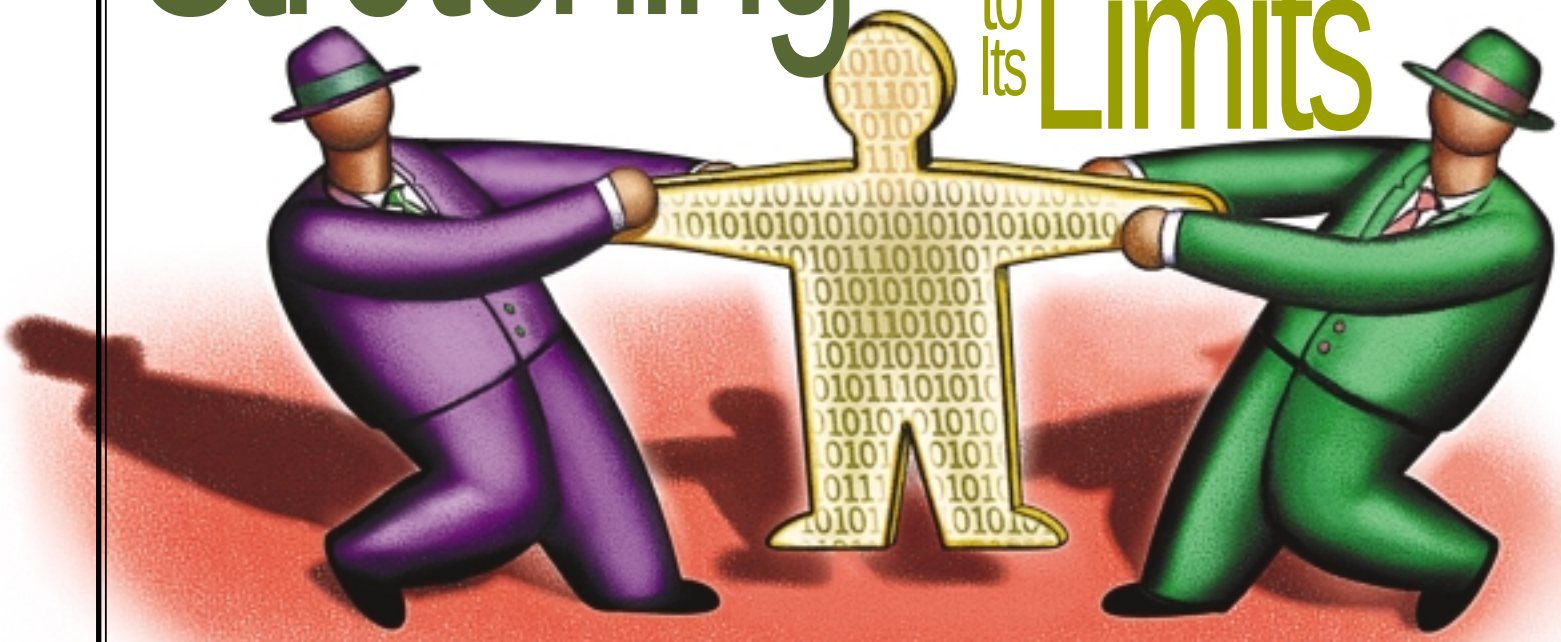
Stretching Web Services to its Limits



Dynamic invocation of simple services

XML has already solidified its position in the Web application space as the solution for data extensibility, and we are becoming increasingly aware of Web services and its potential to revolutionize the distributed application architecture. What we have yet to discover is exactly how this is all going to happen.

We know that the Web services architecture allows for easy integration between applications built independently from each other. We also know that sometime in the near future when we want to extend our applications, we will have a system in place that allows us to methodically search and consume a service from a wide selection of services.

With the overwhelming success of WSDL and the proposed UDDI specification, we have begun to work toward the goal of dynamic discovery and invocation of Web services. What we need to do now is to begin to think about what it actually will take to make these services truly dynamic. Regardless of what method we choose for discovery, our applications will need to go beyond their current scopes in order to truly make Web services dynamic.

By Ron West

Most ColdFusion developers know what WSDL is, but may not have spent a great deal of time or resources trying to figure out how it works – and why would you? CFMX makes things so easy; in most cases it would be a waste of time to learn WSDL in depth. Because WSDL is built on XML, and CFMX now has extensive support for XML, we can use things like XPath to help us discover the properties of a given Web service.

This article will give a brief introduction to the wheres and hows of WSDL, and we'll use XPath to search the WSDL file to discover the properties for a given Web service. For demonstration purposes we will apply our logic to a "Simple" Web service, which works with and returns simple data types like strings. We will not be working with complex data types such as arrays and custom data types. Although complex data types are required in order to handle more intricate Web services, we must walk before we can run.

XML and CFMX

Although true support for XML is new to CFMX, XML itself has been around for quite some time. In fact, there are many techniques available for working with XML, one of those being XPath. XPath is a W3C recommendation that allows for broad XML searching. To use XPath with CFMX you use a function called XMLSearch. XMLSearch() takes two arguments: first the XML document and second the XPath expression.

In CFMX there are a few options for creating an XML doc. In our case we will use cfhttp to get the WSDL file. XMLSearch returns an array of nodes that match your criteria. The array that is returned from the XMLSearch() function is basically an array of structures that describes all of the properties of the XML node. The following structure keys are available from a successful XMLSearch():

- ***XMLName:*** The name of the root element found
- ***XMLNsPrefix:*** The namespace prefix for this element
- ***XMLText:*** Any text contained within the open end tag for this element
- ***XMLComment:*** Any XML comments made for this element
- ***XMLAttributes:*** A structure of name-value pairs for each (if any) of the attributes for this element
- ***XMLChildren:*** A structure containing all of these keys for any children elements

Using these structure keys we will have all of the information needed to discover the properties for this service. To gain more insight on how XPath is used, go to www.w3c.org/xpath. I realize that this is a brief description of XMLSearch and XPath, but once we are done here, you should have more than enough code to test this on your own.

To perform a simple search, such as a node existence test, you could perform the following in a script block:

```
MyVar = XMLSearch(myXMLDoc, "//binding");
```

If your XML document contains a node (element) named "binding" anywhere in the document tree, MyVar would be an array with length of at least one. Likewise if we wanted to find all nodes named "operation" that have an attribute named "attr" with value "hello" we could write the following code:

```
MyVar = XMLSearch(myXMLDoc, "//operation[@attr='hello'");
```

Basically the expressions can be equated to where clauses in a standard SQL statement like: Select * from XML where node = "operation" and attributeName = "attr" and attributeValue = "hello".

WSDL Architecture (the Wheres)

The first task for working with Web services is to discover available services. With UDDI still in the "discovery" stages, we can look to our friends over at XMethods.net. The good folks over there have provided us with numerous Web services. To reduce headaches, we will concentrate only on RPC-style Web services. Let's take a look at the "BorlandBabel" Web service. The service's WSDL file can be found at: http://ww6.borland.com/webservices/BorlandBabel/BorlandBabel.exe/wsdl/IBorlandBabel.

The <description> element is the root element, where you see the namespaces for the collective properties of the WSDL file. We will not cover namespaces in this article (even though they are important, we can proceed without having to go into specifics). Next, the <message> element that defines the structure of the messages sent between the client and the host: we have four <message> elements with the names: "BabelFishRequest", "BabelFishResponse", "SupportedLanguagesRequest", and "SupportedLanguagesResponse".

Notice three of the <message> elements contain <part> elements. The <part> element defines the encoding type for that particular message (it is either a response message or call message). A message without a <part> child element ("SupportedLanguagesRequest") takes no parameters. After that, we have the <portType> element, which defines the parameters for all operations available. The <portType> attribute "name" is used as internal reference, which we will tie together a little later.

Now comes the <binding> element, which is used to provide the operation details. In our example we see two <operation> elements inside the <binding> element, which define the input and output parameters for the services named "BabelFish" and "SupportedLanguages". Finally we have our <service> element, which contains the names of the services available and the location of the service URI.

Now that we have defined the basics of the WSDL file, we can build an organized description of all of the properties of the Web service by mixing in some XMLSearch(). XMLSearch() can be used to search through an XML document using XPath syntax and expressions. Let's get started.

Dissect the Service Properties (the Hows)

There are five basic steps or pieces of information required to consume a Web service:

1. Check style of Web service (make sure that we are working with an RPC-style service)
2. Discover the name of the service
3. Discover what ports (methods) are available for this service
4. Discover what input, output, and/or fault messages are needed for each port (method)
5. Determine the data type for each message

(All references below can be found in Listing 1.)

1. Notice the search string: "//*[contains(name(), 'binding')][@style='rpc']". This means find all nodes that contain the name "binding" and have an attribute named "style" whose value is "rpc". If the length of the array is greater than 0, we have a valid service.

2. In order to enhance our documentation we will now find the name of the service. One feature available in WSDL that can be useful here is the < documentation > element. It is used to return a human-readable description for any element. In this case it would have been nice to have a < documentation> element that would describe the service for us. We could then store that with the service so that users knew what the service was designed for. To find the service name, simply use "//*[contains(name(), 'service')]". We gain access to the service name with XMLAttributes.name. We can use this service name to classify our Web service within our application.

3. Now for the methods. We use "//*[contains(name(), 'binding')]/*[contains(name(), 'operation')][@name]" to return our methods array. It is possible for a service to have multiple methods (and many do). For each method available, there will be one node in our array. To get the names of the methods, we loop through the array and get the value for the attribute "name". From here you would begin to store the data into whatever storage facility was planned. You might store each method in a database and assign it a method ID, which could be used later for the properties of the method.

4. Next we want to discover what input, output, and/or fault messages are required for each method. To do this we access the properties of each child node for the methods defined in step 3. We make another call to XMLSearch() with "//*[contains(name(), 'binding')]/*[contains(name(), 'operation')][@name='#methodName#']/*". Notice we have dynamically supplied the method name in the search string ("@name='#methodName#"). In doing so, we will have direct access to the children elements for that method. The names of the children elements will be input, output, or fault. (You can ignore the "soap:operation" node as this does not reference a message type for this method.)

   Services are not required to define both an input and output and are certainly not bound to using a fault message. It is possible to make a call to a service and not be required to supply an input message and simply receive an output message or vice versa. It is important to remember this key feature when designing dynamic systems, as you would need to abstractly handle a number of possibilities.

5. Finally, we want to discover the data types for each of the messages returned from step 4. This is a little tricky, but if you have hung in so far this should not be difficult to follow. The data types are stored separately from the message ports. We first need to discover the internal reference to the data types for each message by looking into the details of the <portType> element. We make the call "//*[contains (name(), 'portType')]/*[contains(name(), 'operation')] [@name='#methodName#']/*[contains(name(), '#param Name#')]" to gain access to the parameters of the portType for the operation.

   In SQL terms, the line would read something like this: "Select * from XML where name like 'input' and parent like 'BabelFish' "whose" parent like 'portType'. One of the attributes of the returned element will be "Namespace". The "Namespace" attribute defines the internal reference to the message definition, which will have the data types. (It is not necessary to fully understand the use of namespace for the simple service, but in order to work with complex data types this will become an important concept, as it will determine where the data-type definitions are located.) Use the namespace to get at the details for each <part> element of this message. Each <part> element will have two attributes that describe its functionality – the attribute "name" and the attribute "type". The type attribute here represents the encoding type of the message part. It is possible to have multiple parts for any given input message. Each part may have a different data type. There is only one part to an output message. ColdFusion logically maps the encoded data types to ColdFusion data types. The matrix for this mapping is located in the ColdFusion documentation.

What Should We See?

When things run smoothly – and I hope they did for you – you should get the following results for the code provided in Listing 1:

1. A service named "IBorlandBabelservice"
2. Two port (methods) with the following properties
   - a. BabelFish
     - i. Input message with parts:
       1. TranslationMode (of type string)
       2. Sourcedata (of type string)
     - ii. Output message (of type string)
   - b. SupportedLanguages
     - i. Input message with 0 (zero) parts
     - ii. Output message (of type string)

You now have all of the information to dynamically call this service and systematically incorporate it into your application. The metadata for this service can be stored in a database and invoked anywhere in your application. An interesting aspect of this service, and the reason I chose it over the original BabelFish service, is that it uses data returned from one method to instantiate another. The data returned from the "SupportedLanguages" method is a list (whitespace delimited) that contains all languages supported in the call to the "BabelFish" function. To use this service you simply call the SupportedLanguages, and the return variable could be used to populate a selection list for the translation mode in the input message of the BabelFish method. The particular technique here is advantageous over the original BabelFish service because this service provides an automated updating method. If any new languages are supported, they are simply returned in the SupportedLanguages method call. Although we have not outlined a solution to this procedure here, it deserves mention because in the future this will become a common practice.

Incidentally, if you are interested in attempting to use this code with other Simple Web services and are not sure if your Web service fits into this category, you can add the following code to your app: //*[contains(name(), 'complexType')]. If the XMLSearch with this string returns an array with length greater than 0, you are not working with a Simple Web service.

Go Forth from Here

Hopefully this article sets the wheels in motion. Developers and software vendors are adopting Web services at a rapid rate. Once UDDI is solidified and there is a common interface for discovering Web services, we will be left only with dynamic invocation. Now that CFMX natively supports both Web services and XML, we are one step closer to our goal. "Computer, find the best rate for my snowboard weekend. Oh yeah, and make sure the resort has lots of snow." We truly live in a remarkable time.

About the Author

Ron West is a senior applications developer with PaperThin, Inc., a privately held Web content management vendor headquartered in Quincy, Massachusetts. Ron has been working with Web applications for seven years. He is one of the directors of the Rhode Island ColdFusion User Group, and is an established writer for several industry publications.

rwest@paperthin.com

Listing 1:

```
<cfscript>
   //serviceURL = attributes.serviceURL;
   serviceURL =
"http://ww6.borland.com/webservices/BorlandBabel/BorlandBabel.exe/wsdl/IBor
landBabel";
   serviceData = structNew();
</cfscript>
<!--- // get WSDL File --->
<cfhttp url="#serviceURL#" method="GET" resolveurl="false">
<!--- // create CFML XML document --->
<cfset wsdl = xmlParse("#cfhttp.fileContent#")>
<cfscript>
   // test for RPC validity
   str = "//*[contains(name(), 'binding')][@style='rpc']";
   rpcArray = XMLSearch(wsdl, str);
   if ( arrayLen(rpcArray) )
   {
      // discover service name
      str = "//*[contains(name(), 'service')]";
      serviceArray = XMLSearch(wsdl, str);
      if( arrayLen(serviceArray) )
         serviceName = serviceArray[1].XMLAttributes.name;

      // discover the methods
      str = "//*[contains(name(), 'binding')]/*[contains(name(), 'opera-
        tion')][@name]";
      methodArray = XMLSearch(wsdl, str);
      // for each method returned get the properties for this method
      for( i=1; i lte arrayLen(methodArray); i=i+1 )
      {
         mStruct = methodArray[i];
         // method name
         methodName = mStruct.XMLAttributes.name;
         // write some code here to store this off into DB or wherever
           [methodId = storeMethod(methodName);]
         // discover the children nodes which will be our input, output
            and fault codes
         str = "//*[contains(name(), 'binding')]/*[contains(name(),
           'operation')][@name='#methodName#']/*";
         mDetailsArray = XMLSearch(wsdl, str);
         for( j=1; j lte arrayLen(mDetailsArray); j=j+1 )
         {
            param = mDetailsArray[j];
            paramName = param.XMLName;
            // store these with the service [messageId =
               storeMessage(methodId, paramName);]
            // discover the internal reference for each message
            str = "//*[contains(name(), 'portType')]/*[contains(name(),
              'operation')][@name='#methodName#']/*[contains(name(),
              '#paramName#')]";
            portData = XMLSearch(wsdl, str);
            // if we have a port definition
            if( arrayLen(portData) gt 0)
            {
               ref = listLast(portData[1].XMLAttributes.message, ":");
               // discover the datatypes with the internal reference to
                  the message element
               str = "//*[contains(name(), 'message')][@name='#ref#']/*";
               partArray = XMLSearch(wsdl, str);
               dump(partArray);
               // for each message discover it's parts (if any)
               for( c=1; c lte arrayLen(partArray); c=c+1 )
               {
                  part = partArray[c];
                  partName = part.XMLAttributes.Name;
                  partType = listLast(part.XMLAttributes.Type, ":");
                  // store each of these with the message data for this
                     method
               }
            }else // we did not have a port definition
               msg = "No port defined";
         }
      }
   }else // we do not have a valid RPC Style service
      msg = "Not a valid RPC Style Web service";
</cfscript>
```

CF Community *—continued from page 7*

UPDATE statements don't take more than 5–10 minutes to code (for a developer who is comfortable with basic SQL) – a <CFINSERT> or <CFUPDATE> tag takes a little less than half that time at best.

On the downside, <CFINSERT> and <CFUPDATE>:

- Can be more difficult to debug
- Encourage developers not to learn SQL
- Are not self-documenting (SQL Queries are self-documenting)
- Carry more performance overhead because ColdFusion has to do more work
- Limit developers by enforcing variable naming conventions and functionality

While the use of these tags is acceptable for beginning developers who do not yet know the basics of SQL and have a tight deadline to meet, their use isn't justifiable for any developer who could find three or four hours to sit down and learn the basics of SQL INSERT and UPDATE syntax.

<CFFORM> is a different story. <CFFORM> can be used (along with nested <CFINPUT> tags) to generate client-side JavaScript validation for form fields. The JavaScript it generates is compatible with the majority of Web browsers on the Web, and it works (you almost never have to debug it). Though it does shield developers from having to learn and/or write their own JavaScript, the <CFFORM> tag is an excellent solution for Rapid Development.

If a form needs more complex validation or validation on form field types not supported by the <CFINPUT> tag, then developers will have to write their own validation or look elsewhere for help. JavaScript manually written by developers can be a bit more streamlined and functional (as mentioned), but if all a developer's needs are more quickly met by using <CFFORM>, then it's hardly a bad practice.

I strongly suggest that all developers who want to do form validation learn JavaScript as soon as they can rather than continue to rely on <CFFORM> and be bound by its limits. Another advantage to learning JavaScript is that you'll find it much easier to learn to build applications in Flash since its language (ActionScript) is another ECMA Script–based language.

The lesson here is that it is not always (but can be sometimes) a terrible practice to let an application automate code generation for you, but it's never an excuse not to learn another technology. Learning not to be dependent on code generators will expand your possibilities, make you more marketable, and result in more robust and better thought-out applications.