Consuming Web Services from AJAX Applications

Robert Dollinger
rdolling@uwsp.edu
Mathematics and Computing Department,
University of Wisconsin Stevens Point
Stevens Point, WI 54481, USA

Abstract

Web applications require new and innovative techniques to manage the increased amount and variety of information they handle. The response to the challenge of Rich Internet Applications' (RIA) requirements is a new web development methodology called AJAX (Asynchronous JavaScript And XML). AJAX is a set of technologies that work together, to leverage browser capabilities in order to reduce the amount and frequency of server postbacks. With AJAX, regular postback cycles are eliminated or replaced by requests, typically asynchronous, for some specific data that result in partial updates of the Web page. This considerably reduces the load on the Web Server and enhances scalability of the server side resources, while making interactive Web applications more responsive. When an AJAX application makes a server request it is most often talking to a piece of code residing on the Web server. This piece of code may connect to a database or another server in order to collect some data which is then sent to the client. When the client call is for a Web service method, developers are faced with a couple of problems to solve, and decisions they have to make: (1) architectural – call the service directly or via the Web server, (2) technical – how to correctly format a request and how to process the returned response, and (3) security – prevent or deal with cross-site scripting restrictions. In this paper we use a specific example in order to illustrate how to consume Web services in an AJAX application and to provide some general guidelines which make the process more structured and easier to understand.

Keywords: Rich Internet Applications, AJAX, Web Services, GET, POST, SOAP envelope, cross-site scripting

1. INTRODUCTION

Web applications become richer and richer both in the information content they provide and by the interactive features that make them look more an more like Windows-based rich client applications. This requires new and innovative techniques to manage the increased amount and variety of information, and support the shift from the paradigm of Web sites to that of Web based applications. Many people claim that we are on the verge of another programming revolution, a revolution that will free the computer users from the constraints of desktop applications and from the dependence on a specific software provider (McClure, 2006). The response technology now provides to the challenge of Rich Internet Applications' (RIA) requirements is a new web development methodology called AJAX (Asynchronous JavaScript And XML). AJAX is a set of technologies that combine, and work together, to leverage browser capabilities in order to reduce the amount and frequency of server postbacks that entirely recreate the page each time. With AJAX, regular postback cycles are eliminated or replaced by requests, typically asynchronous, for some specific data, that result in partial updates of the

Web page. This considerably reduces the load on the Web Server and enhances scalability of the server side resources, while making interactive Web applications more responsive. When an AJAX application makes a server request it is most often talking to a piece of code residing on the Web server. This piece of code can be an event handler on the Web page, an HTTP handler, a Java Servlet, or other, and it may connect to a database or another server in order to collect some data which is then sent to the client. Very often, the client call is for a Web service method which is when developers are faced with several unexpected issues and design decisions: (1) architectural – call the service directly or via the Web server, (2) technical – how to correctly format a request and how to process the response, and (3) security – prevent or deal with cross-site scripting restrictions. In this paper we use a specific example in order to illustrate how to consume Web services in an AJAX application and provide some general guidelines which make the process more structured and easier to understand.

2. BASICS OF WEB SERVICES

A Web Service (or XML Web Service) is nothing but a piece of code that can be invoked via HTTP requests. Web Services expose functionality that is similar to standard code libraries, and do this in the form of objects with methods that can receive parameters and be invoked to perform some work for a client. Web Services are a language and platform independent remoting technology, based on XML for data serialization and communication management. A Web Service can be invoked from any kind of application, and from any platform: the client can be a C/C++ console application under UNIX, a .NET Windows or Web project, a Java program under SUN's Solaris system or anything else (see Figure 1).

The key for this independence is the use of XML, which provides Web Services the advantage of unprecedented flexibility over older, proprietary technologies like: Microsoft's DCOM (**D**istributed **C**ommon **O**bject **M**odel), SUN's EJB (**E**nterprise **J**ava **B**eans) or OMG's CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture). All of these technologies where confined to one single language or platform or required a specific communication protocol (CORBA).

Currently, Web Services are the adopted standard for systems cooperation and distributed processing, a standard that has been jointly developed by IBM, Microsoft and other major vendors.

All previous remoting technologies required that the communicating systems (servers and clients) recognize the same data types in order to insure compatibility.

XML solves the compatibility problem across all systems that have at least an XML parser, so that data can be converted to and from any type via XML in a platform and language independent manner. The communication protocol of choice in Web Services is HTTP, which is stateless, so the server forgets about a given client the moment its request has been served. This means that developing applications with Web Services follows the same stateless programming model like Web Applications. All Web Services require some support infrastructure that consists of the following sub-services:

1) **A discovery service** – to allow users to locate and identify services they want to use. Web Services can be made public in any of the two ways: UDDI (Universal Description Discovery and Integration) servers and, DISCO (Discovery of Web Services). UDDI works like the "Yellow Pages" of the Web Services. Web Service developers would register their new service with one of the UDDI servers. This approach is less and less used in present days since most providers removed their UDDI servers. This reflects a trend of shifting the main stream of the Web Services from the concept of "publicly available" towards the more closed corporate environment. DISCO, is a feature available for Web Services created with .NET and consists of a *. disco file which allows all Web Services on a given site to be advertised to the outside world.

2) **A description service** – by which the client can find out what the service can do, what methods are available and how should these be invoked. All Web Services fully describe themselves through a XML based grammar called Web Service Description Language (WDSL). The description of a Web Service includes an optional description of the service itself, a list of all methods with their exact signatures and return types, as well as an optional description of what each method does. Each Web Service has a WSDL

schema which can be used to automatically generate a Web Based presentation and testing page. WDSL also serves for the generation of the Web Service-proxy for the client side application.

3) **A transport protocol** – HTTP has been chosen as the wire protocol to transmit data to and from Web Services, such that all consumers can access these services without platform or language restrictions. This means that Web Services are invoked via HTTP Requests and the results are retuned through HTTP Response. Web Service HTTP messages can be formatted in three different ways:

- **GET message** – with parameters following the URL string;

- **POST message** – where parameters are encoded in the request object;

- **SOAP message** – where the request is inserted inside a SOAP envelope with a specific well-defined, XML based format.

The encoding of the response is the same for GET and POST messaging, and is slightly different for SOAP. The GET and POST messages have been inherited from the traditional Web applications, and are fast, simple and easy to use. However, GET and POST can be used to transmit only the basic, primitive data types like numbers, strings and simple arrays. The SOAP protocol has been specially created for the needs of Web Services and it is the only protocol that allows transmitting complex data types such as: tables, datasets or custom objects.

Theoretically, a Web Service can communicate via any of the three protocols, but this is a design decision and many Web Services have only the SOAP protocol activated. In order to successfully access the Web Service a client must be able to correctly format requests and process responses in any of the protocols accepted by the service, and SOAP is the safest choice.

3. BASICS OF AJAX

AJAX stands for Asynchronous JavaScript and XML. It is a web development methodology for creating interactive Web applications. AJAX is based on a set of techniques that have been around for quite awhile, and brings them together in a comprehensive methodology that now is much more than just the sum of its constituents, and does this in a way that makes web pages feel more responsive, faster and interactive. One of the main features of AJAX is that it can bring new content into the page without a postback, which provides a much better user experience. AJAX is very likely to represent a turning point in the history of web development. The AJAX enabled Web applications follow a programming and architectural model which combines features of the traditional postback based Web applications with features of the Windows-based desktop applications. Figure 2 compares the traditional Web based application model with the AJAX enabled application model (Zakas, 2006).

There are several ingredients that make up the AJAX methodology:

- (**X)HTML and CSS (Cascading Style Sheets)** - for mark up and styling.
- **JavaScript** – is the tool to provide local responses to user actions.
- **DOM (Document Object Model)** – for access to the content of the web page.
- **XMLHttpRequest object** - is the vehicle used to exchange data (a)synchronously with the web server.
- **XML, JSON (JavaScript Object Notation**) – provide the format for transferring data between the server and the client; basically, any format will work, sometimes preformatted HTML or plain text may be the best choices.

The most important piece in the list above is the XMLHttpRequest object, which is a development API, programmatically accessible from JavaScript that allows web pages to send and receive data to/from the Web server via the HTTP protocol without a page load. The XMLHttpRequest object was first implemented by Microsoft as an ActiveX object in Internet Explorer (IE) 5, in 2001, as part of the XML support library.

In browsers like Mozilla, Netscape and Safari a compatible version is available as a native object (XMLHttpRequest). The XMLHttpRequest object can be used as a vehicle for various forms of data that can be transported to and from the server, either synchronously or asynchronously. The data can be retrieved from any resource located on the web site, a text or XML file, it can be requested and returned via an HTTP handler, a Java Servlet or provided by a Web service. The data that is returned can be formatted in various ways: plain text, XML document,

pre-formatted HTML or JavaScript Object Notation (JSON).

4. CONSUMING WEB SERVICES – A CASE STUDY

In this section we will illustrate the process of consuming a Web service in various contexts by using as a case study a publicly available Web service, **globalweather.asmx** that can be found at the following URL:

www.webservicex.net/globalweather.asmx

This service provides local weather reports and features two methods:

- a method that returns the list of cities in a country given as parameter;

- a second method that gives a brief description of the local weather in the location identified by a country and city arguments given as parameters.

The first step in order to consume a Web Service, once its location has been identified, is to simply invoke the service through the URL shown above. The service will respond with a test and identification page with links to, and brief descriptions of the methods it provides (see Figure 3).

Following any of the links associated to a method, one would get to the corresponding presentation and test page of that method. Each of these pages provides a testing interface for the method with text boxes to fill with parameter values and an invoke button in order to call the method.

In addition, sample descriptions of both the request and response formats are given for each of the three types of protocols one can use to invoke a service: SOAP, GET and POST. For the GetCitiesByCountry method of the globalweather service a partial view of this page is given in Figure 4.

Many development environments, like .NET, make it much too easy for the developers to consume Web services. None of the details of how to call the service are to be handled by the programmers. A Web Service can be accessed from any kind of .NET application: console, windows or Web, and no matter what language is used for development: VB, C# or other. The .NET user is shielded from all the details of communicating with the Web Service due to a server proxy that is automatically created by .NET, based on the WSDL description of the service. To the .NET programmer the Web Service will be presented as a class with a set of methods ready to be used in the application. The class representing the Web Service is accessible via the namespace associated to it.

In order to use a Web Service the following steps are required:

1) Add a reference to the Web Service to the current project – a service can be selected and referenced in a project from a variety of locations: the current project, the current machine, the local network or an external site at a given URL. A local namespace can be associated to the service for easy reference.

2) Create an instance of the class representing the service - this step will actually create an instance of the service proxy inside the user application, making the service appear as an object in the current workspace.

3) Invoke the methods and process the results - the code invoking a Web method should be able to properly process the returned result, which is often formatted as XML. This processing assumes knowledge of what is the content of the returned results, and of how the retuned information is structured. This knowledge can be acquired through the presentation and testing interface of the Web Service, by testing the Web methods to be used.

5. CONSUMING A WEB SERVICE FROM AN AJAX APPLICATION

Web Services and AJAX applications are a natural combination since they have at least two things in common: the stateless programming model and the intensive use of XML. Currently, one can develop AJAX applications by building our own JavaScript code that creates the XMLHttpRequest object and deals with the details of issuing the server requests, or use one of the available libraries and frameworks.

Surprisingly, none of the libraries, except for the Microsoft ASP.NET AJAX framework, provides support for communication with Web services (Gibbs, 2007). This is only one reason why an AJAX developer has to deal with the details of the XMLHttpRequest object.

Architectural Options

Before consuming a Web service in an AJAX application developers have to make a couple of important decisions. The first decision is related to the underlying application architecture. This is based on the understanding of the fact that in any configuration of a Web based application with AJAX and Web Services, there are at least three different systems involved: the client browser, the application server, and the server providing the Web Service. From the point of view of how the Web services are accessed there are several ways in which the three systems may be connected in a service based AJAX-ed web application:

Access the Web Service Directly from the Browser - JavaScript can be used to issue service requests and receive responses from Web Services directly without contacting the original Web server of the application. This is a fast and simple way of integrating Web Services in an AJAX application, but it is considered unsafe from a security point of view, due to the so-called *cross-site scripting problem*. In spite of this, the details of how a Web service is consumed by the browser side code is worth to be studied anyway because one would use the same approach for some of the cases when the service is located on the same server with the application itself, especially when a proxy service is used on the Web server. The configuration of a Web application, where the Web Service is consumed directly from the browser, is given in Figure 5.

Access the Web Service from the Application Server - in this case it is the application server that connects to the Web Service and receives the service responses. The application server has to deal with all the plumbing and details of consuming a Web Service, but very often, the application servers provide a much better support in doing this when compared to JavaScript.

For example, consuming a Web Service from ASP.NET is made very simple from a programmer's point of view. From the point of view of the communication between the browser and the application server we further distinguish the following cases:

- the browser communicates with the application server by sending XMLHttpRequests to HTTP handlers or anything equivalent, like Java Servlets;

- the application server exposes the Web Services to the browser through its own proxy service that mimics the features of the original Web service.

Figure 6 illustrates these two options.

6. CONSUMING A WEB SERVICE FROM JAVASCRIPT

Since more than one of the architectural options above comes down to the problem of how to communicate with a Web service by using JavaScript and the XMLHtppRequest object we are going to look at the details of this task in the current section. This is exactly the kind of difficult task developers are discouraged from solving on their own, due to its alleged complexities, and this is what many environments are trying to protect developers from by incorporating built-in features that automate the entire process. Another factor, adding to the challenge, is that this topic is poorly, if at all, documented in the literature. Since JavaScript is not provided with any specialized support for consuming Web Services, as it is the case with .NET, contacting a Web Service from the browser has to be done "manually." Which means that user provided code needs to be added to deal with the details of communicating with the web Service. However, as we are going to see, this is not as difficult as it may appear at a first look, if we take the right approach.

Essentially, two steps need to be taken in order to consume a Web Service:

1) Invoke the methods provided by the Web Service;

2) Process the returned results.

Each of these steps will be analyzed in detail.

Invoke the Methods Provided by the Web Service

When consuming a Web Service we have to take into account the fact that every Web method has its own signature determined by the method's name and its list of parameters. In addition to this we have a choice of three different protocols to use: GET, POST and SOAP, each of them with a specific format. The key to successfully invoke a Web method is to properly format the request object, for that specific method, as required by each of these protocols. Fortunately, the

sample format descriptions available in the presentation and test page of each method provide all the clues we need to properly format the request for each of the available protocols. The sample format descriptions provide parameter placeholders for every method parameter, along with the corresponding data type specification. Depending on the protocol one or more header options may be also required. According to this, two things need to be done in order to call a method:

- fill in the parameter placeholders with the proper argument values;

- set whatever header options are needed by using the **setRequestHeader()** method of the XMLHttpRequest Object.

Since the request formats are different for each protocol, there will be differences in the corresponding request formatting code. The **GetCitiesByCountry()** method of the **globalweather** service takes only one single string parameter representing a country name. Accordingly, the GET, POST and SOAP request formats are:

1) HTTP GET Request

The GET request format for the **GetCitiesByCountry()** method is given in the appendix (see HTTP Request/Response Formats).

This is the usual format for any GET type request. It specifies the location of the service (**www.webservicex.net**), the name of the service (**globalweather.asmx**) and the invoked method (**GetCitiesByCountry()**) with a parameter called CountryName. In our case, the code to issue an asynchronous XMLHttpRequest call to the **GetCitiesByCountry** method is as follows:

```
xmlHTTPObj.open("GET",
      "HTTP://www.webservicex.net/
       globalweather.asmx/
       GetCitiesByCoutry?
       CountryName="+
       countryName, false);

xmlHTTPObj.send(null);
```

where xmlHTTPObj is a reference to the XMLHttpRequest object, and the variable countryName holds a value interactively provided by the user of the Web page.

2) HTTP POST Request

The POST request format for the **GetCitiesByCountry()** method is given in the appendix (see HTTP Request/Response Formats).

Accordingly, the invoking JavaScript code is:

```
xmlHTTPObj.open("POST",
      "HTTP://www.webservicex.net/
       globalweather.asmx/
       GetCitiesByCountry", false);

xmlHTTPObj.setRequestHeader(
       "Content-Type",
       "application/x-www-form-
       urlencoded");

 xmlHTTPObj.send(
       "CountryName="
       +countryName);
```

In this case the URL string in the **open()** method only specifies the name of the method, while the parameter is part of the request body, and is specified through the **send()** method of the XMLHttpRequest object.

The request header also contains a **Content-Type** entry with value

application/x-www-form-urlencoded

which is set by a call to the **setRequestHeader()** method and specifies the MIME type of the result.

3) HTTP SOAP Request

The GET and POST requests are simple to create and efficient enough, but can be used only for simple response types, basically numbers, strings and arrays. To consume Web Services that return more complex results, like custom data types, one would use SOAP requests (see HTTP Request/Response Formats in the appendices). A SOAP request is in fact a disguised POST request. A SOAP request uses a so-called SOAP envelope with a specific XML syntax that specifies both the name of the invoked method, the name of the parameters and, at the same time, provides the placeholders for these parameters. The SOAP envelope has a well-defined syntax that is validated against an XML schema thus enforcing correctness criteria on submitted request objects.

This format provides the information needed to write the JavaScript that builds the SOAP request, which involves creating the SOAP

envelope with the current arguments built into, and adding the proper header attributes. The SOAP request has two specific entries in its header:

- a **Content-Type** entry with value **text/xml,** and

- a **SOAPAction** entry with the value represented by the following URL string: **HTTP://www.webserviceX.NET/GetCitiesByCountry.**

Both entries are added to the request by calls to the **setRequestHeader()** method of the XMLHttpRequest Object. The rest of the code deals with building the SOAP envelope string, which is then submitted via the **send()** method. The complete code for a SOAP request of the **GetCitiesByCountry()** method is as follows:

```
xmlHTTPObj.open("POST",
      "HTTP://www.webservicex.net/
       globalweather.asmx", false);

  //add headers

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

  xmlHTTPObj.setRequestHeader(
       "SOAPAction",
      "HTTP://www.webserviceX.NET
       /GetCitiesByCountry");

  //build request envelope

  var envelope=
  ' <soap:Envelope '+
  ' xmlns:xsi="HTTP://www.w3.org/
       2001/XMLSchema-instance"'+
   ' xmlns:xsd="HTTP://www.w3.org/
       2001/XMLSchema"'+
   ' xmlns:soap="HTTP://schemas.
    xmlsoap.org/soap/envelope/">\n'+
  ' <soap:Body>\n'+
  '  <GetCitiesByCountry
       xmlns="HTTP://
       www.webserviceX.NET">\n'+
  '     <CountryName>'+
              countryName+
  '     </CountryName>\n'+
  '   </GetCitiesByCountry>\n'+
  ' </soap:Body>\n'+
  '</soap:Envelope>'

   xmlHTTPObj.send(envelope);
```

Processing the Returned Results

The results returned by a Web Service method are serialized and packaged into a properly formatted response. The format of the response object depends both on the specific web method and on the protocol used. Again, the sample format descriptions provide the information needed on the client side in order to use the results. This involves a process consisting of two steps:

(1) extract the result from the response,

(2) use the result according to the needs of the application.

1) Extracting the result from the response

The result of the **GetCitiesByCountry()** method is a string representing a list of cities in the country that was given as parameter. This string is packaged into an XML formatted response with a placeholder for the result itself. The response is available in text or XML format through the responseText or responseXML properties of the XMLHttpRequest Object. The response formats are different for the three types of protocols: GET, POST and SOAP. In the GET and POST responses the result is provided as the value of a <string> XML element:

```
<string>
        list_of_cities
</string>
```

while in the SOAP response the result is located in an XML element with tag name <GetCitiesByCountryResult>:

```
<GetCitiesByCountryResult>
        list_of_cities
</GetCitiesByCountryResult>
```

A closer look at the sample response formats for the GET and POST protocols reveals the fact that they are identical (see HTTP Request/Response Formats).

<u>HTTP GET and POST Response</u>

Accordingly, the code to extract the result would be the same in both cases

Extracting the result node from the responseXML property can be done in either of the two common approaches for processing XML content: with DOM methods or with XPath. Using the DOM **getElementsByTagName()** method returns a list of nodes with one single element in it, the result node, at index position 0, so the expression:

```
xmlHTTPObj.responseXML.
 getElementsByTagName("string")[0]
```

returns the result node, while the result string itself is given by the **data** property of its **firstChild** node.

Alternatively, with the XPath approach, the **selectNodes()** method will be used:

```
xmlHTTPObj.responseXML.
        selectNodes("//string")
```

which directly returns the result node, since there is only one <string> tag anywhere in the response.

In conclusion either one of the following statements would assign the result to the **textResult** variable:

```
var textResult=xmlHTTPObj.
        responseXML.
        getElementsByTagName(
        "string")[0].firstChild.data;
```

or alternatively

```
var textResult=xmlHTTPObj.
        responseXML.selectNodes(
        "//string").firstChild.data;
```

HTTP SOAP Response

The SOAP response has a different structure than the GET and POST responses (see HTTP Request/Response Formats), but the code to extract it is very similar.

The only thing that needs to change is the name of the tag given as parameter to the **getElementsByTagName()** or **selectNodes()** method. Either of the following statements would extract the result from the SOAP response and assign it to the **textResult** variable:

```
var textResult=xmlHTTPObj.
        responseXML.
        getElementsByTagName(
       "GetCitiesByCountryResult")[0].
        firstChild.data;
```

or alternatively

```
var textResult=xmlHTTPObj.
        responseXML.selectNodes(
        "//GetCitiesByCountryResult").
        firstChild.data;
```

Notice that the code to extract the result node is not influenced by where, and how deep this node is placed in the response, nor by the structure of the response itself. What matters is the tag name of the element containing the result.

Using the result according to the needs of the application

Once the result is extracted from its response shell, it is ready to be used. The format of the result itself is not dependent on the HTTP protocol that was used to get it from the Web Service, but it is specific to every method and is not explicitly documented. One simple way of figuring out the structure of the result is to interactively test the Web service method by using the test and presentation interface. In our case an invocation of the **GetCitiesByCountry()** method with country parameter value "Israel" will provide the response shown in figure 7.

This result is in fact a string representation of an XML document. This result could be processed as a string, but in this case it is more convenient to first convert it into an XML DOM tree and then process it by the tools available for XML content processing. The conversion simply consists of loading the string into an XML DOM document. For the particular case of the IE browser the corresponding code is:

```
var xmlResult=new ActiveXObject(
        "Msxml2.DOMDocument");

xmlResult.loadXML(textResult);
```

From the XML DOM document one can extract the list of cities by any of the following:

```
 var cities=xmlResult.
       getElementsByTagName("City");
```

or alternatively

```
   var cities=xmlResult.selectNodes(
        "/NewDataSet/Table/City");
```

The list of cities can than be used to populate a select list on the Web page.

The second method of the globalweather service is GetWeather() with two parameters: country and city. The procedure for using this methiod is similar to was described so far, taking into account the details specific to this method.

7. THE PROBLEM OF CROSS-SITE SCRIPTING

Accessing from the browser a Web Service located on a different site than the application itself is considered unsafe due to damage that can be done to the user's machine by malicious code originating on other systems. This is known as the **cross-site scripting problem**. The **same origin policy** states that any page, from a given origin, may access and interact with (e.g. using JavaScript) any other resource but only from the same origin. An origin is considered a single domain, identified by a unique URL, accessed by a single protocol, e.g. HTTP. Changing the protocol from HTTP to FTP for example means changing the origin. Same is when sending a request to or receiving a response from a location other than that of the page running the code. The same origin policy applies to the XMLHttpRequests as well as to any other communication forms between the browser and servers. This includes pages from other Web Servers, third party Web Services, data servers and other. Basically, whenever applied, the same origin policy makes sure that all communication between the browser and anyone else goes through the application server. All browsers, except IE, explicitly enforce a default same origin policy. IE uses a more comprehensive security mechanism based on trusted pages and security zones. This mechanism is more flexible, than the all or nothing same origin policy, since it allows communication with explicitly listed select sites that are considered as trusted.

In the case of the Web services that originate from a trusted source, one would like to go around the security restrictions imposed by the Web browsers. This approach is considered acceptable and both IE and the Apache Web server can be configured to allow cross-domain scripting. Actually, many Web applications (mash-ups) and much of the Web based advertising rely on some form of cross-domain scripting. Besides configuring the web servers there are quite a few workarounds for this problem. One approach frequently used is to make the Web service calls from inside a frame. (Woolston, 2006) (Moore 2007).

8. CONCLUSIONS

In this paper we looked at the ways and implications of consuming Web services directly from a client browser in an AJAX application. The alternative architectural option is to consume the service from the server side application. One of the approaches with this option is to have an application specific server module (Htpp handler, Java Servlet or other) as the service consumer. Although not the most efficient, this is a convenient, robust and safe solution. The server side code can then (pre)process the results from the Web service and forward them to the client in a form ready to be used, thus minimizing client code. A second approach is to build a proxy service on the application server. This has the advantage of a uniform interface towards the client, and would use the same client code, previously developed for direct access to the Web service.

The concepts in this paper have been presented to the students of the first "Rich Internet Applications with AJAX" class at University of Wisconsin Stevens Point. We emphasized the idea that consuming a Web service only requires to correctly formatting the request object and then extract and process the result from the response object. We also showed that the details for doing all this are provided through the WSDL description of the service. By learning how to identify and use the specific clues in a particular WSDL one should be comfortable to apply the described methodology and use any given Web service.

Based on this kind of insight, the students where able to creatively apply their knowledge on other suggested Web services, and build their own applications. For example the US weather service returns the results in an XML format, instead of the string format used by the global weather service. In spite of this unexpected challenge the students where able to adapt their approach and correctly consume the new service in their own AJAX enabled Web application.

REFERENCES

Gibbs Matt, Wahlin Dan (2007) Professional ASP.NET 2.0 AJAX, Wiley Publishing, Inc.

McClure Wallace B., Cate Scott, Glavich Paul, Shoemaker Craig (2006) Beginning AJAX with ASP.NET, Wiley Publishing, Inc.

Moore Dana, Budd Raymond, Benson Edward (2007) Professional Rich Internet Applications: AJAX and Beyond, Wiley Publishing, Inc.

Woolston Daniel (2006) Pro AJAX and the .NET 2.0 Platform, Apress.

Zakas Nicholas C., McPeak Jeremy, Fawcett Joe (2006) Professional AJAX, Wiley Publishing, Inc.

Appendices

The Basic XML Web Services Model – Multi Platform and Multi Language

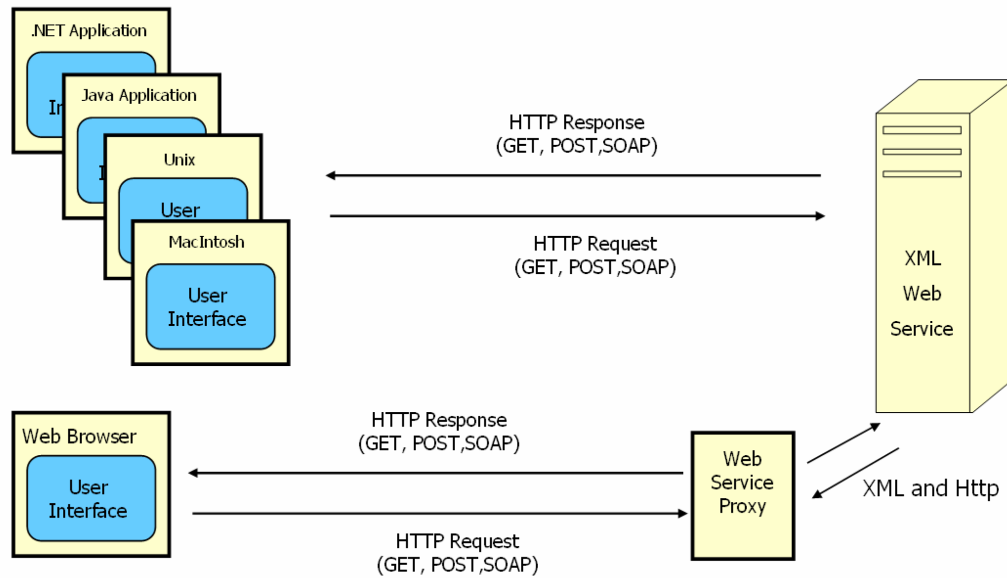
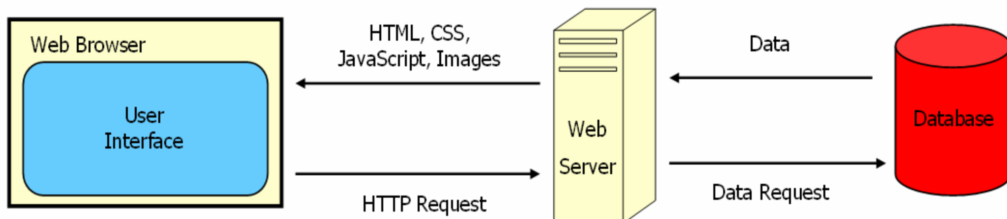


Figure 1. Basic Web Services Model

The Traditional Web Application Model



The AJAX Web Application Model

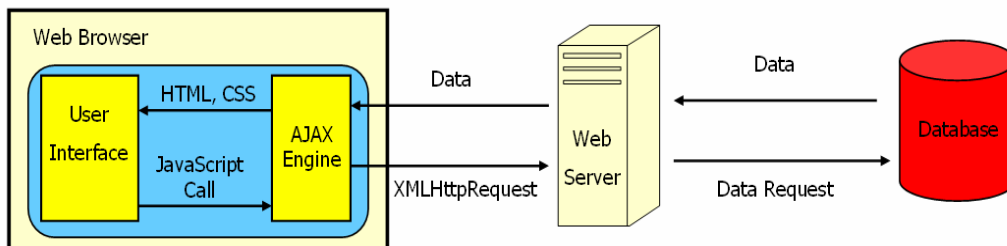


Figure 2. The Traditional versus AJAX Web Application Model

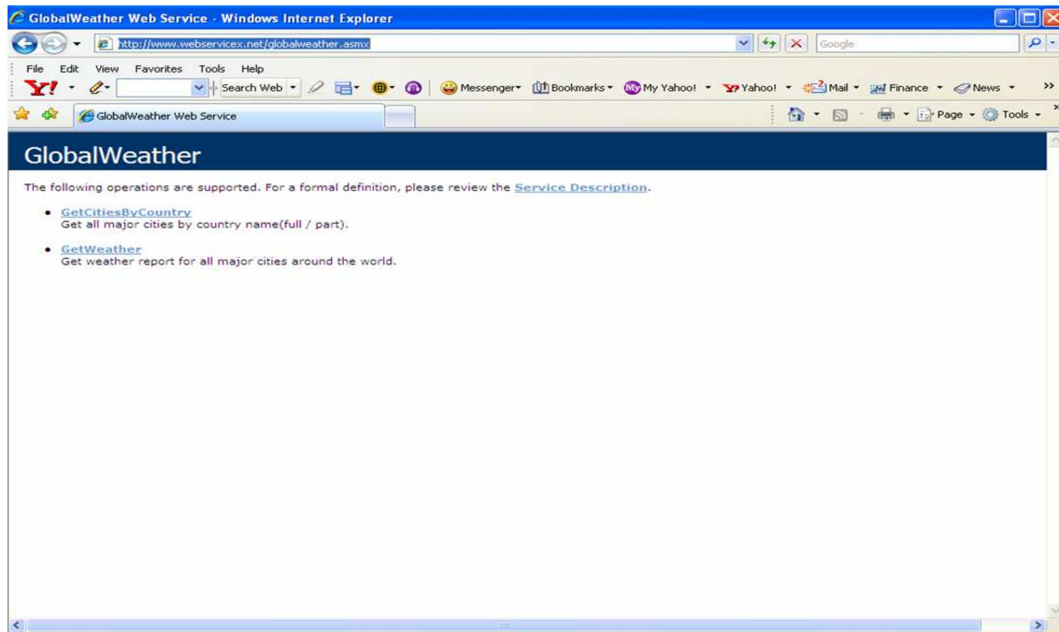


Figure 3. The GlobalWeather Web Service Presentation Page

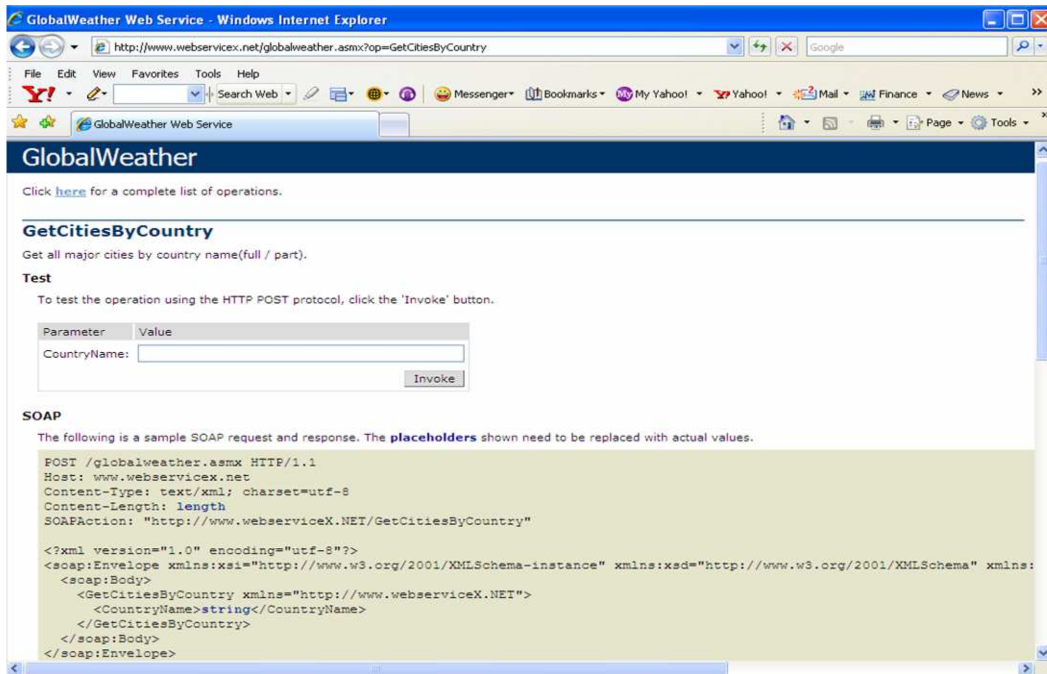


Figure 4. The GetCitiesByCountry Method Presentation Page

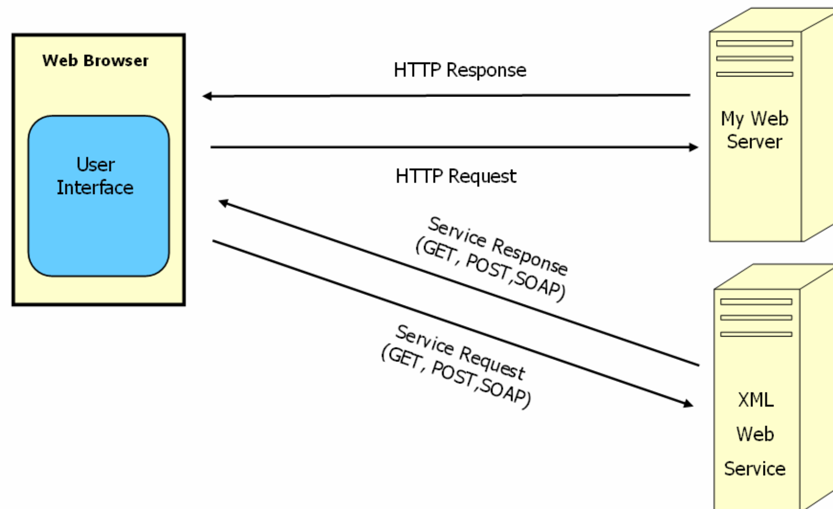
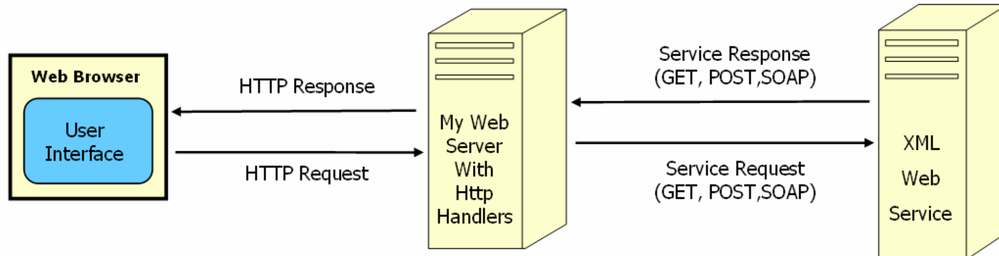


Figure 5. Accessing the Web Service Directly from the Client Browser



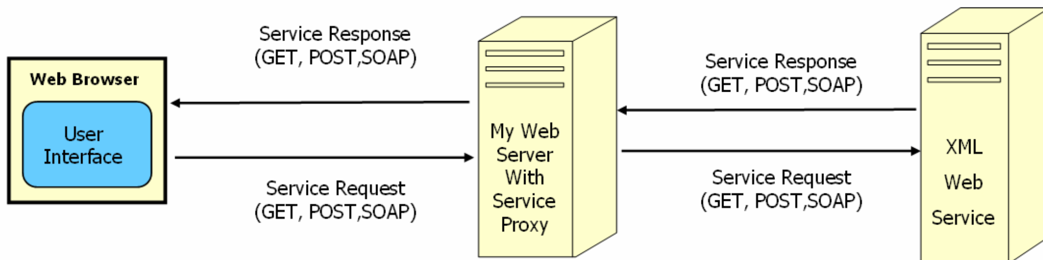


Figure 6. Accessing the Web Service from the Server Side HTTP Handler and by using a proxy Service on the Web Server

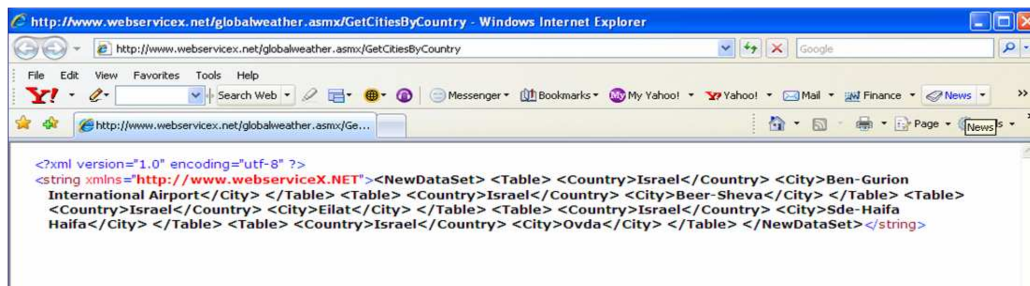


Figure 7. Response and result format for the GetCitiesbyCountry() method with country parameter Israel

HTTP Request/Response Formats

Http GET Request Format

```
GET /globalweather.asmx/GetCitiesByCountry?CountryName=string HTTP/1.1
Host: www.webservicex.net
```

Http POST Request Format

```
POST /globalweather.asmx/GetCitiesByCountry HTTP/1.1
Host: www.webservicex.net
Content-Type: application/x-www-form-urlencoded
Content-Length: length

CountryName=string
```

Http SOAP Request Format

```
POST /globalweather.asmx HTTP/1.1
Host: www.webservicex.net
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.webserviceX.NET/GetCitiesByCountry"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetCitiesByCountry xmlns="http://www.webserviceX.NET">
      <CountryName>string</CountryName>
    </GetCitiesByCountry>
  </soap:Body>
</soap:Envelope>
```

Http GET and POST Response Format

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://www.webserviceX.NET">string</string>
```

Http SOAP Response Format

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetCitiesByCountryResponse xmlns="http://www.webserviceX.NET">
      <GetCitiesByCountryResult>string</GetCitiesByCountryResult>
    </GetCitiesByCountryResponse>
  </soap:Body>
</soap:Envelope>
```