Implement a Web browser-based SOAP Web services client using the Asynchronous JavaScript and XML (Ajax) design pattern.

View more content in this series

This paper is the first of a short series that illustrates the implementation of a cross-platform, JavaScript-based SOAP Web services client based on the Asynchronous JavaScript and XML (Ajax) design pattern for Web applications.

Popularized through its use in a number of well-known Web application services like GMail, Google Maps, Flickr, and Odeo.com, Ajax provides Web developers with a way of expanding the value and function of their Web applications by using asynchronous XML messaging. The Web Services JavaScript Library introduced here expands on the fundamental mechanisms that power the Ajax pattern by introducing support for invoking SOAP-based Web services.

Web services in the browser

Invoking SOAP Web services from within a Web browser can be a tricky exercise, particularly because the most popular Web browsers each handle generating and processing of XML in slightly different ways. There are few standard APIs or capabilities for XML processing that all browsers implement consistently.

One of the mechanisms that browser implementers agree on is the XMLHttpRequest API, which is at the heart of the Ajax design pattern. Thoroughly described in another recently published paper on developerWorks written by Philip McCarthy, XMLHttpRequest is a Javascript object that you can use to perform asynchronous HTTP requests. The paper describes a sequence diagram (see Figure 1) that is very helpful in understanding how the XMLHttpRequest object enables the Ajax design (see Related topics for a link to the full paper).
Figure 1. Philip McCarthy's Ajax Roundtrip sequence diagram

From this diagram you can see exactly how the XMLHttpRequest object functions. Some piece of JavaScript running within the Web browser creates an instance of the XMLHttpRequest and a function that serves as an asynchronous callback. The script then uses the XMLHttpRequest object to perform an HTTP operation against a server. When a response is received, the callback function is invoked. Within the callback function, the returned data can be processed. If the data happens to be XML, the XMLHttpRequest object will automatically parse that data using the browser's built in XML processing mechanisms.

Unfortunately, it's in the details of how the XMLHttpRequest object automatically parses the XML where the primary difficulty with the Ajax approach comes into play. For instance, suppose that the data that I am requesting is a SOAP envelope that contains elements from a number of different XML Namespaces, and I want to grab the value of the $attr attribute on the $yetAnotherElement. (See Listing 1.)
Listing 1. A SOAP Envelope with multiple namespaces

```xml
<s:Envelope
    xmlns:s="http://schemas.xmlsoap.org/soap/envelope/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <s:Header/>
  <s:Body>
    <m:someElement xmlns:m="http://example">
      <n:someOtherElement
        xmlns:n="http://example"
        xmlns:m="urn:example">
        <m:yetAnotherElement
          n:attr="abc"
          xmlns:n="urn:foo"/>
      </n:someOtherElement>
    </m:someElement>
  </s:Body>
</s:Envelope>
```

In the Mozilla and Firefox browsers, extracting the value of the `attr` attribute is a straightforward exercise, as shown in Listing 2.

Listing 2. The method for retrieving the `attr` attribute in Mozilla and Firefox does not work in Internet Explorer

```javascript
var m = el.getElementsByTagNameNS('urn:example', 'yetAnotherElement')[0].getAttributeNS('urn:foo', 'attr');
alert(m); // displays 'abc'
```

A Word about Security

Because of a number of very real security concerns, the XMLHttpRequest object in most Web browsers is restricted by default to interact only with resources and services hosted by the same domain as the Web page the user is viewing. For instance, if I'm currently visiting a page located at http://example.com/myapp/, XMLHttpRequest will only be allowed to access resources that are also located on the example.com domain. This precaution is necessary to keep potentially malicious application code from inappropriately accessing information it otherwise should not have access to. Because the Web services client introduced here is based on XMLHttpRequest, this restriction applies equally to the Web services you will be able to invoke.

If you need to be able to access Web services located on another domain, you can use the following two possible solutions:

- **Digitally sign your JavaScript.** By digitally signing a JavaScript script, you are telling the Web browser that it can be trusted not to perform any malicious activity and that the restriction on what data XMLHttpRequest can access should be lifted.
- **Use a proxy.** A simpler solution is to pass all requests from XMLHttpRequest through a proxy resource located on the same domain as the loaded page. This proxy forwards the requests on to the remote location and returns the results to the browser. From the point of view of the XMLHttpRequest object, the interaction occurs within the existing security configuration.

Unfortunately, this code will not work in Internet Explorer Version 6 because the browser does not implement the `getElementsByTagNameNS` function and, in fact, takes the rather unhelpful
approach of treating XML Namespace prefixes as if they were part of the element and attribute names.

Internet Explorer's lack of great support for XML Namespaces makes it rather difficult to deal with namespace-intensive XML formats like SOAP in a browser-independent manner. To perform something as simple as grabbing the value of an attribute in the result, you have to write special case code such that the expected behavior is consistent across multiple browsers. Luckily, this special case code can be encapsulated and reused.

In order to invoke Web services from within a Web browser and reliably process the SOAP messages, you need to first understand the security issues. (See the sidebar "A Word about Security.") You also need to write a JavaScript script library (Figure 2) that can abstract away the inconsistencies of the underlying browser XML implementations, allowing you to work directly with the Web services data.

![Figure 2. Invoking Web Services from Javascript within the Web Browser using the Web Services JavaScript Library](image)

The Web Services JavaScript Library (ws.js) illustrated in Figure 2 is a collection of JavaScript objects and utility functions that provide a basic level of support for SOAP 1.1-based Web Services. Ws.js defines the following objects:

- **WS.Call**: A Web Service Client that wraps XMLHttpRequest
- **WS.QName**: XML Qualified Name implementation
- **WS.Binder**: Base for custom XML serializers/deserializers
- **WS.Handler**: Base for Request/Response Handlers
- **SOAP.Element**: Base SOAP Element wrapping XML DOM
- **SOAP.Envelope**: SOAP Envelope Object extends SOAP.Element
- **SOAP.Header**: SOAP Header Object extends SOAP.Element
- **SOAP.Body**: SOAP Body Object extends SOAP.Element
- **XML**: Cross-platform utility methods for handling XML
At the core of ws.js is the WS.Call object which provides the methods for invoking a Web service. WS.Call is primarily responsible for the interactions with the XMLHttpRequest object and the processing of SOAP responses.

The WS.Call object exposes the following three methods:

- **add_handler.** Adds a Request/Response handler to the processing chain. Handler objects are invoked before and after a Web service call to allow extensible pre- and post-invocation processing to occur.
- **invoke.** Sends the specified SOAP.Envelope object to the the Web service and invokes a callback when a response is received. Use this method when invoking document-style Web services that use literal XML encoding.
- **invoke_rpc.** Creates a SOAP.Envelope encapsulating an RPC-Style request and sends that to the Web service, invoking a callback when a response is received.

While the WS.Call object is generally not much more than a thin wrapper on top of the XMLHttpRequest object, it does perform a number of actions that will make your life easier. These actions include setting the SOAPAction HTTP header that is required by the SOAP 1.1 specification.

**Using ws.js**

The API presented by the Web services JavaScript Library is rather straightforward.

The SOAP.* objects (SOAP.Element, SOAP.Envelope, SOAP.Header and SOAP.Body) provide the means of building and reading SOAP Envelopes, as shown in Listing 3, so that the underlying details of working with the XML document object model is abstracted away.

**Listing 3. Building a SOAP Envelope**

```javascript
var envelope = new SOAP.Envelope();
var body = envelope.create_body();
var el = body.create_child(new WS.QName('method','urn:foo'));
el.create_child(new WS.QName('param','urn:foo')).set_value('bar');
```

Listing 4 shows the SOAP Envelope that is produced by the code in Listing 3.

**Listing 4. Building a SOAP Envelope**

```xml
<Envelope xmlns="http://schemas.xmlsoap.org">
  <Body>
    <method xmlns="urn:foo">
      <param>bar</param>
    </method>
  </Body>
</Envelope>
```

If the SOAP Envelope that you are creating is representative of an RPC-Style request, the SOAP.Body element provides a set_rpc convenience method (illustrated in Listing 5) that will construct the full body of the request given an operation name, an array of input parameters, and a SOAP encoding style URI.
Listing 5. Building an RPC-Request Envelope

```javascript
var envelope = new SOAP.Envelope();
var body = envelope.create_body();
body.set_rpc(
    new WS.QName('param','urn:foo'),
    new Array(
        {name:'param',value:'bar'},
    ), SOAP.NOENCODING
);
```

Each parameter is passed in as a structure of JavaScript objects with the following expected properties:

- **name**: Either a string or a WS.QName object specifying the name of the parameter. **Required**.
- **value**: The value of the parameter. If the value is not a simple data type (such as string, integer, and so on) then a WS.Binder should be specified that is capable of serializing the value into the appropriate XML structure. **Required**.
- **xsitype**: WS.QName identifying the XML Schema Instance Type of the parameter (for example, if xsi:type="int", then xsitype:new WS.QName('int','http://www.w3.org/2000/10/XMLSchema')). **Optional**.
- **encodingstyle**: A URI identifying the SOAP Encoding Style utilized by this parameter. **Optional**.
- **binder**: A WS.Binder implementation that can serialize the parameter into XML. **Optional**.

For example, to specify a parameter named "abc" with an XML Namespace of "urn:foo", an xsi:type of "int" and a value of "3," I would use the code:

```javascript
new Array({name:new WS.QName('abc','urn:foo'), value:3, xsitype:new WS.QName('int','http://www.w3.org/2000/10/XMLSchema')})
```

Once I have built the SOAP.Envelope for the service request, I would pass that SOAP.Envelope off to the WS.Call objects `invoke` method in order to invoke the method encoded within the envelope:

```javascript
(new WS.Call(service_uri)).invoke(envelope, callback)
```

As an alternative to building the SOAP.Envelope manually, I could pass the operation WS.QName, the parameters array, and the encoding style to the WS.Call object's `invoke_rpc` method, as shown in Listing 6.

Listing 6. Using the WS.Call object to invoke a Web service

```javascript
var call = new WS.Call(serviceURI);
var nsuri = 'urn:foo';
var qn_op = new WS.QName('method',nsuri);
var qn_op_resp = new WS.QName('methodResponse',nsuri);
call.invoke_rpc(
    qn_op,
    new Array(
        {name:'param',value:'bar'}
    ), SOAP.NOENCODING,
    function(call,envelope) {
        // envelope is the response SOAP.Envelope
        // the XML Text of the response is in arguments[2]
    }
);
```
Upon calling either the `invoke` method or the `invoke_rpc` method, the WS.Call object would create an underlying XMLHttpRequest object, pass in the XML elements containing the SOAP Envelope, receive and parse the response, and invoke the callback function provided.

To make it possible to extend the pre- and post-processing of the SOAP messages, the WS.Call object allows you to register a collection of WS.Handler objects, as shown in Listing 7. These are invoked for every request, every response, and every error during the invocation cycle. New handlers can be implemented by extending the WS.Handler JavaScript object.

**Listing 7. Creating and registering response/response handlers**

```javascript
var MyHandler = Class.create();
MyHandler.prototype = (new WS.Handler()).extend({
  on_request : function(envelope) {
    // pre-request processing
  },
  on_response : function(call,envelope) {
    // post-response, pre-callback processing
  },
  on_error : function(call,envelope) {
  }
});

var call = new WS.Call(...);
call.add_handler(new MyHandler());
```

Handlers are most useful for the task of inserting or extracting information from the SOAP Envelopes being passed around. For instance, you could imagine a handler that automatically inserts appropriate Web Services Addressing elements into the header of the SOAP Envelope as in the example shown in Listing 8.

**Listing 8. A sample handler that adds a WS-Addressing Action header to the request**

```javascript
var WSAddressingHandler = Class.create();
WSAddressingHandler.prototype = (new WS.Handler()).extend({
  on_request : function(call,envelope) {
    envelope.create_header().create_child(new WS.QName('Action','http://ws-addressing','wsa')).set_value('http://www.example.com');
  }
});
```

WS.Binder objects (Listing 9) perform custom serialization and deserialization of SOAP.Element objects. WS.Binder implementations must provide the following two methods:

- **to_soap_element.** Serializes a JavaScript object to a SOAP.Element. The first parameter passed in is the value to serialize. The second parameter is the SOAP.Element to which the value must be serialized. The method does not return any value.
- **to_value_object.** Deserializes a SOAP.Element to a JavaScript object. The method must return the deserialized value object.
Listing 9. A sample WS.Binding implementation

```javascript
var MyBinding = Class.create();
MyBinding.prototype = (new WS.Binding()).extend({
  to_soap_element : function(value,element) {
    ...
  },
  to_value_object : function(element) {
    ...
  }
});
```

A simple example

I have provided a sample project to illustrate the basic functionality of the Web Services JavaScript Library. The Web service (shown in Listing 10) used by the demo has been implemented on WebSphere Application Server and provides a simple Hello World function.

Listing 10. A simple Java-based Hello World Web service

```java
package example;

public class HelloWorld {
  public String sayHello(String name) {
    return "Hello " + name;
  }
}
```

After implementing and deploying the service to the WebSphere Application Server, the WSDL description of the service (Listing 11) defines the SOAP message that you need to pass in to invoke the Hello World service.

Listing 11. Snippet from the HelloWorld.wsdl

```xml
<wSDL:portType name="HelloWorld">
  <wSDL:operation name="sayHello">
    <wSDL:input
      message="impl:sayHelloRequest"
      name="sayHelloRequest"/>
    <wSDL:output
      message="impl:sayHelloResponse"
      name="sayHelloResponse"/>
  </wSDL:operation>
</wSDL:portType>
```

Using the Web Services JavaScript Library, you can implement a method that invokes the Hello World Service, as shown in Listing 12.

Listing 12. Using WS.Call to invoke the HelloWorld Service

```html
<html>
<head>
  ...
  <script
    type="text/javascript"
    src="scripts/prototype.js"></script>
  <script
    type="text/javascript"
    src="scripts/ws.js"></script>
</head>
<body>
  ...
</body>
</html>
```
You can then invoke the Hello World service by calling the `sayHello` function from anywhere in our Web application. See Listing 13.

**Listing 13. Calling the sayHello function**

```html
<input name="name" id="name" />
<input value="Invoke the Web Service" type="button" onclick="sayHello($('name').value,$('result'))" />
<div id="container">Result:
  <div id="result">
  </div>
  <div id="soap">
  </div>
</div>
```

A successful call will yield the result illustrated in Figure 3. Running this example in Mozilla, Firefox, and Internet Explorer should all yield the same results.

**Figure 3. The Hello World Example in Firefox**
Next steps

The Web Services JavaScript Library can be used to incorporate basic SOAP Web services into your Web applications in a simple, browser-independent manner. In the next installment of this series, you can explore the use of the library to invoke more advanced Web services based on the WS-Resource Framework family of specifications as well as explore ways in which the Web services capabilities can be expanded and integrated into a Web application.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample project</td>
<td>ws-wsajaxcode.zip</td>
<td>19 KB</td>
</tr>
</tbody>
</table>
Related topics

- **Call SOAP Web services with Ajax** -- Read all parts in this series.
- **Build dynamic Java applications** -- Philip McCarthy's introduction to Ajax for Java developers (developerWorks, September 2005).
- **WebSphere Application Server**: Download a no-charge trial version from developerWorks.

© Copyright IBM Corporation 2005

**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)