Ajax for Java developers: **Build dynamic Java applications**

Ajax paves the way for better Web applications

Philip McCarthy

September 20, 2005

The page-reload cycle presents one of the biggest usability obstacles in Web application development and is a serious challenge for Java™ developers. In this series, author Philip McCarthy introduces a groundbreaking approach to creating dynamic Web application experiences. Ajax (Asynchronous JavaScript and XML) is a programming technique that lets you combine Java technologies, XML, and JavaScript for Java-based Web applications that break the page-reload paradigm.

View more content in this series

Ajax, or Asynchronous JavaScript and XML, is an approach to Web application development that uses client-side scripting to exchange data with the Web server. As a result, Web pages are dynamically updated without a full page refresh interrupting the interaction flow. With Ajax, you can create richer, more dynamic Web application user interfaces that approach the immediacy and usability of native desktop applications.

Ajax isn't a technology, it's more of a pattern -- a way to identify and describe a useful design technique. Ajax is new in the sense that many developers are just beginning to be aware of it, but all of the components that implement an Ajax application have existed for several years. The current buzz is because of the emergence in 2004 and 2005 of some great dynamic Web UIs based on Ajax technology, most notably Google's GMail and Maps applications and the photo-sharing site Flickr. These UIs were sufficiently groundbreaking to be dubbed "Web 2.0" by some developers, with the resulting interest in Ajax applications skyrocketing.

In this series, I'll give you all the tools you need to begin developing your own applications using Ajax. In this first article, I'll explain the concepts behind Ajax and demonstrate the fundamental steps to creating an Ajax interface for a Java-based Web application. I'll use code examples to demonstrate both the server-side Java code and the client-side JavaScript that make Ajax applications so dynamic. Finally, I'll point out some of the pitfalls of the Ajax approach, as well as the broader usability and accessibility issues you should consider when creating Ajax applications.
A better shopping cart

You can use Ajax to enhance traditional Web applications, streamlining interaction by eliminating page loads. To demonstrate this, I'll use the simple example of a shopping cart that is dynamically updated as items are added to it. Incorporated into an online store, this approach would let users continue browsing and adding items to their carts without having to wait after each click for a full-page update. While some of the code in this article is specific to the shopping cart example, the techniques illustrated can be applied to any Ajax application. Listing 1 shows the relevant HTML code the shopping-cart example uses. I'll refer back to this HTML throughout the article.

Listing 1. Relevant fragments of the shopping cart example

```
<!-- Table of products from store's catalog, one row per item -->
<th>Name</th> <th>Description</th> <th>Price</th> <th></th>
... 
<tr>
<!-- Item details -->
<td>Hat</td> <td>Stylish bowler hat</td> <td>$19.99</td> <td>
  <!-- Click button to add item to cart via Ajax request -->
  <button onclick="addToCart('hat001')">Add to Cart</button>
</td>
</tr>
...

<!-- Representation of shopping cart, updated asynchronously -->
<ul id="cart-contents">
  <!-- List-items will be added here for each item in the cart -->
</ul>

Total cost: <span id="total">$0.00</span>
```

The Ajax roundtrip

An Ajax interaction begins with a JavaScript object called XMLHttpRequest. As the name suggests, it allows a client-side script to perform HTTP requests, and it will parse an XML server response. The first step in this Ajax roundtrip is to create an XMLHttpRequest instance. The HTTP method to use for the request (GET or POST) and the destination URL are then set on the XMLHttpRequest object.

Now, remember how that first a in Ajax stands for asynchronous? When you send that HTTP request, you don't want the browser to hang around waiting for the server to respond. Instead, you want it to continue reacting to the user's interaction with the page and deal with the server's response when it eventually arrives. To accomplish this, you can register a callback function with the XMLHttpRequest and then dispatch the XMLHttpRequest asynchronously. Control then returns to the browser, but the callback function will be called when the server's response arrives.

On the Java Web server, the request arrives just like any other HttpServletRequest. After parsing the request parameters, the servlet invokes the necessary application logic, serializes its response into XML, and writes it to the HttpServletResponse.
Back on the client side, the callback function registered on the XMLHttpRequest is now invoked to process the XML document returned by the server. Finally, the user interface is updated in response to the data from the server, using JavaScript to manipulate the page's HTML DOM. Figure 1 is a sequence diagram of the Ajax roundtrip.

**Figure 1. The Ajax roundtrip**

Now that you have a high-level view of the Ajax roundtrip, I'll zoom in for a more detailed look at each step along the way. Refer back to Figure 1 if you lose your place -- the sequence isn't entirely straightforward because of the asynchronous nature of the Ajax approach.

**Dispatching an XMLHttpRequest**

I'll start at the beginning of the Ajax sequence: creating and dispatching an XMLHttpRequest from the browser. Unfortunately, the method to create an XMLHttpRequest differs from browser to browser. The JavaScript function in Listing 2 smoothes out these browser-dependent wrinkles, detecting the correct approach for the current browser and returning an XMLHttpRequest ready to use. It's best to think of this as boilerplate code: simply copy it into your JavaScript library and use it when you need an XMLHttpRequest.
Listing 2. Creating a cross-browser XMLHttpRequest

```javascript
/*
 * Returns a new XMLHttpRequest object, or false if this browser
 * doesn't support it
 */
function newXMLHttpRequest() {
    var xmlreq = false;
    if (window.XMLHttpRequest) {
        // Create XMLHttpRequest object in non-Microsoft browsers
        xmlreq = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        // Create XMLHttpRequest via MS ActiveX
        try {
            // Try to create XMLHttpRequest in later versions
            // of Internet Explorer
            xmlreq = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (e1) {
            // Failed to create required ActiveXObject
            try {
                // Try version supported by older versions
                // of Internet Explorer
                xmlreq = new ActiveXObject("Microsoft.XMLHTTP");
            } catch (e2) {
                // Unable to create an XMLHttpRequest with ActiveX
            }
        }
    }
    return xmlreq;
}
```

Later, I'll discuss techniques to handle browsers that don't support XMLHttpRequest. For now, the examples assume that the `newXMLHttpRequest` function from Listing 2 will always return an XMLHttpRequest instance.

Getting back to the example shopping-cart scenario, I want to invoke an Ajax interaction whenever the user hits the Add to Cart button for a catalog item. The `onclick` handler function named `addToCart()` is responsible for updating the state of the cart through an Ajax call (see Listing 1). As shown in Listing 3, the first thing that `addToCart()` needs to do is obtain an instance of XMLHttpRequest by calling the `newXMLHttpRequest()` function from Listing 2. Next, it registers a callback function to receive the server's response (I'll explain this in detail later; see Listing 6).

Because the request will modify state on the server, I'll use an HTTP POST to do the deed. Sending data through POST requires three steps. First, I need to open a POST connection to the server resource I'm communicating with -- in this case a servlet mapped to the URL cart.do. Next, I set a
header on the XMLHttpRequest stating that the content of the request is form-encoded data. Finally, I send the request with form-encoded data as the body.

Listing 3 puts these steps together.

**Listing 3. Dispatching an Add to Cart XMLHttpRequest**

```javascript
/*@ 
* Adds an item, identified by its product code, 
* to the shopping cart 
* itemCode - product code of the item to add. 
*/
function addToCart(itemCode) {
    // Obtain an XMLHttpRequest instance
    var req = new XMLHttpRequest();

    // Set the handler function to receive callback notifications
    // from the request object
    var handlerFunction = getReadyStateHandler(req, updateCart);
    req.onreadystatechange = handlerFunction;

    // Open an HTTP POST connection to the shopping cart servlet.
    // Third parameter specifies request is asynchronous.
    req.open("POST", "cart.do", true);

    // Specify that the body of the request contains form data
    req.setRequestHeader("Content-Type", 
         "application/x-www-form-urlencoded");

    // Send form encoded data stating that I want to add the
    // specified item to the cart.
    req.send("action=add&item="+itemCode);
}
```

And with that, you've seen the first part of setting up the Ajax roundtrip -- namely creating and dispatching the HTTP request from the client. Next up is the Java servlet code used to handle the request.

**Servlet request handling**

Handling an XMLHttpRequest with a servlet is largely the same as handling a regular HTTP request from a browser. The form-encoded data sent in the POST request's body can be obtained with HttpServletRequest.getParameter() calls. Ajax requests take part in the same HttpSession as regular Web requests from the application. This is useful for the example shopping-cart scenario because it lets me encapsulate the user's shopping-cart state in a JavaBean and persist that state to the session between requests.

Listing 4 is part of a simple servlet that handles Ajax requests to update the shopping cart. A Cart bean is retrieved from the user's session and its state is updated according to the request parameters. The Cart is then serialized to XML, and that XML is written to the ServletResponse. It's important to set the response's content type to application/xml, otherwise the XMLHttpRequest will not parse the response content into an XML DOM.

**Listing 4. Servlet code to handle Ajax requests**

```java
public void doPost(HttpServletRequest req, 
```
Listing 5 shows an example of the XML produced by the `Cart.toXml()` method. It's pretty straightforward. Note the generated attribute on the `cart` element, which is a timestamp produced by `System.currentTimeMillis()`.

**Listing 5. Example XML serialization of the Cart object**

```xml
<?xml version="1.0"?>
<cart generated="1123969988414" total="$171.95">
  <item code="hat001">
    <name>Hat</name>
    <quantity>2</quantity>
  </item>
  <item code="cha001">
    <name>Chair</name>
    <quantity>1</quantity>
  </item>
  <item code="dog001">
    <name>Dog</name>
    <quantity>1</quantity>
  </item>
</cart>
```

If you take a look at Cart.java in the application source code available from the Download section, you'll see that the XML is produced simply by appending strings together. While sufficient for this example, this is pretty much the worst way to produce XML from Java code. I'll suggest some better approaches in the next installment of this series.

So now you know how the `CartServlet` responds to an `XMLHttpRequest`. The next thing is to return to the client side, where you can see how the XML response is used to update page state.
Response handling with JavaScript

The `readyState` property of `XMLHttpRequest` is a numeric value that gives the status of the request's lifecycle. It changes from 0 for "uninitialized" through to 4 for "complete." Each time the `readyState` changes, the `readystatechange` event fires and the handler function attached via the `onreadystatechange` property is called.

In **Listing 3**, you saw how the function `getReadyStateHandler()` was called to create a handler function. This handler function was then assigned to the `onreadystatechange` property. `getReadyStateHandler()` exploits the fact that functions are first-class objects in JavaScript. This means that functions can be parameters to other functions and can also create and return other functions. It is the job of `getReadyStateHandler()` to return a function that checks whether the `XMLHttpRequest` has completed and passes the XML response onto the handler function specified by the caller. **Listing 6** is the code for `getReadyStateHandler()`.

**Listing 6. The `getReadyStateHandler()` function**

```javascript
/*
 * Returns a function that waits for the specified XMLHttpRequest
 * to complete, then passes its XML response
 * to the given handler function.
 * req - The XMLHttpRequest whose state is changing
 * responseXmlHandler - Function to pass the XML response to
 */
function getReadyStateHandler(req, responseXmlHandler) {

    // Return an anonymous function that listens to the
    // XMLHttpRequest instance
    return function () {

        // If the request's status is "complete"
        if (req.readyState == 4) {

            // Check that a successful server response was received
            if (req.status == 200) {

                // Pass the XML payload of the response to the
                // handler function
                responseXmlHandler(req.responseXML);

            } else {

                // An HTTP problem has occurred
                alert("HTTP error: "+req.status);
            }

        }
    }
}
```

**HTTP status codes**

In **Listing 6**, the `status` property of `XMLHttpRequest` is tested to see if the request completed successfully. `status` contains the HTTP status code of the server's response. When performing simple GET and POST requests, you can assume that any code other than 200 (OK) is an error. If the server sends a redirect response (for example, 301 or 302), the browser transparently follows the redirect and fetches the resource from the new location; the `XMLHttpRequest` doesn't see the redirect status code. Also, browsers automatically add a
About getReadyStateHandler()

getReadyStateHandler() is a relatively complex piece of code, especially if you're not used to reading JavaScript. The trade-off is that by including this function in your JavaScript library, you can simply handle Ajax server responses without having to deal with the internals of XMLHttpRequest. The important thing is that you understand how to use getReadyStateHandler() in your own code.

In Listing 3, you saw getReadyStateHandler() called as follows: handlerFunction = getReadyStateHandler(req, updateCart). The function returned by getReadyStateHandler() in this case will check if the XMLHttpRequest in the variable req has completed and then call a function named updateCart with the response XML.

Extracting the cart data

Listing 7 is the code of updateCart() itself. The function interrogates the shopping cart XML document using DOM calls and updates the Web page (see Listing 1) to reflect the new cart contents. Focus here on the calls used to extract data from the XML DOM. The generated attribute on the cart element, a timestamp created when the cart was serialized to XML, is checked to ensure that newer cart data is not overwritten by older cart data. Ajax requests are inherently asynchronous, so this check safeguards against server responses that arrive out of sequence.

Listing 7. Updating the page to reflect the cart XML document

```javascript
function updateCart(cartXML) {
  // Get the root "cart" element from the document
  var cart = cartXML.getElementsByTagName("cart")[0];

  // Check that a more recent cart document hasn't been processed
  // already
  var generated = cart.getAttribute("generated");
  if (generated > lastCartUpdate) {
    lastCartUpdate = generated;

    // Clear the HTML list used to display the cart contents
    var contents = document.getElementById("cart-contents");
    contents.innerHTML = "";

    // Loop over the items in the cart
    var items = cart.getElementsByTagName("item");
    for (var I = 0; I < items.length; I++) {
      var item = items[I];

      // Extract the text nodes from the name and quantity elements
      var name = item.getElementsByTagName("name")[0].firstChild.nodeValue;
      var quantity = item.getElementsByTagName("quantity")[0].firstChild.nodeValue;

      // Create and add a list item HTML element for this cart item
      //...
    }
  }
}
```
And with that, the whistle-stop tour of the Ajax roundtrip is complete, although you may want to get the Web app running and see it in action (see the Download section). The example is very simple, with plenty of scope for improvement. For instance, I've included server-side code to remove items from the cart, but no way to access it from the UI. For a good exercise, try building on the application's existing JavaScript code to implement this functionality.

Challenges of using Ajax

As with any technology, there are plenty of ways to make mistakes with Ajax. Some of the problems I discuss here currently lack easy solutions but will improve as Ajax matures. As the developer community gains experience developing Ajax applications, best practices and guidelines will be documented.

Availability of XMLHttpRequest

One of the biggest issues facing Ajax developers is how to respond when XMLHttpRequest isn't available. While the majority of modern browsers support XMLHttpRequest, there will always be a minority of users whose browsers do not, or whose browser security settings prevent XMLHttpRequest from being used. If you're developing a Web app to be deployed on a corporate intranet, you probably have the luxury of specifying which browsers are supported and assuming XMLHttpRequest is always available. If you're deploying on the public Web, however, you must be aware that by presuming XMLHttpRequest is available, you are potentially preventing users of older browsers, browsers for people with disabilities, or lightweight browsers on handheld devices from using your application.

Therefore, you should endeavor to make your application "degrade gracefully" and remain functional in browsers without XMLHttpRequest support. In the shopping-cart example, the best way to degrade the application would be to have the Add to Cart buttons perform a regular form submission, refreshing the page to reflect the cart's updated status. The Ajax behavior could be added to the page through JavaScript once the page was loaded, attaching a JavaScript handler function to each Add to Cart button only if XMLHttpRequest was available. Another approach would be to detect XMLHttpRequest when a user logged in, and then serve up either an Ajax version of the application or a regular forms-based version as appropriate.

Usability concerns

Some of the usability issues surrounding Ajax applications are more general. For instance, it can be important to let users know that their input has been registered, because the usual...
feedback mechanisms of the hourglass cursor and spinning browser "throbber" do not apply to XMLHttpRequests. One technique is to replace Submit buttons with a "Now updating..." type message so that users do not repeatedly click on buttons while waiting for a response.

Another issue is that users may fail to notice that parts of the page they're viewing have been updated. You can alleviate this problem by using a variety of visual techniques to subtly draw the user's eye to updated areas of the page. Other issues caused by updating the page with Ajax include "breaking" the browser's back button, and the URL in the address bar not reflecting the entire state of the page, preventing bookmarking. See the Related topics section for articles that specifically address the usability issues of Ajax applications.

Server load

Implementing an Ajax UI in place of a regular forms-based one may dramatically increase the number of requests made to the server. For instance, a regular Google Web search causes one hit on the server, occurring when the user submits the search form. However, Google Suggest, which attempts to autocomplete your search terms, sends several requests to the server as the user types. When developing an Ajax application, be aware of how many requests you'll be sending to the server and the resulting server load this will cause. You can mitigate server load by buffering requests on the client and caching server responses in the client, where applicable. You should also attempt to design your Ajax Web applications so that as much logic as possible can be performed on the client, without needing to contact the server.

Dealing with asynchrony

It's very important to understand that there is no guarantee that XMLHttpRequests will complete in the order they were dispatched. Indeed, you should assume that they will not and design your application with this in mind. In the shopping-cart example, a last-updated timestamp was used to make sure that newer cart data would not be overwritten by older data (see Listing 7). This very rudimentary approach works for the shopping-cart scenario, but may not for others. Consider at design time how you will deal with asynchronous server responses.

In conclusion

You should now have a good understanding of the fundamental principles of Ajax and a nuts-and-bolts knowledge of the client- and server-side components that participate in an Ajax interaction. These are the building blocks of a Java-based Ajax Web application. In addition, you should understand some of the high-level design issues that come with the Ajax approach. Creating a successful Ajax application requires a holistic approach -- from UI design through JavaScript design to server-side architecture -- but you should now be armed with the core Ajax knowledge needed to consider these other aspects.

There's good news if you're feeling daunted by the complexity of writing a large Ajax application using the techniques demonstrated here. Just as frameworks like Struts, Spring, and Hibernate have evolved to abstract Web application development away from the low-level details of the Servlet API and JDBC, so toolkits are appearing to ease Ajax development. Some of these focus
solely on the client side, providing easy ways to add visual effects to your pages or streamlining the use of XMLHttpRequest. Others go further, providing means to automatically generate Ajax interfaces from server-side code. These frameworks do the heavy lifting for you, so that you can take a more high-level approach to Ajax development. I'll be looking at some of them in this series.

The Ajax community is fast moving, and there's a great deal of valuable information out there. Before reading the next installment in this series, I recommend that you consult the articles listed in the Related topics section, especially if you're new to Ajax or client-side development. You should also take some time to study the example source code and think about ways to improve it.

In the next article in this series, I'll discuss the XMLHttpRequest API in more detail and suggest ways to create XML easily from your JavaBeans. I'll also show you alternatives to XML for Ajax data transfer, such as the JSON (JavaScript Object Notation) lightweight data-interchange format.
### Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample code</td>
<td>j-ajax1.zip</td>
<td>8 KB</td>
</tr>
</tbody>
</table>
Related topics

- "Ajax for Java developers: Java object serialization for Ajax" (Philip McCarthy, developerWorks, October 2005): Learn approaches to data serialization in Ajax applications.
- "Ajax and scripting Web services with E4X" (Paul Fremantle and Anthony Elder, developerWorks, April 2005): Use Ajax to make SOAP calls in browsers that support the E4X JavaScript extension.
- XMLHttpRequest Usability Guidelines: Suggestions for using Ajax to enhance user experience.
- Ajax Mistakes: Usability problems that Ajax applications should avoid.
- Mozilla Firefox: The DOM Inspector and JavaScript Debugger extension take a lot of the pain out of Ajax development.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)