Ext JS 4 Cookbook—Exploring Further

We've compiled an additional 20 recipes with more useful hints and tips to help you to get the most out of Sencha's Ext JS 4 framework. Following the same format as the book, these extra recipes cover a wide variety topics and we hope they further broaden your knowledge of the framework.

Creating your applications folder structure

This recipe will give you Sencha's best-practice application folder structure. By following these guidelines, your application will be well organized, easily maintainable, and sharing components between applications will be straight-forward.

In addition to these benefits you can use the Ext JS SDK tools to easily create and deploy an optimized copy of your application when you're ready to go live. We have explained how to do this in the recipe *Building your Application with Sencha's SDK Tools*, in *Chapter 12*, *Advanced Ext JS for the Perfect App*.

To illustrate this, we'll create an application to list our project's enhancement log.

Getting ready

We recommend having a web server (for example, Apache or IIS) installed and running on your development computer.

How to do it...

1. If you are not following an MVC pattern for your application use the following directory structure:
    - enhancementLog
        - app
            - enhancement
                - EnhancementGrid.js
        - extjs
        - resources
            - css
            - images
            - ....
        - app.js
        - index.html
2. If you are following an MVC pattern use the following structure:
    - enhancementLog
        - app
            - controller
                - Enhancement.js
            - model
                - Enhancement.js
            - store
                - Enhancement.js
            - view
                - enhancement
                    - Enhancement.js
        - extjs
        - resources
            - css
            - images
        - app.js
        - index.html

How it works...

- The directory `enhancementLog` is where you will store the application files
- The directory `app` will contain the applications classes (in their respective group's sub-folder) and will follow the naming conventions described in the *There's more* section
- The directory `extjs` will contain a copy of the Ext JS framework and associated files
- The directory `resources` is intended for images, additional CSS, and any other static resources the application requires
- `index.html` is the application's entry-point
- `app.js` contains the application's launch logic

There's more...

Sencha has also defined some naming conventions for Ext JS 4 applications that should be followed as they will keep your code organized, well-structured, and readable.

The conventions are summarized as follows:

Class naming conventions

- You are strongly encouraged to use alphanumeric characters only for naming classes
- Camel case for the top-level namespaces and class names only
- Everything else should be lowercase
- Acronyms should also follow the camel case convention

Best practice: `EnhancementLog.enhancement.views.EnhancementGrid`

Bad practice: `Enhancement_Log.enhancement.enhancement_Grid`

Source file naming conventions

- Where possible, class names should map directly to the file path in which they exist
- One class per file only

The `EnhancementLog.controller.Enhancement` class will exist in `/enhancementLog/app/controller/Enhancement.js`.

Methods and variables naming conventions

- You are strongly encouraged to use alphanumeric characters only for naming methods and variables
- Camel case method names and variables

[Example: `getEnhancementStore()` or `decodeEnhancementJson()`]

Properties naming conventions

- You are strongly encouraged to use alphanumeric characters only for naming properties
- Camel case properties that are not static constants
- Static constants should always be uppercase

[Example: `Ext.MessageBox.YES = "Yes"`]

See also

- The *Dynamically Loading Ext JS Classes* recipe in *Chapter 1*, *Classes, Object-Oriented Principles, and Structuring your Application*, as this folder structure is required when loading classes dynamically.
- The folder structure demonstrated will be used in the book. In particular see *Chapter 12*, *Architecting your application with MVC*.

Encoding and decoding HTML strings

Ext JS has easy to use features to encode and decode HTML strings. These are particularly useful if you have a requirement to read or send escaped HTML entities from the user interface.

How to do it...

1. Encode the following HTML with `Ext.String.htmlEncode`:

```
console.log(Ext.String.htmlEncode("<strong>This is text & it is
bold</strong>"));
//"&lt;strong&gt;This is text &amp; it is bold&lt;/strong&gt;"
```

2. Decode the following string to return it to HTML:

```
console.log(Ext.String.htmlDecode("&lt;strong&gt;This is text
&amp; it is bold&lt;/strong&gt;"));
//"<strong>This is text & it is bold</strong>"
```

How it works...

The `Ext.String` class has a number of useful static methods for working with strings. `htmlEncode` and `htmlDecode` are used for encoding and decoding HTML entities.

`htmlEncode` works by looping through each character of the string that's passed to it and swapping <, >, &, and " for their HTML character equivalents (`&lt`, `&gt`, `&amp`, and `&quot`).

`htmlDecode` does exactly the same as `htmlEncode` but instead converts the characters back from their HTML equivalents.

See also

- The *Encoding and Decoding JSON Data* recipe in *Chapter 2*, *Manipulating the Dom, Handling Events, and Making AJAX Requests*, for an example demonstrating how to work with JSON

Detecting available browser features

With so many browsers and platforms on the market it's often difficult to ensure consistency, especially when older browsers don't support features introduced with CSS3 and HTML5.

While Ext JS 4 is compatible with popular browsers (IE 6+, Firefox 3.6+, Chrome 6+, Safari 3+, and Opera 10.5+) by behaving differently depending on the browser, there are differences and incompatibilities between them.

Ext JS offers two useful classes for determining the platform (`Ext.is`) and determining the supported features of an environment (`Ext.supports`). These come in handy when you are looking to introduce, for example, CSS3 features that are only supported by more modern browsers. Checking available browser features enables you to ensure user interface consistency for all users.

How to do it...

1. Determine the platform we are working with:

```
console.log(Ext.is.Mac);
console.log(Ext.is.Windows);
console.log(Ext.is.Desktop);
```

2. Determine the browser:

   ```
   console.log(Ext.isChrome);
   console.log(Ext.isIE);
   console.log(Ext.isIE6);
   console.log(Ext.isWebKit);
   ```

3. Find out if the browser supports certain CSS3 features:

   ```
   console.log(Ext.supports.CSS3BorderRadius);
   console.log(Ext.supports.CSS3BoxShadow);
   console.log(Ext.supports.CSS3DTransform);
   console.log(Ext.supports.CSS3LinearGradient);
   ```

4. Geolocation can be detected with:

   ```
   console.log(Ext.supports.GeoLocation);
   ```

5. We can check for HTML5 features such as:

   ```
   console.log(Ext.supports.History);
   ```

6. Drawing features can be detected using properties like:

   ```
   console.log(Ext.supports.Canvas);
   console.log(Ext.supports.Svg);
   console.log(Ext.supports.Vml);
   ```

7. Finally, if you just want to know the user agent it can be done using:

   ```
   console.log(Ext.userAgent);
   ```

How it works...

Determining the platform the user is browsing from is done using the `Ext.is` class. As you might expect the `Ext.is` class queries the browsers navigator object picking out information from the platform property. Using a series of regular expression tests, the framework is able to determine where the user is browsing from. The `Ext.is` class knows if the client is Android, Blackberry, Desktop, Linux, Mac, Phone, Standalone, Tablet, Windows, iOS, iPad, iPhone, or iPod.

The `Ext.supports` class is useful when we are looking to determine the features available in the browser. The `Ext.supports` class performs a series of tests, unique to the desired feature, to determine a value for its many properties.

The `Ext.userAgent` property returns the value of `navigator.userAgent.toLowerCase()`, which allows us to perform our own queries on it.

Evaluating object types and values

This recipe shows you how to determine an object's data type as a string or a Boolean using features built into Ext JS. Additionally, determining if a variable is null, undefined, a zero-length array, or a zero-length string is demonstrated.

How to do it...

1. Start by defining a few variables with a range of types:

   ```
   var anArray = ["Item 1", "Item 2"];
   var aBoolean = true;
   var aDate = new Date();
   var aFunction = function(){
       return true;
   };
   var aNumber = 26;
   var anObject = {};
   var aString = 'String of Text';
   var anotherString = '26';
   var undefinedVar;
   ```

2. Determine the variable type (as a string):

   ```
   console.log(Ext.typeOf(aBoolean)); //"boolean"
   console.log(Ext.typeOf(aFunction)); //"function"
   ```

3. Determine the variable type (as a Boolean):

   ```
   console.log(Ext.isArray(anArray)); //true
   console.log(Ext.isBoolean(aBoolean)); //true
   console.log(Ext.isDate(aDate)); //true
   console.log(Ext.isDefined(undefinedVar)); //false
   console.log(Ext.isEmpty(undefinedVar)); //true
   console.log(Ext.isFunction(aFunction)); //true
   console.log(Ext.isIterable(anArray)); //true
   console.log(Ext.isNumber(aNumber)); //true
   console.log(Ext.isNumeric(anotherString)); //true
   console.log(Ext.isObject(anObject)); //true
   console.log(Ext.isPrimitive(aNumber)); //true
   console.log(Ext.isString(aString)); //true
   ```

How it works...

The `Ext.typeOf` method returns the name of the object type as a string. It can return: undefined, null, string, number, Boolean, date, function, object, array, regexp, element, textnode, or whitespace.

`Ext.isDefined` returns true when the value is defined and `Ext.isEmpty` returns true if the value is empty (null, undefined, a zero-length array, or a zero-length string). `Ext.isEmpty` takes an optional parameter `allowEmptyString (bool)`, which forces it to return true when the value is a zero-length string.

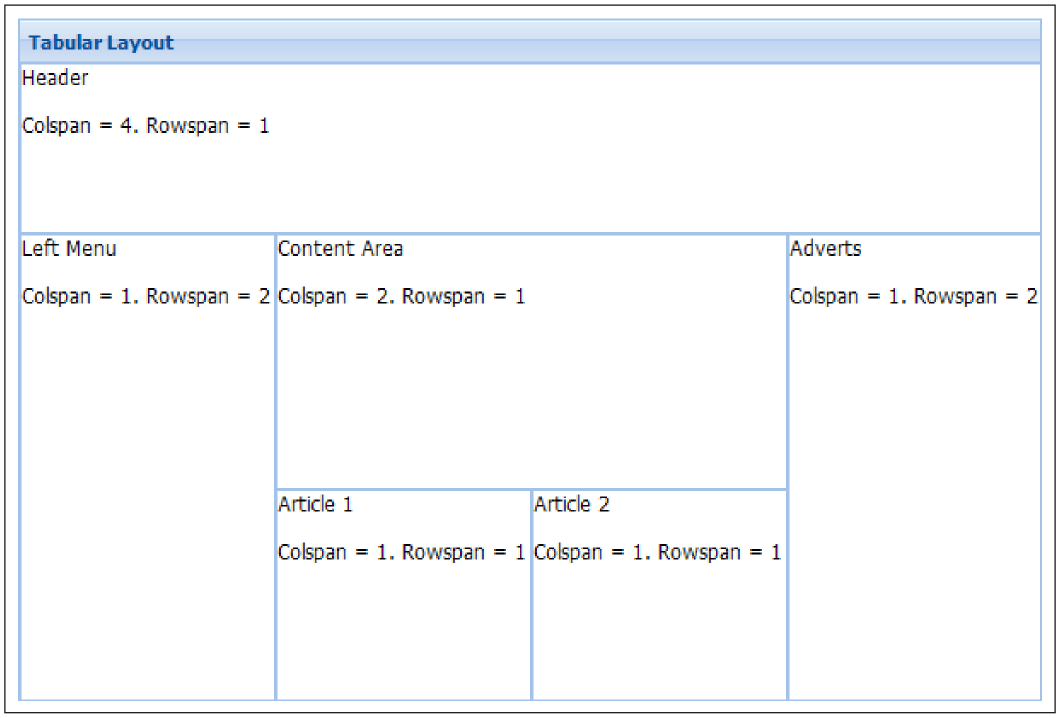
Finally, `Ext.isArray`, `Ext.isBoolean`, `Ext.isDate`, and so on, returns Boolean for the values passed in. If, for example, `value` is an array, then `Ext.isArray(value)` will return true but `Ext.isBoolean(value)` will return false.

Presenting tabular data with the TableLayout

Although scorned by web designers everywhere for their misuse as a layout construction tool, tables are often still the best choice for presenting data to users.

Ext JS provides a straightforward way of building up HTML tables based on the simple principle that each component in a Panel's items collection acts as a table cell. This is achieved by using the `Ext.layout.container.TableLayout` class, which manages the table creation process.

In this recipe, we will take a trip back to 1999 and create a layout for our website using a table. We will walk through creating this simple table-based layout, which demonstrates various configuration options including cells spanning multiple rows and multiple columns, adding CSS classes to cells and adding custom attributes to the table's elements. The final result will look like the following screenshot:



How to do it...

1. We will start by creating a simple `Ext.panel.Panel`, which will house our table layout, and render it to the document's body.

```
Ext.create('Ext.panel.Panel', {
   title: 'Tabular Layout',
   width: 600,
   height: 400,
   renderTo: Ext.getBody()
});
```

2. We next define each of the table's cells and add them to the `items` collection in order from left to right, top to bottom (i.e. working along each row then down to the next one).

```
Ext.create('Ext.panel.Panel', {
    title: 'Tabular Layout',
    width: 600,
    height: 400,
    renderTo: Ext.getBody(),
    items: [{
        width: 600,
        height: 100,
        html: 'Header'
    }, {
        width: 150,
        height: 300,
        html: 'Left Menu'
    }, {
        width: 300,
        height: 150,
        html: 'Content Area'
    }, {
        width: 150,
        height: 300,
        html: 'Adverts'
    }, {
        height: 150,
        width: 150,
        html: 'Article 1'
    }, {
        height: 150,
        width: 150,
        html: 'Article 2'
    }]
});
```

3. Now we define the layout for the outer Panel as a `TableLayout` and define how many columns we want the table to have by adding the following code to the Panel's configuration object.

```
layout: {
    type: 'table',
    columns: 4
},
```

4. Finally, we decorate each of the panel's children with the `colspan` and `rowspan` properties to tell the `TableLayout` how to configure the table. This is identical to how we would do it if we were creating a plain HTML table:

```
Ext.create('Ext.panel.Panel', {
    title: 'Tabular Layout',
    width: 600,
    height: 400,
    renderTo: Ext.getBody(),
    layout: {
        type: 'table',
        columns: 4
    },
    items: [{
        width: 600,
        height: 100,
        html: 'Header',
        colspan: 4
    }, {
        width: 150,
        height: 300,
        html: 'Left Menu',
        rowspan: 2
    }, {
        width: 300,
        height: 150,
        html: 'Content Area',
        colspan: 2
    }, {
        width: 150,
        height: 300,
        html: 'Adverts',
        rowspan: 2
    }, {
        html: 'Article 1',
        height: 150,
        width: 150
    }, {
        html: 'Article 2',
        height: 150,
        width: 150
    }]
});
```

5. When no `rowspan` or `colspan` are defined, they are omitted from the table cell's markup giving a default value of `1` for each.

How it works...

The `layout` configuration option tells the panel that we want a `TableLayout` and we want it to have four columns. The `column` property is used when building the table's structure and decides which child items are included in which row of the resulting table. If this option is omitted, the table would be rendered in a single row.

The `rowspan` and `colspan` properties given to each child item are applied directly to the underlying table cell's markup, leaving the browser to natively calculate the structure.

There's more...

There are a few more configuration options available that give us finer control over the table that is generated by the layout.

TableLayout item configuration options

Along with the `rowspan` and `colspan` options that we have seen earlier, the following configuration options can be added to the items within a container that has a `TableLayout`:

- `cellId`: An ID that is applied to the table cell
- `cellCls`: A CSS class that is applied to the table cell wrapping the item

TableLayout configuration options

There are three further configuration options that can be applied to the `TableLayout` itself. They all give us control over the attributes that each level of the table possesses (table, row, and cell). They are named `tableAttrs`, `trAttrs`, and `tdAttrs` and accept an object whose name/value pairs are passed to the `DomHelper` class when creating the `<table>`, `<tr>`, and `<td>` elements respectively.

For example, to add a `valign` attribute to the `<td>` tags created within the table we can add the following code:

```
layout: {
  type: 'table',
  columns: 4,
   tdAttrs: {
     valign: 'top'
  }
}
```

See also

- The recipes covering HBox and VBox layouts, which can be used in conjunction with each other to create complex and dynamic layouts

Absolutely positioning components

The `Ext.layout.container.Absolute` class inherits from the previous layout we discussed—`Ext.layout.container.Anchor`. This layout allows us to position a component's children absolutely within it, based on a set of specified x and y coordinates.

With this functionality we are also able to size the children using the `Anchor` layout's properties and syntax.

How to do it...

1. Firstly we create a panel and render it to the document's body. This will house our absolutely positioned panel:

```
Ext.create('Ext.panel.Panel', {
    width: 400,
    height: 400,
    renderTo: Ext.getBody(),
    title: "Absolute Layout Panel",
    items: []
});
```

2. Add in a child panel:

```
Ext.create('Ext.panel.Panel', {
    width: 400,
    height: 400,
    renderTo: Ext.getBody(),
    title: "Absolute Layout Panel",
    items: [{
        xtype: 'panel',
        title: 'Panel 1',
        html: Panel 1'
    }]
});
```

3. Set the layout and configure the child panel with x and y coordinates and an anchor value:

```
Ext.create('Ext.panel.Panel', {
    width: 400,
    height: 400,
    renderTo: Ext.getBody(),
    title: "Absolute Layout Panel",
    layout: 'absolute',
    items: [{
```

```
                xtype: 'panel',
                title: 'Panel 1',
                html: 'Positioned 50px from left, 50px from top',
                anchor: '50% 35%',
                x: 50,
                y: 50
            }]
        });
```

How it works...

The `x` and `y` configuration options tell the layout where to position the panel. These values are applied to the left and top CSS style properties and the panel is given a position value of `absolute`.

Positioning panel headers

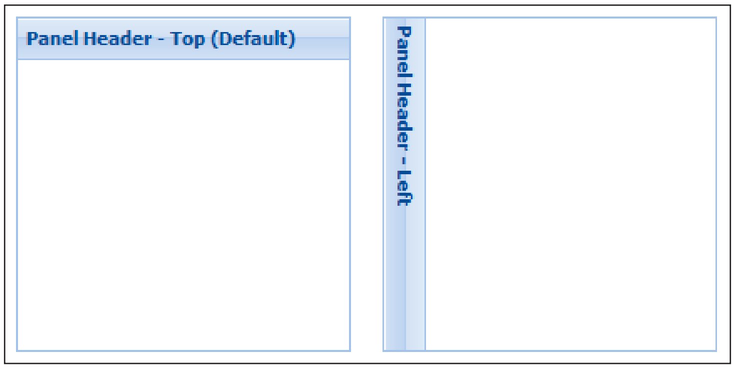
It's possible to change the position of a panel's header in Ext JS 4. This short recipe will demonstrate how to achieve this.

How to do it...

1. Create a panel and render it to the document's body. The title will appear at the top by default:

   ```
   Ext.create('Ext.panel.Panel', {
       title: 'Panel Header - Top (Default)',
       width: 500,
       height: 200,
       style: 'margin: 50px',
       renderTo: Ext.getBody()
   });
   ```

2. Use the `headerPosition` config option to render the title on the left. It will mean the header moves from the top position (as shown in the left image) to the left position (as shown in the right image):



```
Ext.create('Ext.panel.Panel', {
    title: 'Panel Header - Left',
    headerPosition: 'left',
    width: 500,
    height: 200,
    style: 'margin: 50px',
    renderTo: Ext.getBody()
});
```

How it works...

The `headerPosition` configuration option for an `Ext.panel.Panel` can take the following four values: `left`, `right`, `top`, or `bottom`. By default, the `headerPosition` is set to `top`.

As Ext JS lays out the panel, it calls the `addDocked()` method, which adds an `Ext.panel.Header` with your desired configuration (`title`, `border`, `frame`, `tools`, and so on) to the `dockedItems` collection for the panel.

Specifying `top` or `bottom` for the `headerPosition`, docks the header to the top or bottom of the panel. The orientation for the header is set to horizontal for you.

When `left` or `right` are specified, the framework docks the header appropriately and changes the orientation to vertical. The styles are switched; the title is drawn using the `Ext.draw.Component` class, with the text rotated 90 degrees; and the header's layout is set to a VBoxLayout so the header's items are stacked on top of each other.

Animated filtering of a DataView

Ext JS is shipped with a fantastic plugin for the `Ext.view.View` class that animates any presentation changes after updates to the underlying data store. For example, if the store is filtered, the filtered nodes are faded out and the remaining nodes will animate to their new positions.

This recipe will demonstrate adding this plugin (called `Ext.ux.DataView.Animate`) to our bug data view and show how easy it is to have our filter changes animated.

Getting ready

Once again we will build from the previous recipes' code and we will start with the complete Bug data view created in the *Displaying a details window after clicking a DataView node* recipe in *Chapter 4*, *UI Building Blocks—Trees, Panels, and Data Views*.

How to do it...

1. We start by including the `Ext.ux.DataView.Animate` plugin's source (which can be found in the `examples/ux` folder of the library's download package) into our HTML file by linking to it with a `script` tag:

   ```
   <script type="text/javascript" src="ux/Ext.ux.DataView.Animated.
   js"></script>
   ```

2. Next, we add the plugins config option to our `Ext.view.View` definition and instantiate our `Ext.ux.DataView.Animated` class within the plugins array. We pass in two configuration options which define the duration the animations will last and the `idProperty` of the store's Model:

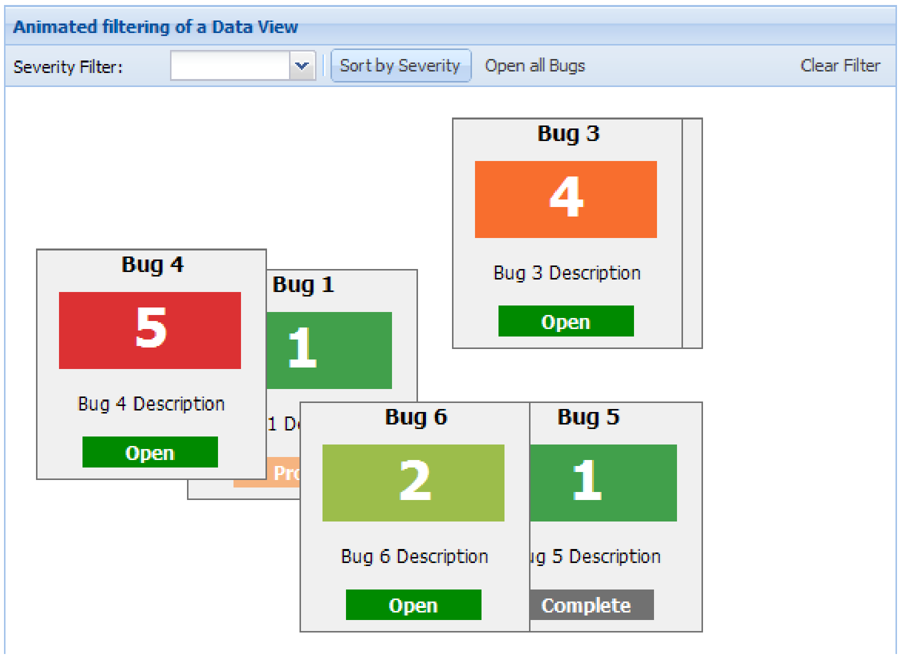
   ```
   ...
       plugins: [
           new Ext.ux.DataView.Animated({
               duration: 500,
               idProperty: 'id'
           })
       ]
   ...
   ```

3. Finally, we update our severity filter combobox's select event handler code. We must suspend any store events while clearing the existing filters so the plugin doesn't try to react to those events:

```
...
select: function(combo, value, options){
    // clear existing filter, prevent change events firing
    dataview.getStore().suspendEvents();
    dataview.getStore().clearFilter();
    dataview.getStore().resumeEvents();

    dataview.getStore().filter('severity', combo.getValue());
}
...
```

The DataView can be seen in mid-animation as shown in the following screenshot (borders have been added to make the positions of each node easier to observe):



How it works...

By creating an instance of the `Ext.ux.DataView.Animated plugin` and including it in the DataView's plugins configuration array, it is initialized during the DataView's construction.

The plugin works by listening to the `datachanged` event and then recalculating the position of each of the nodes and then animates them to those new locations. The process for achieving this is outlined as follows:

- Cache the store's current record set on initialization
- When the store's data is changed (on filter, update, and so on)
- Get the added, removed, and remaining records using the cached copy
- Fade out all the Removed nodes
- Collect the positions of all the Remaining nodes
- Position these nodes absolutely using the collected values
- Calculate their new positions
- Animate the transition between these two locations
- Fade in all the added nodes
- Re-cache the store's current record set

See also

- For a more advanced look at plugins, we recommend reading the recipe *Advanced functionality with plugins*, in *Chapter 12*.

Specifying a form field's length limits

Ext JS fields have several built-in validation checks that can be enabled and configured through some simple options. One of these checks allows us to specify the length range that a value, entered by a user, must fall into.

This recipe will describe how to configure a form field to automatically enforce this validation. We will do this by setting a minimum length of three characters and a maximum length of ten on our `First Name` field.

Getting ready

Once again we will be working with the example form created in the first recipe of *Chapter 4*, so please make sure you are familiar with its structure.

How to do it...

1. We start by adding the `minLength` configuration option to the `First Name` field's definition.

```
{
    xtype: 'textfield',
    fieldLabel: 'First Name',
    name: 'FirstName',
    labelAlign: 'top',
    cls: 'field-margin',
    flex: 1,
    minLength: 3
}
```

2. We then use the `maxLength` option to provide the upper length limit of the field.

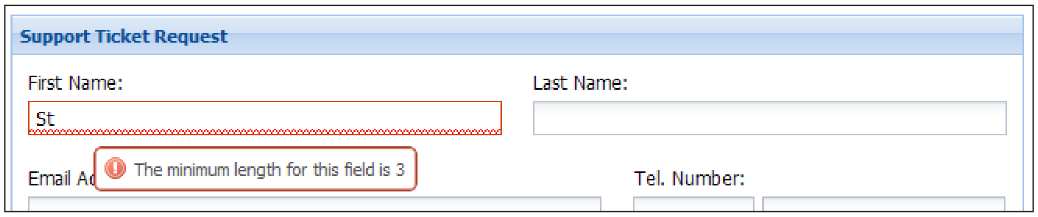
```
{
    xtype: 'textfield',
    fieldLabel: 'First Name',
    name: 'FirstName',
    labelAlign: 'top',
    cls: 'field-margin',
    flex: 1,
    minLength: 3,
    maxLength: 10
}
```

How it works...

The `Ext.form.field.Text` class contains a `getErrors` method, which runs through all of the configured validation the field has. If the test does not pass an error message is pushed onto an errors collection. This collection of error messages is then used to display to the user.

A field's validity is checked whenever the field's value changes and so real-time feedback about validation changes can be provided to the user.

By default, if the length restrictions are violated, the field is marked by a red wavy line and, when the user hovers over the field, a tooltip is displayed containing an error message. This can be seen in the following screenshot:



St

The minimum length for this field is 3

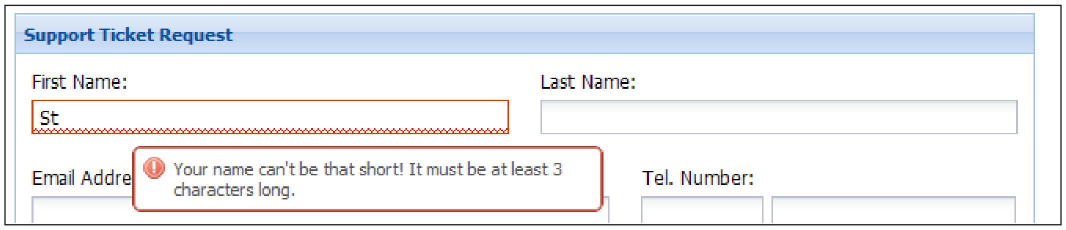
There's more...

In addition to customizing the valid length a field's value must contain, we can also configure what error message appears in the tooltip that is displayed to the user.

We simply specify the `minLengthText` and `maxLengthText` properties of the text field and the framework will use these when creating the error message collection. If you would like to include the configured `minLength` or `maxLength` value in the error message (as the default message does) we can include a placeholder (`{0}`), which will be filled automatically using the `Ext.String.format` method:

```
{
    xtype: 'textfield',
    fieldLabel: 'First Name',
    name: 'FirstName',
    labelAlign: 'top',
    cls: 'field-margin',
    flex: 1,
    minLength: 3,
    minLengthText: 'Your name can\'t be that short! It must be at
    least {0} characters long.',
    maxLength: 10,
    maxLengthText: 'Your name can\'t be more than {0} characters
long!'
}
```

The output of these options being set can be seen as follows:



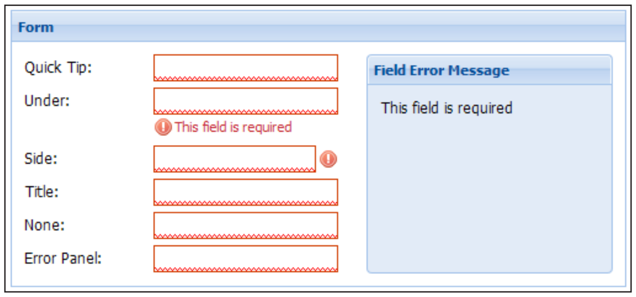
Your name can't be that short! It must be at least 3 characters long.

See also

- For more complex validation see the recipes on VTypes
- The recipes explaining how to create and configure form fields in *Chapter 3*, *Laying Out your Components*

Displaying validation alerts to the user

This recipe will demonstrate different built-in methods for displaying these messages to your users:



This field is required

This field is required

How to do it...

1. Start by creating a simple form panel. We've opted for an hbox layout in this instance to allow us to add a second panel to house the error message for the last text field:

```
Ext.create('Ext.form.Panel', {
    title: 'Form',
    width: 500,
    height: 225,
    renderTo: Ext.getBody(),
    style: 'margin: 50px',
    layout: {
        type: 'hbox',
        align: 'stretch'
    },
    items: []
});
```

2. In the items collection add the left-hand panel and the form fields for the form. Pay particular attention to the msgTarget config option:

```
Ext.create('Ext.form.Panel', {
    ...
    items: [{
        flex: 1,
        padding: 10,
        border: false,
        defaults: {
            allowBlank: false
        },
        defaultType: 'textfield',
        items: [{
            fieldLabel: 'Quick Tip'
        }, {
            fieldLabel: 'Under',
            msgTarget: 'under'
        }, {
            fieldLabel: 'Side',
            msgTarget: 'side'
        }, {
            fieldLabel: 'Title',
            msgTarget: 'title'
        }, {
            fieldLabel: 'None',
            msgTarget: 'none'
        }, {
            fieldLabel: 'Error Panel',
            msgTarget: 'error-container',
            minLength: 5
        }]
    }]
});
```

3. Finally, add the right-hand side panel with a container and give the container an ID of `error-container`:

```
Ext.create('Ext.form.Panel', {
    items: [{
        ...
    }, {
        xtype: 'panel',
        width: 200,
        margin: '5 5 5 0',
        padding: '10',
        title: 'Field Error Message',
        frame: true,
        layout: 'fit',
        items: [{
            xtype: 'component',
            id: 'error-container'
        }]
    }]
});
```

How it works...

Altering how the validation messages are displayed to the user is done with the field's `msgTarget` config option. Valid values for this are:

- `qtip (default)`: the message will be shown in a **Quick Tip** when the user hovers over the field
- `under`: the message is displayed below the field as soon as it has been validated
- `title`: the error is displayed in a standard HTML `title` attribute that will appear when the user hovers over the field
- `side`: a red exclamation icon is shown to the right-hand side of the field and will display the error message in a **Quick Tip** when hovered over
- `none`: only the red wavy line is displayed with no error message shown
- `[element/component id]`: the message is shown in the element/component with the specified ID

The error message text location varies depending on the value for `msgTarget`. For example, if you leave `msgTarget` field blank, then the error message will be displayed in a **Quick Tip** when they hover over it.

In the final field we set the `msgTarget` to the component ID error-container. This tells the framework to display the error message text in the `innerHTML` of the `error-container` element. The other options will be taken care of by the framework and the required styles and markup are added as needed.

See also

- The recipes of *Chapter 5*, *Loading, Submitting, and Validating Forms*, about how to apply validation to fields. Try combining the various options discussed here with those examples.
- *Chapter 6*, which describes all the form field types.

Dynamically generate forms from loaded JSON

When dealing with complex and dynamic data structures it's often the case that we don't know exactly how a form will be structured until runtime. We frequently need to delegate this form generation to the server, but how do we translate that server generated form configuration to our Ext JS application?

This recipe will walk through how to create a simple login form that is generated on the fly, by data stored in a JSON file.

Getting started

This recipe involves making AJAX calls and so requires to be run on a web server.

How to do it...

1. We start by creating a simple form panel that will be the container for all of our dynamically loaded form fields. We give the panel modest configuration to get it to display in our document:

```
var formPanel = Ext.create('Ext.form.Panel', {
    title: 'Login Form',
    width: 500,
    height: 200,
    renderTo: Ext.getBody(),
    style: 'margin: 50px'
});
```

2. Now, we create a JSON file (we have named it `formStructure.json`) that will be the target of our AJAX request. This file will contain the data that we will use to create our form fields. In this example, we are loading from a plain JSON file directly but this could easily be a PHP (or other server-side language) script that generates and outputs JSON data:

```
{
    "success": true,
    "fields": [{
        "xtype": "textfield",
        "fieldLabel": "Username",
        "name": "username",
        "allowBlank": false
    }, {
        "xtype": "textfield",
        "fieldLabel": "Password",
        "name": "password",
        "inputType": "password",
        "allowBlank": false
    }]
}
```

3. Our next step is to make an AJAX call to load this JSON file so that we can process its contents and create our form. We start by adding our AJAX call and displaying the response in the console:

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

4. Within our success handler we must decode the response that was received into a JavaScript object. We will then create a check that the server was successful in building our form data (obviously in this case, it will always be true, but if this JSON was being built dynamically by the server, there is a chance it could fail and so including a success indicator is good practice).

```
Ext.Ajax.request({
    url: 'formStructure.json',
    success: function(response, options){
        var jsonResponse = Ext.decode(response.responseText);
        if(jsonResponse.success){
            // we have the data so create the form!
        }
    }
});
```

5. The final step is to add the received fields to the form itself. This is a simple task because the items in the JSON file's fields array are all valid Ext JS configuration objects and so can be passed straight to the add method of formPanel:

```
Ext.Ajax.request({
    url: 'formStructure.json',
    success: function(response, options){
        var jsonResponse = Ext.decode(response.responseText);

        if(jsonResponse.success){
            formPanel.add(jsonResponse.fields);
        }
    }
});
```

How it works...

By returning valid Ext JS configuration objects from the server it becomes a trivial task adding these component definitions to an existing Ext JS form.

The `add` method of `formPanel` parses these configuration objects in the same manner as the objects within a panel's `items` collection are parsed in a regular panel instantiation. The object's `xtype` is picked up and used to instantiate the correct Ext JS component, configured with all the other properties within the object.

The new form fields are then rendered to the form's content area and any layout calculations performed, so the new components are displayed correctly.

See also

- The *Constructing a complex form layout* recipe in *Chapter 5*, which describes how to create a complex form layout. Why not try and recreate that layout from a dynamically loaded JSON file?

Positioning field labels

Customizing forms to suit your application is relatively straightforward with Ext JS 4. Changing the position of your field's label is a common task and this recipe aims to demonstrate how you can achieve this.

How to do it...

1. Start by adding a text field to a form panel:

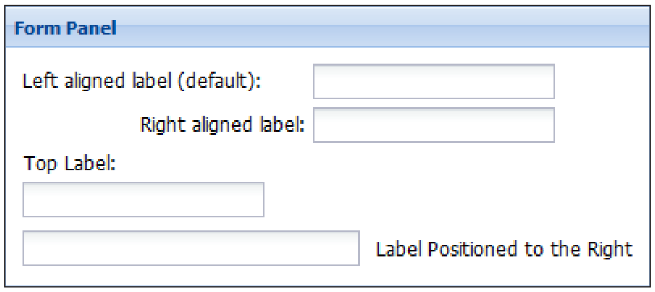
```
Ext.create('Ext.form.Panel', {
    title: 'Form Panel',
    width: 400,
    renderTo: Ext.getBody(),
    bodyPadding: 10,
    defaults: {
        labelWidth: 175
    },
    items: [{
        xtype: 'textfield',
        fieldLabel: 'Left aligned label (default)'
    }]
});
```

2. Add two additional fields with the labelAlign option defined as follows:

```
Ext.create('Ext.form.Panel', {
    ...
    items: [
    ...
    {
        xtype: 'textfield',
        fieldLabel: 'Right aligned label',
        labelAlign: 'right'
    }, {
        xtype: 'textfield',
        fieldLabel: 'Top Label',
        labelAlign: 'top'
    }]
});
```

3. It's also possible to position the label to the right of the field using the hbox layout:

```
Ext.create('Ext.form.Panel', {
    ...
    items: [
       ...
    {
        xtype: 'container',
        layout: {
            type: 'hbox',
            align: 'middle'
        },
        items: [{
            xtype: 'textfield',
            flex: 1,
            hideLabel: true,
            inputId: 'right-positioned-label'
        }, {
            xtype: 'label',
            text: 'Label Positioned to the Right',
            margins: '10 0 0 10', //add top & right margin
            forId: 'right-positioned-label'
        }]
    }]
});
```



How it works...

The first field we added is a straight-forward text field with a label defined. By default Ext JS 4 will position the label to the left of the field and the text is aligned to the left.

In examples two and three we added an additional `labelAlign` config option, which allows us to change the position and alignment of the label. In addition to the value `left`, the `labelAlign` option can take two other values—`top` and `right`:

- `top` as you might expect, positions the `fieldLabel` over the field, aligning the text to the left.
- `right` on the other hand, positions the label to the left but aligns the text to the right. You can see this demonstrated in the field `Right aligned label`.

The final example demonstrates positioning the `fieldLabel` to the right. This is done by creating a container with an `hbox` layout. The container's items collection has two components:

1. The `textfield`: we have configured this with `flex: 1` (so it will stretch to fill the remaining space) and `hideLabel: true` to render it without a label
2. The `label`: this is the second item in the collection to ensure it is in the second column of the `hbox`

We use the `forId` and `inputId` to indicate which form element we are binding the label to. Naturally these must both match. It's the same as:

```
<label for="right-positioned-label ">Right Label</label>
<input type="text" id="right-positioned-label " />
```

There's More...

In addition to positioning a field's label, it is possible to further customize the label with the configuration options set out in `Ext.form.Labelable`. Config options worth noting are:

- `labelSeparator: String`: This lets you define the character(s) inserted at the end of the label's text. By default this is a colon.
- `labelPad: Number`: This is the amount of space between the field and its label. By default this is 5 pixels.
- `labelWidth: Number`: This is the width of the field's label. By default, this is 100 pixels.

See Also

- To read more about the hbox layout go to *Chapter 3*, *Laying Out your Components*, which explains it in detail.

Configuring a grid from metadata

We don't always know the structure that a grid's data is going to have, or the way the columns should be displayed, when writing our JavaScript. This is often something that must be generated on the server at runtime and sent to the client alongside the actual data set.

Thankfully Ext JS allows us to include this metadata in its loaded data packet and use it to configure our grids and stores on the fly. This means that we don't need to create a Model class or define our grid's columns at design time giving us huge flexibility.

In this recipe, we will explore how to create a grid that has its columns loaded in through its store's AJAX call. We will then continue by learning how to throw away the pre-written Model definition and create one on the fly using more metadata provided by the server.

Getting ready

This recipe uses the generic `Ext.data.Model` created in the `invoices-model.js` file. These files will need to be included in your HTML like so:

```
<script type="text/javascript" src="invoices-model.js">
</script>
```

How to do it...

1. After including our Model file we will start by defining the columns' metadata inside our JSON file. We will use the `invoices.json` file used earlier in the chapter as a base, but add the `metaData` property to its top level:

```
...
"metaData": {
    "columns": [{
        "header": "Client",
        "dataIndex": "Client",
        "flex": 1
    }, {
        "header": "Date",
        "dataIndex": "Date"
    }]
}
...
```

2. After doing this we create a Store, which will load data from the updated `invoices-meta.json` file:

```
var invoiceStore = Ext.create('Ext.data.Store', {
    autoLoad: true,
    model: 'Invoice',
    proxy: {
        type: 'ajax',
        url: 'invoices-meta.json',
        reader: {
            type: 'json',
            root: 'rows'
        }
    }
});
```

3. Next we override the reader's `onMetaChange` method, which is called after a store load, if the JSON data returned includes a property called `metaData`. Our overridden method will call its original implementation and then, if present, uses the columns property to reconfigure the grid:

```
var invoiceStore = Ext.create('Ext.data.Store', {
    autoLoad: true,
    model: 'Invoice',
    proxy: {
        type: 'ajax',
        url: 'invoices-meta.json',
        reader: {
            type: 'json',
            root: 'rows',
            onMetaChange: function(meta){
                Ext.data.reader.Json.prototype.onMetaChange.
call(this, meta);

                if(meta.columns){
                    invoicesGrid.reconfigure(invoicesGrid.
getStore(), meta.columns);
                }
            }
        }
    }
});
```

4. We are now able to create a simple grid, with an empty array assigned to the `columns` configuration option, bound to the `invoiceStore` we created:

```
var invoicesGrid = Ext.create('Ext.grid.Panel', {
    title: 'Chapter 8 - Grids',
    height: 300,
    width: 600,
    store: invoiceStore,
    columns: [],
    renderTo: Ext.getBody()
});
```

How it works...

The `onMetaChange` method of a reader class is executed from its `readRecords` method, which parses the Model data from the JSON data loaded by the store, but only if the JSON data contains the `metaData` property.

We use this method to hook into the record processing workflow and perform our own processing of the metadata.

In our override we start by calling the reader's original version of the method so that we don't change the behavior of it. We then add to it by using the grid's `reconfigure` method. This method allows us to attach a different store to the grid and specify the column model that the grid will use.

We call it by passing in the grid's existing store (because we don't want to change this) and the new columns array that exists in the `meta` parameter. The grid will then update all of the necessary internals and refresh the grid for us with our new configuration in place.

There's more...

Now that we have successfully configured the grid's columns we also want to configure the store's Model based on the metadata loaded.

This element is handled almost exclusively by the framework itself through the `onMetaChange` method that we overwrote in the first section. This method will automatically look for the existence of a `fields` property inside the `metaData` object. If this property is found, the reader will create a temporary Model class configured with these fields:

1. We start by adding the fields configuration to our JSON file:

```
...
    "metaData": {
    "columns": [{
         "header": "Client",
        "dataIndex": "Client",
        "flex": 1
    }, {
        "header": "Date",
        "dataIndex": "Date"
    }],
    "fields": [{
        "name": "InvoiceID",
        "type": "string"
    }, {
        "name": "Client",
        "type": "string"
    }, {
        "name": "Description",
        "type": "string"
    }, {
        "name": "Date",
        "type": "date",
        "dateFormat": "c"
    }, {
        "name": "Amount",
        "type": "float"
    }, {
        "name": "Currency",
        "type": "string"
    }, {
        "name": "Status",
        "type": "string"
    }]
}
...
```

2. Next we create a store pointing to our updated JSON file (`invoices-meta-fields.json`) and replace its `model` configuration option (because we are creating it on the fly) with the `fields` option, set to an empty array:

```
var invoiceStoreFields = Ext.create('Ext.data.Store', {
    autoLoad: true,
    fields: [],
    proxy: {
        type: 'ajax',
        url: 'invoices-meta-fields.json',
        reader: {
            type: 'json',
            root: 'rows',
            onMetaChange: function(meta){
                Ext.data.reader.Json.prototype.onMetaChange.
call(this, meta);

                if(meta.columns){
                    invoicesGridFields.
reconfigure(invoicesGridFields.getStore(), meta.columns);
                }
            }
        }
    }
});
```

3. Finally, we create a simple grid, containing an empty columns configuration, bound to this new store:

```
var invoicesGridFields = Ext.create('Ext.grid.Panel', {
    title: 'Chapter 8 - Grids',
    height: 300,
    width: 600,
    store: invoiceStoreFields,
    columns: [],
    renderTo: Ext.getBody()
});
```

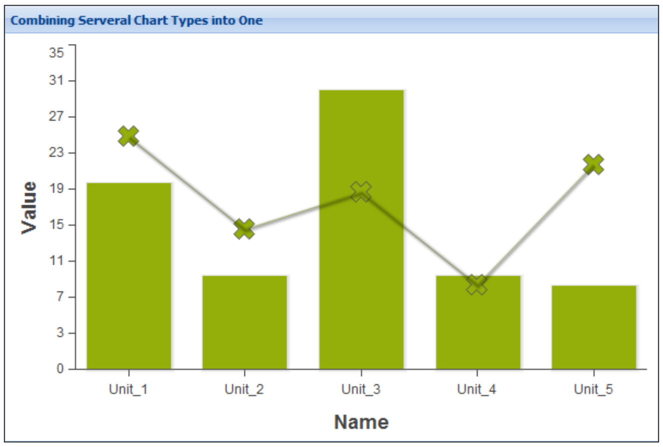
This new setup will load both the data's field structure and the columns that the grid will display on the fly based entirely on data created on the server.

See also

- Working with the framework's data package was important in this recipe. For a more detailed look at the specific components of the data package take at *Chapter 7*, *Working with the Ext JS Data Package*.
- Other recipes that look at dynamically creating Ext JS components from the server-side data can be found in *Chapters 5*, *Working with the Ext JS Data Package* and *Chapter 6*, *Using and Configuring Form Fields*.

Combining several chart types into one

The Ext JS charting package is extremely flexible, allowing you to render charts on top of each other. This example will show you how to render a column chart and a line chart together enabling users to make comparisons between data.



Getting ready

This recipe requires the use of a web server for serving the charts' data. A JSON file is provided with example data.

How to do it...

1. We start by creating an `Ext.data.Model` with three fields: `name`, `value`, and `value2`:

```
Ext.define('Chart', {
    extend: 'Ext.data.Model',
    fields: [{
        name: 'name',
        type: 'string'
    }, {
        name: 'value',
        type: 'int'
    }, {
        name: 'value2',
        type: 'int'
    }]
});
```

2. Now create an `Ext.data.Store` with an AJAX proxy to load our remote data:

```
var store = Ext.create('Ext.data.Store', {
    model: 'Chart',
    proxy: {
        type: 'ajax',
        url: 'BarChart.json',
        reader: {
            type: 'json',
            root: 'data'
        }
    },
    autoLoad: true
});
```

3. Create a panel with a `fit` layout and render it to the document's body:

```
var panel = Ext.create('Ext.Panel', {
    width: 600,
    height: 400,
    title: 'Combining Several Chart Types into One',
    layout: 'fit',
    items: [
        ...
    ],
    style: 'margin: 50px',
    renderTo: Ext.getBody()
});
```

4. In the panel's `items` collection add an `Ext.chart.Chart` bound to the store created in step 2. Give the chart `Numeric` and `Category` axes:

```
var panel = Ext.create('Ext.Panel', {
    ...
    items: [{
        xtype: 'chart',
        animate: true,
        store: store,
        axes: [{
            type: 'Numeric',
            position: 'left',
            fields: ['value'],
            title: 'Value',
            minimum: 0,
            maximum: 35,
            decimals: 0
        }, {
            type: 'Category',
            position: 'bottom',
            fields: ['name'],
            title: 'Name'
        }],
        series: [{
            ...
        }]
    },
    ...
});
```

5. Finally, we're going to add two series to the chart, a column series and a line series. Set the `yField` to `value` and `value2` respectively:

```
var panel = Ext.create('Ext.Panel', {
    ...
    items: [{
        ...
        series: [{
            type: 'column',
            axis: 'left',
            xField: 'name',
            yField: 'value'
        }, {
            type: 'line',
            axis: 'left',
            xField: 'name',
            yField: 'value2',
            markerConfig: {
                type: 'cross',
                radius: 8
            }
        }]
    },
    ...
});
```

How it works...

The clever bit in this example happens in the `series` configuration, which, as you can see, takes an array of series. By providing more than one `Ext.chart.series.Series` we are able to render two separate visualizations on one chart.

The first series, `Ext.chart.series.Column`, is used for drawing the charts columns. The `xField` is set to name so the name appears across the bottom of the chart. The `yField` is set to our integer value value.

The second series, `Ext.chart.series.Line`, has `value2` set in its `yField`. As a result, the line will be drawn based on these values. The marker configuration is used to change the look of the line's markers.

See also

- The *Creating a bar chart with external data* recipe from *Chapter 10*, *Drawing and Charting*, for a more detailed explanation of how to create a bar chart
- The *Creating a line chart with updating data* recipe from *Chapter 10*, which explores line charts in more detail
- The first few recipes from *Chapter 7*, *Working with the Ext JS Data Package*, for more information on using Stores and Models from the Ext JS data package
- The *Customizing labels, colors, and axes* recipe from *Chapter 10*, for an example on how to change the look and feel of your chart

Displaying detailed information when hovering over charts

When a user hovers their mouse over a chart it's often useful to display more detailed information to the user. This recipe will demonstrate how to add tooltips to the visualizations markers.

Getting ready

This recipe requires the use of a web server for serving the charts data. A JSON file is provided with example data.

How to do it...

1. Start by defining a model for the data we are loading into the chart:

```
Ext.define('Chart', {
    extend: 'Ext.data.Model',
    fields: [{
        name: 'name',
        type: 'string'
    }, {
        name: 'value',
        type: 'int'
    }]
});
```

2. Create a store with an AJAX proxy. Set `autoLoad: true` to load the data automatically:

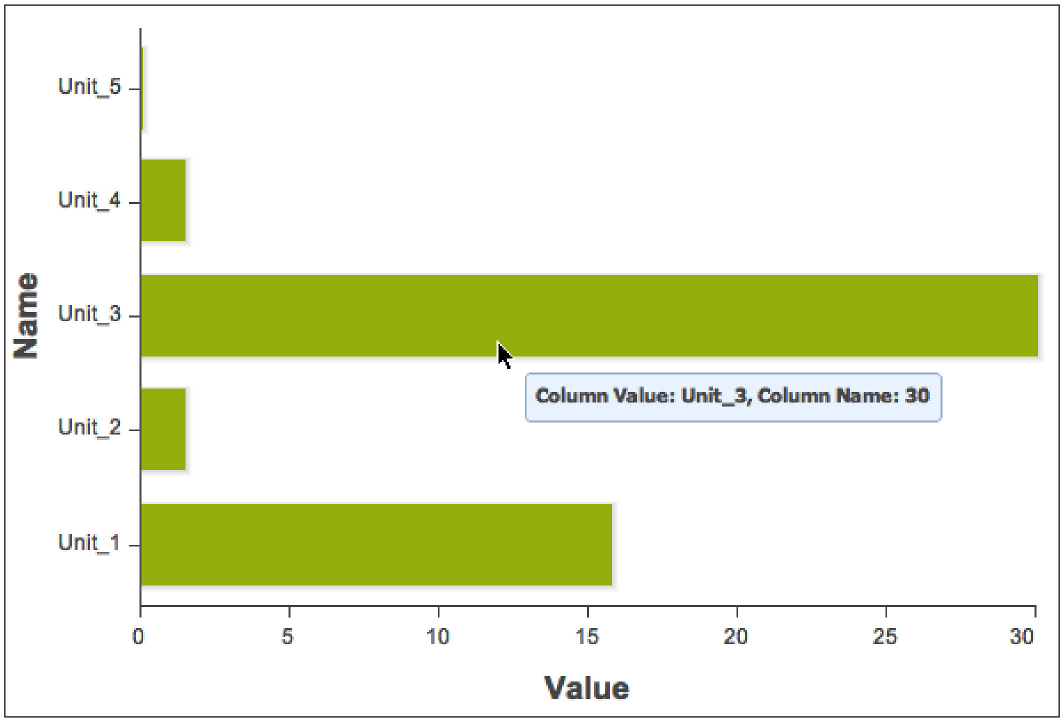
```
var store = Ext.create('Ext.data.Store', {
    model: 'Chart',
    proxy: {
        type: 'ajax',
        url: 'BarChart.json',
        reader: {
            type: 'json',
            root: 'data'
        }
    },
    autoLoad: true
});
```

3. Create a basic chart rendered to the document's body. Give it a `Numeric` and `Category` axis and set a `bar` series:

```
var chartBar = Ext.create('Ext.chart.Chart', {
    width: 600,
    height: 400,
    animate: true,
    store: store,
    axes: [{
        type: 'Numeric',
        position: 'bottom',
        fields: ['value'],
        title: 'Value'
    }, {
        type: 'Category',
        position: 'left',
        fields: ['name'],
        title: 'Name'
    }],
    series: [{
        type: 'bar',
        axis: 'bottom',
        xField: 'name',
        yField: 'value'
    }],
    style: 'margin: 50px',
    renderTo: Ext.getBody()
});
```

4. In the `series` configuration add a tip using the `tips` configuration:

```
var chartBar = Ext.create('Ext.chart.Chart', {
    ...
    series: [{
         type: 'bar',
        axis: 'bottom',
        xField: 'name',
        yField: 'value',
        tips: {
            trackMouse: true,
            width: 235,
            height: 28,
            renderer: function(storeItem, item){
                this.setTitle('Column Value: ' + storeItem.
get('name') + ', Column Name: ' + storeItem.get('value'));
            }
        }
    }],
    ...
});
```



How it works...

The `Ext.chart.series.Series` class contains the `Ext.chart.Tip` mixin, which enhances the series by providing tooltips to the chart.

When the user's mouse hovers over a chart item the tip is shown by the `showTip` method in `Ext.chart.Tip`. The item (retrieved by the `Ext.chart.series.SeriesgetItemForPoint` method) over which the user is hovering is passed into the `showTip` method.

We can make use of the entire configuration available with the `Ext.tip.ToolTip` class for customizing the tip. In this example we have set `trackMouse: true`, which ensures the tooltip follows the mouse, a width, and a height.

The renderer is required for updating the tooltip title with the relevant information from the item we are hovering over. The arguments `storeItem` and `item` provide us with the ability to retrieve information about the series item. In this case, we have used the `get` method on the `storeItem` to retrieve the data for the column.

See also

To learn more about creating basic charts see the recipes in *Chapter 10*, which explains each chart type in-depth.

Persisting user-interface state

Preserving a user's settings and configuration within your web app is an important step to help them enjoy using your product. There is nothing worse than having to, for example, increase the size of a pop-up window to make it easier for you to read and then have it revert back to its original size the next time you open it.

Ext JS provides the `Ext.state` package, which gives support for preserving this state (in cookies or local storage, for example) and having it automatically applied to the relevant components when they are created.

This recipe will show how we can leverage this feature to save the basic characteristics of a panel so they are applied after the page is refreshed. We will then look into how to customize the characteristics that are saved so we can extend the model to suit our needs.

How to do it...

1. Our first step is to tell the `Ext.state.Manager` class which provider we want to use to save our state values. We will use the `Ext.state.LocalStorageProvider` class, which will use HTML5 Local Storage to save the components' details:

```
Ext.state.Manager.setProvider(new Ext.state.
LocalStorageProvider());
```

> HTML5 Local Storage is only available in modern browsers so if you need to support older browsers it might be better to use the `Ext.state.CookieProvider` class instead.

2. We now create a simple `Ext.panel.Panel` with the `collapsible` and `resizable` configuration options set to `true`:

```
var panel = Ext.create('Ext.panel.Panel', {
    renderTo: Ext.getBody(),
    title: 'Ext JS 4 Cookbook',
    html: 'Chapter 12 - Advanced Ext JS for the Perfect App',
    height: 500,
    width: 500,
    collapsible: true,
    resizable: true
});
```

3. Include the `stateful` and `stateId` config options giving them the values `true` and `cookbook-panel` respectively:

```
var panel = Ext.create('Ext.panel.Panel', {
    renderTo: Ext.getBody(),
    title: 'Ext JS 4 Cookbook',
    html: 'Chapter 12 - Advanced Ext JS for the Perfect App',
    height: 500,
    width: 500,
    collapsible: true,
    resizable: true,
    stateful: true,
    stateId: 'cookbook-panel'
});
```

4. We can now collapse or resize our panel, refresh the page, and see it return to the size it was before.

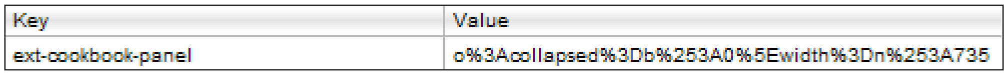
How it works...

Ext JS uses the `Ext.state.Stateful` class as a mixin to provide a way of maintaining state to all components that extend the `Ext.AbstractComponent` class. This class controls when the state is saved and what characteristics of the component are saved.

We started by setting the provider that will be used to save the state details to a storage medium. There are two providers included in the framework—`Ext.state.CookieProvider` and `Ext.state.LocalStorageProvider`. They both extend the `Ext.state.Provider` class. You can create your own custom provider implementation to have your state data saved in your own way, for example, to a server.

The `stateful` property we set to true in step 3 tells this class that we want to monitor state changes and save them. The `stateId` property is also required and must be unique as it is the key that is used to reference the component's state values in the storage provider.

After these configurations are included we can use our developer tools to monitor the contents of the Local Storage and view the encoded state details that have been saved. The following screenshot shows our panel's state after it was resized:



Key	Value
ext-cookbook-panel	o%3Acollapsed%3Db%253A0%5Ewidth%3Dn%253A735

The `Ext.state.Stateful` class works by listening for specific events and saving the current state when they are fired. The `Ext.panel.Panel` components include the `resize`, `expand`, and `collapse` events meaning the panel's state will be saved following these events.

On each save the `Ext.state.Stateful` class calls the component's `getState` method that collects the various property values to save and passes them to the provider class to be saved.

There's more...

As we have mentioned, by default only the collapsed state and size of the component will be saved. We will now show how to include other properties in the state save so we can maintain anything associated with the component.

In our example, we will make the panel's title editable and have the `Ext.state` package save it:

1. We will start with a very similar `Ext.panel.Panel` to our earlier example but we are going to extract it out into its own class extending the `Ext.panel.Panel` class. We will also include a button in its top toolbar that will update its title when clicked:

```
Ext.define('Cookbook.StatefulPanel', {
    extend: 'Ext.panel.Panel',
    renderTo: Ext.getBody(),
    title: 'Ext JS 4 Cookbook',
    html: 'Chapter 12 - Advanced Ext JS for the Perfect App',
    tbar: [{
        xtype: 'button',
        text: 'Update Title',
        handler: function(){
            this.up('panel').setTitle('My New Title');
        }
    }],
    height: 500,
    width: 500,
    collapsible: true,
    resizable: true,
    stateful: true,
    stateId: 'cookbook-panel2'
});

var panel2 = Ext.create('Cookbook.StatefulPanel');
```

2. Next we override our panel's parent class's `getState` method so that we can include our own properties. We first call the parent class's `getState` method so we still include all the usual properties, then we add the panel's `title` property to the object that will be saved:

```
...
getState: function(){
    var state = this.callParent();

    state.title = this.title;

    return state;
}
...
```

3. If we inspect the Local Storage contents (via FireBug or Developer Tools) at this point we will see the panel's title has been included in our state save operation and its value has been saved. However, if we click on our **Update Title** button we see that the Local Storage's value remains unchanged and so the state won't be persisted across page loads.

   In order to have the updated title saved when it is changed, we must include the `titlechange` event as a state save event so the new value is persisted in the Local Storage. We do this by including a constructor in our extended class and calling the `addStateEvents` method (part of the `Ext.state.Stateful` mixin) and passing it the event we want to include. By doing this we tell the mixin to grab a snapshot of the panel's stateful properties and save them immediately after this event fires:

```
...
    constructor: function(config){
        this.callParent([config]);

        this.addStateEvents('titlechange');
    }
...
```

   We will now see the Local Storage values being updated when the **Update Title** button is clicked and see the title persist when the page is refreshed.

Instead of using a specific event to trigger a state save we can do this manually by calling the component's `saveState` method after a property has changed. This is useful when the property you want to persist doesn't fire an event.

Retaining support for the browser back button

Ext JS applications are built around the idea of having a single-page website with the Ext JS framework creating and displaying new UI elements as users move around the app.

This approach moves away from the traditional model of a website where multiple pages are connected via hyperlinks and navigation steps can be retraced by using the browser's **Back** and **Forward** buttons.

By abandoning this traditional approach we lose these buttons' functionality, as the browser never moves forward to a new page. This can cause problems for users who are used to using these buttons to return to the previous section of a website but will instead make them exit the application.

To get around this problem, Ext JS incorporates a utility class (`Ext.util.History`) that allows us to 'fake' page changes and therefore reintroduce the **Back** and **Forward** buttons' functionality.

This recipe will demonstrate how to use this class and how to apply it to a simple application that contains a `Tab` panel. Our aim is to track when the user switches tabs and allow the browser's **Back** and **Forward** buttons to retrace these switches as you would expect a traditional website to.

How to do it...

1. We start by creating an `Ext.tab.Panel`, which will form the basis of our demonstration. We give it three `Ext.panel.Panels` and one `Ext.tab.Panel` as its child components:

```
var tabPanel = Ext.create('Ext.tab.Panel', {
    renderTo: Ext.getBody(),
    height: 400,
    items: [{
        xtype: 'panel',
        title: 'My First Panel',
        html: 'This is the first panel.'
    }, {
        xtype: 'tabpanel',
        title: 'My Second Panel',
        items: [{
            title: 'sub1',
            html: 'Sub panel 1'
        }, {
            title: 'sub2',
            html: 'Sub panel 2'
        }]
    }, {
        xtype: 'panel',
        title: 'My Third Panel',
        html: 'This is the third panel.'
    }, {
        xtype: 'panel',
        title: 'My Fourth Panel',
        html: 'This is the fourth panel.'
    }]
});
```

2. We must now initialize the `Ext.util.History` class by calling its static `init` method:

   ```
   Ext.util.History.init();
   ```

3. In order to maintain a list of the user's history we must hook into our Tab panel's `tabchange` event and record the switch so we can move through each of them later. We do this by making a call to the `add` method of `Ext.util.History`, passing it the new card's ID:

   ```
   tabPanel.on({
       tabchange: function(tabPanel, newCard, oldCard, opts){
           Ext.util.History.add(newCard.id);
       },
       scope: this
   });
   ```

4. Finally we must listen for the change event of `Ext.util.History`, which fires when the current history item is changed (that is, when the **Back**/**Forward** button is clicked). This handler gets passed one parameter containing the value that was passed to the `add` method (in our case the panel's ID). In the function we want to get the relevant component and make it the active panel in its `Tab` panel and any `Tab` panels further up the component tree:

   ```
   Ext.util.History.on('change', function(token){
       if (!Ext.isEmpty(token)) {
           var panel  = Ext.getCmp(token);
           if (panel) { // don't do anything if we can't find the
   card relating to the hash token
               owner = panel.ownerCt;
               // keep moving up the component tree until there are
   no more components
               while (owner && owner.is('tabpanel')) {
                   owner.setActiveTab(panel);
                   // update vars for the next loop
                   panel = owner;
                   owner = owner.ownerCt;
               }
   ```

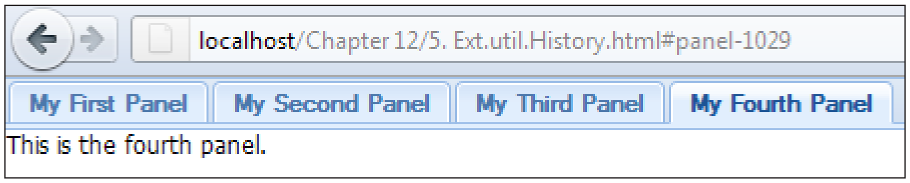
```
            }
        }
    }, this);
```

5. We can now click through our **Tab** panel's tabs and use the browser's **Back** and **Forward** buttons to retrace these steps.

How it works...

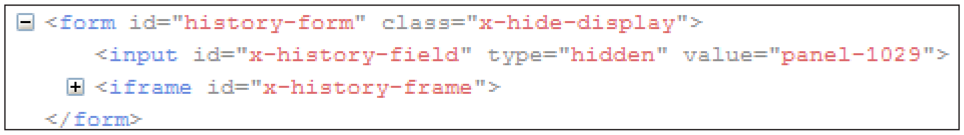
The `Ext.util.History` class works by changing the hash URL of the page (for example, `www.myapp.com#panel1`) when a call to its add method is made. Browsers interpret this as a change in page and so record it in their history collection enabling the **Back** and **Forward** buttons. The framework constantly checks for changes in the hash URL (that is, from a Back button click) and will fire its `change` event when it has.

If we open our example page and navigate to one of the tabs, we can see the URL changes to include the current tab's ID:



This is the fourth panel.

We can also see that a hidden form has been added to our page and it holds the current tab's ID in its `value` attribute:



```
<form id="history-form" class="x-hide-display">
    <input id="x-history-field" type="hidden" value="panel-1029">
  <iframe id="x-history-frame">
</form>
```

After step 3 we can see the browser's **Back** and **Forward** buttons working and the URL changes to the previous tab's ID. In step 4 we must recognize that the hash URL has changed and translate it into a state change in our application.

We use the `Ext.util.History` class's `change` event, which fires when the hash URL of the page changes and passes the new state's hash to the handling function, to perform our 'page change' logic.

We start by checking that a hash token has been provided and, if it hasn't then we simply move on. If we do have a token, then we use it with the `Ext.getCmp` method to retrieve the component it refers to. Now that we have a reference to the panel we want to display, we can start to process it.

After checking the panel we want to show exists, we get a reference to its parent container (which in our case is the Tab panel we created in step 1) so we can tell it to show our desired panel.

The final step uses a `while` loop to traverse up the component hierarchy of `Tab` panels. We use this approach because we want to guarantee that the panel and all its parents are visible, so our solution will work with nested `Tab` panels.

Consider having a `Tab` panel (tab panel 1), which contains a child `Tab` panel (tab panel 2), and it is a child panel of tab panel 2 that we want to make visible. If we did not use a loop and simply changed tab panel 2's active tab to our desired panel, we can't guarantee that tab panel 1's active tab is tab panel 2, meaning our panel isn't actually visible. If we use a loop we can move up the tree and ensure that tab panel 1's active tab is tab panel 2, therefore ensuring the desired panel is visible at all levels.

Our loop's condition states to keep going until the `owner` variable is false (that is, has no value as we are at the root of the tree) or it isn't a tab panel.

Our first step in the loop is to make our `owner` tab panel's active tab our panel. We then update our `owner` and `panel` variables so our next iteration references the next level of the tree. Our `panel` variable now points to the previous `owner` (so it is the component being made visible) and the `owner` variable becomes the `panel` component's owner. If the conditions are met then the loop repeats.

Now this logic is in place, when the user hits the **Back** button the previous tab's ID will be passed into this function and it will be made active allowing the user to move back through the application's screen.

There's more...

This technique works well but has an undesirable side-effect of exposing auto-generated IDs to the user in the address bar. This looks very untidy; it doesn't give the user any indication of where they are in the application and addresses can become very long when proper namespaces are used. We will now show how we can make these hash URLs more user friendly:

1. We start by adding a custom property to the `Ext.util.History` class called `idMapping` and assigning it an empty object literal. This will act as a lookup table for our components' user-friendly value and their actual ID:

   ```
   Ext.util.History.idMapping = {};
   ```

2. Next, we replace one of our `Tab` panel's inline children with an instance of a custom Panel subclass. In the class' `initComponent` method we give it a property called `mappingID` with a value of `NicePanel`. This is the name that will be shown in the address bar instead of the generated ID.

   After this we add this `mappingID` to the `idMapping` object of `Ext.util.History` with the panel's ID as its value:

```
Ext.define('Cookbook.Panel1', {
    extend: 'Ext.Panel',
    title: 'Panel 1',
    initComponent: function(){
        this.callParent(arguments);


        this.mappingID = 'NicePanelName';

        Ext.util.History.idMapping[this.mappingID] = this.id;
    }
});
```

3. Our next change is within the `onTabChange` function, which adds the new panel's ID to the `Ext.util.History` class. We update this line to add the panel's `mappingID` first, and if it isn't present then to fall back to the panel's ID:

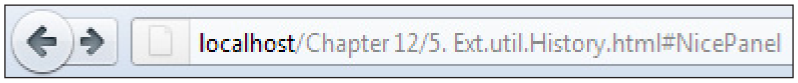
```
var onTabChange = function(tabPanel, newCard, oldCard, opts){
    Ext.util.History.add(newCard.mappingID || newCard.id);
};
```

4. The final change we must make involves the interpretation of this new value in the `change` event handler of `Ext.util.History`. If the specified token exists in the `idMapping` object, then we use its value as the panel's ID otherwise we use the token as it is because it must be a plain ID:

```
Ext.util.History.on('change', function(token){
    if (!Ext.isEmpty(token)) {
        token = !Ext.isEmpty(Ext.util.History.idMapping[token]) ?
Ext.util.History.idMapping[token] : token;
        var card = Ext.getCmp(token);
        if (card) { // don't do anything if we can't find the card
relating to the hash token
            owner = card.ownerCt;
            // keep moving up the component tree until there are
no more components
            while (owner && owner.is('tabpanel')) {
                owner.setActiveTab(card);
                // update vars for the next loop
                card = owner;
```

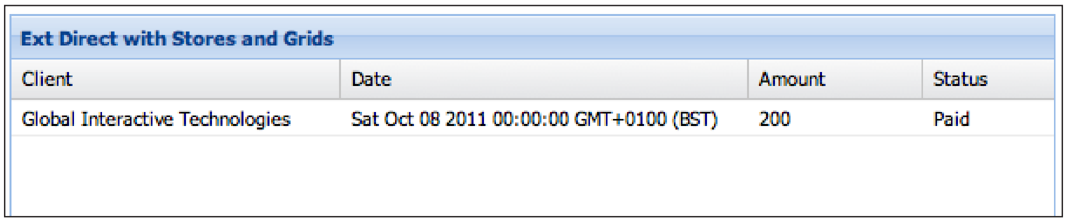
```
                    owner = owner.ownerCt;
                }
            }
        }
    }, this);
```

We can now see that when navigating to this tab the address bar will show the user-friendly hash:


localhost/Chapter 12/5. Ext.util.History.html#NicePanel

Loading stores and grids with Ext Direct

In this recipe, we will explore how Ext Direct can be integrated with our stores for populating and updating tabular data.


Ext Direct with Stores and Grids

Client	Date	Amount	Status
Global Interactive Technologies	Sat Oct 08 2011 00:00:00 GMT+0100 (BST)	200	Paid

Getting ready

We're going to need our working web server again that's capable of parsing PHP and the server-side stack entitled *Extremely Easy Ext.Direct integration with PHP*. The stack is provided by community member *j.bruni* and can be downloaded here: `http://www.sencha.com/forum/showthread.php?102357-Extremely-Easy-Ext.Direct-integration-with-PHP`

If you followed the steps in the *Getting Started with Ext Direct* recipe in *Chapter 12*, this should already be configured.

How to do it...

1. Configure our PHP with the necessary tools to route Ext Direct requests and provide an API to the framework:

```
<?php
require 'Server/ExtDirect.php';
class GridClass {
  public function readGrid() {
    $data = array(
      'InvoiceID' => 1,
      'Client' => 'Global Interactive Technologies',
      'Date' => '2011-10-08 00:00:00',
      'Amount' => 200.00,
      'Status' => 'Paid');

    return $data;
  }
}
ExtDirect::provide('GridClass');
?>
```

2. Add a call to the API inside a `script` tag in our HTML file:

```
<script type="text/javascript" src="GridRouter.php?javascript"></
script>
```

3. Define an `Invoice` model to represent our data:

```
Ext.define('Invoice', {
    extend: 'Ext.data.Model',
    idProperty: 'InvoiceID',
    fields: [{
        name: 'InvoiceID',
        type: 'string'
    }, {
        name: 'Client',
        type: 'string'
    }, {
        name: 'Date',
        type: 'date',
        dateFormat: 'c'
    }, {
        name: 'Amount',
        type: 'float'
    }, {
        name: 'Status',
        type: 'string'
    }]
});
```

4. Create an `Ext.data.Store` with a `DirectProxy` for handling our Ext Direct requests. Bind the store to the `Invoice` model:

```
var invoiceStore = Ext.create('Ext.data.Store', {
    autoLoad: true,
    model: 'Invoice',
    proxy: {
        type: 'direct',
        api: {
            create: undefined,
            read: Ext.php.GridClass.readGrid,
            update: undefined,
            destroy: undefined
        }
    }
});
```

5. Create a basic grid panel and render it to the document's body:

```
Ext.create('Ext.grid.Panel', {
    title: 'Ext Direct with Stores and Grids',
    height: 150,
    width: 600,
    store: invoiceStore,
    columns: [{
        header: 'Client',
        dataIndex: 'Client',
        flex: 1
    }, {
        header: 'Date',
        dataIndex: 'Date'
    }, {
        header: 'Amount',
        dataIndex: 'Amount'
    }, {
        header: 'Status',
        dataIndex: 'Status'
    }],
    renderTo: Ext.getBody(),
    style: 'margin: 50px'
});
```

How it works...

In this example our API exposes the `readGrid` method from our `GridClass` class. The `RemotingProvider` is added with the following configuration:

```
{
    "url": "/ExtDirect/GridRouter.php",
    "type": "remoting",
    "namespace": "Ext.php",
    "descriptor": "Ext.php.REMOTING_API",
    "actions": {
        "GridClass": [
            {
                "name": "readGrid",
                "len": 0
            }
        ]
    }
}
```

Now that the framework has parsed the actions we are able to make requests via `Ext.php.GridClass.readGrid`. The store is bound to an `Invoice` model as we have done in previous examples throughout the book. However, as we want to make requests with Ext Direct we require a `DirectProxy`. The `Ext.data.proxy.Direct` class contains the necessary functionality to make requests through our `RemotingProvider`.

We specify the calls for **CRUD** (**Create, Read, Update, and Destroy**) actions via an `api` configuration. It's here that we define the Ext Direct function calls the framework should make for each request.

Having set `autoLoad: true` on the store, we should notice that a second request is made to `GridRouter.php` to return the data for our grid:

```
{
    "type": "rpc",
    "tid": 1,
    "action": "GridClass",
    "method": "readGrid",
    "result": {
        "InvoiceID": 1,
        "Client": "Global Interactive Technologies",
        "Date": "2011-10-08 00:00:00",
        "Amount": 200,
        "Status": "Paid"
    }
}
```

Although the example only returns one record/row, Ext Direct is capable of returning multiple rows. The result object ends up looking like:

```
"result": [
        {
         ...
        },
        {
         ...
        }
    ...
]
```

See also

- Check out the *Loading and submitting forms with Ext Direct* and *Getting started with Ext Direct* recipes in *Chapter 12* that provide an introduction to Ext.Direct and explore how to incorporate it into your forms
- For information about how to handle exceptions with Ext.Direct take a look at the next recipe

Handling exceptions with Ext Direct

Being able to handle system errors is vital as you may be required to provide feedback to a user if something stops working as expected. This exception handling is not intended for application-level errors, such as field validation, but for server-side errors.

Ext Direct requires that the server-side stack has the ability to switch the displaying of server-side exceptions on or off. Obviously this is for security reasons as you wouldn't want to expose more than you have to in a production environment. This recipe will demonstrate how to handle exceptions with Ext Direct.

Getting ready

We're going to need our working webserver again that's capable of parsing PHP and the server-side stack entitled *Extremely Easy Ext.Direct integration with PHP*. The stack is provided by community member *j.bruni* and can be downloaded here: `http://www.sencha.com/forum/showthread.php?102357-Extremely-Easy-Ext.Direct-integration-with-PHP`

If you followed the steps in the recipe *Getting Started with Ext Direct* in *Chapter 12*, this should already be configured.

How to do it...

1. Start by creating `ExceptionRouter.php` with a method that throws an exception:

   ```
   <?php
   require 'Server/ExtDirect.php';
   class MyClass {
     public function exampleMethod() {

       throw new Exception('Error Message');

     }
   }
   ExtDirect::$debug = true;
   ExtDirect::provide('MyClass');
   ?>
   ```

2. Include the API definition in the HTML file via a `script` tag:

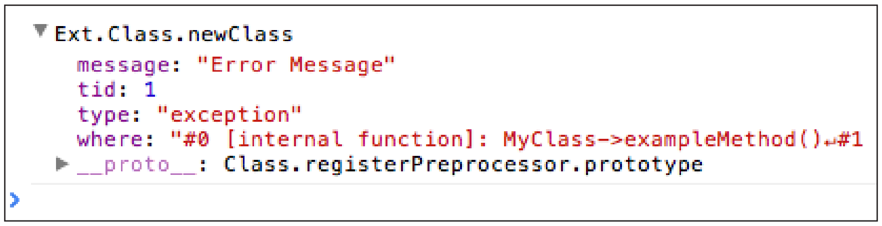
   ```
   <script type="text/javascript" src="ExceptionRouter.
   php?javascript"></script>
   ```

3. Make a call to the method that we've created. The response it returns will contain an exception.

   ```
   Ext.php.MyClass.exampleMethod(function() {
   });
   ```

4. Catch the exception and output the details of it to the console and the alert box:

   ```
   Ext.Direct.on('exception', function(e) {
       Ext.Msg.alert('Exception', e.message);
       console.log(e);
   });
   ```



```
▼Ext.Class.newClass
   message: "Error Message"
   tid: 1
   type: "exception"
   where: "#0 [internal function]: MyClass->exampleMethod()↵#1
  ▶__proto__: Class.registerPreprocessor.prototype
>
```

How it works...

When an exception occurs the server-side stack is required to return certain information to the client:

- Type: a string representing the type of response. Following an exception this will have a value of `"exception"`.
- Message: this is the error message we have included in the new `Exception instance;`.
- Where: this is our server-side stack trace to enable us to fix the error.

> This should only be returned in your development environment. Returning too much information in a production environment may compromise security.

With our server-side stack we've enabled exceptions by adding the line `ExtDirect::$debug = true`; to the PHP file.

By listening for the exception event within the `Ext.direct.Manager` class, we are able to output a basic alert and more detail to our console.

When the exception fag is switched off (`false`) the outcome is that our `result` object contains the string `"Exception"`. However, the specification is unclear on how to handle this in a production environment. This is because we are told not to return `"type": "exception"` in a production environment.

Our recommendation is for you to return the exception in a production environment but exclude the details of where as this is where your sensitive information is likely to be.

See also

The recipes in *Chapter 12* that discuss `Ext.Direct` from the beginning through to how it can be used with forms, stores, and grids.

Handling session timeouts with a TaskManager

When creating a secured web application that requires a user to login, it is likely that we will make their logged-in session expire after a given amount of time. This will be handled by our application's server-side code and so our client-side code will only find out about the session expiration after an AJAX request is made and a failure is returned.

Ideally we would like to have our client-side application stay in sync with our server's expiration status and preempt a timeout at the correct moment and display this to the user.

In this recipe we will describe how to create a Session Manager class that will manage this situation and ensure that your application's client-side code reflects a session timeout before another AJAX request is made.

How to do it...

The approach we will take to implement this feature is to track when the last action that refreshes the user's session occurs. We will assume a call to the server will reset the timeout countdown. We will then periodically check whether the difference between the last action's time and the current time is greater than the pre-set session timeout length.

Follow these steps to create our Session Manager class:

1. We will encapsulate our Session Manager code in a separate class in a new namespace named Cookbook. We start by creating a skeleton class declaration:

   ```
   Ext.define('Cookbook.SessionManager', {
   });
   ```

2. Our new class will be a singleton and extend the `Ext.util.Observable` class so we can have it fire events. We add these characteristics using the `singleton` and extend config properties respectively:

   ```
   Ext.define('Cookbook.SessionManager', {
       extend: 'Ext.util.Observable',
       singleton: true
   });
   ```

   > A singleton is a class that can only have one instance and can be accessed globally. In our example the `Cookbook.SessionManager` class will be accessible through this property anywhere in the application.

3. Next we give our class three `config` properties containing the time of the last activity, the number of seconds until a timeout happens, and our task definition:

   ```
   Ext.define('Cookbook.SessionManager', {
       extend: 'Ext.util.Observable',
       singleton: true,
       config: {
           lastActivityTime: new Date(),
           sessionTimeoutTime: 30,
           sessionTask: null
       }
   });
   ```

4. We now add our class's constructor and use it to add a `sessiontimeout` event and to configure our `sessionTask` property. We configure the `sessionTask` to run the `doSessionTimeoutCheck` method every second:

```
...
constructor: function(config){
    this.initConfig();
    this.addEvents('sessiontimeout');
    this.callParent([config]);
    this.setSessionTask({
        run: this.doSessionTimeoutCheck,
        scope: this,
        interval: 1000
    });
}
...
```

5. We require a few helper methods to start and stop our repeating task and a method to reset our `lastActivityTime` property, so we will now add these:

```
resetLastActivityTime: function(){
    this.setLastActivityTime(new Date());
},
startSessionTimer: function(){
    Ext.TaskManager.start(this.getSessionTask());
},
stopSessionTimer: function(){
    Ext.TaskManager.stop(this.getSessionTask());
}
```

6. We create a method that figures out if the `lastActivityTime` was longer than the `sessionTimeoutTime` and therefore the session will have timed out:

```
hasTimedOut: function(){
  var currentTime = new Date(),
      secondsDifference = new Date(currentTime.getTime() - this.
getLastActivityTime().getTime()).getTime() / 1000;
  return secondsDifference >= this.getSessionTimeoutTime();
}
```

7. The `doSessionTimeoutCheck` method that we referenced in the constructor can now be created. Its job is to determine if a timeout has occurred and raise the `sessiontimeout` event if appropriate.

```
doSessionTimeoutCheck: function(){
    if (this.hasTimedOut()) {
        this.stopSessionTimer();
        this.fireEvent('sessiontimeout', this);
    }
}
```

8. Finally, we add a method that will initialize the manager and start the timeout-checking task. This will be executed when the application starts and any time where a session timeout has occurred and the process must be restarted (that is, after a relogin):

```
initSession: function(){
    this.resetLastActivityTime();
    this.stopSessionTimer();
    this.startSessionTimer();
}
```

9. This gives us our complete class that can be seen as follows:

```
Ext.define('Cookbook.SessionManager', {
    extend: 'Ext.util.Observable',
    singleton: true,
    config: {
        lastActivityTime: new Date(),
        sessionTimeoutTime: 5,
        sessionTask: null
    },

   constructor: function(config){
       this.initConfig();

       this.addEvents('sessiontimeout');

       this.callParent([config]);

       this.setSessionTask({
            run: this.doSessionTimeoutCheck,
            scope: this,
            interval: 1000
        });
    },
    initSession: function(){
        this.resetLastActivityTime();
        this.stopSessionTimer();
        this.startSessionTimer();
    },
    doSessionTimeoutCheck: function(){
        if (this.hasTimedOut()) {
            this.stopSessionTimer();
            this.fireEvent('sessiontimeout', this);
        }
    },
    hasTimedOut: function(){
        var currentTime = new Date(),
            secondsDifference = (currentTime.getTime() - this.
getLastActivityTime().getTime()) / 1000;
        return secondsDifference >= this.getSessionTimeoutTime();
    },
    resetLastActivityTime: function(){
        this.setLastActivityTime(new Date());
    },
    startSessionTimer: function(){
        Ext.TaskManager.start(this.getSessionTask());
    },
    stopSessionTimer: function(){
        Ext.TaskManager.stop(this.getSessionTask());
    }
});
```

We have our `Cookbook.SessionManager` class, so now we can incorporate it into an application situation very easily. We are going to have our application display an alert when a timeout occurs and, when closed, our session will restart. We will also add a button that will make an AJAX request that, when successful, will reset the last activity time:

1. First we define a handler method for the `sessiontimeout` event and hook it up to the event. This simply shows an alert and when it is closed restarts the session:

```
var onSessionTimeout = function(sessionManager){
    alert('Your Session has timed out. Click OK to relogin.');
    sessionManager.initSession();
};
Cookbook.SessionManager.on('sessiontimeout', onSessionTimeout,
this);
```

2. Next we create a button that will make an AJAX call to a static JSON file that returns a simple `success` response. In its `success` function we display an alert and then reset the `lastActivityTime` property to the current time:

```
Ext.create('Ext.button.Button', {
    renderTo: Ext.getBody(),
    text: 'Do AJAX Call',
    handler: function(){
        Ext.Ajax.request({
            url: 'SessionTimeout.json',
            success: function(response, opts){
                alert('AJAX call made successfully, session has
not timed out.');
                Cookbook.SessionManager.resetLastActivityTime();
            }
```

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

3. Lastly, we initialize the Session Manager by calling its `initSession` method:

   ```
   Cookbook.SessionManager.initSession();
   ```

How it works...

The theory behind this class is quite simple and follows the ensuing logic:

- Configure the duration a session lasts for
- Record the time the last activity occurred
- Start a repeating task that checks whether this last activity time is longer than the session duration
- If it is then the session has timed out so we raise an event alerting the application to that fact
- If it isn't we carry on
- If any action occurs that would cause the session timeout to be restarted (for example, an AJAX call to our server-side application) we simply reset the time of the last activity to the current time

We use the `config` property of Ext JS's class system, which means that the getter and setter methods are created for each of the properties in the `config` object. You can see these being used throughout the class. In order to have these configuration items processed and the get and set methods created, we must call the `initConfig` method within the class's constructor.

After calling the `initConfig` method we use the `addEvents` method, provided by the `Ext.util.Observable` class that we extended, to add our custom `sessiontimeout` event to the class. By doing this we are now able to call the `fireEvent` method, passing it the event name and any number of parameters that will be passed through to any handling functions.

The final step in our constructor is to set the `sessionTask` property using its auto-created setter method. This property holds an object that defines a task that can be passed to the `Ext.TaskManager` class.

The task object can contain six properties, which are explained in the documentation. We have only provided three: a `run` property, which defines the function that will be executed after each time period; an `interval` property, which indicates the number of milliseconds between each execution; and a `scope` property, which defines the scope in which the `run` function is executed.

The `Ext.TaskManager` class allows us to define functions that are required to be executed repeatedly, after a given time period. It essentially boils down to using JavaScript's `setInterval` method but provides us with much more control over how the tasks are run.

The next important method in this class is the `hasTimedOut` method that returns a Boolean value determining if the session has timed out. We can break this calculation down into various steps. We start by using the `getTime` method of the `Date` object to get the number of milliseconds since 01/01/1970 for the `lastActivityDate` and the current date. We can then subtract these two numbers from each other to get the number of milliseconds between them, then dividing by 1000 to convert it into seconds. This number is then compared with the `sessionTimeoutTime` property and determines if the session has timed out or not.

The `hasTimedOut` method is invoked within the `doSessionTimeoutCheck` method, which is the method that our `Ext.TaskManager` task executes every second. This method calls the `hasTimedOut` method and, if it has timed out, stops the task (we don't want to keep checking if we know we have already timed out) and fires the `sessiontimeout` event that can then be handled by our application.

There's more...

In our previous example we incorporated an AJAX call that, when successful, would restart our session countdown because the session was refreshed on the server. This works well but requires us to manually add this logic to each of our AJAX requests, which leads to a lot of repetition.

We will now demonstrate how to override the `Ext.data.Connection` class to perform the session timeout detection logic in the background for any AJAX call made. In addition, we will also cover the situation of the server responding to an AJAX call telling us that the user's session has timed out, due to, for example, a login to the application from another computer:

1. We start by adding a new method to our `Cookbook.SessionManager` class called `isTimeoutResponse`. This method will accept a server-response object and return a boolean determining if the response is a session timeout. We will assume that the server will respond with a JSON response of `{"timeout": true}`:

```
isTimeoutResponse: function(response){
    var jsonResponse = Ext.decode(response.responseText);
    return Ext.isEmpty(jsonResponse) ? false : jsonResponse.
timeout;
}
```

This method simply decodes the `responseText` property of the specified response object and returns the value of the `timeout` property, which will be undefined (that is, `false`) in the majority of calls. In order to cater for situations such as request timeouts, where no `responseText` is present, we check that the `jsonResponse` value isn't empty and return `false` if it is.

2. Next we must write our override for the `onComplete` method of the `Ext.data.Connection` class. This method is executed following an AJAX call and determines what `callback` functions need to be executed.

   We will add our own logic to this that will check if the response was a timeout (using the method above) and, if it is, then call the Session Manager's `onSessionTimeout` method, which will raise the `sessiontimeout` event allowing the application to respond appropriately. Start by using the class's `override` method and copying the `onComplete` method into it from the framework itself. This will replace the framework's `onComplete` method with an identical copy so it won't change the behavior yet:

```
Ext.define('Cookbook.override.Connection', {
     override: 'Ext.data.Connection',
     onComplete : function(request) {
        var me = this,
            options = request.options,
            result,
            success,
            response;
        try {
            result = me.parseStatus(request.xhr.status);
        } catch (e) {
            // in some browsers we can't access the status if the 
readyState is not 4, so the request has failed
            result = {
                success : false,
                isException : false
            };
        }
        success = result.success;

        if (success) {
            response = me.createResponse(request);
            me.fireEvent('requestcomplete', me, response, 
options);
            Ext.callback(options.success, options.scope, 
[response, options]);
        } else {
            if (result.isException || request.aborted || request.
timedout) {
                response = me.createException(request);
            } else {
                response = me.createResponse(request);
            }
            me.fireEvent('requestexception', me, response,
options);
            Ext.callback(options.failure, options.scope,
[response, options]);
        }
        Ext.callback(options.callback, options.scope, [options,
success, response]);
        delete me.requests[request.id];
        return response;

    }
});
```

3. Now we can start introducing our custom logic. First we move the calls to the `createResponse` and `createException` methods outside the `if(success)` statement so a response is created earlier on. This change is required so we can pass our response object into our new `isSessionTimeout` method.

   Next we wrap the `if(success)` block in another if statement that will check the response to determine if it was a session timeout response. If it is, we execute the SessionManager's `onSessionTimeout` method. If it isn't then we proceed with the usual post-request logic but include a call to the `resetLastActivityTime` method so our session countdown starts again:

```
if(Cookbook.SessionManager.isTimeoutResponse(response)){
    Cookbook.SessionManager.onSessionTimeout();
} else {
    Cookbook.SessionManager.resetLastActivityTime();
    if (success) {
        me.fireEvent('requestcomplete', me, response, options);
        Ext.callback(options.success, options.scope, [response,
options]);
    } else {
        me.fireEvent('requestexception', me, response, options);
        Ext.callback(options.failure, options.scope, [response,
options]);
    }
    Ext.callback(options.callback, options.scope, [options,
success, response]);
}
delete me.requests[request.id];
return response;
```