Working with Web Services in PowerBuilder

An introduction

WRITTEN BY
DOUG PORTER

Unless you have been hiding in a cave for the past couple of years, you are probably well aware of the buzz surrounding service-oriented architecture (SOA). This concept of interaction among loosely coupled collections of components is often implemented through a series of services accessible over HTTP that processes requests and responses (Web services). As a PowerBuilder developer you may be wondering how you can join the SOA party while still leveraging your existing skills. This article will walk you through three techniques available to the PowerBuilder developer (even if you may not be on a currently supported version).

Accessing a Web service is very similar to visiting a Web page containing parameters in the URL or submitting a form on a Web page. An HTTP request gets sent to a URL via an HTTP GET or POST along with a set of arguments in the form of name/value pairs. In PowerBuilder there are three options available for performing this type of operation: the GetURL/PostURL functions, using Microsoft's XmlHttp object via OLE, and using the Web Service Proxy object.

AUTHOR BIO

Doug Porter is a software developer with DailyAccess Corporation. He is a Sun Certified Java Programmer and Certified PowerBuilder Professional and was a speaker at TechWave 2004. He holds a BA in Spanish and an MS in CIS from the University of South Alabama where he has also worked as an adjunct instructor. Doug works extensively with EAServer, PowerBuilder, and Java developing client/server and Web applications for the financial industry.

```
- <Item>
    <ASIN>B000069KB3</ASIN>
  - <ItemAttributes>
      <Artist>Butch Walker</Artist>
      <ProductGroup>Music</ProductGroup>
      <Title>Left of Self-Centered</Title>
    </ItemAttributes>
  </Item>
- <Item>
    <ASIN>B0002M6T10</ASIN>
  - <ItemAttributes>
      <Artist>Butch Walker</Artist>
      <ProductGroup>Music</ProductGroup>
      <Title>Letters</Title>
    </ItemAttributes>
  </Item>
- <Item>
    <ASIN>B00065VPNK</ASIN>
```

FIGURE 1 |

Demonstration Application to Call Amazon.com Web Services

In this article we will create a demonstration application that interacts with the Amazon.com E-Commerce Web Service (http://aws.amazon.com). Our application will use this service to search through one of their product indexes for items matching our search criteria. To use the Amazon.com Web service you will need to sign up for an access key through their site. The code samples will show where your key should be placed with the indicator "[YourAccessKey]".

By looking at the documentation for the Amazon E-commerce Web service, we can see that to perform an item search operation a minimal set of arguments is required. There are many additional arguments that may be used when performing searches, but we will be using the subset listed below. (Browse the Amazon.com Web service documentation to see the full range of search arguments and operations available.)

- ***Web Service URL:*** http://webservices.amazon.com/onca/xml
- ***Service:*** Name of the service being used; should be AWSECommerceService
- ***AWSAccessKeyID:*** [YourAccessKey]
- ***Operation:*** Operation to be performed (ItemSearch in our case)
- ***Keywords:*** Word or phrase to search for
- ***SearchIndex:*** One of Amazon.com's search indexes (Music, Books, etc.)

Responses from this Amazon Web service will be returned as structured XML that can then be parsed and displayed. A demonstration of the various methods available for parsing XML in PowerBuilder will be covered in a later article.

Before we can send requests, we must decide what to search for. I'm a huge music fan, so for this example we will use the "Music" SearchIndex and for our Keywords argument we will search for information on one of my favorite musicians, the great "Butch Walker" (an Atlanta musician formerly with the Marvelous 3).

GetURL/PostURL

The oldest methods for pulling data over HTTP in PowerBuilder (since version 6.5) are the GetURL and PostURL functions. These functions allow you to send an HTTP GET or POST request

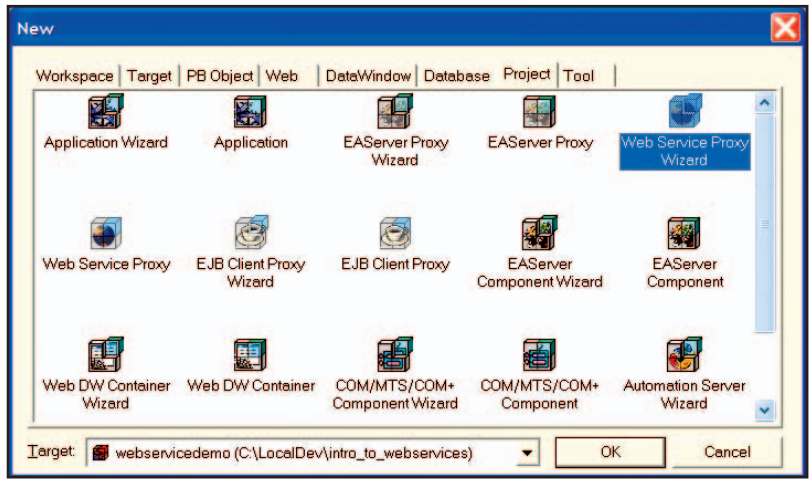

FIGURE 2 |

from within PowerScript and capture the resulting response.

When passing arguments to a URL via an HTTP GET, the URL is generated in the format:

```
BaseURL?<name1>=<value1>&<name2>=
  <value2>&...
```

The full URL for an HTTP GET to our Amazon Web services with our selected parameters would be as follows:

http://webservices.amazon.com/onca/xml?Service=AWSECommerceService&AWSAccessKeyID=[YourAccessKey]&Operation=ItemSearch&SearchIndex=Music&Keywords=Butch Walker

The resulting response would be XML containing a list of items matching our search (see Figure 1).

Now that we know what our request should look like, we can begin to write code to call the Web service. Both the GetURL and PostURL functions require that you first create a standard class of the type InternetResult to capture and process the Web service response via the InternetData(blob) function. Once we have created this class, we will override the InternetData(blob) function with code to process the response. For demonstration purposes, a simple MessageBox will do for now, but we could write the data out to a file or parse it for display in a DataWindow. The following code snippet should be added to the function InternetData(blob) in the InternetResult object:

```
MessageBox("Response", &
            String(data))
Return 1
```

The code in Listing 1 demonstrates a call to the service with GetURL (see Listing 1). Note that to call the GetURL/PostURL functions, you must use GetContextService to get a service reference to the Internet service.

Executing the snippet should generate a MessageBox filled with our XML response from the Amazon Web service. Performing the same operation with a PostURL is almost identical; we just have to set a couple of extra parameters (content headers and port) (see Listing 2).

While this approach works well for standard HTTP requests, the GetURL/PostURL functions do not allow communication over HTTPS. To achieve secure communication with the Web service, we must move on to one of our other two methods.

	GetURL/PostURL	MS XmlHttp	Web Service Proxy
Minimum PB Version	6.5	6.5	9
Supports HTTPS	No	Yes	Yes
REST Requests	Yes	Yes	No
SOAP Requests	Yes	Yes	Yes
XML Parsing	Standard String Parsing	MS XmlDom object	PBDom or DataWindow
Manually Create Request	Yes	Yes	No

TABLE 1 |

Microsoft XmlHttp Object Using OLE

Using PowerBuilder's OLE capabilities, we can take advantage of Microsoft's XmlHttp object to send requests to a Web service. The code to use this object is very similar to what we have seen for GetURL/PostURL. The XmlHttp object can send requests via HTTP GET and POST but adds the benefit of being able to also send requests over HTTPS. Before you may use this object, you must download and install the MSXML package from Microsoft's Web site (this example uses version 4.0 of the package). One nice added bonus included with this package is an XML parser (handy if you don't happen to be on a version of PowerBuilder that can parse XML) (see Listing 3).

Web Service Proxy Object

Last but not least is the Web Service Proxy Object. This object is available to those who are using PowerBuilder version 9 and above. This object examines a Web service's WSDL (Web Service Description Language) file to determine all of the services available that may be requested. We will step through the Web Service Proxy Wizard to create a proxy object for our Amazon Web Service. We can then deploy this object, which will generate a set of objects, methods, and structures we can use to call our Web service and capture the response (because of recent changes in the Amazon WSDL file, PowerBuilder 9.0.3 Build 8565 or above is required to generate a proxy using the wizard). The Web Service Proxy uses SOAP-style requests to interact with our Web service. The other two methods shown used REST-style requests (for more information see the sidebar **SOAP vs REST**).

To create a Web Service Proxy Object, proceed through the following steps:

1. Click New object and select the Web Service Proxy Wizard from the Project tab (see Figure 2).
2. Click Next and select a WSDL File. Enter the URL to the Amazon WSDL file: http://webservices.amazon.com/AWSECommerceService/2005-10-05/AWSECommerceService.wsdl.
3. Click Next and select the available services exposed through the WSDL file.
4. Click Next and select an available port for the Web service.
5. Click Next and, if you like, enter a prefix for the proxy name.

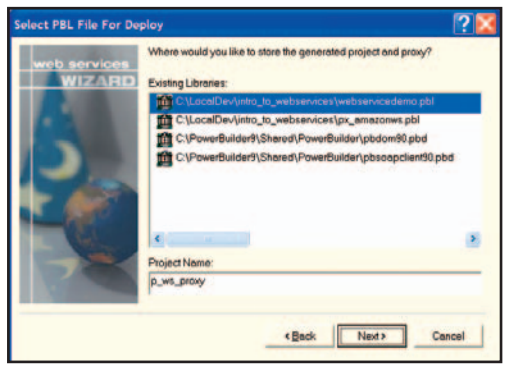

FIGURE 3 |

SOAP vs REST

SOAP: A well-defined protocol for exchanging XML-based messages over a network usually via HTTP. Oftentimes it's used in an RPC (Remote Procedure Call) messaging pattern.

REST: In the context of Web services, describes loosely structured requests over HTTP with no set standard defining the structure of requests or responses from the service.

Differences

Accessing a REST-style Web service involves passing multiple arguments in the form of name/value pairs via a GET or POST request. These arguments provide all of the incoming parameters the Web service may need to create a response (which may or may not be XML). In contrast, a SOAP-style Web service accepts a single incoming argument that consists of well-formed XML. This XML argument holds all of the necessary parameters the Web services requires to create a response (which is then returned as well-formed XML).

SOA != Web Services

SOA (Service-Oriented Architecture): An architectural concept involving the creation of loosely coupled services accessible over a network in a standardized fashion. Normally the consumption of a service is not restricted to a specific language or platform.

Web Service: A service made accessible via the Internet or an intranet. It typically uses HTTP as the transport protocol with requests to and responses from the service implemented using SOAP messages or a REST approach.

Comparison

A service-oriented architecture may be implemented using Web services as the method of exposing and accessing the services, but this is not the only way to achieve SOA. Other possible implementations could involve messaging systems (JMS: Java Messaging Service), CORBA communicating over IIOP, or any other messaging protocol (STOMP: Streaming Text Orientated Messaging Protocol, XMPP: Extensible Messaging and Presence Protocol).

6. Click Next and select a library and name for the proxy object. Click Next and confirm the proxy settings and click Finish (see Figure 3).

Now you should have a proxy object present in your library that can be deployed to create all of the objects needed to access the Amazon Web service. To use the new objects, you will need to create a SoapClient connection (to use the SoapClient object you will need to add either pbsoapclient90.pbd (PowerBuilder 9) or pbsoapclient100.pbd (PowerBuilder 10) to your library list).

If you examine the awseCommerceServicePort object, you will see it has an ItemSearch function that takes various arguments. We can call this function to send a request to our Web service and capture the response.

This is very similar to how you would make a call to a remote object using CORBA from within PowerBuilder (see Listing 4).

Comparison

We have covered three techniques for making requests against a Web service. There are various pros and cons for the methods demonstrated. Table 1 summarizes the differences among these options.

Conclusion

The three methods presented here demonstrate the options available to the PowerBuilder developer with regard to Web services in the new SOA paradigm (for more information on SOA and Web services, see the sidebar **SOA != Web Services**). Those of you on older versions can use the GetURL/PostURL functions and the XmlHttp object, while the developers on supported versions can take advantage of the new Web Service Proxy object. The Web Service Proxy does a fantastic job of abstracting away Web services to the level of distributed method calls. This alone is worth the price of an upgrade if you are going to be working frequently with Web services. Remember, shiny new PowerBuilder CDs are the gift that keeps on giving.

Resources

- *SOA:* http://en.wikipedia.org/wiki/Service-oriented_architecture
- *Web Service:* http://en.wikipedia.org/wiki/Web_service
- *REST:* http://en.wikipedia.org/wiki/REST
- *SOAP:* http://en.wikipedia.org/wiki/SOAP
- *Amazon Web Services:* http://aws.amazon.com
- *Microsoft MSXML Download:* http://www.microsoft.com/downloads/details.aspx?familyid=3144b72b-b4f2-46da-b4b6-c5d7485f2b42&displaylang=en#filelist ▼

dhp@acm.org

Listing 1

```
nvo_internet_result lnvo_internet_result
inet linet_base
integer li_retval
String ls_url,  ls_args

//Initialize InternetResult object and
//Get Internet service reference
lnvo_internet_result = create nvo_internet_result
li_retval = GetContextService("Internet", linet_base)

//Web Service URL, note the question mark included
//on the end.  Don't forget this
ls_url="http://webservices.amazon.com/onca/xml?"

//Argument list of name/value pairs in the format
//<name>=<value>&
ls_args="Service=AWSECommerceService&" + &
        "AWSAccessKeyId=[YourAccessKey]&" + &
        "Operation=ItemSearch&" + &
        "SearchIndex=Music&" + &
        "Keywords=Butch Walker"

//Concatenate URL with Arguments and call GetURL method.
//The response from the web service will be sent to the
//InternetData method in the lnvo_internet_result object
//and execute the code there (popping up a MessageBox)
li_retval = linet_base.GetURL(ls_url + ls_args, &
                              lnvo_internet_result)

//Clean up
destroy linet_base
destroy lnvo_internet_result
```

Listing 2

```
nvo_internet_result lnvo_internet_result
inet linet_base
integer li_retval
long ll_content_length, long ll_port
String ls_url, ls_args, ls_headers
blob lblb_args

//Initialize InternetResult object and
//Get Internet service reference
lnvo_internet_result = create nvo_internet_result
li_retval = GetContextService("Internet", linet_base)

//Web Service URL, note the question mark included
//on the end.  Don't forget this
ls_url="http://webservices.amazon.com/onca/xml?"

//Server Port Number, typically 80 or 0 for default
//Note putting 443 here does not give you HTTPS
ll_port = 80

//Argument list of name/value pairs in the format
//<name>=<value>&
ls_args="Service=AWSECommerceService&" + &
        "AWSAccessKeyId=[YourAccessKey]&" + &
        "Operation=ItemSearch&" + &
        "SearchIndex=Music&" + &
        "Keywords=Butch Walker"

//For PostURL we will convert our argument list
//from a string to a blob and get the length of
//our resulting blob
lblb_args = blob(ls_args)
ll_content_length = Len(lblb_args)

//For PostURL we also have to set two HTML headers
//Content-Length and Content-Type separated by two
//newline characters
ls_headers = "Content-Length: " + &
             String(ll_content_length) + "~n~n" + &
             "Content-Type: " + &
             "application/x-www-form-urlencoded"
```

```
//We call PostURL with our additional parameters (port
being
//The response from the web service will be sent to the
//InternetData method in the lnvo_internet_result object
//and execute the code there (popping up a MessageBox)
li_retval = linet_base.PostURL(ls_url, &
                               lblb_args, &
                               ls_headers, &
                               ll_port, &
                               lnvo_internet_result)


//Clean up
destroy linet_base
destroy lnvo_internet_result
```

Listing 3

```
//First download and install the latest XmlHttp package
//(this link goes to the one listed in
//the connectToNewObject call - "Msxml2.XMLHTTP.4.0"
//http://www.microsoft.com/downloads/details.aspx?
  familyid=3144b72b-b4f2-46da-b4b6-c5d7485f2b42&display
  lang=en#filelist

//XmlHttp object method summary
//http://msdn.microsoft.com/library/default.asp?url=/libra
ry/en-us/xmlsdk/html/xmmscxmldommethods.asp
String ls_get_url, ls_post_url
String ls_args, ls_response
String ls_response_text, ls_status_text
long   ll_status_code
OleObject loo_xmlhttp

ls_get_url = "http://webservices.amazon.com/onca/xml?"
ls_post_url = "http://webservices.amazon.com/onca/xml"

ls_args = "Service=AWSECommerceService&" + &
          "AWSAccessKeyId=[YourAccessKey]&" + &
          "Operation=ItemSearch&" + &
          "SearchIndex=Music&" + &
          "Keywords=Butch Walker"

try
  //Create an instance of our COM object
  loo_xmlhttp = CREATE oleobject
  loo_xmlhttp.ConnectToNewObject("Msxml2.XMLHTTP.4.0")

  //First lets do a GET request
  loo_xmlhttp.open ("GET",ls_get_url + ls_args, false)
  loo_xmlhttp.send()

  //Get our response
  ls_status_text = loo_xmlhttp.StatusText
  ll_status_code = loo_xmlhttp.Status

  //Check HTTP Response code for errors
  //http://kbs.cs.tu-berlin.de/~jutta/ht/responses.html
  if ll_status_code >= 300 then
    MessageBox("GET Request Failed", ls_response_text)
  else
    //Get the response we received from the web server
    ls_response_text = loo_xmlhttp.ResponseText

    MessageBox("GET Request Succeeded", ls_response_text)
  end if


  //Lets do a POST now, notice now we will pass a String
  //in the send() call that contains the arguments in the
  //format name1=value1&name2=value2&...
  loo_xmlhttp.open ("POST",ls_post_url, false)
  loo_xmlhttp.setRequestHeader("Content-Type", &
             "application/x-www-form-urlencoded")
  loo_xmlhttp.send(ls_args)

  //Get our response
  ls_status_text = loo_xmlhttp.StatusText
  ll_status_code = loo_xmlhttp.Status

  //Check HTTP Response code for errors
  //http://kbs.cs.tu-berlin.de/~jutta/ht/responses.html
  if ll_status_code >= 300 then
    MessageBox("POST Request Failed", ls_response_text)
  else
    //Get the response we received from the web server
    ls_response_text = loo_xmlhttp.ResponseText

    MessageBox("POST Request Succeeded",
ls_response_text)
  end if

  //Done so cleanup
  loo_xmlhttp.DisconnectObject()

catch (RuntimeError rte)
  MessageBox("Error", "RuntimeError - "
    + rte.getMessage())
end try
```

Listing 4

```
long ll_retval, ll_rowcount, i
String ls_accesskeyid, ls_subscriptionid, ls_associatetag
String ls_xmlescaping, ls_validate, ls_response
SoapConnection lsoap_conn
awsecommerceserviceport lws_amazon
tns__itemsearchrequest lstr_searchrequest
tns__itemsearchrequest lstr_searchrequestarray[]
tns__operationrequest  lstr_operationrequest
tns__items lstr_items

ls_accesskeyid = "[YourAccessKey]"
ls_subscriptionid = ""
ls_associatetag = ""
ls_xmlescaping = "Single"
ls_validate = "True"
lstr_searchrequest.searchindex = "Music"
lstr_searchrequest.keywords = "Butch Walker"

lws_amazon = create awsecommerceserviceport

//Instantiate SOAP connection
lsoap_conn = create SoapConnection

// Set trace file to record soap interchange data,
ll_retval = lsoap_conn.SetOptions('SoapLog=' + &
                                '"C:\mySoapLog.log"')

ll_retval = lsoap_conn.CreateInstance(lws_amazon, &
                            "awsecommerceserviceport")

if ll_retval <> 0 then
     as_errmsg = "Unable to create proxy.  " + &
                 "Error Code: " + String(ll_retval)
else
     // make call to web service
     try
          lstr_items = lws_amazon.itemsearch
            (ls_subscriptionid, &
             ls_accesskeyid, &
             ls_associatetag, &
             ls_xmlescaping, &
             ls_validate, &
             lstr_searchrequest, &
             lstr_searchrequestarray, &
             lstr_operationrequest)

          ll_rowcount = lstr_items.totalresults

          //Iterate our response for information
          for i=1 to ll_rowcount
            ls_response = ls_response + &
             "ASIN: " + &
             lstr_items.item[i].asin + &
             "  Artist: " + &
             lstr_items.item[i].itemattributes.
               artist[1] + &
             "  Title: " + &
             lstr_items.item[i].itemattributes.
               title + "~r~n"

          next
          MessageBox("Web Service Response",
ls_response)

     catch ( SoapException e )
          MessageBox("Soap Exception", &
                     "Cannot invoke Web Service:  " +
e.text)
     end try

end if

//Cleanup
if isValid(lsoap_conn) then
     destroy lsoap_conn
end if
```