14
Finishing our Application

In this chapter, we are going to learn about how to prepare our applications to go live on a production environment. We will learn how to use the Sencha Command tool to compress and obfuscate our own code; this will allow us to speed things up when downloading all the needed code and classes for our application.

We will create packages for every module and instead of including a file for each class, we will include only one file with all our classes.

The second part of the chapter describes and shows useful plugins and extensions created by the community. One of the biggest advantages of working with Ext is the great community around! There are very smart people behind this library that share their code to help other developers.

Preparing for deployment

So far we have been learning how to architect our JavaScript code, creating classes and layers for specific tasks, and writing maintainable and scalable code; but we need to prepare our application for a production environment.

In *Chapter 2*, *The Core Concepts*, we talked about the loader system in Ext 4, we learned that classes have dependencies and these dependent classes can be loaded automatically when requiring the main class.

In *Chapter 8*, *Architecture*, we used the `ext-dev.js` file to start loading the Ext JS classes dynamically and also our own classes from the modules just when the user needs them. When we are in a development environment it is really helpful to load each class in a separate file, this will allow us to debug the code easily and find and fix bugs.

Loading our modules on demand is a good idea to speed things up when loading our application the first time, but loading all classes one by one is really slow for a production environment. We should create packages of classes and only load one single file that should be compressed; this will increase the performance of our application considerably.

The Sencha Command

In order to compress and obfuscate our JavaScript classes, we can use the Sencha Command. This tool will help us run two tasks on the terminal.

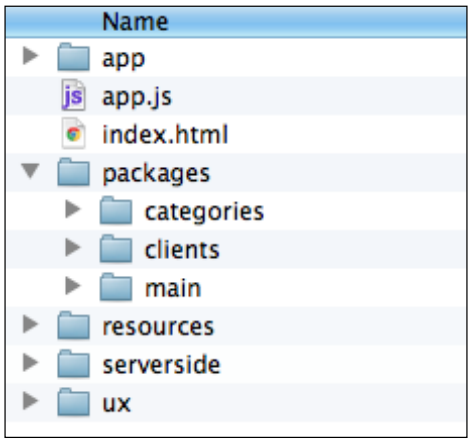
First, we need to create a descriptor file with all the packages and classes that we are using in our application or module. The Sencha command provides an easy way to accomplish this.

After we have our descriptor file we can compress all the needed classes and create a new file that is compressed, obfuscated, and minified.

We should have installed the Sencha Command in our system already. In *Chapter 12*, *Look and Feel*, we used this tool to automatically generate the images of our new theme for legacy browsers (Internet Explorer), now we are going to use the tool to create our packages.

Let's start by creating a new `packages` folder in the root folder of our application, inside a new folder we are going to create folders for each of the modules that we have, in this case we only have two modules (`Clients`, `Categories`), we are going to create another folder for the main layout that is loaded at the beginning and for the common classes across our application.

The following screenshot shows how our folder structure should be:



It's important to mention that the previous folder structure is optional. We may have a deeper level of directories according to our modules and application needs. The main idea is to create small JavaScript files containing all the classes for every module, then we are going to load each file just when the user needs them.

Creating the descriptor file

Let's start creating the descriptor file for our main layout. Before we begin let's make sure that we are using the `ext-dev.js` file to load our modules, our `index.html` file should look like the following example:

```
<!DOCTYPE HTML>
<html manifest="" lang="en-US">
<head>
<meta charset="UTF-8">
<title>Application</title>
<linkrel="stylesheet" href="resources/css/my-ext-theme.css" />
<linkrel="stylesheet" href="resources/css/style.css"/>

  <script type="text/javascript" charset="utf-8" src="../extjs-4.1.0/
ext-dev.js"></script>
<script type="text/javascript" src="app.js"></script>
</head>
<body>

</body>
</html>
```

First we load only the minimal code for the Ext JS framework, and then the rest of the needed classes for the main layout and our modules will be loaded dynamically. If we go to our favorite browser and navigate to the URL of our application, we will see that it's working fine.

Let's open our terminal and go to the root folder where our project is. Now we need to run the following command to create the `descriptor` file of our main layout.

```
$ sencha create jsb -a http://localhost/learning-ext-4/14-finalapp/index.
html -p packages/main/main.jsb3
```

Using the **create jsb** options, we tell the `sencha` command to create our descriptor file. This command receives only two parameters.

The **–a** parameter allows us to set the URL where our application is, in this case we are pointing to **localhost**, but we can even set an HTML file.

The **-p** parameter receives the location and the name of the descriptor file that will be generated, in this example we are saving the file inside the `packages/main` folder, the idea is to save each package in the respective folder. The extension of the descriptor file is `.jsb3`, this is the common extension that is used for these kind of files.

After executing this command we will see that a new `main.jsb3` file has been created inside of the `packages/main` folder, this file contains all the classes that are needed in order to run the main screen. If we open the file generated we can see something similiar to the following code:

```
{
    "projectName": "Project Name",
    "licenseText": "Copyright(c) 2012 Company Name",
    "builds": [
        {
            "name": "All Classes",
            "target": "all-classes.js",
            "options": {
                "debug": true
            },
            "files": [
                {
                    "clsName": "Ext.util.Observable",
                    "name": "Observable.js",
                    "path": "../extjs-4.1.0/src/util/"
                },


                //Many more definitions...


{
                    "clsName": "MyApp.view.Viewport",
                    "name": "Viewport.js",
                    "path": "app/view/"
                }
            ]
        },
        {
            "name": "Application - Production",
            "target": "app-all.js",
            "compress": true,
            "files": [
```

```
                {
                    "path": "",
                    "name": "all-classes.js"
                },
                {
                    "path": "",
                    "name": "app.js"
                }
            ]
        }
    ],
    "resources": []
}
```

First we have the name of the project, we can modify this according to our needs. We can also set a license text that will be added to each package that will be generated.

Then we have the `builds` property, this property allows us to define the files that will be generated, each file contains a `name`, a `target` where this package will be generated, some other options for compressing or debugging and the most important is the array of `files` that belongs to this package.

In the first package called `all-classes.js`, we have a lot of Ext JS classes but also we have our own classes that are needed for the first screen, in this case we have our main viewport. These packages will be for debugging, and will include all the classes just the way they were defined on their original files. We will see comments, white spaces, and everything we define on each one.

The second package is for production, we will compress this file and the Sencha tool will also obfuscate the source code by changing the name of variables, for example, if we have a variable called `eventStore` it will be converted into `a` or something like that. The files that we are going to include in these packages are `all-classes.js` and `app.js`.

Fixing the paths

Once we have the descriptor file generated we need to fix the paths for each file, the path should be relative to the descriptor file. We need to go up three folders for the Ext JS classes and two levels for our own classes. But if we have a different folder structure for the packages we should set the paths accordingly:

```
{
    "projectName": "Project Name",
    "licenseText": "Copyright(c) 2012 Company Name",
    "builds": [
```

```
{
    "name": "All Classes",
    "target": "all-classes.js",
    "options": {
        "debug": true
    },
    "files": [
        {
            "clsName": "Ext.util.Observable",
            "name": "Observable.js",
            "path": "../../../extjs-4.1.1/src/util/"
        },
        {
            "clsName": "Ext.data.IdGenerator",
            "name": "IdGenerator.js",
            "path": "../../../extjs-4.1.1/src/data/"
        },
        //...
        ,
        {
            "clsName": "Ext.tab.Panel",
            "name": "Panel.js",
            "path": "../../../extjs-4.1.1/src/tab/"
        },
        {
            "clsName": "MyApp.view.Viewport",
            "name": "Viewport.js",
            "path": "../../app/view/"
        }
    ]
},
{
    "name": "Application - Production",
    "target": "app-all.js",
    "compress": true,
    "files": [
        {
            "path": "",
            "name": "all-classes.js"
        },
        {
            "path": "../../",
            "name": "app.js"
```

```
                }
            ]
        }
    ],
    "resources": []
}
```

If we don't change these paths correctly we are going to get some errors when running the second command of the Sencha tools.

We should also add the `ext-debug.js` file to the production package; the idea is to only have one file with all the needed code for our main application.

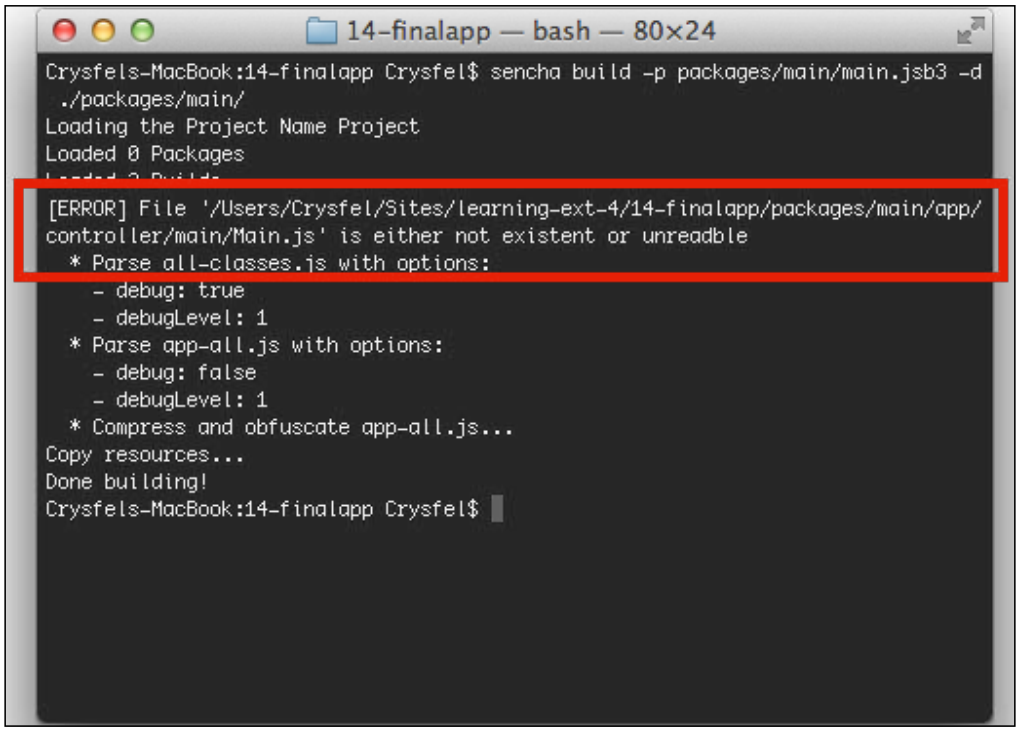
```
{
    "name": "Application - Production",
    "target": "app-all.js",
    "compress": true,
    "files": [
        {
            "path": "../../../extjs-4.1.1/",
            "name": "ext-debug.js"
        },
        {
            "path": "",
            "name": "all-classes.js"
        },
        {
            "path": "../../",
            "name": "app.js"
        }
    ]
}
```

We need to include the `ext-debug.js` file because all the code that we defined in this package will be obfuscated and minified. If we add code that is already minified we will get some errors when compressing.

Compressing the code

Once we have made the needed changes to the descriptor file we are ready to compress the classes and create the production package. We need to run the following command in the root folder of our application:

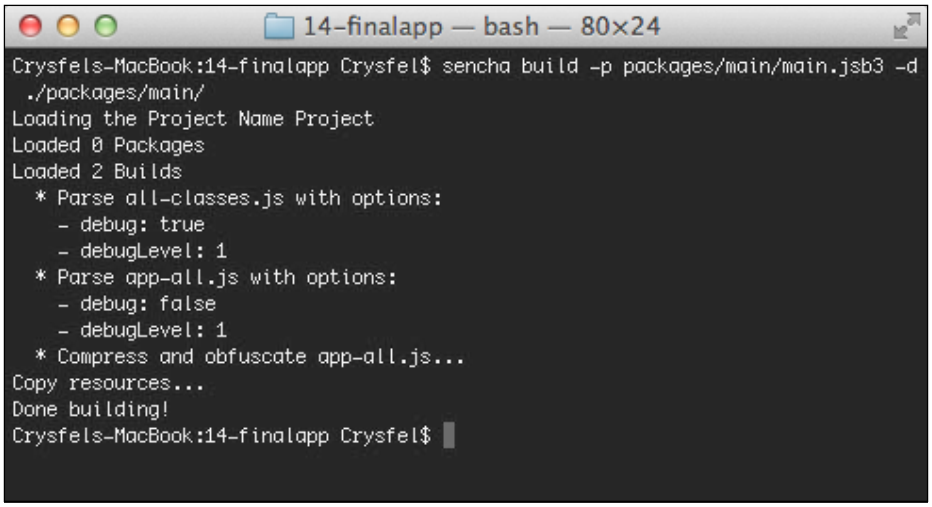
```
sencha build -p packages/main/main.jsb3 -d ./packages/main/
```



We are getting an error because we didn't set the path for our main controller, let's modify the descriptor file and set the correct path as follows:

```
{
    "clsName": "MyApp.controller.main.Main",
    "name": "Main.js",
    "path": "../../app/controller/main/"
}
```

After saving the changes, let's run the command again. We shouldn't get errors anymore and we will have two files in the `packages/main` folder:

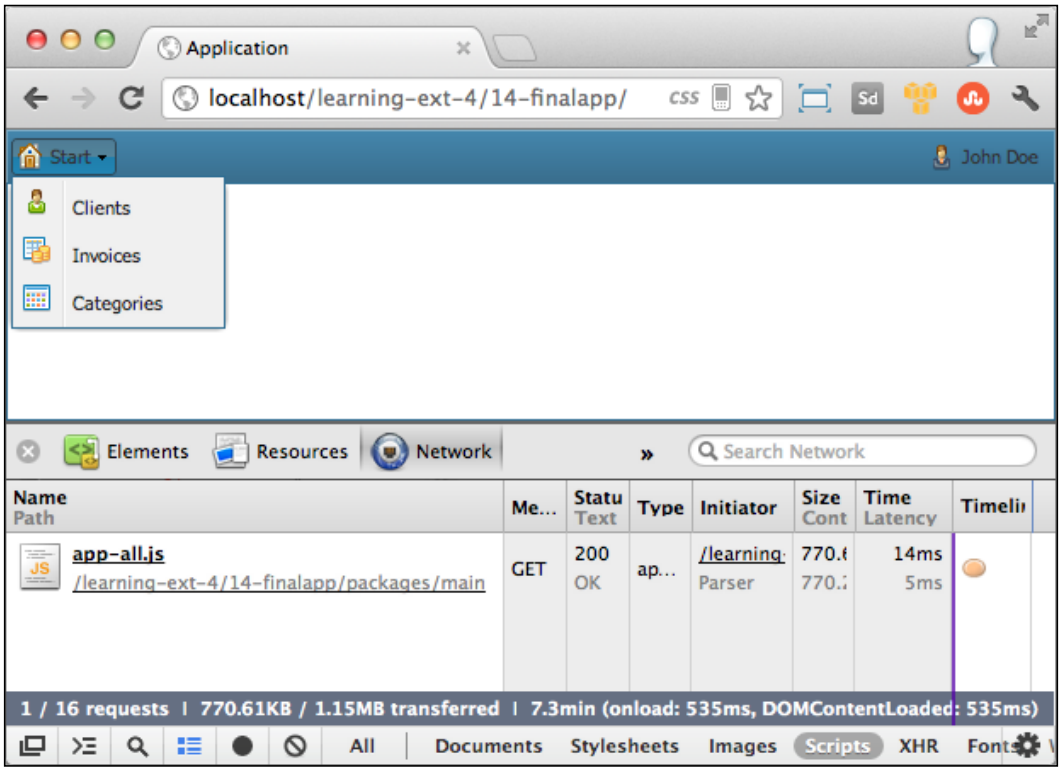


And now we are done! We have our code ready for the production environment, we only need to use the `all-app.js` file in our HTML file.

Let's open the `index.html` and change the following code:

```
<!DOCTYPE HTML>
<html manifest="" lang="en-US">
<head>
<meta charset="UTF-8">
<title>Application</title>
<linkrel="stylesheet" href="resources/css/my-ext-theme.css" />
<linkrel="stylesheet" href="resources/css/style.css"/>
<!--
  <script type="text/javascript" charset="utf-8" src="../extjs-4.1.0/
ext-dev.js"></script>
<script type="text/javascript" src="app.js"></script>
    -->

<script type="text/javascript" src="packages/main/app-all.js"></
script>
</head>
<body>

</body>
</html>
```

We have commented the `ext-dev.js` and `app.js` files because those files are already in our package. Let's open our browser and navigate to our application:



As we can see there's only one file that has been loaded, this file contains everything that is needed to start up our application. We should use this file for production environments because it's compressed and obfuscated.

Packaging the modules

If we try to open a module we will get errors, this is because `Ext.Loader` is trying to load the classes that are missing. We need to modify the code that opens the tab and instead of loading the classes one by one, we need to load only the package for the module.

Before modifying the code, let's create the package for the client's module, we will run the two commands that we have previously learned. In our previous examples, we ran the two commands against our main application, but in this case we need to create an HTML file where we include the controller for the client's module and render the views.

Let's create an HTML file in the root folder of our application. We will call this file `module.html` and we will add the following code:

```
<!DOCTYPE HTML>
<html manifest="" lang="en-US">
<head>
<meta charset="UTF-8">
<title>Application</title>
<linkrel="stylesheet" href="resources/css/my-ext-theme.css" />
<linkrel="stylesheet" href="resources/css/style.css"/>

<script type="text/javascript" src="packages/main/app-all.js"></
script>

<script type="text/javascript">

// Step 1 - Configure the paths for the loader
Ext.Loader.setConfig({   //Step 1
paths   : {
MyApp   : 'app',
Ext     : '../extjs-4.1.1/src'
        }
    });

//Step 2 - Include the client's controller
Ext.require('MyApp.controller.clients.Clients');

Ext.onReady(function(){

//Step 3 - Create an instance of the main view
var module = Ext.create('MyApp.view.clients.MainContainer');

Ext.create('Ext.container.Viewport',{ //Step 4
layout   : 'fit',
renderTo : Ext.getBody(),
items    : module
        });
```

```
setTimeout(function(){ //Step 5
var panel = Ext.ComponentQuery.query('viewport > panel')[0];
panel.hide();
        },100);

    });

</script>
</head>
<body>

</body>
</html>
```

First of all, we are including the main package that we created in the previous steps of this chapter, we are doing this because we don't want to include the same classes over again. Classes such as `Ext.Component` or `Ext.container.Container` are already loaded in the main package.

It's a good idea to define abstract classes for common components across our modules and add those classes to the main package.

In `Step 1`, we are configuring the paths for the loader, we need to do this because inside the `ext-dev.js` file the path for the Ext library is set automatically based on the URL of the file; but in this case we have that code embedded into our package so we need to define that path manually. We are also setting the path for our own application code.

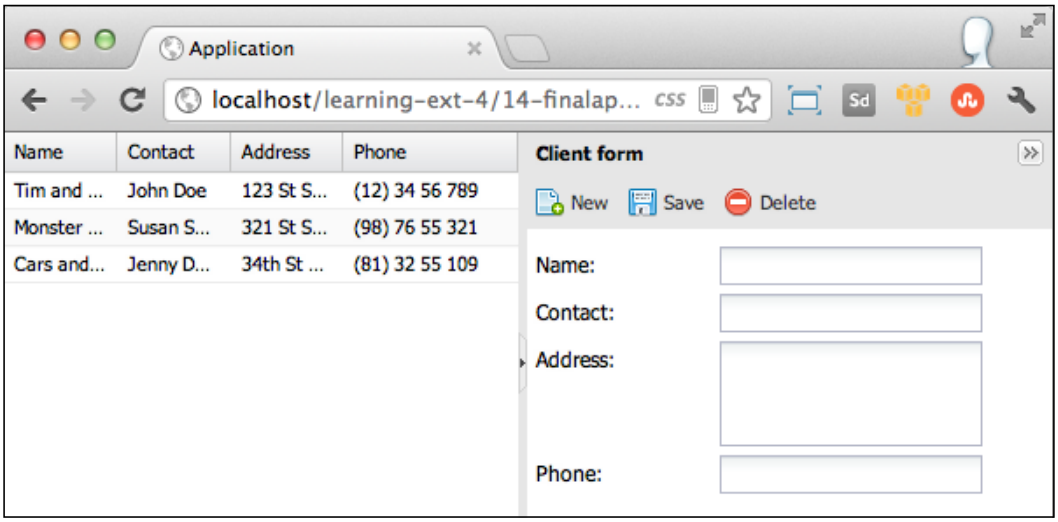
In `Step 2`, we require the client's controller, by doing this we are including all the dependencies too. This is exactly what we are doing when we open a tab in our application.

In `Step 3`, we create an instance of the main view, we are not passing any parameters in here.

In `Step 4`, we create a viewport and add our module, we are doing this in order to see the module on the screen.

In `Step 5`, we are going to hide the panel that is rendered on the viewport of our main application; we need to do that because right now we have two viewports, one is created in our main package and we are creating the other one to render the client's module. This step is not really needed but we are doing it just to show the client's module and not our main application.

Let's open our browser and see if everything is working well in the new HTML file that we have just created:



Now we are able to run the commands to create the descriptor file and compress the code for the client's module.

Depending on the server side technology that we are using we should dynamically create the previous HTML for every module if possible. This way we won't have an HTML file of each module in our application.

Let's create the descriptor file by running the following command:

```
$sencha create jsb -a http://localhost/learning-ext-4/14-finalapp/module.
html -p packages/clients/clients.jsb3
```

After that we should have a new file inside the `packages/clients` folder, in here we have the descriptor file for the client's module.

If we open this file we will notice that there are a few classes in here, all the needed classes for our module and a few others from the Ext JS library. This is because we already included the common classes in our main package and we are not duplicating the same classes again.

Let's change the name of our resulting file for `clients-all-classes.js`. We also need to fix the paths for our own classes and for the Ext JS classes. Finally, we need to remove the `app` file from the production package.

```
//...

{
    "name": "Application - Production",
    "target": "clients-all.js",
    "compress": true,
    "files": [
        {
            "path": "",
            "name": "clients-all-classes.js"
        }
    ]
}

//...
```

Once we have all our changes in place let's compress our code using the following command:

```
$sencha build -p packages/clients/clients.jsb3 -d ./packages/clients/
```

Now we should see two new files in the client's package folder. We need to use the `clients-all.js` file for our production environment.

Using the packages in our application

We have our new module packaged in one single file, however our application is not ready to load the module. Right now we are loading each class on its own, we need to prepare our code to load this single file if we are in a production environment, or if we are in a development environment we will continue loading each class individually.

In order to use packages, we need to modify the code that opens the tab and instead of including only the controller class we need to include the production package that we have previously created.

First, we need to know the environment where our code is running and based on that we will include the packages or the needed files to start up our application for development.

If we are using Java we can set Maven to create the needed HTML files based on the environment that our application will run. If we use Python, PHP, Ruby, or any other technology we can use conditionals to include one of the following scripts:

```
<!-- SCRIPTS FOR DEVELOPMENT ENVIRONMENT -->
  <script type="text/javascript" charset="utf-8" src="../extjs-4.1.0/
ext-dev.js"></script>
<script type="text/javascript" src="app.js"></script>


<!-- SCRIPTS FOR PRODUCTION ENVIRONMENT -->
<script type="text/javascript" src="packages/main/app-all.js"></
script>
```

Also, we need to define a flag to know the environment at runtime, this is necessary when opening the module to include the package or all the classes individually. In our index HTML file, we will define the following flag:

```
<script type="text/javascript">

MyApp.DEVELOPMENT = false;

</script>
```

We can set the value of the flag by looking into the URL of the browser, we can set it to `true` if we are in localhost and `false` if we are not. Also, we can define a property in our server and just assign that value to our JavaScript file.

Now, we need to add a new property to the main menu in order to define the name of the file that we are going to load when our code won't run on a development environment. Let's open our main viewport where we have our menu:

```
Ext.define('MyApp.view.Viewport',{
extend      : 'Ext.container.Viewport',

   //...

initComponent   : function(){
var me = this;

me.items = [{

   //...

  dockedItems : [{
```

```
          //...

        items : [{

         //...

        menu  : [
        {
    text:'Clients',
    iconCls:'clients-icon16',
    controller:'MyApp.controller.clients.Clients',
    packageFile:'packages/clients/clients-all.js'
    },{
    text:'Invoices',
    iconCls:'invoices-icon16',
    controller:'MyApp.controller.invoices.Invoices',
    packageFile:''
    },{
    text:'Categories',
    iconCls:'categories-icon16',
    controller:'MyApp.controller.categories.Categories',
    packageFile:''
    }]

           //...

            }];

    me.callParent();
        }
    });
```

We have already defined the controller that gets included when the user clicks on one of the options, now we are defining the package that we are going to load on production by adding a new property called `packageFile`. For now, we are only defining this property to the client's module but it's the same process to create the package for the other modules.

Let's open the `Main.js` controller that manages the main menu. We are going to modify the code of the `openModule` function as follows:

```
Ext.define('MyApp.controller.main.Main',{
  extend      : 'Ext.app.Controller',

  init   : function(){

  //...

  },
```

```
  openModule : function(menuoption){
    var me = this;

    Ext.Msg.wait('Loading...');

    if(MyApp.DEVELOPMENT){ //Step 1
      Ext.require(menuoption.controller,function(){
        me.initializeModule(menuoption); //Step 2
      });
    }else{
      Ext.Loader.loadScript({
        url    : menuoption.packageFile,
        onLoad  : function(){
          me.initializeModule(menuoption); //Step 2
        }
      });
    }
  },

  initializeModule : function(menuoption){ //Step 3
    var me = this,
      maintabs = Ext.ComponentQuery.query('#maintabs')[0];

    Ext.Msg.hide();

    var controller = me.application.controllers.get(menuoption.
controller);

    if(!controller){
      controller = Ext.create(menuoption.controller, {
        id      : menuoption.controller,
        application  : me.application
      });

      controller.container = me.createContainer(menuoption);
      maintabs.add(controller.container);
      controller.addContent();

      me.application.controllers.add(controller);
      controller.init(me.application);
      controller.onLaunch(me.application);
    }else{
      if(controller.container.isDestroyed){
        controller.container = me.createContainer(menuoption);
        maintabs.add(controller.container);
        controller.addContent();
      }
    }
```

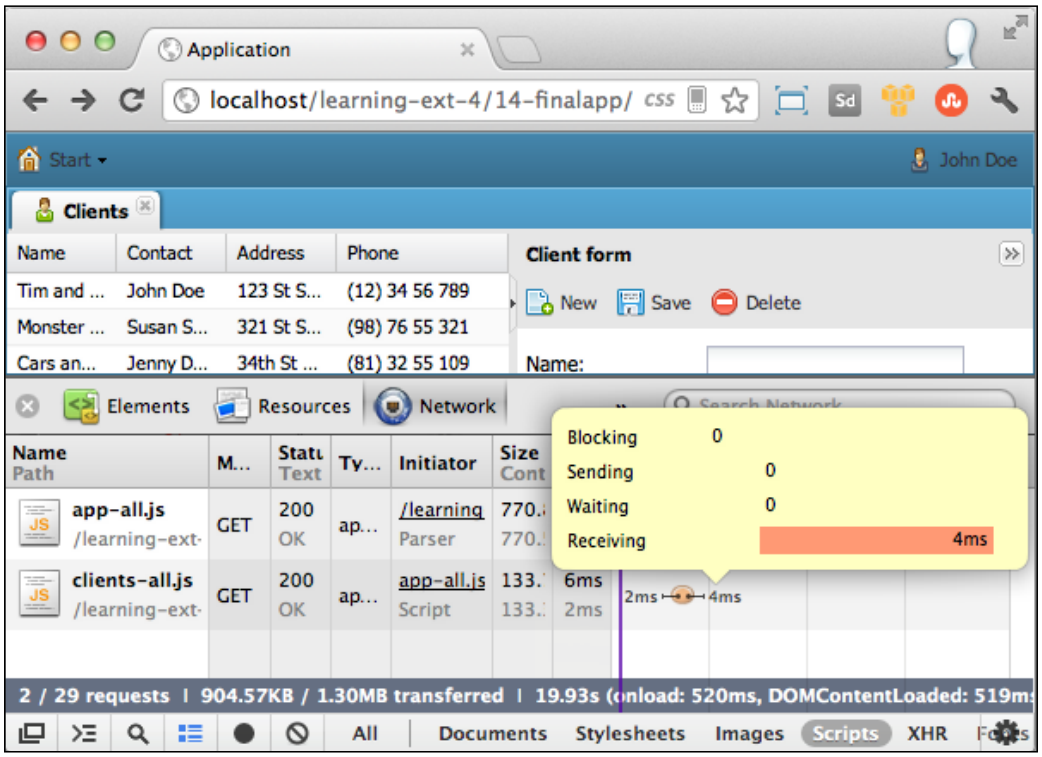
```
            maintabs.show();
            maintabs.setActiveTab(controller.container);
        },

        //...

    });
```

In the first step, we check for the flag that allows us to know the environment that our code is running. If it is development we include the controller as before but if not we load the script that contains all our classes.

The second step is executed once the file or files that we have included gets loaded to our document. We are calling a function called `initializeModule` on both situations, the only difference here is the way we are including the files.

The last step is the code that creates the controller instance if it doesn't exist yet, it creates the tab and shows the content for our module, we already had this code, we just encapsulate the code in a function in order to execute it in both scenarios.

Now we are ready to test our application, let's see how fast the packages are loading now:



We are only loading two files; the client's module gets loaded just when the user clicks on the option of the main menu. The previous screenshot shows that the client's package is loading in only **4ms**. That's pretty good and we can get this time because our code is compressed and because we have only one file with all the needed code.

The same process applies to the other modules, we need to create the descriptor file and then we will be able to concatenate the classes and compress it in one single file.

Our application is now ready to go live, and it is really easy to prepare the packages with this new Sencha tool. In previous versions of Ext, the process of creating the descriptor was a nightmare. We had to create this file manually and then compress it using the YUI compressor, this was a really time consuming task and we had many issues and errors by defining all dependencies and files.

In order to deploy our application, we only need to upload the `index.html` file and the `packages` folder. We should not upload our source code to the server unless we want other people to see it, but in most of the cases we don't want that.

Useful plugins and extensions

Ext JS 4 has a large set of plugins, these plugins add extra behavior to certain components and are very useful when the main component's functionality is not enough for our requirements. In this section we are going to see how to use some of the main plugins Ext JS 4 has integrated in the ux package.

Row expander

The `Ext.ux.RowExpander` adds an extra column to our grids, which enables a second row body that expands or contracts, this is very useful when we want to show extra information in grids. The expand/contract behavior is configurable for clicking the expand/contract column, we can configure this plugin to expand/collapse when we double-click on the selected row or when we press the *Enter* key in a selected row.

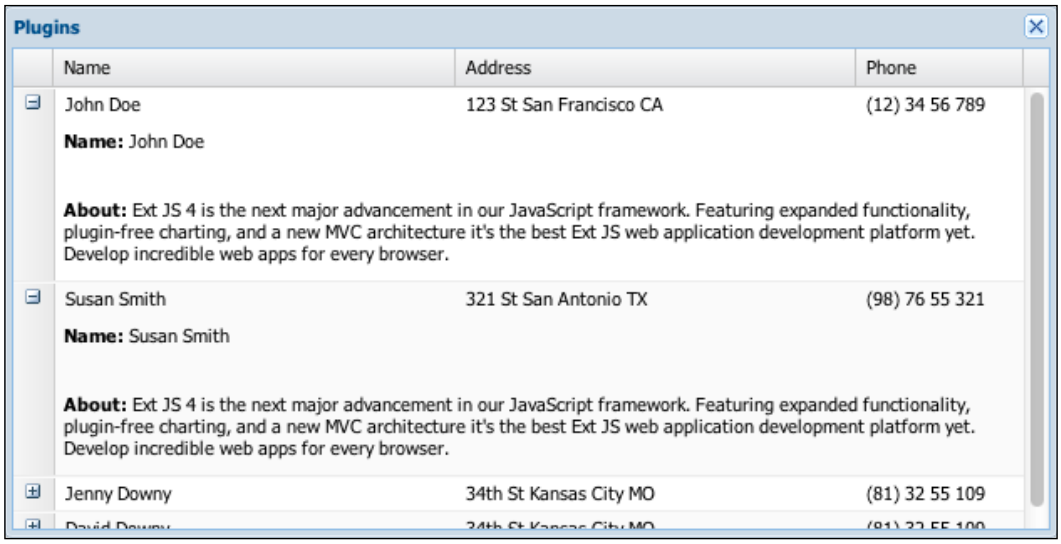
In the following example, we are going to see how the `Ext.ux.RowExpander` plugin is defined:

```
/**
 * @classMyApp.view.ClientsGrid
 * @extendsExt.grid.Panel
 * @author Armando Gonzalez<iam@armando.mx>
 *
 * The client's grid
 */
```

```
Ext.define('MyApp.view.ClientsGrid',{
  extend      : 'Ext.grid.Panel',
  alias       : 'widget.clientsgrid',
  requires  : [
    'Ext.ux.RowExpander'
  ],

  border    : false,
  store    : 'Clients',
  columns    : [
      {text:'Name',dataIndex:'name',flex:1},
      {text:'Address',dataIndex:'address',flex:1},
      {text:'Phone',dataIndex:'phone'}
    ],
  plugins    : [{
      ptype  : 'rowexpander',   // the row expander plugin definition
      rowBodyTpl  : [        // the second row body template
definition
        '<p><b>Name:</b> {name}</p><br>',
      '<p><b>About:</b> {about}</p>'
      ]
  }]
});
```

In the previous code, we have the row expander definition for the client's grid, we only need to add the plugin in the `plugins` array property and add the row body template to the new row body with the data that we want to render in the extra row body.

The following screenshot is the result of the previous code:



Grid printing

This plugin helps us to print the grid information, this plugin was done for Ext JS 3 by Ed Spencer (`http://edspencer.net/2009/07/printing-grids-with-ext-js.html`) and was updated by Loiane Groner (`http://loianegroner.com/2011/09/extjs-4-grid-printer-plugin`) for Ext JS 4. The plugin code is hosted on `https://github.com/loiane/extjs4-ux-gridprinter`, we only need to import the plugin code to our project and configure our grid with the printing behavior.

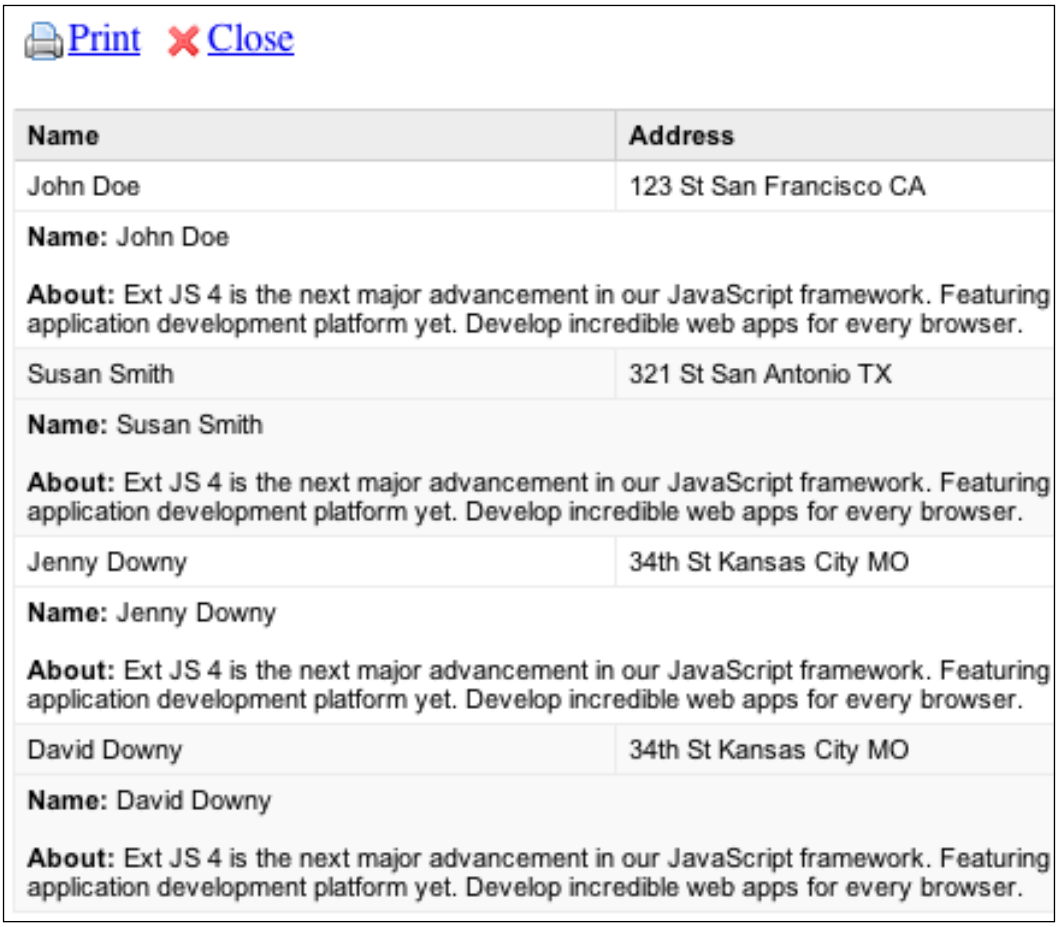
The following code shows how we can use the printing plugin:

```
/**
 * @classMyApp.view.ClientsGrid
 * @extendsExt.grid.Panel
 * @author Armando Gonzalez <iam@armando.mx>
 *
 * The client's grid
 */

Ext.define('MyApp.view.ClientsGrid',{
  extend      : 'Ext.grid.Panel',
  alias       : 'widget.clientsgrid',
  requires  : [
    'Ext.ux.RowExpander',
    'Ext.ux.grid.Printer'
  ],
  border    : false,
  store    : 'Clients',
  columns    : [
      {text:'Name',dataIndex:'name',flex:1},
      {text:'Address',dataIndex:'address',flex:1},
      {text:'Phone',dataIndex:'phone'}
    ],
  plugins    : [{
      ptype  : 'rowexpander',   // the row expander plugin definition
      rowBodyTpl  : [        // the second row body template
definition
        '<p><b>Name:</b> {name}</p><br>',
          '<p><b>About:</b> {about}</p>'
      ]
  }],
  initComponent: function() {
    var me = this;
    me.tbar = this.buildTbar();
  this.callParent(arguments);
  },
  buildTbar : function(){
```

```
        return ['->',{
          text:'Print',
          iconCls:'printer',
          scope:this,
          handler:function(){
            Ext.ux.grid.Printer.print(this);
          }
        }];
      }
    });
```

We need to import the `Ext.ux.grid.Printer` class and when the user clicks on the printing button the plugin will generate a nice HTML page where we can see the printing preview.

The following screenshot is the result of the printing plugin output:



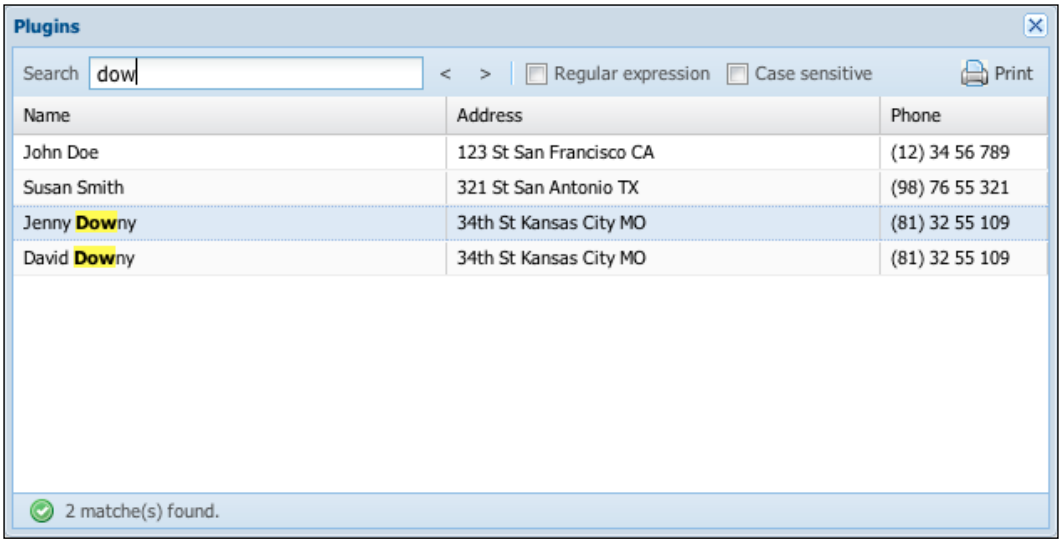
Live search grid

The `Ext.ux.LiveSearchGridPanel` is not a plugin, it is a grid extension that adds live search capabilities to the Ext JS 4 grids, this is very useful when we want to filter the data of our grid locally.

First, we need to import the `Ext.ux.live.SearchGridPanel` to our project files in the `ux` folder, and then we have to create our grid definition:

```
/**
 * @classMyApp.view.LiveSearchClientsGrid
 * @extendsExt.ux.LiveSearchGridPanel
 * @author Armando Gonzalez <iam@armando.mx>
 * The live search grid panel for the clients
 */


Ext.define('MyApp.view.LiveSearchClientsGrid', {
extend: 'Ext.ux.LiveSearchGridPanel',

alias       : 'widget.livesearchclientsgrid',
  requires  : [
    'Ext.ux.LiveSearchGridPanel',
    'Ext.ux.RowExpander',
    'Ext.ux.grid.Printer'
  ],
  border    : false,
  store     : 'Clients',
  columnLines: true,
  columns    : [
      {text:'Name',dataIndex:'name',flex:1},
      {text:'Address',dataIndex:'address',flex:1},
      {text:'Phone',dataIndex:'phone'}
  ],
  afterRender:function(){ //adding the printing button to the top tool
bar
    this.callParent(arguments);
    this.getDockedItems('toolbar')[0].add(this.buildTbar());
  },
  buildTbar : function(){
    return ['->',{
      text:'Print',
      iconCls:'printer',
      scope:this,
      handler:function(){
        Ext.ux.grid.Printer.print(this);
      }
    }];
  }
});
```

In the previous code we have the live search grid definition, and we added the printing button to the top toolbar, the following screenshot is the output of the live grid code definition:



GMapPanel

The `Ext.ux.GMapPanel` class is a Google Maps wrapper that enables easy displays of maps in any `Ext.container.Container`. Google Maps are very useful and with this extension we can have the Google Maps power in any panel, window, or container in our applications.

First, we need to import the Google Maps API with the following code in our main html file:

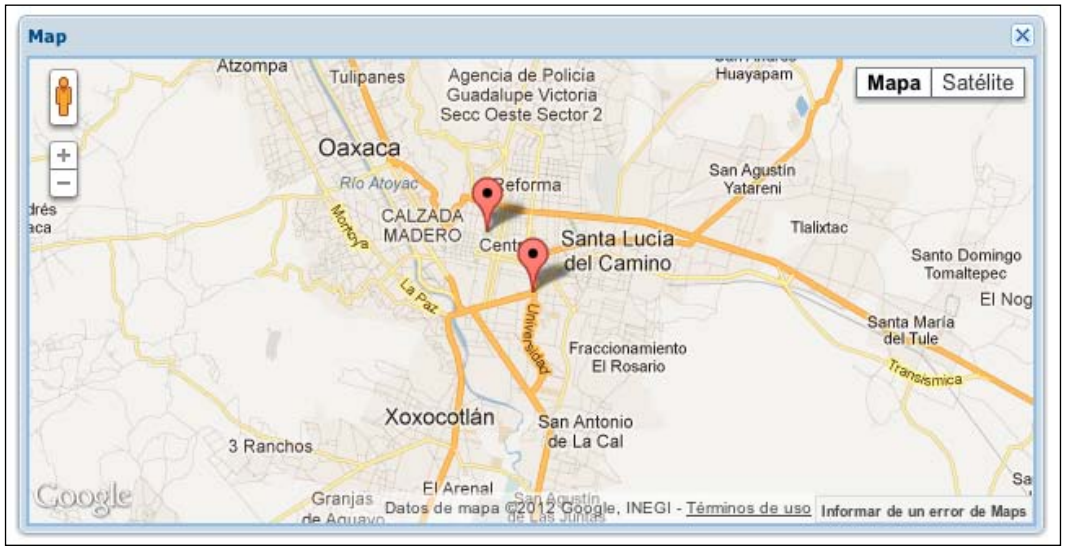
```
<script type="text/javascript" src="https://maps.googleapis.com/maps/
api/js?v=3&sensor=false"></script>
```

The following code describes the `Ext.ux.GMapPanel` use:

```
/**
 * @classMyApp.view.Map
 * @extendsExt.ux.GMapPanel
 * @author Armando Gonzalez <iam@armando.mx>
 * The map panel definition
 */
Ext.define('MyApp.view.Map', {
extend: 'Ext.ux.GMapPanel',
  alias:'widget.map',
```

```
    center: {
    geoCodeAddr: 'Oaxaca',
    marker: {
    title: 'Oaxaca'
            }
        },
    markers: [{
    lat: 17.066323,
    lng: -96.722989,
    title: 'Santo Domingo Museum',
    listeners: {
    click: function (e) {
    Ext.Msg.alert('Awesome place!');
                }
            }
        }]
});
```

In the `Ext.ux.GMapPanel` class, we can define the center location using the `center` property and add some markers to the map using the `markers` property. As we have seen, using Google Maps in our Ext JS 4 applications is very easy, the following screenshot is the result of the `Ext.ux.GMapPanel` configuration:



As mentioned earlier, we can add the map to any of the available containers, we can even use the layouts that Ext provides.

Summary

Preparing our application for production is a very important step in Ext applications. By following the steps described in this chapter we can improve the performance of our application. We can always define a process to automate these tasks by using Maven, or any other similar tool.

There are many more available plugins and extensions that we can use. In this chapter, we review and show how to use them, the process of adding new plugins to our project is very similar. We have a lot of free options or even paid options.

In the next chapter, we will learn about how to generate the documentation of our projects using the JS Duck tool, we will also learn about the Sencha Touch to make our application mobile.