Mastering Ajax, Part 7: Using XML in requests and responses

Learn when it's a good idea -- and when it isn't

Skill Level: Intermediate

Brett McLaughlin
Author and Editor
O'Reilly Media Inc.

10 Oct 2006

Even casual Ajax developers will notice the *x* in Ajax, and realize that it stands for *XML.* XML is one of the most popular data formats in any programming medium, and offers real advantages for server responses in asynchronous applications. In this article, you'll see how servers can send XML in response to a request.

You really can't do any significant programming today without running across XML. Whether you're a Web page designer considering the move to XHTML, a Web programmer working with JavaScript, a server-side programmer using deployment descriptors and data binding, or a back-end developer investigating XML-based databases, the extensible markup language is everywhere. It's no surprise, then, that XML is considered one of the core technologies that underlies Ajax.

However, this opinion reflects the poor choice of names for the core object used in Ajax applications -- `XMLHttpRequest` -- more than it does technical reality. In other words, most people think XML is a core part of Ajax because they assume that the `XMLHttpRequest` object actually uses XML all the time. But that's not the case, and the reasons why are the subject of the first part of this article. In fact, you'll see that in most Ajax applications, XML rarely makes an appearance at all.

XML does have real uses in Ajax, and `XMLHttpRequest` allows for these as well. There's certainly nothing keeping you from sending XML to a server. In earlier articles in this series, you used plain text and name/value parameters to send data, but XML is also a viable format. In this article, you'll look at how to do that. More

importantly, though, I'll talk about *why* you might use XML for your request format, and why, in many cases, you *shouldn't* use it.

XML: Is it really there at all?

It's easy to make assumptions about Ajax applications and their usage of XML; both the technology name (Ajax) and the core object it uses (`XMLHttpRequest`) imply the use of XML, and you'll hear XML linked with Ajax applications all the time. However, this perception is simply wrong, and if you want to really know your stuff when it comes to writing asynchronous applications, you need to *know* that the perception is wrong -- and, better yet, know *why* it's wrong.

XMLHttpRequest: Poor names and HTTP

One of the worst things that can happen to a technology is for it to become so hot that changing basic pieces of it becomes impossible. That's exactly what's happened with `XMLHttpRequest`, the basic object used in Ajax apps. It sounds like it's designed to either send XML over HTTP requests, or perhaps make HTTP requests in some sort of XML format. Whatever the object's name *sounds* like, though, what it actually *does* is simply provide a way for your client code (usually JavaScript in your Web page) to send an HTTP request. That's it; there's really nothing more to it.

Thus, it would be nice to simply change `XMLHttpRequest`'s name to something more accurate, like `HttpRequest`, or perhaps simply `Request`. However, millions of developers are now throwing Ajax into their applications, and because we all know that it takes years -- if not decades -- for the majority of users to move to new browser versions like Internet Explorer 7.0 or Firefox 1.5, such a move is simply not feasible. The end result is that you're stuck with `XMLHttpRequest`, and it's up to developers to realize that the thing is just poorly named.

It's somewhat telling that one of the best known fallback methods for dealing with a browser (especially on Windows) that doesn't support `XMLHttpRequest` is to use the Microsoft `IFRAME` object. Hardly sounds like XML, HTTP, or even a request, does it? Obviously, all those things might be involved, but this should simply make clear the fact that the `XMLHttpRequest` object is a lot more about making requests without requiring a page reload than it is about XML, or even HTTP.

The requests are HTTP, not XML

Another common mistake is to suppose that XML is somehow used behind the scenes -- a view I once held myself, to be honest! However, that view reflects a poor understanding of the technology. When a user opens a browser and requests a Web page from a server, they type in something like `http://www.google.com` or `http://www.headfirstlabs.com`. Even if they don't include the `http://`, the browser will fill in that part in the browser address bar. That first part -- `http://` -- is

a not-so-subtle clue about how communication is occurring: through HTTP, the Hypertext Transfer Protocol. When you write code in your Web page to communicate with a server, whether it's using Ajax or a normal form POST or even a hyperlink, you're just talking HTTP.

> **HTTPS: Still HTTP**
> Those of you newer to the Web might wonder about URLs like `https://intranet.nextel.com`. The `https` is secure HTTP, and just uses a more secure form of the HTTP protocol used by ordinary Web requests. So even with HTTPS, you're still just talking HTTP, albeit with some extra layers of security added to keep away prying eyes.

Given that pretty much all Web communication between browsers and servers takes place through HTTP, the idea that XML is somehow the transport or technology used by `XMLHttpRequest` under the covers just doesn't make any sense. It's certainly possible for XML to be sent *in* the HTTP request, but HTTP is a very precisely defined standard that isn't going away any time soon. Unless you're specifically using XML in your request, or the server is sending you a response in XML, there's nothing but plain old HTTP used in the `XMLHttpRequest` object. So the next time someone tells you, "Yeah, it's called `XMLHttpRequest` because it uses XML behind the scenes," just smile and patiently explain to them what HTTP is, and let them know that while XML can be sent *over* HTTP, XML is a data format, not a transfer protocol. You'll both be the better for the explanation.

Using XML (for real)

So far, I've told you about all the places where XML *isn't* used in Ajax. But the *x* in Ajax and the `XML` in `XMLHttpRequest` are still very real, and you've several options for using XML in your Web applications. You'll look at the basic options in this section, and then really dig into detail in the rest of this article.

Options for XML

In your asynchronous apps, you'll find two basic applications of XML:

- To send a request from a Web page to a server in XML format
- To receive a request from a server in your Web page in XML format

The first of these -- to send a request in XML -- requires you to format your request as XML, either using an API to do so or just stringing together the text, and then sending the result to a server. In this option, the main job at hand is to construct the request in a way that complies with the rules of XML, and that can be understood by the server. So the focus is really on the XML format; you have the data you want to

send, and just need to wrap it up in XML semantics. The rest of this article focuses on this use of XML in your Ajax applications.

The second of these options -- to receive a request in XML -- requires you to take a response from a server, and extract the data from XML (again, using either an API or more of a brute force approach). In this case, your focus is on the data from the server, and it just so happens that you've got to pull that data out of XML to use it in any constructive way. This is the subject of the next article in this series, and you'll really dig into that in detail then.

A preemptory warning

Before I get into the details of using XML, a short cautionary word is in order: XML is not a small, fast, space-saving format. As you'll see in the next several sections and in the next article in this series, there are some great reasons to use XML in this context, and some advantages that XML has over plain text requests and responses (especially for responses). However, XML is almost always going to take up more space and be slower than plain text, because you add all the tags and semantics required for XML to your messages.

If you want to write a blazing fast application that feels like a desktop app, XML might not be the best place to start. If you begin with plain text, and find a specific need for XML, then that's great; however, if you use XML from the beginning, you almost certainly slow down your application's responsiveness. In most cases, it's faster to send plain text -- using name/value pairs like `name=jennifer` -- than to turn the text into XML like this:

```
<name>jennifer</name>
```

Think of all the places where using XML adds time: wrapping the text in XML; sending across extra information (note that I didn't include any surrounding elements, an XML header, or anything else that would probably be part of a more realistic request); having the server parse the XML, generate a response, wrap the response back in XML, and send it *back* to your Web page; and then having your page parse the response and finally use it. So learn when to use XML, but don't start out by thinking that it's going to make your application faster in many situations; rather, it adds flexibility, as we'll begin to talk about now.

XML from the client to the server

Let's look at using XML as the format to send data from a client to a server. First, you'll see how to do this technically, and then spend some time examining when this is a good idea, and when it's not.

Sending name/value pairs

In about 90 percent of the Web apps you write, you'll end up with name/value pairs to send to a server. For example, if a user types their name and address into a form on your Web page, you might have data like this from the form:

```
firstName=Larry
lastName=Gullahorn
street=9018 Heatherhorn Drive
city=Rowlett
state=Texas
zipCode=75080
```

If you were just using plain text to send this data to a server, you might use code that looks something like Listing 1. (This is similar to an example I used in the first article in this series. See Resources.)

Listing 1. Sending name/value pairs in plain text

```
function callServer() {
  // Get the city and state from the Web form
  var firstName = document.getElementById("firstName").value;
  var lastName = document.getElementById("lastName").value;
  var street = document.getElementById("street").value;
  var city = document.getElementById("city").value;
  var state = document.getElementById("state").value;
  var zipCode = document.getElementById("zipCode").value;

  // Build the URL to connect to
  var url = "/scripts/saveAddress.php?firstName=" + escape(firstName) +
    "&lastName=" + escape(lastName) + "&street=" + escape(street) +
    "&city=" + escape(city) + "&state=" + escape(state) +
    "&zipCode=" + escape(zipCode);

  // Open a connection to the server
  xmlHttp.open("GET", url, true);

  // Set up a function for the server to run when it's done
  xmlHttp.onreadystatechange = confirmUpdate;

  // Send the request
  xmlHttp.send(null);
}
```

Converting name/value pairs to XML

The first thing you need to do if you want to use XML as a format for data like this is to come up with some basic XML format in which to store the data. Obviously, your name/value pairs can all turn into XML elements, where the element name is the name of the pair, and the content of the element is the value:

```
<firstName>Larry</firstName>
<lastName>Gullahorn</lastName>
<street>9018 Heatherhorn Drive</street>
```

```
<city>Rowlett</city>
<state>Texas</state>
<zipCode>75080</zipCode>
```

Of course, XML requires that you have a root element, or, if you're just working with a *document fragment* (a portion of an XML document), an enclosing element. So you might convert the XML above to something like this:

```
<address>
  <firstName>Larry</firstName>
  <lastName>Gullahorn</lastName>
  <street>9018 Heatherhorn Drive</street>
  <city>Rowlett</city>
  <state>Texas</state>
  <zipCode>75080</zipCode>
</address>
```

Now you're ready to create this structure in your Web client, and send it to the server ... almost.

Communication, of the verbal kind

Before you're ready to start tossing XML over the network, you want to make sure that the server -- and script -- to which you send data actually accepts XML. Now for many of you, this might seem like a silly and obvious point to make, but plenty of newer programmers just assume that if they send XML across the network, it is received and interpreted correctly.

In fact, you need to take two steps to ensure that the data you send in XML will be received correctly:

1. Ensure that the script to which you send the XML accepts XML as a data format.
2. Ensure the script will accept the particular XML format and structure in which you send data.

Both of these will probably require you to actually talk to a human being, so fair warning! Seriously, if it's important that you be able to send data as XML, most script writers will oblige you; so just finding a script that will accept XML shouldn't be that hard. However, you'still need to make sure that the your format matches what the script expects. For example, suppose the server accepts data like this:

```
<profile>
  <firstName>Larry</firstName>
  <lastName>Gullahorn</lastName>
  <street>9018 Heatherhorn Drive</street>
  <city>Rowlett</city>
```

```
  <state>Texas</state>
  <zip-code>75080</zip-code>
</profile>
```

This looks similar to the XML above, except for two things:

1. The XML from the client is wrapped within an `address` element, but the server expects the data to be wrapped within a `profile` element.
2. The XML from the client uses a `zipCode` element, while the server expects the zip code to be in a `zip-code` element.

In the grand scheme of things, these really small points are the difference between the server accepting and processing your data, and the server crashing miserably and supplying your Web page -- and probably its users -- with a cryptic error message. So you've got to figure out what the server expects, and mesh the data you send into that format. Then -- and only then -- are you ready to deal with the actual technicalities of sending XML from a client to a server.

Sending XML to the server

When it comes to sending XML to the server, you'll spend more of your code taking your data and wrapping it XML than you will actually transmitting the data. In fact, once you have the XML string ready to send to the server, you send it exactly as you would send any other plain text; check out Listing 2 to see this in action.

Listing 2. Sending name/value pairs in XML

```
function callServer() {
  // Get the city and state from the Web form
  var firstName = document.getElementById("firstName").value;
  var lastName = document.getElementById("lastName").value;
  var street = document.getElementById("street").value;
  var city = document.getElementById("city").value;
  var state = document.getElementById("state").value;
  var zipCode = document.getElementById("zipCode").value;

  var xmlString = "<profile>" +
    "  <firstName>" + escape(firstName) + "</firstName>" +
    "  <lastName>" + escape(lastName) + "</lastName>" +
    "  <street>" + escape(street) + "</street>" +
    "  <city>" + escape(city) + "</city>" +
    "  <state>" + escape(state) + "</state>" +
    "  <zip-code>" + escape(zipCode) + "</zip-code>" +
    "</profile>";

  // Build the URL to connect to
  var url = "/scripts/saveAddress.php";

  // Open a connection to the server
  xmlHttp.open("POST", url, true);

  // Tell the server you're sending it XML
  xmlHttp.setRequestHeader("Content-Type", "text/xml");
```

```
  // Set up a function for the server to run when it's done
  xmlHttp.onreadystatechange = confirmUpdate;

  // Send the request
  xmlHttp.send(xmlString);
}
```

Much of this is self-explanatory with just a few points worth noting. First, the data in your request must be manually formatted as XML. That's a bit of a letdown after three articles on using the Document Object Model, isn't it? And while nothing forbids you from using the DOM to create an XML document using JavaScript, you'd then have to convert that DOM object to text before sending it over the network with a GET or POST request. So it turns out to be easier to simply format the data using normal string manipulation. Of course, this introduces room for error and typographical mistakes, so you need to be extra careful when you write code that deals with XML.

Once you construct your XML, you open a connection in largely the same way as you would when you send text. I tend to prefer using POST requests for XML, since some browsers impose a length limitation on GET query strings, and XML can get pretty long; you'll see that Listing 2 switches from GET to POST accordingly. Additionally, the XML is sent through the `send()` method, rather than as a parameter tacked on to the end of the URL you're requesting. These are all fairly trivial differences, though, and easy to adjust for.

You will have to write one entirely new line of code, though:

```
xmlHttp.setRequestHeader("Content-Type", "text/xml");
```

This isn't hard to understand: it just tells the server that you're sending it XML, rather than plain old name/value pairs. In either case, you send data as text, but use `text/xml` here, or XML sent as plain text. If you just used name/value pairs, this line would read:

```
xmlHttp.setRequestHeader("Content-Type", "text/plain");
```

If you forget to tell the server that you're sending it XML, you'll have some trouble, so don't forget this step.

Once you get all this put together, all you need to do is call `send()` and pass in the XML string. The server will get your XML request, and (assuming you've done your pre-work) accept the XML, parse it, and send you back a response. That's really all there is to it -- XML requests with just a few changes of code.

Sending XML: Good or bad?

Before leaving XML requests (and this article) for XML responses, let's spend some real time thinking about the sensibility of using XML in your requests. I've already mentioned that XML is by no means the fastest data format in terms of transfer, but there's a lot more to think about.

XML is not simple to construct

The first thing you need to realize is that XML is just not that easy to construct for use in requests. As you saw in Listing 2, your data quickly becomes pretty convoluted with the semantics of XML:

```
var xmlString = "<profile>" +
  "  <firstName>" + escape(firstName) + "</firstName>" +
  "  <lastName>" + escape(lastName) + "</lastName>" +
  "  <street>" + escape(street) + "</street>" +
  "  <city>" + escape(city) + "</city>" +
  "  <state>" + escape(state) + "</state>" +
  "  <zip-code>" + escape(zipCode) + "</zip-code>" +
  "</profile>";
```

This might not seem so bad, but it's also an XML fragment that has only six fields. Most of the Web forms you'll develop will have ten to fifteen; although you won't use Ajax for all of your requests, it is a consideration. You're spending at least as much time dealing with angle brackets and tag names as you are with actual data, and the potential to make little typos is tremendous.

Another problem here is that -- as already mentioned -- you will have to construct this XML by hand. Using the DOM isn't a good option, as there aren't good, simple ways to turn a DOM object into a string that you can send as a request. So working with strings like this is really the best option -- but it's also the option that's hardest to maintain, and hardest to understand for new developers. In this case, you constructed all the XML in one line; things only get more confusing when you do this in several steps.

XML doesn't add anything to your requests

Beyond the issue of complexity, using XML for your requests really doesn't offer you much of an advantage -- if any -- over plain text and name/value pairs. Consider that everything in this article has been focused on taking the same data you could already send using name/value pairs (refer back to Listing 1) and sending it using XML. At no point was anything said about data that you can send with XML that you *could not* send using plain text; that's because there almost never *is* anything that you can send using XML that you can't send using plain text.

And that's really the bottom line with XML and requests: there's just rarely a

compelling reason to do it. You'll see in the next article in this series that a server can use XML for some things that are much harder to do when using plain text; but it's just not the case with requests. So unless you're talking to a script that *only* accepts XML (and there are some out there), you're better off using plain text in almost every request situation.

In conclusion

You should definitely feel like you're starting to get the XML in Ajax figured out. You know that Ajax apps don't have to use XML, and that XML isn't some sort of magic bullet for data transfer. You should also feel pretty comfortable in sending XML from a Web page to a server. Even more importantly, you know what's involved in making sure that a server will actually handle and respond to your requests: you've got to ensure that the server script accepts XML, and that it accepts it in the format that you're using to send the data over.

You also should have a good idea now of why XML isn't always that great a choice for a data format for requests. In future articles, you'll see some cases where it helps, but in most requests, it simply slows things down and adds complexity. So while I'd normally suggest that you immediately start using the things you learned in an article, I'll instead suggest that you be very careful about using what you've learned here. XML requests have their place in Ajax apps, but that place isn't as roomy as you might think.

In the next article in this series, you'll look at how servers can respond using XML, and how your Web applications can handle those responses. Happily, there's a much larger number of reasons for a server to send XML back to a Web app than the other way around, so you'll get even more use out of that article's technical detail; for now, be sure you understand why XML isn't always a great idea -- at least for sending requests. You might even want to try and implement some Web apps using XML as the data format for requests, and then convert back to plain text, and see which seems both faster and easier to you. Until next article, I'll see you online.

Resources

Learn

- Mastering Ajax: Read the previous articles in this series.
- XML: See developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- xml.com: Start with one of the easiest-to-understand online resources for everything XML if you're not already an experienced XML programmer.
- "Build dynamic Java applications" (Philip McCarthy, developerWorks, September 2005): Take a look at Ajax from the server side, using a Java™ perspective.
- "Java object serialization for Ajax" (Philip McCarthy, developerWorks, October 2005): Examine how to send objects over the network, and interact with Ajax, from a Java perspective.
- "Call SOAP Web services with Ajax" (James Snell, developerWorks, October 2005): Dig into this fairly advanced article on integrating Ajax with existing SOAP-based Web services; it shows you how to implement a Web browser-based SOAP Web services client using the Ajax design pattern.
- The DOM Home Page at the World Wide Web Consortium: Visit the starting place for all things DOM-related.
- The DOM Level 3 Core Specification: Define the core Document Object Model, from the available types and properties to the usage of the DOM from various languages.
- The ECMAScript language bindings for DOM: If you're a JavaScript programmer and want to use the DOM from your code, this appendix to the Level 3 Document Object Model Core definitions will interest you.
- "Ajax: A new approach to Web applications" (Jesse James Garrett , *Adaptive Path,* February 2005): Read the article that coined the Ajax moniker -- it's required reading for all Ajax developers.
- developerWorks technical events and webcasts: Stay current with these software briefings for technical developers.
- developerWorks Web development zone: Expand your Web-building skills with articles, tutorials, forums, and more.

Get products and technologies

- *Head Rush Ajax,* Brett McLaughlin (O'Reilly Media, 2006): Load the ideas in this article into your brain, Head First style.

- *Java and XML, Second Edition* (Brett McLaughlin, O'Reilly Media, Inc., 2001): Check out the author's discussion of XHTML and XML transformations.
- *JavaScript: The Definitive Guide* (David Flanagan, O'Reilly Media, Inc., 2001): Dig into extensive instruction on working with JavaScript and dynamic Web pages. The upcoming edition adds two chapters on Ajax.
- *Head First HTML with CSS & XHTML* (Elizabeth and Eric Freeman, O'Reilly Media, Inc., 2005): Learn more about standardized HTML and XHTML, and how to apply CSS to HTML.
- IBM trial software: Build your next development project with software available for download directly from developerWorks.

Discuss

- developerWorks blogs: Get involved in the developerWorks community.
- Ajax forum on developerWorks: Learn, discuss, share in this forum of Web developers just learning or actively using AJAX.

About the author

Brett McLaughlin

Brett McLaughlin has worked in computers since the Logo days. (Remember the little triangle?) In recent years, he's become one of the most well-known authors and programmers in the Java and XML communities. He's worked for Nextel Communications, implementing complex enterprise systems; at Lutris Technologies, actually writing application servers; and most recently at O'Reilly Media, Inc., where he continues to write and edit books that matter. Brett's upcoming book, *Head Rush Ajax*, brings the award-winning and innovative Head First approach to Ajax. His last book, *Java 1.5 Tiger: A Developer's Notebook*, was the first book available on the newest version of Java technology. And his classic *Java and XML* remains one of the definitive works on using XML technologies in the Java language.