



Extending, Unit Testing, and Packaging

In the first nine chapters of this book we have analyzed various features of Ext JS 4. You learned how to use the UI controls, work with data components, create custom themes, and build applications that follow the MVC architecture. In this chapter you'll find out about some miscellaneous features in Ext JS 4. For example, you'll see how to extend the UI controls by creating custom components. I'll discuss various options involved in creating custom components and plugins. You'll also learn how to write unit tests in JavaScript to test our Ext JS 4 application. Finally you'll learn how to create an Ext JS 4 application from scratch, package it, and deploy it to the web server.

Extending the UI

In Chapter 9 when I discussed the MVC architecture, we created classes that extended some built-in classes such as `Ext.panel.Panel`, `Ext.grid.Panel`. These derived classes just modified the attributes of the built-in classes without really making a drastic change to their look and feel or behavior. In this section let's take a look at how to create custom UI components, custom plugins, etc., from scratch.

Custom Components

Let's start with a `HelloWorld` component that we will use, like this.

```
{xtype:"helloworld"}
```

Let's implement the `helloworld` component in such a way that it emits the following HTML code.

```
<span>Hello World</span>
```

We can implement this component by extending the `Ext.Component` class. The `autoEl` attribute of the `Component` class can be configured to provide the tag name and inner HTML of the element that you want to create.

Listing 10-1 shows the code for the `HelloWorld` component.

Listing 10-1. HelloWorld Component

```
Ext.define("HelloWorld",{
  xtype : "helloworld",
  extend : "Ext.Component",
  autoEl : {
    tag : "span",
    html : "Hello World"
  }
});
```

When you use an instance of the HelloWorld component in an application you will get the Hello World text displayed on the screen with the following generated HTML code snippet.

```
<span id="helloworld-1010" class="x-component x-component-default">Hello World</span>
```

As shown in Listing 10-1 we have specified the HTML data using the autoEl attribute. The autoEl attribute corresponds to the Ext.DomHelper object. The autoEl attribute can have properties like tag, html, cls, and children for decorating the HTML code that will be generated. Any property other than these four items is treated as the attribute of the generated HTML tag.

Let's create a hyperlink component that will be used like this.

```
{
  xtype:"link",
  url:"http://www.apress.com",
  text:"Apress Inc"
}
```

Listing 10-2 shows the code snippet for the link component built using autoEl attribute.

Listing 10-2. Link Component

```
Ext.define("Link",{
  extend : "Ext.Component",
  xtype : "link",
  autoEl : {
    tag : "a",
    html : "Click",
    href : "#"
  },
  initComponents : function(){
    if(this.text)
      this.autoEl.html = this.text;
    if(this.url)
      this.autoEl.href = this.url;
    this.callParent(arguments);
  }
});
```

As shown in Listing 10-2, we have used the `autoEl` property of the `Component` class to create an anchor element. We've overridden the `initComponent()` method where we initialize the `autoEl` attribute using the values passed while creating the instance of this component. The `initComponent()` method can be treated as the constructor for custom components. You can write your initialization logic in it. You need to have a call default `initComponent()` method in the `Component` class using `this.callParent()` method.

When you use the `Link` component in an application you will get a hyperlink on the screen with the following generated HTML code snippet.

```
<a id="link-1010" class="x-component x-component-default"
href="http://www.apress.com">Apress Inc</a>
```

Figure 10-1 shows the output of the link component.

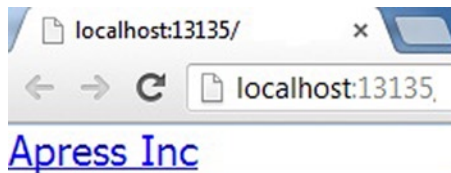


Figure 10-1. Link component

The `autoEl` attribute of the `Component` class can become tedious if you want to build a complex component where the HTML snippet that you're generating is a lot more verbose—such as a `<table>` element, for instance. Here's where you can use `Ext.XTemplate`. The `Ext.Component` class has two attributes: `tpl` for configuring `XTemplate`, and data for providing values to the template.

Let's build our link component using `XTemplate`. Listing 10-3 shows the code for the link component using `XTemplate`.

Listing 10-3. Link Component Using `XTemplate`

```
Ext.define("Link",{
  xtype : "link",
  extend : "Ext.Component",
  tpl : '<a href="{url}">{text}</a>',
  initComponent : function(){
    this.data = {
      text : this.text,
      url : this.url
    };
    this.callParent(arguments);
  }
});
```

As shown in Listing 10-3, the `Link` class has the `tpl` attribute configured. In the `initComponent()` method, we initialize the data attribute with the values for the `url` and `text` properties.

Let's throw in some validation for the `url` that's passed to the `link` component. You can write a validation function in the `XTemplate` instance and invoke it. In Listing 10-3, the `tpl` attribute can be modified to include the function like this:

```
Ext.define("Link",{
  ...
  tpl : Ext.create("Ext.XTemplate",
    '<a href="{[this.validateUrl(values.href)]}">{text}</a>',
    {
      validateUrl: function (url) {
        var valid = url.match(/^(ht|f)tps?:\/\/[a-z0-9-\.\+]\.[a-
z]{2,4}\.\/?([\s<>\#\%\,\{\}\|\|\|\^\\[\]\`]+)?$/);
        return valid?url:"#";
      }
    }
  ),
  ...
});
```

As you notice in the `tpl` attribute, we've created an instance of `XTemplate` using `Ext.create()` method and have a `validateUrl()` function that validates the `url` using a regular expression. If the `url` is valid, the function returns the `url`. Otherwise it returns a hash (`#`). We've invoked this function using the expression `{[this.validateUrl(values.url)]}` where `values` refers to the data attribute of the `XTemplate` class.

Let's add a custom event to this component. When you click the anchor element, let's alert a message. Handling the click event for an anchor element is really easy in the traditional approach where you use the `onclick` handler, but it's not the same in this case. We may have to wire up the click event programmatically with the generated anchor element. Let's override the `onRender()` method in our `Link` class where we programmatically wire up the click event on the anchor element. In Listing 10-3 we modify the `Link` class to add the `onRender()` method.

```
Ext.define("Link",{
  ..
  onRender: function () {
    this.callParent(arguments);
    this.mon(this.el,
      "click",
      function () {
        alert("Clicked");
      },
      this);
  }
});
```

We have overridden the `onRender()` method, where the `click` event is registered with a handler function using the `mon()` method. The `mon()` method is used for adding listeners to events. The arguments for the `mon()` method are the HTML element, event name, event handler function, and the scope in which the handler function is executed.

Let's create our own event—say, **“go”**—which will be called when you click the `link` component. Also, you want to add a listener using the `listeners` property as shown below.

```

{
  xtype:"link",
  url:"http://www.apress.com",
  text:"Apress Inc",
  listeners : {
    go : function(){
      alert("You clicked the link");
    }
  }
}

```

Ext.Component class provides a `addEvents()` method that can be used to add custom events. You can raise the event using the `fireEvent()` method. Listing 10-4 shows the link component with a custom event called go. The go event is fired when you click the anchor element.

Listing 10-4. Link Component With go Event

```

Ext.define("Link", {
  xtype: "link",
  extend: "Ext.Component",
  tpl: '<a href="{url}">{text}</a>',
  initComponents: function () {
    this.data = {
      text: this.text,
      url: this.href
    };
    this.addEvents("go");
    this.callParent(arguments);
  },
  onRender: function () {
    this.callParent(arguments);
    this.mon(this.el,
      "click",
      function () {
        this.fireEvent("go");
      },
      this);
  }
});

```

As shown in Listing 10-4, in the `initComponent()` method we have the go event added using `addEvents()` method. In the `onRender()` method, we have the go event fired using the `fireEvent()` method. The `fireEvent()` method is called from the event handler of the traditional click event.

I explained the use of plugins in components like grid in Chapter 6. Let's develop a custom plugin now.

Custom Plugin

Plugins in Ext JS 4 (which you saw in Chapter 6 on data controls) are used to inject custom functionality to the UI components.

Let's create a plugin for the `Ext.panel.Panel` class. We'll develop a timer plugin and apply it to the panel. The timer plugin will display a timer for 10 seconds. After 10 seconds, the timer is stopped and the panel is disabled. Figure 10-2 shows the screenshot of the Panel with a timer plugin.

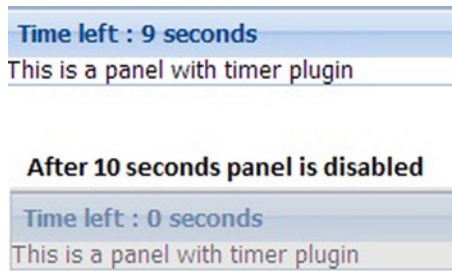


Figure 10-2. Panel with a timer plugin

You can create a custom plugin by extending the `Ext.AbstractPlugin` class. The `AbstractPlugin` class has an `init(component)` method where the `component` argument corresponds to the underlying UI component where the plugin is used. In our case the component refers to the `Panel` object. Inside this `init()` method we'll start a timer that runs for 10 seconds. Listing 10-5 shows the timer plugin.

Listing 10-5. Timer Plugin

```
Ext.define("TimerPlugin", {
    extend: "Ext.AbstractPlugin",
    alias: "plugin.timerplugin",
    limit: 10,
    count: 0,
    intervalId: null,
    init: function (component) {
        var me = this;
        this.intervalId = window.setInterval(function () {
            me.timer(component);
        }, 1000);
    },
    timer: function (component) {
        if (this.count != this.limit) {
            this.count += 1;
            component.setTitle("Time left : " + (this.limit - this.count) + " seconds");
        }
        if(this.count == this.limit) {
            component.disable();
            window.clearInterval(this.intervalId);
        }
    }
});
```

In Listing 10-5 we've created a `TimerPlugin` class that extends `AbstractPlugin`. The `init()` method uses the `window.setInterval()` method, which runs every 1000 milliseconds and calls a `timer()` method.

In the `timer()` method we have the logic to compute the end time and update the panel's title. When the count variable reaches the limit, the panel is disabled.

Listing 10-6 shows the `Panel` object that uses the timer plugin.

Listing 10-6. Panel that Uses the Timer Plugin

```
Ext.create("Ext.panel.Panel", {
    title: "Timer",
    html: "This is a panel with timer plugin",
    plugins : [{ptype:"timerplugin"}]
});
```

Unit Testing Ext JS 4

Code written in any language needs to be tested. That is a well-accepted fact in the programming world, and JavaScript code is no exception. Since JavaScript is used on the client side and a lot of developers mix it with HTML, it can be difficult to test the JavaScript code alone. It's important to keep the JavaScript code decoupled from the UI until you can test it.

We don't have to worry about writing modularized and decoupled JavaScript code in Ext JS 4 due to the MVC architecture we follow. So it's pretty easy to test the Ext JS 4 code. There are a number of tools and libraries like Ext Spec, Siesta, Jasmine, etc., available for testing Ext JS 4 code. We'll use the Jasmine toolkit for testing Ext JS 4 code.

Jasmine is a JavaScript unit testing library. In fact, it's more than a mere testing library. Jasmine is a Behavior-Driven Development (BDD) library. BDD is a development methodology based on Test-Driven Development and Domain-Driven Design. We'll not delve into BDD, however; instead, we'll focus on the unit testing abilities of Jasmine. Jasmine has a simple syntax to unit-test JavaScript code. Jasmine API does not come with the burden of DOM on its shoulders.

While this section is about using Jasmine to test Ext JS 4 code, let's start with an example of using Jasmine for testing plain JavaScript code, where you'll learn the basics of Jasmine. After that, we'll discuss how to use Jasmine to test an Ext JS 4 application.

You can visit <https://github.com/pivotal/jasmine/downloads> and download the latest stable version of Jasmine. At the time of writing, the latest version is 1.3.1. When you download the standalone zip file and extract it, you get the files shown in Figure 10-3.

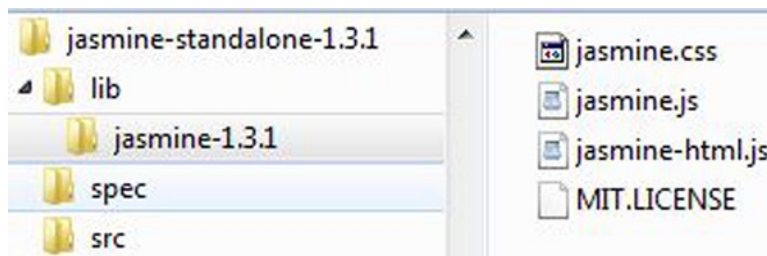


Figure 10-3. Jasmine extract

As shown in Figure 10-3, you need the contents of the lib folder in your application. In our case you need the jasmine-1.3.1 folder. The three files `jasmine.css`, `jasmine.js`, and `jasmine-html.js` need to be referenced in our web page that runs all the tests. Let's create a HTML page—say, `sample-tests.html`—and set up a Jasmine environment. Listing 10-7 shows the code snippet for the `sample-tests.html` file.

Listing 10-7. sample-tests.html

```

<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css" href="jasmine-1.3.1/jasmine.css">
  <script type="text/javascript" src="jasmine-1.3.1/jasmine.js"></script>
  <script type="text/javascript" src="jasmine-1.3.1/jasmine-html.js"></script>
  <script>
    function init() {
      jasmine.getEnv().addReporter(new jasmine.HtmlReporter());
      jasmine.getEnv().execute();
    }
    window.onload = init;
  </script>
</head>
<body>
</body>
</html>

```

As shown in Listing 10-7, we've added the Jasmine stylesheet and scripts. In the `init()` function we add an `HtmlReporter` provided by the Jasmine API to the Jasmine environment that will generate an HTML report for the tests we execute. You can run *sample-tests.html* file just to make sure there are no errors reported by the browser.

Let's write some simple JavaScript code and test it using Jasmine. As our intention is to get accustomed to Jasmine basics, let us write simple `add()` and `subtract()` functions. We'll create a file `calc.js` and implement the `add()` and `subtract()` functions over there. Listing 10-8 shows `calc.js` with `add()` and `subtract()` functions.

Listing 10-8. calc.js

```

function add(num1, num2) {
  return num1 + num2;
}

function subtract(num1, num2) {
  return num1 - num2;
}

```

Let's start writing the tests, using the Jasmine API to test the `add()` and `subtract()` functions shown in Listing 10-8. In Jasmine terminology the tests are called specifications, or specs. You create a suite of specs and execute them.

The basic functions you would use in Jasmine to write the specs are listed below.

- `describe()`
 - You create a test suite using the `describe` function. The `describe()` function is composed of specs. The general format of the `describe()` function is

```

describe("name of the test suite", function(){
  //collection of specs
})

```


- `it()`
 - Each spec or the test is specified by the `it()` function. The format of the `it()` function is

```
it("name of the spec",function(){
  //the actual test code
})
```

- `expect()`
 - The `expect()` function specifies the expectations and is used for performing the actual check or the test. For the Java or C# developers, `expect()` is equivalent to the `assert()` function. The `expect()` function is chained with `Matcher` functions to perform the test. The `expect()` function takes in a value that is matched with the expected value. Here are some examples of using the `expect()` function:

```
expect(123).toEqual(123)
expect(someVar).toBeDefined()
expect(true).toBe(true)
```

There are some more functions in Jasmine, such as the `describe()`, `it()`, and `expect()`, that we can use. We'll not get into those as it's beyond the scope of this chapter. If you want to read more about the Jasmine API, you can read its documentation at <http://pivotal.github.io/jasmine/>.

Let's develop our specs for the `add()` and `subtract()` methods as shown in Listing 10-9. We'll write our code in a file called 'calcspecs.js'.

Listing 10-9. calcspecs.js

```
describe("Addition", function () {
  it("test add() is defined", function () {
    expect(add).toBeDefined();
  });
  it("test add 2 simple numbers", function () {
    expect(add(1,2)).toEqual(3);
  });
});

describe("Subtraction", function () {
  it("test subtract() is defined", function () {
    expect(subtract).toBeDefined();
  });
  it("test subtract 2 simple numbers", function () {
    expect(subtract(11, 2)).toEqual(9);
  });
});

describe("Multiplication", function () {
  it("test multiply() is defined", function () {
    expect(multiply).toBeDefined();
  });
});
```

We've created three suites—namely, Addition, Subtraction, and Multiplication. The Addition and Subtraction suites have two specs each. The Multiplication suite has a spec that checks whether multiply function is defined. This spec will throw an error obviously.

You have to modify the sample-tests.html file to include calc.js and calcspecs.js files. Figure 10-4 shows the output when you run the sample-tests.html file.

```
Jasmine 1.3.1 revision 1354556913 finished in 0.028s
... . X

Failing 1 spec No try/catch 
5 specs | 1 failing

Addition
  test add() is defined
  test add 2 simple numbers

Subtraction
  test subtract() is defined
  test subtract 2 simple numbers

Multiplication
  test multiply() is defined
```

Figure 10-4. Output of sample-tests.html

Now that you've learned the basics of Jasmine, let's test our Ext JS 4 application using Jasmine.

To begin with, let's create a simple MVC application where we display a grid and the details of each item in the grid in a window as shown in Figure 10-5.



Figure 10-5. MVC application to be tested using Jasmine

As shown in Figure 10-4, the grid shows a list of countries and when you click on any item you see the details of the country shown in a pop-up window. Let's implement this application using the MVC architecture. As we have discussed MVC in detail in Chapter 9, we'll take a fast-track approach to get to the testing part. Figure 10-6 shows the app folder that contains the code for the application.

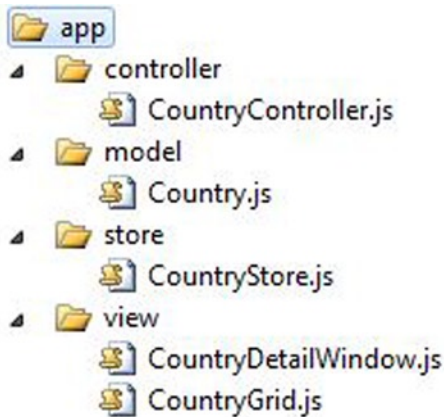


Figure 10-6. File structure for the MVC application to be tested using Jasmine

Listing 10-10 shows the code for the Country and CountryStore classes.

Listing 10-10. Country and CountryStore

```
Ext.define("Chapter10.model.Country",{
  extend : "Ext.data.Model",
  fields : ["name","capital","continent"]
});
Ext.define("Chapter10.store.CountryStore",{
  extend : "Ext.data.Store",
  autoLoad : false,
  model : "Chapter10.model.Country",
  proxy : {
    url : "countries.txt",
    type : "ajax",
    reader : {
      type : "json",
      root : "countries"
    }
  }
});
```

Listing 10-11 shows the code for the view classes CountryGrid and CountryDetailWindow.

Listing 10-11. CountryGrid and CountryDetailWindow Classes

```

Ext.define("Chapter10.view.CountryGrid", {
    extend: "Ext.grid.Panel",
    xtype: "countrygrid",
    store: "CountryStore",
    columns: [
        { header: "Name", dataIndex: "name" },
        { header: "Capital", dataIndex: "capital" }
    ]
});
Ext.define("Chapter10.view.CountryDetailWindow", {
    extend: "Ext.window.Window",
    xtype: "countrydetail",
    title: "Detail", height: 75, width: 200, padding: 2,
    layout : "vbox",
    items: [
        { xtype: "label", id: "capitallabel" },
        { xtype: "label", id: "continentlabel" }
    ]
});

```

Listing 10-12 shows the code for the CountryController class.

Listing 10-12. CountryController Class

```

Ext.define("Chapter10.controller.CountryController", {
    extend: "Ext.app.Controller",
    models: ["Country"],
    stores: ["CountryStore"],
    views: ["CountryGrid", "CountryDetailWindow"],
    refs: [
        { ref: "countryGrid", selector: "countrygrid" },
        { ref: "countryDetail", selector: "countrydetail" },
        { ref: "continent", selector: "countrydetail label[id=continentlabel]" },
        { ref: "capital", selector: "countrydetail label[id=capitallabel]" },
    ],
    init: function () {
        Ext.getStore("CountryStore").load();
        this.control({
            "countrygrid": {
                itemclick: this.onCountryClicked
            }
        });
    },
    onCountryClicked: function (src, record) {
        if (!this.getCountryDetail())
            Ext.create("Chapter10.view.CountryDetailWindow");
        this.getCountryDetail().setTitle(record.get("name"));
        this.getCapital().setText(record.get("capital"));
    }
});

```

```

    this.getContinent().setText(record.get("continent"));
    this.getCountryDetail().show();
  }
});

```

The root namespace name of the application is Chapter10, as you can notice from the code snippets above. We'll ignore the app.js file as it's not of much importance in this example.

Let's set up the Jasmine environment and create the specs. We'll create two specs—one each for testing the CountryStore and CountryController classes. Let's create a folder called app-test and store the jasmine library and the specs in it. We'll configure the test application in a file called app-test.js and the main HTML file that will be executed will be tests.html file. Figure 10-7 shows the file structure of the project after adding the above mentioned files.

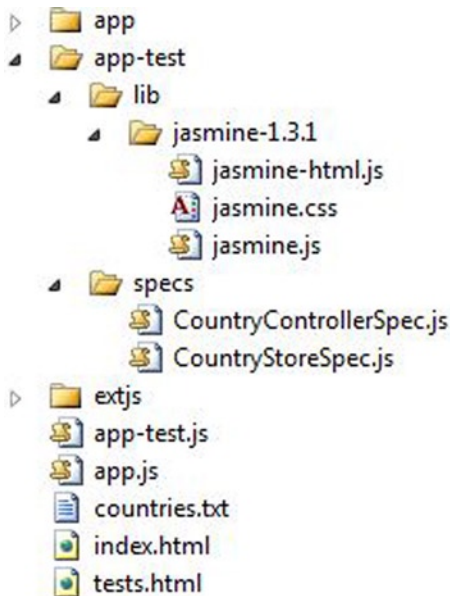


Figure 10-7. File Structure of the MVC application with Jasmine library

As you notice in Figure 10-7, I have used Jasmine version 1.3.1.

Listing 10-13 shows the code snippet for tests.html file that will be executed in the browser.

Listing 10-13. tests.html

```

<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css" href="app-test/lib/jasmine-1.3.1/jasmine.css">
  <script src="extjs/ext-all.js"></script>
  <script type="text/javascript" src="app-test/lib/jasmine-1.3.1/jasmine.js"></script>
  <script type="text/javascript" src="app-test/lib/jasmine-1.3.1/jasmine-html.js"></script>
  <script src="app-test.js"></script>

```

```

<script src="app-test/specs/CountryStoreSpec.js"></script>
<script src="app-test/specs/CountryControllerSpec.js"></script>
</head>
<body>
</body>
</html>

```

We've added the Jasmine related files, `app-test.js` and the two specs in `tests.html`.

Listing 10-14 shows the code snippet for `app-tests.js` file where the test environment is set up.

Listing 10-14. `app-test.js`

```

Ext.Loader.setConfig({enabled : true});

var Application = null;

Ext.onReady(function() {
  Application = Ext.create('Ext.app.Application', {
    name: 'Chapter10',
    controllers: ["CountryController"],
    launch: function() {
      jasmine.getEnv().addReporter(new jasmine.HtmlReporter());
      jasmine.getEnv().execute();
    }
  });
});

```

The interesting difference between what we wrote in `app.js` and what we've written in `app-test.js` is the way we create the instance of `Ext.app.Application` class. As shown in the code snippet in Listing 10-14, we explicitly create an instance of the `Application` class and assign it to a global variable called `Application`. This global variable will be used in the specs we'll implement later. The `launch()` function has the Jasmine environment configured.

Let's create the specs now. You can create any number of specs. You have to add them to the test page. Let's create a spec to test the `CountryStore`. You can give any name for the spec file. The `CountryStore` spec will load the store data and test if the data is properly loaded. Listing 10-15 shows the code snippet for `CountryStoreSpec.js`.

Listing 10-15. `CountryStoreSpec.js`

```

describe("Country Store", function () {
  var store = null;

  beforeEach(function () {
    store = Application.getStore("CountryStore");
    store.load();
    waitsFor(function () {
      return !store.isLoading(); },
      "Unable to load countries.txt",
      5000);
  });

  it("test store data", function () {
    expect(store.getCount()).toEqual(4);
    var country = store.getAt(0);

```

```

    expect(country.get("name")).toEqual("India");
    expect(country.get("capital")).toEqual("New Delhi");
    expect(country.get("continent")).toEqual("Asia");
  });
});

```

In Listing 10-15 we have written a spec to test the store data, where we test for the record count and also check the first records' details. The `beforeEach()` function in Jasmine is a set up method that is called before running each spec. It's not of prime importance here as there's only one spec in this suite.

In the `beforeEach()` function we access the store instance and call the `load()` method. Since the store loads records asynchronously, we give it 5000 milliseconds and check if the store has completed loading the records. The whole asynchronous loading process is achieved using the `waitFor()` function provided in Jasmine.

You can add more specs to test the `CountryStore` instance based on your needs.

Let's create the `CountryControllerSpec.js`, which has the specs to test the `CountryController` class. We'll test the references in the `CountryController` and check if the details of the country are displayed in a window when the row in the grid is clicked. Listing 10-16 shows the code for `CountryControllerSpec.js`.

Listing 10-16. `CountryControllerSpec.js`

```

describe("CountryController", function () {
  var countryController = null;
  var countryGrid = null;
  var countryStore = null;

  beforeEach(function () {
    countryController = Application.getController("CountryController");
    countryGrid = Ext.create("Chapter10.view.CountryGrid");
    countryStore = Application.getStore("CountryStore");
    countryStore.load();
    waitFor(function () {
      return !countryStore.isLoading();
    }, "Unable to load countries.txt", 5000);
  });

  it("test Country Grid", function () {
    var grid = countryController.getCountryGrid();
    expect(grid).toBeDefined();
    expect(grid.columns.length).toEqual(2);
  });

  it("test country grid item click", function () {
    var grid = countryController.getCountryGrid();
    grid.fireEvent("itemclick", grid.getView(), countryStore.getAt(0));
    expect(countryController.getCountryDetail()).toBeDefined();
  });
});

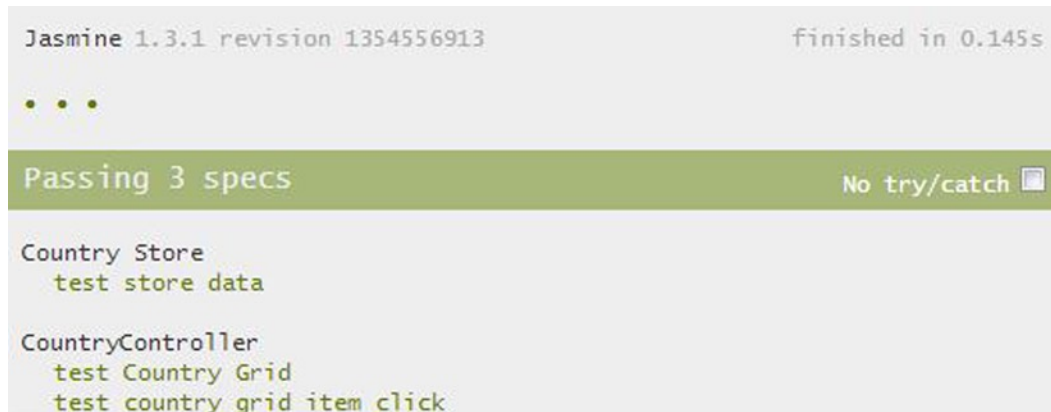
```

In the code snippet in Listing 10-16 we've a `beforeEach()` function where we initialize the `countryStore`, `countryController` and create an instance of the `CountryGrid`.

In the *'test country grid'* spec we access the country grid using the `getCountryGrid()` method generated for the reference variable `countryGrid` in the `CountryController`.

The `'test country grid item click'` spec is interesting. We programmatically fire the `itemclick` event using the `fireEvent()` method by passing the first record in the store as one of the arguments. According to the code in Listing 10-12, clicking the item in the grid will invoke the `onCountryClicked()` method and create an instance of the `CountryDetailWindow`. We check if the `CountryDetailWindow` object is created or not.

Running `tests.html` page will give you the output shown in Figure 10-8.



```

Jasmine 1.3.1 revision 1354556913 finished in 0.145s
...
Passing 3 specs No try/catch
Country Store
  test store data
CountryController
  test Country Grid
  test country grid item click

```

Figure 10-8. The output of the `tests.html` page

We've explored the basics of testing an Ext JS 4 application using Jasmine. Apart from the options discussed here, there are various parts of an Ext JS 4 application that you can effectively test using Jasmine to make your application foolproof. You also need to be careful while unit testing Ajax calls to the server resources as it will take a long time to do that. You can do it by creating mock data and unit testing it. It's also important to be a little cautious while unit testing complex UI interactions like drag and drop behavior, grid, and tree interactions as you may have to tweak your code a bit to make it suitable for testing.

Let's try creating and deploying an Ext JS 4 application using the Sencha Cmd tool.

Packaging

We have been creating an Ext JS 4 application manually from scratch, adding all the source files, CSS files, themes folder, and the MVC folder structure so far. Though it's a one-time job in a project, it's a tedious one.

You can use the Sencha Cmd tool that you met in Chapter 8, while learning about Theming and Styling, to generate an Ext JS 4 application that creates a complete template for generating the application. In technical circles this is commonly referred to as *Scaffolding*.

Let's generate an Ext JS 4 application using the `generate` command shown below.

```
sencha -sdk PathToSDK generate app NameOfTheApp PathToTheApp
```

Figure 10-9 shows the screenshot of the `generate` command run from the command prompt.


```

C:\>sencha -sdk c:\ext-4.2.0 generate app Chapter10 C:\Chapter10
Sencha Cmd v3.1.0.256
[INF] init-plugin:
[INF]
[INF] init-plugin:
[INF] Invoking plugin (C:\ext-4.2.0\.sencha\workspace\plugin.xml) - support
rgets: -before-generate-workspace
[INF]
[INF] -before-generate-workspace:
[INF] Invoking plugin (C:\ext-4.2.0\.sencha\workspace\plugin.xml) - support
rgets: generate-workspace
[INF]
[INF] cmd-root-plugin.init-properties:
[INF]
[INF] init-properties:
[INF]
[INF] init-sencha-command:
[INF]
[INF] init:
[INF]
[INF] -before-generate-workspace:
[INF]

```

Figure 10-9. *sencha generate command*

As shown in Figure 10-9 we've generated the *Chapter10* application. Figure 10-10 shows the contents of the Chapter10 application.

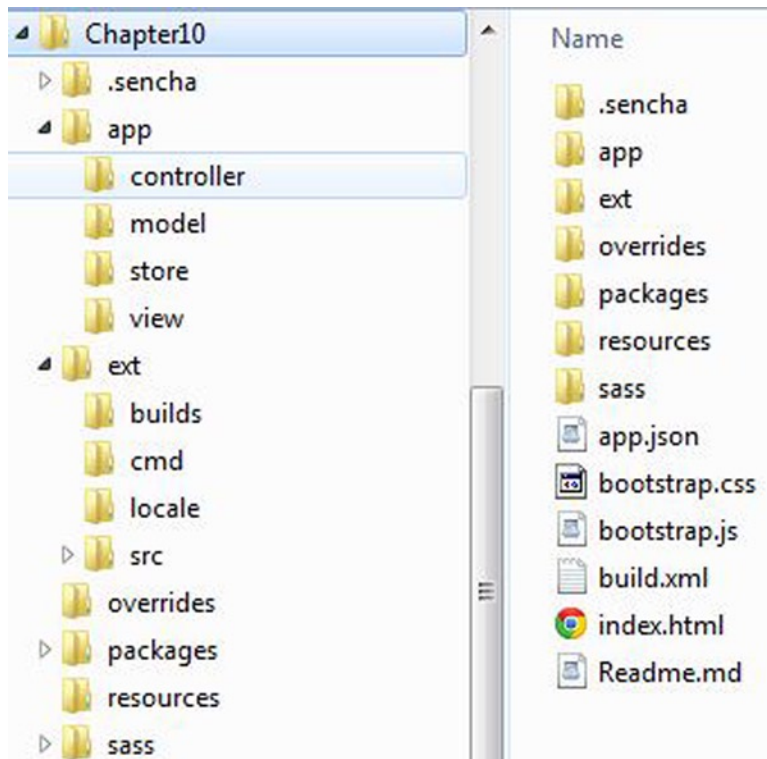


Figure 10-10. *Contents of Chapter10 application*

As shown in Figure 10-10, the Chapter10 seems to be loaded with all the basic folders and files.

- **.sencha** folder contains the project-related configuration files.
- **app** folder has the MVC folder structure. This folder contains the **app.js** file also.
- **ext** folder contains the source files.
- **packages** folder contains the custom packages that you may want to use in your application, like the themes.
- **resources** folder contains the images that you want to use in your application.
- **index.html** file is the executable web page where all the appropriate files have been referenced.

As a developer all you need to do now is load this generated application folder in the IDE of your choice and start building the models, views, controllers, and stores. You can also play with themes and other resources. I loaded this application in Visual Studio, one of my favorite IDEs, as a web application like that shown in Figure 10-11.

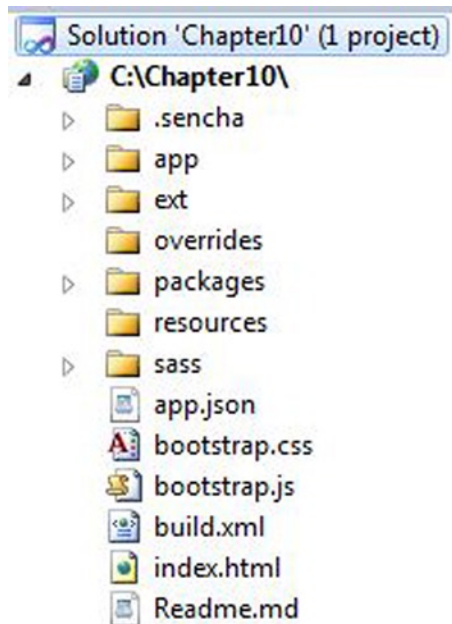


Figure 10-11. Chapter10 application loaded in IDE

I launched the index.html page in a local development web server. Figure 10-12 shows the output of the index.html page.

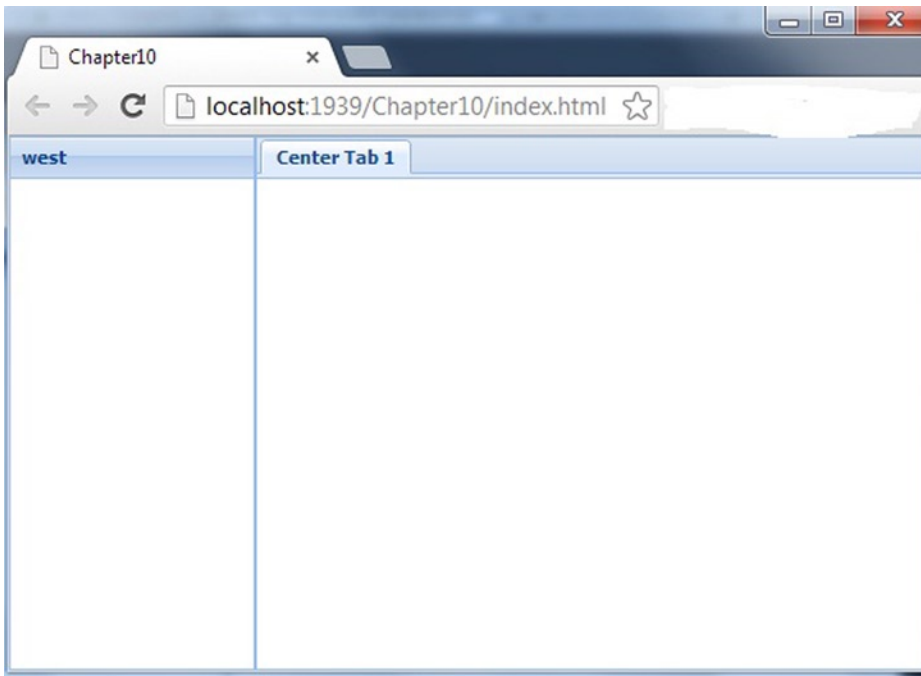


Figure 10-12. Output of *index.html* page

As shown in Figure 10-12, the default view is the `Ext.container.Viewport` instance with border layout. The center region of the Viewport is a tab panel.

You can now modify the code and start building your application. After completing development we need to build it so that it can be deployed to the production web server. Building the application involves tasks like minifying the JavaScript files, generating CSS from SCSS files, finding and including only the source code of those components used in the application, etc. All these tasks can be implemented using the simple build command shown here.

```
sencha build app production
```

You can build the application for a production or a testing environment. Figure 10-13 shows the build command run from the command prompt.

```

C:\Chapter10>sencha app build production
Sencha Cmd v3.1.0.256
[INF] Including theme package ext-theme-classic for app.theme=ext-theme-cl
build
[INF]
[INF]   init-plugin:
[INF]
[INF]   init-plugin:
[INF] Invoking plugin (C:\Chapter10\.sencha\app\plugin.xml) - supported tæ
-before-app-build
[INF]
[INF] -before-app-build:
[INF] Invoking plugin (C:\Chapter10\.sencha\app\plugin.xml) - supported tæ
app-build
[INF]
[INF] cmd-root-plugin.init-properties:
[INF]
[INF]   init-properties:
[INF]
[INF]   init-sencha-command:
[INF]
[INF]   init:
[INF]
[INF] -before-app-build:
[INF]
[INF] app-build-impl:
[INF]
[INF] production:
[INF]
[INF] -before-init-local:
[INF]
[INF]

```

Figure 10-13. *sencha build command*

The build command generates a build folder in your application. The build folder contains the production code that can be copied to the production web server. Figure 10-14 shows the generated build folder.

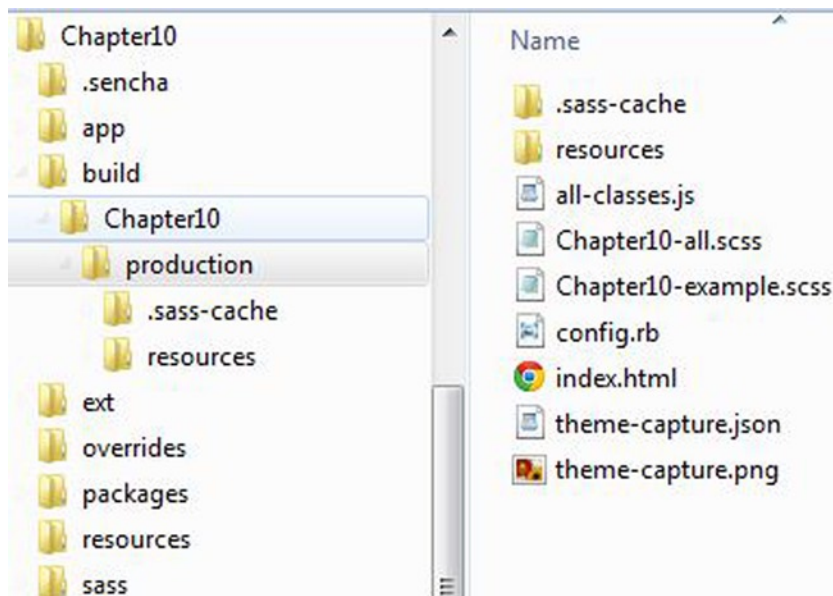


Figure 10-14. *build folder*

As shown in Figure 10-14, the build folder contains *Chapter10/production* folder, which has the files that you can deploy to the application. Ideally you just need to copy the *resources folder*, *index.html*, and *all-classes.js* files to the production server. The *all-classes.js* file contains all the JavaScript code used in the application. This includes the API code that our application needs and the code that we've written. The *all-classes.js* has the JavaScript code in a compressed format, thereby reducing the size. Sencha Cmd uses the YUI (Yahoo User Interface) JavaScript compressor to compress the JavaScript code.

Summary

In this chapter you learned how to extend the Ext JS 4 API by creating custom components and plugins. You can create a custom component by using the `autoEl` attribute to specify the HTML element. If you want to build a complex component, you can use the `XTemplate` using the `tpl` attribute and a `data` attribute for supplying data to the template. Plugins can be developed by inheriting the `Ext.AbstractPlugin` class and overriding the `init()` method.

You can test Ext JS 4 applications using the Jasmine toolkit. You can create specs and specify the expectations on the Ext JS4 application. Ext JS 4 applications can be created from scratch using Sencha cmd tool. You can use the `generate app` command to scaffold the application. Finally you can package the application and build it using a single `sencha build` command.