CHAPTER 4

Controls and Layout

Ext JS 4 provides a suite of UI controls that are used in applications. The UI controls include simple form controls like text boxes, buttons, layout containers like accordion, table, and so on. In this chapter we'll discuss the UI controls and layout containers.

Just like OO UI frameworks such as Swing for Java and WinForms for .NET, EXT JS 4 provides a well-defined hierarchy of UI components starting with a base class that establishes a common set of properties and functionalities to all the components. Let's start exploring the UI components in Ext JS 4 by looking at the Ext.Component class.

Ext.Component

The `Ext.Component` class serves as the base class for all the UI components in Ext JS 4. Ext.Component inherits the `Ext.AbstractComponent` class. It provides the common behavior and properties for all the UI components. The common functions include the basic creation, destruction, and rendering of the components. You can instantiate this class as shown below, though you'll use it very rarely in the raw format.

```
Ext.create("Ext.Component", {
       html: "Raw Component",
       renderTo : Ext.getBody()
});
```

The above code displays a text *Raw Component* in the page. It generates the following HTML snippet.

```
<div id="component-1099 class="x-component x-component-default">Raw Component</div>
```

The Ext.Component generates a `<div>` tag with an automatically generated id and a default CSS class. You'll learn about the CSS classes in *Theming and Styling* chapter.

All the components used in a page can be accessed through a singleton object `Ext.ComponentManager`. Ext. ComponentManager serves as a registry of all the components. You can access all the components by using the `all` property as `Ext.ComponentManager.all.`

You can access the individual components based on their id by using the get method as `Ext.ComponentManager.get("id of the component")`. We'll discuss more about the id attribute later in this section.

It's important to understand the common configuration attributes and methods of the Ext.Component class before you start working with the UI controls that are just derived classes of Ext.Component.

Configuration attributes of Ext.Component

Ext.Component class provides a large number of configuration attributes. You can get the complete list of the attributes from http://docs.sencha.com/extjs/4.2.0/#!/api/Ext.Component.

Let's discuss some of the configuration attributes of the Ext.Component class. These attributes are available to all the UI controls that you'll use.

id

Every component has an automatically generated unique id assigned to it. You can use *Ext.getCmp()* method to access the component by specifying the id. You can assign your own id for the component as well.

```
Ext.create("Ext.Component",{
 id : "mycomp1"
});
```

You can use Ext.getCmp as shown below

```
Ext.getCmp("mycomp1");
```

It's generally not recommended to define your own id, because as the application grows and you start adding components dynamically it may lead to duplication issues.

itemId

You can mark the component with an itemId instead of an id. The component that has itemId assigned to it can be accessed using that itemId through its parent component. Say, you have a Panel that has a component with an itemId. You can access the component using the itemId by invoking the method getComponent() on the Panel.

```
var panel1 = Ext.create("Ext.panel.Panel",{
 // ...
 items : [
 Ext.create("Ext.Component",{
 html : "Raw Component inside panel",
 itemId : "rawcomp1"
})
]
// ...
      });

panel1.getComponent("rawcomp1")
```

The itemId property is preferred to the id as you don't have to worry about the complications that arise due to duplicate id.

autoEl

The autoEl attribute is used to specify a custom HTML element that will encapsulate the the component. This attribute is usually used when we create custom components. You'll learn this in detail in Chapter 10. Here's a component that generates a hyperlink element using autoEl attribute.

```
Ext.create("Ext.Component", {
       renderTo: Ext.getBody(),
       autoEl: {
          html: "Link",
          href: "#",
          tag: "a"
       }
});
```

The code snippet will generate the following HTML snippet.

```
<a class="x-component x-component-default" href="#" id="component-1009">Link</a>
```

listeners

The listeners attribute is used to wire up the events with their handlers. listeners is a simple object that contains the list of all events along with their event handler functions.

```
listeners : {
         eventname1 : function(...) { ... },
         eventname2 : function(...){ ... },
   ...
}
```

renderTo

The component is rendered to the specified HTML element. If a component is added to a container, it's then the container's job to render the components. You can use renderTo as shown below.

```
renderTo : Ext.getBody() //renders to the body element
renderTo : Ext.get("content") //renders to the HTML element with the id "content"
```

hidden, disabled

The hidden and disabled attributes are used to specify visibility and whether the component is disabled or not, respectively. The default value for these attributes is false.

tpl, data

The components have a tpl property that's used to configure the UI template for the Component. The data attribute supplies data to be applied to the template. You'll learn about Templates in detail later in this chapter.

Let's discuss some of the methods in Component class.

Methods in Ext.Component

up

This method is used to navigate to the ancestors of the component that matches the expression passed as an argument. For example, if you have a component, say *cmp1*, calling **cmp1.up("panel")** walks up the container and returns the panel component that's a parent or grandparent or any ancestor. This method returns the immediate parent if the argument is ignored.

enable, disable

These are two commonly used methods that are used to enable and disable the components. Here's how you can use them on a component say comp1.

```
comp1.enable()
comp1.disable()
```

show, hide

These are two commonly used methods that are used to show and hide the components. Here's how you can use them on a component say comp1.

```
comp1.hide()
comp1.show()
```

destroy

The destroy method destroys the component. It removes the reference to the element in the DOM tree.

on, un

The listeners attribute is used to statically register the events and handler functions. The *on* method is used to dynamically do that. The on method accepts the name of the event, the event handler function and the scope or context of the executing handler function as arguments.

```
comp1.on("eventName",function(){...},scope)

mycombobox1.on("change",function(){...},this)
```

In the code snippet above, you've registered the change event on the combobox object. The scope 'this' refers to the context object where the handler function gets executed. The scope is an optional parameter.

The un method is used to remove the event handler for the specified event.

```
comp1.un("eventName",function(){...},scope)
```

You have to specify the same event handler function and scope used in the on method.

addEvents, fireEvent

The Component class provides methods addEvents and fireEvent for adding events and firing the event respectively. These two methods are mainly used when you create custom components with custom events. You can call addEvents on a component comp by writing `comp.addEvents('eventname1','eventname2' ...)`. You can invoke the fireEvent like `comp.fireEvent('eventname')`.

You'll learn more about these functions in Chapter 10 when we discuss creating custom components.

Now let's discuss the events in the Component class.

Events in Ext.Component

The Component class provides a number of lifecycle events. These are raised when the component is created, activated, rendered, destroyed, and so on. All these events can be handled by registering them using the listeners attribute or using the on method. Most of these lifecycle events are actually defined in the Ext.AbstractComponent class. Table 4-1 describes some of the events.

Table 4-1. *Events in a Component*

Event	Description
Added	Raised when the component is added to the container
Removed	Raised when the component is removed from the container
beforerender	Raised before rendering the component on to the HTML element
render	Raised after the component is rendered to the HTML element
afterrender	Raised after completion of the component rendering
beforedestroy	Raised before calling destroying the component or before calling destroy method
destroy	Raised after destroy or after calling the destroy method
beforeactivate	Raised before a component is activated. This is mainly used in accordions and tab panels.
activate	Raised after a component is activated
beforedeactivate	Raised before a component is deactivated
deactivate	Raised after a component is deactivated
beforeshow	Raised before calling the show method on the component
show	Raised after calling the show method on the component
beforehide	Raised before calling the hide method on the component
hide	Raised after calling the hide method on the component

Listing 4-1 shows the code snippet where you create a component and add to a Panel. The component has some of these events handled.

Listing 4-1. Events in Component

```
var pnl = Ext.create("Ext.panel.Panel", {
         items: [
             Ext.create("Ext.Component", {
                 html: "Raw Component",
                 itemId : "rawcomp1",
                 listeners: {
                     activate: function () {
                         console.log("activate")
                     },
                     added: function () {
                         console.log("added")
                     },
                     afterrender: function () {
                                console.log("afterrender")
                     },
                     beforeactivate: function () {
                                console.log("beforeactivate")
                     },
                     beforedeactivate: function () {
                                console.log("beforedeactivate")
                     },
                     beforerender: function () {
                                console.log("beforerender")
                     },
                     beforeshow: function () {
                                console.log("beforeshow")
                     },
                     beforedestroy: function () {
                                console.log("beforedestroy")
                     },
                     destroy: function () {
                                console.log("destroy")
                            },
                     render: function () {
                                console.log("render")
                            },

                    show: function () {
                                console.log("show")
                    },
                    beforehide: function () {
                                console.log("beforehide")
                    },
                    hide: function () {
                                console.log("hide")
                   },
                    enable: function () {
                                console.log("enable")
                    },
```

```
                disable: function () {
                            console.log("disable")
                },
                removed: function () {
                            console.log("removed")
               }
         }
      })
   ],
 renderTo: Ext.getBody()
        });

        console.log("******Calling disable")
        pnl.getComponent("rawcomp1").disable();
        console.log("******Calling enable")
        pnl.getComponent("rawcomp1").enable();
        console.log("******Calling hide")
        pnl.getComponent("rawcomp1").hide();
        console.log("******Calling show")
        pnl.getComponent("rawcomp1").show();
        console.log("******Calling destroy")
        pnl.getComponent("rawcomp1").destroy();
```

In Listing 4-1 we've registered the events using the listeners block. After rendering the component, we call methods like enable, disable, show, hide, or destroy to understand the event handling sequence. Here's the output of this code.

```
added
beforerender
render
afterrender
******Calling disable
disable
******Calling enable
enable
******Calling hide
beforehide
hide
******Calling show
beforeshow
show
******Calling destroy
beforedestroy
removed
```

destroy Another important aspect of the components in Ext JS 4 is *xtype*. Let's take a look at xtype in detail.

xtype

In Ext JS 4 every UI component class has an alias name or a short name known as '*xtype.*' Using xtype in our code offers some advantages. Let's discuss them by creating a Panel with a textbox and a button as shown below.

```
Ext.create("Ext.panel.Panel",{
  items : [
 Ext.create("Ext.form.field.Text",{
   fieldLabel : "Name"
 }),
 Ext.create("Ext.Button",{
   text : "Submit"
 })
 ]
});
```

Let's move the creation of textbox and button out of the Panel as shown below.

```
var nameText = Ext.create("Ext.form.field.Text",{
     fieldLabel : "Name"
});
var submitButton = Ext.create("Ext.Button",{
     text : "Submit"
});
```

The Panel will refer to the nameText and submitButton variables in the items collection.

```
Ext.create("Ext.Panel",{
 items : [
  nameText,submitButton
 ]
});
```

We've stored the textbox and button objects in separate variables and reused them inside the Panel. There are some disadvantages to writing code in this fashion, although it is useful to segregate the container and the individual components.

`Ext.create("Ext.form.field.Text")` creates a text box and holds it in the DOM tree. It occupies memory even if we don't render it on to the screen. Suppose we don't add the nameText variable in the Panel, it would remain in the DOM tree occupying memory. In an application, we want to instantiate UI components only when required and not create them at will. At the same time we want the component creation code maintained separately.

Using the fully qualified class name like *Ext.form.field.Text* everywhere is a tedious task, particularly when we create custom components. It would be better if we can use the xtype of these UI components. Let's rewrite the example as shown below.

```
var nameText = {
   xtype : "textfield",
   fieldLabel : "Name"
};
var submitButton = {
   xtype : "button",
   text : "Submit"
};
```

```
Ext.create("Ext.panel.Panel",{
   renderTo : Ext.getBody(),
   items : [
     nameText,submitButton
   ]
});
```

The nameText and submitButton are plain JavaScript objects. They have an additional xtype property with values textfield and button, respectively. The actual text box and button objects are created when they get added to the Panel and rendered to the body. This not only makes the code simpler but also provides us the lazy instantiation facility, thereby improving the performance.

As we discussed earlier, Ext.Component is inherited by a number of classes. Table 4-2 shows the list of subclasses of Ext.Component.

Table 4-2. *Subclasses of Ext.Component*

Class	Description
Ext.container.AbstractContainer	Base class for the container controls
Ext.button.Button	The button control
Ext.form.Label	The standard label element
Ext.form.field.Base	Base class for all the field components like textfield
Ext.draw.Component	Represents the surface on which you can draw shapes

One of the important subclasses of Ext.Component is Ext.container.AbstractContainer. This class is inherited by Ext.container.Container that serves as the base class for all the container clases like the Panel. Let's discuss the Ext.container.Container class in detail.

Ext.container.Container

Ext.container.Container class is the base class for all the container-based components in Ext JS 4. It provides the common behavior and properties for all the UI containers. The common functions include the addition, udpation, and removal of the components. You can instantiate this class as shown below, though you'll use it very rarely in the raw format.

```
Ext.create("Ext.container.Container", {
     html : "Raw Container",
     renderTo: Ext.getBody()
});
```

In the code snippet above, we've created an instance of Container class. This instance is empty as we've not added any components to it. The code displays a text Raw Container in the page. It generates the following HTML snippet.

```
<div id="container-1009" class="x-container x-container-default">Raw Container
<div id="container-1009-clearEl" class="x-clear" role="presentation"></div>
</div>
```

It's necessary to understand the common configuration attributes and methods of the Container class before you start working with the UI controls that are just derived classes of Container.

Configuration Attributes of Ext.container.Container

Let's discuss some of the configuration attributes of the Container class.

items

The items attribute refers to the collection of components that you'll add to the container. A Container class with a textbox and button component added to it using items is shown below.

```
Ext.create("Ext.container.Container",{
   items : [
            Ext.create("Ext.form.field.Text",{...}),
            Ext.create("Ext.button.Button",{...})
   ]
});
```

layout

This attribute is used to configure the layout for the container, so that the components may be arranged in a particular fashion. You'll learn more about the layout later in this chapter.

defaults

The defaults attribute is used to specify a set of default properties for all the items in the container. It helps you avoid duplication of code. If you want all the items in the container to have a specific width and height, then you can configure that using defaults as shown below.

```
Ext.create("Ext.container.Container",{
        defaults : {
        width:100,height:150
        },
        items : [
            ...
        ]
});
```

Some Methods in Container class.Methods of Ext.container.Container

add

The add method is used to dynamically add components into the container. When the components are added dynamically using the add method the container rearranges itself automatically. You can pass component or an array of components as argument to the add method.

```
var container1 = Ext.create("Ext.container.Container",{
 ...
});
var item1 = Ext.create("Ext.Component",{...});
container1.add(item1);
```

doLayout

doLayout method triggers the container to recalculate the layout and refresh itself.

down

This method, similar to the up method in Component class, is used to navigate to the descendants of the container that matches the expression passed as an argument. For example if you have a container, say *container1*, that has a button calling `container1.down("button")` walks down the container and returns the button component that's a child or grandchild or any descendant.

remove

The remove method is used to remove the components from the container. You can invoke remove method by passing the component or id of the component to be removed as argument.

```
var container1 = Ext.create("Ext.container.Container",{
 ...
});
var item1 = Ext.create("Ext.Component",{...});
container1.add(item1);
container1.remove(item1);
```

Let's discuss some of the events in Container class.

Events of Ext.container.Container

Table 4-3 shows the events in the Container class.

Table 4-3. *Events in Container class*

Event	Description
beforeadd	Fired before adding an item to the container
Add	Fired after an item is added
beforeremove	Fired before removing an item from the container
remove	Fired after removing an item from the container

Listing 4-2 shows the code snippet where you create a component and add to a Container. The Container has these events handled.

Listing 4-2. Events in Container

```
var container = Ext.create("Ext.container.Container", {
   html: "Default Container",
   listeners: {
      beforeadd: function () {
        console.log("beforeadd");
      },
      add: function () {
        console.log("add");
      },
      beforeremove: function () {
        console.log("beforeremove");
      },
      remove: function () {
        console.log("remove");
      }
   }
});

console.log("***Adding comp1");
container.add({
   xtype: "component", html: "Raw",id:"comp1"
});

console.log("***Removing comp1");
container.remove("comp1");
```

In Listing 4-2 we've registered the events using the listeners block. Here's the output of this code.

```
***Adding comp1
beforeadd
add
***Removing comp1
beforeremove
remove
```

`Ext.container.Container` is inherited by several classes that you'll commonly use. Table 4-4 shows the subclasses of Container class.

Table 4-4. *Subclass of Ext.container.Container*

Class	Description
Ext.container.Viewport	Represents the viewable area
Ext.panel.AbstractPanel	Base class for all the panel based containers
Ext.toolbar.Toolbar	Represents a toolbar

We've discussed the basics of the Component and Container classes. All the UI controls in Ext JS 4 are subclasses of these two classes. Let's discuss these UI controls.

Container Controls

Ext.panel.Panel

Ext.panel.Panel with the xtype *'panel'* is the root container class for several container classes. It's probably the most commonly used container class. You can create a Panel as shown below.

```
Ext.create("Ext.panel.Panel",{
     title : "Sample Panel",
     items : [
          ...
     ]
});
```

Ext.panel.Panel is inherited by a number of classes shown in Table 4-5.

Table 4-5. *Panel Controls*

Class	Description
Ext.form.Panel	Represents a form
Ext.menu.Menu	Represents a menu
Ext.window.Window	Represents a floatable, draggable window component
Ext.tab.Panel	Represents a tabbed container

Ext.window.Window

Window represents a floatable, draggable, resizable panel. Windows can be configured to be modal. You can create a Window as shown in Listing 4-3.

Listing 4-3. Window

```
var win = Ext.create("Ext.window.Window", {
  title: "Find and Replace",
  modal: true,
  items: [
   {
     xtype: "textfield",
     fieldLabel: "Find what"
   }
  ],
  buttons: [
   {
     text: "Find next"
   },
   {
     text: "Cancel"
   }
  ]
});
win.show();
```

In Listing 4-3 we've created a Window object with a textbox and two buttons using the items and buttons properties, respectively. Invoking the show method on the window object will show the window as shown in Figure 4-1.

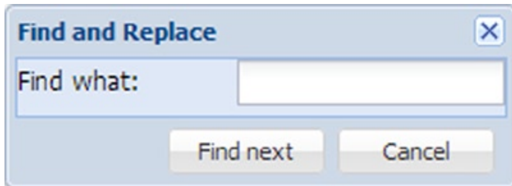

Figure 4-1. *A Window component*

The window will be a modal one masking the background completely as you've configured the modal property to be true.

Ext.menu.Menu

Ext.menu.Menu is the container that's used to display menus. Menu is made up of Ext.menu.Item controls. A menu can be shown as a standalone control or can be added as a child. A standalone menu can be created as shown in Listing 4-4.

Listing 4-4. Menu

```
var editMenu = Ext.create('Ext.menu.Menu', {
  items: [
    {
      text: 'Undo'
    },
    {
      text: 'Cut'
    },
    {
      text: 'Copy'
    },
    {
      text: "Paste"
    }
  ]
});
editMenu.show();
```

The menu is made of menu items. The default xtype of each menu item is a panel, and it has a text property that can be used to configure the text. You'll get the menu displayed as shown in Figure 4-2.

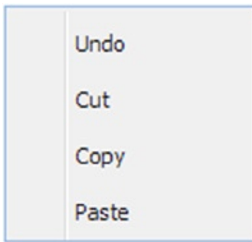

Figure 4-2. *Menu component*

You can add the menu as a child item as well. Let's add the menu to a Button using its menu attribute as shown below.

```
Ext.create("Ext.button.Button",{
    text : "Edit",
    menu : editMenu
});
```

You'll get an Edit button with the menu as shown in Figure 4-3.

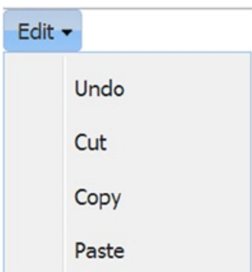

Figure 4-3. *Menu added to a Button*

Ext.tab.Panel

This class is used to create tabbed containers. It can be intepreted as a panel with the child items following a card layout. A tab panel has a tab bar represented by the Ex.tab.Bar class that can be positioned at the top, bottom, left, or right. Each tab in the panel is an object of the Ext.tab.Tab class. You can create a tab panel as shown in Listing 4-5.

Listing 4-5. Tab Panel

```
Ext.create('Ext.tab.Panel', {
  renderTo: Ext.getBody(),
  title: "Documentation",
  plain: false,
  height : 200,
  tabPosition: "bottom",
```

```
  items: [
    {
      title: 'Home',
      html : "Welcome to Ext JS 4"
    },
    {
      title: 'API',
      html : "API docs"
    },
    {
      title: 'Guides',
      html : "Standard guides"
    }
  ]
});
```

The tab panel has three tabs. The tab panel is configured to be plain, with no background for the tab bar. You'll get the tab panel as shown in Figure 4-4.

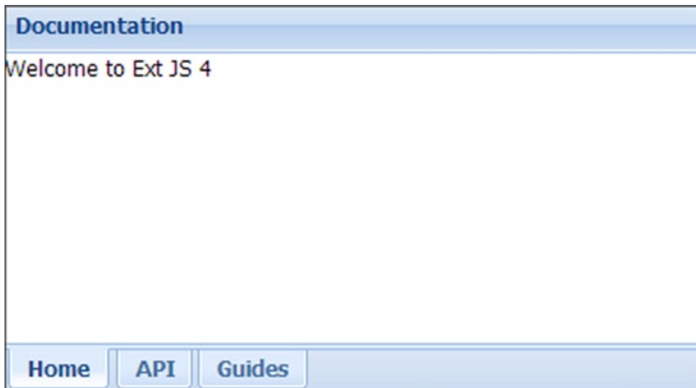

Figure 4-4. *Tabbed Pane*

Ext.form.Panel

Form panel class serves as the container for forms. You can add the controls in Ext.form.field package to the form panel class. The form panel provides support for processing form, validation, and so forth.

Ext.form and *Ext.form.field* are the packages that supply us the form controls. The list of commonly used UI controls along with their xtype is shown in Table 4-6.

Table 4-6. *Form Controls*

Class	xtype
Ext.form.field.Text	textfield
Ext.form.field.TextArea	textarea
Ext.form.field.Checkbox	checkbox
Ext.form.field.ComboBox	combobox
Ext.form.field.Radio	radio
Ext.form.field.Date	datefield
Ext.form.field.Number	numberfield
Ext.form.Label	label
Ext.form.RadioGroup	radiogroup
Ext.form.CheckboxGroup	checkboxgroup
Ext.form.FieldSet	fieldset

Let's create a form using some of these controls. We'll develop a page as shown in the Figure 4-5.

Figure 4-5. *A Form panel*

The controls form has a radio group, date field, number field, text area, and a button. Listing 4-6 shows the code for the form control.

Listing 4-6. Form Panel

```
Ext.create("Ext.form.Panel",
            {
  title : "Controls",
  items : [
   {
    xtype : "radiogroup",
    fieldLabel : "Title",
    vertical:true,columns:1,
   items : [
             {boxLabel:"Mr",name:"title"},
                    {boxLabel:"Ms",name:"title"}
    ]
   },
   {
    xtype : "textfield",
    fieldLabel : "Name"
   },
   {
    xtype : "datefield",
    fieldLabel : "Date of birth"
   },
   {
    xtype : "textfield",
    fieldLabel : "Blog"
   }
   {
    xtype : "numberfield",
    fieldLabel : "Years of experience",
    minValue : 5,
    maxValue : 15
   },
   {
    xtype : "textarea",
    fieldLabel : "Address"
   },
   {
    xtype : "button",
    text : "Submit"
   }
   ],
 });
```

The form controls can be wired up with basic validation rules. For instance, the common validation properties of the text based controls are allowBlank, maxLength, minLength, and so on. In the form we created, we can apply the validation rules to Listing 4-6 as shown below.

```
{
 xtype : "textfield",
 fieldLabel : "Name",
 allowBlank : false,
 maxLength : 50,
 msgTarget : "side"
},
{
      xtype : "datefield",
 fieldLabel : "Date of birth",
 msgTarget : "side"
}
```

The name textfield has validation rules used. The msgTarget displays the error message by the side of the textfield when the validation fails. The default value is qtip where the error message is displayed as a quick tip as shown in Figure 4-6.

Figure 4-6. *Form panel with validation rules*

Another useful property called vtype can be used for using built-in validation rules like e-mail, URL, and so forth. The blog text field we have used in our example can be configured to have a validation type as shown here.

```
{
  xtype : "textfield",
  fieldLabel : "Blog",
  vtype : "url"
}
```

The blog field will display an error message as shown in Figure 4-7.

Figure 4-7. *Text field with url vtype*

We can also register our own validation functions using validator property. The validator function is passed in the value of the field. It returns the error message or true based on the outcome of validation. The address field with a custom validator is shown below.

```
{
 xtype : "textarea",
 fieldLabel : "Address",
 validator : function(val){
  if(val.indexOf("#") != -1 || val.indexOf(".") != -1)
   return "Invalid characters like # or . in address";
  return true;
 }

}
```

The error message for the address field when the validation fails is shown in Figure 4-8.

Figure 4-8. *Text area with custom validation function*

The FormPanel has a submit() method that can be used to submit the form to the server. The form values are submitted to the server using AJAX by default. The server URL can be specified using the url property. The submit button's click event can be handled to submit the form. The form will be submitted only when there are no validation errors. The FormPanel's submit method can be invoked as shown in Listing 4-7.

Listing 4-7. FormPanel With a submit

```
Ext.create("Ext.form.Panel",
             {
   title : "Controls",
```

```
    url : "someUrl",
   items : [
    {
     xtype : "datefield",
     fieldLabel : "Date of Birth",
     name : "dob"
    },
    {
     xtype : "textfield",
     fieldLabel : "Blog",
     name : "blog"
    }
    {
      xtype : "button",
      text : "Submit",
      listeners : {
             "click" : function(src){
              src.up("form").submit();
              }
      }
    }
   ]
 });
```

The click listener for the button navigates to the form using the `up()` method. The form is automatically submitted to the configured `url` attribute. The form data is passed to the server using the name property of the elements. The server resource can access the form elements using their respective names.

The submit method can optionally accept an `Ext.form.action.Action` object as parameter with AJAX callback functions.

```
src.up("form").submit({
success :  function(form,action){
         alert("Successfully submitted");
        },
    failure : function(form,action){
      console.log(action.failureType);
      console.log(action.result);
alert(action.response.status + ", " + action.response.statusText);
      }
   });
```

The success and failure callback functions are invoked after the form submission. We can disable AJAX and opt for a normal form submission instead using the standardSubmit property. Inside the FormPanel we can set standardSubmit property to be true.

Ext.toolbar.Toolbar

This container class is used to create toolbars. The toolbar is composed of various child controls. The default component that is added to a toolbar is a button. You can add items declaratively to a toolbar and also dynamically using the `add()` method present in the Toolbar class. You can also add the following toolbar-related items to a toolbar apart from the regular collection of controls like textfield, label, and so on.

Ext.toolbar.TextItem (tbtext)

This class is used to render a simple text in a toolbar. You can use it as shown below.

```
{xtype:"tbtext", text:"Sample text"}
```

Ext.toolbar.Separator (tbseparator)

This class adds a vertical separator bar in the toolbar. You can use it as shown below.

```
{xtype:"tbseparator"}
```

You can use a Separator with a "-" hyphen symbol instead of configuring it using xtype.

Ext.toolbar.Spacer (tbspacer)

This class adds a default 2px space in the toolbar. You can use it as shown below.

```
{xtype:"tbspacer"}
```

You can use a Spacer with a " " blank space instead of configuring it using xtype.

Ext.toolbar.Fill (tbfill)

This class right justifies the items to be added after adding this item. You can use it as shown below.

```
{xtype:"tbfill"}
```

You can use a Fill with a "➤" right arrow instead of configuring it using xtype.

Ext.toolbar.Paging (pagingtoolbar)

This class is used to display a paging bar when you use data components like grid panel. We'll discuss this topic in Chapter 6 on data controls.

Let's create a panel with a toolbar at the bottom as shown in Figure 4-9.

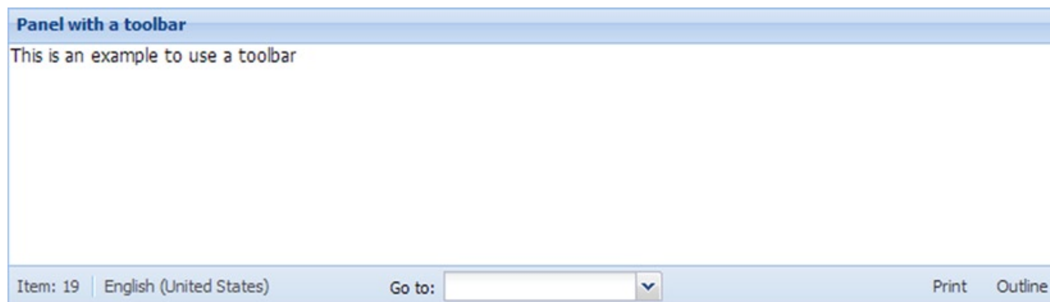

Figure 4-9. *Panel with a toolbar*

The toolbar contains text items, combobox, and buttons. Listing 4-8 shows the code snippet for creating a toolbar.

Listing 4-8. Panel With a toolbar

```
Ext.create("Ext.panel.Panel", {
     renderTo: Ext.getBody(),
     title: "Panel with a toolbar",
     html : "This is an example to use a toolbar",
     dockedItems: [
       {
         xtype: "toolbar",
         dock: "bottom",
         items: [
             {
               xtype: "tbtext",
               text: "Item: 19"
             },
             "-",
             {
               xtype: "tbtext",
               text: "English (United States)"
             },
             " ",
             {
                xtype : "combo",
                fieldLabel : "Go to",
                labelAlign : "right",
             },
             "➤",
             {
                text: "Print",
             },
             " ",
             {
                text: "Outline",
             }
         ]
      }
    ]
});
```

The toolbar is docked to the bottom of the toolbar. The toolbar has the spacer, separator, and tbfill items added using the shortcut notations. The combobox is intentionally empty.

Ext.container.Viewport

All the containers that we have discussed lack the capability to resize themselves when the browser window is resized. You've a specialized container for this purpose called Viewport. ViewPort is present in Ext.Container package and it represents the viewable browser area. Items added to the Viewport automatically get resized when the browser window is resized. Viewport is usually created as the root container of an application. Viewport is a container defined with an Auto layout by default, and it can be changed according to our requirement. We can create a Viewport with border layout as shown below.

```
Ext.create("Ext.container.Viewport",{
 layout : "border",
 items : [
  ..
]
});
```

Ext JS4 provides a number of layout controls that can be used to design our applications. Let's discuss these layout controls.

Layout Controls

All the container classes arrange their items in a specific fashion based on the layout you provide. A container with a table layout arranges the items in a tabular format, the one with a vbox layout arranges the components vertically. Ext.layout.Layout is the base class for all the layout classes. Layout class is inherited by the Ext.layout.container. Container that serves as the base class for all layout controls.

The *Ext.layout.container* package provides the different layout controls that are used to arrange our components. Table 4-7 shows the list of layout controls with a brief description.

Table 4-7. *Layout controls*

Class	xtype	Description
Absolute	absolute	Used to arrange the components by specifying the x- and y-coordinates.
Accordion	accordion	Denotes the accordion style
Anchor	anchor	Arrange the components relative to their container's position.
Border	border	Split the entire page into different regions. It's usually used to design an entire page in the application.
Card	card	The container's items are treated as a pack of cards and only one of them is shown at any point of time.
Form	form	The components are rendered one after the other as a typical form.
Table	table	Arrange the components in a tabular fashion.
HBox	hbox	Arrange the components horizontally.
VBox	vbox	Arrange the components vertically.
Fit	fit	The components of the container with fit layout is arranged to fit the entire area of the container.

The general configuration required for using any layout is given below.

```
Ext.create("container",{
 layout :  {
  type : "xtype of any layout control",
  //propertiesOfTheLayoutControl
 }

});
```

If you don't have any additional properties of the layout to be configured, the layout configuration is

```
Ext.create("container",{
 layout : "xtype of any layout control"
});
```

Let's discuss the various layout components.

Auto Layout

The default layout for the containers in Ext JS4 is Auto layout. This layout manager automatically renders the components in a container.

Fit Layout

The fit layout arranges the contents of the container to occupy the space completely. It fits the items to the container's size. Fit layout is usually used on containers that have a single item. Fit layout is the base class for the Card layout that we'll discuss later in this section. Let's add a text field to a Panel with a fit layout as shown below.

```
Ext.create("Ext.panel.Panel",{
 layout :  "fit",
       height:200,width:200,
       title : "Fit layout panel",
       items : [
                    {
                         xtype : "textfield",
                         fieldLabel : "Email"
                     }
      ]
});
```

The panel has a textfield that will be fit into the container to occupy the complete space as shown in Figure 4-10.

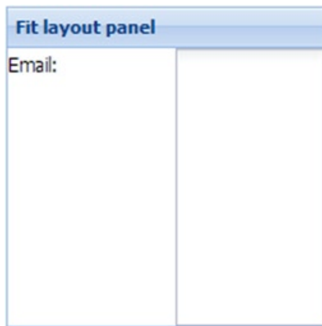

Figure 4-10. *Panel with Fit layout*

Anchor Layout

The anchor layout manager arranges the items of a container relative to the size of the container. Whenever the container is resized, the anchor layout manager rearranges the items relative to the new size of the container. You can configure an anchor property to the child items. You can configure the width and height values in percentage and the offset values in the anchor property as shown below.

```
anchor : "width% height%"
(or)
anchor : "offsetWidth offsetHeight"
```

You can also mix these two options by specifying an offset value and a percentage. Here's a simple panel that has a text field and a button and configured with an anchor layout. The items are configured with anchor attributes. Whenever you click the button, the width and height of the panel are increased by 5px. Here's the code for that.

```
var pnl = Ext.create('Ext.panel.Panel', {
      layout: "anchor",
      height: 200, width: 200,
      title: "Anchor layout panel",
      items: [
          {
            xtype: "textfield",
            fieldLabel: "Name",
            anchor : "90% 15%"
          },
          {
            xtype: "button",
            text: "Resize",
            anchor : "-80 -145",
            listeners: {
               click: function () {
                  pnl.setWidth(pnl.getWidth() + 5);
                  pnl.setHeight(pnl.getHeight() + 5);
               }
            }
```

```
            }
        ],
        renderTo: Ext.getBody()
});
```

By clicking on the resize button continuously, you'll find out that the size of the textfield and button increase proportionately.

Box Layout

`Ext.layout.container.Box` serves as the base class for VBox and HBox layouts. VBox and HBox stand for vertical box and horizontal box, respectively.

A Panel with three buttons (A, B, and C) using a VBox layout is shown below. The buttons are arranged vertically in the center of the panel as shown in Figure 4-11.

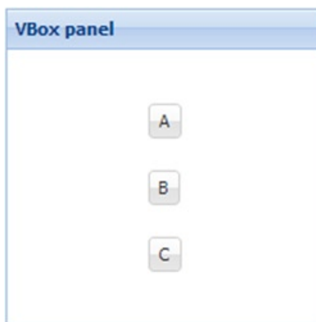

Figure 4-11. *Panel with a VBox layout*

The pack and align properties of the VBox layout are used for positioning the buttons inside the container. Listing 4-9 shows how to do that.

Listing 4-9. VBox Layout

```
Ext.create("Ext.panel.Panel", {
     height: 200, width: 200,
     title : "VBox panel",
     layout : {
 type : "vbox",
 pack : "center",
 align : "center"
     },
     defaults : {xtype : "button",margin:"10"},
     items : [
  {text : "A"},
  {text : "B"},
  {text : "C"},
            ],
    renderTo : Ext.getBody()
});
```

The code in Listing 4-9 can be modified to use the hbox layout. The layout configuration can be modified as shown below.

```
layout : {
          type : "hbox",
          pack : "center",
          align : "middle"
            }
```

The panel will look as shown in Figure 4-12.

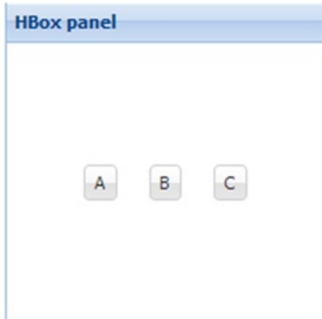

Figure 4-12. *Panel with HBox layout*

Accordion Layout

Accordion layout is an extension of VBox layout. It arranges a set of panels vertically with collapse and expandable features. Listing 4-10 shows the code snippet of a panel that uses the accordion layout.

Listing 4-10. Accordion Layout

```
Ext.create("Ext.panel.Panel", {
   height: 300, width: 300,
   title: "Accordion layout ",
   layout: {
              type : "accordion",
              multi : true
           },
   items: [
     {
  title: "Inbox",
                html : "Inbox contents"
            },
            {
                title: "Outbox",
                html: "Outbox contents"
            },
            {
```

```
                title: "Sent Items",
                html: "Sent Items"
            }
          ],
   renderTo: Ext.getBody()
});
```

The accordion layout is configured with a multi attribute which enables viewing multiple panels. The code throws up the output as shown in Figure 4-13.

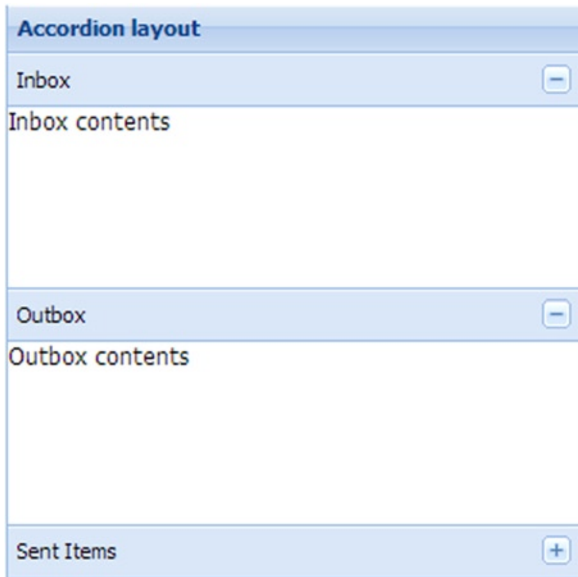

Figure 4-13. *Panel with Accordion layout*

Table Layout

Table layout is used to render a HTML table element. It has all the properties of a table, the most commonly used being the columns attribute. Listing 4-11 shows the code snippet of a panel that uses a table layout.

Listing 4-11. Table Layout

```
Ext.create("Ext.panel.Panel", {
   height: 200, width: 200,
   title: "Table layout ",
   layout: {
             type: "table",
             columns: 2
           },
```

```
    defaults: {
              xtype: "button",
              margin: "10"
            },
    items: [
             {
                text: "A"
             },
             {
                text: "B"
             },
             {
                text: "C"
             },
             {
                text: "D"
             },
             {
               text: "E"
             },
             {
               text: "F"
             }
    ],
    renderTo: Ext.getBody()
});
```

You'll get an output as shown in Figure 4-14.

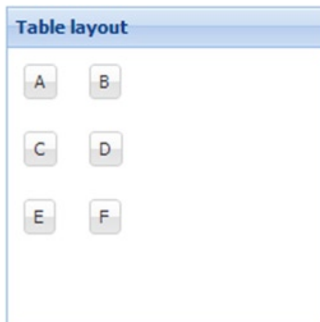

Figure 4-14. *Panel with a Table layout*

Column Layout

Column layout arranges the container in separate columns starting from left to right. Each item in the container that uses column layout is configured with a columnWidth attribute. The sum of the values of columnWidth attributes of all the items need to be equal to the total width of the container. You can provide the values of columnWidth in percentage or a concrete value. The percentage value is provided as a decimal number where the total columnWidth equals 1.

Let's create a panel with column layout as shown in Figure 4-15.

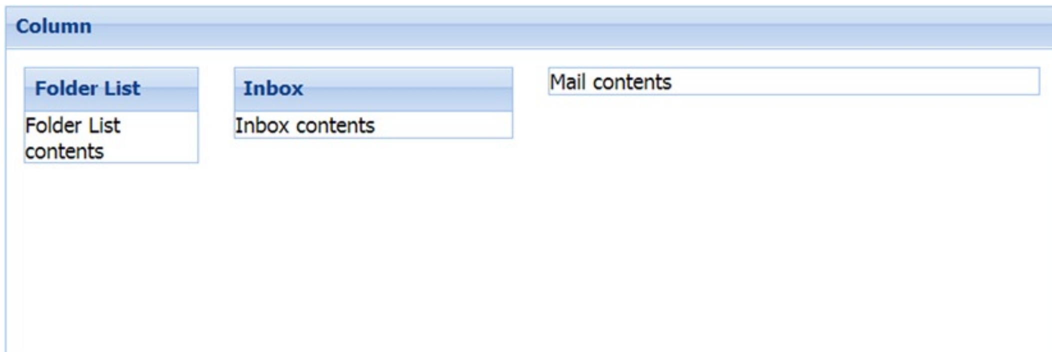

Figure 4-15. *Panel with a Column layout*

Listing 4-12 shows the code for that.

Listing 4-12. Column Layout

```
Ext.create('Ext.panel.Panel', {
  title: 'Column',
  width: 600, height: 200,
  layout: 'column',
  defaults : {margin : "10"},
  items: [
    {
      title : "Folder List",
      html : "Folder List contents",
      columnWidth : 0.20
    },
    {
      title: "Inbox",
      html: "Inbox contents",
      columnWidth : 0.30
    },
    {
      html: "Mail contents",
      columnWidth: 0.50
    }
  ],
  renderTo: Ext.getBody()
});
```

Border Layout

Border layout is usually the master layout in an Ext JS 4 application. You can design a master layout with regions like header, footer, and menus. In Ext JS 4, which is predominantly used for building single-page applications, border layout is used to design the entire layout of the page. The Ext JS 4 API documentation page, shown in Figure 4-16, is a good example of the use of border layout.

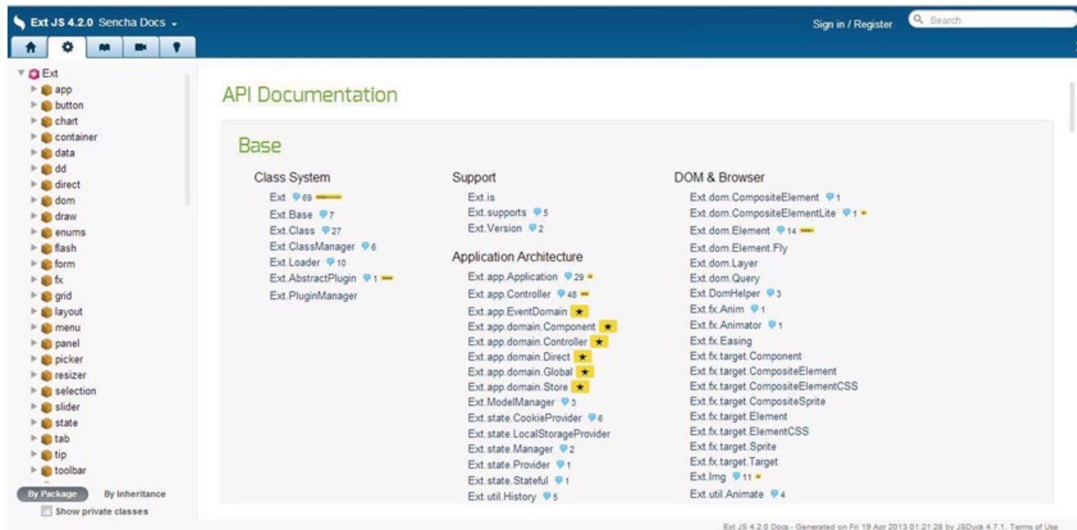

Figure 4-16. *A page that uses Border layout*

The border layout splits the page into different regions like north, south, east, west, and center. The center region has to be mandatorily configured while the other regions are optional. For example, in the figure above, you can say that the page has three regions, a header with a menu bar in the north, the list of classes in a tree format in the east, and the main content in the center.

Let's create a simple example of using Border layout. We'll create a panel that uses border layout as shown in Figure 4-17.

Figure 4-17. *A panel with a Border layout*

Listing 4-13 shows the code snippet for a panel with Border layout.

Listing 4-13. Border Layout

```
Ext.create("Ext.panel.Panel",{
   layout : "border",padding:30,id:"main",height:500,width:400,
   items : [
             {
               xtype : "panel",
               html : "Top ",
               region : "north"
             },
             {
               xtype : "panel",
               html : "Main contents",
               region : "center"
             },
             {
               xtype : "panel",
               html : "Side bar",
               collapsible : false,
               split : true,
               region : "west"
             }
   ],
   renderTo : Ext.getBody()
});
```

The Panel is split into north, west, and center regions. In this example each region has a panel configured in it. The west region has optional properties like collapsible, split, and so forth, configured to be able to hide and resize the region dynamically.

Card Layout

You have a Panel with a number of child components and only one child control needs to be shown at a time. The Panel can use a card layout for this purpose. Card layout, when used on a container, treats its items as a collection of cards and shows only one item at any point of time.

Card layout has an important property called *activeItem* that holds the information about the item that has to be displayed. This property has to be manipulated to change the item to be shown. Listing 4-14 shows the code snippet for using card layout.

Listing 4-14. Card Layout

```
Ext.create("Ext.panel.Panel",{
    layout : "card",padding:30,id:"main",
    items : [
      {
        xtype : "panel",
        title : "Screen 1",
        items : [
```

```
          {
            xtype : "button",
            text : "go to screen 2",
            handler : function(){
              Ext.getCmp("main").getLayout().setActiveItem(1);
            }
          }
        ]
      },
      {
        xtype : "panel",
        title : "Screen 2",
        items : [
          {
            xtype : "button",
            text : "go to screen 3",
            handler : function(){
              Ext.getCmp("main").getLayout().setActiveItem(2);
            }
          }
        ]
      },
      {
        xtype : "panel",
        title : "Screen 3"
      }
    ],
  renderTo : Ext.getBody()
});
```

The main Panel uses the card layout. It has three panel children. The Screen1 and Screen2 panels have a button when clicked change the active item of the card layout. The setActiveItem method on the Card Layout class accepts a number that represents the index of the controls as the parameter. The setActiveItem can accept the id of the control as a parameter as well. If Screen 2 panel's id is "screen2", we can the change the active item by

```
Ext.getCmp("mainpanel").getLayout().setActiveItem("screen2")
```

We can also pass the component as the parameter like this:
Ext.getCmp("mainpanel").getLayout().setActiveItem(Ext.create("Ext.Button",{...})). This can be used if you create a new object and set it as an active item, instead of creating it beforehand and not showing it. Figure 4-18 shows the output of the code in Listing 4-14. You'll get Screen 1 panel, and when you click the "Got to Screen 2" button, you'll get Screen2. Clicking on "Go to Screen 3" button will give you the Screen 3. Please note that only one screen is showed at any point of time in card layout.

Figure 4-18. *Output of Card layout example*

One of the advantages of using Ext JS 4 as we discussed earlier is the support for writing modularized code. In the card layout code snippet shows in Listing 4-10 you can bring in some modularity by organizing the items of the main panel into individual classes. Listing 4-15 shows a modularized version of using card layout.

Listing 4-15. Modularized Version of Card Layout

```
Ext.define("Screen1",{
   extend : "Ext.panel.Panel",
   xtype : "screen1",
   title : "Screen 1",
   items : [
     {
       xtype : "button",
       text : "Go to Screen 2",
       handler : function(){
         Ext.getCmp("viewport").getLayout().setActiveItem(1);
       }
     }
   ]
});

Ext.define("Screen2",{
   extend : "Ext.panel.Panel",
   xtype : "screen2",
   title : "Screen 2",
   items : [
     {
       xtype : "button",
       text : "Go to Screen 3",
       handler : function(){
         Ext.getCmp("viewport").getLayout().setActiveItem(2);
       }
     }
   ]
});

Ext.define("Screen3",{
   extend : "Ext.panel.Panel",
   xtype : "screen3",
   title : "Screen 3"
});

Ext.onReady(function(){
   Ext.create("Ext.container.Viewport",{
     layout : "card",padding:30,id:"viewport",
     items : [
       {
         xtype : "screen1",
         id : "screen1"
       },
```

```
        {
          xtype : "screen2",
          id : "screen2"
        },
        {
          xtype : "screen3",
          id : "screen3"
        }
      ],
      renderTo : Ext.getBody()
  });
}
);
```

In the example above, we've defined three new classes Screen1, Screen2, and Screen3. These classes inherit Ext.panel.Panel and have screen1, screen2, and screen3 as xtypes, respectively. The main container, a Viewport with card layout, contains the instances of these classes. The Screen1 and Screen2 classes have simple buttons with their handlers taking care of the navigation to the next item. The best place to write the handler logic is a Controller class. We'll discuss the controller classes in our MVC chapter.

Summary

In this chapter I discussed the controls and layout of Ext JS 4. All UI controls inherit from Ext.Component class. Ext.Component provides several properties, methods, and events. The Ext.container.Container class serves as the base class for all the container controls like Panel, Viewport, and Toolbars. The form panel represents the standard HTML form with functionalities like validations, processing and so on. You can add the field controls to the form panel. Ext.layout.Layout is the base class for all the layout components. Border Layout defines the master layout of an application and Card layout is used to show one item at any point of time.

In the next chapter you'll learn the data handling mechanisms in Ext JS 4. I'll discuss the Ext.data package in detail and take a look at the core concepts of fetching, saving, and updating data from various data sources.