SAMPLE CHAPTER

# EXT JS IN ACTION

Jesus Garcia

MANNING

*Ext JS in Action*
by Jesus Garcia

**Chapter 4**

# *brief contents*

# 4 A place for Components

***This chapter covers***

- Exploring the `Panel`
- Implementing the many `Panel` content areas
- Displaying an `Ext.Window`
- Using `Ext.MessageBox`
- Learning about and creating `TabPanels`

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

Here we'll cover `Containers`, which provide the management of child items and are one of the most important concepts in the Ext framework. We'll also dive into how the `Panel` works and explore the areas where it can display content and UI widgets. We'll then explore `Windows` and the `MessageBox`, which float above all other content on the page. Toward the end, we'll dive into using `TabPanels` and explore some of the usability issues that may occur when working with this widget.

Upon completion of this chapter, you'll have the ability to manage the full CRUD (create, read, update, and delete) lifecycle for Containers and their child items, which you'll depend on as you develop your applications.

## 4.1 The Panel

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

The Panel's title bar is a busy place that offers both visual and interactive content for the end user. As in Microsoft Windows, you can place an icon at the top left of the Panel, offering your users a visual queue as to what type of Panel they're seeing. In addition to the icon, you can display a title on the Panel.

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

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

Let's build a complex panel with top and bottom Toolbars, with two Buttons each.

### 4.1.1 Building a complex Panel

Because the Toolbar will have Buttons, you need a method to be called when they're clicked:

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

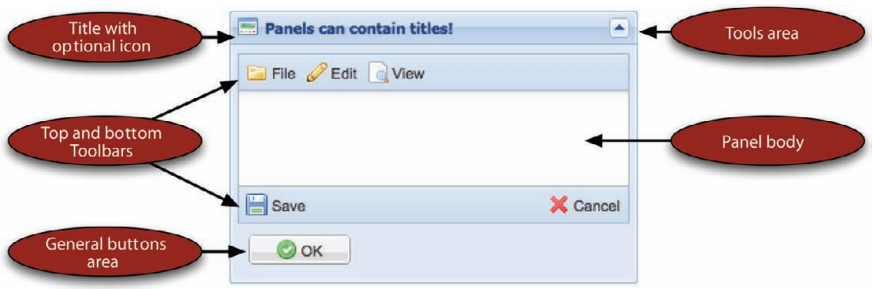


**Figure 4.1 An example of a fully loaded Panel, which has a title bar with an icon and tools, top and bottom Toolbars, and a button bar on the bottom**

This method will be called when a `Button` on any `Toolbar` is clicked. The `Toolbar` `Buttons` will call handlers, passing themselves as a reference, called `btn`. Next, let's define your toolbars; see the following listing.

**Listing 4.1 Building `Toolbars` for use in a `Panel`**

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

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

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

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

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

var myBottomToolbar = [
    {
        text    : 'Save',
        handler : myBtnHandler
    },
    '-',
    {
        text    : 'Cancel',
        handler : myBtnHandler
    },
    '->',
    '<b>Items open: 1</b>',
];
```

1 Button click handler method

2 File Button

3 Edit Button

4 "Greedy" Toolbar fill

5 Top Toolbar instantiation

6 Bottom Toolbar array configuration

In the preceding code example, you do quite a lot and display two different ways of defining a `Toolbar` and its child `Components`. First, you define `myBtnHandler` 1. By default, each `Button`'s handler is called with two arguments, the `Button` itself and the browser event wrapped in an `Ext.Event` object. You use the passed `Button` reference (`btn`) and pass that text to `Ext.MessageBox.alert` to provide the visual confirmation that a `Button` was clicked.

Next, you instantiate the File 2 and Edit 3 `Buttons` and the "greedy" toolbar spacer 4, which will push all toolbar items after it to the right. You assign `myTopToolbar` to a new instance of `Ext.Toolbar` 5, referencing the previously created `Buttons` and spacer as elements in the new toolbar's `items` array.

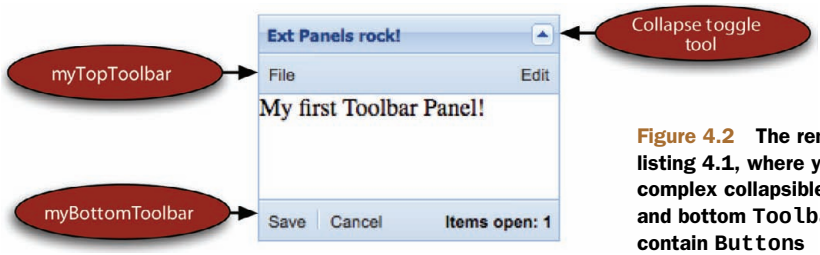


**Figure 4.2 The rendered results of listing 4.1, where you create a complex collapsible `Panel` with top and bottom `Toolbars` that each contain `Buttons`**

That was a lot of work for a relatively simple `Toolbar`. I had you do it this way to "feel the pain" of doing things the old way and to better appreciate how much time (and end developer code) the Ext shortcuts and XTypes save. The `myBottomToolbar` 6 reference is a simple array of objects and strings, which Ext translates into the appropriate objects when its parent container deems it necessary to do so. To get references to the top `Toolbar`, you can use `myPanel.getTopToolbar()` and, inversely, to get a reference to the bottom, `myPanel.getBottomToolbar()`. You'd use these two methods to add or remove items dynamically to or from either `Toolbar`. We'll cover `Toolbars` in much greater detail later. Next, you'll create your `Panel` body:

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

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

`Toolbars` are great places to put content, `Buttons`, or `Menus` that are outside the `Panel` body. There are two areas you still need to explore, `Buttons` and `tools`. To do this, you'll add to the `myPanel` example in the next listing, but you'll do it using the Ext shortcuts with XTypes inline with all of the other configuration options.

**Listing 4.2 Adding `Buttons` and `tools` to your existing `Panel`**

```
var myPanel = new Ext.Panel({
    ...
    buttons     : [
        {
```

1 **Properties from previous example**

2 **buttons array begins**

```
            text    : 'Press me!',                    ← 3 'Press me!' button
            handler : myBtnHandler
        }
    ],                                  ← 4 tools array begins
    tools       : [
        {                                          ← 5 'gear' tool and
            id      : 'gear',                        inline click handler
            handler : function(evt, toolEl, panel) {
                var toolClassNames = toolEl.dom.className.split(' ');
                var toolClass      = toolClassNames[1];
                var toolId         = toolClass.split('-')[2];

                Ext.MessageBox.alert('You Clicked', 'Tool ' + toolId);
            }
        },
        {                                          ← 6 'help' tool and
            id      : 'help',                        inline click handler
            handler : function() {
                Ext.MessageBox.alert('You Clicked', 'The help tool');
            }
        }
    ]
});
```

In listing 4.2, you added to the previous set of config options (1) and included two shortcut arrays, one for `buttons` and the other for `tools`. Because you specified a `buttons` array (2), when the `Panel` renders, it will create a footer `div`, a new instance of `Ext.Toolbar` with a special CSS class `x-panel-fbar`, and render it to the newly created footer `div`. The 'Press me!' `Button` (3) will be rendered in the newly created footer `Toolbar`, and when clicked it will invoke your previously defined `myBtnHandler` method.

If you look at the `myBottomToolbar` shortcut array in listing 4.1 and the `buttons` shortcut array in listing 4.2, you'll see some similarities. This is because all of the `Panel Toolbars` (`tbar`, `bbar`, and `buttons`) can be defined using the same shortcut syntax because they will all get translated into instances of `Ext.Toolbar` and rendered to their appropriate position in the `Panel`.

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

The preceding example is meant to display all of the items that can be utilized on a `Panel`, but it isn't the best example of elegant and efficient user interface design. Although it may be tempting to load up your `Panels` with `Buttons` and `Toolbars`, you must be careful not to overload a `Panel` with too much onscreen gadgetry, which could overwhelm your user and take up valuable screen real estate.

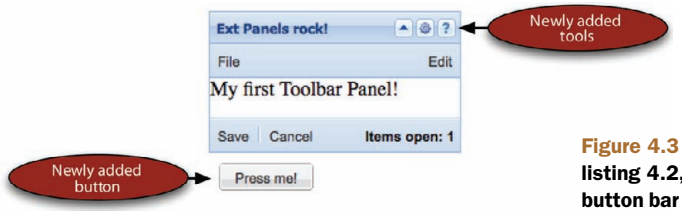


**Figure 4.3 The rendered results from listing 4.2, which add a `Button` in the button bar as well as `tools` to the title bar**

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

## 4.2 Popping up Windows

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

Working with the `Window` class is a lot like working with the `Panel` class, except you have to consider issues like whether you want to disable resizing or want the `Window` to be constrained within the boundaries of the browser's viewport. Let's look into how you can build a window. For this, you'll need a vanilla Ext `Page` with no widgets loaded, as shown in the following listing.

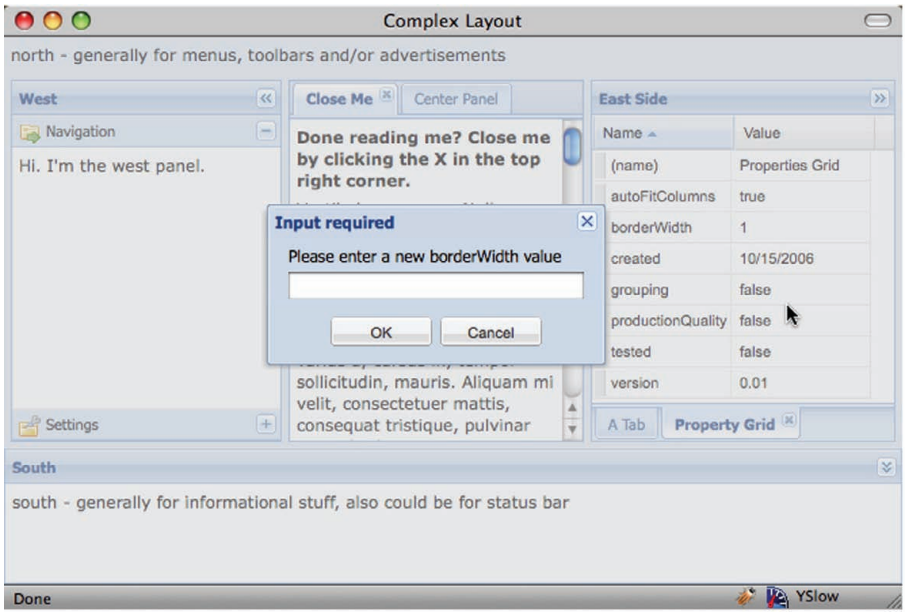


**Figure 4.4 An Ext modal `Window`, which masks the browser's viewport**

**Listing 4.3 Building an animated Window**

```
var win;
var newWindow = function(btn) {
    if (!win) {
        win = new Ext.Window({
            animateTarget : btn.el,
            html          : 'My first vanilla Window',
            closeAction   : 'hide',
            id            : 'myWin',
            height        : 200,
            width         : 300,
            constrain     : true
        });
    }
    win.show();
}
new Ext.Button({
    renderTo : Ext.getBody(),
    text     : 'Open my Window',
    style    : 'margin: 100px',
    handler  : newWindow
});
```

1 Handler creates new Window

2 Instantiate new Window

3 Prevent destruction on close

4 Constrain Window to browser's viewport

5 Create Button that launches Window

In listing 4.3, you do things a little differently in order to see the animation for your `Window`'s `close` and `hide` method calls. The first thing you do is create a global variable, `win`, for which you'll reference the soon-to-be-created `Window`. You create a method, `newWindow`, 1 that will be the handler for your future `Button` and is responsible for creating the new `Window` 2.

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

Last, you create a `Button` 5 that, when clicked, will call your `newWindow` method, resulting in the window animating from the `Button`'s element. Clicking the (x) close tool will result in the window hiding. The rendered results will look like figure 4.5.

Because you don't destroy the `Window` when the close tool is clicked, you can show and hide the `Window` as many times as you wish, which is ideal for `Windows` that you plan to reuse. Whenever you deem that it's necessary to destroy the `Window`, you can call its `destroy` or `close` method. Now that you have experience in creating a reusable

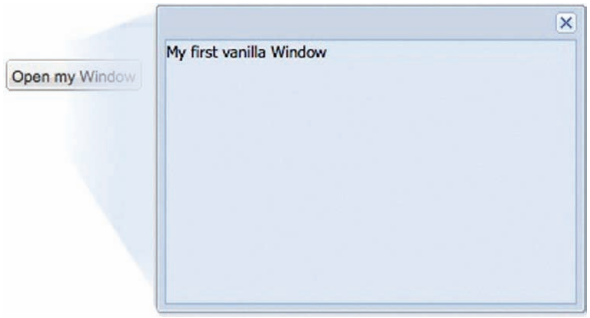


**Figure 4.5** The rendered results from listing 4.3, where you create a `Window` that animates from the `Button`'s element when clicked

`Window`, you can begin exploring other configuration options to further alter the behavior of the `Window`.

### 4.2.1 Further Window configuration exploration

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

Sometimes you need to produce a `Window` that's modal and rigid. To do this, you need to set a few configuration options, as shown in the next listing.

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

```
var win = new Ext.Window({
     height      : 75,
     width       : 200,
     modal       : true,
     title       : 'This is one rigid window',
     html        : 'Try to move or resize me. I dare you.',
     plain       : true,
     border      : false,
     resizable   : false,
     draggable   : false,
     closable    : false,
     buttonAlign : 'center',
     buttons     : [
         {
             text    : 'I give up!',
             handler : function() {
                 win.close();
             }
         }
     ]
 })
 win.show();
```

1 Ensure page is masked

2 Prevent resizing

3 Disable Window movement

4 Prevent Window closure

In listing 4.4, you create an extremely strict modal `Window`. To do this, you have to set quite a few options. The first of these, `modal` 1, instructs the `Window` to mask the rest of the page with a semitransparent `div`. Next, you set `resizable` 2 to `false`, which

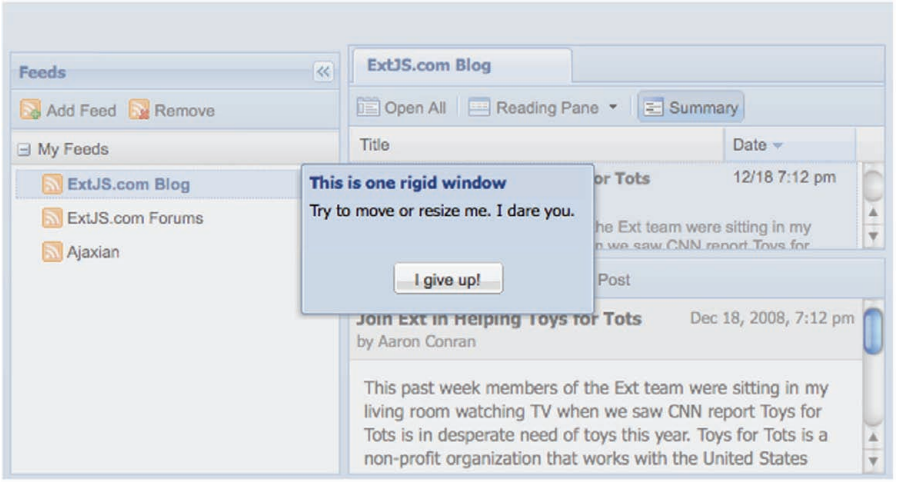


**Figure 4.6 Our first strict modal `Window` rendered in the Ext SDK feed viewer example**

prevents the `Window` from being resized via mouse actions. To prevent the `Window` from being moved around the page, you set `draggable` (3) to `false`. You want only a single center `Button` to close the `Window`, so `closable` (4) is set to `false`, which hides the close tool. Last, you set some cosmetic parameters, `plain`, `border`, and `buttonAlign`. Setting `plain` to `true` will make the content body background transparent. When coupled with setting the `border` to `false`, the `Window` appears to be one unified cell. Because you want to have the single `Button` centered, you specify the `buttonAlign` property as such. The rendered example should look like figure 4.6.

Other times you want to relax the restrictions on the `Window`. For instance, there are situations where you need a `Window` to be `resizable`, but not less than specific dimensions. For this, you allow resize (`resizable`) and specify `minWidth` and `minHeight` parameters. Unfortunately, there's no easy way to set boundaries as to how large a `Window` can grow.

Although there are many reasons for creating your own `Windows`, there are times when you need something quick and dirty, for instance, to display a message or prompt for user data. The `Window` class has a stepchild known as the `MessageBox` to fill this need.

### 4.2.2 Replacing alert and prompt with MessageBox

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

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

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

Your `myCallback` method will leverage Firebug's console to echo out the `Button` that was pressed and the text you entered, if any. The `MessageBox` will pass only two parameters to the callback method: the `Button` ID and any entered text. Now that you have your callback, let's launch an alert message box:

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

Here, you call the `MessageBox.alert` method, which will generate a `Window`, which will look like figure 4.7 (left) and will dismiss when OK is clicked. If you want `myCallback` to get executed upon dismissal, add it as the third parameter. Now that we've looked at alerts, let's see how you can request user input with the `MessageBox.prompt` method:

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

You call the `MessageBox.prompt` method, which you pass the reference of your callback method; it will look like figure 4.7 (right). Enter some text and click Cancel. In the Firebug console, you'll see the `Button` ID pressed and the text entered.

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

I have to confess a little secret. The alert and prompt methods are actually shortcut methods for the much larger and highly configurable `MessageBox.show` method. Next up is an example of how you can use the `show` method to display an icon with a multiline `TextArea` input box.

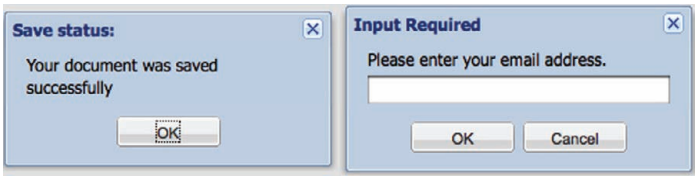


**Figure 4.7 The `MessageBox`'s alert (left) and prompt (right) modal dialog `Windows`**

### 4.2.3 *Advanced MessageBox techniques*

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

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

When the preceding example is rendered, it will display a modal dialog box like the one in figure 4.8 (left). Next, let's see how to create an alert box that contains an icon and three `Buttons`.

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

The preceding code example will display your tri-button modal `alert` dialog `Window`, like the one in figure 4.8 (right).

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

The `buttons` parameter is used as a guide for the singleton to know which `Buttons` to display. Although you pass a reference to an existing property, `Ext.MessageBox.OKCANCEL`, you can display no `Buttons` by setting `buttons` to an empty object, such as `{}`. Otherwise, you can customize which `Buttons` you want to display. To display Yes and Cancel `Buttons`, pass `{ yes : true, cancel : true}` and so on. The singleton

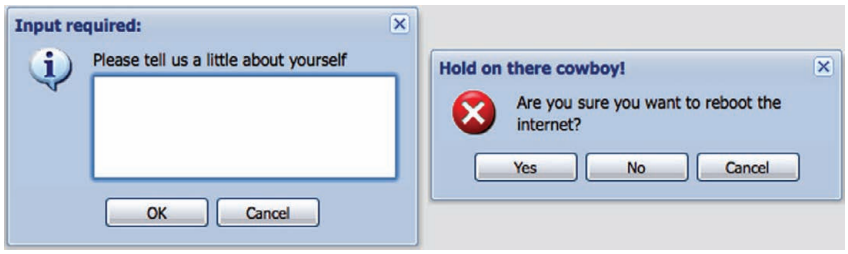


**Figure 4.8** **A multiline input box with an icon (left) and a tri-button icon alert box (right)**

already has a set of predefined popular combinations, which are CANCEL, OK, OKCANCEL, YESNO, and YESNOCANCEL.

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

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

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

### 4.2.4 Showing an animated wait MessageBox

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

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

This will produce a wait box like the one shown in figure 4.9. If the syntax seems a little strange, it's because the first parameter is the message body text, with the second parameter being the title. It's exactly opposite of the alert or prompt calls. Let's say you want to display text in the body of the animating progress bar itself. You could pass a third parameter with a single text property, such as {te xt: 'loading your items'}. Figure 4.9 (right) also shows what it would be like if you added progress bar text to your dummy wait dialog box.

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

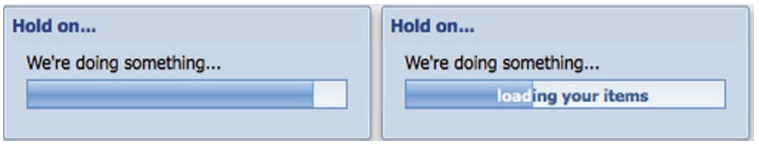


**Figure 4.9 A simple animated MessageBox wait dialog where the ProgressBar is looping infinitely at a predetermined fixed interval (left) and a similar message box with text in the progress bar (right)**

**Listing 4.5 Building a dynamically updating `ProgressBar`**

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

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

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

1 Show ProgressBar

2 Update progress text

3 Update percentage and text

4 Defer MessageBox dismissal

5 Update progress if limit not met

6 Looping .5 second timeout

In listing 4.5, you show a `MessageBox`, with the `progress` option 1 set to `true`, which will show your progress bar. Next, you define a rather involved updater function, aptly named `updateFn` 2, which is called at a predefined interval. In that function, if the number passed equals your limit of 6, you update the progress bar to 100 percent wide and show the completion text 3. You also defer the dismissal of the message box by one and a half seconds 4. Otherwise, you'll calculate a percentage completed and update the progress bar width and text accordingly 5. Last, you create a loop that calls `setTimeout` 6 six consecutive times, which delays your calls of `updateFn` by the iteration times one-half second. The results of this rather lengthy example will look like figure 4.10.

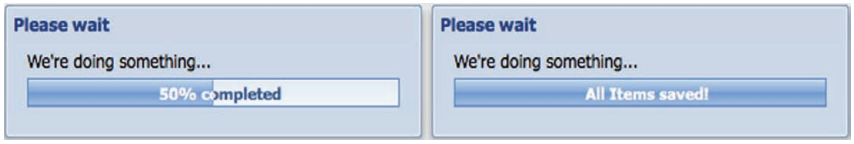


**Figure 4.10 Your automatically updating wait `MessageBox` (left) with the final update (right) before automatic dismissal**

With some effort you can dynamically update your users with a status of operations that are taking place before they can move further.

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

## 4.3 Components can live in tab panels too

The `Ext.TabPanel` class builds on `Panel` to create a robust tabbed interface, which gives the user the ability to select any screen or UI control associated with a particular tab. Tabs within the `TabPanel` can be unclosable, closable, disabled, and even hidden, as illustrated in figure 4.11.

Unlike other tab interfaces, the Ext `TabPanel` supports only a top or bottom tab strip configuration. This is mainly because most modern browsers don't support CSS version 4.0, where vertical text is possible. Although configuring a `TabPanel` may seem straightforward by looking at the API, there are two options that if not understood could cause bleeping to occur in the office.

### 4.3.1 Remember these two options

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

One of the reasons that the `CardLayout` is so darn fast is because it makes use of a common technique called lazy or deferred rendering for its child `Components`. This is controlled by the `deferredRender` parameter, which is set to `true` by default. *Deferred render* means that only cards that get activated are rendered. It's fairly common for `TabPanels` to have multiple children that have complex UI controls, such as the one in figure 4.12, which can require a significant amount of CPU time to render. Deferring

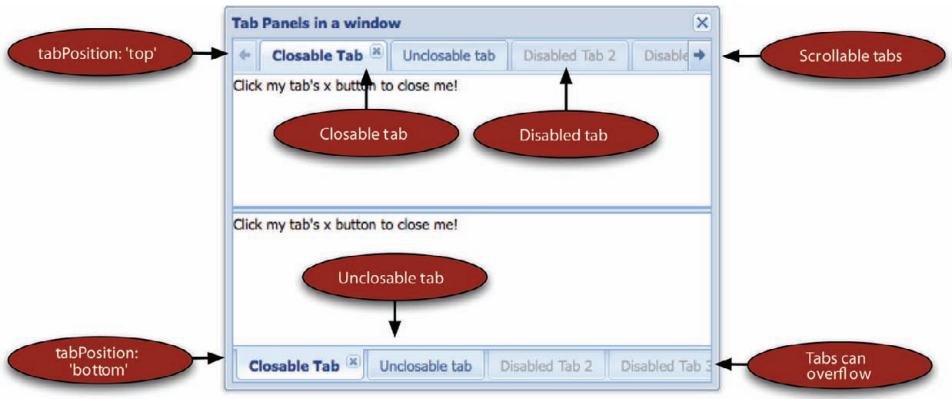


**Figure 4.11 Exploring top- and bottom-positioned tabs**

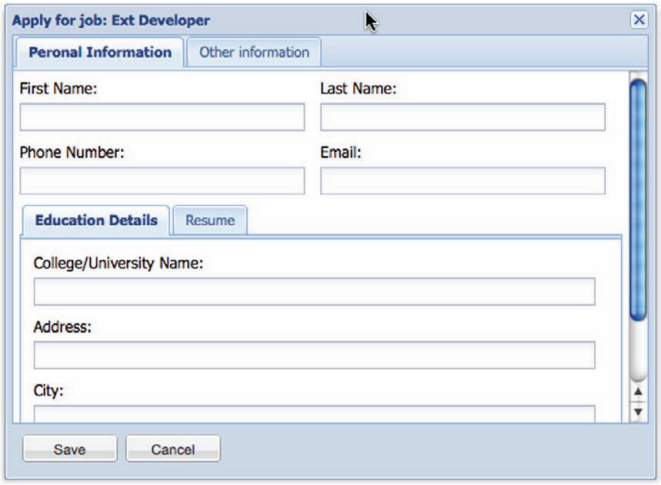


**Figure 4.12 A `TabPanel` with children that have complex layouts**

the render of each child until it's activated accelerates the `TabPanel`'s initial rendering and gives the user a better-responding widget.

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

Another configuration option that fellow developers often overlook is `layoutOnTabChange`, which forces a `doLayout` method call on child items when that tab is activated. This is important because in deeply nested layouts, sometimes the parent's resize event may not cascade down properly, forcing a recalculation on child items that are supposed to conform to the parent's content body. If your UI starts to look funky, like the one in figure 4.13, I suggest setting this configuration option to `true`, and the issue will be solved.

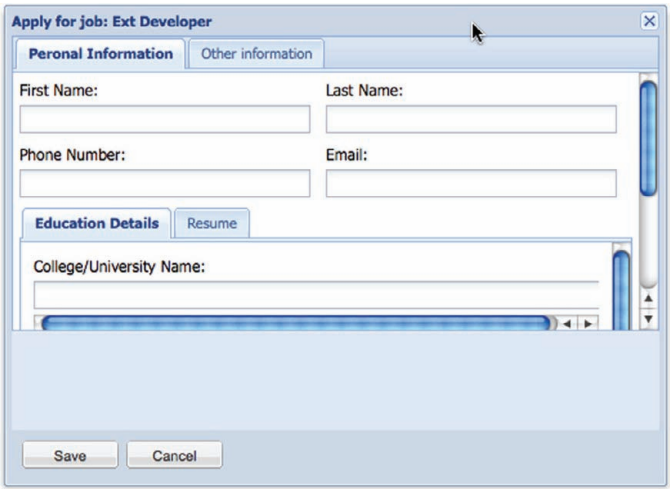


**Figure 4.13 A child `Panel` whose layout has not been properly recalculated after a parent's resize**

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

### 4.3.2 Building our first TabPanel

The `TabPanel` is a direct descendant of `Panel` and makes clever use of the `CardLayout`. `TabPanel`'s main job is managing tabs in the tab strip. This is because child management is performed by the `Container` class and the layout management is performed by the `CardLayout`. Let's build out our first `TabPanel`, as shown in the following listing.

**Listing 4.6 Exploring a TabPanel**

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

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

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

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

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

1 A simple, static tab

2 A simple, closable tab

3 A closable yet disabled tab

4 Our TabPanel

5 Container for our TabPanel

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

You then go on to instantiate your `TabPanel` (3). You set the `activeTab` parameter to `0`. You do this because you want the first tab to be activated after the `TabPanel` (4) is rendered. You can specify any index number in the `TabPanel`'s `item`'s mixed collection. Because the mixed collection is an array, the first item always starts with `0`. You also set `enableTabScroll` to `true`, which instructs the `TabPanel` class to scroll your tab strip *if* the sum of the tab widths exceeds that of the viewable tab strip. Last, your `TabPanel`'s `items` array has your three tabs specified.

Next, you create a `Container` for your `TabPanel`, an instance of `Ext.Window` (5). You specify a Fit layout for the `Window` and set the `tabPanel` reference as its single item. The rendered code should look should look like the `TabPanel` shown in figure 4.14.

Now that you have your first `TabPanel` rendered, you can start to have fun with it. You've probably closed the "I am closable" tab, which is okay. If you haven't done so, feel free to explore the rendered UI control and close out the only closable tab when you're comfortable doing so, which will leave only two tabs available, "My first tab" and "Disabled tab."

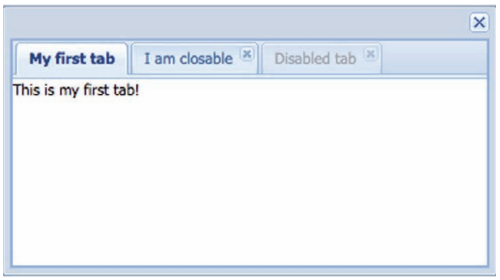


**Figure 4.14 Your first `TabPanel` rendered inside a `Window`**

### 4.3.3 Tab management methods you should know

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

The first of these is `setActiveTab`, which activates a tab, as if the user had selected the item on the tab strip, and accepts either the index of the tab or the `Component` ID:

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

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

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

Executing the prior code will result in a new tab with the title of "New closable tab," which gets activated automatically. Calling `setActiveTab` after an add operation is akin to calling `doLayout` on a generic container. You also have the capability to enable and disable tabs at runtime, but this requires a different approach than simply calling a method on the `TabPanel`.

The `TabPanel` doesn't have enable or disable methods, so in order to enable or disable a child, you need to call those methods of the child items themselves. You can leverage listing 4.6 to enable our disabled tab.

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

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

In addition to enabling and disabling tabs, you can also hide them. To hide a tab, however, the `TabPanel` does have a utility method, `hideTabStripItem`. This method accepts a single parameter but three possible data values: the tab index number, the tab `Component` ID, or a reference to the `Component` instance itself. In your case, you'll use the ID because that's a known:

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

The inverse of this is `unhideTabStripItem`:

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

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

### 4.3.4 Caveats and drawbacks

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

To offer some relief of these shortcomings, the `TabPanel` can be configured to resize tabs automatically or even scroll them if they go beyond the viewport. Although these features help ease the problem, they don't solve them.

In order for you to fully understand these issues, we should explore them further. Let's start out with a `TabPanel` inside a `Viewport`, shown in the following listing.

**Listing 4.7 Exploring scrollable tabs**

```
Ext.QuickTips.init();

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

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

1 **Ext Viewport with embedded TabPanel**

2 **Deferred anonymous function execution**

In listing 4.7, you create a viewport with your `TabPanel` 1, which contains a single child. Next, you create an anonymous function 2 and `defer` its execution by half a second. You specify 30 as the number of dynamic tabs to create dynamically in the `for` loop. For each new tab that you create, you include the tab number in the tab `title`, `html`, and `tabTip`. The rendered code should look like the tab panel in figure 4.15.

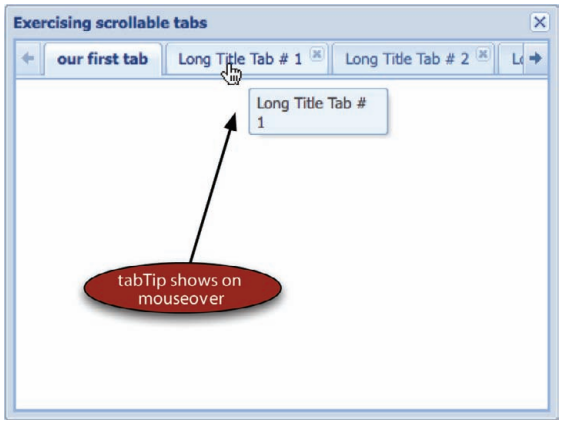


**Figure 4.15 A `TabPanel` with a scrolling tab strip, which includes `mouseover` tooltips for the dynamic tabs**

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

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

Refreshing the newly modified `TabPanel` in figure 4.16 (bottom) results in tabs that are either unusable or hard to use. Specifying `resizeTabs` as `true` instructs the `TabPanel` to reduce the width of a tab as much as it needs to in order to display the tabs without scrolling. Autosizing a tab works if the tab title doesn't get truncated or hidden. This is where the diminishing usability of `TabPanels` becomes apparent. If the tab title isn't completely visible, the user must activate each tab in order to find the correct one. Otherwise, if the tab tooltips are enabled, the user must mouseover each tab in order to locate the one they wish to activate. As you can see in figure 4.16, the tooltips can enhance the speed of the tab search but don't remedy the issue completely.

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

In exploring the `TabPanel`, you learned how to create tabs that could be static, controlled, disabled, or even hidden and to programmatically control them. You also learned of two of the configuration options, `deferredRender` and `layoutOnTabChange`, that could cause some of your hair to fall out. We exercised some of the common tab-management methods and discussed some of the caveats for using this UI control.

There you have it. You have now seen how easy it is to create a scrolling `TabPanel`. Always remember to enabling scrolling when the number of tabs can increase beyond the width of the `TabPanel`.

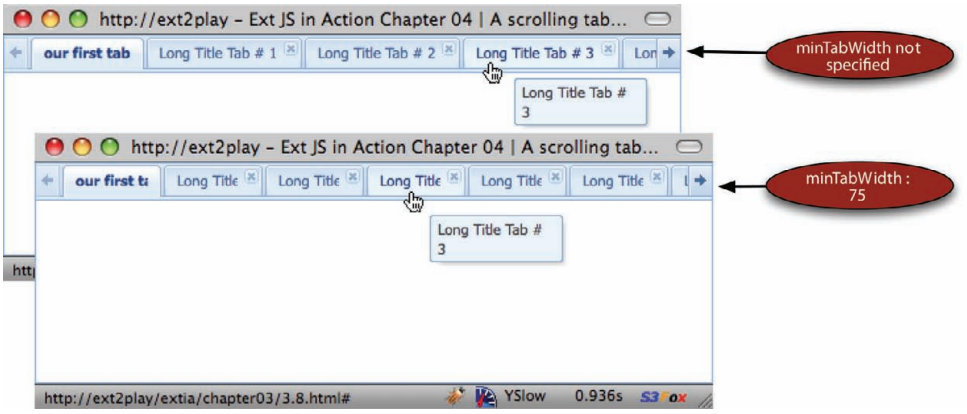


**Figure 4.16 A `TabPanel` that has no minimum tab width (top) specified and a `TabPanel` (bottom) that has a minimum tab width of 75 specified**

## 4.4 Summary

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

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

In exercising the `Window` class and its cousin, the `MessageBox`, you learned how you could replace the generic alert and prompt dialog boxes to get the user's attention to display or request user input. You also had some fun fooling with the animated wait `MessageBox`.

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

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

JAVASCRIPT/WEB DEVELOPMENT

# EXT JS IN ACTION

Jesus Garcia


This cross-browser JavaScript library provides an extensive collection of high-quality widgets, an intuitive and extensible component model, and a rich API that enterprise developers find especially comfortable to use. And they have used it to build rock-solid web applications, in many very different companies including Adobe, Aetna, Amazon, Best Buy, Hallmark, Panasonic, Pixar, Siemens, Sony, and Visa.

**Ext JS in Action** is a comprehensive guide to Ext JS. By following its rich examples, patterns, and best practices, you'll achieve the kinds of results you only see in top JavaScript applications. This book thoroughly explores every class, component, and model, and shows you how to build rich, dynamic user interfaces and responsive applications. You will learn Ext JS inside and out—and your apps will stand out from the crowd.

## What's Inside

- Explore the depths of Ext JS 3.0
- Create rich and dynamic UIs
- Extend the framework and write plug-ins
- Watch the author develop an Ext JS app

This book assumes a reader with a foundation in JavaScript, but no previous exposure to Ext JS.

**Jesus "Jay" Garcia** is an Ext JS community leader. Since 2006, he has deployed and optimized Ext JS world-class applications for many corporations.

For online access to the author and a free ebook for owners of this book, go to manning.com/ExtJSinAction

"Outstanding resource for using Ext JS!"
—Dan McKinnon
MITRE Corporation

"An easy-to-understand walk-through of Ext JS 3."
—Mitchell Simoens
Senior Web Developer

"Takes the complexity out of a complex interface."
—Ric Peller
Management Dynamics

"Really EXTends your knowledge!"
—Jeroen Benckhuijsen
Atos Origin

"This is a handy book!"
—Orhan Alkan, Oracle

ISBN 13: 978-1-935182-11-5
ISBN 10: 1-935182-11-0
54999
9 781935 182115

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]