Ext JS 6 Tutorial

Marko Wirtz

Ext JS 6 Tutorial

By Marko Wirtz

Table of Contents

Disclaimer

Introduction

Each software developer wants to come up with apps which are highly interactive. This is because interactivity is one of the mechanisms by which you can improve the usability of your apps and user friendliness. Ext JS 6 is a JavaScript framework which can be used for developing such apps. The framework is very easy for anyone to learn. For JavaScript experts, you will find it easy for you to learn how to use the framework. This framework forms the center of discussion for this book. The explanations have also been made simple for ease of understanding.

Chapter 1- Upgrading a Sencha Touch App Modern Toolkit of Ext JS 6

Most of you have mobile/tablet apps which you need to upgrade and use in ES 6. In this chapter, you will learn how to do this and the reason why you should do it.

Basic Mobile Upgrade

With this type of upgrade, we will just remain with the basic MVC pattern. The following steps are necessary:

Begin by downloading the trial version of Ext JS 6.

In your Sencha app, look for the namespace which you used. I used the namespace “Din.”

Generate a modern app for Ext JS 6.

Navigate to the folder for “ext” framework, and then use the following command to generate the app:

```
ext> sencha generate app -modern Din ../din1
```

You can then open the project in your browser, and you will observe the demo app for Ext JS 6.

In the file system, rename the folder “<myproject>/app,” to something which looks as “app-backup.”

Do the same for the file “app.js,” and rename it as “app-backup.js.”

You can then copy the folder “app” from the Sencha Touch project, and then paste it into the new Ext JS 6 project.

Run the command given below from the command line:

```
sencha app refresh
```

The above command can give you some build errors due to changes in the API. Just correct these, and then proceed to the next step.

After you realize that you have no errors, just open the app in the browser.

Chapter 2- Ext.data.Model-cfg Validators

An Entity or a Model is used to represent an object which is being managed by your application.

Fields

To define models, this has to be done in the form of a set of fields and any arbitrary properties and methods which are relevant to the model. Consider the example given below:

```
Ext.define('MyUser', {
   extend: 'Ext.data.Model',
   fields: [
   {name: 'name',  type: 'string'},
   {name: 'age',   type: 'int', convert: null},
   {name: 'telephone', type: 'string'},
   {name: 'present', type: 'boolean', defaultValue: true, convert: null}
   ],
   changeName: function() {
   var old= this.get('name'),
   new= old + " The Boss";
```

```
    this.set('name', new);
    }
});
```

At this point, instances of the model “MyUSer” can be created and then call the logic which we have defined. This is shown in the code given below:

```
var user = Ext.create('MyUser', {
    id   : 'ABCD56789',
    name : 'John',
    age  : 25,
    telephone: '665-777-9087'
});

user.changeName();
user.get('name'); //returns “John The Boss”
```

Validators

In Ext JS 6, models usually have a built-in support for the field validators. If you need to add validators to models, use the code given below:

```
Ext.define('User', {
    extend: ‘Ext.data.Model’,
    fields: [
    { name: ‘name’,     type: ‘string’ },
    { name: ‘age’,      type: ‘int’ },
    { name: ‘telephone’,    type: ‘string’ },
    { name: ‘gender’,   type: ‘string’ },
    { name: ‘username’, type: ‘string’ },
    { name: ‘present’,    type: ‘boolean’, defaultValue: true }
    ],
    validators: {
    age: ‘presence’,
    name: { type: ‘length’, min: 2 },
    gender: { type: ‘inclusion’, list: [‘Male’, ‘Female’] },
    username: [
    { type: ‘exclusion’, list: [‘Admin’, ‘Operator’] },
    { type: ‘format’, matcher: /([a-z]+)[0-9]{2,3}/i }
    ]
    }
});
```

To retrieve the results of our validation, we can do it through the associated validation record. This is shown in the code given below:

```
var instance = Ext.create('MyUser', {
    name: 'Joel',
    gender: 'Male',
    username: 'joeljohn'
});

var validation = instance.getValidation();
```

The object which will be returned will be an instance of "Ext.data.Validation," and the result of the field "validators" will be the fields. In case one or more validation errors occur, then the validation object will be termed to be "dirty."

Associations

Models are usually associated with other models. Fields can be used for the purpose of defining these associations or the other data, such as the many-to-many, etc.

Foreign-Key Associations - One-to-Many

A referencing config can be added to the appropriate field for the purpose of defining associations between models. Consider the example given below:

```
Ext.define('MyPost', {
   extend: 'Ext.data.Model',
   fields: [
   { name: 'user_id', reference: 'MyUser' }
   ]
});
Ext.define('Comment', {
   extend: 'Ext.data.Model',
   fields: [
   { name: 'user_id', reference: 'MyUser' },
   { name: 'post_id', reference: 'MyPost' }
   ]
});
Ext.define('User', {
   extend: 'Ext.data.Model',
   fields: [
   'name'
   ]
});
```

The use of “reference” on the appropriate field will tell the model about the field which has the foreign-key and the type of the model which is being identified. This means that the value of the field will be set to the value of the field “idProperty” of our target model.

One-to-Many without Foreign-Keys

If you are in need of defining an association without having a field for “foreign-key,” you will be forced to use either “belongsTo” or “hasMany.” This is shown in the code given below:

```
Ext.define(‘MyPost’, {
    extend: ‘Ext.data.Model’,
    belongsTo: ‘MyUser’
});
Ext.define(‘Comment’, {
    extend: ‘Ext.data.Model’,
    belongsTo: [ ‘MyPost’, ‘MyUser’ ]
});
//The  MyUser is as shown above
```

The declarations shown have changed greatly from the previous releases.

Foreign-Key Associations - One-to-One

This is just a special case of the association “one-to-many.” It is defined as shown below:

```
Ext.define(‘MyAddress’, {
    extend: ‘Ext.data.Model’,
    fields: [
    ‘address’,
    ‘city’,
    ‘state’
    ]
});
Ext.define(‘MyUser’, {
    extend: ‘Ext.data.Model’,
    fields: [{
    name: ‘addressId’,
    reference: ‘Address’,
    unique: true
    }]
});
```

Many-to-Many

This is usually used by a User or in a Group. A user can belong to many groups, and a group is made up of several users. Such an association is defined by use of the association “manyTomany” as shown below:

```
Ext.define(‘MyUser’, {
    extend: ‘Ext.data.Model’,
    fields: [
    ‘name’
    ],
    manyToMany: ‘Group’
});
Ext.define(‘MyGroup’, {
    extend: ‘Ext.data.Model’,
    fields: [
    ‘name’
    ],
    manyToMany: ‘MyUser’
});
```

As in the other types of associations, only one side needs to be defined.

Using a Proxy

Models are very good when it comes to the representation of data types and the relationships, but the data will need to be loaded and then saved somewhere. A proxy is used for the purpose of loading and saving of data, and this can be done directly on a model as shown below:

```
Ext.define('MyUser', {
    extend: 'Ext.data.Model',
    fields: ['id', 'name', 'email'],

    proxy: {
    type: 'rest',
    url : '/users'
    }
});
```

In the above case, we have set a “Rest” proxy, and this is aware of how to load and then save some data to and then from a RESTful backend. This works as shown below:

```
var user = Ext.create('User', {name: 'John Joel', email: 'john@gmail.com'});
```

user.save(); //POST /users

When we call the “save” method on the new instance of our model, this will tell the configured RestProxy that we need to persist the data of the model onto the server. The Restproxy will figure out that the proxy has not been saved before it does not have an ID and then it takes the appropriate action, and in this case, it just sends a “POST” request to the URL which we have just configured.

The static “load” method is used for loading data via the proxy. This is done using the code given below:

```
//Using the configured RestProxy for making a GET request to the /users/123

MyUser.load(567, {

    success: function(user) {

    console.log(user.getId()); //logs 567

    }

});
```

It is also easy for us to update and then destroy easily as shown in the code given below:

```
//the user Model we had loaded in our last snippet:

user.set(‘name’, ‘Joel John’);

//telling the Proxy to save our Model. A PUT request will be performed in this case to
the /users/567 as the Model already has an id
```

```
user.save({
    success: function() {
    console.log('The updating of the user was done');
    }
});
//telling the Proxy to destroy our Model. Performs DELETE request to the /users/123
user.erase({
    success: function() {
    console.log('The User has been destroyed!');
    }
});
```

HTTP parameter names are good whenever we are using an ajax proxy. The default setting is that the id of the model should be set in the HTTP parameter with the name id. For the name of the parameter to be changed, we have to use the “idParam” configuration of the proxy.

Usage in Stores

We will always need to load instances of the model which are to be displayed, and then manipulated in the UI. This can simply be done by creation of a store as shown below:

```
var store = Ext.create(‘Ext.data.store’, {
    model: ‘User’
});
//using the Proxy to set up on the Model for loading the Store data
store.load();
```

A store can be seen as a collection of multiple instances of a model, and these are usually loaded from a particular server. A store can also be used for the maintenance of a set of model instances which are to be synchronized with other servers via a Proxy.

Chapter 3- Controllers

Controllers are used for binding an application together. This means that they just have to listen to events and then take an appropriate action. A controller which can be used for the purpose of user management can be defined as shown below:

```
Ext.define ('OurApp.controller.Users', {
    extend: 'Ext.app.Controller',
    init: function() {
    console.log('Initialized by Users! This will happen before ' +
    'the function launch()  of the Application is called');
    }
});
```

The init function can be seen as a special method which is called once an application has been launched. This is called before the launch function of the application has been executed. This will create an area where you can run your code prior to the creation of the ViewPort.

The control function of the controller will make it easy to listen to the view classes and then take an action with the handler function. Let us update the MyUsers controller so that it can inform us whenever the panel has been rendered. This is shown below:

```
Ext.define (‘OurApp.controller.Users’, {
    extend: ‘Ext.app.controller’,
    control: {
    ‘viewport > panel’: {
    render: ‘onPanelRendered’
    }
    }
    onPanelRendered: function() {
    console.log(‘The panel has been rendered’);
    }
});
```

Event Domains

The ref system in controllers is very critical.

Using refs

The ref system is very important when it comes to the use of controllers. This makes it easy for us to get reference to the views of our page. Consider the example given below:

```
Ext.define('OurApp.controller.Users', {
    extend:'Ext.app.controller',

    refs: [{
    ref: 'list',
    selector: 'grid'
    }],

    control: {
    'button': {
    click: 'refreshGrid'
    }
    },

    refreshGrid: function() {
    this.getList().store.load();
    }
});
```

With the above example, there is a grid on the page, and this has a single button which will be used for the refreshing of the page when the button has been clicked. In the refs array, we will set up a reference to this grid.

Generated Getter Methods

Refs are not the only mechanism which can be used for generation of getter methods which are more convenient. The controllers have to work together with models and stores, and the framework will provide us with a number of ways to get to those. Consider the example given below:

```
Ext.define (‘OurApp.controller.Users’, {
    extend: ‘Ext.app.Controller’,
    models: [‘MyUser’],
    stores: [‘AllUsers’, ‘AdminUsers’],
    init: function() {
    var MyUser, allUsers, ed;

    MyUser = this.getUserModel();
    allUsers = this.getAllUsersStore();
    ed = new MyUser({ name: ‘Ed’ });
    allUsers.add(ed);
    }

});
```

If you specify the models and the stores which the controller will be caring for, they will be dynamically loaded from their correct locations. The getter methods for all of these will also be created. In the above example, a new instance of User Model will be created and then added to the store named “AllUsers.” Of course, you could have done something more with it, but this one was just for a demonstration purpose.

Chapter 4- Ext.Anim

This is used for the purpose of executing simple apps which are defined in the library Ext.anims. The run method can be used for running the properties given below:

```
Ext.Anim.run (this, ‘fade’, {
    out: false,
    autoClear: true
});
```

Whenever you are using the above method, it is good for you to ensure that you require the “Ext.anim.” This is shown in the code given below:

```
Ext.requires(‘Ext.Anim’);
```

The Ext.setup can be used as shown below:

```
Ext.setup({
    requires: [‘Ext.Anim’],
    onReady: function() {
    //Add something here
    }
});
```

When using Ext.application, this can be implemented as shown below:

```
Ext.application ({
    requires: [‘Ext.Anim’],
    launch: function() {
    //Add something here
    }
});
```

Consider the code given below, which shows a function which takes two arguments can be created:

```
// This example can be executed directly on your JavaScript console
// Creating a function which will take two arguments and then return the //sum of them
arguments
var add = new Function(“x”, “y”, “return x + y”);
// Calling the function
adder(4, 5);
// > 9
```

Chapter 5- Arrays

The property “Array” is a constructor for instances of an array. An array is an object in JavaScript.

Creating an Array

Consider the example given below, which shows how an array can be created in Ext JS 6:

```
var myArray = new Array();

myArray[0] = “Hello”;

myArray[99] = “there”;

if (myArray.length == 100)

print(“The array ahs a length of 100.”);
```

In the above example, we have created an array named “myArray,” and we have set its length to be 100, that is, from 0 to 99.

Two-dimensional Arrays

Consider the example given below, which shows how a two-dimensional array can be created:

```
var myBoard =

[ [‘J’,‘k’,‘B’,‘Q’,‘K’,‘M’,‘N’,‘K’],

[‘J’,‘J’,‘J’,‘J’,‘J’,‘J’,‘J’,‘J’],

[‘ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘],

[‘ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘],

[‘ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘],

[‘ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘,’ ‘],

[‘j’,‘j’,‘j’,‘j’,‘j’,‘j’,‘j’,‘j’],

[‘j’,‘k’,‘b’,‘q’,‘k’,‘m’,‘n’,‘k’]];

print(myBoard.join(‘\n’) + ‘\n\n’);

myBoard[4][4] = myBoard[6][4];

myBoard[6][4] = ‘ ‘;

print(myBoard.join(‘\n’));
```

Once you run the above code, you will get the following as the output:

J, k, B, Q, K, M, N,K

J, J, J, J, J, J, J, J

, , , , , , , ,

, , , , , , , ,

, , , , , , , ,

, , , , , , , ,

j, j , j, j, j, j, j, j

j, k, b, q, k, m, n, k

Accessing Elements of an Array

The elements of an array are nothing, but just the object properties, so they have to be accessed in the same manner. Consider the code given below which shows how this happens:

var array = new myArray(“John”, “Mercy”, “Donald”);

array[0]; // “John”

array[1]; // “Mercy”

// etc.

array.length; // 3

// Even though indices are a kind of properties, the below notationwill throw a syntax error

```
array.2;

// You should note that in JavaScript, names for an object property are strings. Consequently,

array[0] === array[“0”];

array[1] === array[“1”];

// etc.

// However, this has to be considered carefully

array[02]; // “Fire”. The number 02 will be converted as the “2” string

array[“02”]; // undefined. We have no property named “02”
```

Relationship between Numerical properties and Length

The length property of an array and its numerical properties are connected. The code given below shows how this relationship works:

```
var x = [];

x[0] = ‘x’;

console.log(x[0]); // ‘x’

console.log(x.length); // 1

x[1] = 32;

console.log(x[1]);

console.log(x.length);

x[13] = 56789;
```

```
console.log(x[13]); // 56789

console.log(x.length);

x.length = 10;

console.log(x[13]); // undefined, we are reducing the length of elements after the length+1 has been removed

console.log(x.length);
```

The match we get between a string and a regular expression can be used for making an array. This array usually has elements and properties which can provide us with information about the match. Consider the example given below, which shows how this can be done:

```
// Matching one of the j’s followed by either one or more k’s followed by one j

// Remember matched j’s and the following k

// Ignoring the case

var ch = /j(k+)(j)/i;

var array = ch.exec(“jkfgjkjsgd”);
```

Chapter 6- Working with Dates

The “dates” instance can be created for working with dates and times in Ext JS 6.

In case you do not specify any arguments, the constructor will then create a “date” object for the date of the day and the time depending on your local time. In case some arguments are supplied and then others are left out, the missing arguments will then be set to 0. When supplying the arguments, you have to at least specify the year, the month, and the date. The hours, minutes, seconds, and milliseconds can be omitted if need be.

The following are some of the ways that dates can be assigned:

```
day = new Date();

birthdate = new Date(“April 20, 1992 03:24:00”);

birthdate = new Date(1992,04,20);

birthdate = new Date(1990,04,20,3,24,0);
```

Some time might have elapsed between the dates which you specify. Consider the example given below, which shows how this can be done:

```
// using static methods

var start = Date.now();

// the event which you would like to time should be added here:

doSomething ();
```

```
var end = Date.now();

var elapsedTime = end - start; // time in terms of milliseconds

// if you are having the Date objects

var start = new Date();

// the event you would like to time should go here:

doSomething ();

var end = new Date();

var elapsedTime = end.getTime() - start.getTime(); // time in terms of //milliseconds
// if you are in need of testing a function and getting back to its return

function printElapsedTime (fTest) {

    var nStartTime = Date.now(), vReturn = fTest(), nEndTime = Date.now();
    alert(“Elapsed time is: ” + String(nEndTime - nStartTime) + “
    milliseconds”);

    return vReturn;

}

functionReturn = printElapsedTime(function);
```

The code given below shows dates formatted as per ISO 8601:

```
// How to use a function for getting the exact format you desire…

function ISODateString(j){

function pad(n){return n<10 ? ‘0’+n : n}

return j.getUTCFullYear()+’-‘

    + pad(j.getUTCMonth()+1)+’-‘

    + pad(d.getUTCDate())+‘T’
    + pad(j.getUTCHours())+’:’

    + pad(j.getUTCMinutes())+’:’

    + pad(j.getUTCSeconds())+‘Z’}


var j = new Date();

print(ISODateString(j)); // will print something such as 2016-02-28T20:03:12Z
```

Chapter 7- String

A string is a global object which may be used for constructing instances of strings. For us to create string objects, we can call the constructor “new String().” The String object will wrap the string primitive data type for JavaScript using the methods which are described below. If you need to create a primitive string, you can also call the Global object “String()” without using the object “new” at its front. In JavaScript, String laterals are primitive strings.

JavaScript will automatically convert between String objects and String primitives. JavaScript will automatically convert your String primitive to get a temporary string object, referred to as a method, then the temporary String object is discarded. Consider the example given below which shows how this can be done:

string_obj = new String(s_prim = s_also_prim = “example”);

strung_obj.length;

string_prim.length;

string_also_prim.length;

‘example’.length;

“example”.length;

You have to note that a string literal has to be denoted using single or double quotation marks. The method “valueOf” can be used for conversion of string objects to primitive types. When these have been evaluated in JavaScript, both the String objects and the String primitives will give different results. The Primitives have to be treated as a source code, whereas the string objects have to be treated as character sequence object. This is shown in the code given below:

```
string1 = “2 + 2”;              // creating a string primitive
string2 = new String(“2 + 2”);   // creating a String object
eval(string1);                  // Will return the number 4
eval(string2);                  // will return the string “2 + 2”
eval(string2.valueOf());        // will return the number 4
```

Character Access

Sometimes, you may need to access a particular character contained in a string. There are two ways that this can be done. The first method involves the use of “charArt” which is shown below:

```
return ‘cow’.charAt(1); // returns “o”
```

In the second method, the string has to be treated as an array. Each character in the string corresponds to an index. This is shown in the code given below:

```
return ‘cow’[1]; // returns “o”
```

Sometimes, we might also need to compare strings. In ExtJS, we make use of the “less than” and “greater than” so as to compare our strings. Compare the example given below:

```
var x = “x”;
var y = “y”;
if (x < y) // true
    print(x + ” is less than ” + y);
else if (x > y)
    print(x + ” is greater than ” + y);
else
    print(x + ” and ” + y + ” are equal.”);
```

Chapter 8- Data Object Manipulation

The class DomHelper usually provides us with a layer of abstraction from the DOM and enables us to create elements via DOM or by use of HTML fragments. Consider the code given below which shows the common insertion methods which are used:

```
var dh = Ext.DomHelper; // creating the shorthand alias
// specification of an object
var specification = {
    id: ‘my-ul’,
    tag: ‘ul’,
    cls: ‘my-list’,
    // append the children after they have been created
    children: [     // a ‘cn’ child can also be specified other than ‘children’
    {tag: ‘li’, id: ‘item0’, html: ‘List Item 0’},
    {tag: ‘li’, id: ‘item1’, html: ‘List Item 1’},
    {tag: ‘li’, id: ‘item2’, html: ‘List Item 2’}
    ]
};
var l = dh.append(
    ‘my-div’, // the context element ‘my-div’ may either be the id or our actual node
    specification      // the specification object
);
```

The parameters for element creation specification can also be passed as an array of specification objects. This can be used for the purpose of insertion of multiple nodes for siblings into a particular container very efficiently. Consider the example given below, which shows how more list items can be added to the example given above:

```
dh.append(‘my-ul’, [
    {tag: ‘li’, id: ‘item3’, html: ‘List Item 3’},
    {tag: ‘li’, id: ‘item4’, html: ‘List Item 4’}
]);
```

Templating

Templating is a very powerful feature. The above example can be implemented with templating as shown below:

```
// creating the node
var l = dh.append(‘my-div’, {tag: ‘ul’, cls: ‘my-list’});
// getting the template
var tpl = dh.createTemplate({tag: ‘li’, id: ‘item{0}’, html: ‘List Item {0}’});
```

```
for(var j = 0; j < 5; j++){
    tpl.append(l, j); // using the template for appending to the actual node
}
```

With the use of a template, this can be implemented as shown below:

```
var html = '"{0}" href="{1}" class="nav">{2}';

var tpl = new Ext.DomHelper.createTemplate(html);

tpl.append('blog-roll', ['link1', 'http://www.mysite.com/', "My Site"]);
tpl.append('blog-roll', ['link2', 'http://www.mysite2.org/', "Mysite2 Site"]);
```

When using named parameters, the above example can be implemented as shown below:

```
var html = '"{id}" href="{url}" class="nav">{text}';

var tpl = new Ext.DomHelper.createTemplate (html);

tpl.append('blog-roll', {

    id: 'link1',

    url: 'http://www.mysite.org/',

    text: " MySite site"

});

tpl.append('blog-roll', {

    id: 'link2',

    url: 'http://www.mysite2.com/',
    text: “Mysite2 Site”
});
```

To apply templates, we use regular expressions. The performance of this will be good, but this can still be improved by compilation of the template. Consider the code given below, which shows how this can be done:

```
var html = ’”{id}” href=”{url}” class=“nav”>{text}’;
var tpl = new Ext.DomHelper.createTemplate (html);
tpl.compile();
// … using the template as usual
```

Chapter 9- Class System

Ext JS 6 comes with a number of classes. The names for classes should contain only alphanumeric characters. In the case of numbers, they can be used, but these should only be used for a technical purpose. Classes contain a number of components including properties and methods.

A single method can be used when we are creating classes. Consider the example given below, which shows how this can be done:

```
Ext.define(‘My.example.Human’, {
    name: ‘Unknown’,
    constructor: function(name) {
    if (name) {
    this.name = name;
    }
    },
    play: function(gameType) {
    alert(this.name + ” is playing: ” + gameType);
    }
});
var john = Ext.create(‘My.example.Human’, ‘John’);
```

```
bob.play(“Soccer”); // alert(“John is playing: Soccer”);
```

Configuration

Note that configurations have to be encapsulated from the rest of the class members. The setter and getter methods for each config property will be generated automatically into the prototype of the class during the creation of the class and if the methods have not been defined readily.

Consider the configuration example given below:

```
Ext.define(‘My.own.Window’, {
   extend: ‘Ext.Component’,
   /** @readonly */
   isWindow: true,
   config: {
   title: ‘Add the Title Here’,
   bottomBar: {
   height: 50,
   resizable: false
   }
   },
   applyTitle: function(yourTitle) {

   if (!Ext.isString(yourTitle) || title.length === 0) {

   alert('Error: The Title has to be a valid and non-empty string');

   }

   else {

   return yourTitle;

   }

   },

   applyBottomBar: function(bottomBar) {

   if (bottomBar) {

   if (!this.bottomBar) {

   return Ext.create('My.own.WindowBottomBar', bottomBar);

   }

   else {

   this.bottomBar.setConfig(bottomBar);

   }

   }

   }

});

/** The child component to be completed in the example. */

Ext.define('My.own.WindowBottomBar', {

   config: {
```

```
    height: undefined,

    resizable: true

    }

});
```

As shown in the above code, the “config” property gets processed by the “Ext.” The above configuration can be used as shown below:

```
var window = Ext.create(‘My.own.Window’, {

    title: ‘Hello there!’,

    bottomBar: {

    height: 60

    }

});

alert(window.getTitle()); // alerts “Hello there”

window.setTitle(‘Something New’);

alert(window.getTitle()); // alerts “Something New”

window.setTitle(null); // will alert “Error: Title must be a valid non-empty string”

window.setBottomBar({ height: 100 });

alert(window.getBottomBar().getHeight()); // alerts 100
```

Statistics

To define static members, we can use the config “statics.” This is shown in the code given below:

```
Ext.define('Computer', {

    statics: {

    instanceCount: 0,

    factory: function(brand) {

    // 'this' in the static methods refers to the class itself

    return new this({brand: brand});

    }

    },

    config: {

    brand: null

    }

});

var hpComputer = Computer.factory('HP');

var lenovoComputer = Computer.factory('Lenovo');

alert(lenovoComputer.getBrand()); // use of the auto-generated getter for getting the
value of the config property. Alerts "Lenovo"
```

Chapter 10- Layouts and Containers

Ext JS 6 has a very powerful layout system. It dictates how we position and size the components we use in our app.

Containers

Any app in Ext JS is made up of a number of components. A container is just a special type of component which can have several other components in it. A Panel represents the component which is commonly used. The container given below is an illustration of how a panel can be used for adding other components to it:

```
Ext.create('Ext.panel.Panel', {
    renderTo: Ext.getBody(),
    width: 400,
    height: 300,
    title: 'A Container Panel',
    items: [
    {
    xtype: 'panel',
    title: '1st Child Panel',
    height: 100,

    width: ‘75%’

    },

    {

    xtype: ‘panel’,

    title: ‘2nd Child Panel ‘,

    height: 100,

    width: ‘75%’

    }

    ]

});
```

Layouts

A layout is used for determining the positioning and sizing of the container components. In the example we gave above, the layout of the panel has not been specified. The auto layout is the default layout used for containers. Consider the code given below:

```
Ext.create(‘Ext.panel.Panel’, {

    renderTo: Ext.getBody(),

    width: 400,

    height: 200,
    title: ‘A Container Panel’,

    layout: ‘column’,

    items: [

    {

    xtype: ‘panel’,

    title: ‘!st Child Panel’,

    height: 100,

    columnWidth: 0.5

    },

    {

    xtype: ‘panel’,

    title: ‘2nd Child Panel’,

    height: 100,

    columnWidth: 0.5

    }

    ]

});
```

In Ext JS, there are full sets of layouts, and one can use any layout that they need to use.

How Layout System Works

Note that when you resize a container, the re-layout is triggered, or after addition or removal of child components. Consider the code given below, which best illustrates this:

```
var contPanel = Ext.create('Ext.panel.Panel', {
    renderTo: Ext.getBody(),
    width: 400,
    height: 200,
    title: 'A Container Panel',
    layout: 'column',
    suspendLayout: true // will suspend the automatic layouts as we do several things which could trigger a layout on our own
});
// Adding a couple of child items.  Both of these can be added at the same time when we pass an array to add(),
// but let us pretend that we need to add them separately for a reason.
containerPanel.add({
    xtype: 'panel',
    title: '1st Child Panel ',
    height: 100,
    columnWidth: 0.5
});
containerPanel.add({
    xtype: ‘panel’,

    title: ‘2nd Child Panel ‘,

    height: 100,

    columnWidth: 0.5

});

// Turning our suspendLayout flag off.

containerPanel.suspendLayout = false;

// Triggering a layout.

containerPanel.updateLayout();
```

Chapter 11- Components

The UI of an Ext JS app is made up of a number of widgets which are referred to as components. All of these classes have to be derived from the class “Axt.Components,”and this class makes it easy for us to instantiate, create, and manage these widgets.

Consider the code given below, which shows how instantiation of components can be done:

```
var cPanel1 = Ext.create(‘Ext.panel.Panel’, {
    title: ‘1st Child Panel ‘,
    html: ‘A Panel’
});
var cPanel2 = Ext.create(‘Ext.panel.Panel’, {
    title: ‘2nd Child Panel’,
    html: ‘Another Panel’
});
Ext.create(‘Ext.container.Viewport’, {
    items: [ cPanel1, cPanel2 ]
});
```

The containers make use of child managers for the purpose of position and sizing of the components.

XTypes and Lazy Instantiation

Each component must have a symbolic name which is referred to as an “xType.” However, the instantiation of components has to be done depending on how the app needs to be used.

The code given below demonstrates lazy instantiation and how child components of a container can be rendered by use of a Tab Panel. Each tab has an event listener, and this will alert us once a tab has been rendered:

```
Ext.create(‘Ext.tab.Panel’, {
   renderTo: Ext.getBody(),
   height: 100,
   width: 200,
   items: [
   {
   // defining the xtype of the Component configuration explicitly.
   // This will tell the Container (which is the tab panel in our case)
   // to instantiate the Ext.panel.Panel if it deems necessary
   xtype: ‘panel’,
   title: ‘Tab 1’,
   html: ‘First tab’,
   listeners: {
   render: function() {
   Ext.MessageBox.alert(‘Rendered 1’, ‘Tab 1 has been rendered.’);
   }
   }
   },
   {
   // xtype for all the Component configurations in the Container
   title: ‘Tab 2’,
   html: ‘The 2 tab’,
   listeners: {
   render: function() {
   Ext.MessageBox.alert(‘Rendered 1’, ‘Tab 2 has been rendered.’);
   }
   }
   }
   ]
   });
```

You can then run the above code, and you will get an alert message that Tab 1 has been rendered. This is because it forms our default tab, and its Container Tab Panel will instantiate and render it immediately.

If you need to get the alert for the second tab, you must first click on it.

Showing and Hiding

Each component in Ext JS 6 is associated with hide and show methods. In this, we use CSS so as to hide the component. The default CSS for doing this is “display:none.” However, if you don’t need to use this, change it by use of the configuration “hideMode.” The code given below demonstrates how this can be done:

```
var panel = Ext.create(‘Ext.panel.Panel’, {
    renderTo: Ext.getBody(),
    title: ‘Test’,
    html: ‘A Test Panel’,
    hideMode: ‘visibility’ // using the CSS visibility property for showing and hiding this
component
    });
    panel.hide(); // hiding the component
    panel.show(); // showing the component
```

Floating Components

The floating Component is placed outside the flow of document by use of the CSS absolute positioning, and this does not participate in the layout of the container. Some components are considered to be floating by default, and an example of such a component is the Windows. However, it is possible for us to make any component to be a floating component by use of the “floating” configuration. This is shown in the code given below:

```
var panel = Ext.create(‘Ext.panel.Panel’, {
    width: 200,
    height: 100,
    floating: true, // making the panel an absolutely-positioned floating component
    title: ‘Test’,
    html: ‘A Test Panel’
});
```

In the above code, we have instantiated a panel, but we have not rendered it. Under normal circumstances, a component should have a “renderTo” configuration already specified, or it can be added as a component of the container, but when it comes to the case of floating components, none of these methods is needed. In this case, we use the “show” method so as to render our component to the users. The example given below shows how this can be done:

```
panel.show(); // rendering and showing the floating panel
```

You should take note of the following components which are related to floating point components:

draggable()- this will allow for dragging of a floating point component around your screen.

shadow- this will customize the look of the shadow of a floating point component.

alignTo()- this will align a floating component into a specific element.

Center()- this will center a floating component at its center.

Creating Custom Components

Composition or Extension

When you want to create a new UI class, your decision should be whether the class is to create an instance of the component or just extend that component.

In this case, we recommend that you extend the nearest base class to the functionality which you require. This is because Ext JS provides us with an automated lifecycle management and this takes care of automatic rendering when it is needed, automatic sizing, and positioning of components when they are managed by an appropriate manager and automatic removal and destruction from the container.

Subclassing

With the class system in Ext JS, extension of any part of the framework has been made easy. Consider the code given below, which shows how a subclass can be created:

```
Ext.define(‘My.custom.Component’, {
   extend: ‘Ext.Component’,
   newMethod : function() {
   //…
   }
});
```

The above example will create a new class named “My.custom.Component,” and this will inherit the functionalities of the class “Ext.Component” and other properties or methods which are defined.

Template Methods

Consider the code given below:

```
Ext.define(‘My.custom.Component’, {
    extend: ‘Ext.Component’,
    onRender: function() {
    this.callParent(arguments); // calling the superclass “onRender” method

    // performing extra rendering tasks here.
    }
});
```

In the above code, we have created a Component Class for implementing the “onRender” method.

Which Class to Extend

When choosing the kind of class to extend, you have to consider the aspect of efficiency and the capabilities which the base class has to provide. The following are some of the capabilities of the panel:

Header

Footer

Border

Header tools

Footer buttons

Top toolbar

Bottom toolbar

Component

If the UI component we have does not need to contain some other components, meaning that it just needs to encapsulate the HTML which can perform requirements, then you have to extend "Ext.Component." Consider the example given below which wraps an image element of HTML and then it allows us to get and set the "src" element of the image attribute. A "load" event will also be fired when the image has been loaded. Here is the code for the example:

```
Ext.define(‘Ext.ux.Image’, {

   extend: ‘Ext.Component’, // subclass Ext.Component

   alias: ‘widget.managedimage’, // the component having an xtype of the ‘managedimage’

   autoEl: {

   tag: ‘img’,

   src: Ext.BLANK_IMAGE_URL,

   cls: ‘my-managed-image’

   },

   // Adding a custom processing to our onRender phase.

   // Adding a ‘load’ listener to our element.

   onRender: function() {

   this.autoEl = Ext.apply({}, this.initialConfig, this.autoEl);

   this.callParent(arguments);

   this.el.on(‘load’, this.onLoad, this);

   },

   onLoad: function() {

   this.fireEvent(‘load’, this);

   },

   setSrc: function(src) {

   if (this.rendered) {

   this.el.dom.src = src;
```

```
    } else {

    this.src = src;

    }

    },

    getSrc: function(src) {

    return this.el.dom.src || this.src;

    }

});
```

The code given below shows how the above class can be used:

```
var image = Ext.create(‘Ext.ux.Image’);

Ext.create(‘Ext.panel.Panel’, {

    title: ‘An Image Panel’,

    height: 200,

    renderTo: Ext.getBody(),

    items: [ image ]

});

image.on(‘load’, function() {

    console.log(‘image loaded: ‘, image.getSrc());

});

image.setSrc(‘http://www.mysite.com/img/photo-large.png’);
```

That is it. However, note that the class has only been used for demonstration purposes.

Chapter 12- Data Package

The data package is used for loading and saving data for the app. It is made up of the following three main classes:

Ext.data.Model

Store

Ext.data.proxy.Proxy

Models

In any application, a model is used for representation of an entity.

Creating a Model

When defining our models, it is always good that we begin with a base class which is common. With that, it will be easy for you to configure specific aspects of the models at a single place. Consider the code given below which shows how this can be done:

```
Ext.define(‘OurApp.model.Base’, {
    extend: ‘Ext.data.Model’,
    fields: [{
    name: ‘id’,
    type: ‘int’

    }],

    schema: {

    namespace: ‘OurApp.model’,  // generating the auto entityName

    proxy: {     // Ext.util.ObjectTemplate

    type: ‘ajax’,

    url: ‘{entName}.json’,

    reader: {

    type: ‘json’,

    rootProperty: ‘{entName:lowercase}’

    }

    }

    }

});
```

Schema

A schema represents entities which are associated with one another. When a model has defined a schema, then that schema will be inherited by all models which you derive.

Consider the example given below:

```
{
 “success”: “true”,
 “user”: [
   {
   “id”: 1,
   “name”: “John Joel”
   },
   {
   “id”: 2,
   “name”: “Mercy Donald”
   },
   {
   “id”: 3,
   “name”: “Gideon Boss”
   },
   {
   “id”: 4,
   “name”: “Jane Nicholas”
   }
 ]
}
```

Stores

Models are used together with stores, and this is just a collection of records. Consider the code given below, which demonstrates how a store can be created and its data loaded:

```
var store = new Ext.data.Store ({
    model: ‘OurApp.model.User’
});
store.load({
    callback:function(){
    var fname = this.first().get(‘name’);
    console.log(fname);
    }
});
```

Inline data

With stores, data can also be loaded inline. Internally, a store will convert each object which we pass in as the data into models of appropriate types of the model. This is shown in the code given below:

```
new Ext.data.Store({
    model: ‘OurApp.model.User’,
    data: [{
    id: 1,
    name: “John Joel”
    },{
    id: 2,
    name: “Mercy Donald”
    },{
    id: 3,
    name: “Jane Nicholas”
    },{
    id: 4,
    name: “Robert John”
    }]
});
```

Sorting and Grouping

With stores, sorting can be performed grouping and filtering both remotely and locally. The code given below shows how this can be done:

```
new Ext.data.Store({
    model: ‘OurApp.model.User’,
    sorters: [‘name’,‘id’],
    filters: {
    property: ‘name’,
    value   : ‘John Joel’
    }
});
```

In the above example, the data will first be sorted by data and then by id.

Associations

The Associations API can be used for linking models together. Most applications have different models, but these are always related. This leads to creation of relationships. Consider the code given below, which shows how associations are used:

```
Ext.define(‘OurApp.model.User’, {
    extend: ‘OurApp.model.Base’,
    fields: [{
    name: ‘name’,
    type: ‘string’
    }]
});
Ext.define(‘OurApp.model.Post’, {
    extend: ‘OurApp.model.Base’,
    fields: [{
    name: ‘userId’,
    reference: ‘User’, // the entName for OurApp.model.User
    type: ‘int’
    }, {
    name: ‘title’,
    type: ‘string’
    }]
});
```

Rich model relationships can easily be expressed in a particular application. A given model can have as many relationships as possible with other models. Also, the models can be defined in any order. Consider the code given below, which best demonstrates this:

```
// Loading User with an ID of 1 and the related posts and comments
// using the User’s Proxy
OurApp.model.User.load(1, {
    callback: function(user) {
    console.log('User: ' + user.get('name'));
    user.posts(function(posts){
    posts.each(function(p) {
    console.log('Post: ' + p.get('title'));
    });
    });
    }
});
```

With the above association, a new function will be added to the model. Each model of the user has numerous Posts, and these have added the function “user.posts()” which we used. When we call the function “user.posts(),” we will return a store which has been configured with a Post model. Associations are not just good when it comes to loading of data. They are good when it comes to creation of new records. Consider the example given below, which shows how this can be done:

```
user.posts().add({
    userId: 1,
    title: 'Post 4'
});
user.posts().sync();
```

});

The above will instantiate a new Post, and this will then be given the id of the user automatically in the userId field.

The inverse of our association will also generate new methods on our Post model. This is shown in the code given below:

```
OurApp.model.Post.load(1, {
   callback: function(p) {
   p.getUser(function(user) {
   console.log('We got the user from post: ' + user.get('name'));
   });
   }
});
OurApp.model.Post.load(2, {
   callback: function(p) {
   p.setUser(100);
   }
});
```

Loading Nested Data

Once the definition of associations has been done, the process of loading a record can also be used for loading records which are associated in just a single request. Consider the server response which is shown below:

```
{
   “success”: true,
   “user”: [{
   “id”: 1,
   “name”: “John Joel”,
   “posts”: [{
   “title”: “1st Post “
   },{
   “title”: “2nd Post “
   },{
   “title”: “3rd Post “
   }]
   }]
}
```

The framework is able to parse out nested data automatically in a single response as shown above. Instead of us having to make a request for the Posts data and another one for the User data, all of the data can be returned in a single server response.

Validations

Models provide us with a great way that we can be able to validate data. We need to demonstrate how this can be done. We are in need of adding some validations to the model we had recently created recently. This is shown in the code given below:

```
Ext.define('OurApp.model.User', {
    extend: 'Ext.data.Model',
    fields: ...,
    validators: {
    name: [
    'presence',
    { type: 'length', min: 7 },
    { type: 'exclusion', list: ['Bender'] }
    ]
    }
});
```

In most cases, validators are used for the purpose of governing the kind of data which we enter in a particular field of a form.

Consider the example given below, which shows how different validations can be done against a particular user:

```
// now let us try to create a new user having as many validation

// errors as possible

var nUser = new OurApp.model.User({

    id: 10,

    name: ‘John’

});

// running some validation on our new user we have created

console.log(‘Is the User valid?’, nUser.isValid());

//will return ‘false’ as there are validation errors

var errs = nUser.getValidation(),

    err  = errs.get(‘name’);

console.log(“The Error is: ” + err);
```

The function “getValidation()” is the key function in our above program, as it will run all of the validations which have been configured and then return all the errors which have resulted from the fields which have been configured.

Consider the code given below, which shows how the field “name” can be validated in terms of the number of characters which it can have:

```
nUser.set(‘name’, ‘Joel Mercy John’);

errors = nUser.getValidation();
```

The user record will then fulfill all of the needed validations.

Chapter 13- Events

In Ext JS 6, the components and the classes will have to fire events at a particular point in their lifecycle. These allow the app to respond to any changes made in the code.

For events to be executed, we have to implement listeners for the purpose of listening to the events. Consider the code given below, which shows how this can be done:

```
Ext.create('Ext.Panel', {
   html: 'A Panel',
   renderTo: Ext.getBody(),
   listeners: {
   afterrender: function() {
   Ext.Msg.alert('It has been rendered');
   }
   }
});
```

With the example given above, once the preview button has been clicked, the panel will be rendered on the screen, and the alert message will then be shown. All the events which are fired by a particular class will be listed in the API page of the class.

Listening to Events

Consider the example given below:

```
Ext.create('Ext.Button', {
    text: 'Click Here',
    renderTo: Ext.getBody(),
    listeners: {
    click: function() {
    Ext.Msg.alert('You clicked me!');
    }
    }
});
```

A single component may have as many event listeners as possible. In the above example, we have called the function "this.hide()" so as to hide the button onMouseOver. The button will then be displayed later after a second, after calling the function "this.hide()," which fires the "hide" event.

The code should be as given below:

```
Ext.create('Ext.Button', {
    renderTo: Ext.getBody(),

    text: ‘The Button’,

    listeners: {

    mouseover: function() {

    this.hide();

    },

    hide: function() {

    // Waits for 1 second (1000ms), and then shows our button again

    Ext.defer(function() {

    this.show();

    }, 1000, this);

    }

    }

});
```

The event listeners will have to be called each time an event has fired, so the hiding and showing of the button can be done as many times as one may need.

Adding Listeners Later

In our previous examples, we had passed listeners to our component when our class had been instantiated. However, the “on” function can be used for adding listeners when we already have an instance.

Consider the code given below:

```
var btn = Ext.create(‘Ext.Button’, {
    renderTo: Ext.getBody(),
    text: ‘The Button’
});
btn.on(‘click’, function() {
    Ext.Msg.alert(‘Event listener has been attached by .on’);
});
```

The “.on” method can also be used for specifying multiple listeners, which is similar to using multiple configurations. The previous example we had can be modified as follows:

```
var butn = Ext.create(‘Ext.Button’, {
    renderTo: Ext.getBody(),
    text: ‘The Button’
});


btn.on({
    mouseover: function() {
    this.hide();
    },
    hide: function() {
    Ext.defer(function() {
    this.show();
    }, 1000, this);
    }
});
```

Removing Listeners

Just as you know that listeners can be added at any time, they can also be removed at any time. This is done using the “un” function. For us to remove a listener, we have to reference the function itself.

Consider the example given below:

```
var doIt = function() {

    Ext.Msg.alert(‘listener has been called’);

};

var btn = Ext.create(‘Ext.Button’, {

    renderTo: Ext.getBody(),

    text: ‘The Button’,

    listeners: {

    click: doIt,

    }

});

Ext.defer(function() {

    button.un(‘click’, doIt);

}, 3000);
```

In the above code, we have a function which we have then linked to the variable “doIt,” and this variable has the custom function.

Scope Listener Option

The scope will set the value of this inside the handler function. This will by default set to the instance of the class which is firing our event. Consider the code given below:

```
var panel = Ext.create(‘Ext.Panel’, {
    html: ‘A Panel HTML’
});

var btn = Ext.create(‘Ext.Button’, {
    renderTo: Ext.getBody(),
    text: ‘Click Here’
});
btn.on({
    click: {
    scope: panel,
    fn: function() {
    Ext.Msg.alert(this.getXType());
    }
    }
});
```

Conclusion

We have come to the conclusion of this guide. Ext JS 6 has recently brought numerous changes to the Ext JS framework. If you are in need of developing apps with a high degree of interactivity, this is the best JavaScript framework for you to use. The framework can run on a variety of different platforms, showing how flexible it is. The framework provides us with a number of components which we can add to our apps interface. This means that we can add numerous components to the user interface part of our app and come up with a very amazing app. In the case of forms, they should be very well validated so that we ensure that the data we get from users is the correct one. Ext JS 6 supports validation of such fields so that we get the right data from users. Event handling is very important in any app. You should first listen to the occurrence of the event and then take the appropriate action. Classes are very good for the purpose of grouping parts of code; therefore, you should learn how to use these in your Ext JS 6 app. My hope is that this book has greatly helped you learn how to use Ext JS 6 for apps development.