Build Ext JS extensions and plug-ins

Joe Lennon, Software developer, Core International

Joe Lennon is a 25-year-old mobile and web application developer from Cork, Ireland. Joe works for Core International, where he leads the development of Core's mobile HR self service solutions. Joe is also a keen technical writer, having written many articles and tutorials for IBM developerWorks on topics such as DB2 pureXML, Flex, JavaScript, Adobe AIR, .NET, PHP, Python and much more. Joe's first book, *Beginning CouchDB* was published in late 2009 by Apress. In his spare time, Joe enjoys travelling, reading and video games.
(An IBM developerWorks Contributing Author)

Summary: The Ext JS framework is built in such a manner that makes it easy to extend. Ext JS extensions allow developers to create new classes that are derived from existing Ext JS classes, allowing the new class to use both its own new methods and the methods of the class that it extends. The Ext JS plug-in architecture allows developers to create new classes that do not depend on any existing Ext JS class (although they typically extend Ext.util.Observable). In this article learn how to build your own Ext JS extensions and plug-ins. Learn about the Ext JS classes, components, and how to extend them to create your own extensions and plug-ins.

Tags for this article: ajax, ext_js, extensions, java_technology, javascript, joe_lennon, plug-ins

Date: 23 Nov 2010
Level: Intermediate
PDF: A4 and Letter (144KB | 17 pages)Get Adobe® Reader®

Introduction

Ext JS is a comprehensive JavaScript framework that includes a vast collection of features, including cross-browser-compatible JavaScript helper utilities for DOM traversal and manipulation and data object stores. It also handles Ajax and direct web remoting, has an impressive collection of UI controls

and widgets, charts and graphs, a powerful data grid control, and much, much more.

When developing applications using Ext JS, or indeed any JavaScript library or framework, you can be sure that you are going to need functionality that isn't included out-of-the-box, or is present but doesn't work quite the way you want it to. Thankfully, Ext JS includes a powerful class and component ecosystem that makes it easy to extend existing features or build new components entirely. In this article, we will explore this extensibility, specifically looking at the concepts of extensions and plug-ins. You will learn what each of these concepts means and what is different between the two. You will then see how to build your own extensions and plug-ins and how to source existing plug-ins built by other developers for use in your own applications, saving you from reinventing the wheel.

Back to top

What is an extension?

An extension for Ext JS is a derived class or a subclass, which is designed to allow the inclusion of additional features to existing Ext JS classes. Extensions can be as basic as "preconfigured classes," which basically supply a set of default values to an existing class configuration, useful if you frequently need to supply these parameters when using a particular component. For example, your application may use a series of pop-up windows using the `Ext.Window` class, but these windows will always have the same width and height values. Rather than specifying this width and height every time you use `Ext.Window`, you could create an extension of `Ext.Window` with the preconfigured sizing properties. Not only does this save you from repeating the same code over and over, but it also makes your application much easier to maintain. And if you need to change the default size of your application's pop-up windows, you will only need to change it in one place rather than in several different locations in your code.

Preconfigured classes are the most basic type of extensions. In addition to supplying values for the existing properties in a class, subclasses can also add new properties and methods that are not included in the parent class (that is, the class that is being subclassed or derived from).

If you are familiar with the basics of object-oriented programming, you will know that when an object is instantiated, the class constructor function is automatically called. This function usually performs some initialization routines such as setting default values of class properties. By default, when you create a subclass, its constructor merely calls the constructor of its parent class. However, by creating a constructor for the subclass, you can override the constructor of the parent class, performing functionality specific to the subclass (setting default value properties that were added by the subclass, for example). Ext JS makes it very straightforward to do this and ensures that you can call the constructor of the parent or super class easily from the subclass constructor function, so that you don't have to add in the code from the parent class constructor into the subclass constructor.

Overriding methods in a parent class is not limited to the constructor. You can also override regular functions in a subclass and call the overridden parent function from within the subclass.

Back to top

What is a plug-in?

A plug-in is a class that is used to provide additional functionality to an existing component. Rather than directly instantiating an object of a plug-in class, plug-ins are attached to a component using the "plugins" configuration option for that component. One of the key advantages of using plug-ins is that

the plug-in can be used not only by the component it is attached to, but also by all subclasses derived from that component.

A nice feature of using plug-ins in Ext JS is that multiple plug-ins can be attached to a single component, providing additional functionality as required. This means that additional functionality can be broken up and only used when needed, improving the performance of the application. Of course, plug-ins should be written in such a way that they do not conflict with other plug-ins, otherwise they will not be able to be used concurrently in a component.

Back to top

Differences between extensions and plug-ins

At first it can be difficult to appreciate the difference between an extension and a plug-in in Ext JS, particularly given their similarities in terms of what they do. At the end of the day, both provide extended functionality to Ext JS classes. The primary difference lies in how they are written.

Ext JS extensions are written as subclasses of an existing class. They can provide additional properties or functions, or even modify the behavior of the parent class by overriding the constructor and other functions. To use an extension, you directly create an object of the extension subclass, which can then use, add, or override features to the parent class it is derived from. Extensions are best used when you need to change core functionality of an existing Ext JS component or feature, or when you want to build a brand new component altogether.

Ext JS plug-ins are written as classes that always have an init function. Unlike extensions, you will not create an object of a plug-in class directly, but rather you will register your plug-in with the component it attaches to. The options and functions defined in the plug-in will then become available to the component itself. Plug-ins are best used when you need to add features to a component. Because extensions are so tightly coupled with the classes they are derived from, plug-ins also present a viable alternative when you want your add-on to be easily detachable and interoperable with multiple components and derived components. Because plug-ins must be attached to an existing component, they are not usually suitable for building new components from scratch.

Back to top

Building an Ext JS extension

In this section, you will learn how to build Ext JS extensions. You will first learn how to create preconfigured classes, which will make it easy to reuse code and keep your application in a maintainable state by creating subclasses that preset configuration options to default values. You will then learn how to create a more complex extension which changes the functionality of an existing Ext JS class and adds some new features.

Creating preconfigured classes

As previously stated, preconfigured classes are the most basic form of Ext JS extension. They simply define a set of preset configuration option values that allows you to use a component without having to pass default values each and every time. An example of this would be an `Ext.Window` component, where in your application many of the properties of the window are always the same, for example width, height, and title. For an example of this, look at this particular scenario, without the use of a preconfigured class (see Listing 1).

Listing 1. unextended.html

```
<html>
    <head>
        <title>Popup Window - Unextended Version</title>
        <link rel="stylesheet" type="text/css" href="ext/resources/
css/ext-all.css" />
        <script type="text/javascript" src="ext/adapter/ext/
ext-base.js"></script>
        <script type="text/javascript" src="ext/ext-all.js"></script>
        <script type="text/javascript">
        Ext.onReady(function() {
            var win1 = new Ext.Window({
                width: 500,
                height: 300,
                modal: true,
                closeAction: 'hide',
                resizable: false,
                maximizable: true,
                collapsible: true,
                draggable: false,
                title: 'Window 1',
                html: '<h2>Window 1</h2>'
            });

            var win2 = new Ext.Window({
                width: 500,
                height: 300,
                modal: true,
                closeAction: 'hide',
                resizable: false,
                maximizable: true,
                collapsible: true,
                draggable: false,
                title: 'Window 2',
                html: '<h2>Window 2</h2>'
            });

            var win3 = new Ext.Window({
                width: 500,
                height: 300,
                modal: true,
                closeAction: 'hide',
                resizable: false,
                maximizable: true,
                collapsible: true,
                draggable: false,
                title: 'Window 3',
                html: '<h2>Window 3</h2>'
            });

            var button1 = Ext.get('button1');
            var button2 = Ext.get('button2');
            var button3 = Ext.get('button3');

            button1.on('click', function() {
                win1.show(this);
```

```
            });

            button2.on('click', function() {
                win2.show(this);
            });

            button3.on('click', function() {
                win3.show(this);
            });
        });
        </script>
    </head>

    <body>
        <button id="button1">Show Window 1</button>
        <br />
        <button id="button2">Show Window 2</button>
        <br />
        <button id="button3">Show Window 3</button>
    </body>
</html>
```

In Listing 1 above, there are three buttons, each of which will show a separate `Ext.Window` component. Each of these windows has the same value for the following properties:

- `width`
- `height`
- `modal`
- `closeAction`
- `resizable`
- `maximizable`
- `collapsible`
- `draggable`

As you can see clearly in Listing 1, each of these eight properties is set in the configuration for each of the three `Ext.Window` objects. Not only does this result in extra lines of code, but it also means that if you decide to change the default property values of the `Ext.Window`, you need to make this change in three different places, which is less than ideal. This is where preconfigured classes come into play as shown in Listing 2.

Listing 2. extended.html

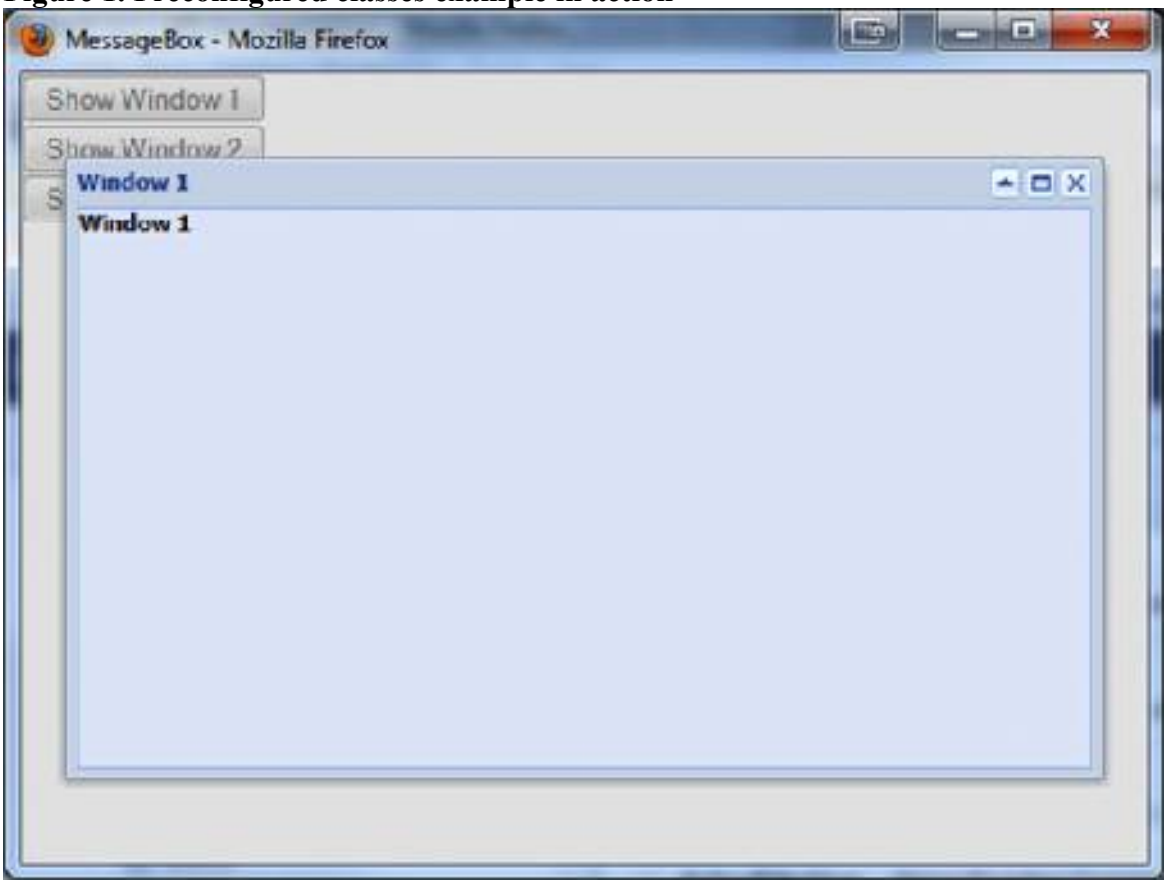
```
<html>
    <head>
        <title>Popup Window - Extended Version</title>
        <link rel="stylesheet" type="text/css" href="ext/resources/css/ext-all.css" />
        <script type="text/javascript" src="ext/adapter/ext/ext-base.js"></script>
        <script type="text/javascript" src="ext/ext-all.js"></script>
        <script type="text/javascript">
        MyWindow = Ext.extend(Ext.Window, {
            width: 500,
            height: 300,
            modal: true,
```

```
            closeAction: 'hide',
            resizable: false,
            maximizable: true,
            collapsible: true,
            draggable: false
        });

        Ext.onReady(function() {
            var win1 = new MyWindow({
                title: 'Window 1',
                html: '<h2>Window 1</h2>'
            });

            var win2 = new MyWindow({
                title: 'Window 2',
                html: '<h2>Window 2</h2>'
            });

            var win3 = new MyWindow({
                title: 'Window 3',
                html: '<h2>Window 3</h2>'
            });

            var button1 = Ext.get('button1');
            var button2 = Ext.get('button2');
            var button3 = Ext.get('button3');

            button1.on('click', function() {
                win1.show(this);
            });

            button2.on('click', function() {
                win2.show(this);
            });

            button3.on('click', function() {
                win3.show(this);
            });
        });
        </script>
    </head>

    <body>
        <button id="button1">Show Window 1</button>
        <br />
        <button id="button2">Show Window 2</button>
        <br />
        <button id="button3">Show Window 3</button>
    </body>
</html>
```

Much of the code in this listing is the same as in Listing 1. However, you will notice above the `Ext.onReady` call that I have created a new subclass, `MyWindow`. This is created using the `Ext.extend` function, specifying the default configuration values for the eight properties that are common across all three of my `Ext.Window` objects. As you can see, you can then create objects of

this new subclass, specifying the two properties (`title` and `html`) that vary between each of the windows. The resulting application looks the same (see Figure 1) for both code listings, but I'm sure you'll agree that Listing 2 is a much more maintainable method.

Figure 1. Preconfigured classes example in action



Extending an Ext JS component

Predefining the configuration options for a component is all well and good, but what about extending the functionality of a component to provide additional features? In this section, you will learn how to create a subclass of `Ext.Window` that behaves like a web browser window.

You can use the <iframe> HTML element to display web pages inside another web page, so we'll take advantage of that to create a new `ExtBrowserWindow` class that will display this web page inside an `Ext.Window`. The goal is to simplify the process. To create a pop-up Ext browser window, all you need to do is supply a unique ID and a URL to display. In fact, in this example, you will even supply a default URL to navigate to, should none be supplied.

Listing 3 shows the full source code for the example.

Listing 3. extbrowser.html

```
<html>
    <head>
        <title>Ext Browser Window Extension</title>
        <link rel="stylesheet" type="text/css"
href="ext/resources/css/ext-all.css" />
        <script type="text/javascript" src="ext/adapter/ext
```

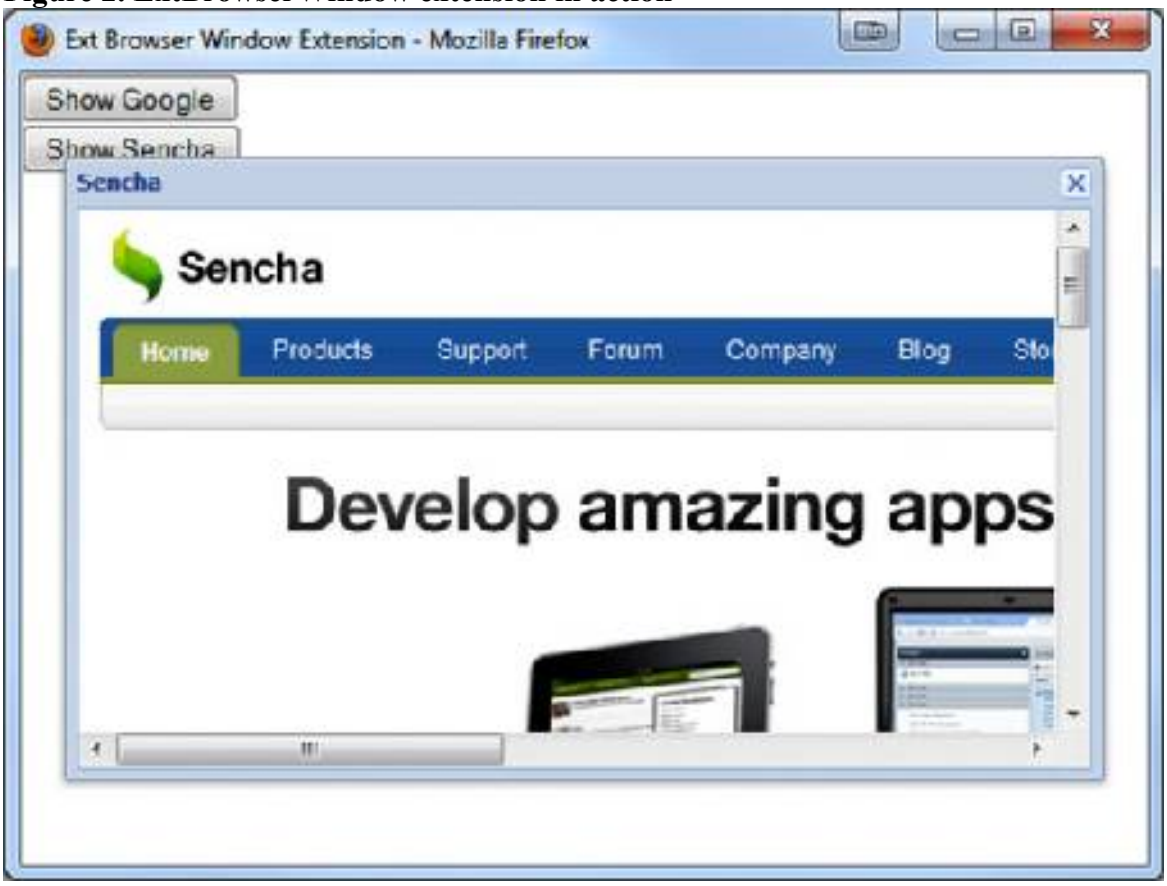
```
/ext-base.js"></script>
        <script type="text/javascript" src="ext/ext-all.js"></script>
        <script type="text/javascript">
        ExtBrowserWindow = Ext.extend(Ext.Window, {
            closeAction: 'hide',
            width: 500,
            height: 300,
            title: 'Sencha',
            url: 'http://www.sencha.com',
            onRender: function() {
                this.bodyCfg = {
                    tag: 'iframe',
                    src: this.url,
                    cls: this.bodyCls,
                    style: { border: '0px none' }
                };
                ExtBrowserWindow.superclass.onRender.apply(this, arguments);
            }
        });

        Ext.onReady(function() {
            var win1, win2;

            var button1 = Ext.get('button1');
            var button2 = Ext.get('button2');

            button1.on('click', function() {
                if(!win1) {
                    win1 = new ExtBrowserWindow({
                        id: '1',
                        title: 'Google',
                        url: 'http://www.google.com'
                    });
                }
                win1.show(this);
            });

            button2.on('click', function() {
                if(!win2) {
                    win2 = new ExtBrowserWindow({
                        id: '2',
                    });
                }
                win2.show(this);
            });
        });
        </script>
    </head>

    <body>
        <button id="button1">Show Google</button>
        <br />
        <button id="button2">Show Sencha</button>
    </body>
</html>
```

Similar to the preconfigured class example, you are creating a subclass of the `Ext.Window` class and supplying some default configuration options such as width, height, and title. In this instance, however, you are also setting a new option that is specific to this new class, `url`. This is the property that will determine the web page URL to display in the window. In the example, this is set to default to the Sencha home page (see Resources).

The `ExtBrowserWindow` subclass also overrides the `onRender` event function and applies a configuration to the body of the `Ext.Window`, telling it that it should use the iframe tag, with the `src` attribute set to the url configuration option and the style set to have no border. Finally, the corresponding `onRender` event handler in the superclass will be called.

The actual instantiation of the `ExtBrowserWindow` object is much the same as in the previous examples, except this time we're passing our user-defined configuration option `url` in the first button call (which will open Google). In the second example, we only use the default values specified in the class, which will show the Sencha home page. Figure 2 shows the `ExtBrowserWindow` extension in action.

Figure 2. ExtBrowserWindow extension in action



This is a very basic example of an Ext JS extension, but it could easily be extended further to include some more features. For example, you could add typical web browser features like back and forward buttons, an address bar, or even bookmarks.

Back to top

Building an Ext JS plug-in

In this section, you will learn how to build an Ext JS plug-in. For the sake of simplicity, I will show you how to create the same functionality as in the previous section, but this time it will be created as a plug-in rather than an extension. It it standard practice to create plug-ins in an external JavaScript source file, so start off by creating a new file named BrowserPlugin.js, the contents of which are shown in Listing 4.

Listing 4. BrowserPlugin.js

```
BrowserPlugin = Ext.extend(Object, {
    init: function(component) {
        component.width = 500;
        component.height = 300;
        component.bodyCfg = {
            tag: 'iframe',
            src: component.url,
            cls: component.bodyCls,
            style: { border: '0px none' }
        };
    }
});
Ext.preg('browserPlugin', BrowserPlugin);
```

The plug-in is very simple: It contains the basic requirements of an Ext JS plug-in. The `init` function sets the width and height of the component accordingly and tells the component that it should render as an <iframe> element, with the `src` attribute coming from the URL configuration option, as it was in the extension example. Finally, the `BrowserPlugin` class is registered as a plug-in to allow it to be loaded lazily. This basically means that you don't need to instantiate the object manually; Ext JS will take care of it for you.

Next, create plugin.html, the file which will use the plug-in you just created. Its contents are listed in Listing 5.

Listing 5. plugin.html

```
<html>
    <head>
        <title>Ext Browser Window Extension</title>
        <link rel="stylesheet" type="text/css" href="ext/resources/css
/ext-all.css" />
        <script type="text/javascript" src="ext/adapter/ext
/ext-base.js"></script>
        <script type="text/javascript" src="ext/ext-all.js"></script>
        <script type="text/javascript" src="BrowserPlugin.js"></script>
        <script type="text/javascript">
        Ext.onReady(function() {
            var win1, win2;

            var button1 = Ext.get('button1');
            var button2 = Ext.get('button2');

            button1.on('click', function() {
                if(!win1) {
```

```
                    win1 = new Ext.Window({
                        title: 'Google',
                        closeAction: 'hide',
                        plugins: ['browserPlugin'],
                        url: 'http://www.google.ie'
                    });
                }
                win1.show(this);
            });

            button2.on('click', function() {
                if(!win2) {
                    win2 = new Ext.Window({
                        title: 'Sencha',
                        closeAction: 'hide',
                        plugins: ['browserPlugin'],
                        url: 'http://www.sencha.com'
                    });
                }
                win2.show(this);
            });
        });
        </script>
    </head>

    <body>
        <button id="button1">Show Google</button>
        <br />
        <button id="button2">Show Sencha</button>
    </body>
</html>
```

Again, this code is fairly familiar. You'll notice that below the loading of ext-all.js the new BrowserPlugin.js file is being loaded. The primary difference between the use of extensions and plug-ins can be seen in the click event handlers for the two buttons. When we created an extension, we instantiated an object of the new extension's subclass (`ExtBrowserWindow`), but when using a plug-in you instead use the regular Ext JS component you wish to display. In this case that is the `Ext.Window` component. In the configuration options for the component, you then specify that you wish to load the "browserPlugin" plug-in using the "plugins" configuration option.

The result of this is identical to that of the extension we created in the previous section, and you can see a screen shot of it in action in Figure 2, as seen earlier in this article. In this section, we have covered the basics of creating Ext JS plug-ins. Documentation and reading material on the subject is fairly sparse to say the least, so the best way to learn more advanced techniques is to download, use, and tinker with plug-ins and extensions created by other users in the Ext JS community. In the next section, you will learn how to use an existing extension in your own applications.

Back to top

Using existing Ext JS plug-ins

You can find existing Ext JS extensions and plug-ins in a couple of ways. The first (and most complete) source is the Ext User Extensions and Plug-ins forum on the Sencha website (see Resources). An

unofficial user extension repository is also available (see Resources). Most plug-ins and extensions are simply a JavaScript source file that you can download or copy and use in your own application. Most of these plug-ins and extensions are available for free use under relaxed licenses, but be sure to check the license for the extension in question before using in a commercial project.

When including an extension or plug-in, ensure that you do so after you have included the source JavaScript and CSS files for Ext JS itself. Extensions and plug-ins often override default Ext JS component functionality and require that these are loaded in advance of loading themselves.

The Ext JS project examples include a series of demos of custom extensions in action. In these demos, view the HTML source of the page to find the URL of the extension or plug-in JavaScript source file (if you are using JavaScript you can click on the link to open the source file directly). Copy this code and paste it into a file to use in your own projects. Each demo has a link to the JavaScript code that actually puts the plug-in into practice, so click on that to see how the example works.

See Resources for links to Ext JS samples that are built on user extensions.

As you can see, a vast quantity of the official Ext JS samples and demos are built on user-contributed extensions. The user community plays a key role in the enhancement of the Ext JS framework, and several user extensions have gone on to become standard components in the framework itself.

Take some time to explore the demos above and look at the extension source code to get an appreciation for how complex an Ext JS extension or plug-in can be. User extensions in Ext JS usually are in the namespace Ext.ux, so look out for this when trying to find the source code for an extension.

Back to top

Conclusion

In this article you learned the basics of Ext JS extension and plug-in development. You discovered what both of these concepts mean, what the differences are between them, and how to actually create them. You created a preconfigured class that saves time and valuable lines of code that can lead to more efficient applications that are easier to maintain. You then learned how to create an Ext JS extension—a browser window that subclassed the `Ext.Window` component. You learned how to define new properties and call corresponding functions in the superclass and how to use the new class in your application code. You then learned how to do the same thing, but in the form of an Ext JS plug-in. You learned how to create the plug-in, defining the mandatory `init` function, and how to register the plug-in to make it available to Ext JS components. With the basics learned in this article, you should now be ready to start learning about more complex concepts in Ext JS extension and plug-in development.

Back to top

Download

Description	Name	Size	Download method
Article source code	extjs.example.zip	3KB	HTTP

Information about download methods

Resources

Learn

- Visit the home page of Sencha, the creators of Ext JS library.
- View Ext JS 3.2 samples that are built on user extensions.
- Visit Saki's Extensions, Plugins and Know-How: Jozef Sakalos, aka Saki, is a member of the Ext JS support team.
- "Build Ajax applications with Ext JS" (John Fronckowiak, developerWorks, July 2008): Get an overview of the object-oriented JavaScript design concepts behind Ext JS, and learn how to use the Ext JS framework for rich Internet application UI elements.
- "Compare JavaScript frameworks" (Joe Lennon, developerWorks, February 2010): Get an overview of the frameworks that greatly enhance JavaScript development.
- "Use Ext, Aptana, and AIR to build desktop applications" (Joe Lennon, developerWorks, July 2008): Use the open source Aptana Studio IDE, the Adobe AIR plug-in for Aptana, and the open-source JavaScript framework Ext to create a contact management utility.
- "Mastering Ajax, Part 2: Make asynchronous requests with JavaScript and Ajax" (Brett McLaughlin, developerWorks, January 2006): Learn how to use Ajax and the XMLHttpRequest object to create a request/response model that never leaves users waiting for a server to respond.
- "Create Ajax applications for the mobile Web" (Michael Galpin, developerWorks, March 2010): Learn how to build cross-browser smart phone web applications using Ajax.
- "Where and when to use Ajax in your applications" (Jesse Skinner, developerWorks, February 2008): Learn how you can use Ajax to improve your websites while avoiding bad user experiences.
- "Improve the performance of Web 2.0 applications" (Jian Qiao Sun and Hua Pin Shen, developerWorks, December 2009): Explore different browser-side cache mechanisms.
- The developerWorks Web development zone specializes in articles covering various web-based solutions.
- The developerWorks tutorial series "Learning PHP" takes you from the most basic PHP script to working with databases and streaming from the file system.
- PHP.net is the central resource for PHP developers.
- Check out the "Recommended PHP reading list."
- Browse all the PHP content on developerWorks.
- Expand your PHP skills by checking out IBM developerWorks' PHP project resources.
- To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.
- Stay current with developerWorks technical events and webcasts.

Get products and technologies

- Get Ext JS (3.2.1 at the time of writing), the cross-browser JavaScript library for building rich Internet applications.
- Visit the official Ext JS samples and demos page.

- Visit the unofficial Ext JS user extension repository.
- XAMPP provides easy installation of Apache, PHP, MySQL, and other goodies.
- Innovate your next development project with IBM trial software, available for download or on DVD.

Discuss

- Visit the community forums for Ext: User Extensions and Plugins.
- Create your My developerWorks profile today and set up a watchlist on Ext JS. Get connected and stay connected with My developerWorks.
- Find other developerWorks members interested in web development.
- Share what you know: Join one of our developerWorks groups focused on web topics.
- Roland Barcia talks about Web 2.0 and middleware in his blog.
- Follow developerWorks' members' shared bookmarks on web topics.
- Get answers quickly: Visit the Web 2.0 Apps forum.
- Get answers quickly: Visit the Ajax forum.

About the author

Joe Lennon is a 25-year-old mobile and web application developer from Cork, Ireland. Joe works for Core International, where he leads the development of Core's mobile HR self service solutions. Joe is also a keen technical writer, having written many articles and tutorials for IBM developerWorks on topics such as DB2 pureXML, Flex, JavaScript, Adobe AIR, .NET, PHP, Python and much more. Joe's first book, *Beginning CouchDB* was published in late 2009 by Apress. In his spare time, Joe enjoys travelling, reading and video games.