

Article

AJAX Design Strategies

By Ed Ort and Mark Basler, October 2006

Contents

- Overview
- What is AJAX?
- AJAX Technologies
- AJAX and Server-Side Java Technologies
- Design Strategies
- Design Strategy 1: Do It Yourself
- Design Strategy 2: Use a Client-Side JavaScript Technology Library
- Design Strategy 3: Use a Client-Server Framework
- Design Strategy 4: Do the Wrap Thing
- Design Strategy 5: Go Remote
- Design Strategy 6: Go All Java Technology
- Summary
- For More Information
- About the Authors



Download PDF

Get AJAX Training

Get AJAX training in these new courses:

- Developing JavaServer Faces Web Applications With AJAX Using Sun Studio Creator (DTJ-2105)
- Developing JavaServer Faces Components With AJAX (DTJ-3108)

Use your SDN member priority code WW26SDN and get a 10% training discount.

Overview

Web applications have entered a new era driven by web site goals such as fast response to user actions and user collaboration in creating and sharing web site content. The popular term attributed to these highly responsive and often collaborative sites is [Web 2.0](#).



Some prime examples of Web 2.0 are web sites such as [Google Maps](#) and [Flickr](#). Google Maps offers a highly responsive user interface (UI). For instance, you can view a map, then move your cursor across it to see adjacent areas almost immediately. Flickr is a site on which users store and share photographs -- users manage almost all the site's content.

Other Web 2.0 sites provide a similarly rich user experience by doing things such as integrating services from other web sites or incorporating a steady stream of new information. For example, the Google map service can be brought into another web site, such as a site for purchasing cars, to present a map that highlights the location of auto dealerships that sell a particular car model. The term used for these site integrations is "mashups." Or a sports-oriented site can continually update scores without requiring the user to request a page

update.

This article is about the primary technique in use today for making Web 2.0 sites highly responsive: Asynchronous JavaScript and XML (AJAX).

What is AJAX?

A number of excellent articles that describe AJAX are available, for example, [Asynchronous JavaScript Technology and XML \(AJAX\) With Java 2 Platform, Enterprise Edition](#). In brief, AJAX is a set of technologies that together allows a web site to be -- or appear to be -- highly responsive. AJAX enables this because it supports asynchronous and partial refreshes of a web page.

AJAX allows a web site to be -- or appear to be -- highly responsive 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 enters information into a form on a web page and clicks a Submit button -- the 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, as is the case for conventional, "click, wait, and refresh" web applications. Instead, the client then updates the page based on the response. Typically this means that only part of the page is updated. In other words, the web page is treated like a template: The client and the server exchange data, and the client updates parts of the template based on the data the client receives from the server. One way to think of this is that web applications that use AJAX are driven by events and data, whereas conventional web applications are driven by pages.

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. The client does not have to wait for a response from the server before continuing, as is the case for the traditional, synchronous, approach.

Another important aspect of AJAX-enabled sites is that the events that trigger AJAX responses aren't limited to submitting data in a form or clicking on a link. Moving a mouse over an area on a web page, typing part of an entry in a field, or -- as in the Google Maps case -- dragging a map around with a cursor can be enough to trigger an AJAX response. This dynamic interaction between the user and the web page moves web applications closer to what users experience in highly responsive desktop applications, often termed *rich desktop applications*. So Web 2.0 applications are often called *rich Internet applications*.

AJAX Technologies

The following technologies are typically included in AJAX:

- Cascading Style Sheets (CSS), a markup language for defining the presentation style of a page, such as fonts and colors.
- JavaScript, a scripting language. One element of JavaScript technology that is key to AJAX is `XMLHttpRequest`, an object that is used to exchange data between the web client and web server.
- Document Object Model (DOM), which provides a logical view of a web page as a tree structure.
- XML, the format for sending data from the web server to the client. However, you can use other formats, such as HTML, JavaScript Object Notation (JSON), or plain text.

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, server-side application systems play a key role in processing AJAX applications. 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. See the [AJAX and Server-Side Java Technologies](#) section of this article.

Figure 1 illustrates how these technologies work together to handle a user action that triggers an AJAX response

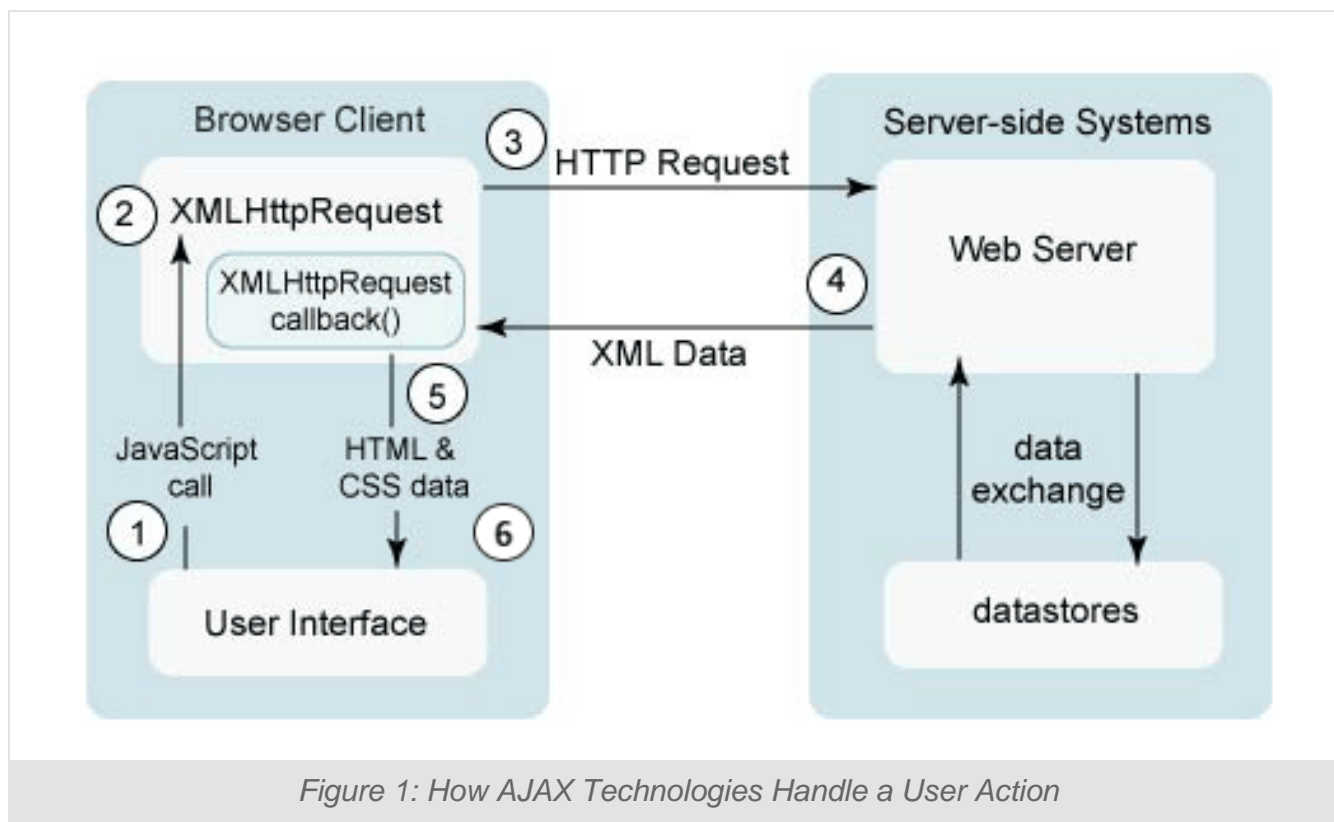


Figure 1: How AJAX Technologies Handle a User Action

1. A user generates an event on the client. This results in a JavaScript technology call.
2. A JavaScript technology function creates and configures an `XMLHttpRequest` object on the client, and specifies a JavaScript technology callback function.
3. The `XMLHttpRequest` object makes a call -- an asynchronous HTTP request -- to the web server.
4. The web server processes the request and returns an XML document that contains the result.
5. The `XMLHttpRequest` object calls the callback function and exposes the response from the web server so that the request can be processed.
6. The client updates the HTML DOM representing the page with the new data.

You can incorporate these technologies into a web application in a variety of ways. For example, you can write JavaScript technology code in an HTML page, or you can use a library such as one in the [Dojo toolkit](#) to provide part or all of the JavaScript technology functionality you need. In addition, you have server-related choices regarding AJAX.

AJAX and Server-Side Java Technologies

All the technologies included in AJAX are either client technologies such as JavaScript technology, or -- like XML -- are pertinent to the interchange of data between the client and server. The fact is that the request to the server that the `XMLHttpRequest` object makes is an HTTP request. To the server, handling an AJAX request is no

different than handling an HTTP request from a conventional web application -- any server technology can be used to handle the request, including server-side Java technologies in the Java EE Platform such as servlets, JSP technology, and JavaServer Faces technology. In fact, Java EE technologies fit very well with the AJAX methodology. JavaServer Faces technology and other Java EE technologies that include support for data validation, user identity management, and persistence, are particularly well-suited for AJAX.

Java EE technologies fit very well with the AJAX methodology.

For example, you can use a servlet to process a request, manage state for the client, access enterprise resources, and generate the XML for the response. Or you can create custom JavaServer Faces components to handle server-side processing as well as encapsulate the JavaScript technology and

CSS for client-side processing. In fact, a [library of AJAX-enabled custom JavaServer Faces components](#) is already available as part of the [Java Blueprints Solutions Catalog](#).

Design Strategies

As a developer, you have a variety of options in building AJAX into a web application. These options range from a complete "do-it-yourself" strategy, where you do all the AJAX coding, to strategies that take advantage of libraries and frameworks that provide some or all of the AJAX coding for you. And you can combine multiple strategies. For example, you might use a client-server framework based on JavaServer Faces technology in combination with a client-side JavaScript technology library such as a library in the Dojo toolkit.

The remainder of this article discusses some of these design strategies and identifies a few of their advantages and disadvantages. These strategies assume that you're using a Java EE implementation on the server, such as [Sun Java Application Server](#).

Design Strategy 1: Do It Yourself

In this approach you do all the coding to build AJAX into a web application. This means you do all the JavaScript technology, CSS, and DOM coding on the client, as well as the coding for page presentation. On the server, you do the coding to handle the `XMLHttpRequest` call, and to return an appropriate response, such as XML.

Let's look at an example that adds AJAX functionality to a web application. The example appears in the article "[Creating an AJAX-Enabled Bookstore Application, a Do-It-Yourself Approach](#)". That example is based on bookstore2, one of the Duke's Bookstore examples documented in the [Java EE 5 Tutorial](#). You can find the instructions for building the bookstore2 web application in [The Example JSP Pages section](#) of the Java EE 5 Tutorial.

The original bookstore2 web application lists books available for purchase. You can see what the list looks like in [here](#). The interaction between a user and the current bookstore2 application is a conventional synchronous interaction. A user clicks on a book title in the list to get further information about that book. This requires the server to do a full refresh of the page.

The example that appears in the "Creating an AJAX-Enabled Bookstore Application" article adds some AJAX functionality to enable pop-up balloons in the web application. This added functionality provides for a more dynamic UI. When a user moves the mouse cursor over a listed book, details about that book appear in a pop-up balloon, as [Figure 2](#) shows. Because it's driven by

Refer to the article "[Creating an AJAX-Enabled Bookstore Application, a Do-It-Yourself Approach](#)" to view the code for an AJAX-enabled version of bookstore2, one of the Duke's Bookstore examples documented in the [Java EE 5 Tutorial](#). The "Do-It-Yourself" article is part of a series of articles titled "[Hands-On Java EE5](#)". The series presents code that implements various approaches to enabling

AJAX, the interaction between the user and the web application to produce the pop-up balloon is asynchronous. The user sees the pop-up balloon almost instantaneously -- without waiting for the server to refresh the entire page.

AJAX in the bookstore2 application, including using the Dojo toolkit and using custom JavaServer

The screenshot shows a web browser window titled "Duke's Bookstore". The address bar shows "http://localhost:8080". The page content includes the "Duke's Bookstore" logo with a character holding a tray, the date "Wednesday, August 9, 2006 5:09:11 PM", and a "Book Detail" pop-up window. The pop-up displays details for "My Early Years: Growing up on *7" by Duke (1995), including a price of \$30.75 and an inventory of 20. Below the pop-up is a list of books with their prices and "Add to Cart" buttons.

Book Detail		
My Early Years: Growing up on *7 by Duke (1995)		
Web Site by Duke	Here's what the critics say: What a cool book.	
Web Site by Duke	Price: \$30.75	
Web Site by Duke	Inventory: 20	
From Oak to Java: The Revolution of a Language by Kevin Novation	\$10.75	Add to Cart
Java Intermediate Bytecodes by James Gosling	\$30.95	Add to Cart
The Green Project: Programming for Consumer Devices by Ben Thrilled	\$30.00	Add to Cart
Duke: A Biography of the Java Evangelist by Itzal Tru	\$45.00	Add to Cart

Copyright © 2003-2006 Sun Microsystems, Inc.

Figure 2: A Web Application With AJAX-Enabled Pop-up Balloons

Recalling the steps illustrated in Figure 1, here's what the AJAX-related code in the bookstore2 web application must do to produce the pop-up balloons in response to the user's act of moving the cursor over the title of book in the list:

1. Map the event to a JavaScript technology function.
2. Create and configure an `XMLHttpRequest` object.
3. Make a request to the server through the `XMLHttpRequest` object.
4. Process the request on the server and return an XML document that contains the result.
5. Process the result in a JavaScript technology `callback()` function.
6. Update the DOM representing the page with the new data.

Accompanying the "Creating an AJAX-Enabled Bookstore Application" article are the files for the AJAX-enabled bookstore2 application. The files are packaged as a NetBeans IDE 5.5 web application project. The article shows you how to open and run the project using the [NetBeans IDE 5.5](#). The article also shows you how to view the contents of the files. Using the NetBeans IDE 5.5, you can see how each of the steps just listed is implemented in code. The "Creating an AJAX-Enabled Bookstore Application" article focuses on this code.

You'll notice that a major part of that code is client-side JavaScript technology code. For example, here's the JavaScript technology code in the AJAX-enabled bookstore2 application that creates and configures an `XMLHttpRequest` object and uses it to make an asynchronous request to a server component:

```
bpui.alone.showPopupInternal=function(popupx, bookId) {

    bpui.alone.req=bpui.alone.initRequest();
    ...

    url="../PopupServlet?bookId=" + escape(bookId);
    bpui.alone.req.onreadystatechange = bpui.alone.ajaxReturnFunction;
    bpui.alone.req.open("GET", url, true);
    bpui.alone.req.send(null);
}

...

bpui.alone.initRequest=function() {
    if (window.XMLHttpRequest) {
        return new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        return new ActiveXObject("Microsoft.XMLHTTP");
    }
}
```

And here's the JavaScript technology code in the callback function that processes the data in the response to the asynchronous request. Notice that the callback function uses methods in the DOM API such as

`getElementsByTagName()` to extract data from the XML response. The `inner.HTML` property is used to update the DOM representation of the page.

```
bpui.alone.ajaxReturnFunction=function() {
    // statically setup popup for simple case
    var componentId="pop0";
    // check return of the call to make sure it is valid
    if (bpui.alone.req.readyState == 4) {
        if (bpui.alone.req.status == 200) {
            // get results and replace dom elements
            var resultx=bpui.alone.req.responseXML.getElementsByTagName("response")[0];
            document.getElementById(componentId + "_title").innerHTML=
                resultx.getElementsByTagName("title")[0].childNodes[0].nodeValue;
            document.getElementById(componentId + "_message").innerHTML=
                resultx.getElementsByTagName("message")[0].childNodes[0].nodeValue;;
            // show popup with the newly populated information
            document.getElementById(componentId).style.visibility='visible';
        } else if (bpui.alone.req.status == 204){
            alert("204 returned from AJAX call");
        }
    }
}
```

The AJAX-enabled bookstore2 application also needs to include code in the server-side component -- in this case a servlet -- to handle the `XMLHttpRequest` and return an appropriate response. The "Creating an AJAX-Enabled Bookstore Application" article examines the servlet code as well as the code in other files that are related to the pop-up balloons, such as a CSS file.

In short, taking the do-it-yourself approach requires writing a significant amount of AJAX-related code -- some of it fairly complex -- to implement the pop-up balloons in the bookstore2 application.

Pros and Cons of the Do-It-Yourself Approach

The do-it-yourself approach is certainly one way to build AJAX into a web application, but is it the right approach for you? Here are some advantages ("pros") and disadvantages ("cons") of using this approach.

Pro

Offers fine-grained control over AJAX processing. When you write all the AJAX-related code rather than using a JavaScript technology library or other approach that provides the AJAX functionality, you can specifically control the AJAX-related processing. You can include functionality that a library might not provide and customize or optimize the AJAX-related code in a way that's best suited for your needs. For example, you can add security controls or optimize the code for performance.

Cons

Requires a lot of AJAX-related coding. If you examine the code in the AJAX-enabled bookstore2 application, you'll see that it requires quite a bit of AJAX-related coding. If other

approaches provide the AJAX functionality you need and require less coding, using those approaches makes sense. For example, client-side JavaScript technology libraries such as those in the Dojo toolkit encapsulate some AJAX operations such as creating and configuring an `XMLHttpRequest`, which reduces the amount of JavaScript technology code that you need to provide. If a library provides the AJAX functionality you need, it's typically simpler to use it in your application than to code that functionality from scratch. See the [Design Strategy 2: Use a Client-Side JavaScript Technology Library](#) section of this article.

Requires knowledge of multiple languages and technologies. AJAX-related coding requires you to know JavaScript technology, CSS, and DOM -- which are not necessarily familiar to Java technology programmers. Other approaches, such as using jMaki technology, allow you to incorporate AJAX functionality using Java technology-based techniques. See the [Design Strategy 4: Do the Wrap Thing](#) section of this article.

Requires developers to contend with browser incompatibilities. Although most modern browsers handle AJAX code, not all browsers handle that code in the same way. In particular, various browsers differ in how they handle JavaScript technology code. Significantly, the `XMLHttpRequest` object is not yet part of the JavaScript technology standard. In fact, Internet Explorer (IE) does not currently support the `XMLHttpRequest` object. Instead IE uses an ActiveX control for AJAX-related communication with the server. That's why the `bpui.alone.initRequest()` function includes the following test:

```
bpui.alone.initRequest=function() {
    if (window.XMLHttpRequest) {
        return new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        return new ActiveXObject("Microsoft.XMLHTTP");
    }
}
```

In addition, you must code for the possibility that a browser does not handle AJAX, as is true for many older versions of browsers. Other approaches, such as using client-side JavaScript technology libraries, jMaki, Direct Web Remoting (DWR), and the Google Web Toolkit (GWT), handle browser incompatibilities. See the [Design Strategy 5: Go Remote](#) section of this article.

Requires developers to contend with UI issues. AJAX-enabled web applications introduce a number of UI issues. For example, how do you support bookmarks properly for an AJAX-enabled web page if its content changes but its URL does not? How do you bookmark a specific state of that page? For that matter, how do you properly support the Back button? For other than simple applications, developers must consider these UI issues and include code that handles them appropriately. Client-side JavaScript technology libraries such as those provided in the Dojo toolkit, Prototype, and Script.aculo.us, include features that handle some of these issues.

Can be difficult to debug. Because of the distribution of AJAX-related code between the client and the server, and the combination of JavaScript technology, DOM, CSS, and other types of code, debugging AJAX-related code can be difficult. Fortunately new debugging tools, such as the Mozilla Firebug Debugger, are emerging to make debugging of AJAX-related code easier. Also, GWT provides an environment for debugging both client and server code. See the [Design](#)

Strategy 6: Go All Java Technology section of this article.

Exposes JavaScript technology code -- a potential security risk. JavaScript technology code on the client side of a web application is viewable. Anyone, including hackers, can examine the code by viewing the source through a browser. Mischief may follow. Hackers can attempt to reverse engineer your web application. They might even try to hijack `XMLHttpRequests` from your AJAX-enabled web application and return destructive responses.

When should you use this approach?

Use the do-it-yourself approach when you need fine-grained control your web application's AJAX functionality.

Design Strategy 2: Use a Client-Side JavaScript Technology Library

Instead of writing all the client-side JavaScript technology code for an AJAX-enabled application, you can take advantage of JavaScript technology libraries that provide AJAX functionality such as those in the Dojo toolkit, Prototype, Script.aculo.us, and Rico. To find others, see [Survey of AJAX/JavaScript Libraries](#).

The [Dojo toolkit](#) is an open-source JavaScript technology toolkit whose libraries and APIs can simplify building AJAX into a web application. For example, the `dojo.io` library abstracts AJAX-related communication with the server and so hides low-level `XMLHttpRequest` operations. Using the `dojo.io` library, here's what the `bpui.alone.showPopupInternal()` function in the AJAX-enabled bookstore2 application would look like:

```
dojo.require("dojo.io.*");
...
// This function is called after initial timeout that represents the delay
bpui.alone.showPopupInternal=function(popupx, bookId) {
    // retrieve data through dojo call
    var bindArgs = {

        url: "../PopupServlet?bookId=" + escape(bookId),
        mimetype: "text/xml",
        load: bpui.alone.ajaxReturnFunction};

    // dispatch the request
    bpui.alone.req=dojo.io.bind(bindArgs);
```

The `dojo.require()` function dynamically loads the JavaScript technology code for the specified library -- in this case the `dojo.io` library. The updated `bpui.alone.showPopupInternal()` function uses the `dojo.io.bind()` method to make an asynchronous request to the server. The `bind()` method wraps the `XMLHttpRequest`, so you don't need to create and configure an `XMLHttpRequest` object as you do in the do-it-yourself approach. 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, and the identification of the callback function. You can specify other parameters as well, such as `error` for specifying an error handler.

Other libraries in the Dojo toolkit provide a variety of APIs that simplify the coding for things such as animation, DOM manipulation, drag-and-drop support, and UI effects. In addition, the toolkit provides a sophisticated event-handling mechanism. However, the Dojo toolkit is particularly known for its library of built-in widgets -- prebuilt UI components that you can easily plug into a web application and customize as needed. The Dojo widget API also enables you to create your own widgets.

[Prototype](#) is a JavaScript technology framework. It provides a library whose components include objects that simplify the use of JavaScript technology. One component presents an AJAX object that, like the `dojo.io` library in the Dojo toolkit, encapsulates the `XMLHttpRequest` object and hides low-level `XMLHttpRequest` 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 technology libraries that support AJAX, drag and drop, UI effects, as well as other functions that can be plugged into a web application.

Pros and Cons of Using a Client-Side JavaScript Technology Library

Here are some pros and cons of using a client-side JavaScript technology library to build AJAX into a web application.

Pros

Hides low-level AJAX "plumbing". Libraries such as the `dojo.io` library in the Dojo toolkit as well as the Prototype library encapsulate some of the AJAX details, such as creating and configuring `XMLHttpRequest` objects and performing operations on those objects. These libraries allow you to code at a higher, more abstract level, saving you from having to provide more detailed code.

Reduces the need for JavaScript technology coding. Libraries such as the Dojo toolkit DOM manipulation library or Prototype DOM manipulation objects provide routines that can be easier to use and require less code than their equivalent JavaScript technology APIs. In addition, incorporating prebuilt widgets such as those in the Dojo toolkit is typically much simpler than coding the JavaScript technology functionality that those widgets provide.

Handles browser incompatibilities in processing AJAX. Client-side JavaScript technology libraries such as those in the Dojo toolkit hide many of the differences in the way different browsers handle AJAX. For example, the `dojo.io.bind()` method request wraps an `XMLHttpRequest` that handles IE differences in the way an asynchronous request is made to a server component. The `dojo.event` library presents an event-handling system that hides differences in the way different browsers treat JavaScript technology events. The objects that Prototype provides for DOM manipulation hide browser differences in handling DOM-related operations. In addition, client-side JavaScript technology libraries such as those provided in the Dojo toolkit, Prototype, and Script.aculo.us, include "graceful degradation" features that handle cases in which AJAX is not supported -- saving you from having to code for that possibility.

Handles some common AJAX issues such as bookmarking and support for the Back button. A number of the client-side JavaScript technology libraries provide support for bookmarking and the Back button. For example, you can add support for bookmarking in the bookstore2 web application simply by setting a `dojo.io.bind()` parameter named `changeURL` to `true`:

```

dojo.require("dojo.io.*");
...
// This function is called after initial timeout that represents the delay
bpui.alone.showPopupInternal=function(popupx, bookId) {
    // retrieve data through dojo call
    var bindArgs = {

        url: "../PopupServlet?bookId=" + escape(bookId),
        mimetype: "text/xml",
        load: bpui.alone.ajaxReturnFunction};
        changeURL:true;

    // dispatch the request
    bpui.alone.req=dojo.io.bind(bindArgs);

```

Established user communities for these toolkits can help in answering questions. Well-established client-side libraries such as the Dojo toolkit also have well-established user communities that provide support and information through community-based forums and blogs. In addition, communities for open-source libraries such as the Dojo libraries participate in expanding and enhancing the libraries. This includes support and contributions from companies. For example, the [Dojo Foundation](#), a nonprofit organization designed to promote the adoption of Dojo and JavaScript, has sponsorship and active participation from companies such as IBM and Sun Microsystems.

Cons

Requires some knowledge of JavaScript technology. Client-side JavaScript technology libraries do not eliminate the need for JavaScript technology coding. At best, they reduce the need for writing JavaScript technology code. In the updated bookstore2 example, using the Dojo toolkit eliminates the need to write JavaScript technology code to create, configure, and use an `XMLHttpRequest` object, but you still need to write JavaScript technology code to call the `dojo.io.bind()` function that wraps the request for an `XMLHttpRequest` object.

Might need to mix and match JavaScript technology libraries. This can be viewed as a pro and a con. If a single JavaScript technology library does not give you the functionality you need, you have the flexibility of getting additional functionality from another JavaScript technology library, that is, from the same or another toolkit. You are not limited to using one client-side JavaScript technology library in your web application. For instance, you can use a Prototype object for AJAX-related communication with the server and widgets from the Dojo toolkit for the UI. That flexibility is a good thing. However different client-side library sources have their own syntax requirements, so you might have to learn different ways of working with these libraries.

Might not meet all AJAX needs. The likelihood is that using client-side JavaScript technology libraries will not meet all the client-side JavaScript technology needs for building AJAX into a web application. Because of that, you'll need to use another approach to fill the remainder of the requirement.

When should you use this approach?

Use client-side JavaScript technology libraries if they can simplify the JavaScript technology code you would need to write for your web application.

Design Strategy 3: Use a Client-Server Framework

Another approach to building AJAX into a web application is to use a client-server framework. There are a number of client-server frameworks that you can use, including JavaServer Faces technology frameworks such as those in the [Java EE 5 SDK](#), [ICEfaces](#), [Ajax4jsf](#), and [Project Dynamic Faces](#). There are also other types of client-server frameworks, such as [Direct Web Remoting \(DWR\)](#), and the [Google Web Toolkit \(GWT\)](#), that are not based on JavaServer Faces technology. This section covers the use of JavaServer Faces technology. The section [Project Dynamic Faces -- a Variation on Adding AJAX to JavaServer Faces Technology Components](#) discusses that framework. The section [Design Strategy 5: Go Remote](#) discusses DWR, and the section [Design Strategy 6: Go All Java Technology](#) discusses GWT.

[JavaServer Faces technology](#), often referred to as JSF, is designed to simplify building functionally rich UIs for web applications. Central to the technology is a powerful component model. The model offers a set of APIs for representing UI components and for managing their state. These APIs also give developers a way to programmatically handle events on components, as well as convert and validate input data. JavaServer Faces technology UI components are actually server-side components. They run on the server and get rendered to the client, and they can respond to events on the client.

One advantage of using JavaServer Faces technology is that it allows page authors to use UI components on their pages without having to know the details of how the components work. Typically, the components are expressed using JSP tags on a JSP page -- although that's not the only way to represent components. For a page author, including JavaServer Faces UI components in a page is as simple as referencing the tag libraries for the components and then using the JSP tags for those components on the page. JSP tag libraries for a core set of JavaServer Faces components are provided as a standard part of the technology. Perhaps equally important, JavaServer Faces UI components are designed to be easily importable into integrated development environments (IDEs) such as the [Sun Java Studio Creator IDE](#). This enables application developers to build AJAX into a web application by dragging and dropping visual components.

Significantly, the JavaServer Faces component model is extensible so that a component developer can create custom components. This also means that a component developer can create JavaServer Faces components that have AJAX functionality or add AJAX functionality to existing components. As mentioned earlier in this article, a [library of AJAX-enabled custom JavaServer Faces components](#) is already available as part of the [Java Blueprints Solutions Catalog](#).

Creating and using custom components that include AJAX functionality does not necessarily reduce the amount of AJAX-related coding. A component developer still needs to provide the JavaScript technology, CSS, and DOM code to implement the AJAX functionality for the custom component, but the use of custom components does shield the page author from the coding details.

You can see that by examining any of the AJAX-enabled JavaServer Faces components in the Blueprints Catalog. Let's look at one of them: a custom JavaServer Faces component that uses the phase listener approach. The custom component includes AJAX functionality similar to that in the [AJAX-enabled bookstore 2](#)

application. When a user moves a mouse over the component, it triggers a function that displays a pop-up balloon as shown in Figure 3.

CompB Example - Custom Component using a PhaseListener

Mouse over link to see popup (test1A)

(With a JSP Managed Bean fully)

Mouse over link to see popup

(With a JSP Managed Bean fully)

Mouse over link to see popup

(With a JSP fulfilling AJAX Req)

Mouse over link to see popup

(With a JSP fulfilling AJAX Request)

test1A Lookup

The itemId that is performing this lookup is 'test1A'.

This popup is trigger through the onmouseover and onmouseout event handlers. This popup request is fulfilled using a **JSP managed bean**, that is called by the CompBPhaseListener

Figure 3: JSF Custom Component Using a Phase Listener

You can find a detailed description of the custom component in "[Using PhaseListener Approach for Java Server Faces Technology with AJAX](#)". A *phase listener* is an interface that is notified when each stage of the JavaServer Faces request processing lifecycle begins and ends. Using a phase listener is only one of several ways to handle JavaServer Faces technology requests. For example, you could [use a servlet with JavaServer Faces Technology and AJAX](#). See the Blueprints Catalog for these and other approaches.

To use the custom component in a JSP page, a page author references the custom component's tag library along with the standard JavaServer Faces technology tag libraries, and then specifies the tag for the component in the page:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib prefix="ui" uri="http://java.sun.com/blueprints/ui" %>
...
<f:view>
    <ui:compB id="pop1" url="faces/bpui_compB_action/CompBBean/processMethod?itemId=" />
</f:view>
```

`<ui:compB>` is the tag for a custom component that includes the pop-up balloon functionality. The `url` parameter contains information that the server needs to handle requests related to the component.

The JSP page also includes the code that maps the mouseover event to a JavaScript technology function that

handles the event.

```
<a href="#" onmouseover="bpui.compB.showPopup('pop1', event, 'test1A')"
onmouseout="bpui.compB.hidePopup('pop1')" style="cursor: pointer">
  <b>Mouse over link to see popup (test1A)</b></a><br/>
  <small><i>(With a JSF Managed Bean fulfilling the AJAX Request)</i></small><br/><br/>
```

That's basically all the page author needs to code regarding the pop-up balloon.

However, a component developer still needs to provide the JavaScript technology function that the mouseover event will trigger. As usual, the function creates and configures an `XMLHttpRequest` object and uses it to make an asynchronous request to a server component.

```
bpui.compB.showPopupInternal=function(popupx, itemId) {
  // initialize the AJAX request
  bpui.compB.req=bpui.compB.initRequest();
  // retrieve the correct popup object that is being shown
  popObject=bpui.compB[popupx];

  // concatenate the itemId value to the URI
  url=popObject.urlx + escape(itemId);
  // set the correct popup's callback function
  bpui.compB.req.onreadystatechange = popObject.ajaxReturnFunction;
  bpui.compB.req.open("GET", url, true);
  // send the request
  bpui.compB.req.send(null);
}
```

And the component developer must provide the JavaScript technology code for the callback function that updates the DOM representation of the page.

```
this.ajaxReturnFunction=function() {
  // make sure response is ready
  if (bpui.compB.req.readyState == 4) {
    // make sure it is a valid response
    if (bpui.compB.req.status == 200) {
      // populate the popup with the info from the response
      var resultx=bpui.compB.req.responseXML.getElementsByTagName("response")[0];
      document.getElementById(componentId + "_title").innerHTML=
        resultx.getElementsByTagName("title")[0].childNodes[0].nodeValue;
      document.getElementById(componentId + "_message").innerHTML=
        resultx.getElementsByTagName("message")[0].childNodes[0].nodeValue;;
      // show popup with the newly populated information
      document.getElementById(componentId).style.visibility='visible';
    } else if (bpui.compB.req.status == 204){
      // error, just show alert
      alert("204 returned from AJAX call");
    }
  }
}
```

```

}
}

```

This approach differs from the do-it-yourself approach in that the JavaScript technology is encapsulated in the custom component on the server. A renderer for the custom component renders the markup for the component on the page as well as the markup for a `<script>` tag. That tag points to the JavaScript technology file for the component. In rendering the custom component on the page, a call is made to the URL specified in the custom tag. The first part of that URL, `faces`, is mapped to a `FacesServlet` servlet. Note that the JavaServer Faces framework implements the **Model-View-Controller (MVC) design pattern** and the `FacesServlet` is the Controller in the framework. The mapping to the `FacesServlet` is performed in the web application's deployment descriptor, `web.xml`.

During the JavaServer Faces request processing lifecycle, the framework invokes the custom component's phase listener, which is registered in the component's `faces-config.xml` deployment descriptor. The phase listener checks the URL to see if the request is related to the custom component. If the request is associated with the custom component, the phase listener fulfills the request based on the URLs' contents.

As [Figure 4](#) illustrates, after the page is rendered and a user initiates a mouseover event, the JavaScript technology function that handles the event creates and configures an `XMLHttpRequest` object (1) and uses it to make an asynchronous request. Various components on the server take part in handling the request, including the `FacesServlet` and the phase listener (2), as well as a managed bean (3), one of whose methods constructs an appropriate XML response. The callback function then updates the DOM based on the returned XML (4), populates the pop-up balloon, and displays it (5).

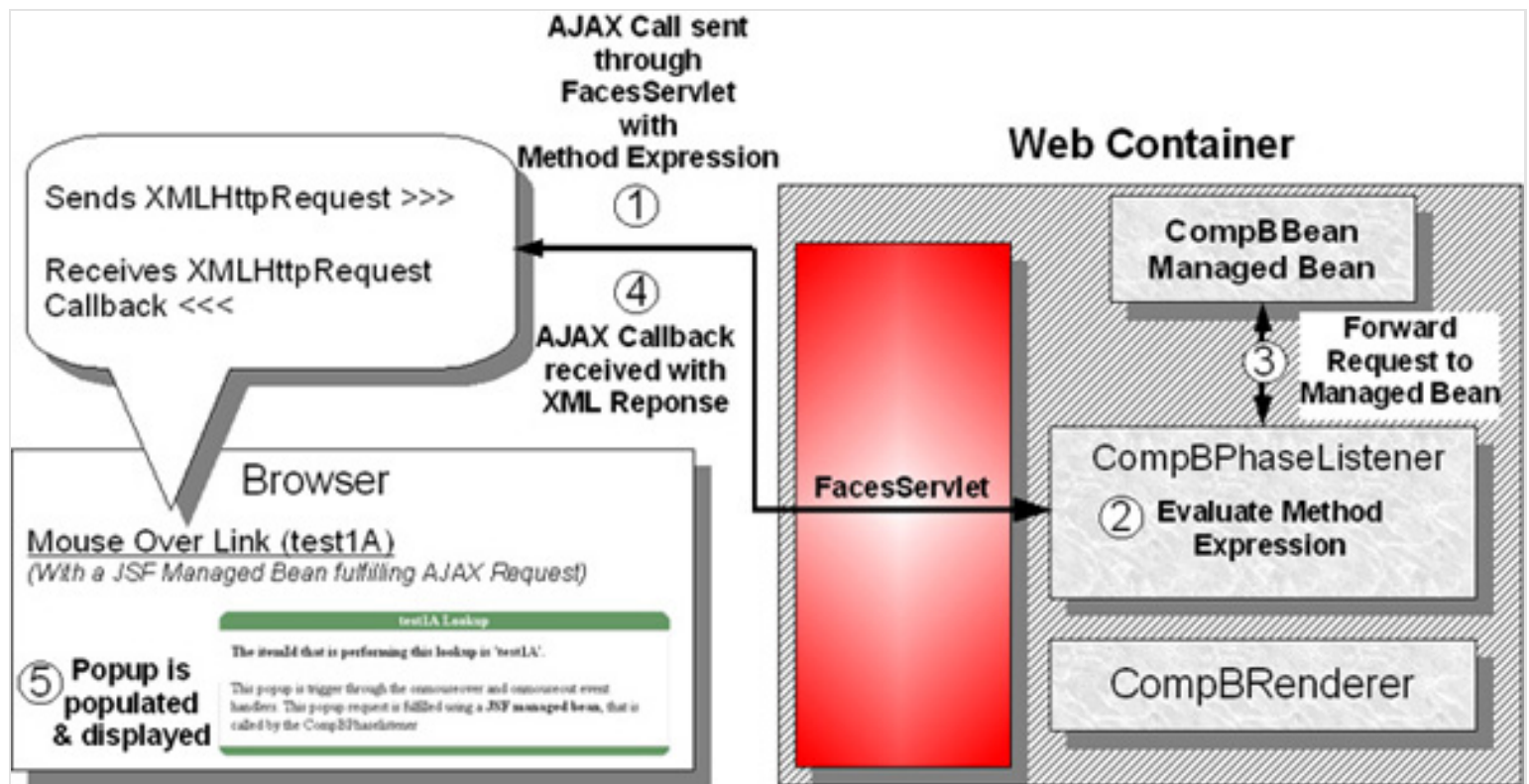


Figure 4: Processing an AJAX Request Using a Phase Listener

Pros and Cons of Using JavaServer Faces Technology

Here are some pros and cons of using JavaServer Faces technology to build AJAX into a web application.

Pros

Page authors do not need to know JavaScript technology, CSS, and DOM. As shown in the previous example, to use an AJAX-enabled component on a page, a page author only needs to reference the custom component's tag library, specify the tag for the component, and map a pertinent event to a JavaScript technology function that handles the event. In this example, the page author also needs to specify the appropriate URL for the component tag. None of this requires the page author to have any knowledge of JavaScript technology, CSS, or DOM coding.

AJAX-enabled custom components are reusable. JavaServer Faces UI components are reusable. A page author can use them whenever and wherever they are needed. This is also true for AJAX-enabled components.

Component developers can take advantage of JavaServer Faces technology features. JavaServer Faces technology provides a rich architecture for handling events and converting and validating user input. Component developers can take advantage of these features in UI components that include AJAX functionality.

Application developers can add AJAX to a web application by dragging and dropping visual components. JavaServer Faces UI components, including custom components, are designed to be easily importable into integrated development environments such as the [Sun Java Studio Creator IDE](#). In fact, a set of AJAX-enabled JavaServer Faces components is packaged with the Sun Java Studio Creator IDE. If an AJAX-enabled component is incorporated into an IDE, an application developer can add it to a web application simply by dragging and dropping it into the visual view of the application.

Cons

Has many of the same disadvantages as the do-it-yourself approach. Using JavaServer Faces technology does not eliminate many of the issues that face developers who take the do-it-yourself approach. Component developers still need to write JavaScript technology code -- potentially a lot of it, and know the intricacies of working with CSS and DOM. Component developers must still contend with browser incompatibilities and UI issues such as support for the Back button and for bookmarking. However, a project called [Dynamic Faces](#) addresses some of these disadvantages. Dynamic Faces makes it simpler to add AJAX functionality to JavaServer Faces technology UI components.

Project Dynamic Faces -- a Variation on Adding AJAX to JavaServer Faces Technology Components

Project Dynamic Faces is the run-time part of the open-source [jsf-extensions project](#), whose objective is to make it simpler to add AJAX functionality to JavaServer Faces technology components. Dynamic Faces extends the support for AJAX in the JavaServer Faces technology framework.

A major advantage of using Dynamic Faces is that it reduces the amount of JavaScript technology code that you need to write to enable AJAX functionality in a component. In fact, using Dynamic Faces can eliminate the need for JavaScript technology code entirely. For example, suppose a page contains a menu of choices, two buttons,

and an output text area. You want to AJAX-enable the components so that depending on what the user selects, the content of the text area is updated asynchronously, without a full page refresh. To do that, the page author specifies the Dynamic Faces tag library on the page and sets up an "AJAX zone" for the components.

```
<%@ taglib prefix="jsfExt"
    uri="http://java.sun.com/jsf/extensions/dynafaces" %>

<jsfExt:ajaxZone id="selectzone"
    action="#{updateOutput}">
    <h:commandButton id="Selection1" ...
        ActionListener=.../>
    <h:commandButton id="Selection2" ...
        ActionListener=.../>
    <h:outputText value= .../>
    <h:selectOneMenu .../>
</jsfExt:ajaxZone>
```

Learn more about Project Dynamic Faces, Project jMaki, and other projects that make developing interactive and dynamic web applications easier, in the article "[New Technologies for Ajax and Web Application Development: Project jMaki, Project Dynamic Faces, and Project Phobos](#)"

Specifying the components in an `<ajaxZone>` tag tells Dynamic Faces which components in the JavaServer Faces component tree to enable for AJAX.

Notice that no JavaScript technology code is required here. In particular, there is no need to create, configure, and perform operations on an `XMLHttpRequest` object or provide any code on the server to handle the AJAX request. Dynamic Faces takes care of all of the low-level details related to the AJAX request. When a user clicks either of the two buttons, it invokes the action referenced by `#{updateOutput}`. This is simply JavaServer Faces technology-related coding without any JavaScript.

Dynamic Faces also makes it easy to code a "one-to-many" AJAX request, in which an event fired on one AJAX-enabled component asynchronously updates a number of other AJAX-enabled components. Dynamic Faces allows you to do this with one AJAX request. Without Dynamic Faces, you would need to issue a separate AJAX request for each component, what some call the "swim lane" approach. Dynamic Faces also provides a JavaScript technology library that you can use for fine-grained control over components. For instance, you can use the Dynamic Faces JavaScript technology library to associate one type of event with some components in an AJAX zone and associate another type of event with other AJAX components in the zone.

Finally, because Dynamic Faces is built on the JavaServer Faces framework, you can still take advantage of all the features of the framework such as input validation and conversion and state management.

When should you use this approach?

Use JavaServer Faces technology if you want to take advantage of tools such as Java Studio Creator to build web applications by dragging and dropping components. Use Dynamic Faces to simplify adding AJAX functionality to JavaServer Faces technology components. Continue to use JavaServer Faces technology to build AJAX into your web applications if you already use JavaServer Faces technology for your web applications.

Design Strategy 4: Do the Wrap Thing

Client-side JavaScript technology libraries and client-server frameworks offer distinct advantages for building AJAX into a web application. But is there a way to merge the two, to make JavaScript technology libraries available within a client-server framework? The answer is yes. One effort that attempts to meet that objective is the open-source [jMaki project](#). jMaki (pronounced jay-MAH-kee) is a framework for wrapping JavaScript technology widgets in JSP tags or JavaServer Faces components. The name jMaki cleverly joins Java technology -- that's what the *j* in jMaki stands for -- and wrap -- *Maki* is the Japanese word for wrap.

jMaki provides a library of wrapped widgets from various popular client-side JavaScript technology libraries such as the Dojo toolkit and Script.aculo.us. More than 30 of these jMaki widgets are already available. You can view these widgets in the [jMaki Widget Gallery](#). The jMaki framework also provides a way to add to the set of jMaki widgets.

A page author includes a jMaki widget on a page in the same way as a JSP tag or JavaServer Faces component. For example, one of the tags provided with jMaki is an inline editor widget from the Dojo widget library. Here's how a page author would include the widget as a JavaServer Faces component:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib prefix="a" uri="http://java.sun.com/jmaki-jsf" %>
...
<f:view>
  <a:ajax name="dojo.inlineedit"/>
</f:view>
```

Notice the reference to a jMaki tag library just after the references in the example to the JavaServer Faces tag libraries. jMaki includes two tag libraries -- one for JSP tags and another for JavaServer Faces components. In response to the tag callout on the page, jMaki renders the appropriate JavaScript technology code -- as well as HTML and CSS -- for the widget.

It takes three files to define a widget to jMaki: an HTML template file that specifies the page layout for the widget, a CSS file that specifies the style parameters for the widget, and a JavaScript technology file that specifies the widget's behavior. To add a widget to the set of jMaki widgets, you need to provide the three files for the widget. For example, here's the JavaScript technology file for the inline editor widget:

```
dojo.require("dojo.widget.*");
dojo.require("dojo.widget.InlineEditBox");
dojo.require("dojo.event.*");

dojo.widget.createWidget(widget.uuid);

var editable = dojo.widget.byId(widget.uuid);
editable.onSave = function() {
    var id = widget.uuid;
    alert("saving " + id);
};

jmaki.attributes.put(widget.uuid, editable);
```

The JavaScript technology code imports the Dojo libraries for the widget and includes a call to the JavaScript technology function that creates an instance of the widget.

You can use jMaki to wrap a widget in a JavaScript technology library -- that is, a widget that exists in a client-side JavaScript technology library such as the Dojo or Prototype widget library but that isn't already provided in jMaki -- or create your own widget and wrap it. Of course, if you create your own widget you must write the JavaScript technology code that fully specifies its behavior. If that behavior includes AJAX functionality, you must write the JavaScript technology code that implements that functionality. This includes creating and configuring an `XMLHttpRequest` object to make an asynchronous request, and coding a callback function.

Pros and Cons of Using jMaki

Here are some pros and cons of using jMaki to build AJAX into a web application.

Pros

Hides the JavaScript technology details. Because jMaki wraps widgets from JavaScript technology libraries such as those in Dojo, it offers all of the advantages of using those widgets, such as hiding low-level AJAX plumbing, handling browser incompatibilities, and handling UI issues such as bookmarking. You can use the widgets you need from one or more JavaScript technology libraries without having to deal with each library's unique syntax. jMaki presents developers with a consistent syntax no matter what the source of the widget. And of course, you don't need to write the JavaScript technology code that's already provided in the widgets.

Moreover, because jMaki makes widgets available as JSP tags or as JavaServer Faces components, it offers the advantages of JSP technology and JavaServer Faces technology. Page authors do not need to know JavaScript technology, CSS, and DOM. Page authors include a widget on a page simply by referencing the appropriate jMaki tag library and then specifying the tag for the widget. For widgets wrapped as JavaServer Faces components, developers can take advantage of the JavaServer Faces architecture for handling events and converting and validating user input. Application developers can also use widgets wrapped as JavaServer Faces components in a drag and drop application development environment such as the Sun Java Studio Creator IDE and NetBeans IDE. A [jMaki plug-in module](#) is available for use with NetBeans IDE 5.5 Beta.

Integrated into the Project Phobos scripting framework. In addition to working with JSP and JavaServer Faces technology, jMaki also works with the [Project Phobos](#) scripting framework. Phobos is a project dedicated to creating a framework for building web applications using scripting languages. The framework also allows developers to use the Java programming language rather than a scripting language, when appropriate, to perform a task. [This integration of jMaki into the Project Phobos framework](#) allows developers to use jMaki widgets within the framework.

Con

Requires some knowledge of JavaScript. Unless you're using a widget already wrapped in jMaki, you still need to know some JavaScript technology. As illustrated in the previous example, if you use jMaki to wrap a widget from a JavaScript technology library, you need to write the JavaScript technology code to import the appropriate packages for the widget as well as call the JavaScript technology function to create an instance of the widget. If you create your own widget, you must write the JavaScript technology code that specifies its behavior and the CSS that specifies its style. If that behavior includes AJAX functionality, you must write the JavaScript technology code that implements that functionality, including any DOM-manipulation.

When should you use this approach?

Use jMaki when you already use JSP technology or JavaServer Faces technology in your web applications and need to access widgets from client-side JavaScript technology libraries.

Design Strategy 5: Go Remote

Another approach to enabling AJAX functionality in a web application is to use remoting frameworks such as [Direct Web Remoting \(DWR\)](#) and [JSON-RPC](#). These frameworks enable developers to build AJAX into an application through Remote Procedure Call (RPC)-like calls in JavaScript technology code. For example, return to the pop-up balloon scenario that this article discussed earlier. You could create a simple Java class on the server to handle the AJAX requests for the pop-up balloon content. You could call the class `Popup` and the method in the class that provides the book details for the pop-up balloon `getDetails`. On the client, you code the JavaScript technology function that is triggered in response to the appropriate mouseover event. In the JavaScript technology code, you call the `getDetails` method:

```
<script type="text/javascript" src="dwr/engine.js"></script>
<script type="text/javascript" src="dwr/util.js"></script>

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

showPopupInternal=function(popupx, bookId) {
    Popup.getDetails(bookId, ajaxReturnFunction)
}

...
```

The parameter to the method, `ajaxReturnFunction`, is the callback function. You must write the callback function in the client JavaScript technology code. However, you don't need to code the low-level details of creating, configuring, and performing operations on an `XMLHttpRequest` object. The DWR framework takes care of that for you. The framework generates the equivalent of a Remote Method Invocation (RMI) stub on the client for the class. That stub is a JavaScript technology class that includes the low-level `XMLHttpRequest` operations. The framework then marshals the data exchanged between the generated JavaScript technology class and the server. This includes converting parameters that the client-side JavaScript technology passes, and converting the values that the server-side Java technology returns. On the server, the framework creates a servlet called `DWRServlet` that receives the request and dispatches it to the `Popup` class for processing.

Notice the two DWR JavaScript technology library files identified at the beginning of the code. The `dwr/util.js` file contains a utility library that you can use to update a web page through JavaScript technology. For instance, you can use the utility function `$(id)` instead of the equivalent DOM API `document.getElementById(id)` to access and manipulate an element in the DOM representation of a page. The `dwr/engine.js` file contains functions that marshal calls between the client stub and the server.

Pros and Cons of Using DWR

Here are some pros and cons of using DWR to build AJAX into a web application.

Pros

Hides low-level AJAX "plumbing". You make an AJAX request through an RMI-like function call to a Java method on the server. The DWR framework handles the low-level details such as creating and configuring `XMLHttpRequest` objects and performing operations on those objects.

Allows Java application developers to make AJAX requests using a familiar syntax. The RMI-like syntax of DWR calls should be familiar to most Java application developers.

Allows developers to expose existing business logic to an AJAX client with minimum effort. If you have business logic in a class on the server that can process an AJAX request, you can access that logic from a client-side JavaScript technology function with a simple method call.

Provides security controls. DWR offers various controls to protect remote classes and methods from unwarranted exposure. A remote class can be exposed through DWR only if it's specified in a configuration file, `dwr.xml`. These controls can be further refined to expose only specific methods on a remote class. DWR also supports Java Platform, Enterprise Edition (Java EE) role-based access controls. The allows specific Java EE roles to have access to specific remote classes or methods.

Cons

Requires knowledge of JavaScript. Although the DWR framework generates JavaScript technology code for low-level details of AJAX communication, you still need to write the JavaScript technology code that is triggered in response to an appropriate event -- such as an `onmouseover` or `onclick` event. You also still need to write the JavaScript technology code for the callback function. However you can reduce the amount of JavaScript coding by using DWR with jMaki widgets, AJAX-enabled JavaServer Faces components, or other components that encapsulate JavaScript technology.

Only works with Java objects. DWR is designed specifically for a JavaScript technology client and Java objects on the server. By comparison, different implementations of JSON-RPC exist for different languages.

When should you use this approach?

Use DWR if you have business logic in server-side Java objects that you want to use to process AJAX requests.

Design Strategy 6: Go All Java Technology

JavaServer Faces technology and jMaki are approaches that use the Java programming language to build AJAX into an application. But they still require developers to write some JavaScript technology code. There are, however, Java technology-only approaches. One of these approaches uses the [Google Web Toolkit \(GWT\)](#). GWT is a free -- though not open-source -- client-server framework that allows developers to build AJAX-enabled applications exclusively in the Java programming language. This includes the client side of the application -- a GWT Java-to-JavaScript technology compiler converts the Java programming language code on the client to JavaScript technology code and HTML. On the server, you can take advantage of server-side Java technologies such as servlets, JSP technology, and JavaServer Faces technology.

Building a UI for an AJAX-enabled application using GWT is similar to building a UI in a UI framework such as Java Foundation Classes/Swing (JFC/Swing). You format the user interface by laying out a set of GWT UI widgets in a virtual container called a panel. This is like formatting the UI of a JFC/Swing application by putting the UI components in a container such as `JPanel`. GWT provides a library of commonly used widgets. You can also create composite widgets that combine multiple GWT widgets, or create your own widgets.

The JFC/Swing paradigm continues in GWT's event model. To enable a widget so that it responds to an event such as a mouse click, you implement one of GWT's listener interfaces. Then you call the listener method appropriate to the event, passing as a parameter the type of widget associated with the event. For example, the following class displays two buttons and calls the `onClick` method to trigger the appropriate event handler when the user clicks either of the buttons.

```
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.FlowPanel;

public class ListenerExample extends Composite implements ClickListener {
    private FlowPanel fp = new FlowPanel();
    private Button b1 = new Button("Button 1");
    private Button b2 = new Button("Button 2");

    public ListenerExample(){
        setWidget(fp);
        fp.add(b1);
        fp.add(b2);
        b1.addClickListener(this);
        b2.addClickListener(this);
    }
}
```

```
public void onClick(Widget sender) {
    if (sender == b1) {
        //b1 click handler
        ...

    else (sender == b2) {
        //b2 click handler
        ...
    }
}
}
```

After creating the class or classes for your application, you can run the application in what's called "hosted mode". This runs your client in a Java virtual machine*. GWT provides a browser and a simple servlet so that you can do a test run of the application. You can also set breakpoints that hook back into your Java code. GWT runs with most standard Java technology IDEs. An important benefit of running in hosted mode is that it allows you to use your IDE's Java technology debugging facilities, such as the debugger in the Sun Java Studio Creator IDE or in the NetBeans IDE. After debugging the application, you run it through the GWT Java-to-JavaScript technology compiler to generate the JavaScript technology code and HTML. GWT also provides a "web mode" where you can run the compiled version of the application -- that is, the version that contains the JavaScript technology code and HTML.

One other important feature of GWT is its RPC mechanism, which allows the client to communicate with the server through method calls. This makes it relatively easy for the client and server to exchange Java objects over HTTP, such as those that carry server requests and data responses. Before you make a call, you must define service interfaces for the client and server, and implement the interface for the service on the server. Then you make the call from the client. GWT automatically serializes the client's request and deserializes the server's response.

Pros and Cons of Using GWT

Here are some pros and cons of using GWT to build AJAX into a web application.

Pros

Allows developers to build AJAX-enabled applications in the Java programming language.

Developers do not need to know JavaScript technology, CSS, or DOM. The GWT Java-to-JavaScript technology compiler compiles the client-side Java code into JavaScript technology code and HTML.

Provides GWT hosted mode, an environment that enables debugging. You can run your AJAX-enabled application in hosted mode. This allows you to use your IDE's debugging facilities to test and debug the application.

Handles browser incompatibilities in processing AJAX. The GWT Java-to-JavaScript technology compiler generates browser-compliant JavaScript technology code, saving developers from having to code for browser incompatibilities.

Developed and supported by Google. Among other things this means that Google has provided good documentation for using GWT as well as access to community-oriented sources such as developer forums.

Con

Generated JavaScript technology code has its drawbacks. Even though you don't need to know JavaScript technology to use GWT, you might sometimes need to examine the generated JavaScript technology code -- for example, if the browser displays a JavaScript technology error. For developers who are unfamiliar with JavaScript technology, understanding the JavaScript technology code that GWT generates can be difficult. Also because GWT generates the JavaScript technology code, you lose fine-grained control over the processing.

When should you use this approach?

Use GWT if you want to develop AJAX-enabled applications using the Java programming language exclusively.

Summary

Table 1: A Summary of AJAX Design Strategies Covered in This Article

Design Strategy	Description	Major Pro	Major Con	When to use
Do it yourself.	Do all the coding to build AJAX into a web application.	Fine-grained control over AJAX processing.	Requires a lot of AJAX-related coding.	When you need fine-grained control over your web application's AJAX functionality.
Use a client-side JavaScript technology library.	Use JavaScript libraries such as those in the Dojo toolkit.	Reduces the need for JavaScript coding.	Might need to mix and match JavaScript libraries, each with its own syntax.	When it can simplify the JavaScript technology code you would need to write for your web application.
Use a client-server framework.	Use a client-server framework such as the JavaServer Faces technology framework or Dynamic Faces.	Can create reusable, AJAX-enabled custom components.	Easy to use for page authors but presents some of the same disadvantages to developers as the do-it-yourself approach. Dynamic Faces mitigates some of these disadvantages.	When you want to take advantage of tools such as the Sun Java Studio Creator IDE to build web applications by dragging and dropping components. Or if you're already using JavaServer Faces technology to build web applications.

Do the wrap thing.	Use a technology such as jMaki to wrap JavaScript technology libraries, making them available within a Java technology-based client-server framework.	Can hide JavaScript technology details.	Might require some knowledge of JavaScript technology.	When you already use JSP technology or JavaServer Faces technology in your web applications and need to access widgets from client-side JavaScript technology libraries.
Go remote.	Use a remoting framework such as DWR or JSON-RPC.	Can make AJAX requests using a familiar RMI-like syntax.	Only works with Java objects.	When you have business logic in server-side Java objects that you want to use to process AJAX requests.
Go all Java technology.	Use frameworks such GWT for building AJAX applications in the Java programming language.	Can build AJAX-enabled applications exclusively in the Java programming language.	The JavaScript technology code that GWT generates can be complex and difficult to understand.	When you want to develop AJAX-enabled applications using the Java programming language exclusively.

For More Information

- [AJAX Developer Resource Center](#)
- [AJAX FAQ for the Java Developer](#)
- [Hands-On Java EE 5](#)
- [Creating an AJAX-Enabled Bookstore Application, a Do-It-Yourself Approach](#)
- [Asynchronous JavaScript Technology and XML \(AJAX\) With Java 2 Platform, Enterprise Edition](#)
- [Java BluePrints Solutions Catalog](#)
- [Java BluePrints AJAX Components](#)
- [AJAX Components in the Java Studio Creator IDE](#)
- [Programming Model for AJAX](#)
- [Including AJAX Functionality in a Custom JavaServer Faces Component](#)
- [JSF Extensions](#)
- [Project jMaki](#)
- [Direct Web Remoting](#)
- [Java Pet Store Demo 2.0](#)
- [Java EE Technologies](#)
- [Java EE SDK](#)
- [New Technologies for Ajax and Web Application Development: Project jMaki, Project Dynamic Faces, and Project Phobos](#)

About the Authors



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



Mark Basler a senior software engineer, is part of the Java BluePrints team and helped create the [Java Blueprints Solution Catalog](#) and the [Java Pet Store Demo 2.0](#), reference applications that demonstrate how to design and develop AJAX-enabled Web 2.0 applications. His other contributions include the design and development of key components for Sun's Download Center, eCommerce suites, and Sun Java System Application Server.

* The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java platform.

Rate and Review

Tell us what you think of the content of this page.

Excellent **Good** **Fair** **Poor**

Comments:

If you would like a reply to your comment, please submit your email address:

Note: We may not respond to all submitted comments.