Article

Introduction to Ajax for Page Authors

Print-friendly Version

By Ed Ort, January 2007

Contents

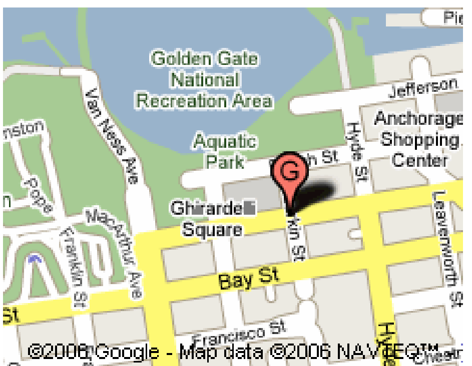
- Ajax and You
- What You Need to Know About Ajax
- Ajax and JavaScript Technology
- Adding Ajax to a Web Page
- Using Ajax-Enabled Components
- Using Ajax-Enabled Widgets
- Using jMaki Widgets
- Writing the JavaScript Code Yourself
- Using JavaScript Libraries
- Summary
- For More Information
- About The Author

Get Ajax Training

- Developing JavaServer Faces Web Applications With Ajax Using Sun Studio Creator (DTJ-2105)

Use your SDN member priority code and get a 10% training discount. Log in to your SDN account and find the priority code under Developer Discounts. Not yet an SDN member? Join now.

Ajax and You



It's almost impossible today to be involved in web application design or development and not be aware of Ajax, a technology that includes but is not limited to Asynchronous JavaScript and XML. That's because Ajax is currently the primary technique for driving the high responsiveness and interactivity of some of the most popular applications on the web such as Google Maps and Flickr. These applications are representative of a new generation of highly responsive, highly interactive web applications, referred to as Web 2.0 applications, that often involve users collaborating online and sharing content.

Ajax has different implications for developers working in different roles. For example, component developers

creating custom components for web applications build Ajax functionality into the design. Page authors use these Ajax components, along with widgets, JavaScript technology, and other techniques, to incorporate Ajax functionality into their web applications. Ajax impacts other roles too. For example, enterprise application developers need to add logic in server-side components to handle Ajax-related requests directed to the server.

This article focuses on page authors and describes various techniques that you can use to add Ajax functionality to a web page.

What You Need to Know About Ajax

As a page author, you don't need to understand every detail of the Ajax methodology to incorporate it into a web page. However, you should have a general idea of what Ajax is and how it works. As this article mentioned earlier, Ajax enables the high responsiveness of many web applications. For example, a web site such as Google Maps uses Ajax to provide a highly responsive user interface (UI). You can view a map, then move your cursor across it to see adjacent areas almost immediately.

This article assumes that you're using a web browser that supports the Ajax methodology. Most popular browsers, including Mozilla browsers and Internet Explorer, do.

Ajax enables high responsiveness because it supports asynchronous and partial refreshes of a web page. A *partial refresh* means that when an interaction event fires -- for example, a user moves the cursor across a Google map -- a web server processes the information and returns a limited response specific to the data it receives. Significantly, the server does not send back an entire page to the client of the web application -- in this case a web browser -- as is the case for conventional "click, wait, and refresh" web applications. The client then updates the page based on the response.

Asynchronous means that after sending data to the server, the client can continue processing while the server does its processing in the background. This means that a user can continue interacting with the client without noticing a lag in response. For example, a user can continue to move the mouse over a Google map and see a smooth, uninterrupted change in the display because extended parts of the map have been loaded asynchronously. The client does not have to wait for a response from the server before continuing, as is the case for the traditional synchronous approach.

Ajax and JavaScript Technology

Underlying all of the approaches for adding Ajax functionality to a web page is JavaScript technology. In this article, you'll see that in some approaches, such as using an Ajax-enabled JavaServer Faces component, the component encapsulates the JavaScript code,so you don't have to do any JavaScript coding. In other approaches, such as using a widget in a JavaScript library, the library provides most of the JavaScript code, but you will need to add a little JavaScript code yourself. If you don't find components or widgets that provide the Ajax functionality you need, you must do a lot more JavaScript coding yourself. However, you don't have to rely on any one of these approaches -- You can use various approaches in combination.

Scripts written in the JavaScript scripting language handle much of the Ajax-related processing in the client as illustrated in Figure 1.

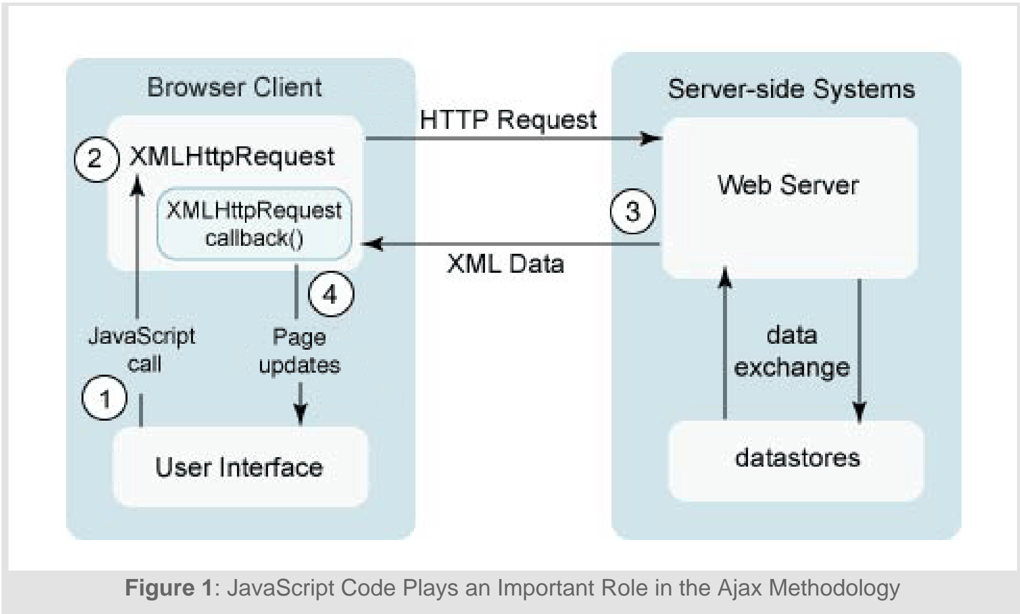


Figure 1: JavaScript Code Plays an Important Role in the Ajax Methodology

1. When a user generates an event on the client, it calls a JavaScript function in the client.
2. The JavaScript function creates and configures a JavaScript object that is used to exchange data between the client and web server. In Mozilla-based browsers, the object is an `XMLHttpRequest` object. In Internet Explorer, it's an `XMLHTTP` object -- this is an ActiveX object rather than a native JavaScript object. The JavaScript function also specifies a JavaScript callback function.
3. The server processes the data and returns a response, typically in XML format, although other formats can be used.
4. The callback function processes the response and updates an internal representation of the page based on the new data. This changes what the user sees on the page.

Like other web applications, an Ajax-enabled web application uses a markup language such as HTML or XHTML to present web pages, or a server-side technology such as JavaServer Pages (JSP) technology to generate web pages. In addition, an Ajax-enabled web application typically uses cascading style sheets (CSS), a stylesheet language, to define presentation style, such as fonts and colors. A server-side application system such as Java Platform, Enterprise Edition (Java EE), that includes support for data validation, user identity management, and persistence fits very well with the AJAX methodology.

For more information on what Ajax is and how it works see the article "Asynchronous JavaScript Technology and XML (AJAX) With the Java Platform". You can find additional learning resources at the AJAX Developer Resource Center.

Adding Ajax to a Web Page

You can add Ajax functionality to a web page in many ways. In general, these approaches vary in the amount of JavaScript code you need to incorporate into the page. At one end of the spectrum, you provide all the JavaScript code required to enable Ajax in the page. At the other end of the spectrum, you take advantage of techniques that allow you to build Ajax into a web page with little or no JavaScript coding. You can also combine approaches.

Let's first look at one of the approaches that requires little or no JavaScript coding -- using an Ajax-enabled component from a component framework.

Using Ajax-Enabled Components

Perhaps the easiest way to incorporate Ajax into a web page is to add an Ajax-enabled component from a component framework such as the JavaServer Faces technology framework. Other component frameworks exist, but many of them -- such as the Shale Framework, ICEfaces, Ajax4jsf, and the JSF Extensions project -- are built on top of JavaServer Faces technology. So let's focus on the JavaServer Faces technology framework.

JavaServer Faces technology, often referred to as JSF, allows you to build functionally rich web pages for web applications using UI components, such as those for labels, buttons, text fields, and scrollbars. These components run on a server and are rendered to the page. The components can be enabled to respond to events such as a user clicking a button or entering characters in a text field, and to convert and validate input data.

Component developers find the technology particularly powerful because it contains a set of application programming interfaces (APIs) for representing UI components and for managing their state. The ability to manage state is important because it provides a way to synchronize the properties of a UI component between the server and client. This synchronization makes it possible to perform multiphase operations on a UI component such as converting and validating input data for the component prior to handling an event.

> Project Dynamic Faces, the runtime part of the JSF Extensions project, allows you to Ajax-enable standard JavaServer Faces components as you use them in a web page. For a description of Project Dynamic Faces, see the article "New Technologies for Ajax and Web Application Development: Project Dynamic Faces".

JavaServer Faces technology includes standard libraries for a basic set of UI components. However, the component model is extensible so that a component developer can add custom components. This extensibility allows a component developer to create custom components that have Ajax functionality or add Ajax functionality to existing components.

An Autocompletion Component

After creating an Ajax-enabled custom component, the component developer adds it to a JavaServer Faces component library. A number of libraries of Ajax-enabled custom JavaServer Faces components are available. For example, a library of Ajax-enabled components is available as part of the Java Blueprints Solutions Catalog. One of the components in that library is an autocompletion component. Let's examine this component in action and see how you can add it to a web page.

The autocompletion component is designed to work with a form. When a user types into a text field that is

associated with the component, it triggers a JavaScript function -- an event handler -- that suggests characters that complete the entry. For example, a demonstration of the autocompletion component in the Java Blueprints Solutions Catalog shows it in use with a name and address form. When a user types one or more characters in the City or State field in the form, the autocompletion component triggers an event handler that displays a pop-up list of cities or states beginning with those characters, as Figure 2 shows.

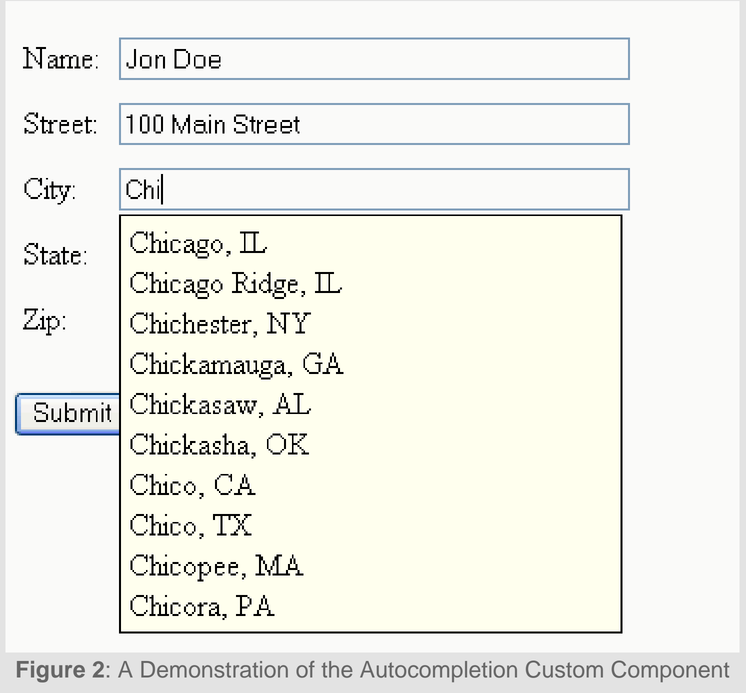


Figure 2: A Demonstration of the Autocompletion Custom Component

When the user selects a city in the list, the action triggers an event handler that fills in the city, state, and zip code for the selection. Note that the autocompletion actions are performed asynchronously. The application displays a suggested list of cities and fills in the city, state, and zip code without needing to refresh the page.

Including the Autocompletion Component in a Page

You can use an Ajax-enabled custom component just as you would any other JavaServer Faces custom component. You can express a component using a JavaServer Pages technology (JSP) tag on a JSP page, although you're not limited to using JSP technology to display the page. For example, you can express a component in an XML document. In addition, JavaServer Faces components are designed to be easily importable into integrated development environments (IDEs) such as the Sun Java Studio Creator IDE or the NetBeans IDE. If an Ajax-enabled component is available in an IDE, you can drag and drop the component from the IDE's palette onto the page.

> You can use an Ajax-enabled custom component just as you would any other JavaServer Faces custom component. You can express a component using a JSP tag on a JSP page. Or, if an Ajax-enabled component is available in an IDE, you can drag and drop the component from the IDE's palette onto the page.

The JSP Tag Approach

To use a custom component such as the autocompletion component in a JSP page, you add the following to the code for the JSP page:

- A reference to the custom component's tag library
- The tag for the component

Following is part of the code for the JSP page shown in Figure 2. The code snippet highlights (1) the reference to the custom component's tag library and (2) the tag for the component. You can view the entire JSP page here.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
(1) <%@taglib prefix="ui" uri="http://java.sun.com/blueprints/ui/14"%>
...

<f:view>
...

<h:form id="autofillform">
   <h:panelGrid cellpadding="5" cellspacing="0" columns="3" style="margin-bottom: 20px">
   ...

      <h:outputText value="City:" />
                 (2) <ui:autoComplete size="40" maxlength="100" id="cityField"
                      completionMethod="#{ApplicationBean.completeCity}"
                      value="#{SessionBean.city}" required="true"
                      ondisplay="function(item) { return extractCity(item); }"
                      onchoose="function(item) { return chooseCity(item); }"
                 />
       ...


          <h:outputText value="State:" />
                 (2) <ui:autoComplete size="2" maxlength="100" id="stateField"
                     completionMethod="#{ApplicationBean.completeState}"
                     value="#{SessionBean.state}" required="true" />

   ...
   </h:panelGrid>
...
</f:view>
```

Click here for a full-size version of the code example

In the code snippet, `<ui:autocomplete>` is the tag for the custom component that provides the autocompletion functionality. The tag attributes and attribute values of a custom component depend on the component's design. One advantage of using JavaServer Faces technology is that it allows you to use one or more components on a web page without having to know the details of how the component is implemented. However, you might want to know what the tag attributes and values for the component mean. In this example,

- The `completionMethod` attribute references methods in a managed bean, `ApplicationBean`, that is bound to the custom component. A *managed* bean is a JavaBeans technology component that is managed by the JavaServer Faces framework. The first specification of the `completionMethod` attribute references the `completeCity` method. This method returns a list of cities that begin with the characters that the user enters. It also returns the state and zip code for the city. The second specification of the `completionMethod` attribute references the `completeState` method, which returns a list of states that begin with the characters that the user enters.
- The `value` attribute holds the current value of the city or state, which is bound to another managed bean, `SessionBean`.
- The `ondisplay` attribute identifies an event handler that strips out the zip code from the data returned by the server, so that only the city and state are displayed in the pop-up list.
- The `onchoose` attribute identifies an event handler that uses the data returned by the server to fill in the state, and zip code when the user selects a city.

As the page author, you need to get from the component developer the name, attributes, and attribute values of any custom component-related tags that will be used on the page.

You might ask where the link is to the event handler that processes the user's entry into the city or state field. The answer is that it's rendered by the custom component, saving you the trouble of having to code it. The custom component renders an `<input>` tag, which is incorporated into the HTML for the page. For example, here's the `<input>` tag that the custom component renders for the state field:

```
<input
  id="autofillform:stateField" autocomplete="off" type="text"
  name="autofillform:stateField" size="2"
  onfocus="doCompletion('autofillform:stateField','menu-popup1','#{ApplicationBean.completeState}',null,null);"
  onkeyup="doCompletion('autofillform:stateField','menu-popup1','#{ApplicationBean.completeState}',null,null);"
  onblur="stopCompletionDelayed('menu-popup1');"
/>
```

Click here for a full-size version of the code example

The JavaScript attributes, `onfocus`, `onkeyup`, and `onblur`, link events on the State field to event handlers. For example, the `onkeyup` attribute triggers the JavaScript function `doCompletion` every time a user enters a character in the State field. The `doCompletion` function makes an asynchronous call to the method `completeState` in the bean `ApplicationBean` to return a list of states that begin with the characters that the user enters.

What's important here is that you don't have to provide any JavaScript code to get and display the pop-up list of suggested cities or states or to process a selection from the list. Neither do you need to provide CSS to specify the presentation style of the pop-up list. The custom component encapsulates the code required for those actions.

The IDE Approach

If an Ajax-enabled component is available through an IDE, you can incorporate it into a page by dragging it from the IDE's palette and dropping it on the page. For example, the autocompletion component is available in the Sun Java Studio Creator IDE. To use the component in a web page, you open the page in Sun Java Studio Creator's Visual Designer. Then you simply drag the component from the Sun Java Studio Creator palette and drop it on an appropriate place in the page such as next to a Label component. The article "Using the AJAX Text Completion Component" describes how to create a number of different web applications using the autocompletion component in the Sun Java Studio Creator IDE. In one application, the autocompletion component is used in a way similar to the address form completion example in Figure 2. However, instead of completing the name of a city or state, the component completes a word, as Figure 3 shows. A dictionary web service that comes with the Sun Java Studio Creator IDE supplies the suggested words.

Ajax-enabled components are also available in the NetBeans IDE 5.5 with the Visual Web Pack add-on. The Visual Web Pack is ideal for people who use the NetBeans IDE to develop and consume web services and enterprise beans in web applications. It allows developers to take advantage of the same powerful graphical user interface development capabilities that are in the Sun Java Studio Creator IDE while only using one IDE: NetBeans. For example, using the Visual Web Pack, you can add Ajax-enabled components to a web page simply by dragging them from within the NetBeans IDE palette and dropping them on the Visual Designer.

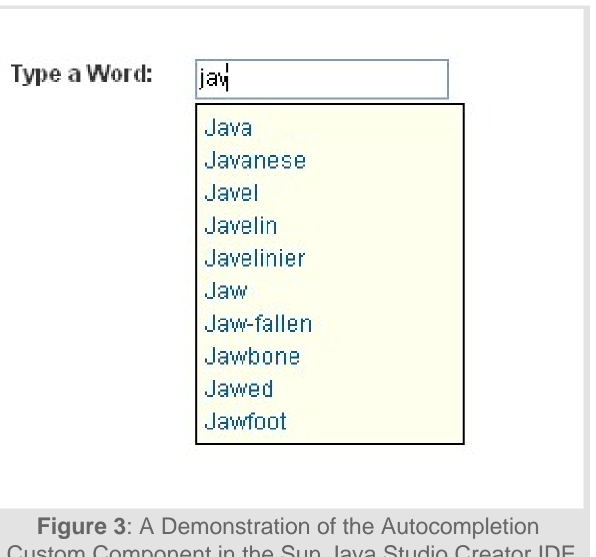


Figure 3: A Demonstration of the Autocompletion Custom Component in the Sun Java Studio Creator IDE

Assuming that the dictionary service is available in the IDE -- the article "Using the AJAX Text Completion Component" describes how to make the service available in the IDE -- all you need to do to build the page and include the word-completion functionality is the following:

- Drag a Label component from the Basic section of the Components palette and drop it on the page, as Figure 4 shows. You can then set the text property for the Label component as appropriate.

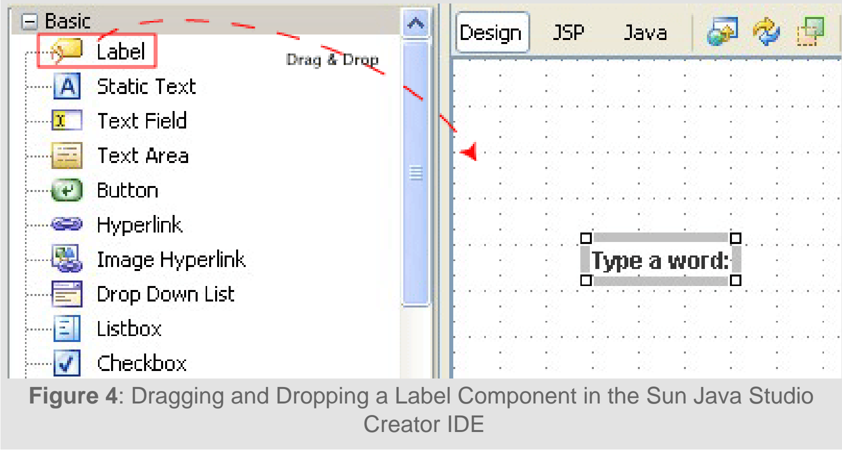


Figure 4: Dragging and Dropping a Label Component in the Sun Java Studio Creator IDE

- From the BluePrints AJAX Components section of the palette, drag the Auto Complete Text Field component and drop it on the page next to the Label component, as Figure 5 shows.

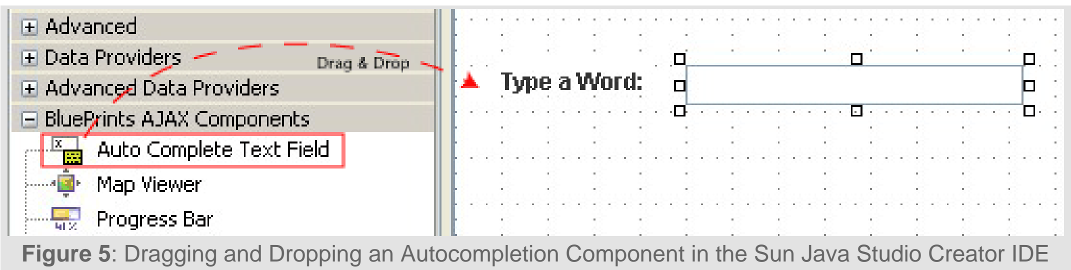


Figure 5: Dragging and Dropping an Autocompletion Component in the Sun Java Studio Creator IDE

As is the case for the autocompletion component shown in Figure 2, a method in a bean returns the contents for the pop-up list. In the example of the component in the Sun Java Studio Creator IDE, the bean calls the dictionary web service. The component developer should provide in the component the code that accesses the dictionary service. Alternatively, you might need to edit the code in the component, as appropriate. The article "Using the AJAX Text Completion Component" shows different examples that use the autocompletion component and shows the code to add or change in the component for each example.

Using Ajax-Enabled Widgets

Another way to add Ajax functionality to a web page is to use widgets that are enabled for Ajax. A *widget* is a prebuilt UI component that you can plug into a web application and customize as needed. Sounds a lot like a JavaServer Faces technology component, right? In fact, the UI components in the JavaServer Faces component libraries are widgets, and they include widgets that are enabled for Ajax. However, there are ways other than JavaServer Faces technology to access and use Ajax-enabled widgets. One way you can get Ajax-enabled widgets is from JavaScript libraries such as those in the Dojo toolkit or script.aculo.us. To include a widget from a JavaScript library in a web page, you need to include some JavaScript code in the page.

> You can select the Ajax-enabled widgets you need from one or more JavaScript libraries and use them in the same web page.

A Dojo Widget

Let's look at an example of a widget from a JavaScript library. One of the widgets available in the Dojo toolkit is a combobox widget. The widget is similar in operation to the autocompletion components shown previously. The widget displays a drop-down menu of choices. As a user enters text in a text field, the choices are dynamically reduced to those that match the entered text. For example, Figure 6 shows a page in which the combobox widget is used to select a U.S. state. The example is taken from the demonstration of the combobox widget on the Dojo toolkit site.

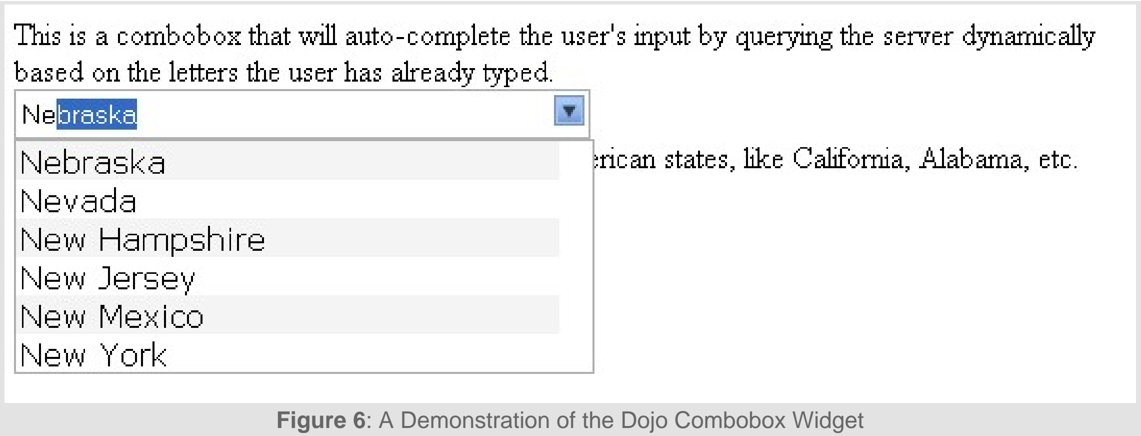


Figure 6: A Demonstration of the Dojo Combobox Widget

Including the Dojo Combobox Widget in a Page

The Dojo toolkit contains various JavaScript libraries, each of which is organized in packages. One of the libraries contains the JavaScript functions needed for the Dojo widgets. Each package in the library is for a specific widget. To include a Dojo widget on a page, you need to (1) import the package for the widget using the JavaScript function `dojo.require()` and (2) include a reference to the widget in the appropriate place on the page. The following example highlights these two steps and shows the code for the page used in the combobox demonstration.

```
<html>
<head>
<title>Dojo ComboBox Widget Demo</title>

<script type="text/javascript" src="../../dojo.js"></script>
<script type="text/javascript">
(1) dojo.require("dojo.widget.ComboBox");
</script>
</head>

<body>
This is a combobox that will auto-complete the user's input by querying the server dynamically
based on the letters the user has already typed.
(2) <select dojoType="ComboBox" value="this should never be seen - it is replaced!"
        dataUrl="../../tests/widget/comboBoxData.js" style="width: 300px;"
        name="combo.box1" maxListLength="15">
</select>
Try starting to type the name of one of the 50 American states, like California, Alabama, etc.

</body>
</html>
```

Click here for a full-size version of the code example

The `dojo.require()` function dynamically loads the JavaScript code for the specified library into the page. The `dojoType` attribute references the combobox widget. In this example, the reference is in an HTML `<select>` tag for the drop-down list. The `dataURL` parameter points to a file that contains the state names for the combobox.

Notice the reference to the `dojo.js` file in the `<script>` tag in the header area of the code. This file, called the Dojo bootstrap file, contains the basic JavaScript functions provided by the Dojo toolkit and is required in all pages that use Dojo widgets. The Dojo bootstrap file is in the Dojo toolkit distribution, which is available on the Dojo toolkit site. The application developer needs to copy from the distribution the Dojo bootstrap file as well as any other needed Dojo library files and place them in the root directory of the web application structure or a subdirectory of the root directory.

Note that the Dojo toolkit provides not only the JavaScript code for a widget but also the HTML and CSS for the presentation of the widget on a page.

Using jMaki Widgets

One of the issues in using widgets from a JavaScript library is that it requires page authors to do some JavaScript coding. For example, to use the Dojo combobox widget, you need to write JavaScript code to call the `dojo.require()` function and to include the Dojo bootstrap file. If you're not accustomed to writing JavaScript code, you might prefer an alternative approach to using these widgets. The jMaki framework provides just such an alternative.

jMaki, pronounced jay-MAH-kee, is a framework for wrapping JavaScript widgets in JSP custom tag handlers or JavaServer Face components. The name jMaki cleverly represents the concept of wrapping things within Java technologies. The *j* in jMaki stands for Java technology, and *Maki* is the Japanese word for wrap. The framework provides a library of wrapped widgets from various JavaScript libraries such as the Dojo toolkit and script.aculo.us. You can view these widgets in the jMaki Widget Gallery. The jMaki framework also provides a way for developers to add to the set of jMaki widgets.

jMaki simplifies things for page authors. You can add a jMaki widget to a web page without doing any JavaScript coding. You include a jMaki widget in a web page in the same way as you include a JSP custom tag or JavaServer Faces component. If you use JSP technology or JavaServer Faces technology to develop web pages, jMaki allows you to access and use widgets in a familiar way. And because jMaki wraps widgets in JSP tag handlers or JavaServer Faces components, it extends the benefits of JSP and JavaServer Faces technology to the widgets. For example, a jMaki widget wrapped in a JavaServer Faces component can be customized to trigger event handlers and validate user entries. In addition, jMaki makes the JavaScript code, HTML, and CSS for a widget accessible so that you can easily customize it.

Recently jMaki added support for the PHP scripting language and Project Phobos, a new scripting framework that enables a blending of scripting language and Java technology. In addition to accessing a jMaki widget as a JSP custom tag or JavaServer Faces component, you can now access it using PHP code, or use the jMaki library in Project Phobos. For further information see the article "New Technologies for Ajax and Web Application Development: Project Phobos". Also see the blog entry "jMaki supports PHP!" and the blog entry "jMaki in Phobos".

A jMaki Autocompleter Widget

Let's look at an example of a jMaki widget. One of the widgets distributed with jMaki is a combobox widget. It's the wrapped version of the combobox widget provided in the Dojo toolkit. As Figure 7 shows, the jMaki combobox widget works exactly the same way as the Dojo combobox widget.

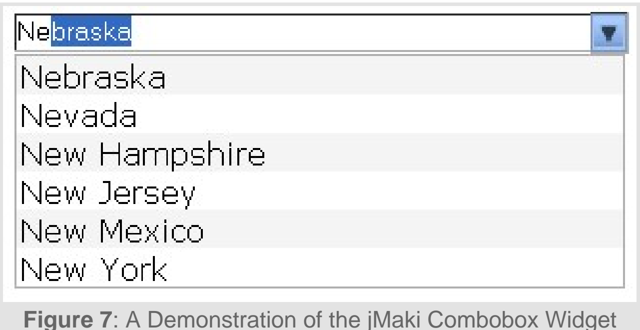


Figure 7: A Demonstration of the jMaki Combobox Widget

You can combine jMaki with other technologies such as JavaServer Faces technology or Project Dynamic Faces to add interesting Ajax-enabled features to a web page. To see some examples, see Project Dynamic Faces and jMaki. Examine the JSP files for the examples to see how to combine these technologies on a web page.

Including the jMaki Combobox Widget in a Page

You can include a jMaki widget on a web page as either a JSP custom tag or a custom JavaServer Faces component. In either case, you reference the appropriate jMaki tag library. If you want to include a widget as a JSP custom tag, you specify the following jMaki tag library:

```
<%@taglib prefix="a" uri="http://java.sun.com/jmaki"%>
```

If you want to include a widget as a JavaServer Faces component, you specify the following jMaki tag library:

```
<%@taglib prefix="a" uri="http://java.sun.com/jmaki-jsf"%>
```

Then you specify the tag for the widget in the appropriate place on the page. For example, here's an example of the jMaki combobox widget used as a custom JSP tag.

```
<%@ taglib prefix="a" uri="http://java.sun.com/jmaki" %>

<html>
...

<h2>jMaki Combobox</h2>
...
<a:ajax id="cb1" name="dojo.combobox" service="comboBoxData.js" />

...

</html>
```

Here's an example of the jMaki combobox widget used as a JavaServer Faces component.

```
<%@ taglib prefix="a" uri="http://java.sun.com/jmaki-jsf" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

<html>
...

<h2>jMaki Combobox</h2>
...


<f:view>
    <a:ajax id="cb1" name="dojo.combobox" service="comboBoxData.js" />
</f:view>
...


</html>
```

The `<a:ajax>` tag is the custom JSP tag for jMaki widgets. The `name` attribute of the custom JSP tag identifies the widget, in this case, `dojo.combobox`, the combobox widget distributed with jMaki. jMaki provides the basic resources required by the widget, including the JavaScript code that enables Ajax for the widget. However, if the widget needs any additional resources, these are referenced in the tag's `service` attribute. In this example, the `service` attribute points to a file that accesses the state names. However, the `service` attribute could point to a JSP file, a servlet, or a managed bean for the state names.

Whether a jMaki widget is included as a JSP custom tag or a custom JavaServer Faces component, you need to get from the application developer the name, attributes, and attribute values for the widget's JSP tag. You can also find the names of the widgets distributed with jMaki by looking at the source code for the widget in the jMaki Widget Gallery.

One other thing to note: A web application that uses jMaki widgets must include the contents of the jMaki distribution package. Further, the application must place some of the jMaki files in specific locations within the application directory structure. The application developer, not the page author, needs to take these actions. However, if you would like to learn more about what steps are required to set up an application for jMaki, see the article "New Technologies for Ajax and Web Application Development: Project jMaki".

Learn more about jMaki in the article "New Technologies for Ajax and Web Application Development: Project jMaki".

Using jMaki in the NetBeans IDE

jMaki widgets are also available in the NetBeans IDE through the jMaki NetBeans plug-in. If the plug-in is installed, you can incorporate one or more jMaki widgets in a web page by opening the source file for the page in the IDE, dragging the widgets from the jMaki palette, and then dropping them at the appropriate points in the source file. The appropriate tag library references are added to the source code for the page, and the custom tags for the widgets are added at the point of the drop.

For example, Figure 8 shows what happens when you drag and drop the combobox widget from the jMaki palette in the NetBeans IDE to the source file of a web page.

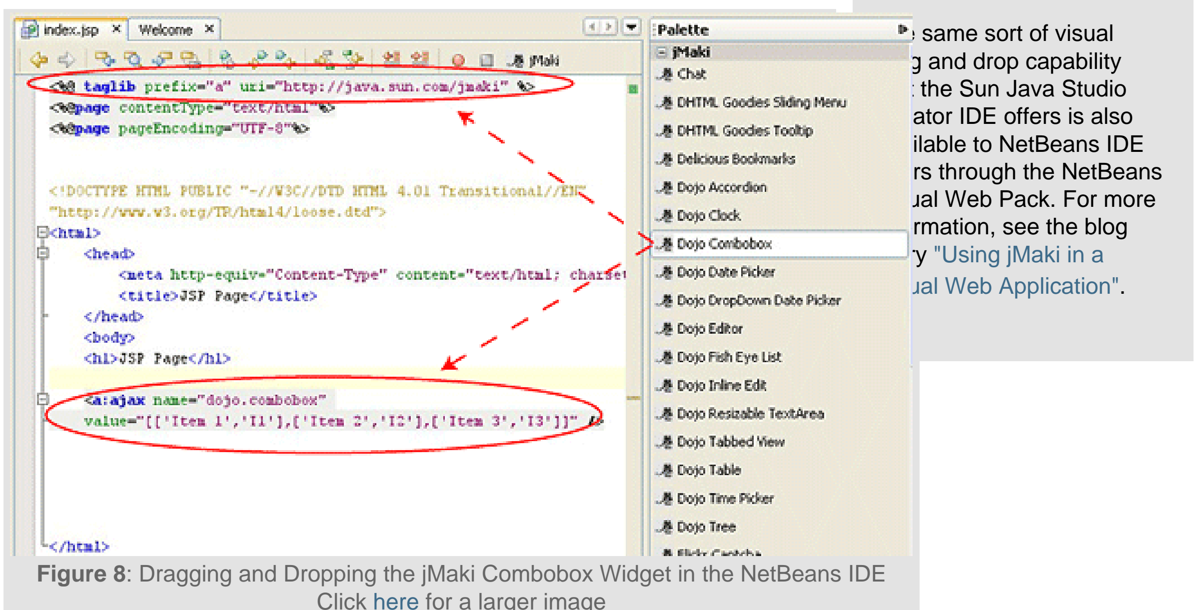


Figure 8: Dragging and Dropping the jMaki Combobox Widget in the NetBeans IDE
Click here for a larger image

> same sort of visual
> g and drop capability
> the Sun Java Studio
> ator IDE offers is also
> ilable to NetBeans IDE
> rs through the NetBeans
> ual Web Pack. For more
> rmation, see the blog
> y "Using jMaki in a
> ual Web Application".

Here the widget is dropped after the `<h1>` tag. In response, an `<a:ajax>` tag is generated at the point of the drop. In addition, a reference to the jMaki tag library is generated in the header for the page, unless it already appears in the header. The `value` attribute in the custom tag for the widget contains the data for the selection choices, in this case, `Item 1`, `Item 2`, and `Item 3`. To use a different set of selection choices, you need to change the code for the `<a:ajax>` tag in the page.

> You can also add jMaki widgets to the Sun Java Studio Creator IDE. To learn how, see the blog entry "Giving jMaki a Whirl".

Writing the JavaScript Code Yourself

If you don't find components or widgets that provide the Ajax functionality you need, you'll need to do the JavaScript coding yourself. Let's look at an example.

> You might need to do some of the JavaScript coding for enabling Ajax on a web page. Or you might need to supplement another approach with some JavaScript coding. Remember that you can use various approaches in combination.

Writing the JavaScript Code for Autocompletion

Imagine that no autocompletion component or autocompletion widget exists. Let's examine how you could use JavaScript code to add autocompletion functionality to a web page. In the following steps, you'll add JavaScript code to an HTML page. The JavaScript code is designed to interact with a servlet on the server to handle autocompletion. You can also find a description of this example in the document "Auto-Completion With AJAX: Design Details" in the Java Blueprints Solutions Catalog.

What You Need to Code

As you might remember from Figure 1, here's what you need to do in the web page to provide Ajax functionality:

1. Map an event to a JavaScript function.
2. Create and configure an `XMLHttpRequest` object for Mozilla browsers or an `XMLHTTP` object for Internet Explorer.
3. Make a request to the server through the `XMLHttpRequest` or `XMLHTTP` object.
4. Process the result from the server -- typically an XML document -- in a JavaScript callback function.
5. Update the internal representation of the page with the new data.

Let's look at what you need to code for each of these steps.

1. Map an Event to a JavaScript Function

In the HTML page, you first construct a text field and then map the entries in the text field to a JavaScript function that handles the entries.

```
<input type="text"
        size="20"
        id="complete-field"
        name="id"
        onkeyup="complete();">
```

The `onkeyup` attribute of the `<input>` tag specifies a JavaScript function, `complete()`. When a user types into the input field, the entry fires the `complete()` function to handle the event and passes to the function the input field `complete-field`.

2. Create and Configure an `XMLHttpRequest` Object or an `XMLHTTP` Object

In the header of the HTML file, you add the JavaScript code for the `complete()` function. Start by adding code that checks whether the browser in use is a Mozilla browser or Internet Explorer and creates the appropriate HTTP communication object depending on the browser type.

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

function complete() {
    if (window.XMLHttpRequest) {
            req = new XMLHttpRequest();
    }
    else if (window.ActiveXObject) {
            req = new ActiveXObject(
            "Microsoft.XMLHTTP");
    }
```

3. Make a Request to the Server Through the `XMLHttpRequest` Object or `XMLHTTP` Object

To the `complete()` function, add code that uses the appropriate HTTP object to make a request to the server. The entire `complete()` function looks like this:

```
function complete() {
    if (window.XMLHttpRequest) {
            req = new XMLHttpRequest();
    }
    else if (window.ActiveXObject) {
            req = new ActiveXObject(
            "Microsoft.XMLHTTP");
    }

    var url = "autocomplete?action=complete&id=";
    initRequest(url);
    req.onreadystatechange = processRequest;
    req.open("GET", url, true);
    req.send(null);
}
```

After initializing the `XMLHttpRequest` or `XMLHTTP` object, `req`, the `complete()` function makes a request to the server with the statements in Table 1.

Table 1. Statements Used to Communicate with a Server Through an `XMLHttpRequest` Object or `XMLHTTP` Object

Statement	Description

`var url = "autocomplete?action=complete&id=";`	Sets the URL for the communication between the `req` object and the servlet. The URL is set to `autocomplete`, which is mapped -- in another file -- to a servlet. The statement adds the value from the `complete-field` text field as a URL parameter.
`initRequest(url);`	Creates the request.
`req.onreadystatechange = processRequest;`	Specifies the callback function `processRequest()` to process the data returned by the servlet.
`req.open("GET" url, true);`	Specifies that the `req` object will use an HTTP `GET` method to communicate with the servlet. The `url` parameter identifies the servlet's URL. The parameter `true` indicates that the communication will be asynchronous.
`req.send(null);`	Initiates the asynchronous call to the servlet. The call passes the `complete-field` in the request.

4. Process the Result From the Server in a JavaScript Callback Function

The request to the server is processed by a servlet. The servlet handles the request and returns the data for the autocompletion, for example, a list of states that begin with the characters that the user entered. The data is returned in the `XMLHttpRequest` or `XMLHTTP` object and is typically in the form of an XML document. Let's assume that an XML document is returned in this example. The callback function `processRequest()` processes the data in the XML document. Here is the code for the `processRequest()` function:

```
function processRequest() {
    if (req.readyState == 4) {
        if (req.status == 200) {
          parseMessages();
        } else {
            clearTable();
        }
    }
}
```

Both `readyState` and `status` are properties of the `XMLHttpRequest` or `XMLHTTP` object. The `readyState` property identifies the state of the request. A `readyState` value of `4` indicates that the request has completed. The `status` property specifies a status code returned from the server. A code of `200` means that the request was performed successfully. The `ProcessRequest` function calls the `parseMessages()` function if the request to the server was completed and performed successfully. Otherwise, the `ProcessRequest` function calls the `clearTable()` function, which prevents the autocompletion items from being displayed.

5. Update the Internal Representation of the Page With the New Data

The JavaScript function `parseMessages()` processes the data in the XML document and updates the internal representation of the web page based on the data:

```
function parseMessages() {
    if (!names) names = document.getElementById("names");
    clearTable();
    var employees = req.responseXML.getElementsByTagName("employees")[0];
    if (employees.childNodes.length > 0) {
        var autorow = document.getElementById("menu-popup");
        autorow.style.visibility = "visible";
    }
    for (loop = 0; loop < employees.childNodes.length; loop++) {
        var employee = employees.childNodes[loop];
        var firstName = employee.getElementsByTagName("firstName")[0];
        var lastName = employee.getElementsByTagName("lastName")[0];
        var employeeId = employee.getElementsByTagName("id")[0];
        appendEmployee(firstName.childNodes[0].nodeValue,lastName.childNodes[0].nodeValue,
                       employeeId.childNodes[0].nodeValue);
    }
}
```

Click here for a full-size version of the code example

The web page is represented internally as a tree structure through the Document Object Model (DOM). The `parseMessages()` function uses JavaScript functions such as `getElementsByTagName()` to access and manipulate elements in the DOM representation of the page.

Including JavaScript Code From a File

Although the page author can do all the JavaScript coding to enable Ajax in a web page, it's more typical that the application developer will do the JavaScript coding and put it in a separate file. In fact, it's best to cleanly separate the content of a web application from the JavaScript code in an Ajax-enabled application. If the JavaScript code is in a separate file, all you need to do is reference the file in a `<script>` tag within the header of your web page. For example, if the JavaScript code is in a file named `autocomplete.js`, put the following `<script>` tag in the header:

It's best to cleanly separate the content of a web application from the JavaScript code and CSS in an Ajax-enabled application.

```
<script type="text/javascript" src="autocomplete.js"><script>
```

In response, the JavaScript code in the file is included in the code for the page. If all the JavaScript code is in a separate file, the only other thing you need to do to add Ajax functionality to a web page is map the pertinent event to a JavaScript function as described in step 1.

Specifying Presentation Style

Ajax-enabled components and widgets provide the CSS that defines their presentation style. If you don't use Ajax-enabled components or widgets or if their presentation style is not what you want, you'll need to code the CSS yourself.

For example, here is the CSS for the pop-up list in Figure 2 that is generated by the autocompletion component in the Java BluePrints Solutions Catalog:

```
.popupItem {
    background: #ffffee;
    color: #000000;
    text-decoration: none;
    display: block;
    padding: 1px;
}

.popupItem:hover {
    background: #7a8aff;
    color: #fffafa;
}

.popupFrame {
    background: #ffffee;
```

```
    border: solid 1px black;
    padding: 3px;
    overflow: hidden;
    z-index: 50000;
}
```

If you want the pop-up list to display in the same presentation style as shown in Figure 2, you need to either use the autocompletion custom component, use a widget that provides the same CSS, or code the same CSS yourself. If you want the pop-up list to display in a different presentation style, you need to code the CSS accordingly. You can provide the CSS in the web page or in a separate file. If the CSS is in a separate file, you reference the file in a `<link>` tag within the header of your web page. For example, if the CSS is in a file named `styles.cs`, put the following `<link>` tag in the header:

```
<link type="text/css" rel="stylesheet" href="../popup.css" />
```

Using JavaScript Libraries

One of the approaches mentioned earlier in this article uses widgets from a JavaScript library such as those in the Dojo toolkit or script.aculo.us. However, these libraries provide more than widgets. If you do the JavaScript coding to add Ajax functionality to a web page, you can take advantage of additional features in JavaScript libraries to simplify your coding. For example, the `dojo.io` library in the Dojo toolkit abstracts Ajax-related communication with the server and so hides low-level `XMLHttpRequest` or `XMLHTTP` operations. Using the `dojo.io` library, here's what the `complete()` function in the do-it-yourself version of the autocompletion example would look like:

```
dojo.require("dojo.io.*");

    // Retrieve data through a Dojo call.
    var bindArgs = {

        url: "autocomplete?action=complete&id="+key,
        mimetype: "text/xml",
        load: processRequest};

    // Dispatch the request.
    req=dojo.io.bind(bindArgs);
```

The `dojo.require()` function dynamically loads the JavaScript code for the `dojo.io` library. The updated `complete()` function uses the `dojo.io.bind()` method to make an asynchronous request to the server. The `bind()` method wraps the `XMLHttpRequest` or `XMLHTTP` object, so you don't need to create and configure either of these objects. Instead, you provide the following parameters to the `bind()` method:

- The URL of the server-side component to communicate with
- The format of the response
- The identification of the callback function

Other libraries in the Dojo toolkit provide a variety of functions that simplify the code for things such as DOM manipulation and event handling.

Prototype is another popular source for JavaScript functions that can simplify the Ajax-enabling code in a web page. Prototype is a JavaScript framework that provides a library whose components include objects that simplify the use of JavaScript. One component presents an Ajax object that, like the `dojo.io` library in the Dojo toolkit, encapsulates an `XMLHttpRequest` or `XMLHTTP` object and hides low-level `XMLHttpRequest` or `XMLHTTP` operations. Another component presents objects and methods that make it easier to work with DOM elements.

Script.aculo.us and Rico are built on top of Prototype. Both provide JavaScript libraries that support Ajax as well as other functions that can be plugged into a web application. You can take advantage of additional JavaScript libraries -- see Survey of AJAX/JavaScript Libraries for an extensive list.

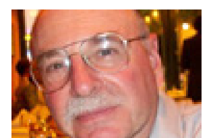
Summary

You can add Ajax functionality to a web page in many ways. In general, these approaches vary in the amount of JavaScript code you need to incorporate into the page. Some approaches, such as using Ajax-enabled JavaServer Faces components, encapsulate the JavaScript code in the component, so you don't have to do any JavaScript coding. Other approaches, such as using widgets from the Dojo toolkit, provide some of the JavaScript code. In still other approaches, you do most or all of the JavaScript coding. Choose the approach or combination of approaches that best fits the functional requirements of your web application and with which you're most comfortable.

For More Information

- AJAX Developer Resource Center
- Asynchronous JavaScript Technology and XML (AJAX) With the Java Platform
- AJAX Design Strategies
- Java EE 5 SDK
- JavaServer Faces Technology
- JSF Extensions Project
- Project jMaki
- Java BluePrints Solutions Catalog
- Sun Java Studio Creator IDE
- AJAX Components in the Java Studio Creator IDE
- NetBeans IDE 5.5
- NetBeans Visual Web Pack 5.5
- New Technologies for Ajax and Web Application Development: Project jMaki
- New Technologies for Ajax and Web Application Development: Project Dynamic Faces
- New Technologies for Ajax and Web Application Development: Project Phobos
- Survey of AJAX/JavaScript Libraries

About The Author



Ed Ort is a is a staff member of java.sun.com. He has written extensively about relational database technology, programming languages, and web services.

Copyright Sun Microsystems, Inc.

Unless otherwise licensed, code in all technical manuals herein (including articles, FAQs, samples) is provided under this License.

XML Content Feeds