Using Ajax with Web services
Combining two leading-edge technologies is easier than you think

Brian M. Carey
May 18, 2010

Web services are a great way to expose functionality in a language- and platform-independent manner. Asynchronous JavaScript and XML (Ajax) is a technical means to access the contents of another resource without invoking a new request on the current Web page. Using the two together, Web developers can create powerful applications that leverage state-of-the-art technologies and provide an enhanced user experience.

What is a Web service?

A Web service is a way of exposing application functions using the World Wide Web. It does this by using open protocols, so any consuming application that has access to the Web can also access the Web service.

A typical Web service interaction consists of a consumer (the remote application that uses the Web service) sending an XML message to the service. The service then parses the message and sends back a response, also in XML format. The consumer then parses the response and, in some fashion, uses the information retrieved from the service.

Frequently, the XML language that is used in Web services is SOAP. SOAP was originally an acronym for Simple Object Access Protocol, but for some reason that is no longer the case. The protocol consists of three parts: the envelope, data type rules, and a means of defining operation requests and responses.

Web services are defined with another XML document, known as a Web Services Description Language (WSDL). The WSDL specifies the operations that are exposed using the Web service,
the data type definitions used by the operations, the protocol used to communicate with the Web service, and the location of the Web service itself.

The advantage to Web services is that they enable applications that are written in different programming languages and deployed on different platforms to communicate with one another over the vast expanse of the Internet.

What is Ajax?

Ajax is one of the latest bleeding-edge technologies Web developers use to enable rich client presentation. It accomplishes this by invoking a new request without disrupting the current view. An XML document is returned that is then displayed to the user, frequently as a subpage within the current presentation. In short, Ajax gives you the benefit of server-side dynamic content while still looking like client-side dynamic content.

Ajax generally fulfills its billing through the use of the XMLHttpRequest DOM API, which, until the advent of Ajax, Web developers rarely used. The request itself can be either one of the GET or POST varieties.

As with any other request, a response is returned, which can be an error. If the response is not erroneous, the actual text of the response is used to update the current view.

Remembering a famous statement by Voltaire (namely, "the Holy Roman Empire was neither holy, nor Roman, nor an empire"), thorough research into various Ajax implementations leads one to realize that Ajax does not require JavaScript code, does not require XML, and does not need to be asynchronous. After stripping all of that out, the only part left in the acronym is the conjunction (and). But the acronym sounds cool, so the industry has decided to keep it.

How Ajax and Web services fit together

Think of it this way: a rich client experience coupled with services accessible anywhere on the Internet. Yeah, that's pretty cool.

Ajax, you have seen, executes a request under the covers and usually spits out the response (or some part of it) back to the Web page without an entire refresh of the page. Now, while that request can be a simple HTTP request, it can also be a SOAP message sent to an exposed Web service. The JavaScript side of the Ajax routine can then parse the response (also in SOAP format) and extract the necessary data that is returned to the application and presented to the user.

It's really that simple.

The business requirements for fishinhole.com

The board of directors at fishinhole.com wants you to make the company's inventory more readily accessible to other Web applications. They think that if other Web sites, including sport fishing forums, blogs, and even competing tackle retailers can easily access a list of fishinhole.com's inventory, it will increase sales by 23.7%.
You resist the temptation to ponder how they arrived at the 23.7% figure and instead focus on how you can expose your inventory to other Web applications. It doesn't take you long to reach the conclusion that you need to create a Web service. The Web service enables calling consumers to submit a request for an inventory of lures based on the lure type. The lure types currently available are Casting, Trolling, and Other. Your Web service returns a list of lures based on the type of lure provided by the consumer.

You also realize that, for now, your company is out of Other type lures. You have to handle that gracefully in your Web service.

### Deploying a simple Web service

You use PHP to create the simple Web service. PHP, together with NuSOAP, is one of the easiest means that I have ever seen for creating a Web service quickly.

First, make sure you grab NuSOAP (see Related topics), and put all of the PHP files in the same directory where your PHP Web service will be deployed.

After you have NuSOAP installed, it's time to start writing the actual Web service. Listing 1 contains the whole thing.

#### Listing 1. webservice.php

```php
<?php
require_once('nusoap.php');
$server = new soap_server;
$server->register('hello');
$server->register('retrieveByType');
function hello($name) {
    return 'Hello, ' . $name;
}
function retrieveByType($type) {
    if ($type == 'trolling') {
        $arr[0] = 'Donzai Deep Swimmer 5 1/4 inch';
        $arr[1] = 'Yosubi Squid-like 4 inch';
        $arr[2] = 'Fortunata Imperial High Action';
    } else if ($type == 'casting') {
        $arr[0] = 'Silver Spring Mirrors Size 00';
        $arr[1] = 'Gold Spring Mirrors Size 0';
        $arr[2] = 'Mini Minnow Blue';
    } else {
        $arr[0] = 'None found!';
    }
    return $arr;
}

$HTTP_RAW_POST_DATA = isset($HTTP_RAW_POST_DATA) ? $HTTP_RAW_POST_DATA : '';
$server->service($HTTP_RAW_POST_DATA);
?>
```

First, note the `require_once('nusoap.php')` line. This line enables the PHP Web page to use classes defined within nusoap.php. You might wonder why all of the other PHP files associated with NuSOAP are necessary. Well, the fact of the matter is that nusoap.php relies on those files in much the same way that the page relies on nusoap.php.
The next line instantiates a `soap_server` object. Unsurprisingly, this enables you to create a Web service that uses the SOAP protocol.

The next line registers the `retrieveByType` function so that it is exposed as a Web service operation. If you look farther down the code, you see that there is a defined function called `retrieveByType`. So why is this step necessary? Because if you don't register the function, it is a simple PHP function that is available only to this PHP page or other PHP pages that include this one. So this line informs the `soap_server` object that you are exposing this function as an operation available to consumers of the Web service.

The next block of code actually implements the `retrieveByType` method. This is a simple PHP function that accepts one parameter: `type`, which can be `trolling`, `casting`, or `other`. As you know, those are the three lure types available to customers of fishinhole.com.

The `retrieveByType` method returns an array. The array consists of a list of lures specific to the type requested. Right now, there are three different lure types available for `trolling` and three different lure types available for `casting`. Note that there is a "catch-all" that includes `other` and any unrecognized types. For those, the Web service simply returns `None Found!` as the only element of the array.

The last two lines execute when the Web service is accessed. The first line checks to see if there is any `POST` data. If not, it sets the `POST` data to an empty string. The second line executes the Web service with the data from the `POST`. The `POST` data contains the SOAP message. You will see more about this when you examine the consumer.

Save this page as `webservice.php` and put it in the same directory where you installed NuSOAP. Obviously, you need to place this page somewhere that can process PHP files. Most hosting solutions support PHP these days, so you shouldn't have trouble finding one if you don't have a PHP processor handy.

Now, quickly test the Web service by accessing the following URL: `http://yourhost/yourdirectory/webservice.php`. Obviously, you need to substitute the actual host and directory where you placed the file for `yourhost` and `yourdirectory`, respectively.

The response you get should be a SOAP response (see Listing 2). If not, your Web service isn't working correctly.
Listing 2. SOAP response

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<SOAP-ENV:Envelope
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmns:xsd="http://www.w3.org/2001/XMLSchema"
xmns:soap: ENC="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode xsi:type="xsd:string">SOAP-ENV:Client</faultcode>
<faultactor xsi:type="xsd:string" />
<faultstring xsi:type="xsd:string">method 'not defined in service'/faultstring>
<detail xsi:type="xsd:string" />
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Writing the page that accesses the Web service

Start with the JavaScript code that actually accesses the Web service, shown in Listing 3.

Listing 3. `invokeService()` JavaScript function

```javascript
function invokeService(type) {
  soapMessage = '<?xml version="1.0" encoding="ISO-8859-1"?>';
  soapMessage+='<SOAP-ENV:Envelope SOAP-ENV:encodingStyle=""';
  soapMessage+=' xmlns:xsd="http://www.w3.org/2001/XMLSchema"';
  soapMessage+=' xmlns:SOAP-ENC="http://schemas.microsoft.org/soap/encoding/"';
  soapMessage+=' <SOAP-ENV:Body> <ns1:retrieveByType xmlns:ns1="http://fishinhole.com">';
  soapMessage+=' <type xsi:type="xsd:string">'+ type + '</type>';
  soapMessage+=' </ns1:retrieveByType> </SOAP-ENV:Body> </SOAP-ENV:Envelope>';

  if(window.XMLHttpRequest) {
    httpRequest=new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    httpRequest=new ActiveXObject("Microsoft.XMLHTTP");
  }

  httpRequest.open("POST",url,true);
  if (httpRequest.overrideMimeType) {
    httpRequest.overrideMimeType("text/xml");
  }
  httpRequest.onreadystatechange=populateDiv;
  httpRequest.setRequestHeader("Man", url + " HTTP/1.1")
  httpRequest.setRequestHeader("MessageType", "CALL");
  httpRequest.setRequestHeader("Content-Type", "text/xml");

  httpRequest.send(soapMessage);
  valTimeout=setTimeout("timeout(httpRequest);",120000);
}
```

The function is called `invokeService` and accepts one parameter: `type`. Needless to say, this corresponds to the `type` parameter accepted by the Web service operation (`retrieveByType`). In other words, the `type` parameter is a string consisting of casting, trolling, or other.

The first several lines of the function assemble the SOAP message. It is well beyond the scope of this document to provide an exhaustive overview of SOAP. However, certain parts of the XML are fairly intuitive. Note that one of the XML elements corresponds directly to the operation
name (retrieveByType). The child element of that element is named according to the name of the parameter specified in the webservice.php file (type). And that element's value is the string parameter that is passed into this JavaScript function, also called type.

The next few lines create a cross-browser-compliant request object. This is the object that is used to access the Web service.

After the request object is established, the function sets the callback function, which in this example is the populateDiv() function. This is the function that displays the returned inventory list on the Web page.

Then the function sets the header. Of specific interest here is that you are establishing a content type compliant with SOAP: text/xml. Also, note the use of the url variable. When you create your own Web page, you need to set that variable to the URL used by your Web service. It will look something like this: http://www.myhost/myservicedir/webservice.php.

Finally, you use the request object to send the SOAP message and establish a timeout in the event that the service does not respond.

Next, take a look at the JavaScript code that responds to the Ajax invocation and displays the inventory, shown in Listing 4.

**Listing 4. populateDiv() JavaScript function**

```javascript
function populateDiv(){
    try {
        if(httpRequest.readyState==4) {
            if(httpRequest.status==200) {
                clearTimeout(valTimeout);
                var text = httpRequest.responseText;
                if (window.DOMParser) {
                    parser=new DOMParser();
                    xmlDoc=parser.parseFromString(text,"text/xml");
                } else {
                    xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
                    xmlDoc.async="false";
                    xmlDoc.loadXML(text);
                }
                var html = "";
                for (i=0;i<xmlDoc.getElementsByTagName("item").length;i++) {
                    html += "<br/>
                    xmlDoc.getElementsByTagName("item")[i].childNodes[0].nodeValue;
                }
                var resultDiv=document.getElementById("resultDiv");
                resultDiv.innerHTML = html;
            }
        }
    } catch(e) {
        alert("Error!"+e.description); 
    }
}
```
The first couple of lines should be familiar to anyone who has used Ajax. Recall that this function is invoked every time the request object experiences a state change. Of particular interest here is when the request returns a valid response (the 200 code).

When the valid response is returned, you clear the timeout and grab the response text. Remember, the response text is actually a SOAP response, so it is in XML format. That means you need to use the JavaScript programming language to parse the XML for salient information.

This brings you to the next few lines. Those lines instantiate a cross-browser-compliant XML document object that can be parsed by the JavaScript programming language. Because the SOAP response is an XML document, it can be parsed just like any other XML document.

Next you create some HTML. You start with an empty HTML string. Then, you parse the SOAP response for \texttt{item} elements. Remember, the Web service returns an array. So it is likely that there is more than one \texttt{item} element. The \texttt{for} loop essentially says “for each \texttt{item} element, do the following.”

The “following” is as follows: The JavaScript code grabs the first child of the \texttt{item} element. In this case, there is only one child, and it is a simple string. Then it extracts the value of that child, which is one item in the inventory. For aesthetic purposes, I prepend a \texttt{<br/>} tag to the inventory item. That way each item in the list is displayed on its own line. Finally, the whole line is concatenated to the existing HTML so you have a complete list when the \texttt{for} loop is completed.

When the HTML is complete, it’s time to put it on the page. In this case, the contents of the \texttt{div} element named \texttt{resultDiv} are overwritten with the HTML you just created. The result is a list of inventory items that appear on the screen when the user selects a new type from the drop-down box on the page.

Speaking of HTML, look at the HTML necessary for the actual Web page, shown in Listing 5.

\textbf{Listing 5. The HTML for the client}

```html
<body>
   <div style="position:relative;left:0px;background-color:blue;margin:0px;">
      <h2 align="center"><font color="#ffffff">FishinHole.com Web Service</font></h2></div>
   <table align="center" cellpadding="6px" cellspacing="6px" width="400" border="0">
      <tr>
         <td width="80" valign="center"><font color="black">
               Lure Type:</font></td>
         <td>
            <select name="lureType" id="lureType" onchange="changeTypes()">
               <option value="">-SELECT-</option>
               <option value="trolling">Trolling</option>
               <option value="casting">Casting</option>
               <option value="other">Other</option>
            </select>
         </td>
      </tr>
      <tr>
         <td colspan="3">
            <div id="resultDiv"></div>
         </td>
      </tr>
   </table>
</body>
```
There's nothing particularly complicated here. You have a simple drop-down box with three lure types: trolling, casting, and other. I use the `onchange` attribute so that when the user selects a new lure type, the JavaScript code that invokes the Ajax request is automatically executed.

Also note that the `resultDiv` div element is present. This is where the inventory list appears.

**Testing**

Put the HTML page that you just created on any platform that interprets HTML and JavaScript code. If you have a Microsoft® Windows® operating system, you can put it on your hard drive.

Next, simply access the Web page. You should see a drop-down in the middle of the screen. Right now, the selected item should be `-SELECT-`. Simply change that to **Trolling**. Wait a few seconds, and you should see an inventory list of three items appear on the screen. Congratulations, your test is a success!

If you are experiencing problems, check to ensure that your `url` variable is set accordingly. Also, JavaScript code may report an exception with a description in a pop-up window. That gives you a clue about what is wrong.

**Conclusion**

Web services are a powerful means of exposing operations to people who have access to the Internet. Ajax is a means of providing Web application users with a rich experience by changing the display without refreshing the page. You can use the two together to create a powerful Web application that emulates a distributed object application and presents a professional interface.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>The client-side HTML file used in this article</td>
<td>client.html_download.zip</td>
<td>2KB</td>
</tr>
<tr>
<td>The Web service PHP file used in this article</td>
<td>webservice.php_download.zip</td>
<td>1KB</td>
</tr>
</tbody>
</table>
Related topics

- Read "Automate data entry with Web services and Ajax" (developerWorks, February 2008) to find out how a Web service and Ajax can improve an application.
- Read Introduction to NuSOAP to learn more about NuSOAP.
- Learn more about Parsing XML in JavaScript.
- What are Web Services provides an introduction to Web services.
- Obtain the NuSOAP - SOAP Toolkit for PHP from SourceForge.
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.