Mastering Ajax, Part 8: Using XML in requests and responses

Ajax client/server communication can be a tricky business

Skill Level: Intermediate

Brett McLaughlin
Author and Editor
O'Reilly Media Inc.

07 Nov 2006

In the last article of the series, you saw how your Ajax apps can format requests to a server in XML. You also saw why, in most cases, that isn't a good idea. This article focuses on something that often *is* a good idea: returning XML responses to a client.

I don't really enjoy writing articles that are primarily about something that you *shouldn't* do. Most of the time, it's a pretty silly type of thing to write. I spend half an article explaining something, just so I can spend the rest of the article explaining what a bad idea it is to use the techniques you've just learned about. Such was the case, to a large degree, with last month's article (if you missed it, check out the link in Resources), which taught you how to use XML as the data format for your Ajax apps' requests.

Hopefully, this article will redeem the time you spent learning about XML requests. In Ajax apps, while there are very few reasons to use XML as the sending data format, there are a lot of reasons why you might want a server to send XML *back* from a server, *to* a client. So everything you learned about XML in the last article will definitely start to have some value in this article.

Servers can't say much (sometimes)

Before you dive into the technical details of getting an XML response from a server, you need to understand why it's such a good idea for a server to send XML in

response to a request (and how that's different from a client sending that request in XML).

Clients speak in name/value pairs

As you'll recall from the last article, clients don't need to use XML in most cases because they can send requests using name/value pairs. So you might send a name like this: `name=jennifer`. You can stack those up by simply adding an ampersand (&) between successive name/value pairs, like this: `name=jennifer&job=president`. Using simple text and these name/value pairs, clients can send requests with multiple values to a server easily. There's rarely a need for the additional structure (and overhead) that XML provides.

In fact, almost all the reasons you'd need to send XML to a server can be grouped into two basic categories:

- **The server *only* accepts XML requests.** In these cases, you don't have a choice. The basics in last month's article should give you all the tools you need to send these sorts of requests.
- **You're calling a remote API that only accepts XML or SOAP requests.** This is really just a specialized case of the previous point, but it's worth mentioning on its own. If you want to use the APIs from Google or Amazon in an asynchronous request, there are some particular considerations. I'll look at those, and a few examples of making requests to APIs like this, in next month's article.

Servers can't send name/value pairs (in a standard way)

When you send name/value pairs, the Web browser sending the requests and the platform responding to that request and hosting a server program cooperate to turn those name/value pairs into data that a server program can work with easily. Practically every server-side technology -- from Java™ servlets to PHP to Perl to Ruby on Rails -- allows you to call a variety of methods to get at values based on a name. So getting the `name` attribute is trivial.

This isn't the case going in the other direction. If a server replied to an app with the string `name=jennifer&job=president`, the client has no standardized, easy way to break up the two name/value pairs, and then break each pair into a name and value. You'll have to parse the returned data manually. If a server returns a response made up of name/value pairs, that response is no easier (or harder) to interpret than a response with elements separated by semicolons, or pipe symbols, or any other nonstandard formatting character.

> **Give me some space!**
> In most HTTP requests, the escape sequence `%20` is used to represent a single space. So the text "Live Together, Die Alone" is

sent over HTTP as `Live%20Together,%20Die%20Alone`.

What that leaves you with, then, is no easy way to use plain text in your responses and have the client get that response and interpret it in a standard way, at least when the response contains multiple values. If your server simply sent back the number 42, say, plain text would be great. But what about if it's sending back the latest ratings for the TV shows *Lost, Alias,* and *Six Degrees,* all at once? While you can chose many ways to send this response using plain text (see Listing 1 for a few examples), none are particularly easy to interpret without some work by the client, and none are standardized at all.

Listing 1. Server response for TV ratings (various versions)

```
show=Alias&ratings=6.5|show=Lost&ratings=14.2|show=Six%20Degrees&ratings=9.1

Alias=6.5&Lost=14.2&Six%20Degrees=9.1

Alias|6.5|Lost|14.2|Six%20Degrees|9.1
```

Even though it's not too hard to figure out how to break up these response strings, a client will have to parse and split the string up based on the semicolons, equal signs, pipes, and ampersands. This is hardly the way to write robust code that other developers can easily understand and maintain.

Enter XML

When you realize that there's no standard way for a server to respond to clients with name/value pairs, the reasoning behind using XML becomes pretty clear. When sending data to the server, name/value pairs are a great choice because servers and server-side languages can easily interpret the pairs; the same is true for using XML when returning data to a client. You saw the use of the DOM to parse XML in several earlier articles, and will see how JSON provides yet another option to parse XML in a future article. And on top of all that, you can treat XML as plain text, and get values out of it that way. So there are several ways to take an XML response from a server, and, with fairly standard code, pull the data out and use it in a client.

As an added bonus, XML is generally pretty easy to understand. Most people who program can make sense of the data in Listing 2, for example.

Listing 2. Server response for TV ratings (in XML)

```
<ratings>
 <show>
  <title>Alias</title>
  <rating>6.5</rating>
 </show>
 <show>
```

```
  <title>Lost</title>
  <rating>14.2</rating>
 </show>
 <show>
  <title>Six Degrees</title>
  <rating>9.1</rating>
 </show>
</ratings>
```

The code in Listing 2 has no mystery about what a particular semicolon or apostrophe means.

Receiving XML from a server

Because the focus of this series is on the client side of the Ajax equation, I won't delve into much detail about how a server-side program can generate a response in XML. However, you need to know about some special considerations when your client receives XML.

First, you can treat an XML response from a server in two basic ways:

- As plain text that just happens to be formatted as XML
- As an XML document, represented by a DOM `Document` object.

Second, presume a simple response XML from a server for example's sake. Listing 3 shows the same TV listings as detailed above (this is, in fact, the same XML as in Listing 2, reprinted for your convenience). I'll use this sample XML in the discussions in this section.

Listing 3. XML-formatted TV ratings for examples

```
<ratings>
 <show>
  <title>Alias</title>
  <rating>6.5</rating>
 </show>
 <show>
  <title>Lost</title>
  <rating>14.2</rating>
 </show>
 <show>
  <title>Six Degrees</title>
  <rating>9.1</rating>
 </show>
</ratings>
```

Dealing with XML as plain text

The easiest option to handle XML, at least in terms of learning new programming

techniques, is to treat it like any other piece of text returned from a server. In other words, you basically ignore the data format, and just grab the response from the server.

In this situation, you use the `responseText` property of your request object, just as you would when the server sends you a non-XML response (see Listing 4).

Listing 4. Treating XML as a normal server response

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var response = request.responseText;

      // response has the XML response from the server
      alert(response);
    }
  }
}
```

In this code fragment, `updatePage()` is the callback, and `request` is the `XMLHttpRequest` object. You end up with the XML response, all strung together, in the `response` variable. If you printed out that variable, you'd have something like Listing 5. (Note that the code in Listing 5 normally is one, continuous line. Here, it is shown on multiple lines for display purposes.)

Listing 5. Value of response variable

```
<ratings><show><title>Alias</title><rating>6.5</rating>
</show><show><title>Lost</title><rating>14.2</rating></show><show>
<title>Six Degrees</title><rating>9.1</rating></show></ratings>
```

The most important thing to note here is that the XML is all run together. In most cases, servers will not format XML with spaces and carriage returns; they'll just string it all together, like you see in Listing 5. Of course, your apps don't care much about spacing, so this is no problem; it does make it a bit harder to read, though.

Review earlier articles
To avoid lots of repetitive code, these later articles in the series only show the portions of code relevant to the subject being discussed. So Listing 4 only shows the callback method in your Ajax client's code. If you're unclear on how this method fits into the larger context of an asynchronous app, you should review the first several articles in the series, which cover the fundamentals of Ajax apps. See Resources for links to those earlier articles.

At this point, you can use the JavaScript `split` function to break up this data, and basic string manipulation to get at the element names and their values. Of course, that's a pretty big pain, and it ignores the handy fact that you spent a lot of time

looking at the DOM, the Document Object Model, earlier in this series. So I'll urge you to keep in mind that you can use and output a server's XML response easily using `responseText`, but I wont show you much more code; you shouldn't use this approach to get at the XML data when you can use the DOM, as you'll see next.

Treating XML as XML

While you *can* treat a server's XML-formatted response like any other textual response, there's no good reason to do so. First, if you've read this series faithfully, you know how to use the DOM, a JavaScript-friendly API with which you can manipulate XML. Better yet, JavaScript and the `XMLHttpRequest` object provide a property that is perfect for getting the server's XML response, and getting it in the form of a DOM `Document` object.

To see this in action, check out Listing 6. This code is similar to Listing 4, but rather than use the `responseText` property, the callback uses the `responseXML` property instead. This property, available on `XMLHttpRequest`, returns the server's response in the form of a DOM document.

Listing 6. Treating XML as XML

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var xmlDoc = request.responseXML;

      // work with xmlDoc using the DOM
    }
  }
}
```

Now you have a DOM `Document`, and you can work with it just like any other XML. For example, you might then grab all the `show` elements, as in Listing 7.

Listing 7. Grabbing all the show elements

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var xmlDoc = request.responseXML;

      var showElements = xmlDoc.getElementsByTagName("show");
    }
  }
}
```

If you're familiar with DOM, this should start to feel familiar. You can use all the DOM methods you've already learned about, and easily manipulate the XML you received from the server.

You can also, of course, mix in normal JavaScript code. For instance, you might iterate through all the show elements, as in Listing 8.

Listing 8. Iterating through all the show elements

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var xmlDoc = request.responseXML;

      var showElements = xmlDoc.getElementsByTagName("show");
      for (var x=0; x<showElements.length; x++) {
        // We know that the first child of show is title, and the second is rating
        var title = showElements[x].childNodes[0].value;
        var rating = showElements[x].childNodes[1].value;

        // Now do whatever you want with the show title and ratings
      }
    }
  }
}
```

With this relatively simple code, you treated an XML response like it's XML, not just plain unformatted text, and used a little DOM and some simple JavaScript to deal with a server's response. Even more importantly, you worked with a standardized format -- XML -- instead of comma-separated values or pipe-delimited name/value pairs. In other words, you used XML where it made sense, and avoided it when it didn't, like in sending requests to the server.

XML on the server: A brief example

Although I haven't talked much about how to generate XML on the server, it's worth seeing a brief example, without much commentary, just so you can come up with your own ideas on how to deal with such a situation. Listing 9 shows a simple PHP script that outputs XML in response to a request, presumably from an asynchronous client.

This is the brute force approach, where the PHP script is really just pounding out the XML output manually. You can find a variety of toolkits and APIs for PHP and most other server-side languages that also allow you to generate XML responses. In any case, this at least gives you an idea of what server-side scripts that generate and reply to requests with XML look like.

Listing 8. PHP script that returns XML

```
<?php

// Connect to a MySQL database
$conn = @mysql_connect("mysql.myhost.com", "username", "secret-password");
if (!conn)
  die("Error connecting to database: " . mysql_error());
```

```
if (!mysql_select_db("television", $conn))
  die("Error selecting TV database: " . mysql_error());

// Get ratings for all TV shows in database
$select = 'SELECT title, rating';
$from   = '  FROM ratings';
$queryResult = @mysql_query($select . $from);
if (!$queryResult)
  die("Error retrieving ratings for TV shows.');

// Let the client know we're sending back XML
header("Content-Type: text/xml");
echo "<?xml version=\"1.0\" encoding=\"utf-8\"?>";
echo "<ratings>";

while ($row = mysql_fetch_array($queryResult)) {
  $title = $row['title'];
  $rating = $row['rating'];

  echo "<show>
  echo "<title>" . $title . "</title>";
  echo "<rating>" . $rating . "</rating>";
  echo "</show>";
}

echo "</ratings>";

mysql_close($conn);

?>
```

You should be able to output XML in a similar way using your own favorite server-side language. A number of articles on IBM developerWorks can help you figure out how to generate an XML document using your preferred server-side language (see Resources for links).

Other options for interpreting XML

One very popular options for dealing with XML, beyond treating it as unformatted text or using the DOM, is important and worth mentioning. That's *JSON,* short for JavaScript Object Notation, and it's a free text format that is bundled into JavaScript. I don't have room to cover JSON in this article, so I'll come back to it in just a few months; you'll probably hear about it as soon as you mention XML and Ajax apps, however, so now you'll know what your co-workers are talking about.

In general, everything that you can do with JSON, you can do with the DOM, or vice versa; it's mostly about preference, and choosing the right approach for a specific application. For now, stick with the DOM, and get familiar with it in the context of receiving a server's response. In a couple of articles, I'll spend a good amount of time on JSON, and then you'll be ready to choose between the two on your next app. So stay tuned: lots more XML is coming in the next couple of articles.

In conclusion

I've talked about XML nearly non-stop since the last article in this series began, but have still really only scratched the surface of XML's contribution to the Ajax equation. In my next article, you'll look in more detail at those particular occasions in which you *would* want to send XML (and see in which of those cases you'll need to receive XML back as well). In particular, you'll examine Web services -- both proprietary ones and APIs like Google -- in light of Ajax interaction.

Your biggest task in the short term, though, is to really think about when XML makes sense for your own applications. In many cases, if your app is working well, then XML is nothing more than a technology buzzword that can cause you headaches, and you should resist the temptation to use it just so you can say you have XML in your application.

If you've a situation where the data the server sends you is limited, though, or in a strange comma- or pipe-delimited format, then XML might offer you real advantages. Consider working with or changing your server-side components so that they return responses in a more standard way, using XML, rather than a proprietary format that almost certainly isn't as robust as XML.

Most of all, realize that the more you learn about the technologies around Ajax, the more careful you have to be about your decisions. It's fun to write these Web 2.0 apps (and in coming articles, you'll return to the user interface and see some of the cool things that you can do), but it also takes some caution to make sure you don't throw technologies at a working Web page just to impress your friends. I know you can write a good app, so go out and do just that. When you're finished, come back here for next month's article, and even more XML.

Resources

Learn

- Mastering Ajax: Read the previous articles in this series.
- developerWorks XML zone: See developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- developerWorks Web Development zone: Find resources for Web 2.0, Ajax, wikis, PHP, mashups, and other Web projects.
- developerWorks Open source zone: Explore resources for open source development and implementation.
- Cache in with JSON (Bakul L. Patel, developerWorks, October 2006): Learn to cache validation metadata on the client side.
- xml.com: Start with one of the easiest-to-understand online resources for everything XML if you're not already an experienced XML programmer.
- "Write XML documents with StAX" (Berthold Daum, developerWorks, December 2003): Read this short tip on just one way to create XML documents efficiently with the low-level, cursor-based StAX API.
- "Servlets and XML: Made for each other" (Doug Tidwell, developerWorks, May 2000): In this article, learn how Java servlets can work with XML, and generate XML from the server side.
- "Using Python modules xml2sql and dtd2sql" (David Mertz, developerWorks, June 2001): Generate SQL statements to create and fill a database in this demo of a couple of the more popular XML-related modules from Python.
- "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.

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.