Using the WebService Behavior

This article explains how to use the WebService behavior to call remote methods from Web Services. The **WebService** behavior is implemented with an HTML Component (HTC) file as an attached behavior, so it can be used in Microsoft Internet Explorer 5 and later versions. Most of this article focuses on how to use the **WebService** behavior, but it also touches briefly on Web Services that are used by the **WebService** behavior. The About the WebService Behavior provides additional discussion and information on the main benefits and features of the **WebService** behavior; it also provides numerous links to related Web sites.

The following topics are discussed in this document.

Terminology

This section defines certain terms that are used frequently in this article.

Web Service	Programmable application logic accessible using standard Internet protocols. Web Services provide well-defined interfaces, or contracts, that describe the services provided. The **WebService** behavior uses Web Services.
WebService behavior	The primary subject of this article. A reusable DHTML component that uses Web Services by communicating over HTTP using SOAP.
callService method	One of the methods exposed by the **WebService** behavior.
method call	The invocation of a method.

Attaching the WebService Behavior

The first step in using the **WebService** behavior is to attach it to an element using the STYLE attribute. It is also necessary to set the id attribute so that this element can be easily referenced in script, as shown in the following example.

```
<body>
<div id="service" style="behavior:url(webservice.htc)"></div>
</body>
```

This is all that's needed to attach the behavior to the document. The behavior can also be applied using other variations of Cascading Style Sheets (CSS) style sheet syntax. To begin working with the **WebService** behavior, it is easiest to download the WebService HTC File and copy it to the same directory as the Web page that uses the behavior. By referencing a local file, any Dynamic HTML (DHTML) behavior-

related cross-domain security issues are avoided, and the WebService HTC File download is invisible to the client.

The next topic explains how to initialize the behavior and identify any Web Services that will be used by the Web page.

Identifying Web Services

Any given Web Service exists at a specific URL, so the **WebService** behavior provides the useService method to map the Web Service URL to a FriendlyName, which is passed as a parameter to the method. After this method has been called, the friendly name for the Web Service can be used directly in script to reference the Web Service URL, which helps to keep the script code readable and as simple as possible.

The basic syntax of the **useService** method is as follows:

```
id.useService("ServiceURL","FriendlyName");
```

The **id** that the **useService** method is applied on is the value of the **id** attribute specified on the HTML tag to which the behavior is attached. Based on the preceding behavior attachment sample, the following code example shows how the **useService** method can be coded.

```
service.useService("/services/math.asmx?WSDL","MyMath");
```

Note

The specification of the WSDL query string at the end of the URL means that a Web Services Definition Language file describes the Web Service. The WSDL file consists of XML file with a `.xsd` file extension. This WSDL file describes the Web Service class, which is referenced in script by the friendly name MyMath. The **WebService** behavior supports WSDL version 1.1 , see Web Services Description Language (WSDL) 1.1 for more information.

Once the **useService** method has been called, the Web Service can be addressed by its friendly name MyMath. It is recommended that the friendly name used in the client script matches the class name that implements the methods being called, but this is not a requirement. To ensure that the **useService** method works correctly, it should be placed inside a handler for the onload event, so that the first attempt to call a method in the behavior only occurs after the page has been downloaded and parsed. The code placed inside this handler may define friendly names for one or more Web Services. The following sample shows how this can be accomplished.

```
<script language="JavaScript">
function init()
{
  service.useService("/services/math.asmx?WSDL","MyMath");
  service.useService("/loaninfo.asmx?WSDL","loanInfo");
  service.useService("/financebroker.asmx?WSDL","finance");
}
</script>
<body onload="init()">
<div id="service" style="behavior:url(webservice.htc)">
</div>
</body>
```

Note The current implementation of the **WebService** behavior requires that the Web Services being referenced directly by the **useService** method reside on the same domain as the Web server that delivers the Web page.

As shown in the preceding sample, each call to **useService** is applied with the same attached behavior, and each line of code uses the **id** of the tag to which the behavior is attached. In this example, a div element is

used—but any HTML element can be used. The ServiceURL points to the URL of a Web Service, which implements one or more methods. When the **useService** method is called, the behavior associates the ServiceURL with the friendly name MyMath.

When the **useService** method is called, the behavior downloads the SDL description for the Web Service, so that it has all the information it requires to construct the appropriate method calls. Because the friendly names are defined before making calls to **useService**, performance is somewhat improved when the method call is actually made, because by that time the behavior has already obtained the Web Service description.

It is quite possible that two different Web Services could each have the same name, even though the URL is unique. Indeed, it is certain that this will commonly occur once a large enough number of Web Services exist. It's also quite possible for such services to have similar classes and methods. The **useService** method avoids namespace conflicts by specifying a unique friendly name for each service, as shown in the following:

```
service.useService("/services/math.asmx?WSDL","MyMathService");
service.useService("/techlibs/math.asmx?WSDL","MyMathTech");
```

In the previous sample, two calls to **useService** are made to a service named MyMath, so unique friendly names for each service are chosen. If the same friendly name for the Web Service had been specified in each call to **useService**, the last executed call would have overridden the settings from the previous one.

In the example so far, a hypothetical Service is being accessed to call methods from a simple math class, which in this case is implemented with C#. The class implements two methods, which add or subtract two integers. The contents of the math.asmx file are shown as follows:

```
<%@ WebService Language="C#" class=MyMath %>

using System;
using System.Web.Services;

public class MyMath {

   [WebMethod]
   public int add(int a, int b)
   {
       return a + b;
   }

   [WebMethod]
   public int subtract(int a, int b)
   {
       return a - b;
   }

}
```

Note Two of the simplest possible methods are implemented in this example. Each method in the class MyMath is identified as a WebMethod.

Calling Methods on Web Services

Once the behavior has been attached to the document and the friendly name for at least one WebService has been defined, the **callService** method can be used to invoke methods.

Due to the stateless nature of HTTP, the **WebService** behavior communicates with Web Services asynchronously. In many cases, it is necessary to identify each call to the Web Service uniquely, as generally each method call can return a different result. This is why the **callService** method returns a unique value when it is invoked, so that the returned result can be identified with a specific instance of a method call. Here's the basic syntax for **callService**.

The **WebService** behavior enables both synchronous and asynchronous remote procedure calls to Web

Services. By default, the asynchronous mode of remote method invocation is used. The async property of the call object is used to control the mode of invocation. In most cases, this is the best way to use the **WebService** behavior, and therefore, most of the emphasis of this article is on asynchronous method invocation. The synchronous approach of remote method invocation is discussed in Synchronous Method Invocation.

```
iCallID = id.FriendlyName.callService([CallbackHandler,] "MethodName",  Param1, Param2, ...);
```

The **callService** method has an optional first parameter that specifies a callback handler function to process the results received from the method call. If no callback handler is specified, then the event handler for the onresult event is used. The **onresult** event is fired by the **WebService** behavior once the result of the remote call has completed. Both approaches are explained in Handling Results from WebService Calls.

Continuing with the sample, the following approach can be used to return a unique value from a call to a Web Service method named add.

```
<script language="JavaScript">
var iCallID;
iCallID = service.MyMath.callService("add",5,6);
</script>
```

Note Because the **callService** method does not return the result from the add function immediately, call returns a unique identifier that is stored in the variable iCallID in the preceding example. The value of this identifier enables the positive identification of a particular result.

The previous sample shows how **callService** can be used to invoke the add method, which simply adds up two numbers. Notice that the **callService** to the add method does not specify a type or a parameter name for the returned result. The class definition is done in the Web Service, using the Service Description Language (SDL) format to specify the XML grammar that represents the capabilities exposed by the Service. SDL uses XML grammar to describe the methods being exposed by the Web Service, so that third parties can obtain information and call methods from Web Services in a consistent manner.

In summary, three types of parameters can be specified by **callService**. The first is optional; it is the name of a callback handler function. The second type is the Method Name, passed in as a String. The remaining type is a number of arguments to be passed directly into the method on the Web Service, and the number of parameters depend entirely on the method being called. A more complex data structure can be passed to a method by specifying an object as one of the parameters. More usage samples can be found in the code samples that follow.

Handling Results from WebService Calls

The **callService** method initiates an asynchronous communication between the **WebService** behavior and the Web Service. Eventually the Web Service responds, or a time-out error occurs in the WebService behavior. In either case, the result of the call should be processed by an event handler or callback function to determine whether an error occurred and to evaluate and interpret the results. Each technique is presented and explained in the following section.

Obtaining Results with the onresult Event

When the WebService behavior has received the XML data packets containing the results from a previous call, it fires an **onresult** event. It will only fire this event if a callback function was not specified in the **callService** method. The **onresult** event has an event object, which contains a number of important properties. These properties fall into two categories, the first containing the results, assuming a method was called successfuly, and the second set providing information on errors. Each property is described in The result Object.

The following code shows how the **onresult** event handler can be used to interpret the results from Web Service calls. In the sample code, the iCallID variable has global scope and contains a unique value that is returned from the method **callService**. The iCallID variable stores the unique value of the call, which is later tested in the onWSresult function in two places to see if its value matches with the event.result.id

property. The other conditional expressions test for an error, as indicated by comments in the code.

```
<SCRIPT language="JavaScript">
// All these variables must be global,
// because they are used in both init() and onresult().
var iCallID = 0;
var intA = 5;
var intB = 6;

function init()
{
    // Establish the friendly name "MyMath" for the WebServiceURL
    service.useService("/services/math.asmx?WSDL","MyMath");
    // The following method doesn't specify a callback handler, so onWSresult() is used
    iCallID = service.MyMath.callService("add", intA, intB);
}

function onWSresult()
{
    // if there is an error, and the call came from the call() in init()
    if((event.result.error)&&(iCallID==event.result.id))
    {
        // Pull the error information from the event.result.errorDetail properties
        var xfaultcode   = event.result.errorDetail.code;
        var xfaultstring = event.result.errorDetail.string;
        var xfaultsoap   = event.result.errorDetail.raw;

        // Add code to handle specific error codes here
    }
    // if there was no error, and the call came from the call() in init()
    else if((!event.result.error) && (iCallID == event.result.id))
    {
        // Show the arithmetic!
        alert(intA + ' + ' + intB + ' = ' + event.result.value);
    }
    else
    {
        alert("Something else fired the event!");
    }
}
</SCRIPT>
<body onload="init()">
<div id="service" style="behavior:url(webservice.htc)" onresult="onWSresult()">
</div>
</body>
```

Note If `event.result.error` is false, then the `errorDetail` properties are undefined.

Obtaining Results with a Callback Function

The following code shows how a method call can be made using the callback function to handle the results.

```
<SCRIPT language="JavaScript">
// All these variables must be global,
// because they are used in both init() and onResult().
var iCallID = 0;
var intA = 5;
var intB = 6;

function init()
{
    // Establish the friendly name "MyMath" for the WebServiceURL
    service.useService("/services/math.asmx?WSDL","MyMath");
    // The following uses a callback handler named "mathResults"
    iCallID = service.MyMath.callService(mathResults, "add", intA, intB);
}

function mathResults(result)
{
    // if there is an error, and the call came from the call() in init()
    if(result.error)
    {
        // Pull the error information from the event.result.errorDetail properties
        var xfaultcode   = result.errorDetail.code;
        var xfaultstring = result.errorDetail.string;
        var xfaultsoap   = result.errorDetail.raw;

        // Add code to handle specific error codes here
    }
    // if there was no error
    else
    {
        // Show the arithmetic
        alert(intA + ' + ' + intB + " = " + result.value);
    }
}
</SCRIPT>
<body onload="init()">
<div id="service" style="behavior:url(webservice.htc)">
</div>
</body>
```

Note If `result.error` is false, then the `errorDetail` properties are undefined.

As you can see, the two versions of code are quite similar. The main difference is that the callback approach doesn't use the **event** object to obtain the results. Instead, the callback function receives a parameter containing the results. In the example the object is named result, and this object exposes the same set of properties as the event.result object used in Obtaining Results with the onresult Event.

In summary, three different types of parameters can be specified in the **callService** method. The first is an optional parameter specifying the name of the callback function, second is the name of the method being called, and the remaining parameters are passed directly as arguments of the method being called in the Web Service.

Choosing an Approach

There are certain issues to consider before deciding whether it is preferable to use the **onresult** event or a callback function to handle the returned value from the method call. Callbacks and events are quite different approaches, and you should select the approach that best fits the needs of the application. The main difference between events and callbacks is that an event is essentially an anonymous broadcast and can be picked up by any function that "listens" for it.

There are several scenarios where multiple behaviors might be attached to a page, each performing a particular function, possibly with events firing from one of more of these components. In such a situation, callback functions with specific names would be recommended because this would help to avoid any possible confusion that might arise if two components are designed with the same named event. This is a general

comment about using behaviors and is not specific in any way to using the **WebService** behavior.

Callbacks can be more work to code initially but may be preferable in many scenarios. For example, several callback handlers can be created to handle distinctly different calls being made to Web Services that return completely different types of result. In such cases, it may be too cumbersome to combine the code logic in a single event handler. In other situations, using a single event handler to manage all the permutations might be preferable.

The result Object

Whether an event handler or a callback function is used to interpret results, an object containing results data is generated whenever **callService** is invoked. As shown in the preceding examples, the result object is referenced as event.result or as the name of the parameter being passed to the user-defined callback function. Using either approach the result object exposes several properties.

When scripting with the behavior, using either method of result handling, the error property should be checked to determine if the remote method invocation was processed succesfully. It is always good programming practice to verify a function call.

If an error or problem is encountered for a method call, the **error** property is true and the errorDetail is generated at runtime. The **errorDetail** object exposes several properties, which can be evaluated to diagnose the error.

The Call Object

The **call** object has several properties that can be used to define parameters that may be required when using certain Web services. These properties are discussed in this section.

Synchronous Method Invocation

The **callService** method is processed asynchronously by default. Therefore, in order to process methods synchronously, it is necessary to set the **Boolean async** property to false.

Authentication

The userName and password properties can be used when calling a Web Service that requires basic authentication.

Port Naming

The portName property can be used when it is necessary to define the port name of a Web Service.

The following sample code shows how a synchronous method call can be made using the **call** object. In the code, all the optional parameters are specified.

```
<SCRIPT language="JavaScript">
var iCallID;

function init()
{
    var callObj = new Object();
    callObj.funcName = "add";      // Name of the remote function.
    callObj.portName = "Port1";    // Name of the port.
    callObj.async = false;         // A Boolean that specifies the type of call
    callObj.timeout = 120;         // Timeout value for the method call (seconds)
    callObj.userName = "guest"     // Basic authorization username
    callObj.password = "something" // Basic authorization password

    // SOAP header information
    callObj.SOAPHeader = "<SOAP-ENV:Header>";
    callObj.SOAPHeader += "<t:Transaction xmlns:t='some-URI' SOAP-ENV:mustUnderstand='1'>";
    callObj.SOAPHeader += 5;
    callObj.SOAPHeader += "</t:Transaction>";
    callObj.SOAPHeader += "</SOAP-ENV:Header>";

    // Establish the friendly name "MyMath" for the WebServiceURL
    service.useService("/services/math.asmx?WSDL","MyMath");
    // The following uses a callback handler named "mathResults"
    iCallID = service.MyMath.callService(mathResults, callObj, intA, intB);
}

function mathResults(result)
{
    // if there is an error, and the call came from the call() in init()
    if(result.error)
    {
        // Pull the error information from the event.result.errorDetail properties
        var xfaultcode   = result.errorDetail.code;
        var xfaultstring = result.errorDetail.string;
        var xfaultsoap   = result.errorDetail.raw;

    }
    // if there was no error
    else
    {
        // Show the arithmetic
        alert(intA + ' + ' + intB + " = " + result.value);
    }
}
</script>
<body onload="init()">
<div id="service" style="behavior:url(webservice.htc)">
</div>
</body>
```

SSL Authentication

In certain Web applications, especially those managing personal or financial information, Secure Sockets Layer (SSL) authentication is used to verify the identity of the caller. In a scenario where a portfolio is being managed over the Web, for example, an update to a user's account information can require several calls to the Web service.

The use of Web services over HTTP is stateless, and, therefore, each remote method call to a Web service using SSL causes a request for authentication. In scenarios where several calls are made, repeated requests for authentication result in a frustrating and impractical user experience.

The **WebService** behavior does not persist SSL authentication data by default. However, the behavior does provide a useOptions object, which can be used to avoid the repetition of authentication challenges. When a **useOptions** object is created with the appropriate settings, the SSL authentication information for each

method call is persisted.

The **useOptions** object is created by calling the createUseOptions method. The **useOptions** object exposes the **Boolean** property, reuseConnection, which must be set to true in order for SSL authentication data to be persisted for each call to a Web service. Once the **useOptions** object has been created with the appropriate settings, the **useOptions** object is passed as a parameter to the **useService** method. For more information and code samples illustrating this technique, see the createUseOptions method.

Calling Methods on Remote Servers

Using the **WebService** behavior, it is not possible to call a remote method directly on a server that resides in a different domain from the server hosting the Web page. However, Web servers can communicate directly with other Web servers, even if they reside in different domains. Therefore, it is possible to use the Web server that is hosting the Web page as a proxy to other Web servers, as a means for the **WebService** behavior to access resources on remote domains. The following diagram illustrates how **WebService** calls can be passed to remote servers.

The **WebService** behavior uses a trusted Microsoft ActiveX object to communicate with Web Services. Provided that the client allows trusted ActiveX controls, which are allowed by default in Windows Internet Explorer, the **WebService** behavior can communicate with Web Services. However, if specific browser settings prevent trusted controls from running, the **WebService** behavior will not function.

To configure a page that uses the **WebService** behavior to call methods from remote Web Services, the Web server delivering pages to the client must act as a proxy for Web Services residing elsewhere. The following code shows how a remote method on another server can be called.

```
[WebMethod]
public int Add(int a, int b)
{
    // establishes an object to call a remote
    // method implemented by remote Server B
    MyMath m=new MyMath();
    return m.Add(a,b);
}
```

Related Topics

- WebService
- WebService HTC File
- HTC Reference
- Web Services Description Language (WSDL) 1.1
- About the WebService Behavior
- WebService Behavior: Supported Data Types
- result
- errorDetail
- Introduction to DHTML Behaviors

- Using HTML Components to Implement DHTML Behaviors in Script
- Using DHTML Behaviors

Community Content

Creating Custom SOAP Header in JS

Iam trying to add the following SOAP header using javascript as specified in the code above using callObj.SOAPHeader:
<soap:Header>
<MyHeader xmlns="http://tempuri.org/">
<Created>2009-10-12T10:20:32.219327Z</Created>
<Expires>60000</Expires>
</MyHeader>
</soap:Header>

but <soap:Header> elements are not set in the request . The same thing works with C# but not in JS. Can u pls suggest how to acheive this?

10/12/2009
HarshaSri

Got Permission Denied when try to use WebService Behavior

My application is written in ASP.NET 1.1, recently I need to add a new page to call send email web service which is resides on the same server. Below is my source code:

```
<script type="text/javascript">
var iCallID = 0;
var service = null;
function Init()
{
service = document.getElementById("service123");
service.useService("http://localhost/SendEmail.asmx?WSDL","MyMath");
}

function yyyy()
{

try
{
iCallID = service.MyMath.callService(CallbackSave, "SendMessage", "fromAddress","toAddress","Test","For
me","MyType",1);
}
catch (e)
{

alert(e.description );
}
return true;
```

```
}
function CallbackSave(res)
{

if (res.value == 1)
{
document.write("Sent successfully");
}
}
</script>
</HEAD>
<body MS_POSITIONING="GridLayout" onload="Init()">
<div id="service123" style="BEHAVIOR: url(webservice.htc)"></div>
<form id="frmDefault" method="post" runat="server">
<input id="xxxx" onclick="yyyy();" type="button" value="mmmm!" >
<div>
</div>
</form>
</body>
</HTML>
```

When I click the button "mmmmm!" to call the web service, I always got the error message "Permission Denied", I am not sure why I got the error. I will appreciate if anyone can help me on this. Thanks!

12/1/2008
Tom Cat1