Paper 004-2009

Tips and Techniques for Analytic Web Services

Dan Jahn and Brad Klenz, SAS Institute Inc., Cary, NC

ABSTRACT

SAS® BI Web services enable SAS programmers to write analytical routines and make them available to other applications through an open standard interface. When you create a SAS program to be used in a Web service, you can use some techniques that make your SAS code easier to write and easier for client applications to use. Techniques are also available for client applications when they call analytic Web services. These techniques address issues in the following areas: working with missing values, accommodating long-running routines in a Web service, customizing the contract that defines the service interface, dealing with large amounts of data, and working in a stateless protocol. An examination of each potential issue, along with tips on overcoming these issues, is included.

INTRODUCTION

One of the fundamental goals of a Web service is to allow a wide variety of clients to call and use the service. An easier-to-use service is a better service. Service providers can do a number of things to make using the service easier for clients. A little bit of extra work when creating a service can save clients a great amount of effort, as well as increase the number of clients that can call the service.

To understand the benefits of a well-described Web service interface, we must examine the interactions the client application has with the service both at run time and development time. At run time, the client application must create an XML document, send it to the service, and read the response XML document. The tricky part is creating a properly formatted XML document for the request and reading the response. Most application development environments provide two ways of calling a service: you can code the XML yourself; or you can generate a proxy that will create the XML document from a more native representation. Then, you can send that document and transform the response document back to a native representation. In Java, the tool that generates this native representation proxy is called 'wsdl2java'. In .Net, there is 'wsdl.exe' and 'svcutil.exe'. In many applications, you can interactively create a mapping (BizTalk, Sharepoint, InfoPath). In SAS, that tool is the SAS XML LIBNAME engine, which provides a native representation in the form of a SAS library (PROC SOAP is the tool you use if you code the XML yourself).

Regardless of the tool a client chooses to use, the developer of the service will simplify the development of the client by following a few tips:

- Be aware of missing values.
- Configure options on servers to remove timeouts.
- Avoid the use of xs:any.
- Use input and output parameters to define the schema.
- Define a custom schema for output.
- Define a custom schema for input.
- Use paging for large amounts of data.
- Split the analytics from the data transfer.
- Test your service.

TIP: READ MORE THAN JUST THIS PAPER

This paper is not intended as an introduction to SAS BI Web Services. There are several other sources for introductory material:

- The SAS Global Forum paper "Creating Web Services Using SAS Analytics" (Klenz and Jahn 2008) takes three real-world examples and shows how they can be used with Web services. Along the way, that paper demonstrates how to deal with large amounts of data.

- The SAS Global Forum paper "Using SAS® Business Intelligence Web Services and PROC SOAP in a Service-Oriented Architecture" (Jahn 2008) introduces both PROC SOAP and SAS BI Web Services along with the big picture of how they can be monitored and managed.

- The online documentation for PROC SOAP and SAS BI Web Services provides detailed information about how to configure and use Web services with SAS.

The following tips will be much more useful and make more sense once you are familiar with SAS and Web services.

TIP: BE AWARE OF MISSING VALUES

Missing values are a common occurrence in analytical processing. The SAS XML engine has default behavior to write missing values to output XML data streams. This default behavior uses an attribute named MISSING= on the output data elements.

To show how this default behavior works, first we will create a SAS data set with missing values and write them out with the XML engine. Here is the SAS code to do that:

```
libname m xml "c:\public\pulse_missing.xml";

data m.pulse;
  age=43; runpulse=172; output;
  age=36; runpulse=.;   output;
  age=45; runpulse=.n;  output;
run;
```

The resulting XML file will look like this:

```
<?xml version="1.0" encoding="windows-1252" ?>
<TABLE>
   <PULSE>
      <age> 43 </age>
      <runpulse> 172 </runpulse>
   </PULSE>
   <PULSE>
      <age> 36 </age>
      <runpulse missing="." />
   </PULSE>
   <PULSE>
      <age> 45 </age>
      <runpulse missing="N" />
   </PULSE>
</TABLE>
```

Notice the use of the MISSING= attribute in the above XML. Although the MISSING= attribute allows for flexibility, especially with SAS special missing values, some client applications will prefer an alternate representation.

In XML, an attribute on numeric values called "nil" is used to specify values that are similar to the concept of missing values in SAS. The nil attribute can be specified as an alternative by using tagsets with the XML engine in your SAS program.

Before adding the tagset to your SAS program, you must change your XML schema to indicate that you will be using the nil attribute. This is done by adding a nillable attribute to the elements that can contain missing values. These will most likely be added to all numeric variables, although something like numeric ID variables may be an exception. Here is what your schema element definitions will look like:

```
<xs:element name="age" type="xs:double" nillable="true"  />
<xs:element name="runpulse" type="xs:double" nillable="true" />
```

Tagsets are defined with PROC TEMPLATE. When defining the tagset for XML output, there will be a section that defines how missing values are written. In the existing XML LIBNAME generic tagset (tagsets.sasxmog), you will see how the MISSING= attribute is defined. You can see the source for the tagset with this SAS code:

```
* Get the source for the generic XML tagset;
proc template;
  source tagsets.sasxmog;
run;
```

There are 2 changes we need in the tagset: we need to add a definition for the xsi namespace and we need to change how missing values are handled. Here is the start of the template definition code:

```
libname templat 'c:\temp\templat';
proc template;
   path templat.sgf sashelp.tmplmst;

define tagset Tagsets.Sasxmog / store=templat.sgf;
   notes "SAS-XML generic XML-Data";

   define event SASTable;
      start:
         put '<TABLE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">' NL;
         break;
```

You can comment out this line, and add a new line to specify how the nil attribute will be used instead. Here is an example of that code:

```
      define event MLEVDAT;
         do / if $is_MISSING ;
*                putq " missing=" $value_MISSING;
            put   ' xsi:nil="true"' ;
            put   ' />' ;
            put   CR    ;
            break ;
         done ;
```

Now that the updated tagset is defined, you will need to explicitly name it on the XML LIBNAME statement. That is done with the TAGSET= option as follows:

```
libname m xml "c:\public\pulse_missing_nil.xml"
tagset=templat.sgf;
```

You should also modify the tagset to add a definition for the xsi namespace (xmlns:xsi="http://www.w3.org/2001/XMLSchema- instance"). The resulting XML will now look like this when missing values are present:

```
<TABLE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
   <PULSE>
      <age>43</age>
      <runpulse>172</runpulse>
   </PULSE>
```

```
   <PULSE>
      <age>36</age>
      <runpulse xsi:nil="true" />
   </PULSE>
   <PULSE>
      <age>45</age>
      <runpulse xsi:nil="true" />
   </PULSE>
</TABLE>
```

Although the special missing values are not distinguishable from regular missing values, the client applications calling your Web service may find this XML easier to consume.

TIP: CONFIGURE OPTIONS ON SERVERS TO REMOVE TIMEOUTS

SAS jobs frequently take a long time since SAS jobs are often used to analyze huge amounts of data. Sometimes it is nice to have a timeout so a client is not waiting forever for a response. Sometimes you do not want the timeout. There are several places where a timeout can be configured, so you may need to configure the timeout value in several places:

- in the SAS BI Web Services for .Net server
- in the SAS BI Web Services for Java server
- in a .Net client
- in a Java client

Note that only the StoredProcessTimeout is specific to SAS. The others are part of .Net or Java.

In the .Net server, you can set the responseDeadlockInterval property of the processModel element in the machine.config file. You can also set the executionTimeout attribute on the httpRuntime element in the web.config file.

In a Java server, as well as in a Java client, there are two properties that can cause a timeout to occur: the SO_TIMEOUT property and the CONNECTION_TIMEOUT property (both values are in milliseconds). These can be programmatically set in your Java code, or you can set them in the axis2 config file. Here is an example of how they might appear in an axis2 config file (other elements in the transportSender element may also be necessary):

```
<transportSender name="http"
      class="org.apache.axis2.transport.http.CommonsHTTPTransportSender">
  <parameter name="SO_TIMEOUT" locked="false">60000</parameter>
  <parameter name="CONNECTION_TIMEOUT" locked="false">60000</parameter>
</transportSender>
```

SAS has added a property called StoredProcessTimeout. If you do want to specify a timeout, it is recommended that you use the StoredProcessTimeout property since that has the best chance of actually stopping the SAS code from running. (If your SAS code is in a tight loop that does no I/O, it might not stop before it gets out of that loop.)

In a .Net client, the generated proxy has a proxyClass.Timeout property, where proxyClass is the class created by wsdl.exe.

TIP: AVOID THE USE OF XS:ANY

In the Web services world, a contract is defined using a Web Service Definition Language (WSDL) document. The WSDL describes the operations supported by the service. Each operation can have a schema that describes what the XML needs to look like when you call that service and what the XML that you get back from the service will look like. Starting in SAS 9.2, SAS BI Web Services provides two ways that you can customize what the contract looks like in services you create: prompts and data sources/data targets.

Within SAS BI Web Services, there are three things you can do to a Stored Process definition that will result in a service that uses xs:any:

1) Use a custom schema that specifies xs:any in the schema. (See below for how to use a custom schema.)

2) Define a Data Source or a Data Target whose type is XML Stream but does not provide a schema (by checking the **Specify schema** check box).

3) In the **Results** field on the **Execution** tab of the Stored Process definition, check Stream, and also specify the XMLA Web Service keyword on the same Stored Process definition. (If you do not specify the XMLA Web Service keyword, the generated service will use an attachment instead of inline XML, so the type would be a base64binary type.)

Using any of these options results in xs:any being present in your generated WSDL. That forces clients calling your application to use some other mechanism besides the WSDL to figure out what that XML document needs to look like. There are some client applications which simply will not work with a service that uses xs:any.

TIP: USE INPUT AND OUTPUT PARAMETERS TO DEFINE THE SCHEMA

In the Stored Process definition, the use of input and output parameters will result in XML schema getting defined in your WSDL. For example, specifying a Prompt type (note that in Stored Processes, input parameters are also referred to as 'prompts') of **Numeric** with **Allow only integer values** checked results in a schema getting generated whose type is 'xs:int'.

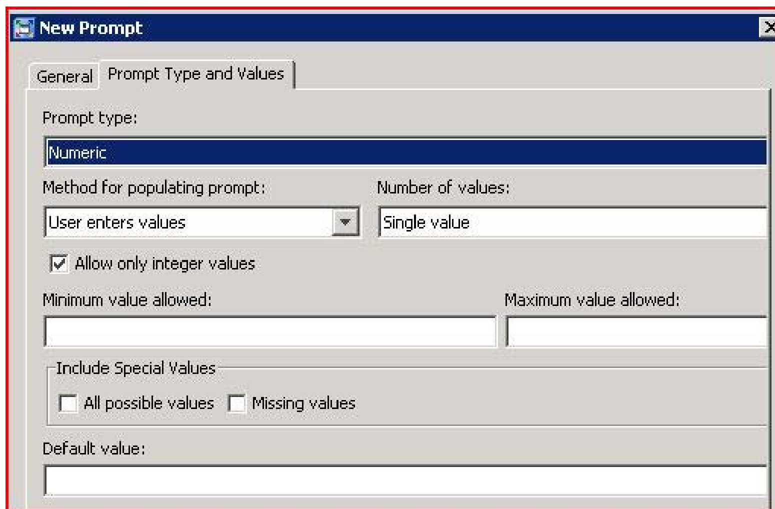

Figure 1. This prompt type generates 'xs:int' in the WSDL.

TIP: DEFINE A CUSTOM SCHEMA FOR OUTPUT

Another way you can avoid the use of xs:any and get a custom schema in your generated WSDL is to define a custom schema for your Stored Process' Data Targets. As an example, we will generate and specify a custom schema for a SAS program that returns some records from sashelp.zipcode. To generate a schema for sashelp.zipcode, follow these steps:

1. Start an interactive SAS session, and enter this code:

```
filename zipcode 'c:\temp\zipcode.xsd';

* Use the 'xml'   (not XML92) engine to generate a schema;
libname zipcode xml xmlmeta=schema;

data zipcode.codes;
set sashelp.zipcode;
run;
```

2. Make sure your XML LIBNAMEoutput is directed to a file (the filename statement above). We did this step in #1, but note that if this code were to be used in a Stored Process, that filename statement should not be used, and instead we would define a Data Target whose name is zipcode.
3. Add the xmlmeta=schema option to the XML LIBNAME statement, as above.

4. Run the program. This will generate an XML schema called zipcode.xsd that DOES NOT HAVE A NAMESPACE attribute. But we need a namespace.
5. Edit the generated schema (in our case, c:\temp\zipcode.xsd), and add targetNamespace=" http://www.tempuri.org/xml/namespace/zipcode" and xmlns="http://www.tempuri.org/xml/namespace/zipcode" and elementFormDefault="qualified" so the top lines of zipcode.xsd looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:od="urn:schemas-microsoft-com:officedata"

targetNamespace="http://www.tempuri.org/xml/namespace/zipcode"
            xmlns="http://www.tempuri.org/xml/namespace/zipcode"
            elementFormDefault="qualified">
   <xs:element name="TABLE">
…. 
```

6. Load the schema in the SAS XML Mapper (file->Open XML Schema).
   a) Generate an Automap (Tools->Automap using XSD).
   b) Under XMLMap Settings, set the version to 1.9.
   c) When you change the version to 1.9, a new table appears under the AUTO_GEN tree, called '[None]'. Select that new table. This makes the **Output** tab available.
   d) Under the **Output** tab, select the Output table (TABLE).
   e) Add xmlns as the name, and http://www.sas.com/xml/namespace/sashelp/class (matching the namespace we defined in step 5).
   f) Save the XML Map.

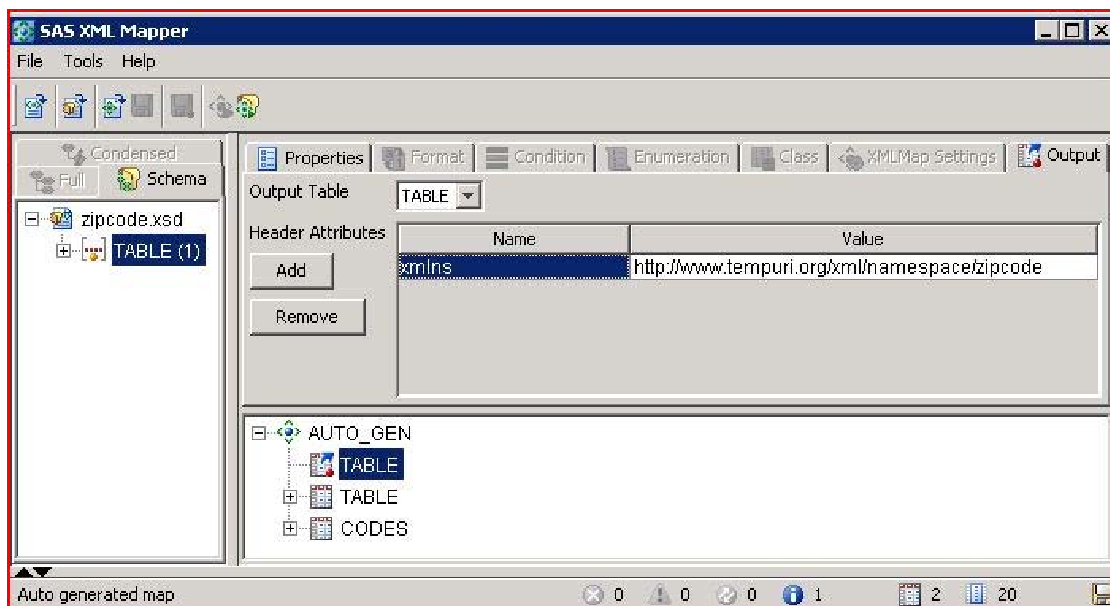

Figure 2. The SAS XML Mapper after step e.

7. Define a Data Target that uses this schema.
   a) Use type = XML Steam, label=dt, fileref=dt.
   b) Select the **Specify schema** check box.
   c) Enter a URI to the schema, such as file://c:\sasrepository\SGF2009\zipcode.xsd.
   d) Enter a reference namespace, such as `http://www.tempuri.org/xml/namespace/zipcode` (this matches the targetNamespace of our schema).
   e) Set the reference name to TABLE.
   f) Set the reference type to Schema element.

Figure 3. Define the Data Target using SAS Management Console.

8. Test your service. See the last tip in this paper for how to test a service.

TIP: DEFINE A CUSTOM SCHEMA FOR INPUT

There are several techniques you can use to generate a schema. In this example, we create the schema by hand-coding what we want the XML to look like, using a tool to generate the schema from that XML, and then using SAS XML Mapper to map the XML into a SAS data set.

The default format for an input data stream is a rectangular table, which corresponds very well with a SAS data set. This format may not be the best format for the consuming application. XML maps can be used to change the structure of the input data stream.

One example of this is a need for varying number of input measures in a multiple regression Web service.
For one call to our regression Web service, we want to fit a model of expected oxygen intake derived from measures taken during an exercise session. Our input data would look like this:

Oxygen intake (dependent variable)	Age (independent variable 1)	Weight (independent variable 2)	Run time (independent variable 3)	Run pulse (independent variable 4)
44.609	44	149.47	11.37	178
45.313	40	135.07	10.07	185
59.571	42	128.15	8.17	166
…				

This data could be sent to the Web service in XML format like the following:

```
<TABLE>
   <FITNESS>
      <Oxygen> 44.609 </Oxygen>
      <Age> 44 </Age>
      <Weight> 149.47 </Weight>
      <RunTime> 11.37 </RunTime>
      <RunPulse> 178 </RunPulse>
   </FITNESS>
   <FITNESS>
      <Oxygen> 45.313 </Oxygen>
      <Age> 40 </Age>
      <Weight> 135.07 </Weight>
      <RunTime> 10.07 </RunTime>
      <RunPulse> 185 </RunPulse>
   </FITNESS>
   <FITNESS>
      <Oxygen> 59.571 </Oxygen>
      <Age> 42 </Age>
      <Weight> 128.15 </Weight>
      <RunTime> 8.17 </RunTime>
      <RunPulse> 166 </RunPulse>
   </FITNESS>
</TABLE>
```

We would also like for our regression Web service to be able to fit models on data with a different number of independent variables. In a second call to our Web service, we want to fit a model of expected bank deposit volumes derived from macroeconomic indicators. This input data might have only three independent variables and look like the following:

Deposit volume (dependent variable)	CPI (independent variable 1)	Overnight rate (independent variable 2)	Exchange rate (independent variable 3)
360	115.2	5.5	0.66534
358	115.4	5.5	0.65703
357	115.5	5	0.64144
…			

For this data, the XML format would typically look like the following:

```
<TABLE>
   <DEPOSITS>
      <Volume> 360 </Volume>
      <CPI> 115.2 </CPI>
      <Overnightrate> 5.5 </Overnightrate>
      <Exchangerate> 0.66534 </Exchangerate>
```

```
    </DEPOSITS>
    <DEPOSITS>
       <Volume> 358 </Volume>
       <CPI> 115.4 </CPI>
       <Overnightrate> 5.5 </Overnightrate>
       <Exchangerate> 0.65703 </Exchangerate>
    </DEPOSITS>
    <DEPOSITS>
       <Volume> 357 </Volume>
       <CPI> 115.5 </CPI>
       <Overnightrate> 5 </Overnightrate>
       <Exchangerate> 0.64144 </Exchangerate>
    </DEPOSITS>
</TABLE>
```

The challenge with creating this Web service is that we need an XML schema that can describe the input data for both of these examples, plus the other variations with different numbers of independent variables. Since the number of measures can change from one call of the service to the next, an input data stream that has the measures as optional rows would be best. In this case, the values for the independent variables would be repeated sub-elements of the dependent value for each observation. Here is an example of this XML format using the fitness data:

```
<table>
  <regdata>
    <dependent name='oxygen'>44.609</dependent>
      <independent name='age'>44</independent>
      <independent name='weight'>149.47</independent>
      <independent name='runtime'>11.37</independent>
      <independent name='runpulse'>178</independent>
    </regdata>
  <regdata>
    <dependent name='oxygen'>45.313</dependent>
      <independent name='age'>40</independent>
      <independent name='weight'>135.07</independent>
      <independent name='runtime'>10.07</independent>
      <independent name='runpulse'>185</independent>
    </regdata>
  <regdata>
    <dependent name='oxygen'>59.571</dependent>
      <independent name='age'>42</independent>
      <independent name='weight'>128.15</independent>
      <independent name='runtime'>8.17</independent>
      <independent name='runpulse'>166</independent>
    </regdata>
</table>
```

This format results in a single XML schema that can be used for all the Web service calls that we want to support. We can generate the schema using a variety of XML tools to infer the schema from a sample XML file. This schema looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
targetNamespace="http://www.sas.com/xml/schema/analytics/regression-1.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="table">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="regdata">
```

```
            <xs:complexType>
              <xs:sequence>
                <xs:element name="dependent">
                  <xs:complexType>
                    <xs:simpleContent>
                      <xs:extension base="xs:decimal">
                        <xs:attribute name="name" type="xs:string"
                            use="required" />
                      </xs:extension>
                    </xs:simpleContent>
                  </xs:complexType>
                </xs:element>
                <xs:element maxOccurs="unbounded" name="independent">
                  <xs:complexType>
                    <xs:simpleContent>
                      <xs:extension base="xs:decimal">
                        <xs:attribute name="name" type="xs:string"
                            use="required" />
                      </xs:extension>
                    </xs:simpleContent>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

This is the schema we use when registering the Stored Process for the Web service. This schema will then be used in the WSDL to describe our input data for the Web service.

By default, the SAS XML engine expects input data tables to be in a simple row and column format. To customize the mapping of XML documents into SAS data sets, you can use the SAS XML Mapper. Using SAS XML Mapper and the following three steps, we use an example of the input XML data and create a map file that describes the transformations from the input format to the resulting SAS data set(s) that we need:

1. Load the schema in the SAS XML Mapper (file->Open XML Schema).
2. Generate an Automap (Tools->Automap using XSD).
3. Save the XML Map.

In the case of our regression Web service, we will be creating two SAS data sets from the input XML. The first SAS data set will contain the values of the dependent variable. The second data set will contain all the values of the independent variables.

Following the three steps above, the generated map looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- ############################################################ -->
<!-- SAS XML Libname Engine Map -->
<!-- Generated by XML Mapper, 902000.1.3.20080219191837_v920 -->
<!-- ############################################################ -->
<SXLEMAP name="AUTO_GEN" version="1.2">

    <!-- ############################################################ -->
    <TABLE name="dependent">
```

```
        <TABLE-DESCRIPTION>dependent</TABLE-DESCRIPTION>
        <TABLE-PATH syntax="XPath">/table/regdata/dependent</TABLE-PATH>

        <COLUMN name="_dependent_ID" ordinal="YES">
            <INCREMENT-PATH beginend="BEGIN"
syntax="XPath">/table/regdata</INCREMENT-PATH>
            <TYPE>numeric</TYPE>
            <DATATYPE>integer</DATATYPE>
        </COLUMN>

        <COLUMN name="name">
            <PATH syntax="XPath">/table/regdata/dependent/@name</PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>32</LENGTH>
        </COLUMN>

        <COLUMN name="dependent">
            <PATH syntax="XPath">/table/regdata/dependent</PATH>
            <TYPE>numeric</TYPE>
            <DATATYPE>double</DATATYPE>
        </COLUMN>

    </TABLE>

    <!-- ############################################################ -->
    <TABLE name="independent">
        <TABLE-DESCRIPTION>independent</TABLE-DESCRIPTION>
        <TABLE-PATH syntax="XPath">/table/regdata/independent</TABLE-PATH>

        <COLUMN name="_independent_ID" ordinal="YES">
            <INCREMENT-PATH beginend="BEGIN"
syntax="XPath">/table/regdata</INCREMENT-PATH>
            <TYPE>numeric</TYPE>
            <DATATYPE>integer</DATATYPE>
        </COLUMN>

        <COLUMN name="name">
            <PATH syntax="XPath">/table/regdata/independent/@name</PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>32</LENGTH>
        </COLUMN>

        <COLUMN name="independent">
            <PATH syntax="XPath">/table/regdata/independent</PATH>
            <TYPE>numeric</TYPE>
            <DATATYPE>double</DATATYPE>
        </COLUMN>

    </TABLE>

</SXLEMAP>
```

Now that we have created the map, we need to use it in our Stored Process. The map must be stored in a location accessible to the Stored Process when it is executing. The map is specified using the XMLMAP= option on the

LIBNAME statement for the XML engine. Here is the SAS code for our Stored Process that specifies the map and uses it to read in the two input data sets defined by the map:

```
filename   SXLEMAP 'C:\Public\StoredProcess\regmap.map';
libname    regin xml xmlmap=SXLEMAP;

data dependent;
  set regin.dependent;
run;

data independent;
  set regin.independent;
run;
```

This creates two input data sets, named “dependent” and “independent.” The data sets have one row for each measurement value, with an ID variable to link dependent value with the corresponding independent values for the same observation.

An example of the “dependent” data set using the fitness data would look like this:

_dependent_ID	name	dependent
1	oxygen	44.609
2	oxygen	45.313
3	oxygen	59.571

An example of the corresponding “independent” data set would look like this:

_independent_ID	name	independent
1	age	44
1	weight	149.47
1	runtime	11.37
1	runpulse	178
2	age	40
2	weight	135.07
2	runtime	10.07
2	runpulse	185
3	age	42
3	weight	128.15
3	runtime	8.17
3	runpulse	166

Note that within our data sets and SAS code we are using generic names for the dependent and independent values. This is because our Web service does not rely on or need the names of the variables from the client application calling the Web service.

The “tall and skinny” structure of the data sets provides maximum flexibility in the number of independent variables and number of measurement observations. As we use the data for the analysis (regression) in our SAS code, we will need the data transposed into the traditional row and column format used by PROC REG (and most other SAS procedures). The following SAS code will transpose the independent values from separate rows into columns for each observation.

```
proc transpose data=independent out=indeptrans(drop=_name_)
      prefix=independent;
  by _independent_ID;
```

```
  var independent;
run;
```

Once the independent values are transposed, we can merge the independent values with the dependent values to create our analysis data set. The merge is done with SAS code like this:

```
data regtrans;
  merge dependent(rename=(_dependent_ID=id))
        indeptrans(rename=(_independent_ID=id)) ;
  by id;
  drop id;
run;
```

The resulting analysis data set would be in the traditional format for PROC REG and look like this:

dependent	independent1	independent2	independent3	independent 4
44.609	44	149.47	11.37	178
45.313	40	135.07	10.07	185
59.571	42	128.15	8.17	166

One additional, necessary item is a list of the independent variables passed in for this call of the Web service. Since the number of independent variables can change with each call, we need the current list for the Model statement of PROC REG. Since the names for the independent variables follow a reliable pattern, we can use SAS code like the following to determine the current list of variables and save that list in a macro variable:

```
data _null_;
  length indeps $ 32000;
  if 0 then set regtrans;
  array x{*} _NUMERIC_;
  arrayvars=dim(x);
  do i = 1 to arrayvars;
    varname = vname(x{i});
    if  upcase(varname) =: "INDEPENDENT" then
         indeps = trim(indeps) || " " || varname;
  end;
  call symput("indeps", indeps);
  stop;
run;

%put Indeps=&indeps;
```

We now have everything we need to do our regression using PROC REG. Here is the code that would do the regression:

```
proc reg data=regtrans;
   model dependent = &indeps;
run;
quit;
```

This example describes a very robust technique for building a flexible regression Web service. The Web service can fit models with any number of independent variables. The input XML data is also described with a single schema. This single schema allows for a WSDL that is easily consumed by client applications.

TIP: USE PAGING FOR LARGE AMOUNTS OF DATA

When it comes to data, there is “large,” and then there is “LARGE.” The limit on the server side is about 40MB of XML, but many clients may have smaller limits (such as Flash clients). There are several strategies for dealing with really large amounts of data that have been previously covered (Klenz and Jahn 2008). For some clients, large may be more data than can fit on a screen at a time. This is a good time to use paging. To implement paging in a Web

service, we will create a Stored Process that takes two input parameters – the number of observations on a page and the page number we want to retrieve.

```
/* Before this code is run in a Stored Process,
   the SAS code will assign the dt fileref
   and it will set the input parameters
   because they are defined in metadata.  If we
   want to run this interactively, we can uncomment
   these lines.

INPUTS: pagesize (int)
        pagen (int)
OUTPUT: dt (dataTarget with zipcode.xsd schema)

filename dt 'c:\temp\a.xml';
%let pagesize=2;
%let pagen=25;
*/

libname dt xml xmlmeta=schema;

proc sql outobs=&pagesize;
  create table dt.zips as
  select ZIP,CITY,STATECODE
  from sashelp.zipcode
  where (&pagen-1)*&pagesize < monotonic()
;
quit;
```

TIP: SPLIT THE ANALYTICS FROM THE DATA TRANSFER

The above tip on using paging works really well when the data set has already been created. However, if you have to create a new data set on every call, that paging strategy may perform poorly, because you recalculate the entire table on every call. The technique here is to have one Stored Process that you can call to generate a new data set and a second Stored Process that returns the data through paging. The first Stored Process will only be called once, while the second Stored Process may be called hundreds of times. You do need to be aware of data lifetime issues and multi-user issues. If you have multiple users calling your services at the same time, you MUST generate the data into a unique table name.

TIP: TEST YOUR SERVICE

After following these tips, you will want to test your service. A nice tool for testing services is soapUI from eviware (http://www.soapui.org/). There are several features of soapUI that are really useful:

1) When you load the WSDL in soapUI, it generates a sample request document and you can observe the actual XML that clients will be sending to your service. Many of the Web service development tools try to hide this from you.

2) After calling the service, you can use soapUI to validate the response against what the WSDL says the Web service is supposed to return. In the **Response** tab, right-click in the message and click **Validate**. A popup will confirm that the returned message conforms to the service's WSDL, or a window at the bottom of the **Response** tab will show you what parts of the returned message failed to validate. This is especially important when you are defining your own schemas, since validation of the response against the WSDL may not be done by the service.

In addition to the validation features, soapUI is also useful for determining what a SOAP request needs to look like. A common development process for creating a Web service call from SAS is to use soapUI to call the service, copy the request XML into your SAS code, and use that for the PROC SOAP request fileref.

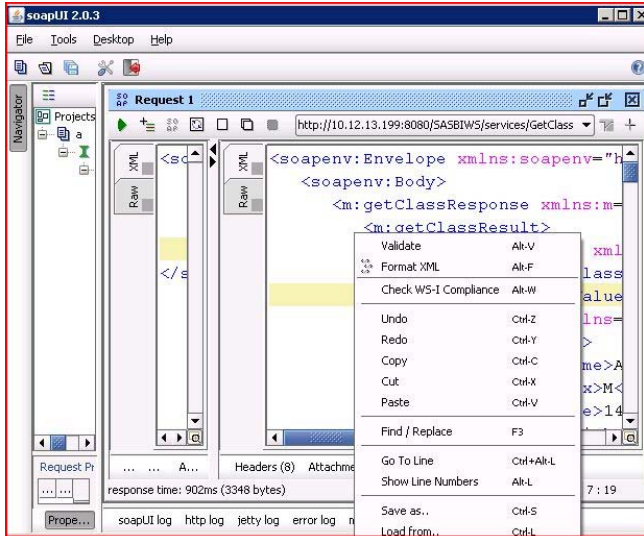

Figure 4. Using soapUI for validation.

CONCLUSION

One of the key underlying themes here is that service providers need to adapt services for the callers. You can expand the reach of the services you create by following the tips and techniques outlined here. Having more reach means more clients can use your service. More clients using the same service means less redundancy, allowing your organization to do more with less and enabling you to maintain one version of the truth.

REFERENCES

Jahn, D. 2008. “Using SAS® Business Intelligence Web Services and PROC SOAP in a Service-Oriented Architecture.” *Proceedings of the SAS Global Forum 2008 Conference*. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/rnd/papers/sgf2008/soap.pdf.

Klenz, B., and Jahn, D. 2008. “Creating Web Services Using SAS Analytics.” *Proceedings of the SAS Global Forum 2008 Conference*. Cary, NC: SAS Institute Inc. Available at http://www2.sas.com/proceedings/forum2008/010-2008.pdf.

SAS Institute Inc. 2005. SAS Note 23726. “Why is my SAS BI Web Services for .NET application timing out?” Available at http://support.sas.com/kb/23/726.html

ACKNOWLEDGMENTS

The authors would like to thank Zachary Marshall and Tony Dean for their help in reviewing this paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Dan Jahn
SAS Institute Inc.
SAS Campus Dr.
Cary, NC 27513
Work Phone: 919-677-8000
E-mail: dan.jahn@sas.com

Brad Klenz
SAS Institute Inc.
SAS Campus Dr.
Cary, NC 27513
Work Phone: 919-677-8000
E-mail: brad.klenz@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.