Paper 013-2010

Using Base SAS® to Talk to the Outside World: Consuming SOAP and REST Web Services Using SAS® 9.1 and the New Features of SAS® 9.2®.

Curtis E. Mack, Looking Glass Analytics

ABSTRACT

There has long been a trend in the computing industry of breaking isolatable computing components into separate internet accessible services. By consuming these services, many different applications can include the same functionality without having to install and maintain them separately. Many of these services are published as SOAP services, and more recently they have been published using the REST protocol. This paper will show how these services can be consumed from within a Base SAS application. Four different techniques will be show, custom coding via a SOCKETS, the newer PROC SOAP and PROC HTML, and the brand new XML92 LIBNAME engine with the XMLTYP=WSDL option.

This paper is written for the experienced SAS programmer who has little to no knowledge of web services. However, it would also be valuable to someone with experience using each independently but not together.

INTRODUCTION

Web services have been available for quite awhile now. They are the standard method for delivering data to web based applications and have many applications in batch processing and other non-web based applications as well. Google® has been a pioneer in this area with a wealth of services doing things such as generating maps, creating chart graphics, geocoding, and many others. Microsoft also has a suite of services under their Bing® brand. The list is endless and always growing. Some of these services are public and free and even more are available under many different licensing structures. Many companies also publish internally accessible services to disseminate their own information and business logic. More and more primary data sources are starting to offer web services as a way to disseminate their information. SAS® offers tools for publish web services under their Integration Technology product and other tools to create them easily are included in development tools such as Visual Studio.

The Protocols

There are two main protocols being used in web services, SOAP and REST. They each have differing strength and weaknesses.

SOAP

- Object based
- More structured
- Well integrated into code development tools.
- More powerful
- Harder to code without tools

REST

- Parameter Based
- Less Structured
- More Human Readable
- Easier to code without tools
- Has become more popular

SOAP

SOAP (Simple Object Access Protocol) is an XML based standard for publishing web services. These services use an object based paradigm, and the SOAP interface is a way of serializing the needed objects in both directions so that the processes on both sides can work with the same object concepts even though they may materialize them completely differently. Serialization is the process s of converting an object structure into a format that can be transmitted using a stream of standard characters. In the case of SOAP this is done using the XML standard. Each SOAP service publishes a service agreement in the form of a WSDL (Web Service Definition Language) file. This is an XML schema describing the objects and methods of the service. This file can be obtained by calling the "wsdl" action on the web service. This is done by adding the text "?wsdl" to the end of the service's url. (e.g. "http://wslite.strikeiron.com/censusinfolite01/CensusInfoLite.asmx?WSDL") Everything you need to know to consume the service can be found in this file. The trick is finding the pieces of information you actually need in this file which is usually rather large. Keep in mind that it is designed to be read by a machine. First let's look at what it takes to make an SOAP call with or without SAS.

These SOAP examples are using services from the company Strike Iron. They offer many web based services, including a selection of free services such as the CensusLite service used in this demonstration. To learn more please visit them at www.StrikeIron.com

A SOAP call is a broken into three parts, the header (in black on figure 1) which contains non-XML text describing the HTTP package that transmits the call, the envelope (in white on figure 1) which is the XML tag specifying which service and method is being called, and the body (in red on figure 1) which contains the XML serialized object the method being called is requires as a parameter.

There is a bit of a war in the SOAP community over how to structure the content of the SOAP Body. One format is referred to as "RPC". Unfortunately for some of you this acronym appears to refer to a much older protocol used for submitting jobs on remote machines. This use of the acronym has no relationship to that one. It specifies that the body content must contain the name of the action being called, and that all of the object definitions are completely aliased to their namespaces. This appears to be an older format but there are many services still using it. Another format is referred to as the "Document" format. In it the body contains only the object being passed, without namespace references. Yet a third format is called "Document Wrapped" and as the name implies it like the "Document" format only contains the object being passed. It however wraps that information in a tag that tells the service which action is being called. This is the format the Microsoft development tools create by default, and appears to be becoming the standard. There are other formats but they are mostly just slight variations on the RPC or Document formats. For now, the most important thing is to understand what your target service expects, and format your request accordingly.

To use a service, you need four pieces of information from the WSDL file:

1. The URL of the service
2. The method or "Action" you wish to use (there are often more than one at a service)
3. The object to be passed to the action, and its structure.
4. The object the service will return from the action and its structure.

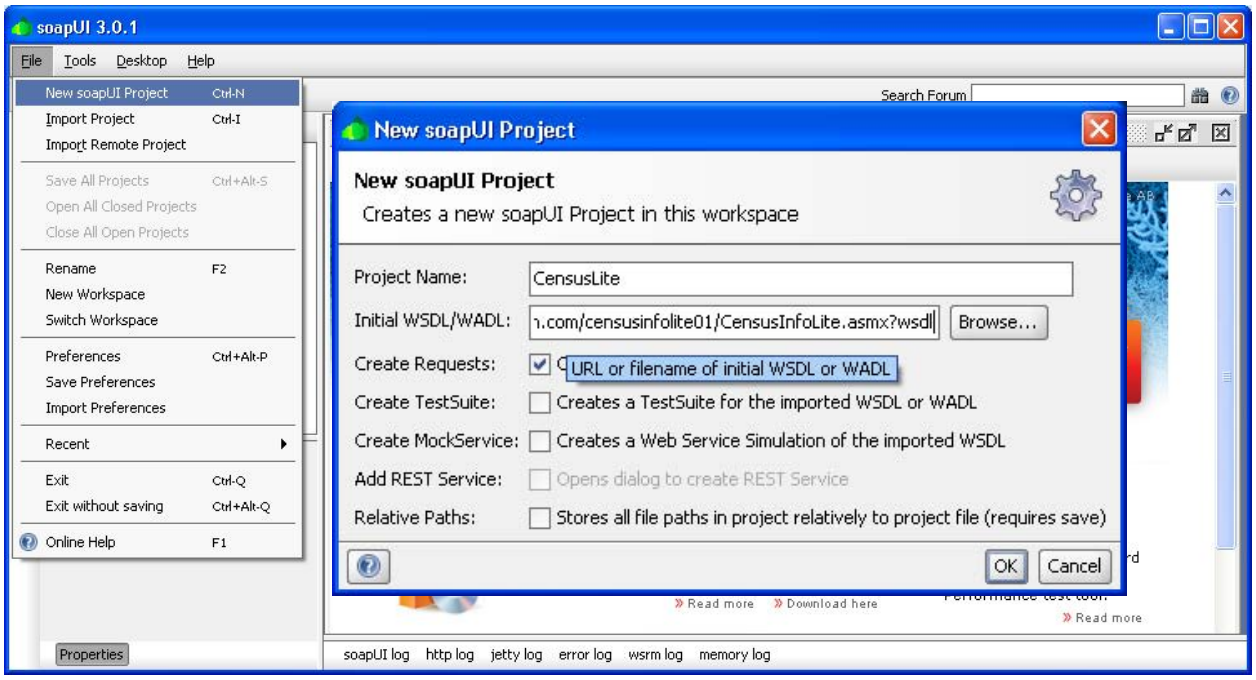
As mention earlier all of this can be obtained from the site's WSDL file. You could just open that file in a browser and read it yourself and get something like this.

```
<?xml version="1.0" encoding="utf-8" ?>
-<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        -xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
        -xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        -xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
        -xmlns:tns="http://www.wslite.strikeiron.com"
        -xmlns:s=http://www.w3.org/2001/XMLSchema
        -xmlns:soap12=http://schemas.xmlsoap.org/wsdl/soap12/
        -xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
        -targetNamespace="http://www.wslite.strikeiron.com"
        -xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
 <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Retrieves U.S. demographic  information from the U.S. Census Bureau's
Census 2000. Provides  demographic, social, economic, and housing characteristics for the given ZIPCode.
</wsdl:documentation>
- <wsdl:types>
-       <s:schema elementFormDefault="qualified" targetNamespace="http://www.wslite.strikeiron.com">
-               <s:element name="GetCensusInfoForZIPCode">
-                       <s:complexType>
-                               <s:sequence>
                                        <s:element minOccurs="0" maxOccurs="1" name="ZIPCode" type="s:string" />
                                </s:sequence>
                        </s:complexType>
                </s:element>
-               <s:element name="GetCensusInfoForZIPCodeResponse">
-                       <s:complexType>
-                               <s:sequence>
                                        <s:element minOccurs="0" maxOccurs="1" name="GetCensusInfoForZIPCodeResult" t
                        type="tns:CensusOutput" />
                                </s:sequence>
                        </s:complexType>
 </s:element>
...
```

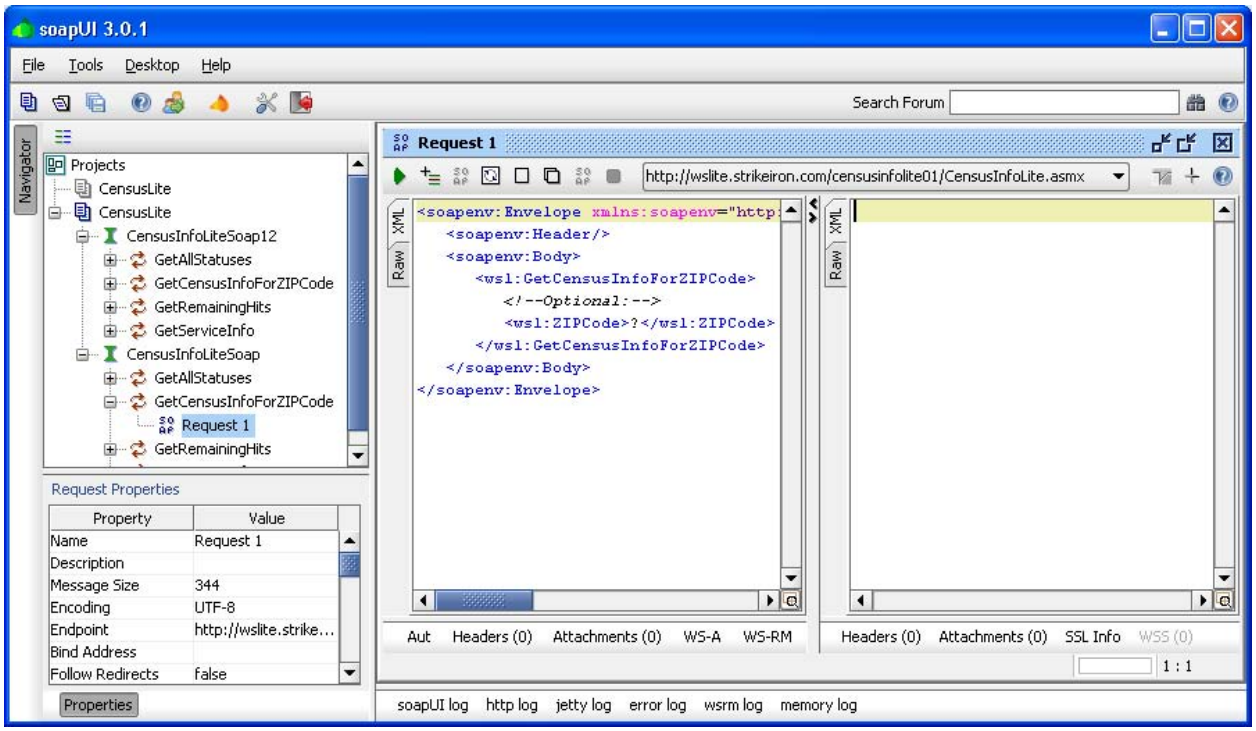
This file goes on for pages. Fortunately there are good tools to help you get the information you need from the WSDL file without much effort. One that works well and has a free version is SoapUI. We will now step through the process of getting the needed information from a WSDL file using this tool. The first step (and possibly the hardest) is obtaining the WSDL file for the service which you wish to access. Hopefully you have this already; otherwise you will need to look it up.

After opening the SoapUI tool, create a new project give it a name and the path to WSDL of the desired service. This could be a file on a local drive, but it is simplest to just enter the URL of the service

followed by "?WSDL". Here is what that should like like in SoapUi.

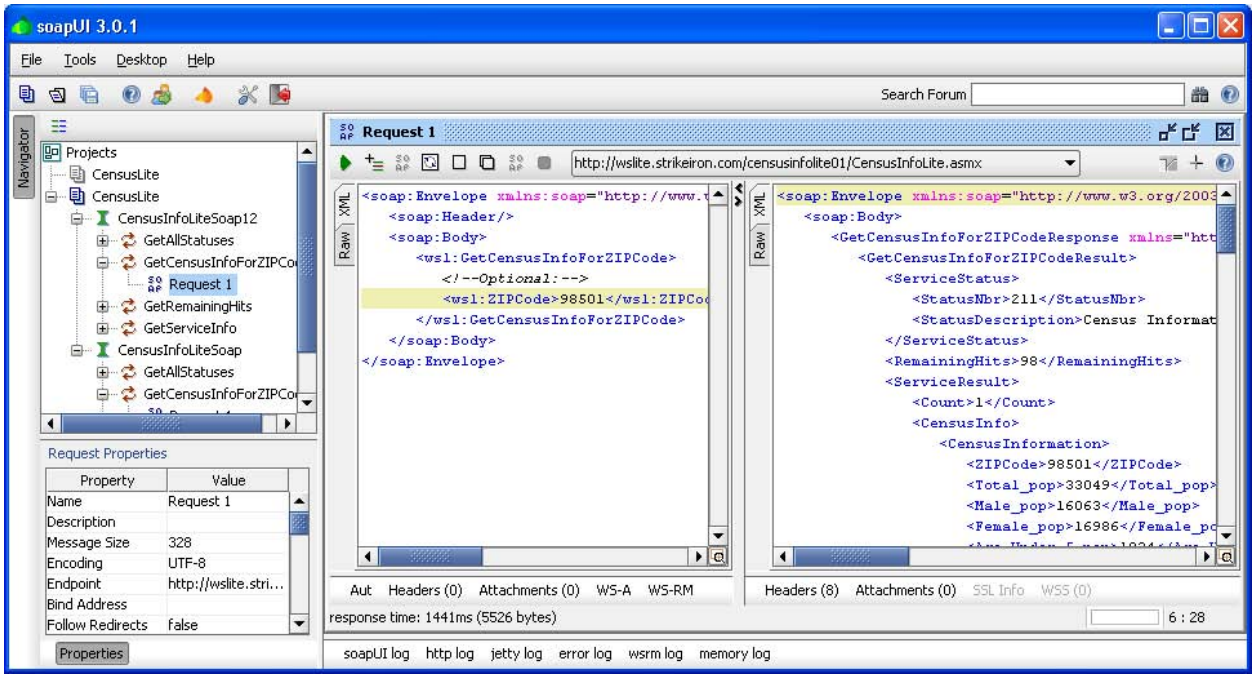


Soap UI will then read the WSDL file and create partial test requests for each of the services offered by that site. In the example below we have opened the "GetCensusInfoForZIPCode" action from the "CensusInfoLiteSoap" service.

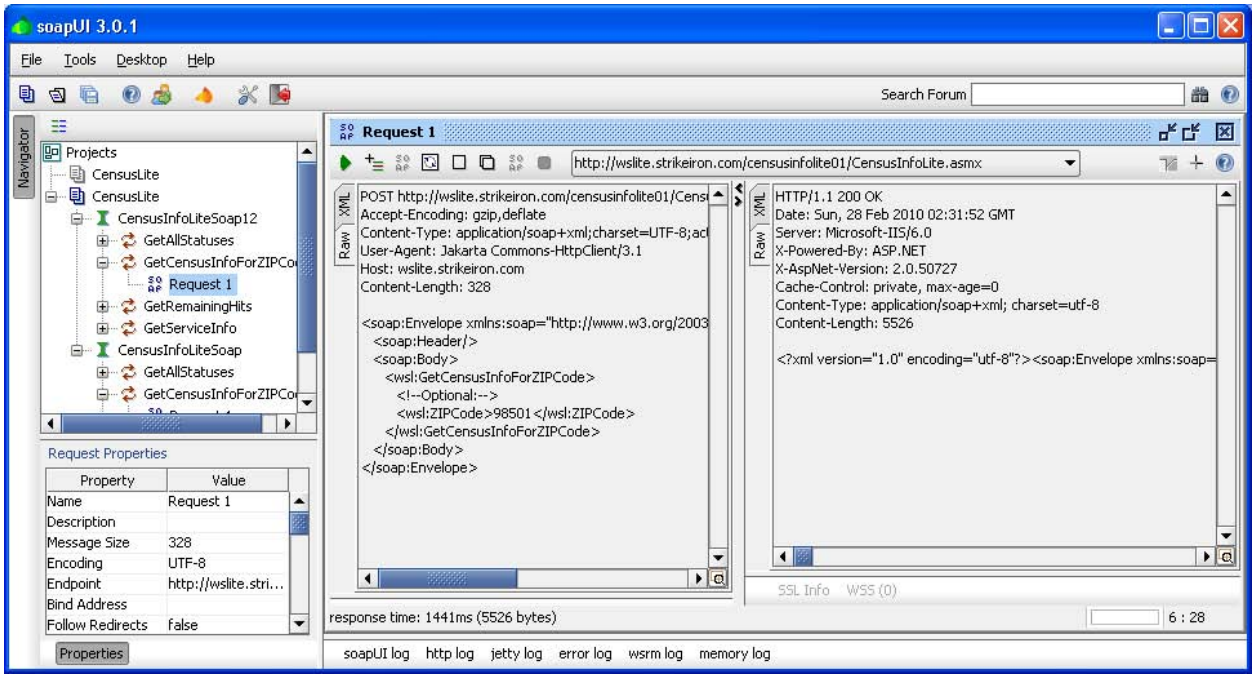


As you can see SoapUI has already generated the skeleton of a XML Envelope needed in a call to that action. The black question mark(s) indicate where a parameter should be entered.

In the next example we enter a ZIP Code of 98501 into the ZIPCode parameter. When we press the green arrow, SoapUI makes that call and shows us what is returned by the service.



In this case the service returns an XML structure that gives use various Census statistics for that ZIP Code. Clicking on the "Raw" tabs of each of these windows we are switch a view that shows us exactly what is sent to and received from the service.



To make an SOAP call in SAS 9.1 or earlier, the text in the left window is exactly what we must somehow send to the service and the text on the right is exactly what we will receive in return.

Let's take a closer look at the structure of the SOAP request in figure 1.

These are main parts of a SOAP request

1) Header - This is the HTML call to tell the server to expect a SOAP request (In black)
   a) A POST command with the filename of the service
   b) The URL of the service
   c) The name of the action being called
   d) A HTTP content type
   e) The length in bytes of the SOAP envelope being sent(Next)
2) The SOAP Envelope (in white)
   a) The XML specifications
   b) The Opening SOAP tag
      i) XML Namespace Alias for the service being called
      ii) XML Namespace Alias for the SOAP envelope standard
   c) The SOAP Body
      i) The Opening SOAP Body Tag
      ii) The Action Call (in Red)
         (1) The opening action tag
         (2) The XML describing the object the action is expecting as input
         (3) The closing action tag
      iii) The Soap Body closing tag
   d) The Soap Envelope closing tag

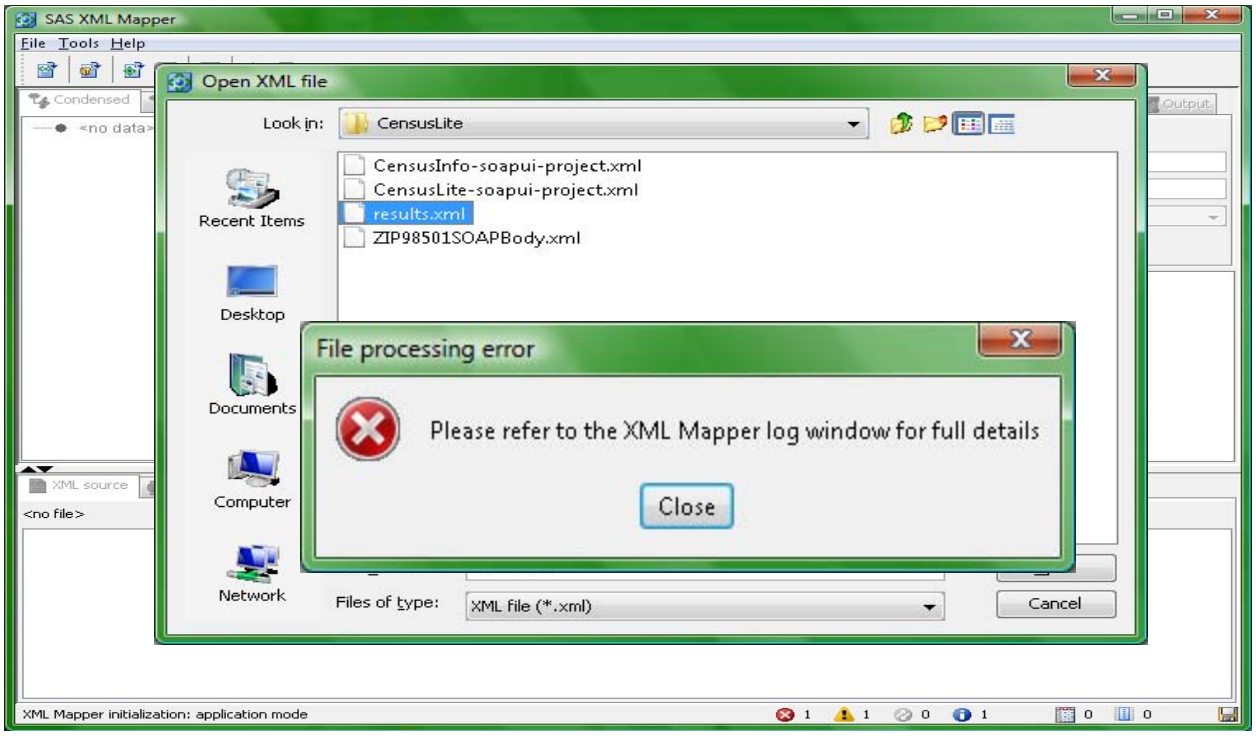
```
POST http://wslite.strikeiron.com/censusinfolite01/CensusInfoLite.asmx HTTP/1.1'
Host: wslite.strikeiron.com
SOAPAction: "http://www.wslite.strikeiron.com/GetCensusInfoForZIPCode"
 Content-Type: application/soap+xml; harset=utf-8
 Content-Length: 340
<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope
      xmlnsxmlns:wsl=http://www.wslite.strikeiron.com
      xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">'
      <soap12:Body>
            <wsl:GetCensusInfoForZIPCode>
                  <wsl:ZIPCode>98501</wsl:ZIPCode>
            </wsl:GetCensusInfoForZIPCode>
      </soap12:Body>
 </soap12:Envelope>
```

Figure 1

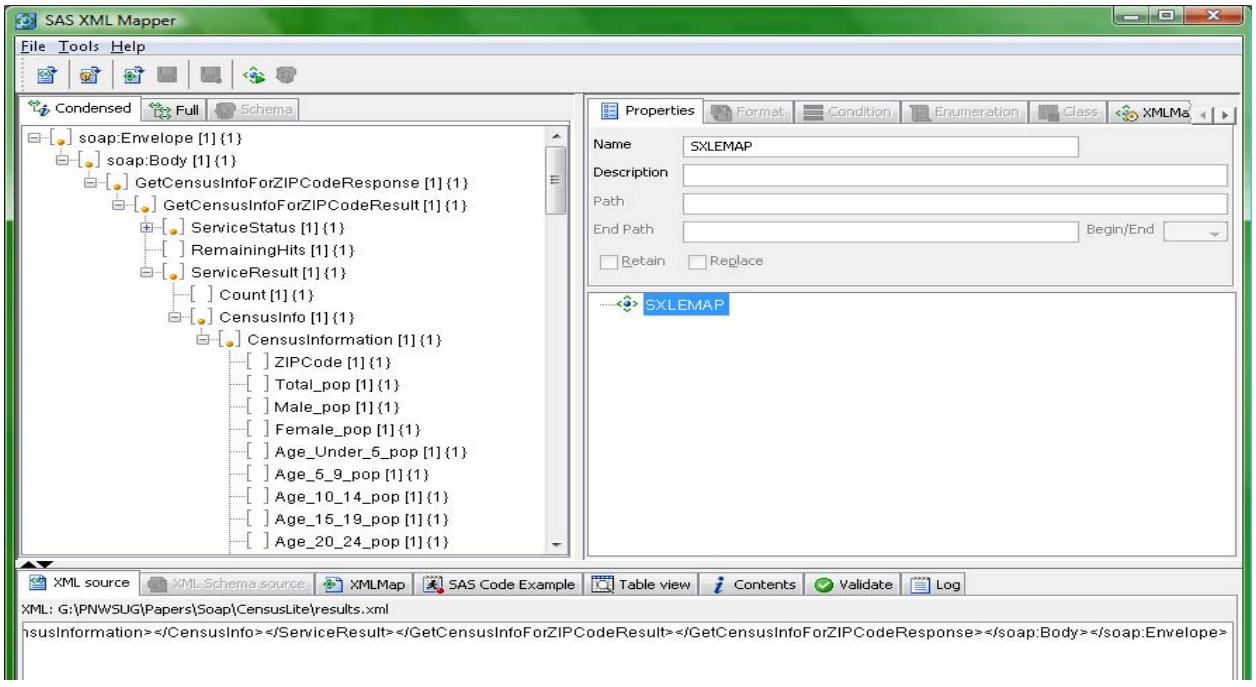
When this is passed to the service, the service will return the results.

Thanks to the SAS XML Mapper, reading the XML formatted results from an SOAP call can be pretty simple. This GUI tool creates a XML map which is itself an XML file that tells the SAS XML libname engine how to convert the object describe in an XML file into a SAS table. Since the objects returned by these services can be very complicated with multiple nested hierarchies of information, SAS needs instructions on which data elements become the rows and columns of a SAS dataset. In the following example, the result returned by above SoapUI call was saved to a file named "results.xml". We open SAS XML Mapper and then us it to open that file. Unfortunately, this is what is returned.

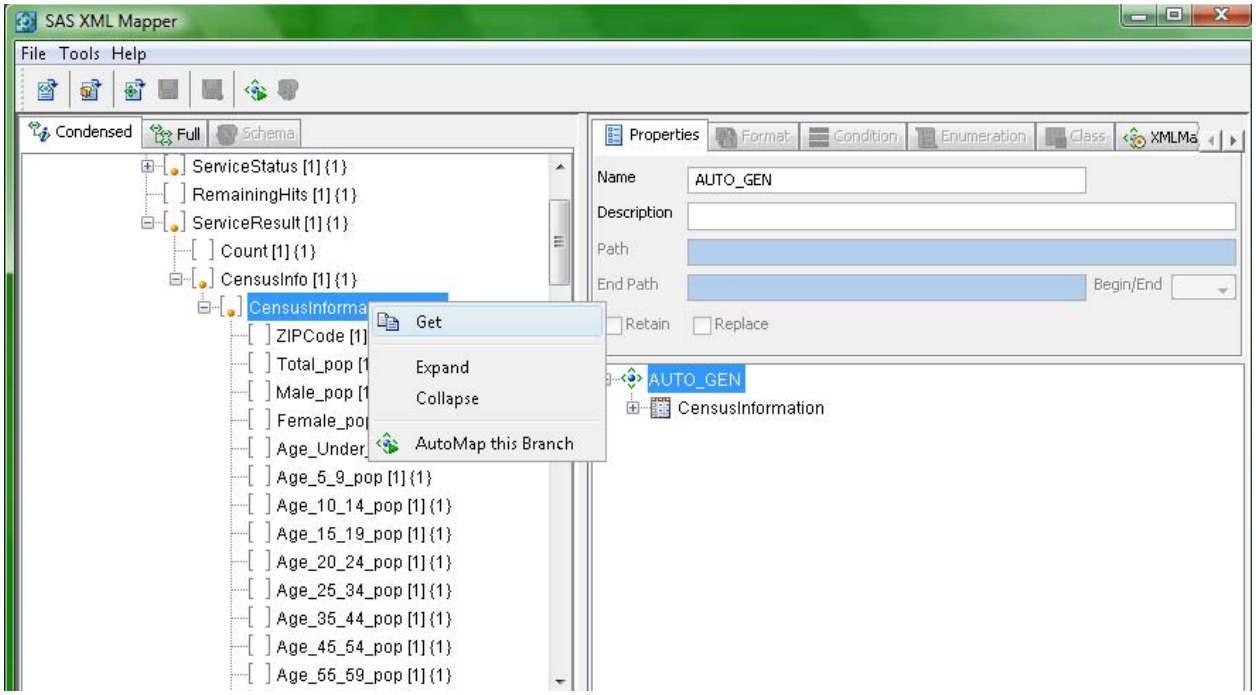
Some installations of SAS will not include the XMLMapper. It is part of the Base SAS package, but it often isn't installed by default.



It turns out that XML Mapper occasionally fails when reading complicated XML structures. There are ways to work around this. Frequently we do not need all the branches of an XML structure. If you edit the XML file and remove these branches, XML Mapper can often handle the result. As long as you maintain the hierarchy of the data you want to capture, the XML map generated on the edited version will also work on the unedited version. If you need more than one set of information from a single XML file, you can edit the file multiple times to create multiple XML maps. When the edited version of the file is opened, it looks like this.



The next step is to click and drag the desired data elements from the XML on the left to the table definition on the right. In this example, as is typical, there are several levels of hierarchy which we do not need. We select the "CensusInformation" element and clicked "AutoMap this branch" from the right mouse button pop-up menu.



We then saved the XML map to a file which we will us later.

The challenge now is how to get Base SAS to make that SOAP call and save the results. We will explore three ways to do this.

SOAP Custom Coding Method

SAS 9.2 Phase 1 (TS1M0) on Vista requires Hot Fix F9BA26 for this socket functionality to work.

This technique has been available in SAS for quite a while. It uses SAS's SOCKET filename engine to establish a direct link to the service. All of the communication is handled by code, so all of the text in **Error! Reference source not found.** must be generated and sent to the service via HTTP. Appendix A has an example macro that will make a simple SOAP call and store the resulting XML in a text file. Appendix B contains the code used to make the call used in these examples. The will step through the code generated by the macro to explain how the call is made. The green boxes are selected sections of that generated code. "SOAP_Call " macro.

In example 1 the most important thing to notice is the "SoapSrv" filename statement is using a SOCKET engine. This is the way the code opens a direct HTTP stream to the site

```
filename SoapSrv socket "wslite.strikeiron.com:80" lrecl=32767 termstr=CRLF;
filename SoapRtrn "G:\PNWSUG\Papers\Soap\CensusLite\results.xml" RECFM=N;
filename ContXML "G:\PNWSUG\Papers\Soap\CensusLite\ZIP98501SOAPBody.xml";
```

Example 1

hosting the service. This reference points to the root URL of the published service through port 80. Note that this not the full service reference, but just the high level domain. There are also a couple other parameters that are required to prevent the data streams from being truncated. Next, look at the "SoapRtrn" filename. This is the file where the information returned by the service will be stored. The record format is set to "N". This prevents the results from being arbitrarily broken into records. If this were to happen the resulting XML file would be difficult to read, particularly by software tools. The "ContXML" filename references the file containing the XML that will be sent as the body of the service request. In this example it looks like the XML in **Error! Reference source not found.**2. This XML is formatted to match the object expected by the service we are calling; in this case we are requesting Census information for ZIP Code 98501.

```
<wsl:GetCensusInfoForZIPCode>
      <wsl:ZIPCode>98501</wsl:ZIPCode>
</wsl:GetCensusInfoForZIPCode>
```

Figure 2

The trickiest part of generating a SOAP call manually is the HTTP requirement for the "Content-Length" parameter. This is the length of the entire SOAP package including the Envelope, Header, and Body. This is the reason a separate a DATA step is needed to count the number of bytes in body XML (Example 2). This step reads the data as shown in figure 2 into a SAS dataset containing just one variable while summing the length of each value. An extra character will be added between each record, so one is added to the total byte count for every record.

```
data _ContXML(drop = TotalXMLLength);
  retain TotalXMLLength 0;
  length content $32767;
  infile ContXML end=xmleof;
  input;
  content = strip(_infile_);
  TotalXMLLength = TotalXMLLength + length(content) + 1;
  if xmleof then call symput('TotalXMLLength',TotalXMLLength);
run;
```

Example 2

The next section of the code is the DATA step that makes the HTTP call, and reads the returned results. The beginning of that DATA step is show in example 3. The first thing to notice here is that the INFILE and FILE statements both point to the same "SoapSrv" filename reference. This is the nature of working with sockets. The DATA step will first write to that socket, and then it will read the results back from the same socket.

```
data _null_;
  length content1 content2 $32767;
  retain mode 1 TotalReturn ReturnLength 0;
  infile SoapSrv truncover;
  file SoapSrv;
```

Example 3

The next part of that DATA step is wrapped in a "if _n_ = 1" block so that it only executes on the first loop of the data vector (example 4). The first part of this section constructs the HTTP header of the call, and the opening and closing SOAP tags. Since the HTTP header must contain the parameter 'ContentLength' which is length of the entire SOAP envelope, all of these strings must be created up front so that the lengths of the start tag, the length of the body that was calculated earlier (the macro code resolved that to 101 in this example), and the length of the end tag can be summed before the HTTP header is constructed. To get this length correct, it must also sum in the number of end-of-line characters that will be included in the envelope. It adds 4 to cover those between the header, soap tags, and body. It also calculates the number that will be needed to work within the 32,767 byte maximum line length. To do this it divides the number of bytes in the body by that maximum line length and rounds down. In this example we have a very small body so that resolves as " int(101 / 32767)" which is of course zero. Longer bodied SOAP calls need this however. The put statement near the end writes the Header and opening SOAP tag to the socket. The last line starts tracking the number of bytes already sent to the socket. Once again, this is so it can work within the line length limit.

```
if _n_ = 1 then do;
      content1 = '<soapenv:Envelope  '||
      'xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"' || " " || 'xmlns:' ||
      "wsl=""http://www.wslite.strikeiron.com""" || ">" ||  '<soapenv:Body>';
      content2 = "</soapenv:Body>" || "</soapenv:Envelope>";
      ContentLength = length(content1) + length(content2) + 101 + 4 + int( 101 / 32767);
       call symput ('ContentLength',ContentLength);
       put "POST http://wslite.strikeiron.com/censusinfolite01/CensusInfoLite.asmx HTTP/1.1" /
            'Content-Type: text/xml; charset=utf-8' /
            "SOAPAction: ""http://www.wslite.strikeiron.com/GetCensusInfoForZIPCode""" /
            "Host: wslite.strikeiron.com" /
            'Content-Length: ' ContentLength /;
            put content1 @;
            currentLineLength = length(content1);
```

Example 4

The next section of the DATA step is still within the "if" block that only executes on the first record (example 5). It reads the body of the SOAP call from the DATASET created in example 2. It does this using the SCL Open, Fetch, and Close statements. If you are not familiar with this technique, it is a way of reading a DATA set without it being in the SET statement. Since the current DATA Step expects it data to come from the socket infile statement, this technique was needed. This loop reads each record and writes it straight to the socket. The rest of the logic is needed to insert line breaks if the maximum line length is approached.

```
      fc = open("_ContXML");
      do until(fetch(fc));
            thisLine = strip(getvarc(fc,1));
            if sum(length(thisLine),currentLineLength) + 1 >= 32767 then do;
                  put;
                  currentLineLength = 0;
            end;
            put thisLine@@;
            currentLineLength = sum(length(thisLine),currentLineLength) + 1;
      end;
      rc = close(fc);
```

Example 5

Now that the headers and body have been written, all that is left is closing the SOAP tag (example 6). At least one would think that is all that is needed, but you can see that this code writes out the closing SOAP tag stored in "content2", but it then writes out another HTTP call. It is simply sending the command to close the connection, which is good practice anyways. However, this is also a trick that was needed for this very frustrating reason. The next step in the process is to read the package returned from the service. It unfortunately will not have an end-of-line or end-of-file character at the end. As such the DATA step will never see the last record. This extra call forces the service to send another package acknowledging the close, and therefore we will get an end-of-line character between them. Up until now we have been writing to the socket. The final line of this section closes the "if" block, so the next sections are executing for each record returned from the service and reading the results.

```
        put;
        put content2 ;
        put 'OPTIONS / HTTP/1.1' /
            "Host: http://wslite.strikeiron.com/censusinfolite01/CensusInfoLite.asmx" /
            'Connection: Close' /;
end;
```

Example 6

The service will return a package much like what we just sent to it. Therefore there are three "modes" in the process of reading them. First is reading the HTTP header and SOAP tag, second is reading the body, and the final is reading the closing SOAP tag and the acknowledgement that the connection has been closed. Example 7 has the first mode. It simple reads the record coming in as being part of the header. All it does is capture the "Content-Length" of the body that will follow. If it sees a blank line, this indicates the beginning of the body, and it switches to mode 2.

```
if mode = 1 then do;
        input thisRec $ 1 - 32767;
        if thisRec =: "Content-Length:" then do;
          ReturnLength = input(scan(thisRec,2,':'),10.);
        end;
        call symput('ReturnLength',ReturnLength);
        if thisRec = " " then mode = 2;
 end;
```

Example 7

Mode 2 treats each record as part of the body. It will read each one and write the results out to a file (example 8). Since this DATA step has already declared its output FILE statement as being the socket, it must now re-declare the output FILE to be the output destination. There is no special character string indicating that the end of the body has been reached. This is why the "Content-Length" was captured earlier. When the total number of bytes returned equals that value, the process switches to mode 3.

```
    if mode = 2 then do;
          nextReturn = ReturnLength - TotalReturn;
          input thisRec $varying32767. nextReturn;
          file SoapRtrn;
          put thisRec;
          TotalReturn = TotalReturn + length(thisRec);
          if TotalReturn >= ReturnLength then mode = 3;
    end;
```

Example 8

Mode 3 is just some clean-up. It simply reads the rest of the records sent and does nothing with them. When all of this is done, we have a file containing the XML return from the service. This end of what the "SOAP_CALL" macro does.

The next steps in a larger process are be to read that XML and do something with it. This is where the XML map discussed earlier comes into play. In figure 3 that file is used to define a LIBNAME statement that point to the resulting XML. Now the results can be treated as a readonly dataset and used however needed. In our example, we get a dataset with one row with variables for several Census counts for ZIP Code 98501 like this:

```
filename XMLMap "c:\CensusLite.map";

libname results XML "c:\results.xml"
xmlmap=  XMLMap access=READONLY;
```

Figure 3

VIEWTABLE: Work.Zip98501census

	CensusInfor	ZIPCode	Total_pop	Male_pop	Female_pop	Age_Under_5_pop	Age_5_9_pop	Age_10_14_pop	Age_15_19_pop	Age_2
1	1	98501	33049	16063	16986	1934	2286	2460	2372	2035

That was a lot of work, and for those of you who have switch to SAS 9.2, work you do not need to do. However, it is always good to understand what is going on behind the scenes.

PROC SOAP

For those who have made the transition to SAS 9.2, there is now a much simpler way to make SOAP calls. The PROC SOAP procedure handles all of the port communication for you, all you need to supply is the service URL, action name (with namespace) , and a file containing the contents of the SOAP Body. Other advantages are it offers support for authetication and proxy servers. Figure 4 contains the equivalent PROC SOAP version of the example call we used describing the manual method. It reads the same XML file containing the body XML. The results can be read using the same XML map. But wait, the SAS developers have been busy, if you are using SAS 9.2 Phase 2 or later there may be a still easier way.

```
FILENAME request "temprq.xml" ;
FILENAME response "tempre.xml" ;
proc soap in=request
          out=response
         url=http://wslite.strikeiron.com/censusinfolite01/CensusInfoLite.asmx
          soapaction="http://www.wslite.strikeiron.com/GetCensusInfoForZIPCode";
run;
```

Figure 4

XMLType = WSDL

If you are using SAS 9.2 Phase 2 or later there is an even newer aproach to reading SOAP services, LIBNAME statements usign the XML engine and the “XMLType = WSDL” option. Figure 5 shows how our example might be written using this method. Unfortunately,

```
filename Census url
"http://wslite.strikeiron.com/censusinfolite01/CensusInfoLite.asmx?WSDL";
libname Census XML92 xmltype=WSDL;

proc datasets library= Census details;
run;
```

Figure 5

this example does not work. As of the time this was written, this method was very particular of the services it would support. It only supports SOAP calls using the "Document Wrapped" format mentioned earlier. Also, since it doesn't use XML Maps, the objects the called service returns must be "rectangular", meaning they have a simple row and column structure. Despite these limitations, this represents an exciting direction in SAS support of SOAP services and it will be interesting to see how this progresses. It may also be a good option for those creating their own Microsoft Windows based services, as they use the correct format and the author of such a service can define them to return objects SAS can easily recognize as tables.

REST

The REST model is a much less structured protocol, and is mostly a formalization of methods that have been used in the web sites for years before. It involves calling the URL of a service with all of the parameters of the call appended to the end of the URL as name value pairs separated by ampersands. This is the same technique used to call many web sites and how web forms submit the data they collect to their host servers. There is no standard for defining the parameters of a service nor the structure of what it returns. This information must be researched by the developer and the rules implemented in code.

A typical REST call looks something like figure 6. The URL up to the first "?" is the path to the service. Everything after that are name/parameter pairs separated by ampersands. Equal signs separate the parameter name and its value. The "%20" strings in the parameters are escaped space characters. Spaces should not be included in URLs, so this is how they are passed in parameters. There are other escape sequences needed for other special characters. The methods to read a REST service in SAS are pretty straight forward.

```
http://local.yahooapis.com/MapsService/V1/geocode?appid=XXXXXXXXXXXX-
&street=1600%20Pennsylvania%20Avenue%20NW&city=Washington&state=DC
```

Figure 6

REST Custom Coding Method

This technique has been available for many versions of SAS. Due to the existence of the URL filename engine, coding a call to a REST service is relatively simple. You just declare a FILENAME statement using the URL engine and point it to the fully qualify URL of the service with all of the parameters included. By opening that file reference using an INFILE statement in a DATA Step like you would any other file, you can read the service's response. The most difficult process is constructing that URL string using MACRO syntax like in the example in Figure 7. Most of the complexity of this example comes from the MACRO escape syntax needed to prevent any special characters from being resolved by the MACRO interpreter. Notice the URLENCODE function called by the first line. This handy function does all of the special character escaping required for the parameters being passed in the URL. This converts those spaces in the address into "%20" strings. The DATA step reads from the URL INFILE and writes the results to another file. Since the service can return anything from a text file, to XML, or even an image, we do not want SAS interpreting the incoming end-of-line characters as record breaks so we use the length statement on the INFILE statement in conjunction with the $varying INFORMAT to make SAS read the entire record without regard to special characters. We also want to prevent SAS from adding its own end-of-line characters, so we specify a "recfm=n" on the FILE statement.

```
%let AddressVal = %qsysfunc(URLENCODE(1600 Pennsylvania
Avenue));
%let CityVal = %qsysfunc(URLENCODE(Washington));
%let StateVal = DC;

%let url = %nrstr(http://local.yahooapis.com/MapsService/V1/geocode)
%nrstr(?appid=XXXXXXXXXXX-)
%nrstr(&street=)%superq(AddressVal)
%nrstr(&city=)%superq(CityVal)
%nrstr(&state=)%superq(StateVal);

filename InURL url "%superq(url)" lrecl=4000;
filename OutXML "OutXML.xml";
data _null_;
   infile InURL length=len;
   input record $varying4000. len;
   file OutXML noprint notitles recfm=n;
   put record $varying4000. len;
run;
```

The result can be of any format the service chooses. In this example it returns the XML document shown in figure 8. This can be read using the same techniques used earlier.

```
<?xml version="1.0" ?>
<ResultSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="urn:yahoo:maps" xsi:schemaLocation="urn:yahoo:maps
http://api.local.yahoo.com/MapsService/V1/GeocodeResponse.xsd">
      <Result precision="address">
            <Latitude>38.898590</Latitude>
            <Longitude>-77.035971</Longitude>
            <Address>1600 Pennsylvania Ave NW</Address>
            <City>Washington</City>
            <State>DC</State>
            <Zip>20006</Zip>
            <Country>US</Country>
      </Result>
 </ResultSet>
- <!-- ws01.ydn.gq1.yahoo.com uncompressed Fri Sep 25 10:06:25
PDT 2009   -->;
```

PROC HTTP

If you are using SAS 9.2 Phase 1 or later there is a new somewhat simpler aproach to reading REST services, PROC HTTP. Figure 9 shows our same Yahoo example using the PROC HTTP method. This method requires that the name value parameter pairs be stored in a single line file. This file is then used as one of the parameters to the PROC HTTP call, along with the URL of the service and the file in which to store the results. The resulting file is identicle to that generated using the manual method, and can be read the same way.

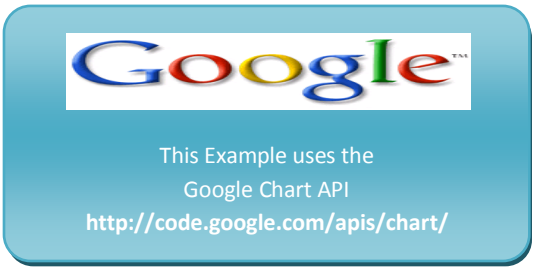
```
filename OutXML "OutXML.xml";
filename address "address.txt";

data _null_;
      file address;
      put 'appid=XXXXXXXXXXX-' @;
      put '&street=1600 Pennsylvania Avenue' @;
      put '&city=Washington' @;
      put '&state=DC';
run;
PROC HTTP
       in=address
      out=OutXML
      url="http://local.yahooapis.com/MapsService/V1/geocode";
RUN;
```

Figure 9

There are several advantages to this technique. Having the parameter string in a file removes the step of creating the complete URL using MACRO syntax, along with all the MACRO quoting that requires. PROC HTTP also automatically handles the URL formating of any special characters in the parameters. This method can also handle both GET and POST formated requests HTTP. Finally, the complexities or handling a non-text result file are transparent.

To demonstrate a service returing a non-text file, figure 10 is an example of using PROC HTTP to access a service that returns an image. This is a good example of how the REST protocol's loose structure requires the developer to do some research in order to build the parameter strings. Here are the parmeters of this call:

- The CHT parameter specifies that we want a Ven Diagram
- The first three values of the CHD parameter specify the relative sizes of three circles: A, B, and C.
- The fourth value specifies the area of circle A intersecting B.
- The fifth value specifies the area of circle A intersecting C.




```
%let WorkDir = G:\PNWSUG\Papers\Soap\GoogleChart;
filename in "&WorkDir\in";
filename out "&WorkDir\out.png";
data _null_;
  file in;
  put 'cht=v&chd=t:100,80,50,30,25,10,10&chs=500x500&chl='
run;
proc http in=in
           out=out
           url="http://chart.apis.google.com/chart?"
           method="post"
          ct="application/x-www-form-urlencoded";
run;
```

Figure 10

- The sixth value specifies the area of B intersecting C.
- The seventh value specifies the area of A intersecting B intersecting C.
- The CHS parameter is the desired image size
- The CHL parameter specifies the labels, in this case there are none.

As you can see, this would be imposible to figure out without looking at the documentation for the service. The resulting image is shown in figure 11.

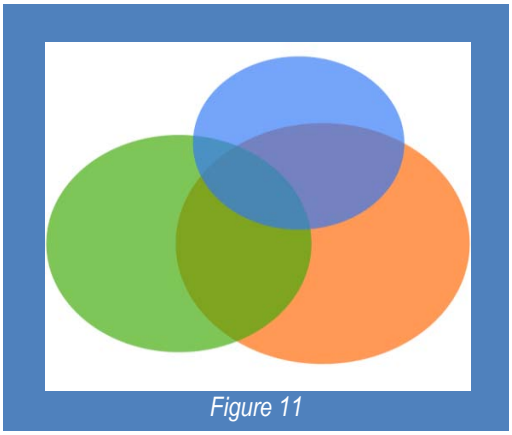


Figure 11

Conclusion

There are many different ways to consume web services from within base SAS. Given the myriad of services out there, on top of those that could be developed in house, this distributed approach to application design should be in your tool kit. SAS is actively developing in this area of their product lending further support to this being an established model that is gaining in popularity.

Warning

All of these public web services have Terms of Use agreements. Please read them before implementing an application that uses these services!

Biography

Curtis currently works for Looking Glass Analytics as a SAS consultant and GIS service manager. Prior to that, he worked for the Washington State Department of Social & Health Services Division of Research and Data Analysis, and the US Census Bureau. Hehas worked extensively with SAS for fifteen years. He has expertise in Geographic Information Systems, database design/programming particularly using Oracle and PL/SQL, and in demographic analysis. He holds a bachelors degree in Geography from the University of Washington.

Curtis Mack
Looking Glass Analytics
Curtis.Mack@lgan.com
www.LGAN.com

Acknowledgements

Apendix A

The SOAP_CALL macro

```
%macro SOAP_Call(Host,ServiceLocation,SoapAction,
                 SchemaReference,ContentXML,OutputXML);
  %let outLrecl = 32767;
  *filename SoapSrv "c:\temp\test.txt";
  filename SoapSrv socket "&Host:80" lrecl=&outLrecl termstr=CRLF;
  filename SoapRtrn "&OutputXML";
  filename ContXML "&ContentXML";
  filename junk "c:\temp\junk2.txt" lrecl=&outLrecl;
* read the Content XML into a dataset all cleaned and ready to go, while
counting the number of
bytes in the file.;
  data _ContXML(drop = TotalXMLLength);
    retain TotalXMLLength 0;
    length content $&outLrecl;
    infile ContXML end=xmleof;
    input;
    content = strip( infile );
    TotalXMLLength = TotalXMLLength + length(content) + 1; * Had to add 1
byte for the spaces that put will put
between each output element;
    if xmleof then call symput('TotalXMLLength',TotalXMLLength);
  run;
  %put TotalXMLLength = &TotalXMLLength;

  * Make the call;
  data junk;
    length content1 content2 $&outLrecl;
    retain mode 1 TotalReturn ReturnLength 0;
    infile SoapSrv truncover;
    file SoapSrv;
    if _n_ = 1 then do;
* SOAP xml open tags;
      content1 =
      '<soapenv:Envelope  '||
              'xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"' ||
" " ||
              'xmlns:' || %sysfunc(quote(%superq(SchemaReference))) || ">" ||
        '<soapenv:Body>';
*SOAP xml close tags;
      content2 =
        "</soapenv:Body>" ||
      "</soapenv:Envelope>";
* Determine the total length of the xml.  Add three for the spaces between
put statments;
      ContentLength = length(content1) + length(content2) + &TotalXMLLength +
4 + int(&TotalXMLLength / &outLrecl);
      call symput ('ContentLength',ContentLength);

      * Make the HTML call to the server telling it to expect a SOAP packet;
      put "POST &ServiceLocation HTTP/1.1" /
          'Content-Type: text/xml; charset=utf-8' /
          "SOAPAction: ""&SoapAction""" /
          "Host: &Host" /
```

```
          'Content-Length: ' ContentLength /;
      * Send the SOAP XML reading the contents from the _ContXML dataset;
      put content1 @;
      currentLineLength = length(content1);
      fc = open("_ContXML");
      do until(fetch(fc));
        thisLine = strip(getvarc(fc,1));
        if sum(length(thisLine),currentLineLength) + 1 >= &outLrecl then do;
          put;
          currentLineLength = 0;
        end;
        put thisLine@@;
        currentLineLength = sum(length(thisLine),currentLineLength) + 1;
      end;
      rc = close(fc);
      put;
      put content2 ;
* Make another generic request from the server.  This was needed because I
could not get SAS to read the last line of the information returned because
there was no end-of-line. This step just gets the server to send some more
information with some end of line characters in a way that does not generate
a server error;
      put 'OPTIONS / HTTP/1.1' /
          "Host: &ServerLocation" /
          'Connection: Close' /;
    end;
* First step, read through the HTML headers looking for the length of the
SOAP XML data returned;
    if mode = 1 then do;
      input thisRec $ 1 - &outLrecl;
      if thisRec =: "Content-Length:" then do;
         ReturnLength = input(scan(thisRec,2,':'),10.);
      end;
      call symput('ReturnLength',ReturnLength);
* A blank line signifies that the next data will be the SOAP XML;
      if thisRec = " " then mode = 2;
    end;
* Next, extract read the XML and save it to the secified file;
    if mode = 2 then do;
      nextReturn = ReturnLength - TotalReturn;
      input thisRec $varying&outLrecl.. nextReturn;
      file SoapRtrn;
      put thisRec;
      TotalReturn = TotalReturn + length(thisRec);
      if TotalReturn >= ReturnLength then mode = 3;
    end;
* Ignore any records that come after the XML, these are just the results of
the OPTIONS call;
    if mode = 3 then do;
      input;
    end;
  run;
  %put ReturnLength: &ReturnLength;
%mend SOAP_Call;
```

Apendix B

```
options mprint;
%let WorkDir = G:\PNWSUG\Papers\Soap\CensusLite;

%SOAP_Call(wslite.strikeiron.com,
           http://wslite.strikeiron.com/censusinfolite01/CensusInfoLite.asmx,
           http://www.wslite.strikeiron.com/GetCensusInfoForZIPCode,
           %nrstr(wsl="http://www.wslite.strikeiron.com"),
           &WorkDir\ZIP98501SOAPBody.xml,
           &WorkDir\results.xml);

filename CensMap "&WorkDir\CensusInformation.map";
libname Census xml "&WorkDir\results.xml" xmlmap=CensMap
               access=READONLY;

Data ZIP98501Census;
  set Census.Censusinformation;
run;
```