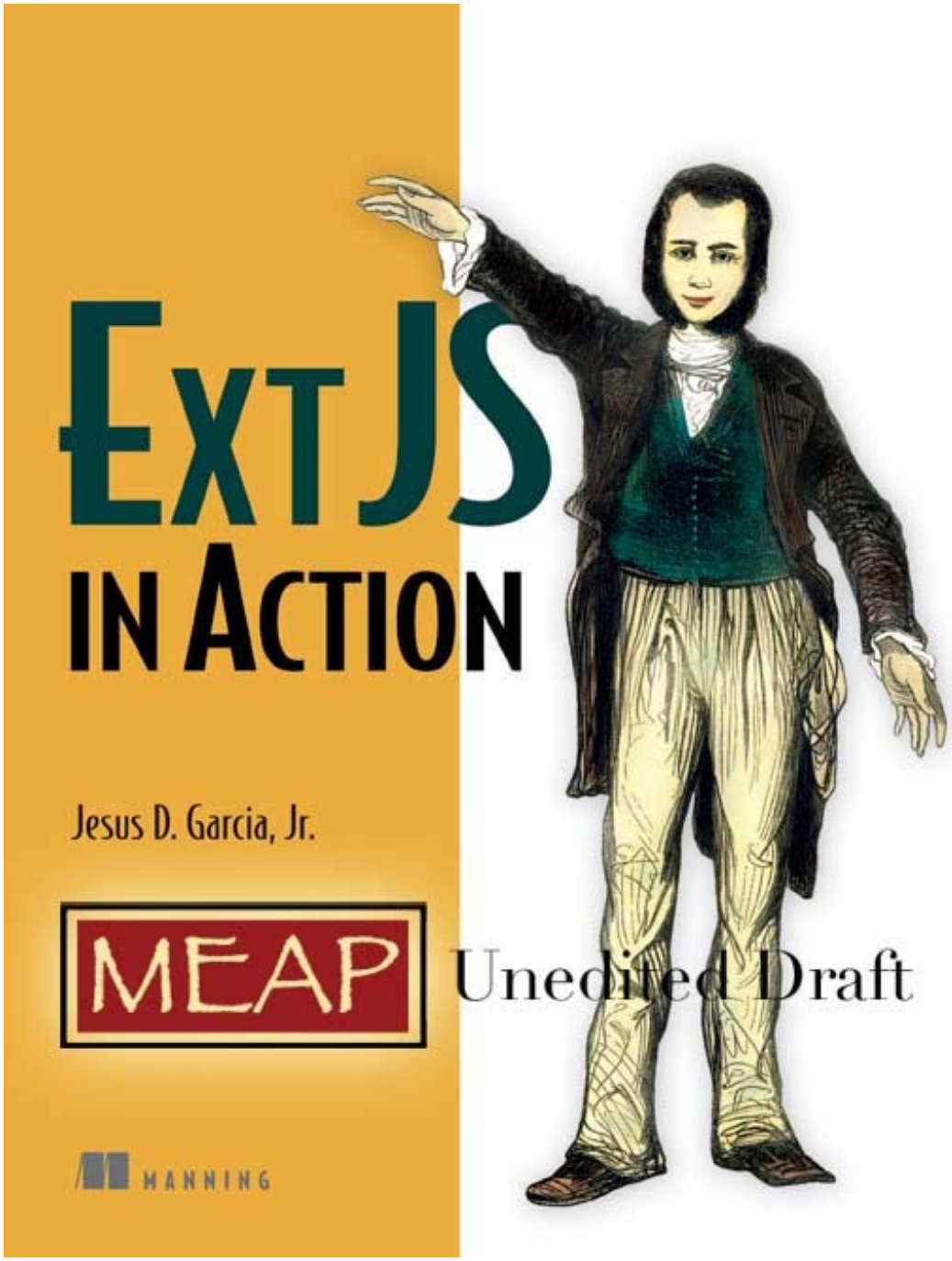


MEAP Edition
Manning Early Access Program

For more information on this and other Manning titles go to
www.manning.com

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 popup 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.

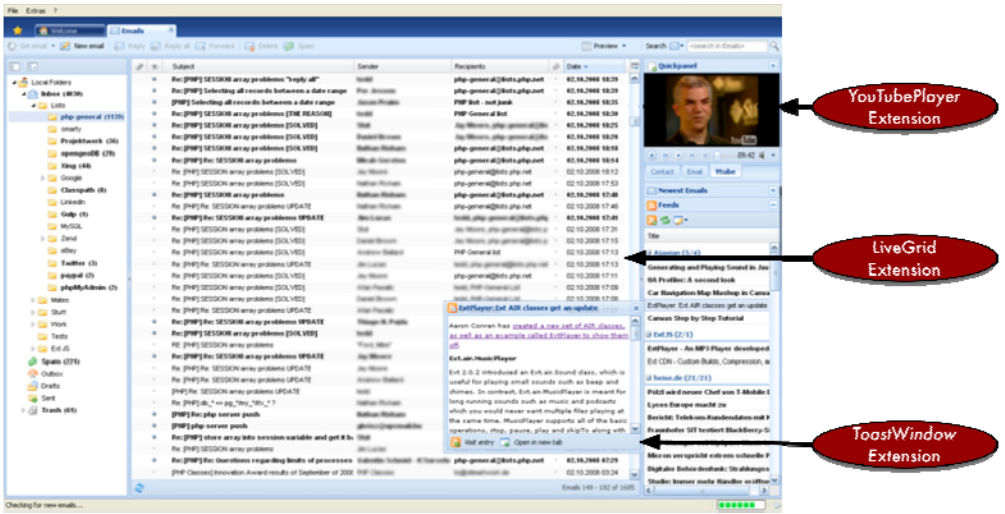


Figure 1.1 Conjoon is an open source personal information manager that is a great example of a web application that leverages the framework to manage a UI that leverages 100% of the browser's viewport. You can download it at http://conjoon.org/ .

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`.

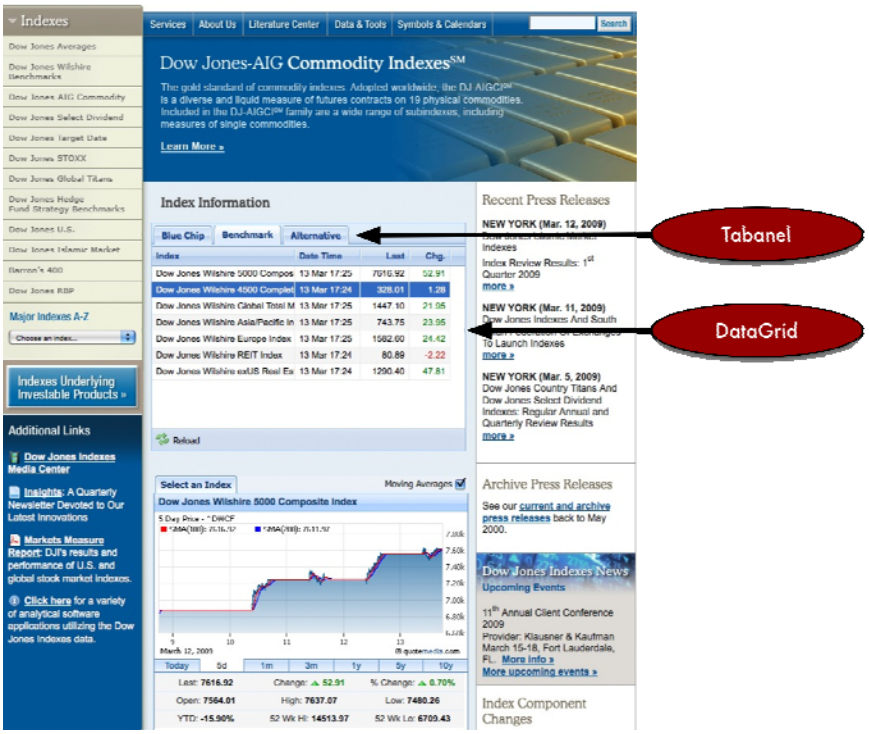


Figure 1.2 The Dow Jones indexes web site, http://djindexes.com, is one example of Ext JS embedded in a traditional Web 1.0 site.

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 1.2), 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.

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

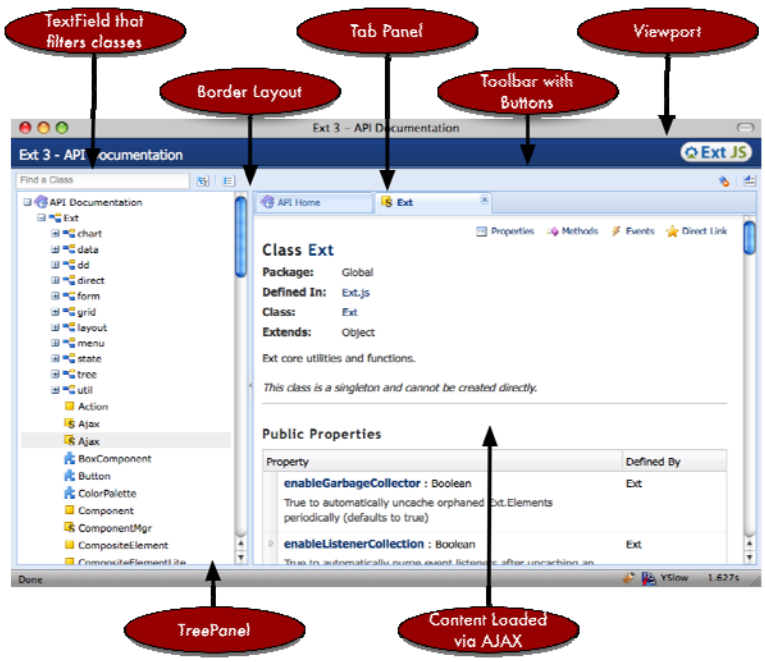


Figure 1.3 The Ext API documentation contains a wealth of information and is a great resource to learn more about components and widgets. It contains most of what you need to know about the API including constructor configuration options, methods, events, properties, component hierarchy and more.

The API documentation tool is choc-full of gooey GUI goodness and 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 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.

Okay, so the documentation is really good. What about rapid application development? Can Ext accelerate our development cycles?

1.1.3 Rapid development with prebuilt widgets

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, I think 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 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. Below is an illustration of the six areas of purpose.

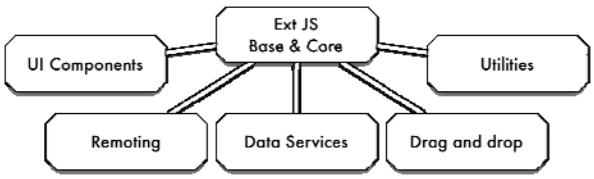


Figure 1.4 The six areas of purpose for Ext Classes; Core, UI, Remoting, Data Services, Drag and Drop and Utilities.

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 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 serve, 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 complex topic. We'll spend an entire chapter on this subject alone in chapter 11.

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, lets 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 4 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.

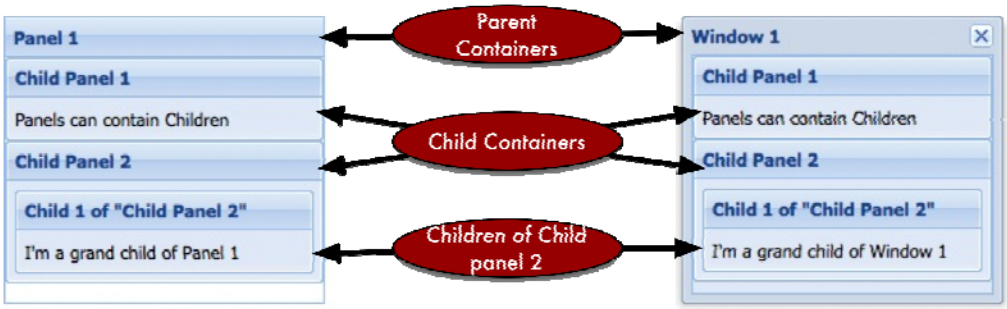


Figure 1.5 Here, we see two parent Containers, Panel (left) and Window(right), managing child items, which includes nested children.

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 contains 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`.

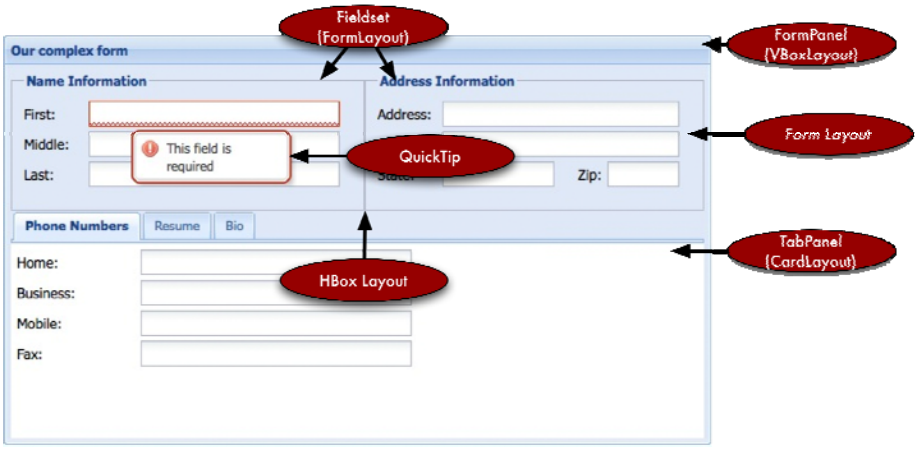


Figure 1.6 Commonly used descendants of Container, the FormPanel, TabPanel, FieldSet and Quick Tip and the layouts used to compose this UI Panel. We will build this in Chapter 5, where we learn about forms.

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 5, 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.

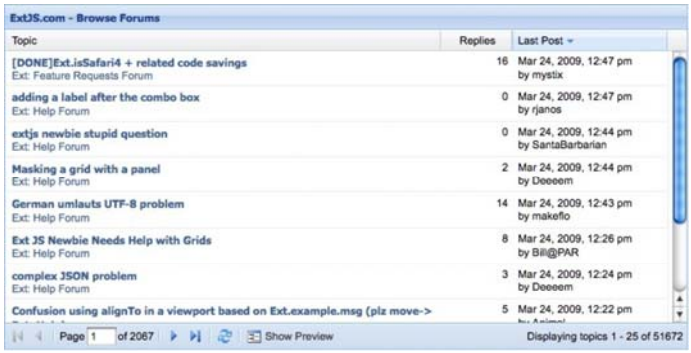


Figure 1.7 The Data Grid as see in the "Buffered Grid" Example in the Ext SDK.

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.

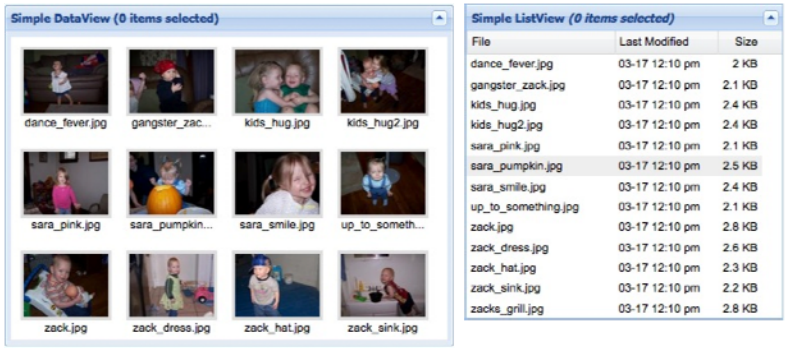


Figure 1.8 The DataView (left) and ListView (right) as seen in the Ext SDK Examples.

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

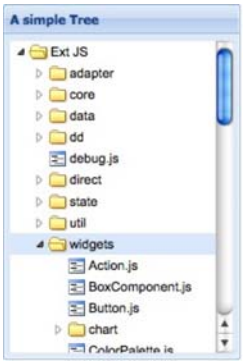
DOM utility that allows you to create a template, with placeholders for data elements, which can be filled in by individual records in a store and stamped out on the DOM. In Figure 1.8, the `DataView` (left) is displaying data from a store, which includes references to images. It uses a pre-defined template, 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 picture 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.

1.3.5 Form Input fields

Ext has a palette of eight input fields from which to use. They range from simple `TextFields`, 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.

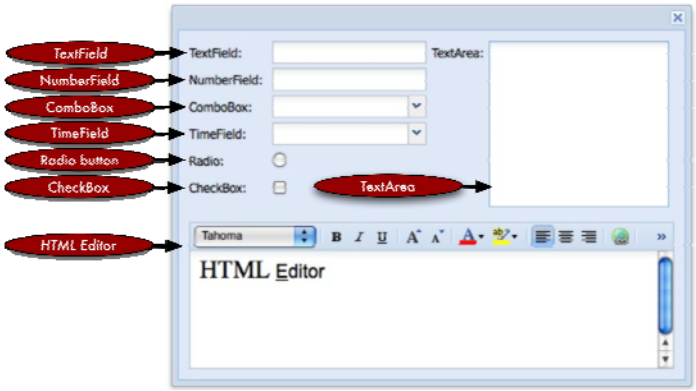


Figure 1.10 All of the out of the box form elements displayed in an encapsulating Window.

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 feedaback 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.

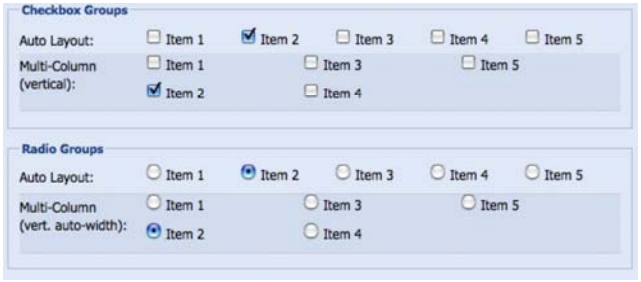


Figure 1.11 An example of the Checkbox and Radio Group convenience classes in action with automatic layouts.

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 5. But for now, lets 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 `ListBox`. The following figure is an example of a custom `ComboBox` in action.

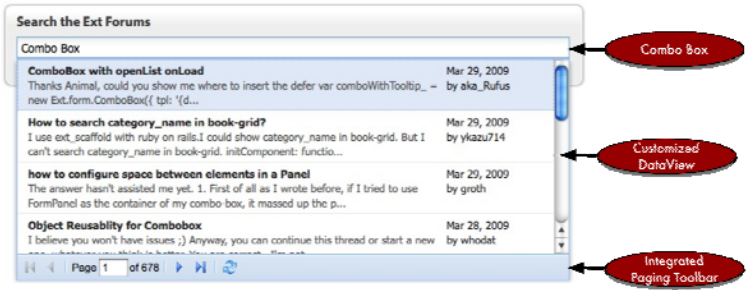


Figure 1.12 A Custom ComboBox, which includes an integrated Paging Toolbar, as seen in the downloadable Ext Examples.

In Figure 1.12, a custom combo box is being leveraged to search the Ext forums. The `CombBox` 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. Lets look at the illustration below and discuss what these items are and do.

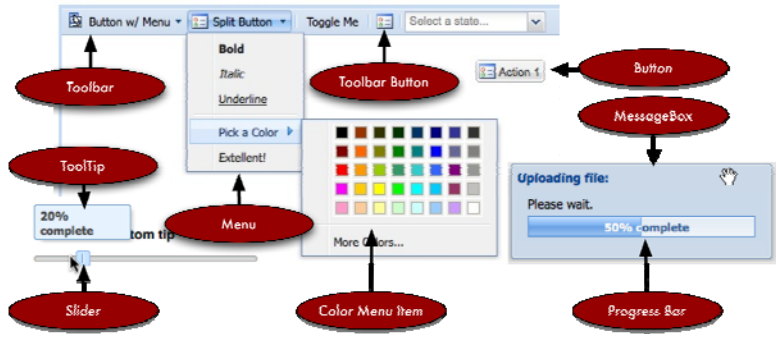


Figure 1.13 A collection of miscellaneous UI widgets and controls.

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

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. Lets 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

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.

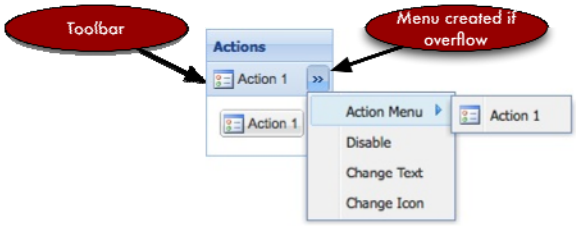


Figure 1.14 The new ToolbarLayout is responsible for detecting the toolbar's size and creating menu stubs if menu items were to overflow.

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.

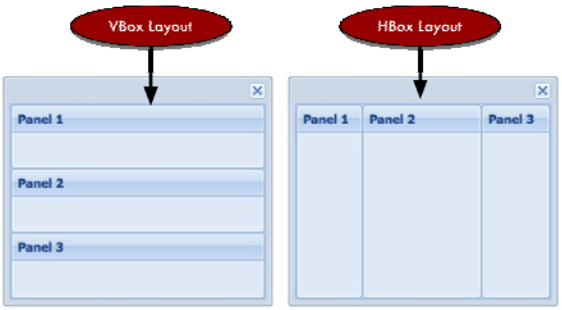


Figure 1.15 An example of the VBox and HBox layouts in action.

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 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.

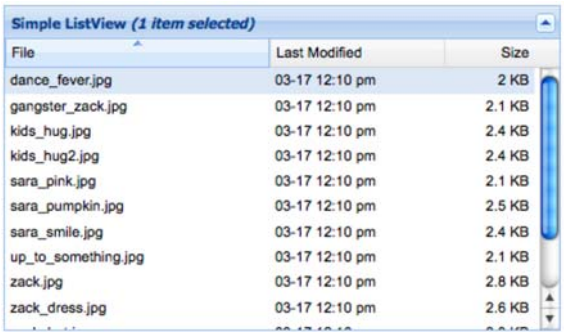


Figure 1.16 The new Ext.ListView class, which is like a very light weight DataGrid.

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 thing 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.

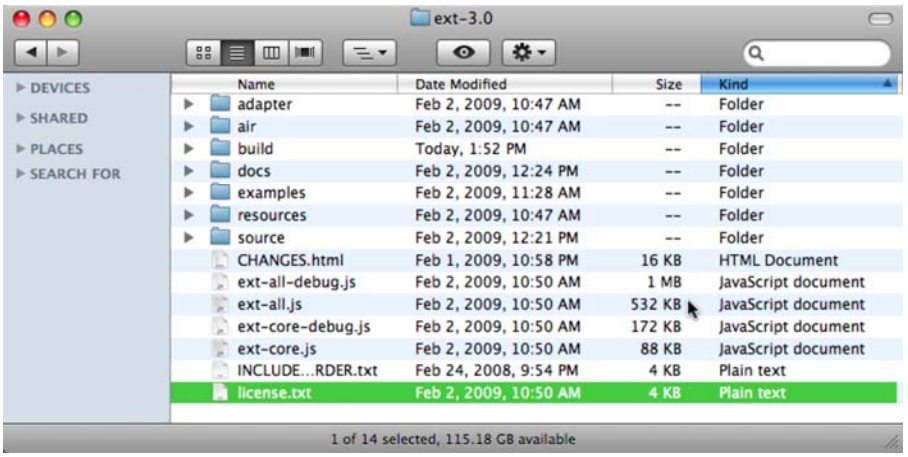


Figure 1.18 A view of the Ext JS SDK contents.

Looking at the contents of the SDK, we see a lot of stuff. The reason there are so man 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. The following table explains what each of the folders are and what they do.

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. I think now is a good time to talk about how to setup 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"
     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">
</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 be come an issue. This also becomes a problem if you are using SSL, where sis 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, after all of this discussion, you're probably itching to start using Ext JS. So, what are you waiting for? Let's go ahead and 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">                                        // 3
     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({                                    // 1
      id       : 'myWindow',
      title    : 'My first Ext JS Window',
      width    : 300,
      height   : 150,
         layout   : 'fit',
         autoLoad : {                                             // 2
         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 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.

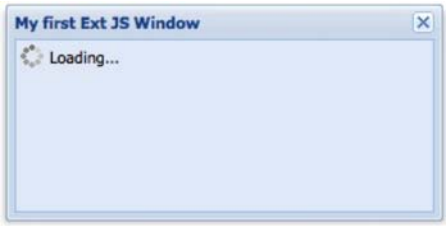


Figure 1.19 Our first Ext JS window attempting to load content via AJAX.

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() {                                    /* 2
*/
    var win     = Ext.getCmp('myWindow');
    var winBody = win.body;
    winBody.highlight();
  }

  highlightWindow.defer(1000);                                    /* 3
*/
</script>
```

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 `higlightWindow` **{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 **{3}**. 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.

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. Other reasons to use it is because it allows us to control the scope from which the deferred method is being executed and pass custom parameters, which `setTimeout` lacks.

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.

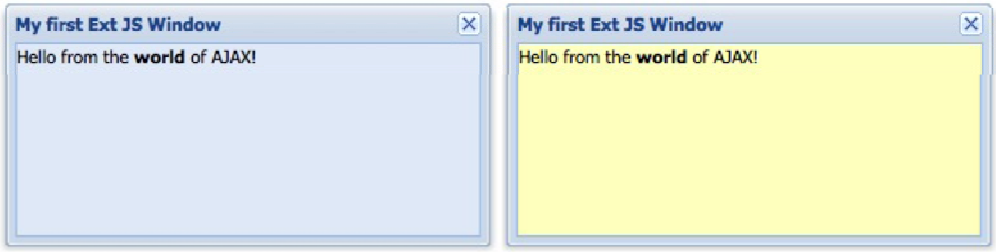


Figure 1.20 Our Ext JS Window loading the our HTML fragment (left) and the highlight effect performed on the Window's content body(right).

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'd like to 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

An Ext JS Primer

One of the largest issues newcomers to the Ext JS camp face is the understanding the core of the framework and related concepts. Many issues are posted by developers in the community forums, which revolve around fundamental concepts such as layout models, data store loading, component initialization and event management. What I found is that very often we get excited to toy with the code in the examples and are so eager to build applications with our newly discovered toolset that we tend to overlook the important information. While this excitement is a great to have, it needs to be tamed with discipline.

This chapter will enable you to better understand the core fundamental concepts of Ext JS, which will plant the seeds required for you to spring up a better understanding of why things in the framework work and behave the way they do. Here is where you will learn how to launch Ext Components, ensuring that everything the framework requires to build its UI is available. We will look into the heart of Ext, known as the Ext.Element class, which is a robust cross-browser DOM element management suite. Next, we will discuss and exercise Observable, which is Ext's core event management utility providing us the ability to register event and event handlers on DOM elements and framework Components. Lastly, we will dissect the Ext Component lifecycle, which is an elaborate component start to finish model, providing a unified approach for components to be managed.

2.1 Starting off the right way

When most developers want to initialize their JavaScript, they typically add an onLoad attribute to the body tag of the html page that is to be loaded:

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

While this method of invoking JavaScript works, it's not ideal for AJAX enabled Web 2.0 sites or applications because the onLoad event is generally fired when the DOM is ready and all content has been loaded and rendered by the browser. Web 2.0 code generally wants to start managing and manipulating DOM elements when the DOM is ready but before any images are loaded.

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 Ext JS pulls 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`.

Here is a simple example of using `Ext.onReady` to fire up a window:

Listing 2.1 An Example Ext.onReady call

```
Ext.onReady(function() {
        var panel = new Ext.Window({
                title  : 'Test Window',
                width  : 100,
                height : 100,
                frame  : true,
                html   : 'I love Ext JS!'
        });
        panel.show()
});
```

In the preceding example, we pass an anonymous function to Ext.onReady as the only parameter. Moving forward, if `Ext.onReady` is not explicitly detailed in example code, please assume that you must launch your code with it. Now that we know how to launch our Ext widgets properly, we need to explore how we can configure them so they can interact with the users using event management.

2.2 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. This is one of the core blocks of knowledge from which you will build your knowledge of the framework, which will aid you in developing user interfaces, which provide truly rich user interaction. Knowing how Ext components communicate with events is equally as important.

2.2.1 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'.

2.2.2 Events at a glance

To further elaborate on how events work on a very high level, we'll take a look at a simple user interaction, which is opening an item on your OS desktop. Let's think about the steps:

After you locate the item on the desktop visually, you move your mouse to hover over the icon. The act of moving your mouse is firing events, which changes the mouse X and Y coordinate values, causing the OS to update the mouse cursor icon's position on the screen.

Next, you double click the icon, which causes the OS to actually fire a 'double click' event for which the desktop manager is already listening for. The desktop double click 'listener' method catches and 'handles' the event by performing an action, which is opening the item that is double clicked.

Events also come from your keyboard. For every key that is hit, an event is fired and something occurs on the screen, such as text being displayed, a menu being shown or modifying application behavior.

This model works very well for desktops. What about the web browser? How do events play a role there?

2.2.3 Events in the browser

Just like the OS event paradigm, the browser too fires events with user interaction. This powerful model allows us to leverage typical user input from humans and perform complex tasks, such as refreshing a grid or filtering a list. Just about every interaction that is

performed with the browser fires events that can be leveraged. This includes clicking on an element, scrolling, typing and even browser window resizing.

2.2.4 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!

The way events are managed across browsers is added to our ever grow list of cross-browser incompatibilities. For instance, here is the W3C approved way of attaching an event listener to an element:

```
var el = document.getElementById('myDiv');
el.addEventListener('click', doSomething, false);
```

Browsers that closely follow the W3C standards like Firefox, Safari and Opera use the method above, where a referenced element's `addEventListener` method can be called. You pass in the native browser event, a reference to a function and a `boolean` which represents whether to fire the event in capture or bubble mode. Here is the Microsoft way:

```
var el = document.getElementById('myDiv');
el.attachEvent('click',doSomething);
```

Microsoft's implementation of adding an event listener does not allow for events to be 'captured' as Internet Explorer only facilitates event bubbling. Having to write and manage your own code for detecting browser type and applying proper code could easily get messy.

Luckily for us, Ext JS takes care of that and presents a unified interface for us to leverage:

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

Here, we use the native Ext 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 ever-important 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 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 method handling 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`.

Now that have learned and exercised simple event handling on an Element, let's explore how user generated events flow in the DOM.

2.2.5 Event flow in the DOM

Event capture and bubbling are terms that describe how the events travel through the DOM tree. As depicted in Figure 2.1, capture is when the event travels down from the document down the DOM to the source element, and bubbling is when the event flows up from the source element up the DOM to the document. Recall that Netscape and Microsoft had two completely separate approaches with regard to event flow direction. Understanding the how each works is key to preventing event handler duplicate calls or misfires.

Let's say you have an event listener for both the `div` and `anchor` elements, which respond to the click event and both listeners perform tasks completely independent of each other. In event capture, when a user clicks on the anchor tag, the event flows down the entire DOM tree to the element that was clicked and the click handler for the `div` would be fired first and then fired for the `anchor` element.

What if you didn't want the `div`'s event handler to fire when its child `anchor` tag is clicked? You would have to condition the `div`'s click event handler to inspect the event's `currentTarget` attribute and perform its action based on which node is described in that property.

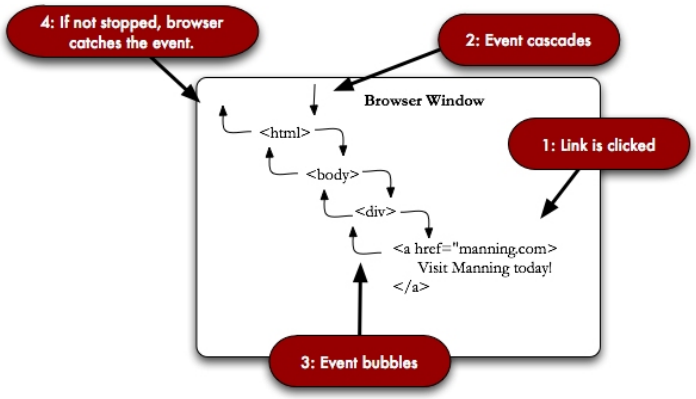


Figure 2.1 Here, we visualize an event flowing down the DOM (capture) and then returning up the tree (bubbling).

Event bubbling is exactly the inverse of capture, where the event's source is the node from which the user action is performed to the `document` object, which performs a default action based on what type of node was clicked.

Let's revisit the prior scenario where you have separate click listeners for the div and anchor elements. This time, we'll demonstrate with code. Suppose you have the following HTML and you want to apply separate click listeners to both the div and the anchor.

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

We will register a click handler 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('From myDiv, source id = ' +  elRef.id);
});
```

After rendering the page expand firebug and click on the anchor. You'll see a firebug console message indicating that you clicked the `myHref` tag. Hold on… we only assigned a

listener to the anchor's parent element, 'myDiv'. Please refer to figure 2.1 again. The reason you see the event get fired is because the event bubbled up from the anchor element to its parent.

Now, let's attach a click handler to the anchor itself:

```
Ext.get('myHref').on('click', function(eventObj, elRef) {
    console.log('From myHref, source id = ' + elRef.id);
});
```

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

```
From myHref, source id = myHref
From myDiv, source id = myHref
```

Having both event listeners fired could be troublesome and considered a waste of resources if only one is needed. Let's explore how we can stop an event from bubbling back up, preventing an event misfire.

2.2.6 Burst the bubble

To prevent both events from firing, 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) {
    eventObj.stopEvent();
    console.log('From myHref, source id = ' + elRef.id);
});
```

Refresh the page and click the anchor again. This time, you'll only see one firebug console message per click. 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.

Let's modify our example to perform this task:

```
Ext.get('myDiv').on('contextmenu', function(eventObj, elRef) {
    console.log('From myDiv, source id = ' +  elRef.id);
});

Ext.get('myAnchor').on('contextmenu', function(eventObj, elRef) {

eventObj.stopEvent();
    console.log('From myAnchor, source id = ' + elRef.id);

new Ext.menu.Menu({
```

```
            items : [
                    {
                            text : "Custom Context Menu"
                    }
            ]

        }).show(elRef);
    });
```

We change the event label from 'click' to 'contextmenu' on both event handler registrations. In the anchor event handler, we stop the event propagation, log the id of the element from which the event handler is referencing in the firebug console and create and display a new context menu.

When you right click on the div, you'll see the default browser context menu. However, if you right click on the anchor, 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.

2.2.7 Components have events too

Almost every Ext widget and component has custom events that it fires when it deems necessary. This is due to the fact that just about everything extends `Ext.util.Observable`, which gives the widgets the ability to exhibit similar behavior to complex UI environments such as your desktop. 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`.

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

2.2.8 Registering Event names

Before an Ext-based event can be fired, it must be added to the list of events for that component. 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 myComponent = new Ext.util.Observable();

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

If you want to register more than one event, you can pass one event per argument: `myComponent.addEvents('sayhello', 'saygoodbye');`

Or pass in an object that has a list of event labels and if they should be enabled by default:

```
myComponent.addEvents({
```

```
    'sayHello'   : true,
    'sayGoodbye' : true
  });
```

2.2.9 Registration of Event Listeners

Now that we have our events registered, we need to register an event handler:

```
  myComponent.addListener('sayHello', function() {
     console.log('hello');
  });
```

Here, we are providing an event handler for our custom event, 'sayHello'. 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:

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

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 a new event handler for a custom 'goodBye' event. This time, create the event handler so that it accepts two parameters, `firstName` and `lastName`:

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

Here, we define a method named sayGoodbyeFn, which accepts the two parameters. We call myComponent.on, which is shorthand for myComponent.addListener to register the event handler. Let's fire the 'sayGoodbye' event with a first and last name:

```
  myComponent.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 like:

```
  Goodbye John Smith!
```

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:

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

Here, we call myComponent.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.

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

Managing events and event listeners is as straightforward as described. Being that we now have a solid grasp on event management, let's dive into managing elements in the DOM with the ever-powerful Ext.Element class.

2.3 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 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.

2.3.1 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 widget in the framework and can be generally accessed by the `getEl` method or the `el` 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" and provide the robust UI that we have come to love. 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 the DOM elements, but rather perform complex tasks such as manage dimensions, alignments and coordinates with relative ease. You can also easily update the element via AJAX, manage child nodes, animate, full event management and so much more.

2.3.2 Using Ext.Element for the first time

Use of the `Ext.Element` is extremely easy and makes some of the hardest tasks simple. Using our prior knowledge of event listener registration, we will apply a click event handler to a referenced element, which will apply an animated resize effect to it. The first thing we need to do is globally set CSS for all div tags, which will size them and place a visible border around them:

```
div {
    border  : 1px solid #AAAAAA;
    width   : 200;
    height  : 35;
    cursor  : pointer;
    padding : 2px 2px 2px 2px;
    margin  : 2px 2px 2px 2px;
}
```

You'll need a div with the id of 'div1', in the body of your HTML document similar to the following:

```
<div id='div1'>Click to highlight</div>
```

Next, we'll create a reference to the element using `Ext.get`, which locates our HTML element in the DOM by a unique identifier by using `document.getElementById` and wrapping the `Ext.Element` class around it:

```
Ext.get('div1').on('click', function() {

  var newWidth  = this.getWidth() + 20;
  var newHeight = this.getHeight() + 20;
  this.setSize(newWidth, newHeight, true);

});
```

This should look very familiar to you, as it is similar to the click event handler registration code in the event management section. Here you see the use of the `this` keyword, which is a reference to the object from which a method is being called in. Here pop quiz: What is the scope from which the event handler is called? Feel free to reference the event management section if the answer does not immediately come to you. I will wait. Ready? If you came up to the conclusion that the scope is the object from which the event was fired, you're correct.

Setting dimensions is just a single facet of the many sides of element management with the Element class. Let's explore how we can append and remove child elements to a particular DOM node.

2.3.3 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 children.

First, we'll add a div to our document with the id of 'div5':

```
<div id='div5' class="myDiv">
     <div>
            div5 has 1 child node, me.
     </div>
</div>
```

This 'div5' has one child node, which is another div. Next, let's create the click handler that will create the children:

Listing 2.2 Creating a child node upon element click

```
Ext.get('div5').on('click', function() {
     var myChildDivs = this.select('div');
     var numChildren = myChildDivs.elements.length

     if (numChildren < 3) {
            this.createChild({
                   tag  : 'div',
                   html : 'Child ' + (numChildren + 1)
            });
     }
     else {
            this.mask('No more children please!');
     }
})
```

Inside the click handler, we use the method select, which allows us to query all descendants for particular properties. In this case, we're simply using a simple 'div' selector, which will detect all child divs. We reference myChildDivs to the results of the `select` call,

which returns an instance of `Ext.CompositeElement` containing a list of all elements found. We then reference numChildren to the length of the list's elements array. With that number, we try to determine if the number of children is less than two. If it is, we create a child for div5 using the this.createChild. If the number of children exceeds 2, then we use the mask method, which will mask the element, preventing further mouse user interaction with that element.

Element.mask works by adding a child div inside of the element that is to be masked, which takes 100% of its parent element's height and width. The mask div also has an extremely high z-index, which ensures that it sits above all of the other sibling elements. It is because of this high z-index property that it can capture all clicks. One important thing to note, however, is that the mask prevents clicks, but can be easily bypassed by a user pressing the tab key on their keyboard.

2.3.4 Removing Child Nodes

Removing nodes is much easier than adding. All you need to do is pass a reference of the node to be removed to the Ext.element's remove method and presto - node gone! Let's explore how we can do this. We need to create some reference Elements on our page:

```
<div id='div6' class="myDiv">
    <div>Child 1</div>
    <div>Child 2</div>
    <div>Child 3</div>
</div>
```

Next, let's create our click handler, which will help us remove elements on demand:

Listing 2.3 Removing child nodes upon element click

```
Ext.get('div6').on('click', function() {
    var myChildDivs = this.select('div');
    var numChildren = myChildDivs.elements.length;
    var lastChild   = this.child('div:last-child');

    if (numChildren > 1) {
           lastChild.remove();
    }
    else {
           lastChild.update('Please keep me!');
    }
});
```

This example is similar to the example where we create child nodes. We locate the last child for div6 by calling `this.child` and pass a selector, which is a shortcut for `Ext.DomQuery.selectNode`. Using the passed selector string, `selectNode` queries the DOM for the last child `div` element for `div6`. If a DOM node is found by `selectNode`,

Ext.Element wraps the located DOM node with another instance of Ext.Element, and returns it.

If the number of children is greater than one, then we remove the last child, else we will update the body of the last child with the string `'Please keep me!'`

2.3.5 Applying effects

Just about every AJAX-enabled application employ effects in their UI. These effects could be simple fade in/out, resize or even movement of UI elements. While the Ext UI does not heavily use effects, its core does contain methods that allow us to use them at our discretion.

`Ext.Element` makes the task of applying effects extremely simple. This is due to the fact that the `Ext.fx` class is applied to the prototype of `Ext.element`, which makes effect management a snap. Let's see how we can leverage `Ext.Element` to apply effects to elements. We need to create our element first:

```
<div id='div7' class="myDiv">
   Click to fade me out
</div>
```

Next, let's apply a click handler to our element, which will apply a `fadeOut` effect:

```
Ext.get('div7').on('click', function() {
    this.fadeOut();
});
```

We call `this.fadeOut` in the click handler, which fades out the element. Simple - right? There is one issue though, our element faded out, but did not reappear.

With the effects engine, you can specify parameters to control the behavior of the effect method being called. You can also include a callback method, which gets called (executed) after the effect method has completed its job. Let's modify our example to use a callback, which fades our item back in after a half second:

```
Ext.get('div7').on('click', function() {
    this.fadeOut({
           scope    : this,
           callback : function() {
                  this.fadeIn.defer(500, this);
           }

    });
});
```

Now, after you click on div7, the item will fade out then reappear after a half second. We pass an object literal to the fadeOut method call, which specifies the scope (the

Ext.Element) from which an anonymous function (callback) is to be called. In that callback function, we call `this.fadeIn.defer`, which allows us to defer the execution of the fadeIn method. We pass the time of a half second (500 milliseconds) and the scope from which the deferred (`this.fadeIn`) method is to be called.

What about Ext Components? Can we apply effects to Ext components? Absolutely! Let's apply an effect to one of the basic Ext widgets, the `Ext.Panel`:

Listing 2.4 Applying an effect to a Panel

```
var myButton = new Ext.Button({
     text    : 'hide me',
     handler : function() {
            myPanel.el.switchOff({
                   callback : function() {
                          myPanel.el.slideIn.defer(500, myPanel.el, []);
                   }
            });
     }
});

myPanel = new Ext.Panel({
     width    : 200,
     height   : 100,
     title    : 'Effects are cool!',
     frame    : true,
     renderTo : Ext.getBody(),
     items    : myButton
});
```

Here, we introduce you to two new widgets, the `Ext.Button` and `Ext.Panel`. The button allows you to trigger the effect calls, while the panel serves as an example of any Ext widget. The click handler for the button is very similar to the `div7` click handler, except we reference the (private) `el` property of `myPanel`, which is an instance of Ext.Element, to execute the effects on the component's DOM node.

While effects are cool, they are expensive, and may seem choppy or just incomplete if a host to the browser is under a heavy load or simply does not have the resources to satisfy the demand of the desired effect. If you choose to use effects in your page or application, be sure to thoroughly think about system performance, as improperly displayed effects may annoy users. Also, be sure not to overdo it either. Too much sliding, fading, etc will result in a UI that is hard to use and not well received by users.

Now that we have had some fun applying effects to an Ext widget, let's explore Ext's Component management model.

2.4 The Component Model

The Ext Component model provides a centralized model that provides many of the essential component related tasks, which include everything from the component lifecycle to

query methods. Having a solid grasp on the how the component model works in Ext 3.0 will allow you to better utilize the framework, especially when managing child items of a container.

All UI widgets are a descendant of `Ext.Component`. The base class, `Ext.Component`, manages the many facets of a component's life such as creation, rendering, event handling, state control and destruction. Figure 2.2 partially depicts how many items actually subclass `component`, directly or indirectly.

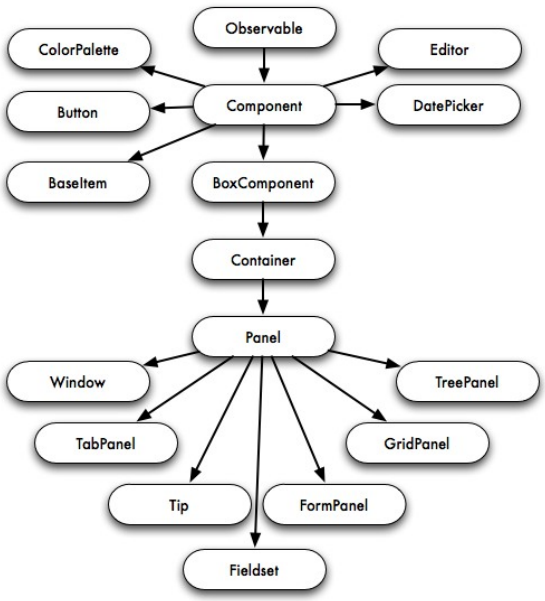


Figure 2.2 This illustration of the Ext class hierarchy focuses on some of the common descendants of Ext.Component and depicts how widely used the Component model is used in the framework.

2.4.1 Lazy or direct rendering

The Ext.Component class supports both direct and lazy (on-demand) rendering models. Direct rendering is 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.

If the `renderTo` or `applyTo` attributes are not specified, a call to the component's render method must be done in order to get the component to inject elements into the DOM, effectively rendering itself.

If a component is a child of another component (specified in the `items` attribute of the configuration object), `applyTo` and `renderTo` need not be specified, as its parent will manage the call to its `render` method when it is required. This is known as lazy rendering.

2.4.2 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 built on the framework. Each component 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. Registration is simple:

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

The act of registering a component to the `Ext.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 initializes, it looks to see if it has `this.items` and will inspect `this.items` for plain old JavaScript objects (POJSO). If a POJSO is specified, it will attempt to create a component using the `ComponentMgr.create` method to instantiate an instance of a component based on what XType is specified in that POJSO. If an XType is not defined in that POJSO, the said visual component will use its `defaultType` property when calling `ComponentMgr.create`.

I realize that this may sound a bit confusing at first. I think we can better understand this concept if we exercise it. Let's create a window with an accordion layout that includes three children. First, let's create our POJSOs for two of our window 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'
}
```

`panel1` has an explicit `xtype` definition of 'panel', which in turn gets used to create an instance of `Ext.Panel` by means of a call to `ComponentManager.create` by `panel1`'s parent component class, as 'panel' is the xtype for that `Ext.Panel`. Objects `panel1` and `panel2` are very similar, but have two distinct differences. Object `panel1` has an `xtype` specified while `panel2` does not. Recall that a `defaultType` is used as the `xtype` for a POJSO item if it is not explicitly defined in the POJSO. Next, we'll create a POJSO for our third panel:

Listing 2.5 Building a form panel

```
var formPanel = {                                          {#1}
   xtype  : 'form',                                        {#2}
   title  : 'Form Panel',
   defaults : {
      width : 150
   },
   items  : [                                              {#3}
      {
         xtype      : 'textfield',
         fieldLabel : 'Name'
      },
      {
         xtype      : 'checkbox',
         fieldLabel : 'Do you love Ext?'
      },
      {
         xtype      : 'combo',
         fieldLabel : 'Choose Version',
         store      : [ '3.0', '2.0', '1.0' ]
      }
   ]
}
```

#1 Create the reference to a plain object
#2 XType defined as form, referenced as Ext.form.FormPanel in the component class
#3 Here are the child items, a list of objects. Each has an xtype specified.

The `formPanel` POJSO **{#1}** has an xtype of "form" **{#2}**, which is an XType for the `Ext.form.FormPanel` class. Within the `formPanel` object, we define `items` **{#3}**, which is an array of objects that contains XTypes for fields. Next, let's create our window, which will use these xtypes:

Listing 2.6 Building our Ext window

```
new Ext.Window({
    width        : 300,
    height       : 300,
```

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

In our new instantiation of `Ext.Window`, we pass items, which is an array of references to the three of our POJSOs. The rendered window should display as described in Figure 2.3. Clicking on a collapsed panel will expand and collapse any other expanded panels and clicking on an expanded panel will collapse it.

Figure 2.3 The accordion layout is extremely useful when you want to display multiple panels in a vertical stack with one panel expanded.

One of the lesser-known advantages of using XTypes is cleaner code. Because you can use plain object notation, you can specify all of your xtype child items inline, resulting in cleaner, more streamlined code. Here is the previous example reformatted for all xtype'd children:

Listing 2.7 A form panel inside of a window using XTypes

```
new Ext.Window({
   width        : 300,
   height       : 300,
   title        : 'Accordion window',
   layout       : 'accordion',
   border       : false,
```

```
      layoutConfig : {
         animate : true
      },
      items : [
         {
            xtype : 'panel',                                          {#1}
            title : 'Plain Panel',
            html  : 'Panel with an xtype specified'
         },
         {
            title : 'Plain Panel 2',                                  {#2}
            html  : 'Panel with <b>no</b> xtype specified'
         },
         {
            xtype  : 'form',                                          {#3}
            title  : 'Form Panel',
            defaults : {
               width : 150
            },
            items  : [                                                {#4}
               {
                  xtype      : 'textfield',
                  fieldLabel : 'Name'
               },
               {
                  xtype      : 'checkbox',
                  fieldLabel : 'Do you love Ext?'
               },
               {
                  xtype      : 'combo',
                  fieldLabel : 'Choose Version',
                  store      : [ '3.0', '2.0', '1.0' ]
               }
            ]
         }
      ]
   }).show();
```

{#1} The “Plain Panel 1” XType object
{#2} The “Plain Panel 2” XType object
{#3} The “Form Panel” XType object
{#4} The Form Panel child XType objects

One main disadvantage to having inline XTypes is extremely long lists of items. Many experienced Ext developers use XTypes to the very extreme, where a list of XTypes can nest more than 3 levels and can span many printable pages, which makes it very hard for other developers to follow along. I have followed a personal rule for using XTypes, which prevents this mess: If it spans more than one screen page, it’s too big! Abstract the XTypes to references or build a factory method or class to generate the configuration. Naturally, you must decide how to layout your code. Always keep in mind that someone else will eventually read your code.

In order to fully appreciate and leverage components, we 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 out UIs and troubleshooting issues.

2.5 The Component Life Cycle

Ext Components, just like everything in the real world, have a life cycle where they are created, used and destroyed. In Ext, this lifecycle is broken up into three major phases: initialization, render and destruction as displayed in figure 2.4.

Figure 2.4 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`.

2.5.1 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 2.5.

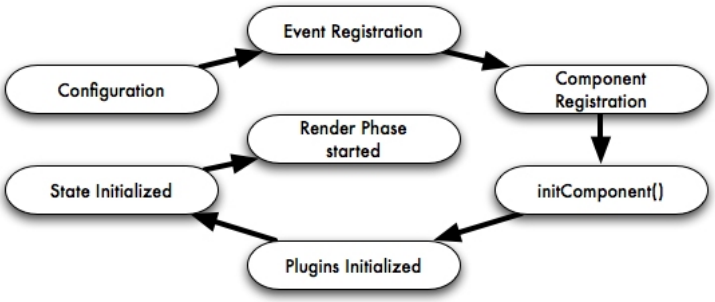


Figure 2.5 The initialization phase of the component lifecycle executes important steps such as event and component registration as well as the calling the initComponent method. It's important to remember that a

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.

2.5.2 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 2.6 illustrates the steps of the render phase.

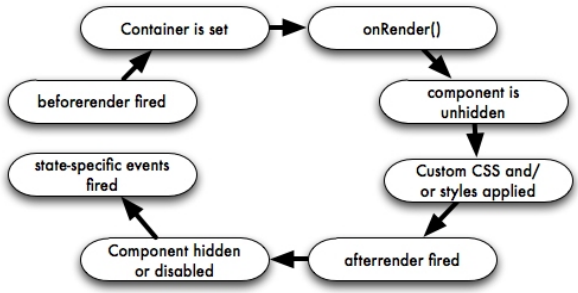


Figure 2.6 The render phase of the a component's life can utilize a lot of CPU as it requires elements to be added to the DOM and calculations to be performed to properly size and manage them.

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:

```
myComponent.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:

8. 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.

9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.

16. 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.
17. Once the render phase is complete, unless the component is disabled or hidden, it's ready for user interaction. It stays alive until it's `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.

2.5.3 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 2.7.

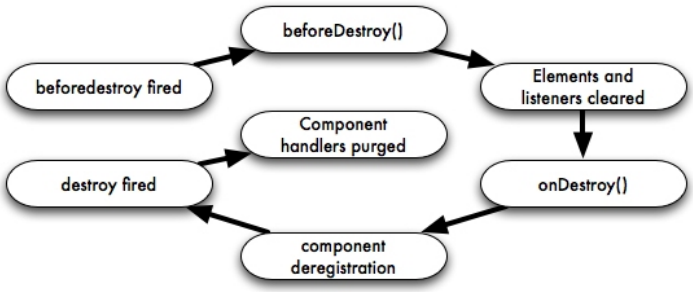


Figure 2.7 The destruction portion of a component's life is equally as important as its initialization as event listeners and DOM elements must be deregistered and removed, reducing over-all memory usage.

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:

18. beforedestroy is fired: This, like many "before<action>" events, is a cancelable event, preventing the component's destruction if its event handler returns false.
19. 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`.
20. 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.

21. 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.
22. Component is unregistered from ComponentMgr: The reference for this component in the ComponentMgr class is removed.
23. 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.
24. Component's event handlers are purged: All event handlers are deregistered from the Component.
25. 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.
26. 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.

2.6 Summary

This chapter contained a lot of key core competencies that are required for successful and effective use of Ext. Here, we started really dig into Ext and broaden our knowledge of the framework by exploring some of its most intimate inner workings.

We learned how events flow in two of the most popular Web browsers and how Ext handles them with the Observable class. We also glanced at how events are used in components and widgets to provide component-level communication. Events are a powerful tool that is used create truly Rich Internet Applications and are a necessary evil.

Ext.Element is an amazing element wrapper that provides a suite of management utilities for DOM Nodes, which include effects. 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) el property, but only after it has been rendered.

We explored how to reduce the amount of our code by using XTypes, which is part of the component model. Another great way to reduce the expense of complex widget layouts is lazy (or deferred) rendering, which is where Ext does not visually produce a widget until it is required to do so as DOM manipulation is the slowest function of JavaScript. Lazy rendering is commonly used in widgets like the TabPanel, which uses the card layout scheme.

We took a real in-depth look at the Component Models, which gives the framework a unified method of managing instances of components. I highly suggest reviewing the component lifecycle when start on your first composite component or custom extension. I

find that I have to review the subject every now and again to make sure that I properly adhere to the model when creating my custom widgets.

Moving forward, we'll start to exercise the framework's UI machinery, laying the foundation for you to build just about any type of UI you desire using your browser.

03

A place for Components

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 cover Containers, which provide the management of child items and is one of the most important concepts in the framework. We're also 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.

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'
   height : 100
}

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

var formPanel =  {                                                {#2}
   xtype       : 'form',
   height      : 125,
   autoScroll  : true,
   id          : 'formpanel',
   defaultType : 'field',
   title       : 'form panel',
   items       : [
      {
         fieldLabel : 'Name'
      },
      {
         fieldLabel : 'Age'
      }
   ]
};

var myWin = new Ext.Window({                                      {#3}
   height   : 400,
   width    : 400,
   defaults : {
      frame : true
   }
   items  : [
      panel1,
      panel2,
      formPanel
   ]
});

myWin.show();
```

{Annotation #1} The first and second child panels
{Annotation #2} The last child, a form panel
{Annotation #3} We're using a Window as a container

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 a form panel {#2}. We also create myWin {#3}, an instance of Ext.Window, which contains the previously defined. The rendered UI should look like the one in Figure 3.1.

Figure 3.1 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.

3.1.1 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    : 'addedPanel',
   html  : 'Hello there!'
});
```

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(2, {
    title : 'Inserted Panel',
    id    : 'insertedPanel',
    html  : 'It is cool here!'
});
Ext.getCmp('myWin').doLayout();
```

We simply insert a new Panel at index 2, which is right under Panel2. 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 3.2:

Figure 3.2 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 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. 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.

3.1.2 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.

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 returns 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. Let's see how we can use Ext to create a UI using all of the browser's available viewing space.

3.1.3 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 : 'fit',
    items : {
        xtype : 'panel',
```

```
            title : 'My First Viewport',
            html  : 'My First Viewport Body',
            frame : true
        }
    });
```

The rendered Viewport will display a simple panel that resizes to the entire browser's viewport, which responds to the browser's window resize event. 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 get around this, you can use the card layout with the viewport and "flip" through different application screens, which will "fit" to the viewport. We'll dive into layouts in chapter 4, 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.

3.2 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 3.3. 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.

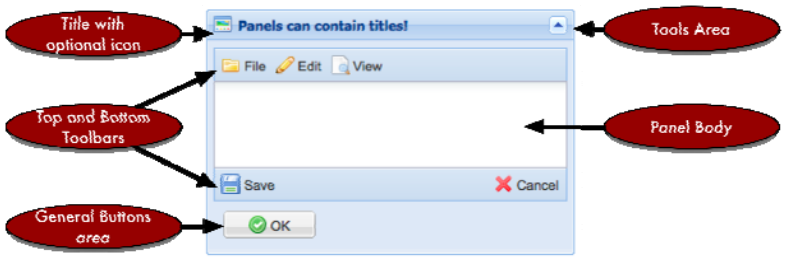


Figure 3.3 An example of a fully loaded Panel, which has a title bar with an Icon and tools, top and bottom toolbars, and a button bar on the bottom.

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.

3.2.1 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 3.2 Building toolbars for use in a Panel

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

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 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, lets create our panel body:

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

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 3.4. 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.

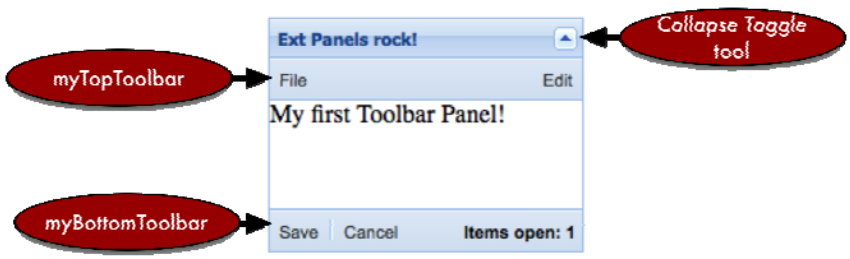


Figure 3.4 The rendered results of Listing 3.4, where we create a complex-collapsible panel with a top and bottom toolbar that have each contain buttons.

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 3.3 Adding buttons and tools to our existing panel

```
var myPanel = new Ext.Panel({
   …                                                          {#1}
   buttons     : [                                            {#2}
      {
         text    : 'Press me!',                               {#3}
```

```
            handler : myBtnHandler
         }
      ],
      tools       : [                                                    {#4}
         {
            id      : 'gear',                                           {#5}
            handler : function() {
               Ext.MessageBox.alert('You Clicked', 'The gear tool');
            }
         },
         {
            id      : 'help',                                           {#6}
            handler : function() {
               Ext.MessageBox.alert('You Clicked', 'The help tool');
            }
         }
      ]
   });
```

{Annotation #1} The original properties from the previous myPanel example
{Annotation #2} The buttons Array begins
{Annotation #3} The 'Press me!' button, which will appear below the panel body.
{Annotation #4} The tools array begins
{Annotation #5} The 'gear' tool and inline click handler
{Annotation #6} The 'help' tool and inline click handler

In Listing 3.3, 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 3.1 and the buttons shortcut array in Listing 3.3 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 3.5.

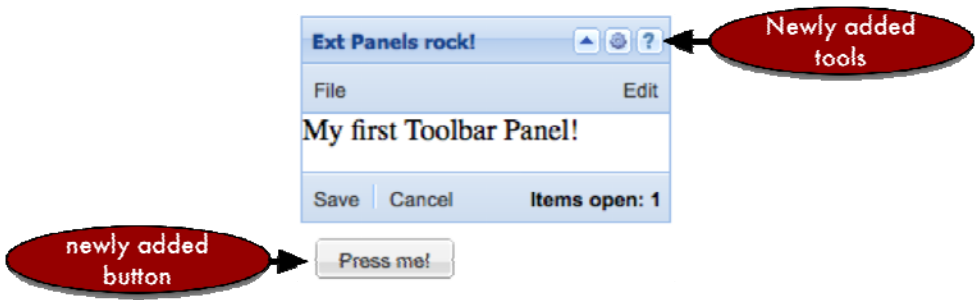


Figure 3.5 The rendered results from Listing 3.5, which add a button in the button bar as well as tools to the title bar.

The preceding example is a 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.

3.3 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 3.6 is a perfect example of how we can leverage this class to focus the user's attention and request input.

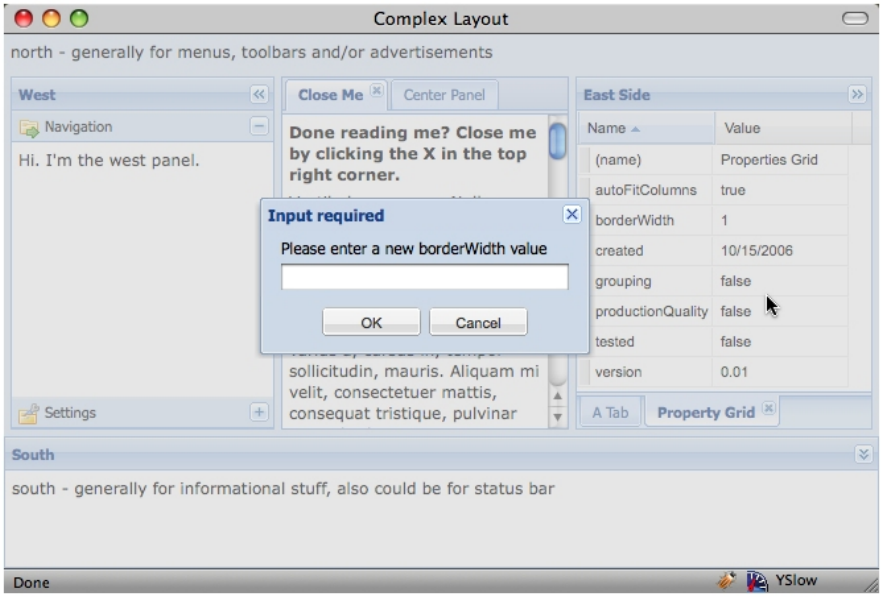


Figure 3.6 An Ext Modal Window, which masks the browser's viewport.

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 browsers 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 3.4 Building an animated window

```
var win;
var newWindow = function(btn) {                                    {#1}
   if (! win) {
      win = new Ext.Window({                                       {#2}
         animateTarget : btn.el,
         html          : 'My first vanilla Window',
         closeAction   : 'hide',                                   {#3}
         id            : 'myWin',
         height        : 200,
         width         : 300,
         constrain     : true                                      {#4}
      });
   }
   win.show();

}
new Ext.Button({                                                   {#5}
    renderTo : Ext.getBody(),
    text     : 'Open my Window',
```

```
    style    : 'margin: 100px',
    handler  : newWindow
});
```

{Annotation #1} newWindow, a handler that creates our new window
{Annotation #2} instantiating a new Window and referencing it to the global win variable
{Annotation #3} preventing the automatic destruction upon click of the close tool
{Annotation #4} ensuring the window stays within the browser's viewport
{Annotation #5} Creating a button that launches the window in the browser

In listing 3.4, 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 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 3.7.

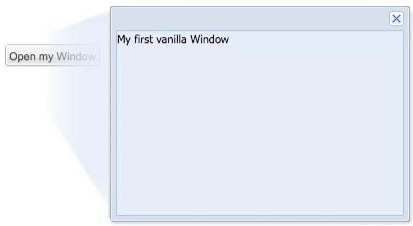


Figure 3.7 The rendered results from Listing 3.4, where we create a window that animates from the button's element upon click.

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. Whenever you deem that it is necessary to destroy the window, you can call it's

destroy or close method. Now that we have experiencing in creating a reusable Window, we can now begin exploring other configuration options to further alter the behavior of the Window.

3.3.1 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 3.5 Creating a rigid modal Window

```
var win = new Ext.Window({
   height      : 100,
   width       : 200,
   title       : 'This is one rigid window',
   html        : 'Try to move or resize me. I dare you.',
   modal       : true,                                          {#1}
   resizable   : false,                                         {#2}
   draggable   : false,                                         {#3}
   closable    : false,                                         {#4}
   plain       : true,                                          {#5}
   border      : false,
   buttonAlign : 'center',
   buttons     : [
      {
         text    : 'I give up!',
         handler : function() {
            win.close();
         }
      }
   ]
})
win.show();
```

{Annotation #1} Ensuring the page is masked
{Annotation #2} prevent resizing from occurring
{Annotation #3} disabling window movement
{Annotation #4} preventing window closure
{Annotation #5} Cosmetic parameters

In Listing 3.5, 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 3.8.

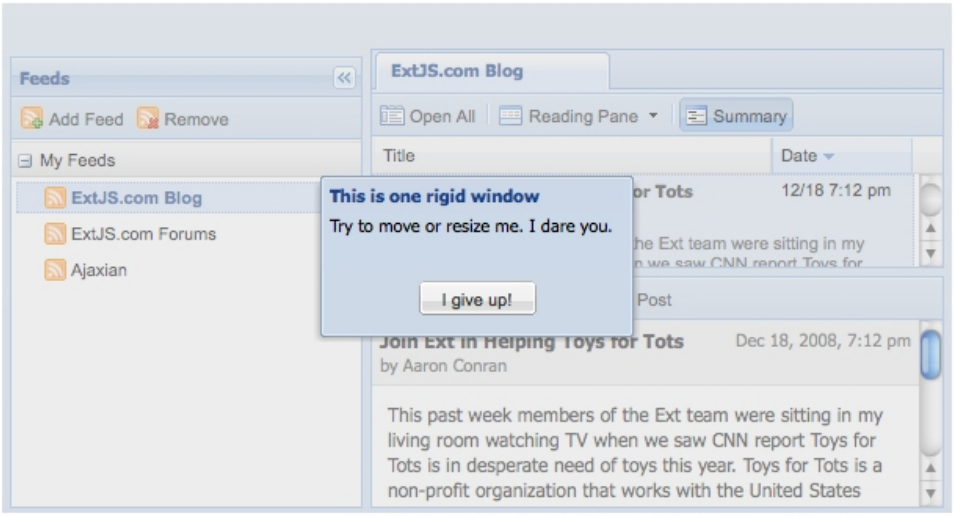


Figure 3.8 Our first strict modal Window rendered in the Ext SDK feed viewer example.

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.

3.3.2 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 our callback method.

```
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 3.9 (left) and will dismiss when the OK button is pressed. If you wanted myCallback to get executed upon dimissal, 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 3.9. Enter some text and press the Cancel button. In the FireBug console, you'll see the button ID pressed and the text entered.

Figure 3.9 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.

3.3.3 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 3.10 (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 3.10 (right).

Figure 3.10 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,

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.

```
.x-window-dlg .myIcon {
    background:transparent url(/path/to/myIcon.gif) no-repeat top left;
}
```

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.

3.3.4 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 3.11. 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 3.11 also shows what it would be like if we added progress bar text to our dummy wait dialog. If the syntax

Figure 3.11 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 3.6 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){                                    {2}
   return function(){
      if(num == 6){
         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);
}
```

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

In Listing 3.6, 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 updaterFn{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 the we will calculate a percentage completed and update the progress bar width and text accordingly{5}. Lastly, we create our loop that calls setTimeout 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 3.12.

Figure 3.12 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. Lets 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.

3.4 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 3.13.

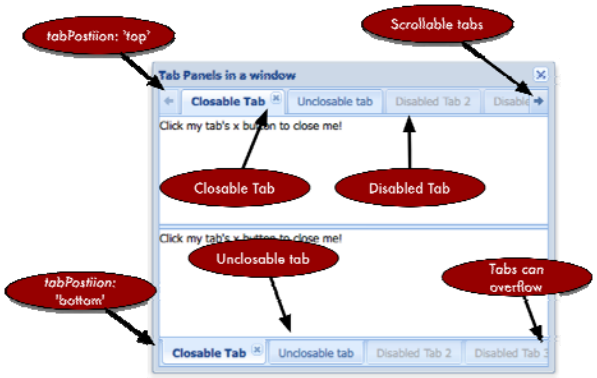


Figure 3.13 Exploring top and bottom positioned tabs.

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 3.0, 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.

3.4.1 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 3.14, 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.

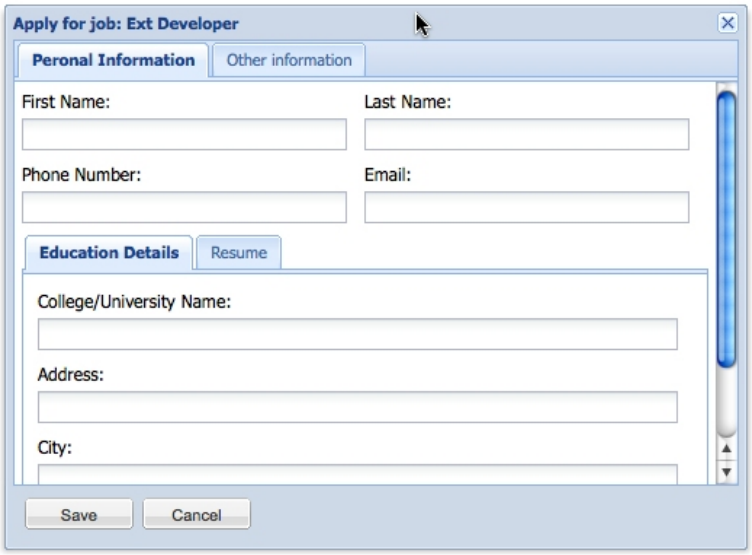
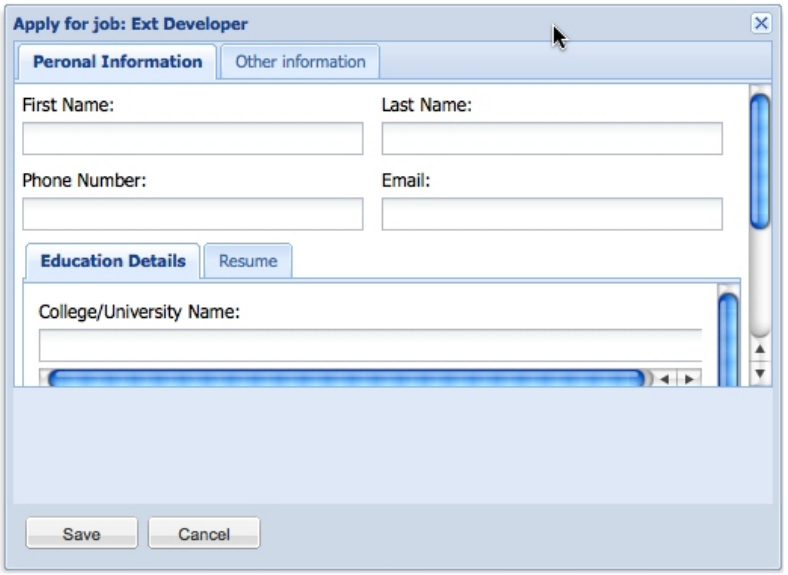


Figure 3.14 A Tab Panel with children that have complex layout.

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 3.15, I suggest setting this configuration option to true and the issue will be solved.



3.15 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, lets move on to build our first tab panel.

3.4.2 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 3.7 Exploring a Tab Panel for the first time.

```
var simpleTab = {                                                    {#1}
        title : 'My first tab',
```

```
        html  : 'This is my first tab!'
}

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,
        layoutOnTabChange : true,
        enableTabScroll   : true,
        id                : 'myTPanel',
        items             : [
                simpleTab,
                closableTab,
                disabledTab,
        ]
});

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

{Annotation #1} A simple, static tab
{Annotation #2} A simple, closable tab
{Annotation #3} A closable, yet disabled tab
{Annotation #4} Our actual Tab Panel
{Annotation #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 POJSO 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{#2} tab.

We then go on to instantiate our Tab Panel{#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.

Next, we create a Container for our Tab Panel, 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 should look like the one in Figure 3.16.

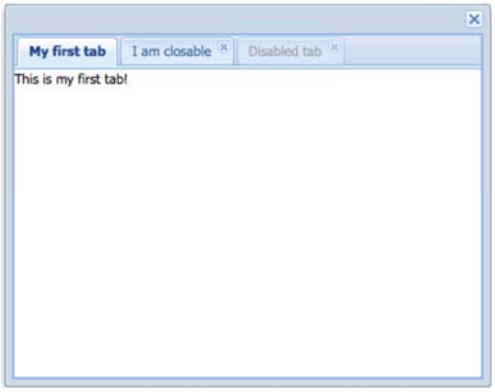


Figure 3.16 Our first tab panel rendered inside of a Window.

Now that we have our first Tab Panel 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".

3.4.3 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 Tab Panel.

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 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 Tab Panel

The Tab Panel 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 3.1 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 Tab Panel 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 Tab Panel 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 Tab panel 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.

3.4.4 Working with caveats and drawbacks

While the Tab Panel 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 Tab Panel 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 Tab Panel 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 Tab Panel inside of a Viewport.

Listing 3.8 Exploring scrollable tabs

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

 (function (num) {
{#2}
   for (var i = 1; i == 20; i++) {
      var title = 'Long Title Tab # ' + i;
      Ext.getCmp('myTPanel').add({
         title    : title,
         html     : 'Hi, i am tab ' + i,
         tabTip   : title,
         closable : true
      });
   }
}).defer(1000);                                                   {#3}
```

{Annotation #1} An Ext Viewport with an embedded tab panel, which includes a single starter tab.
{Annotation #2} An deferred anonymous function execution
{Annotation #3} The third parameter for Function.defer is the arguments array

In Listing 3.8, we create a viewport with our tab panel{#1} which contains a single child. Next, we create an anonymous function{#2} and defer its execution by 1 second. 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 3.17.

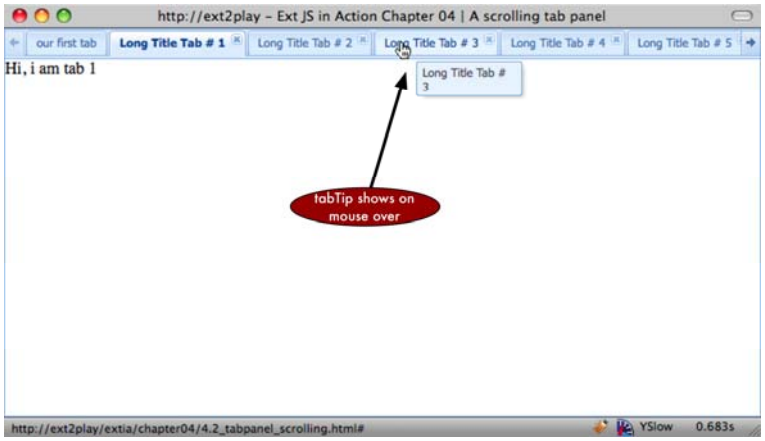


Figure 3.17 A Tab Panel with a scrolling tab strip, which includes mouse over tooltips for the dynamic tabs.

Now that we have our tab panel 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 Tab Panel XType configuration:

```
autoSizeTabs : true,
minTabWidth  : 75,
```

Refreshing the newly modified tab panel in Figure 3.X (bottom) results in tabs that are either unusable or hard to use. Specifying autoSizeTabs as true instructs the tab panel 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 tab panels 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.

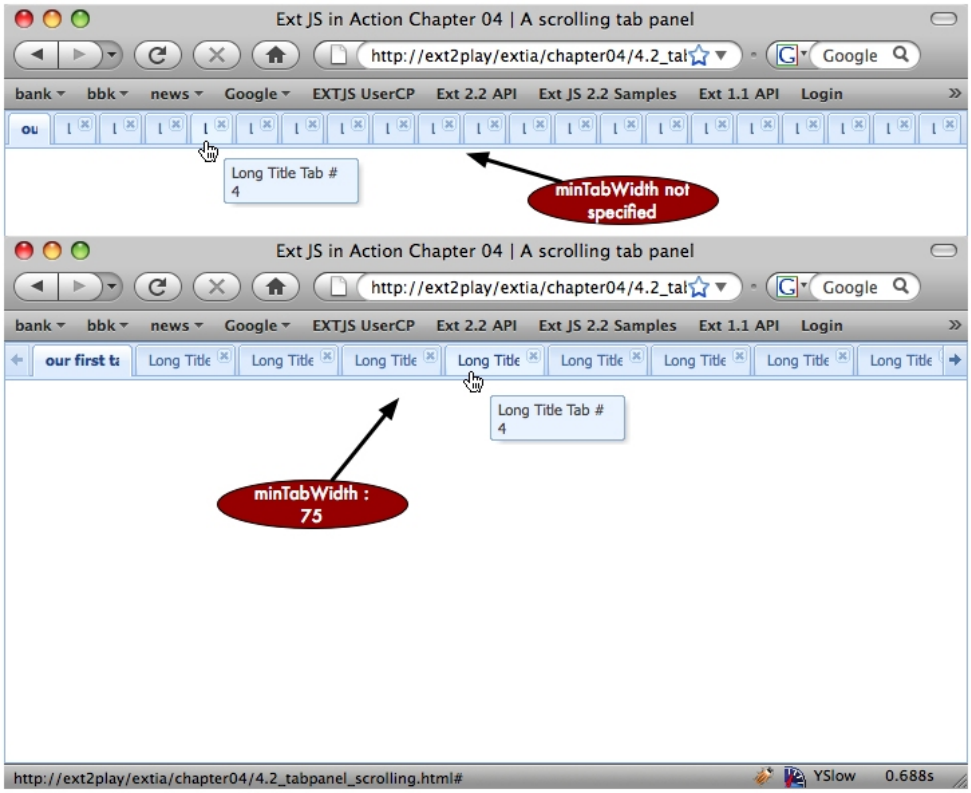


Figure 3.18 A tab panel that has no minimum tab width (top) specified and a tab panel (bottom) that has a minimum tab width of 75 specified.

No matter which route you choose for implementing the Tab Panel, 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 Tab Panel, 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 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.

3.5 Summary

We covered an extreme amount of core-Ext JS material in this chapter, walking up the UI class hierarchy from the Container to the Swiss Army Knife of UI display widgets, the Panel, which is enough to make just about any developer's head spin.

In exploring Containers, we learned that they are the lowest level general Component to support child items, and is a super class to many UI widgets including the Panel and Window. Using the Window class as a general container, we 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.

We got a chance to explore the Panel class, which provides a plethora of options to display user interactive content including toolbars, buttons, title bar icons and miniature tools.

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 Tab Panels, 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.

04

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.

4.1 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 boy 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.

4.2 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 4.1 illustrates the Ext.layout class hierarchy.

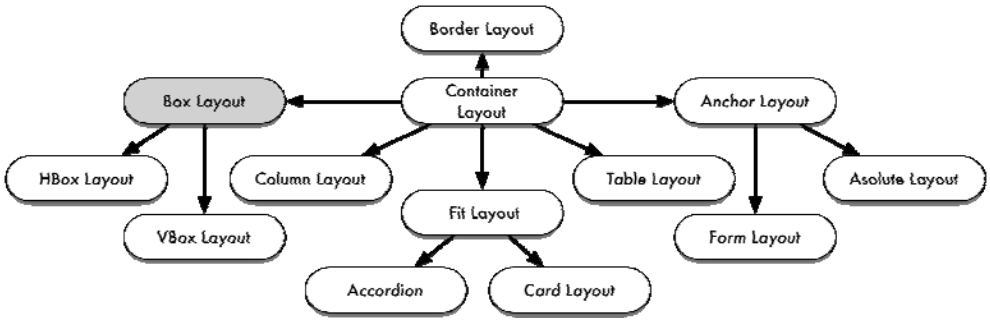


Figure 4.1 The layout class hierarchy, where layouts are descendants of the Container layout

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 4.1 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();
        }
    }
  ]
});
```

{Annotation #1} The first child item, a Panel
{Annotation #2} The third child item, another Panel
{Annotation #3} The Window which contains the three child items
{Annotation #4} Set autoScroll to true, to allow the content body to scroll automatically
{Annotation #5} The three child items are referenced
{Annotation #6} The toolbar with the add item button

In Listing 4.1, 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 4.2.

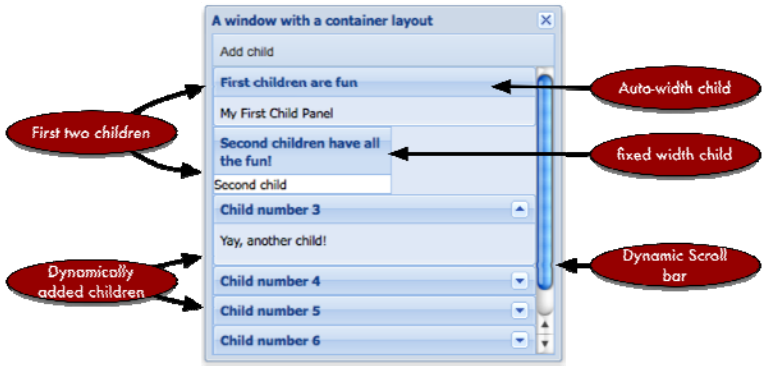


Figure 4.2 The results of our first implementation of the container layout.

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 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.

4.3 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 4.2 The Anchor layout using percentages

```
var myWin = new Ext.Window({                                    {#1}
   height     : 300,
   width      : 300,
   layout     : 'anchor',                                       {#2}
   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();
```

{Annotation #1} The parent container, myWin
{Annotation #2} Layout set to 'anchor'
{Annotation #3} Panel1's anchor parameters, 100% width, 25% of parent's height.
{Annotation #4} Panel2's anchor parameters, full width, 50% of parent's height.
{Annotation #5} Panel1's anchor parameters, full width, 25% of parent's height.

In Listing 4.2, 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 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 4.4.

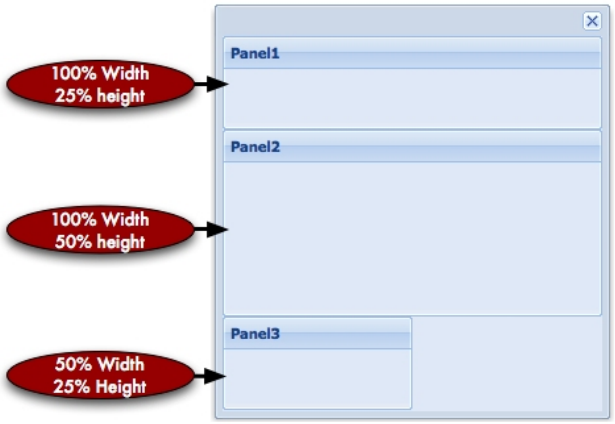


Figure 4.3 The rendered results of our first implementation of the Anchor layout.

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 4.7:

```
items      : [
  {
    title     : 'Panel1',
    anchor    : '-50, -150',
    frame     : true
  },
  {
    title     : 'Panel2',
    anchor    : '-10, -150',
    frame     : true
```

```
    }
]
```

The rendered panel from the preceding layout modification should look like figure 4.4. 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.

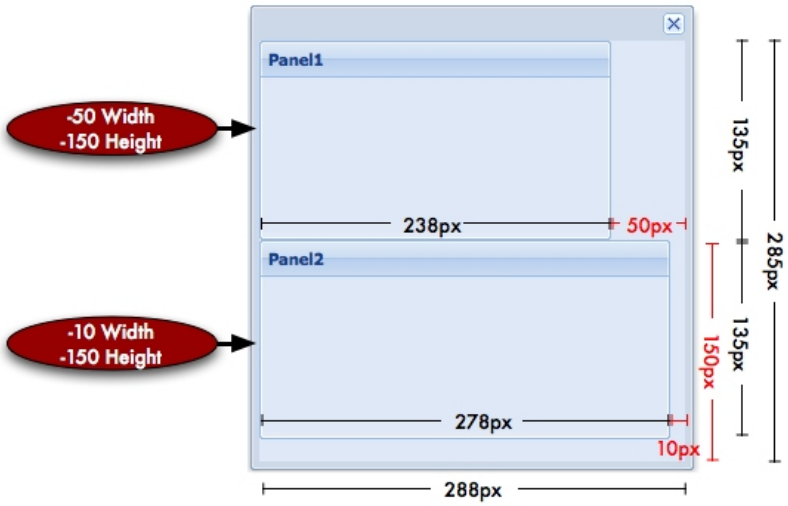


Figure 4.4 Using offsets with an Anchor layout with sizing calculations.

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 Panel 2 should be.

```
Panel1 Width  = 288px - 50px  = 238px
Panel1 Height = 285px - 150px = 135px

Panel2 Width  = 288px - 10px  = 278px
Panel2 Height = 285px - 150px = 135px
```

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 it 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.

4.4 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 4.3 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}
            fieldLabel : 'Bio'
         },
         {
            xtype          : 'panel',                                   {#8}
            fieldLabel     : '',
            labelSeparator : '',
            frame          : true,
            title          : 'Instructions',
            html           : 'Please fill in the form',
            height         : 55
         }
      ]
});

myWin.show();
```

{Annotation #1} Setting the window's layout to 'form'
{Annotation #2} Setting the default label width to 50px
{Annotation #3} Setting the default XType to 'field'
{Annotation #4} First Child Item, Textfield, constant width
{Annotation #5} Second Child Item, Textfield
{Annotation #6} Third Child Item, a combo box
{Annotation #7} Fourth child Item, a text area
{Annotation #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 4.5.

Figure 4.5 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 4.9 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 4.4 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 4.4. The rendered changes should look like figure 4.6. When you resize the example window, you'll see how well the child items resize and conform to their parent container.

Figure 4.6 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.

4.5 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, without having to muck with CSS.

Let's create a window with an absolute layout:

Listing 4.5 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',
            frame  : true
        },
        {
            title  : 'Panel2',
            x      : 90,                                          {#3}
            y      : 120,
            height : 75,
            width  : 77,
            html   : 'x: 90, y: 120',
            frame  : true
        }
    ]
});

myWin.show();
```

{Annotation #1} Setting the window's layout to 'absolute'
{Annotation #2} Panel1's X and Y coordinates
{Annotation #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 4.7.

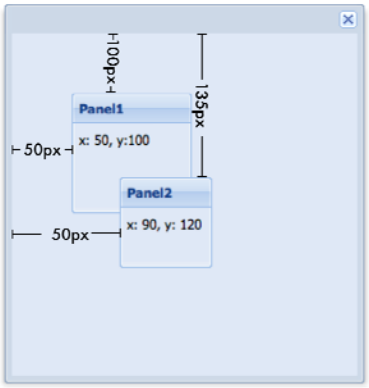


Figure 4.7 The results of our anchor layout implementation.

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.

4.6 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 4.6 The fit layout

```
var myWin = new Ext.Window({
   height     : 200,
   width      : 200,
   layout     : 'fit',                                          {#1}
   border     : false,
   items      : [
      {
         title : 'Panel1',                                     {#2}
         html   : 'I fit in my parent!',
         frame  : true
      }
   ]
});

myWin.show();
```

{Annotation #1} The window's layout set to 'fit'
{Annotation #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 4.8.

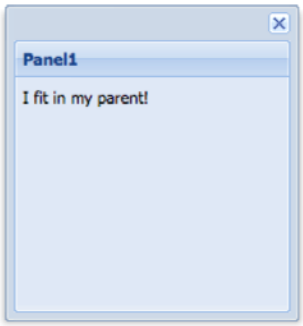


Figure 4.8. Using the fit layout for the first time

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.

4.7 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 4.7 The Accordion layout

```
var myWin = new Ext.Window({                                          {#1}
   height       : 200,
   width        : 300,
   border       : false,
   title        : 'A Window with an accordion layout',
   layout       : 'accordion',                                        {#2}
   layoutConfig : {                                                   {#3}
      animate : true
   },
   items : [
      {                                                              {#4}
         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' ]
            }
         ]
      },
      {                                                              {#5}
         xtype  : 'panel',
         title  : 'Bio',
         layout : 'fit',
         items  : {
            xtype : 'textarea',
            value  : 'Tell us about yourself'
         }
      },
      {                                                              {#6}
         title : 'Instructions',
         html  : 'Please enter information.',
         tools : [                                                   {#7}
            {id : 'gear'}, {id:'help'}
         ]
      }
   ]
});

myWin.show();
```

{Annotation #1} myWin, an instance of Ext.Window
{Annotation #2} specifying an Accordion layout
{Annotation #3} specifying the Accordion layout's layoutConfig
{Annotation #4} the first child item, a form panel
{Annotation #5} the second child item, a panel, which contains a textarea
{Annotation #6} the last child item, a vanilla panel, which has some tools

Listing 4.7 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 4.9.

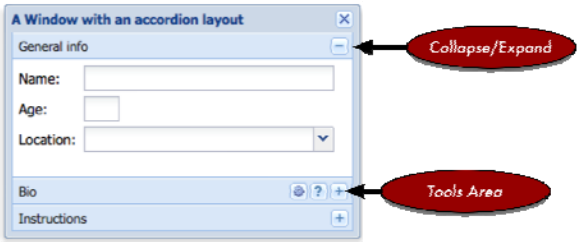


Figure 4.9 The Accordion layout is an excellent way to present the user with multiple items with a single visible component.

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.

4.8 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);                {#1}
        }
        else if (btn.text == 'Back' && index > 0) {
                myWin.layout.setActiveItem(index - 1);                {#2}
        }

        indicatorEl.update((index + 1) + ' of ' + (numItems + 1));
}
```

{Annotation #1} Activating the next card
{Annotation #2} Activating the previous card

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{1} or Back{2} 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 4.8 The Card layout in action

```
var myWin = new Ext.Window({
   height       : 200,
   width        : 300,
   border       : false,
   title        : 'A Window with a Card layout',
   layout       : 'card',                                              {#1}
   activeItem   : 0,                                                   {#2}
   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',                                          {#3}
           handler : handleNav
        },'-',
        {
           text    : 'Forward',
           handler : handleNav
        },'->',
        {
           xtype  : 'box',                                            {#4}
           id     : 'indicator',
           style  : 'margin-right: 5px',
           autoEl :  {
              tag  : 'div',
              html : '1 of 3'
           }
        }

     ]

});

myWin.show();
```

{Annotation #1} Setting the layout to 'card'
{Annotation #2} Set the Container's active item to 0
{Annotation #3} The back button and forward buttons
{Annotation #4} The BoxComponent with the id of 'indicator'

In Listing 4.8, 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 property{#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 pervious 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 4.10.

Figure 4.10 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 BoxComponent. 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.

4.9 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.

Listing 4.9 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();
```

{Annotation #1} Auto scroll set to true on the Container, so it will show the scrollbars if the dimensions of a particular Component exceeds that of its body element.
{Annotation #2} Set the layout to 'column'
{Annotation #3} Set a relative column width attribute, 30%
{Annotation #4} Set a fixed width of 100 pixels
{Annotation #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 4.9, 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 4.11.

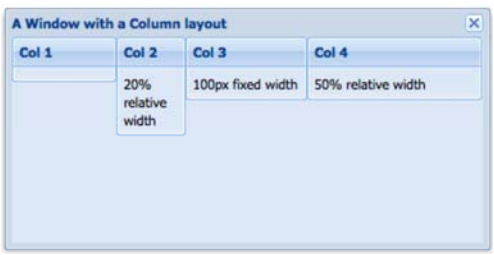


Figure 4.11 Our first column layout, which uses relative column widths with a fixed width entity.

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. Let's 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 4.14 running in your browser.

```
Ext.getCmp('col1').add({
   height : 250,
   title  : 'New Panel',
   frame  : true
});
Ext.getCmp('col1').doLayout();
```

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.

4.10 The 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 4.10 HBox layout, exploring the packing configuration

```
new Ext.Window({
   layout       : 'hbox',                                          {#1}
   height       : 300,
   width        : 300,
   title        : 'A Container with an HBox layout',
   layoutConfig : {                                                {#2}
      pack : 'start'
   },
   defaults : {
      frame : true,
   },
   items : [
      {
         title  : 'Panel 1',
         height : 100
      },
      {
         title  : 'Panel 2',
         height : 75,
         width  : 100
      },
      {
         title  : 'Panel 3',
         height : 200
      }
   ]
}).show();
```

{Annotation #1} Setting the layout to hbox
{Annotation #2} Specifying the layout configuration.

For Listing 4.10, 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 4.15 will result in one of the rendered windows in Figure 4.12. The default value for the pack attribute is 'start'.

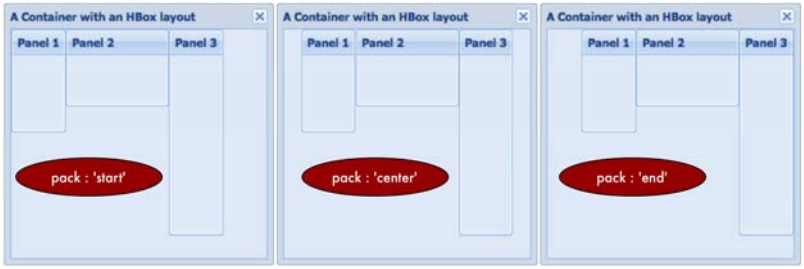


Figure 4.12 Different results with the three pack options.

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 4.13 illustrates how we can change the way the children are sized and arranged based on a few different combinations.

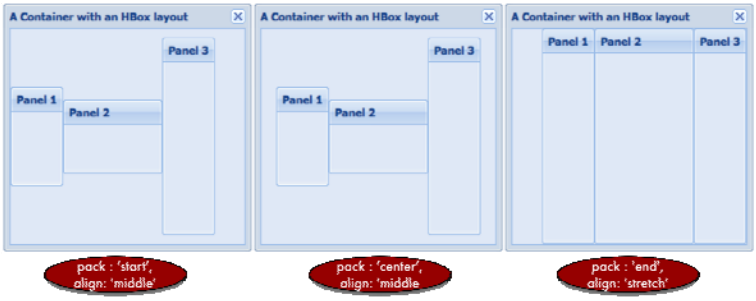


Figure 4.13 The 'stretch' alignment will always override any height values specified by on the child items.

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 4.15 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".

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.

4.11 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. Let's create a 3 x 3 table layout:

Listing 4.11 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 : {                                                    {#3}
      height : 50,
      width  : 50
   },
   items : [
      {
         html : '1'
      },
      {
         html : '2'
      },
      {
         html : '3'
      },
      {
         html : '4'
      },
      {
         html : '5'
      },
      {
         html : '6'
      },
      {
         html : '7'
      },
      {
         html : '8'
      },
      {
         html : '9'
      }
   ]
});

myWin.show();
```

{Annotation #1} Specifying the layout as table
{Annotation #2} Set the number of columns for the table to 3
{Annotation #3} Set default size for each box to 50x50

The code in Listing 4.11 creates a Window Container that has nine boxes stacked in a 3x3 formation like in Figure 4.14. 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.

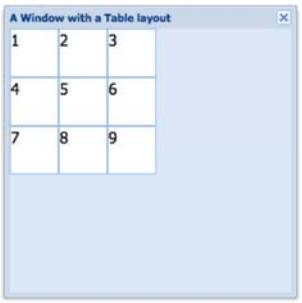


Figure 4.14 The results of our first simple table layout.

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. Let's modify our table so the child items can span multiple rows or columns

Listing 4.12 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'
      },
      {
         html : '7'
      },
      {
         html : '8'
      },
      {
         html    : '9',
         colspan : 3,                                                {#4}
         width   : 150
      }
   ]
```

{Annotation #1} Set colspan to 3 and width to the total width to 150 pixels
{Annotation #2} Set rowspan to 2 and height to 100 pixels
{Annotation #3} Set rowspan to 2 and height to 100 pixels
{Annotation #4} Set colspan to 3 and width to 150 pixels

In Listing 4.12, we reuse the existing Container code from Listing 4.14 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 4.15.

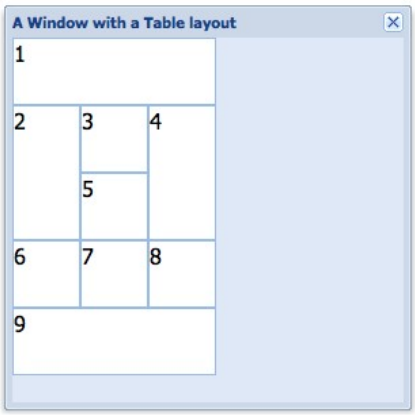


Figure 4.15 When using the table layout, you could specify rowspan and colspan for a particular Component, which will make it occupy more than one cell in the table.

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.

4.12 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 4.16 illustrates what a border layout implementation from the Ext SDK.

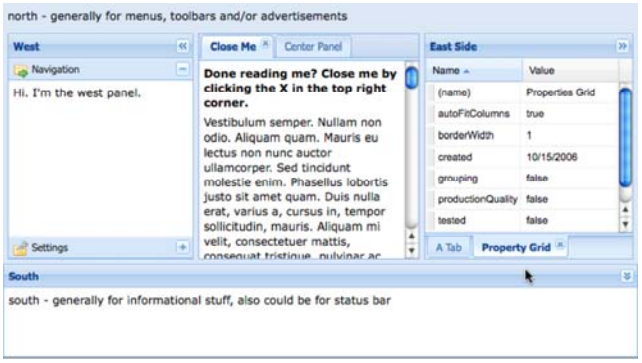


Figure 4.16 The border layout is what attracts many new developers to the Ext Framework and is widely used in many applications to divide the screen into task-specific functional areas.

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 4.1.6, earlier in this chapter.

Listing 4.13 Flexing the Border Layout

```
new Ext.Viewport({
   layout   : 'border',
   defaults : {                                                    {#1}
      frame  : true,
      split  : true
   },
   items : [
      {
         title       : 'North Panel',                              {#2}
         region      : 'north',
         height      : 100,
         minHeight   : 100,
         maxHeight   : 150,
         collapsible : true
      },
      {
         title       : 'South Panel',                              {#3}
         region      : 'south',
         height      : 75,
         split       : false,
         margins     : {
            top : 5
         }
      },
      {
         title       : 'East Panel',                               {#4}
         region      : 'east',
         width       : 100,
         minWidth    : 75,
         maxWidth    : 150,
         collapsible : true
      },
      {
         title        : 'West Panel',                              {#5}
         region       : 'west',
         collapsible  : true,
         collapseMode : 'mini'
      },
      {
         title  : 'Center Panel',                                  {#6}
         region : 'center'
      }
   ]
});
```

{Annotation #1} Setting split to true in the defaults object, which is a BorderLayout.Region specific parameter
{Annotation #2} The static "north" region panel.
{Annotation #3} The resizable "south" region panel, which is also collapsible
{Annotation #4} The resizable and collapsible East panel
{Annotation #5} The mini-collapse west panel
{Annotation #6} The no-frills Center panel.

In Listing 4.13, 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 4.17, 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 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.

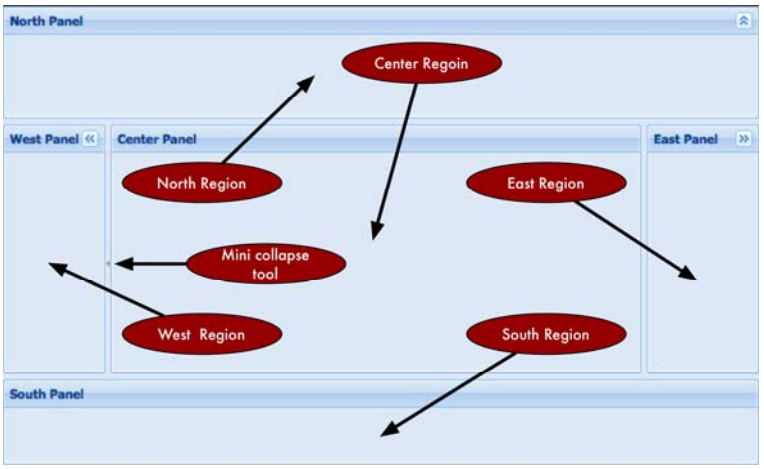


Figure 4.17 The Border Layout's versatility and ease of use makes it one of the most widely used in Ext-based RIAs.

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 4.18 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.

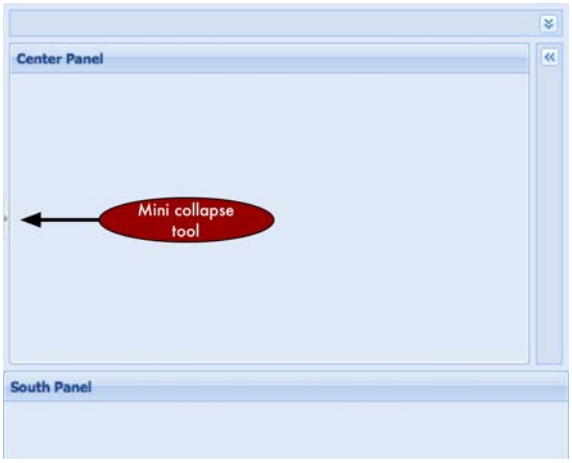


Figure 4.18 Our Border layout where two of the regions, north and east, are collapsed in regular mode and the west panel is collapsed in miniature mode.

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 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 4.19:

```
{
   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 4.18, 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. Let's take a swipe at replacing the center region's child component:

Listing 4.14 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'
            }
        ]
    });
    centerRegion.doLayout();
```

Listing 4.14 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.

4.13 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.

05
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.

5.1 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.

5.1.1 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

properties for the BasicForm via the initialConfig property, which is defined as an object. Here is a sample FormPanel with an initialConfig object:

```
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.

5.2 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'lll create our items array, which will contain the XType definitions of the different text fields.

Listing 5.1 Our text fields

```
Ext.QuickTips.init();

var fpItems =[
   {
      fieldLabel : 'Alpha only',                                    {#1}
      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
{5} 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 fpItems 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{#5} 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. 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'
});
```

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 it 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. OK, now that we have explored VTypes, lets go on ahead to build the form from which our text fields will live:

Listing 5.2 Building the FormPanel for our text fields

```
var fp = new Ext.form.FormPanel({
   renderTo     : Ext.getBody(),
   width        : 400,
   height       : 150,
   title        : 'Exercising textfields',
   frame        : true,
   bodyStyle    : 'padding: 5px',
   labelWidth   : 125,
   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 5.2 should be very familiar to you. But, lets 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 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 5.1.

Figure 5.1 The rendered results of our FormPanel which contains our four text fields.

Notice how in Figure 5.1 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:

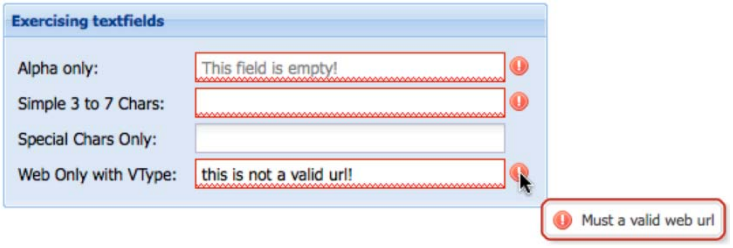


Figure 5.2 "side" validation error messages.

Each field can have it's 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 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. Lets see how we can create password and file upload fields using the TextField.

5.2.1 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:

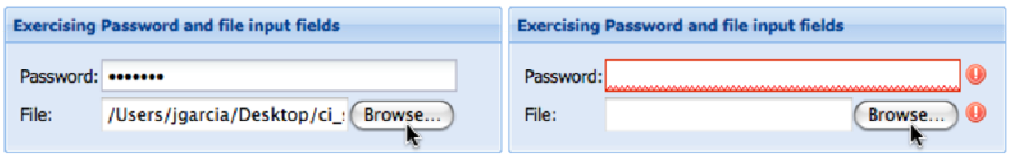


Figure 5.3 Our password and file upload fields with data filled in (left) and an example of the side validation error icons (right).

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 5.3 (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.

5.2.2 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',
        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.

5.2.3 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 decimalPrecsion, 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, let's look at its distant cousin, the ComboBox.

5.3 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.

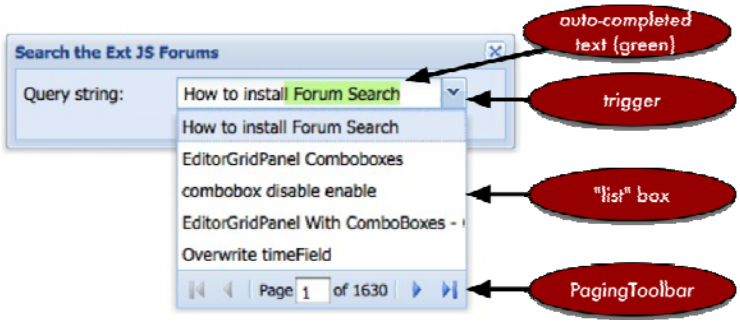


Figure 5.4 An example UI of a remote-loading and paging ComboBox with type ahead

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 5.2.

5.3.1 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. Let's move on to build our first combo box using an XType configuration object:

Lisiting 5.3 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
{#4} Ensure there are no remote data AJAX requests

In listing 5.3, 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 refrence 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:

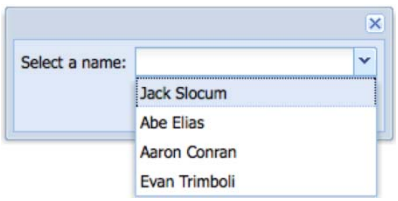


Figure 5.5 An example rendering of our ComboBox from Listing 5.3 inside of a Window.

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 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.

5.3.2 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 5.4 Implementing a remote loading combo box

```
var remoteJsonStore = new Ext.data.JsonStore({
   root          : 'records',                                    {#1}
   baseParams    : {                                             {#2}
      column : 'fullName'
   },
   fields     : [                                                {#3}
      {
         name    : 'name',
         mapping : 'fullName'
      },
      {
         name    : 'id',
         mapping : 'id'
      }

   ],
   proxy : new Ext.data.ScriptTagProxy({                          {#4}
      url : 'http://tdg-i.com/dataQuery.php'
   })
});

var combo = {
   xtype          : 'combo',
   fieldLabel     : 'Search by name',
   forceSelection : true,                                         {#5}
```

```
    displayField   : 'name',                                    {#6}
    valueField     : 'id',                                      {#7}
    hiddenName     : 'customerId',                              {#8}
    loadingText    : 'Querying....',                            {#9}
    minChars       : 1,                                        {#10}
    triggerAction  : 'name',                                   {#11}
    store          : remoteJsonStore
};
```

{1} Specifing 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
{5} 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} Epecify a name for the hidden field who's value is set by the valueField
{9} Specify custom loading text
{10} Send a request as soon as a single character is entered
{11} Allow the trigger to send a request for all data, even if there is text present in the text field

In Listing 5.4, 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 'fullNname' 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{5} 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 5.6 An example rendition of our remote-loading combo from Listing 5.4

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

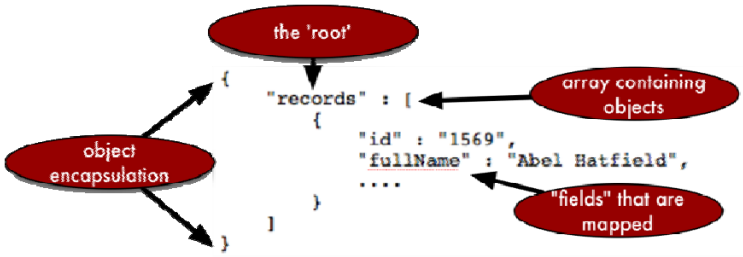


Figure 5.7 An exploded view of a slice of the served up JSON.

In examining a snippet of the resulting JSON in Figure 5.7, you can see the root that we specified in our remote combo's JSON store and the fullName field we mapwped 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 5.6 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 5.4, 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 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.

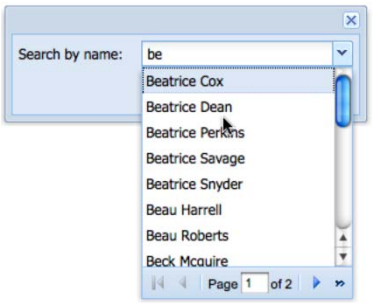


Figure 5.8 Adding pagination to our remote Combo Box.

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. Alhough 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 take a quick glance at the innards of the Combo.

5.3.3 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, lets move forward in creating our custom combo.

5.3.4 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 k now of the fields. In modifying listing 5.4, 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     : #555555;
}
```

In the preceding CSS, we create a class for each of the divs in our template. Nowwe 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 5.9.

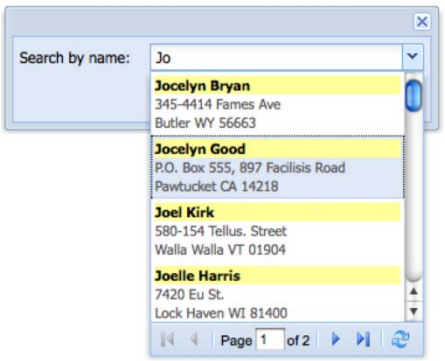


Figure 5.9 An example rendition of our customized combo box.

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. Lets see how we can create a TimeField.

5.3.5 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 11:45 PM. Here is an example of how to do that:

```
{
   xtype      : 'timefield',
   fieldLabel : "Please select time",
   anchor     : '100%'
}
```

Here is an example of how the field above would render:

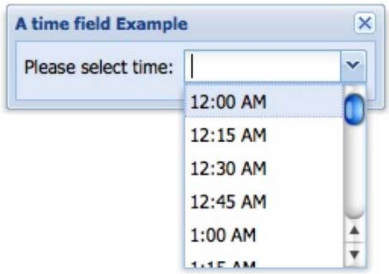


Figure 5.10 Our rendered generic TimeField.

The TimeField is configurable however, where you can set the range of time, increments and even the format. Lets 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 5PM:

```
...
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.

5.4 WYSIWhat?

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. Lets move on to building our first HTML editor.

5.4.1 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:

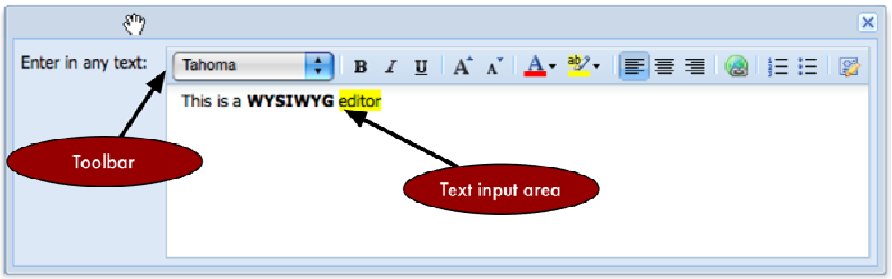


Figure 5.11 Our first HTML Editor in an Ext Window.

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. The HTML Editor is a great tool, but it, like many things has some limitations.

5.4.2 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%',
        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.

5.5 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. Lets build out a DateField:

```
var dateField = {
     xtype      : 'datefield',
     fieldLabel : "Please select a date",
     anchor     : '100%'
}
```

Yes, it's that easy. Lets look at the how the DateField renders.

Figure 5.12 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:

```
["01/15/2000", "01/31/2009"] disables these two exact dates
["01/15"] disables this date every year
["01/../2009"] disables every day in January for 2009
["^01"] 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.

5.6 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. Lets create some checkboxes, where we use custom values.

Listing 5.5 Building Checkboxes

```
var checkboxes = [
   {
      xtype      : 'checkbox',
      fieldLabel : "Which do you own",
      boxLabel   : 'Cat',                                          {1}
      inputValue : 'cat'                                           {2}
   },
   {
      xtype          : 'checkbox',
      fieldLabel     : "",
      labelSeparator : ' ',
      boxLabel       : 'Dog',
      inputValue     : 'dog'
   },
   {
      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 5.5 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 and the inputValue{2}, of course, overrides the default Boolean value. An example rendering of the code above is as follows:

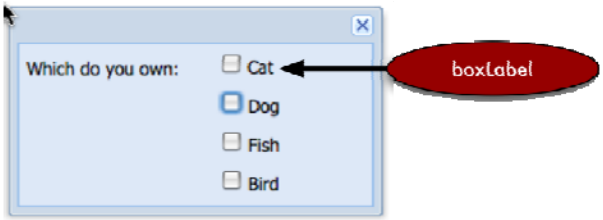


Figure 5.13 Our first four checkboxes.

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 5.6 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 layout your checkboxes in a single horizontal line as seen in figure 5.14. Specifying the number of columns is as simple as setting the columns attribute to the number of desired columns.

Figure 5.14 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 5.15 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.

5.7 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, let's take a sneak peak of what we're going to achieve.

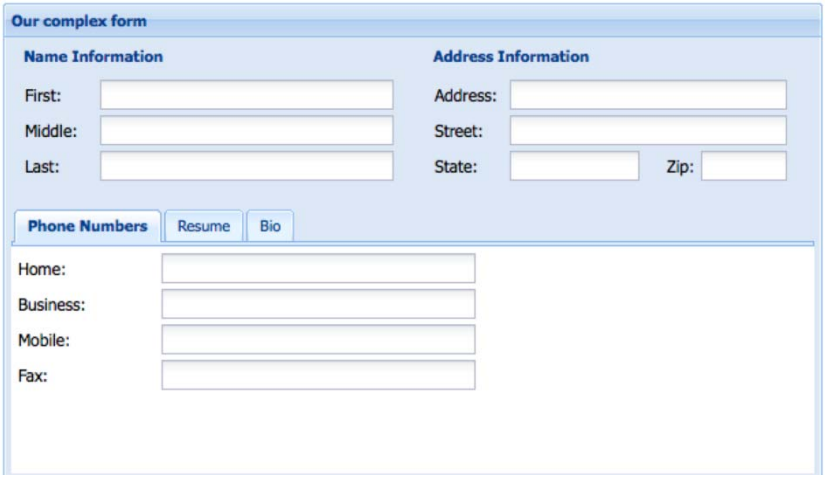


Figure 5.16 A sneak peek of the Complex form panel we're going to build

In constructing our complex form, we're going to have to construct two FieldSets one for the name information and the 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.

OK, 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 5.7 Constructing two FieldSets

```
var fieldset1 = {
   xtype       : 'fieldset',                                    {1}
   title       : 'Name',
   flex        : 1,
   border      : false,                                         {2}
   labelWidth  : 50,
   defaultType : 'field',
   defaults    : {
      anchor     : '-10',
      allowBlank : false
   },
   items : [
      {
         fieldLabel : 'First',
         name       : 'firstName'
      },
      {
         fieldLabel : 'Middle',
         name       : 'middle'
      },
```

```
        {
           fieldLabel : 'Last',
           name       : 'firstName'
        }
     ]
}
```

{1} Setting the xtype property to 'fieldset'
{2} Removing the border from the fieldest

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 5.8 Building our second fieldset.

```
var fieldset2 = Ext.apply({}, {                                    {1}
   flex       : 1,
   title      : 'Address Information',
   items      : [
      {
         fieldLabel : 'Address',
         name       : 'address'
      },
      {
         fieldLabel : 'Street',
         name       : 'street'
      },
      {
         xtype       : 'container',                                {2}
         border      : false,
         layout      : 'column',
         anchor      : '100%',
         items       : [
            {
               xtype  : 'container',                          {3}
               layout : 'form',
               width  : 200,
               items  : [
                  {
                     xtype      : 'textfield',                 {4}
                     fieldLabel : 'State',
                     name       : 'state',
                     anchor     : '-20'
                  }
               ]

            },
            {
```

```
                xtype       : 'container',                          {5}
                layout      : 'form',
                columnWidth : 1,
                labelWidth  : 30,
                items       : [
                   {
                      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} Creating a column-layout Container which will contain two other containers for our state and zip fields
{3} A form-layout Container for our State text field
{4} The actual state text field
{5} Another form-layout Container for the Zip Code TextField
{6} The Zip Code TextField

In Listing 5.8, 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{5} 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. Lets build a home for the two FieldSets:

```
var fieldsetContainer = {
   xtype  : 'container',
   layout : 'hbox',
   height  : 120,
```

```
      layoutConfig : {
         align : 'stretch',
      },
      items  : [
         fieldset1,
         fieldset2
      ]
   }
```

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.

OK, 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 5.9 Building a tabpanel with form items

```
var tabs = [
   {
      xtype     : 'container',                                    {1}
      title     : 'Phone Numbers',
      layout    : 'form',
      bodyStyle : 'padding:5px 5px 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',                                       {2}
      name   : 'resume'
```

```
    },
    {
       title  : 'Bio',
       xtype  : 'htmleditor',
       name   : 'bio'
    }
]
```

{1} The Container that has four Text Fields
{2} The two HTML Editors as Tabs

In Listing 5.9 we wrote a lot of code to construct 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 it child items.

Listing 5.10 Piecing it all together

```
var myFormPanel  = new Ext.form.FormPanel({
   renderTo     : Ext.getBody(),
   width        : 700,
   title        : 'Our complex form',
   height       : 350,
   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 fielsetContainer 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.

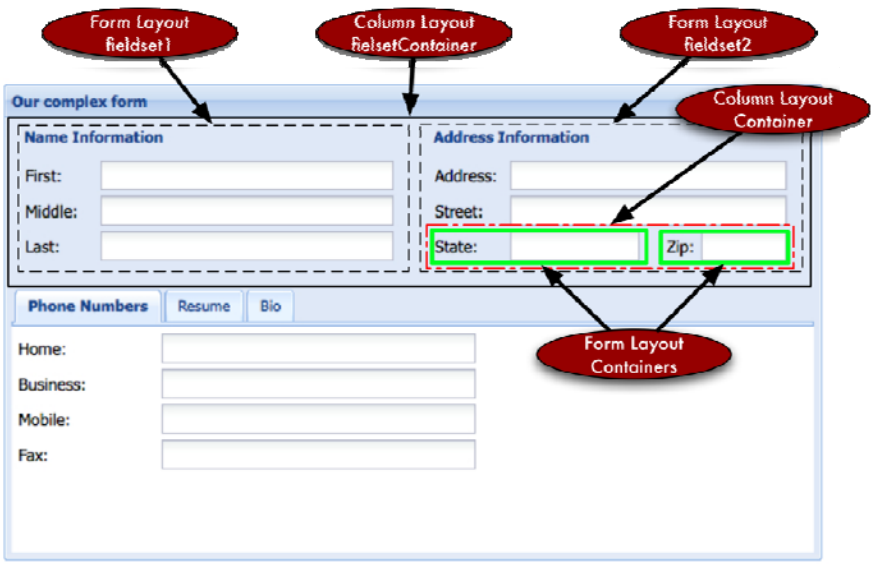


Figure 5.17 The results of our first complex layout form with the different containers used to comprise the complex layouts.

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.

5.8 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.

5.8.1 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.

5.8.2 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 5.11 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 5.12, 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 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.

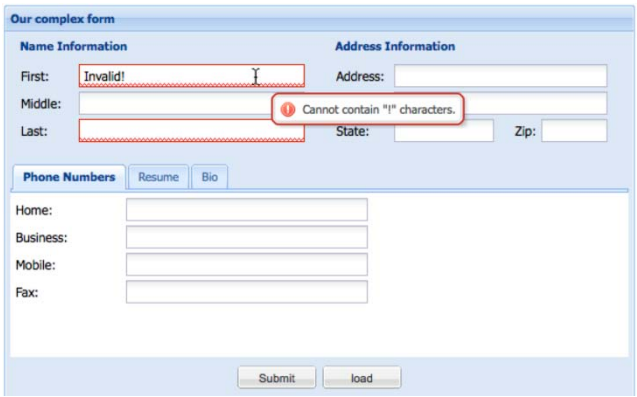


Figure 5.18 The results from our server side errors object using the standard QuickTip error msg.

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.

5.8.3 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. Lets create some mock data and save it in a file called data.php.

```
{
   success : true,
   data    : {
        firstName : "Jack",
        lastName  : "Slocum",
        middle    : "",
        address   : "1 Ext JS Corporate Way",
        city      : "Orlando",
        state     : "Florida",
        zip       : "32801",
        home      : "123 345 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.

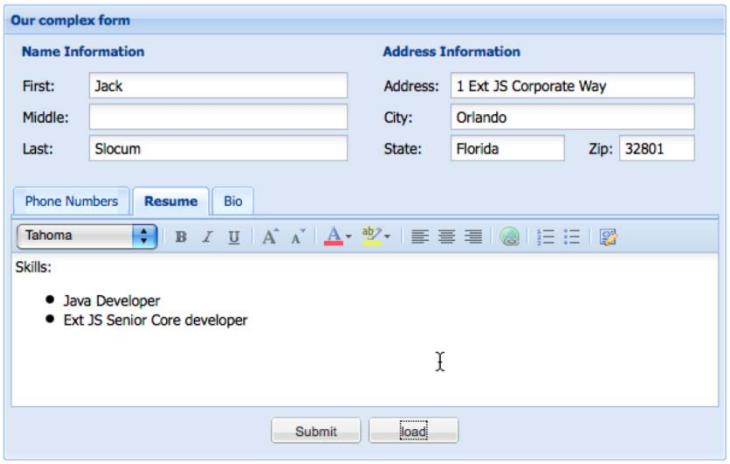


Figure 5.19 The results of loading our data via XHR.

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.

5.9 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!

6

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.

6.1 Introducing GridPanel

At a first glance, the GridPanel may look like a glorified HTML table, which has 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 6.1.

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.

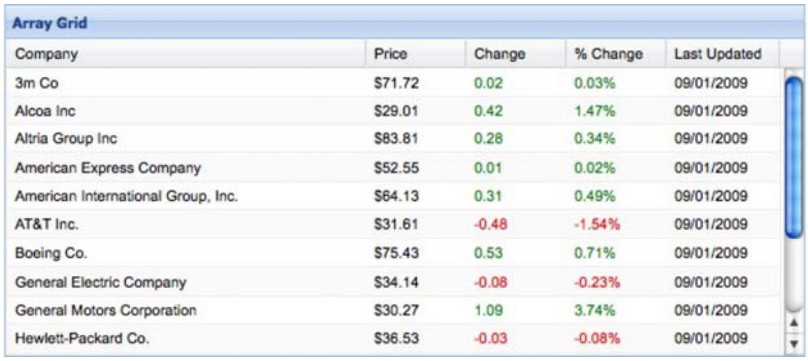


Figure 6.1 The "Array Grid' Example, found in the examples folder of the downloadable SDK.

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.

6.1.1 Looking under the hood

The key supporting classes that drive the GridPanel are the ColumnModel, GridView, SelectionModel and a DataStore. Let's take a quick glance at an implementation of a grid panel and see how each class plays a role in making the GridPanel work.

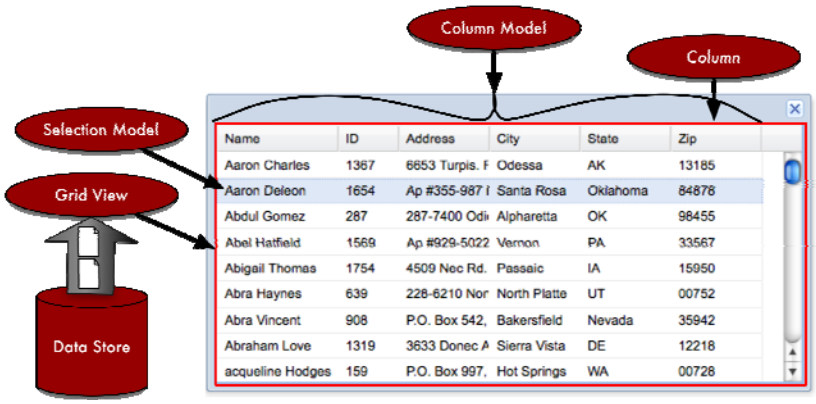


Figure 6.2 The GridPanel's five supporting classes, the DataStore, GridView, ColumnModel, Column and selection model.

In Figure 6.2, 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` poperty, 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.

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.

6.2 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.

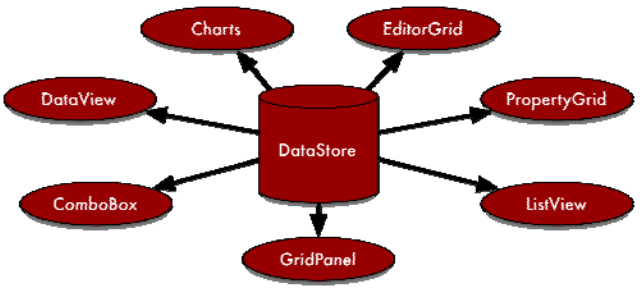


Figure 6.3 The DataStore and the classes it feeds data to. This illustration does not depict class hierarchy.

As we can see in Figure 6.3, 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.

6.2.1 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.

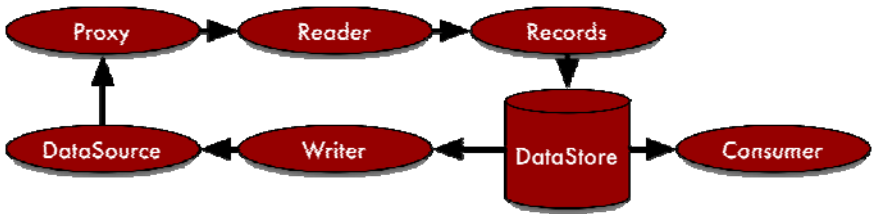


Figure 6.4 The data flow from a data source to a DataStore consumer.

As we can see in Illustration 6.4, 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.

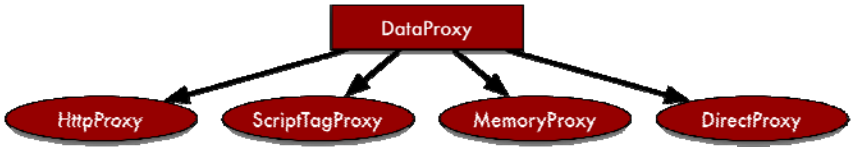


Figure 6.5 The DataProxy and its four descendants. Each are responsible for retrieving data from a specific data source.

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 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.

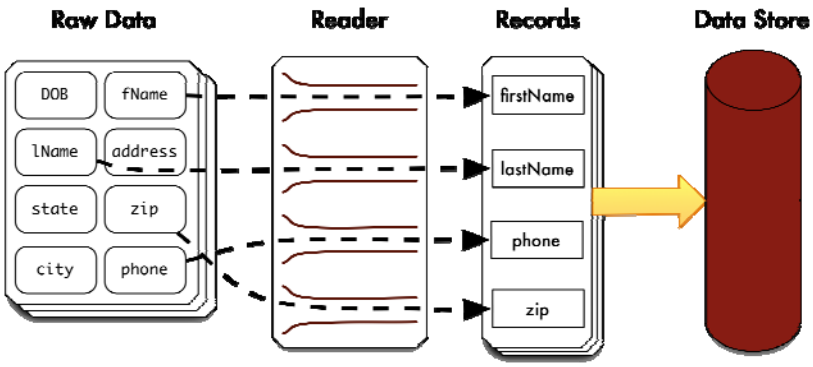


Figure 6.6 A Reader maps raw or unformatted data so that they can be inserted into Records, which then get spooled into a data Store

As you can see in Figure 6.6, 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`.

6.3 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.

6.3.1 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 6.1 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
    {  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({                                     // 5
    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 the 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 the 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 crated. From a 30,000 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.

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 to 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 am 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.

6.3.2 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 the we can configure the GridPanel itself.

Listing 6.2 Creating an ArrayStore

```
var colModel = new Ext.grid.ColumnModel([                    // 1
     {
         header    : 'Full Name',
         sortable  : true,
         dataIndex : 'fullName'                               // 2
     },
     {
         header    : 'State',
         dataIndex : 'state'
     }
]);

var gridView = new Ext.grid.GridView();                      // 3
var selModel = new Ext.grid.RowSelectionModel({              // 4
    singleSelect : true
})

var grid = new Ext.grid.GridPanel({                          // 5
    title      : 'Our first grid',
```

```
    renderTo   : Ext.getBody(),
    autoHeight : true,
    width      : 250,
    store      : store,                                    // 6
    view       : gridView,
    colModel   : colModel,
    selModel   : selModel
});
```

{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 **{1}**, 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 Tore, ColumnModel, GridView and Selection Model. Loading the page will generate a grid panel that looks like the following illustration.

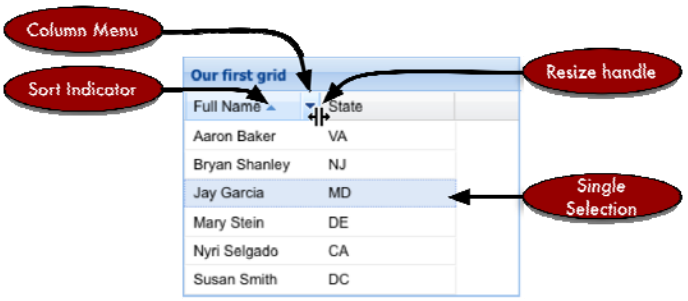


Figure 6.7 Our first grid rendered on screen demonstrating the single-select configured RowSelectionModel and the sortable "Full Name" column.

As we can see in Figure 6.7, 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.

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.

6.4 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.

6.4.1 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 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.

6.4.2 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 6.3 Creating an ArrayStore

```
var recordFields = [                                                    //1
    { 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'   }
];

var remoteJsonStore = new Ext.data.JsonStore({                          //2
    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` **{1}**, 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.

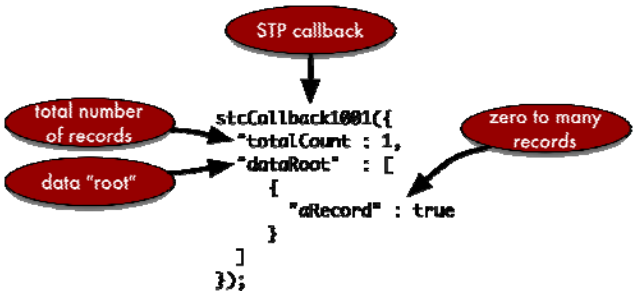


Figure 6.8 The format that a remote server must respond within

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.

Now that we've got that out of the way, we can move on to configure our advanced `ColumnModel`.

6.4.3 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 6.4 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.

Listing 6.5 Configuring oour advanced column model

```
var columnModel = [
     {
         header    : 'ID',
         dataIndex : 'id',
         sortable  : true,
         width     : 50,
         resizable : false,
         hidden    : true,                                          // 1
         renderer  : colorTextBlue                                  // 2
     },
     {
         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                          // 3
        },
        {
            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 `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`.

6.4.4 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 6.6 Configuring oour 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.

6.4.5 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 6.7 Placing our GridPanel inside a Window

```
new Ext.Window({                                                    // 1
    height : 350,
    width  : 550,
    border : false,
    layout : 'fit',
    items  : grid
}).show();

Ext.StoreMgr.get('ourRemoteStore').load({                           // 2
     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 6.7, 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 **{2}**, 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.

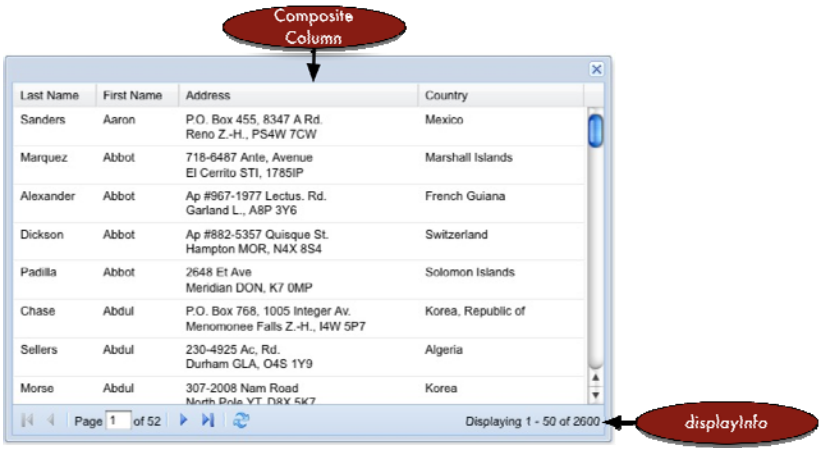


Figure 6.9 the results of our advanced paging GridPanel implementation.

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 6.10 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.

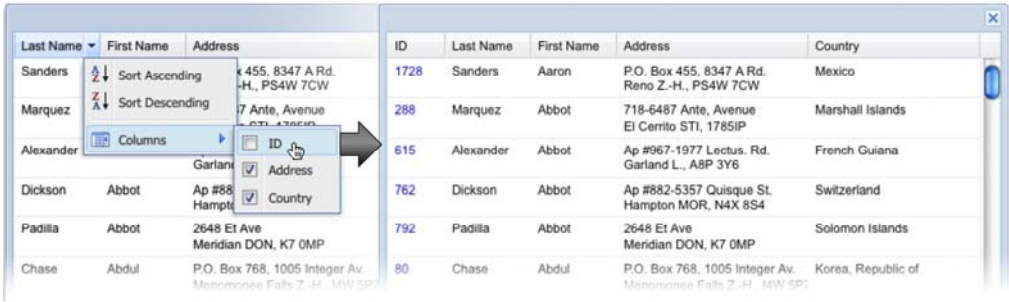


Figure 6.11 Enabling the ID column via the Columns menu.

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 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.

6.4.6 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 6.8 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}',
           lastName ,  firstName);

   Ext.MessageBox.alert('', msg);
}

var doRowDblClick = function(thisGrid, rowIndex)  {                     // 2
   var record = thisGrid.getStore().getAt(rowIndex);
   doMsgBoxAlert(record);
}

var doRowCtxMenu = function(thisGrid, rowIndex, evtObj) {               // 3
   evtObj.stopEvent();                                                  // 4

   var record = thisGrid.getStore().getAt(rowIndex);

   if (! thisGrid.rowCtxMenu) {                                         // 5
       thisGrid.rowCtxMenu = new Ext.menu.Menu({
           items : {
               text    : 'View Record',
               handler : function() {
                  doMsgBoxAlert(record);
               }
           }
       });
   }

   thisGrid.rowCtxMenu.showAt(evtObj.getXY());
}
```

{1} Create a utility method to generate an Ext alert dialog
{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 is 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` stops the native context menu from showing.

We then move on to get a reference to the record the event occurred on, exactly like the other event handler. 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 record reference we created just a bit earlier.

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.

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 6.9 Attaching our event handlers to our grid

```
var grid = {
     xtype            : 'grid',
     columns          : columnModel,
     store            : remoteJsonStore,
     loadMask         : true,
     bbar             : pagingToolbar,
     autoExpandColumn : 'addressCol',
     stripeRows       : true,
     listeners        : {                                    // 1
        rowdblclick    : doRowDblClick,
        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?

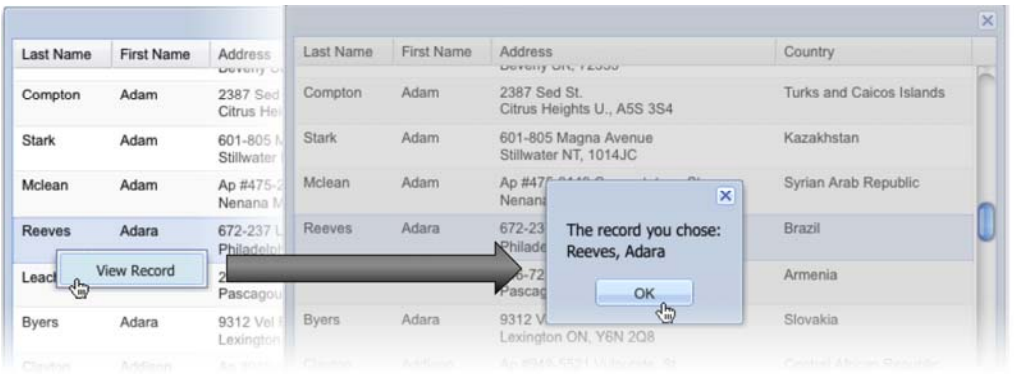


Figure 6.12 The results of our context menu handler addition to our advanced gird.

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.

6.5 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 `GriPanel`, 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`.