Paper 1937-2018

Exploring Web Services with SAS®

Richard Carey, Demarq

ABSTRACT

Web services are the building blocks of the APIs that drive the modern web. From Amazon Web Services to Philips Hue light bulbs, web services enable almost limitless integration. In this paper, we explore what a web service is and how we can use them via SAS®. We look at mechanisms to call web services from SAS using the HTTP and SOAP procedures, how to create web services from SAS® Stored Processes using SAS® BI Web Services, and consider the future of web services and SAS with SAS® Viya®, along with some practical examples.

INTRODUCTION

This paper aims to showcase calling and developing web services using the tools and techniques available in SAS. As a SAS user, you are well placed to take advantage of the proliferation of web based API's available today.

As well as calling web services, SAS Integration Technologies provides the SAS BI Web Service web application, allowing you to surface the power of SAS to your organisation via a SOAP and REST based web services interface.

This paper assumes a basic knowledge of the HTTP protocol, and the XML and JSON data formats.

WHAT IS A WEB SERVICE?

DEFINITION

The W3C defines a web service as "a software system designed to support interoperable machine-to-machine interaction over a network" (World Wide Web Consortium, 2004). That is a very broad definition, so for the purposes of this paper we will narrow it to look at services that communicate via HTTP, intended for consumption by another machine rather than a user directly.

Put simply, web services allow disparate applications and services to share data and functionality. For example, many web sites will embed a Google Map of their location on their web site, phone apps offer integration with social media, and Siri can read you the weather forecast for your current location. All of these rely on the integration of functionality provided by disparate services, published via an API and accessed over HTTP.

The "Internet of Things" is also driving change in this area. Everything from light bulbs to refrigerators are now connected to the internet. Your phone can turn on your lights as you approach home, your fridge can order food when it notices you are running low.

SAS has all the necessary tools to take part in this revolution. Ok, you might not need to use SAS to order your weekly food shop, but as this paper will show, you could if you wanted to!

A WORD ON REST, SOAP, AND OTHER STANDARDS

There are several web service protocols available that present standardised frameworks for building and consuming web services. Two of the most prevalent are SOAP (Simple Object Access Protocol) and REST (REpresentational State Transfer). The majority of modern web service API's are RESTful services, although you will still find SOAP services within the enterprise and older applications.

SOAP Services

The SOAP protocol defines the web service message structure and a mechanism for describing procedure calls and responses. Communication with a SOAP web service is handled with XML messages and can be transmitted over any reliable communication protocol.

The functionality provided by a SOAP service is commonly described by an XML file in the Web Service Description Language (WSDL) format. This file defines the input XML structure required by the service and the structure to expect as output.

RESTful Services

RESTful services are built to an architectural style rather than a strict protocol like SOAP. They make use of the HTTP stateless operators (often referred to as verbs) such as “GET”, “POST” or “PUT”, to operate on resources identified with Uniform Resource Locators (URLs).

Message format is not restricted, but is most often XML, JSON or HTML.

CALLING WEB SERVICES FROM SAS

SAS provides two main procedures for interacting with web services, the HTTP procedure and the SOAP procedure. They both follow the following broadly similar logical process; set up any input required (the request), call the service passing the request file as input, then process the output (the response). However, proc SOAP is written specifically to call SOAP web services, whereas proc HTTP is more generalised and flexible in areas such as method specification and authentication support. We will focus on the HTTP procedure here.

THE HTTP PROCEDURE

The HTTP procedure issues Hypertext Transfer Protocol requests. As such, it can support most RESTful web services, as well as SOAP.

No Input Required

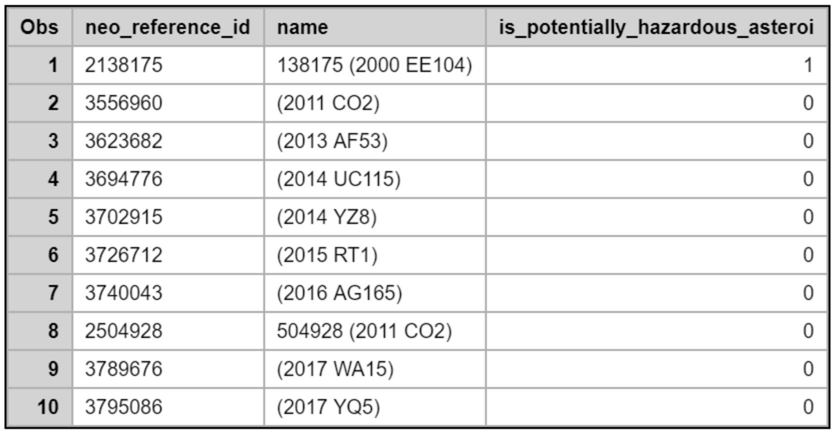
In this example we call a service provided by NASA, the Near Earth Object Web Service, to list asteroids passing close to the earth:

```
filename resp temp;

proc http url="http://www.neowsapp.com/rest/v1/feed/today"
          out=resp
          method="GET";
run;

libname jout JSON fileref="resp";

proc print data=jout.NEAR_EARTH_OBJECTS__018_01_02
run;
```

The FILENAME statement sets up a temporary output file to hold the web service response.

The HTTP procedure is used to call the URL of the service, setting the METHOD option to “GET” (the HTTP verb), and the OUT option to the “resp” fileref.

The output from the service is JSON formatted, so the JSON libname engine is used to read in the response via the “resp” fileref, which is then printed using the PRINT procedure. Figure 1 shows the output from PROC PRINT:



Obs	neo_reference_id	name	is_potentially_hazardous_asteroi
1	2138175	138175 (2000 EE104)	1
2	3556960	(2011 CO2)	0
3	3623682	(2013 AF53)	0
4	3694776	(2014 UC115)	0
5	3702915	(2014 YZ8)	0
6	3726712	(2015 RT1)	0
7	3740043	(2016 AG165)	0
8	2504928	504928 (2011 CO2)	0
9	3789676	(2017 WA15)	0
10	3795086	(2017 YQ5)	0

Figure 1 - Output from the Call to the Near Earth Object Web Service

More complex JSON can be accessed using the JSON libname engine's "map" feature, similar to using an XML map to read XML formatted data.

Providing Input

There are several common ways to provide input to RESTful web services.

Using the Query String

A "GET" request will often pass input in the query string section of the URL. This example shows an input query sent in this way to the Open Movie Database API:

```
filename resp temp;

%let query_string=s=%sysfunc(urlencode(star
wars))%nrstr(&type=movie&apikey=<get-your-own!>);

proc http url="http://www.omdbapi.com/?&query_string"
            out=resp
            method="GET";
run;

libname jout JSON fileref="resp";

proc print data=jout.SEARCH (drop=ordinal: type poster);
run;
```

First, a FILENAME statement is used to set up a temporary file to hold the response from the web service.

Then, the query string is built. The Open Movie Database API expects a parameter detailing the search term(s) and an API key (apikey) as a minimum. We also specify a filter to results of type "movie" only. Each parameter is specified as a name=value pair, separated by an ampersand, in the following format:

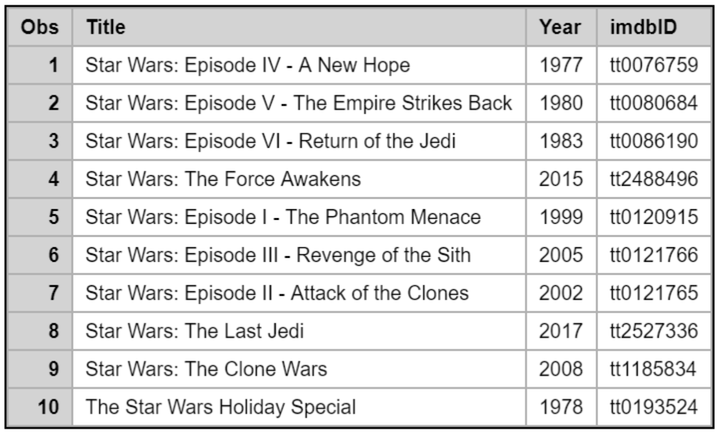
```
s=<search term>&type=<result type>&apikey=<API key>
```

The search term is URL encoded to avoid any issues with the interpretation of spaces using the URLENCODE function. The ampersands are quoted using the macro quoting function NRSTR to prevent SAS from attempting to resolve them as macro variables and printing a WARNING to the log when it fails to do so.

The HTTP procedure is used to call the URL, with the query string appended. The OUT option is set to the "resp" fileref set earlier, the METHOD option set to "GET".

The web service returns JSON formatted data, which is read using a LIBNAME statement using the JSON engine.

Figure 2 shows the output of the PRINT procedure.



Obs	Title	Year	imdbID
1	Star Wars: Episode IV - A New Hope	1977	tt0076759
2	Star Wars: Episode V - The Empire Strikes Back	1980	tt0080684
3	Star Wars: Episode VI - Return of the Jedi	1983	tt0086190
4	Star Wars: The Force Awakens	2015	tt2488496
5	Star Wars: Episode I - The Phantom Menace	1999	tt0120915
6	Star Wars: Episode III - Revenge of the Sith	2005	tt0121766
7	Star Wars: Episode II - Attack of the Clones	2002	tt0121765
8	Star Wars: The Last Jedi	2017	tt2527336
9	Star Wars: The Clone Wars	2008	tt1185834
10	The Star Wars Holiday Special	1978	tt0193524

Figure 2 - PRINT Procedure Output from The Open Movies Database

Using an Input Request

The following request does a POST to an open API set up to help when testing web service calls. We will send a new message in our input request, formatted as JSON data.

```
filename req_in temp;
filename resp_out temp;
filename head_out '~/header_out.txt';

data _null_;
  file req_in;
  put "
    {
      title: 'Pretend Post',
      body: 'This is a pretend post for SGF2018.',
      userId: 1
    }";
run;

proc http url="http://jsonplaceholder.typicode.com/posts"
            in=req_in
            out=resp_out
            headerout=head_out
            method="POST";
run;
```

First, the required filename references are setup for the request, response and the output header. A data step with the FILE statement is used to create our input request; in this case some simple JSON.

The HTTP procedure calls the required URL for the endpoint, and the IN, OUT and HEADEROUT options are set to the filerefs. The METHOD option is set to "POST".

The output header written to the "head_out" fileref is shown below.

```
HTTP/1.1 201 Created
 Date: Sun, 25 Feb 2018 13:02:13 GMT
```

```
Content-Type: application/json; charset=utf-8
Content-Length: 130
Connection: keep-alive
X-Powered-By: Express
Vary: Origin, X-HTTP-Method-Override, Accept-Encoding
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Pragma: no-cache
Expires: -1
Access-Control-Expose-Headers: Location
Location: http://jsonplaceholder.typicode.com/posts/101
X-Content-Type-Options: nosniff
Via: 1.1 vegur
Server: cloudflare
```

There are several pieces of useful information in the header. Line 1 shows the HTTP return code for the request, in this case "201 created", showing our post was successfully registered. The HTTP return code is the first thing to test in an error checking routine for a web service call.

The header also shows the format of the output that was sent. This information is held in the "Content-Type" field, and in this case is JSON formatted data in the UTF8 encoding.

The output from the service itself, written to the "resp_out" fileref, is shown below:

```
{
    {
            "title": 'Pretend Post',
            "body": 'This is a pretend post for SGF2018.',
            "userId": 1,
            "id": 101
    }
 }
```

CREATING WEB SERVICES IN SAS

THE SAS BI WEB SERVICES APPLICATION

Included with SAS Integration Technologies, the SAS BI Web Services Application (SASBIWS) is a middle tier application that allows SAS Stored Processes to be called via SOAP and REST endpoints, effectively turning them into Web Services.

Hello World!

We can call the sample "Hello World!" stored process as an example of this concept, using the SOAP protocol. In this example the request is built and sent using the free program SoapUI.

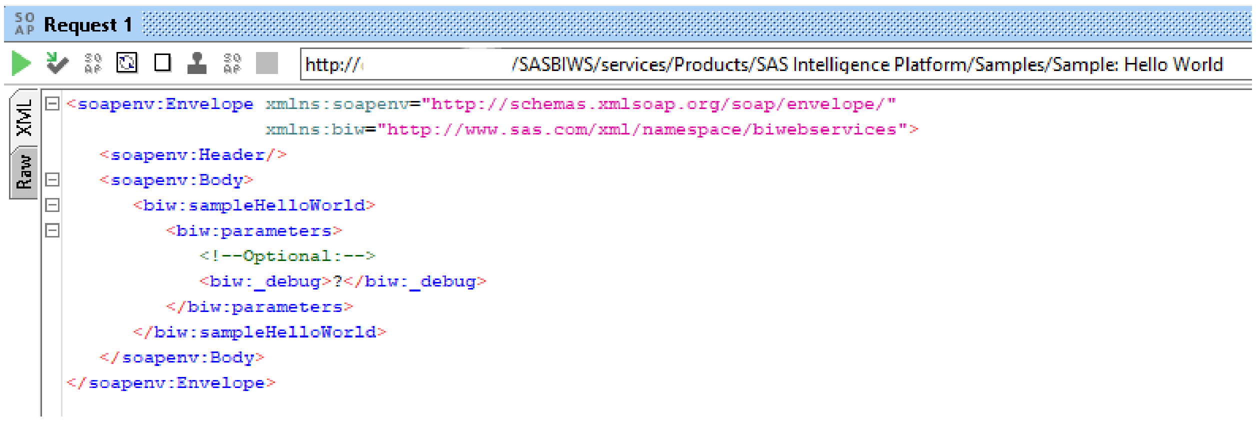
The URL is built from the web application server root, the path to the SASBIWS SOAP endpoint (SASBIWS/services) and the metadata folder path to the stored process appended together:

http://hostname:port/SASBIWS/services/Products/SAS%20Intelligence%20Platform/Samples/Sample%3A%20Hello%20World

In this example, the web application server URL is http://hostname:port/, the SOAP endpoint is "SASBIWS/services" and the default path to the "Hello World" sample stored process is "/Products/SAS Intelligence Platform/Samples/Sample: Hello World". Note that any spaces and special characters are URL encoded.

For the SOAP endpoint, the WSDL can be accessed by appending "?wsdl" to the URL.

The sample request built automatically from the WSDL using the functionality provided by SoapUI is shown in Figure 3.

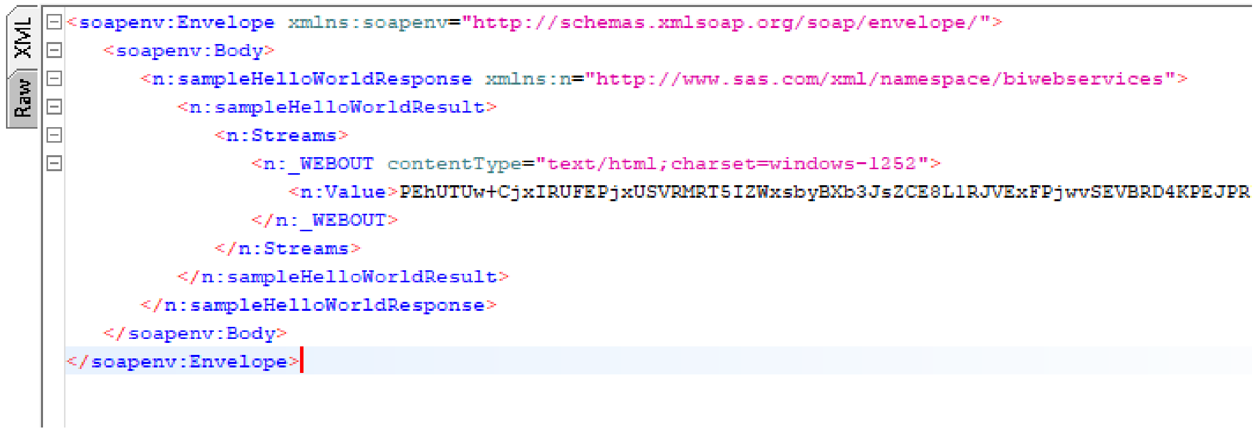


```
http://        /SASBIWS/services/Products/SAS Intelligence Platform/Samples/Sample: Hello World
```

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:biw="http://www.sas.com/xml/namespace/biwebservices">
   <soapenv:Header/>
   <soapenv:Body>
      <biw:sampleHelloWorld>
         <biw:parameters>
            <!--Optional:-->
            <biw:_debug>?</biw:_debug>
         </biw:parameters>
      </biw:sampleHelloWorld>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 3 - Default Request Built from the "Hello World" Sample Stored Process

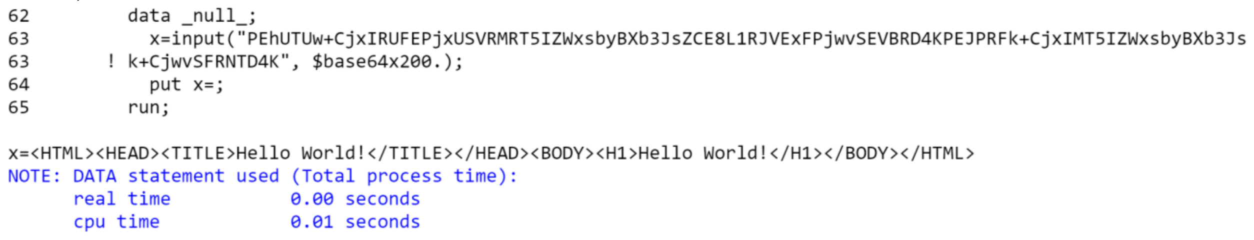
If we run this request, we get the response shown in Figure 4.



```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Body>
      <n:sampleHelloWorldResponse xmlns:n="http://www.sas.com/xml/namespace/biwebservices">
         <n:sampleHelloWorldResult>
            <n:Streams>
               <n:_WEBOUT contentType="text/html;charset=windows-1252">
                  <n:Value>PEhUTUw+CjxIRUFEPjxUSVRMRT5IZWxsbyBXb3JsZCE8L1RJVExFPjwvSEVBRD4KPEJPR
               </n:_WEBOUT>
            </n:Streams>
         </n:sampleHelloWorldResult>
      </n:sampleHelloWorldResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

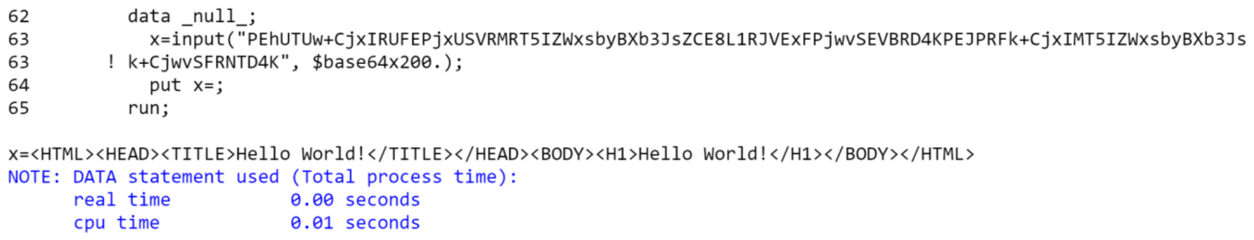
Figure 4 - Response from the "Hello World!" Stored Process Sample

The output of the “Hello World” stored process is a simple HTML page, streamed to the _WEBOUT fileref. In the response, SAS has returned this HTML, Base 64 encoded. If we decode this string we can see the HTML, shown in



```
62         data _null_;
63           x=input("PEhUTUw+CjxIRUFEPjxUSVRMRT5IZWxsbyBXb3JsZCE8L1RJVExFPjwvSEVBRD4KPEJPRFk+CjxIMT5IZWxsbyBXb3Js
63       ! k+CjwvSFRNTD4K", $base64x200.);
64           put x=;
65         run;

x=<HTML><HEAD><TITLE>Hello World!</TITLE></HEAD><BODY><H1>Hello World!</H1></BODY></HTML>
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
```

Figure 5.



```
62         data _null_;
63           x=input("PEhUTUw+CjxIRUFEPjxUSVRMRT5IZWxsbyBXb3JsZCE8L1RJVExFPjwvSEVBRD4KPEJPRFk+CjxIMT5IZWxsbyBXb3Js
63       ! k+CjwvSFRNTD4K", $base64x200.);
64           put x=;
65         run;

x=<HTML><HEAD><TITLE>Hello World!</TITLE></HEAD><BODY><H1>Hello World!</H1></BODY></HTML>
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds
```

Figure 5 - Decoding Base64 using SAS

This output is not very useful, but there are several ways we can write a stored process specifically with web services in mind.

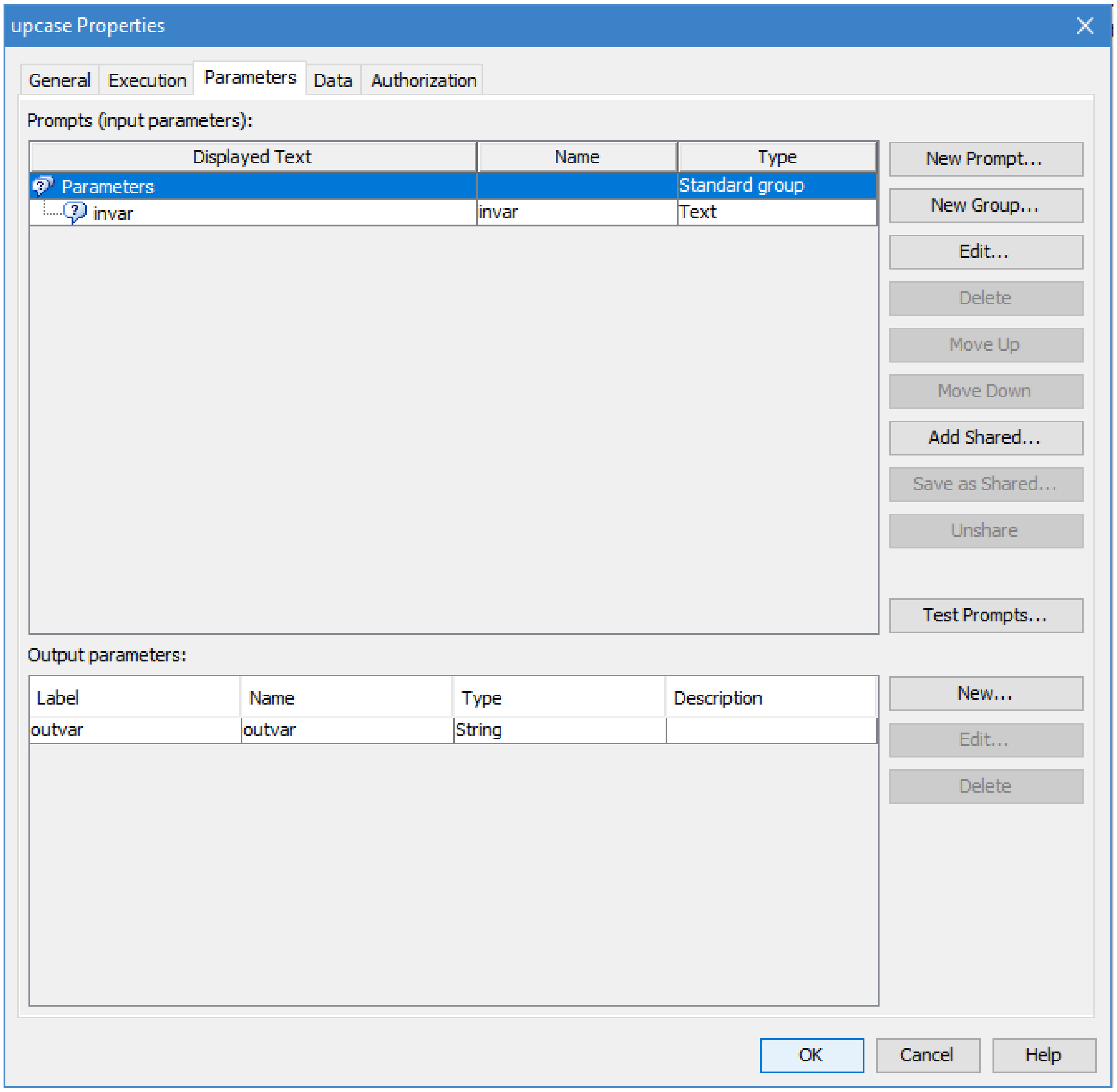
A Better Example – Parameters

The SAS BI Web Services Application offers several mechanisms to call a single stored process. In this paper we will cover SOAP as well as RESTful calls using both XML and JSON.

Let’s create a better example, that takes some input and produces a readable output in the response. The following very simple SAS code takes a string passed as a macro variable and returns it as upper case:

```
  %let outvar=%sysfunc(upcase(&invar));
```

We define the input and output parameters expected in the stored process definition. The “Parameters” tab of the properties window is shown in Figure 6



Prompts (input parameters):

Displayed Text	Name	Type
Parameters		Standard group
invar	invar	Text

Output parameters:

Label	Name	Type	Description
outvar	outvar	String	

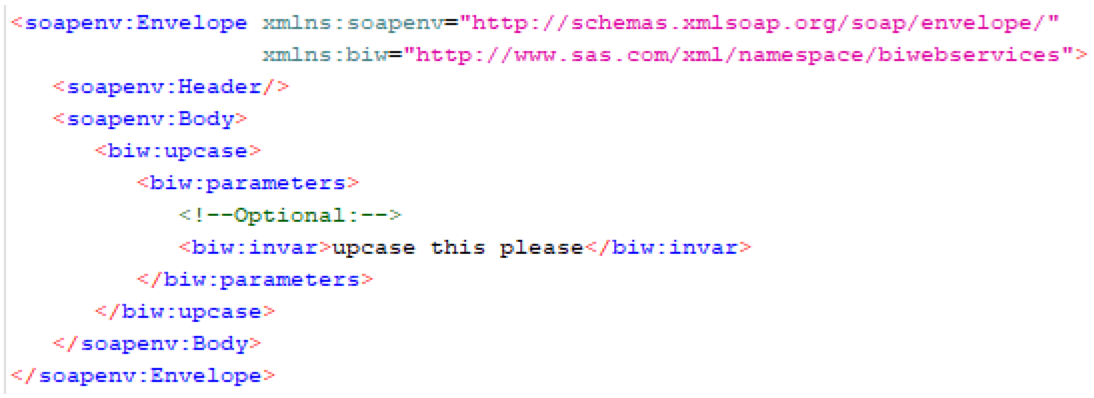
Figure 6 - The Parameters Tab of the Stored Process Properties

Calling as a SOAP Service

Using SoapUI, we can call the new “upcase” service. The WSDL is accessed as before, by appending the metadata path to the stored process to the SASBIWS/services/ URL, along with the “?wsdl” extension:

http://hostname:port/SASBIWS/services/Shared%Data/Stored%20Processes/upcase?wsdl

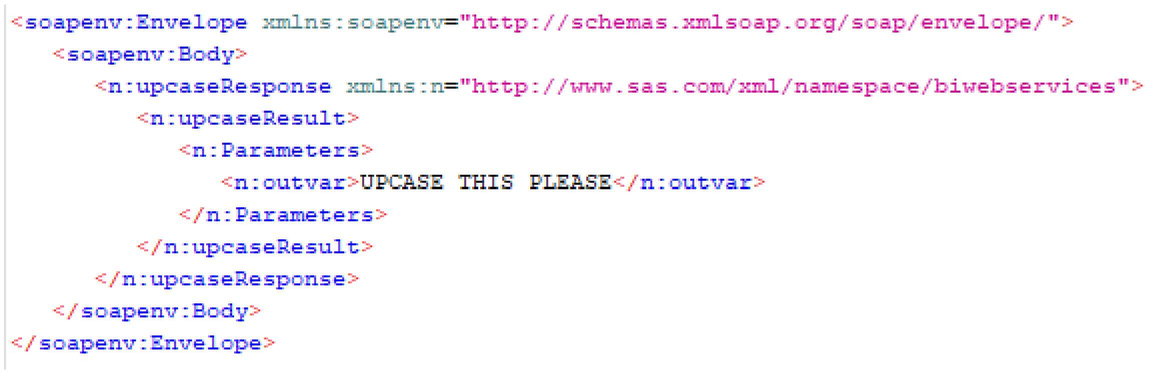
The generated request, with input provided, is shown in Figure 7.



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:biw="http://www.sas.com/xml/namespace/biwebservices">
   <soapenv:Header/>
   <soapenv:Body>
      <biw:upcase>
         <biw:parameters>
            <!--Optional:-->
            <biw:invar>upcase this please</biw:invar>
         </biw:parameters>
      </biw:upcase>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 7 – SOAP Request to the "upcase" Service

The response is shown in Figure 8.



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Body>
      <n:upcaseResponse xmlns:n="http://www.sas.com/xml/namespace/biwebservices">
         <n:upcaseResult>
            <n:Parameters>
               <n:outvar>UPCASE THIS PLEASE</n:outvar>
            </n:Parameters>
         </n:upcaseResult>
      </n:upcaseResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 8 - SOAP Response from the "upcase" Service

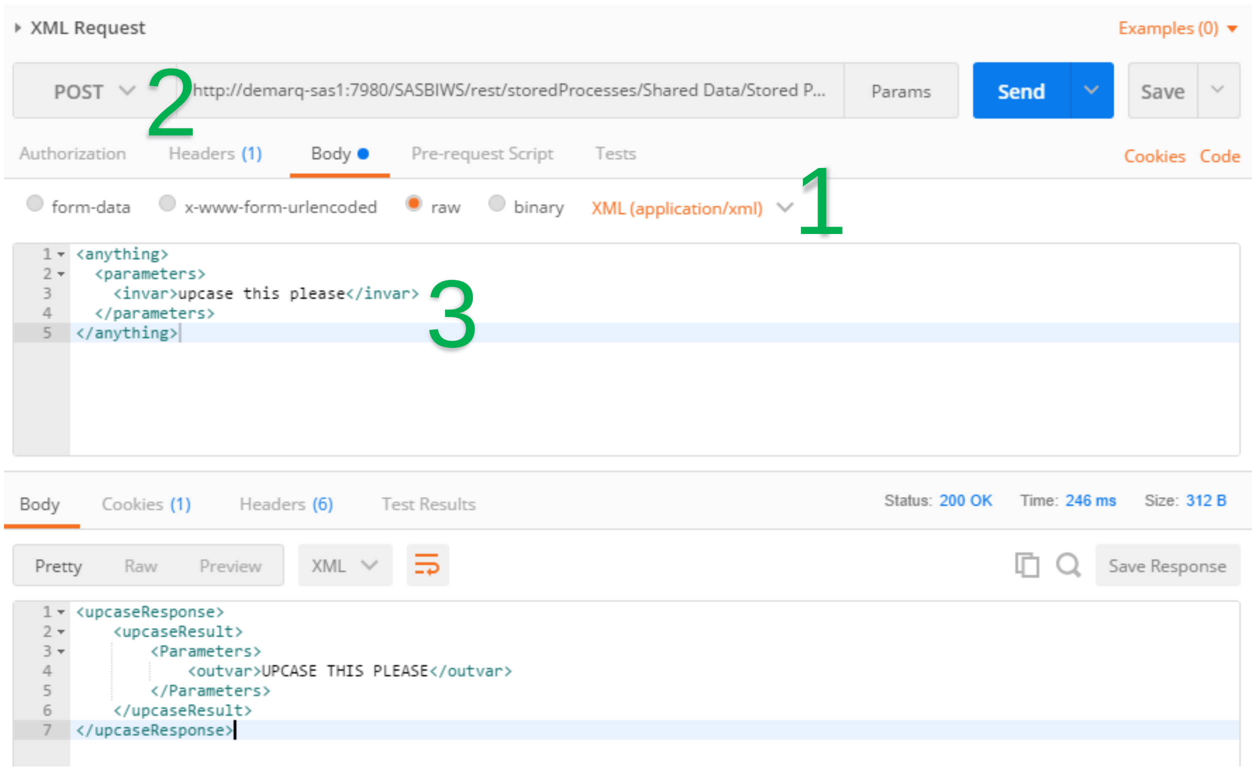
Calling as a RESTful Service Using XML

We can call the same service using the REST protocol and XML. The URL stub for XML calls to RESTful services is http://hostname:port/SASBIWS/rest/storedProcesses/. The metadata path to the stored process is appended, as for SOAP, to become:

http://hostname:port/SASBIWS/rest/storedProcesses/Shared%Data/Stored%20Processes/upcase

To make example calls to RESTful services a suitable REST client is required. The following screenshots use Postman, a free client at the time of writing.

The screenshot in Figure 9 shows a call to the upcase stored process made via the REST interface.



```
<anything>
  <parameters>
    <invar>upcase this please</invar>
  </parameters>
</anything>
```

```
<upcaseResponse>
    <upcaseResult>
        <Parameters>
            <outvar>UPCASE THIS PLEASE</outvar>
        </Parameters>
    </upcaseResult>
</upcaseResponse>
```

Figure 9 - RESTful Request to "upcase" Service Using XML

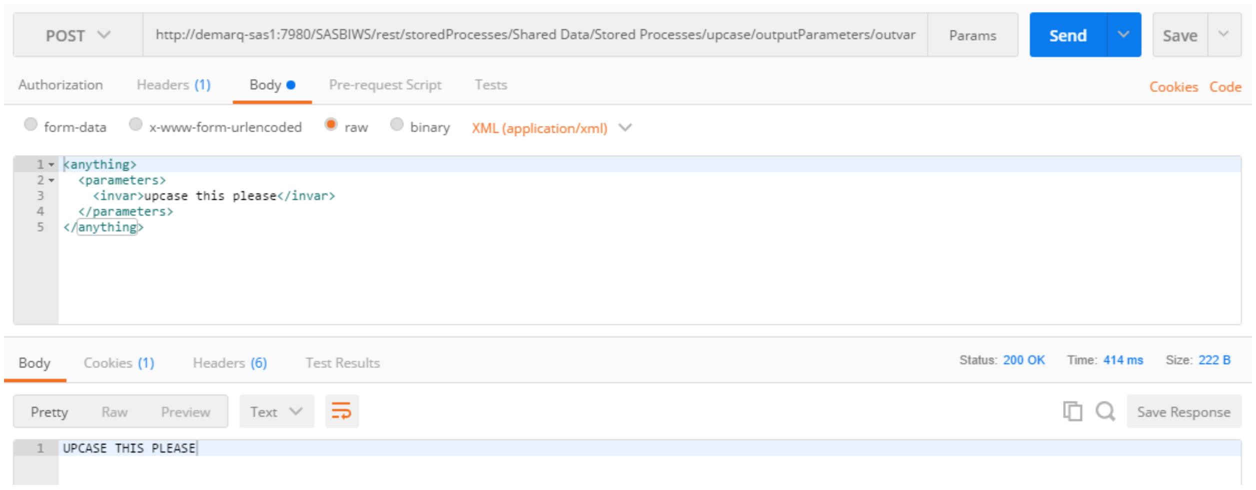
There are a few important settings:

1. The HTTP header value for “Content Type” must be set to “application/xml”
2. SAS uses two HTTP verbs for RESTful calls. Stored processes that require no input use GET, whilst POST is used to when parameters or stream input are required.
3. The parameter must be provided matching the XML format shown in Figure 9. This can be derived from the equivalent SOAP request; remove all SOAP elements and references to the namespace (the part before the colon in opening and closing XML tags).

Alternatively, you can access output parameters directly, without the verbose XML structure, by appending the target resource to the URL in the format outputParameter/parameterName. For example:

http://hostname:port/SASBIWS/rest/storedProcesses/Shared%Data/Stored%20Processes/upcase/outputParameters/outvar

The screenshot in


POST http://demarq-sas1:7980/SASBIWS/rest/storedProcesses/Shared Data/Stored Processes/upcase/outputParameters/outvar

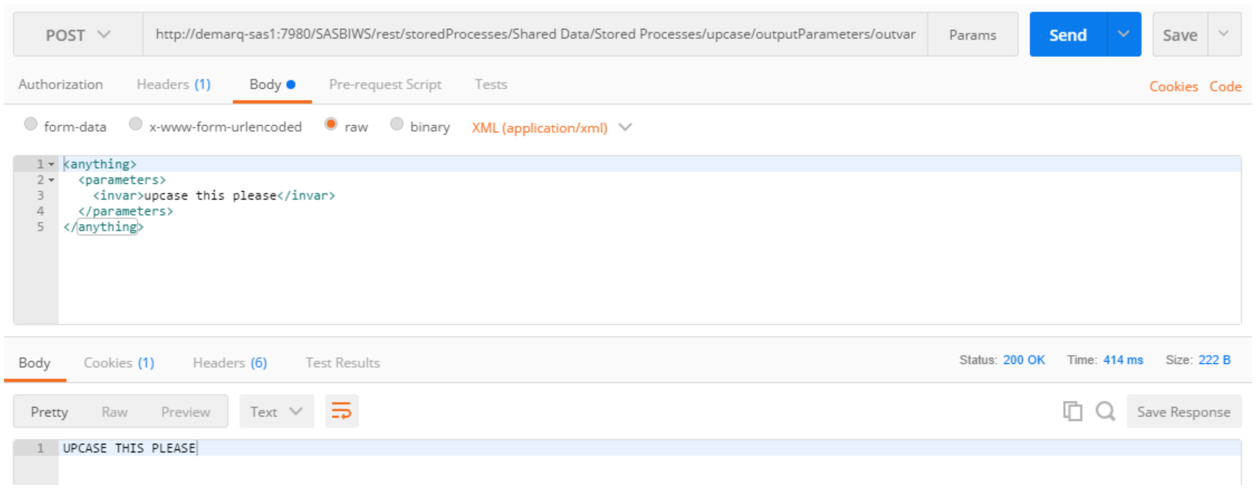
Body: raw, XML (application/xml)

```
<anything>
  <parameters>
    <invar>upcase this please</invar>
  </parameters>
</anything>
```

Status: 200 OK Time: 414 ms Size: 222 B

```
UPCASE THIS PLEASE
```

Figure 10 shows a call made in this way to the upcase stored process. Note that only the value of the parameter is returned.


POST http://demarq-sas1:7980/SASBIWS/rest/storedProcesses/Shared Data/Stored Processes/upcase/outputParameters/outvar

Body: raw, XML (application/xml)

```
<anything>
  <parameters>
    <invar>upcase this please</invar>
  </parameters>
</anything>
```

Status: 200 OK Time: 414 ms Size: 222 B

```
UPCASE THIS PLEASE
```

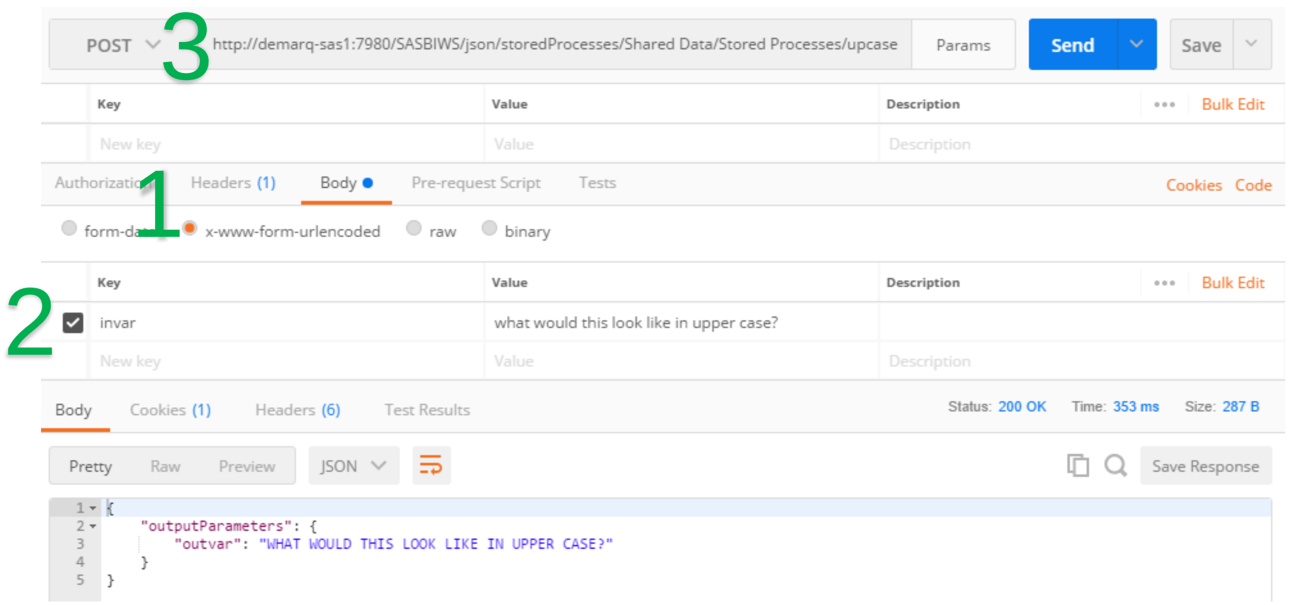
Figure 10 - Accessing Output Parameters Directly Using the REST Interface

Calling as a RESTful Service Using JSON

JSON is also supported as an output format. The JSON endpoint is constructed using the URL stub http://hostname:port/SASBIWS/json/storedProcesses/. The metadata path to the stored process is appended as for SOAP and RESTful XML calls. This makes the URL for our sample process:

http://hostname:port/SASBIWS/json/storedProcesses/Shared%Data/Stored%20Processes/upcase

The screenshot in Figure 11 shows a call to the upcase stored process, using the JSON REST interface.



POST http://demarq-sas1:7980/SASBIWS/json/storedProcesses/Shared Data/Stored Processes/upcase

Body: [✓ x-www-form-urlencoded]

✓	Key	Value	Description
✓	invar	what would this look like in upper case?	

Status: 200 OK Time: 353 ms Size: 287 B

```
{
    "outputParameters": {
        "outvar": "WHAT WOULD THIS LOOK LIKE IN UPPER CASE?"
    }
}
```

Figure 11 - Call to the "upcase" Service Using JSON and the REST Interface

1. The Content-Type HTTP header must be set to application/x-www-form-urlencoded
2. Input parameters can be provided in the body as name value pairs, or passed via the query string (appended to the URL)
3. The HTTP verb/method must be set to POST if input is required, GET if not

It is not possible to access output parameters directly using the JSON interface.

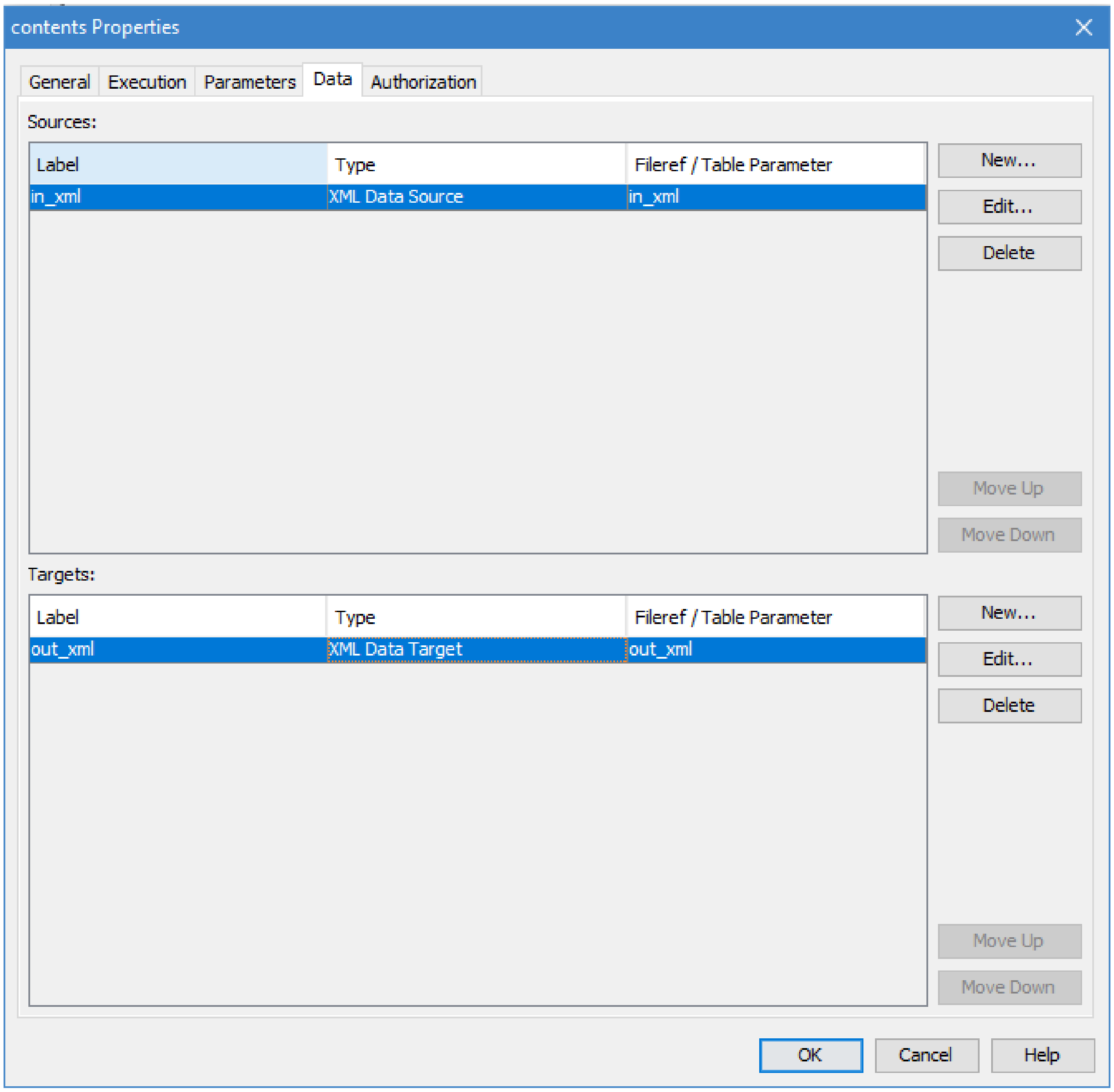
Using Tables

As well as parameters, tables can be passed to web services. Let's define a new example, the "contents" service. The SAS code for the service is shown below:

```
libname in_tab XMLv2 fileref=in_xml;

proc contents data=in_tab.heros out=work.contents;
run;

libname out XMLv2 fileref=out_xml;

data out.heros;
  set work.contents;
run;
```

A XMLv2 libname engine library is assigned, reading from the "in_xml" fileref. The CONTENTS procedure is run on the "heros" table read from this libname, and an output data set is created. The output table is written back to an XML libname using a data step, referencing the "out_xml" fileref.

The stored process properties "Data" tab is shown in Figure 12. The input and output filerefs are defined to correspond to those used in the code. Both are set as type "XML Data Source".



Label	Type	Fileref / Table Parameter
in_xml	XML Data Source	in_xml

Label	Type	Fileref / Table Parameter
out_xml	XML Data Target	out_xml

Figure 12 - Properties of the "contents" Stored Process

Note that the code does not explicitly create the filerefs; there are no FILENAME statements. The references are assigned automatically if referred to in the code.

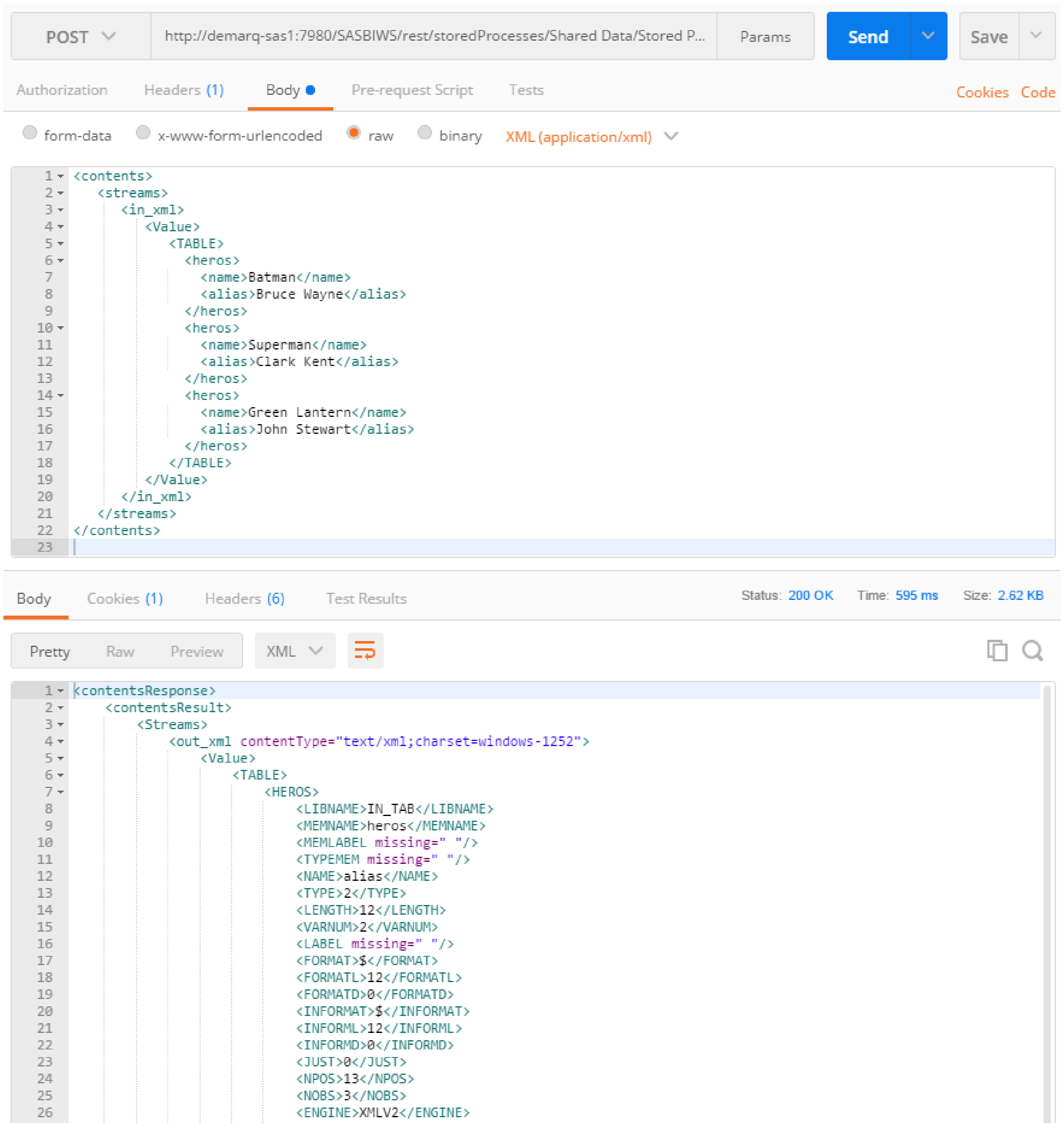
The request format is slightly different when providing tables as input. A valid RESTful request for the “contents” service is shown below:

```
<contents>
   <streams>
      <in_xml>
         <Value>
                <TABLE>
                  <heros>
                      <name>Batman</name>
                      <alias>Bruce Wayne</alias>
                  </heros>
```

```
                    <heros>
                        <name>Superman</name>
                        <alias>Clark Kent</alias>
                    </heros>
                    <heros>
                        <name>Green Lantern</name>
                        <alias>John Stewart</alias>
                    </heros>
                  </TABLE>
            </Value>
         </in_xml>
      </streams>
   </contents>
```

As when using input parameters, you can shortcut the request creation process by generating a SOAP request automatically from the WSDL using a tool such as SoapUI, and then modifying it for a RESTful call.

The response to this request is shown as a screenshot in Figure 13.



http://demarq-sas1:7980/SASBIWS/rest/storedProcesses/Shared Data/Stored P...

Body: raw, XML (application/xml)

```
<contents>
   <streams>
      <in_xml>
         <Value>
            <TABLE>
              <heros>
                <name>Batman</name>
                <alias>Bruce Wayne</alias>
              </heros>
              <heros>
                <name>Superman</name>
                <alias>Clark Kent</alias>
              </heros>
              <heros>
                <name>Green Lantern</name>
                <alias>John Stewart</alias>
              </heros>
            </TABLE>
         </Value>
      </in_xml>
   </streams>
</contents>
```

Status: 200 OK   Time: 595 ms   Size: 2.62 KB

```
<contentsResponse>
    <contentsResult>
        <Streams>
            <out_xml contentType="text/xml;charset=windows-1252">
                <Value>
                    <TABLE>
                        <HEROS>
                            <LIBNAME>IN_TAB</LIBNAME>
                            <MEMNAME>heros</MEMNAME>
                            <MEMLABEL missing=" "/>
                            <TYPEMEM missing=" "/>
                            <NAME>alias</NAME>
                            <TYPE>2</TYPE>
                            <LENGTH>12</LENGTH>
                            <VARNUM>2</VARNUM>
                            <LABEL missing=" "/>
                            <FORMAT>$</FORMAT>
                            <FORMATL>12</FORMATL>
                            <FORMATD>0</FORMATD>
                            <INFORMAT>$</INFORMAT>
                            <INFORML>12</INFORML>
                            <INFORMD>0</INFORMD>
                            <JUST>0</JUST>
                            <NPOS>13</NPOS>
                            <NOBS>3</NOBS>
                            <ENGINE>XMLV2</ENGINE>
```

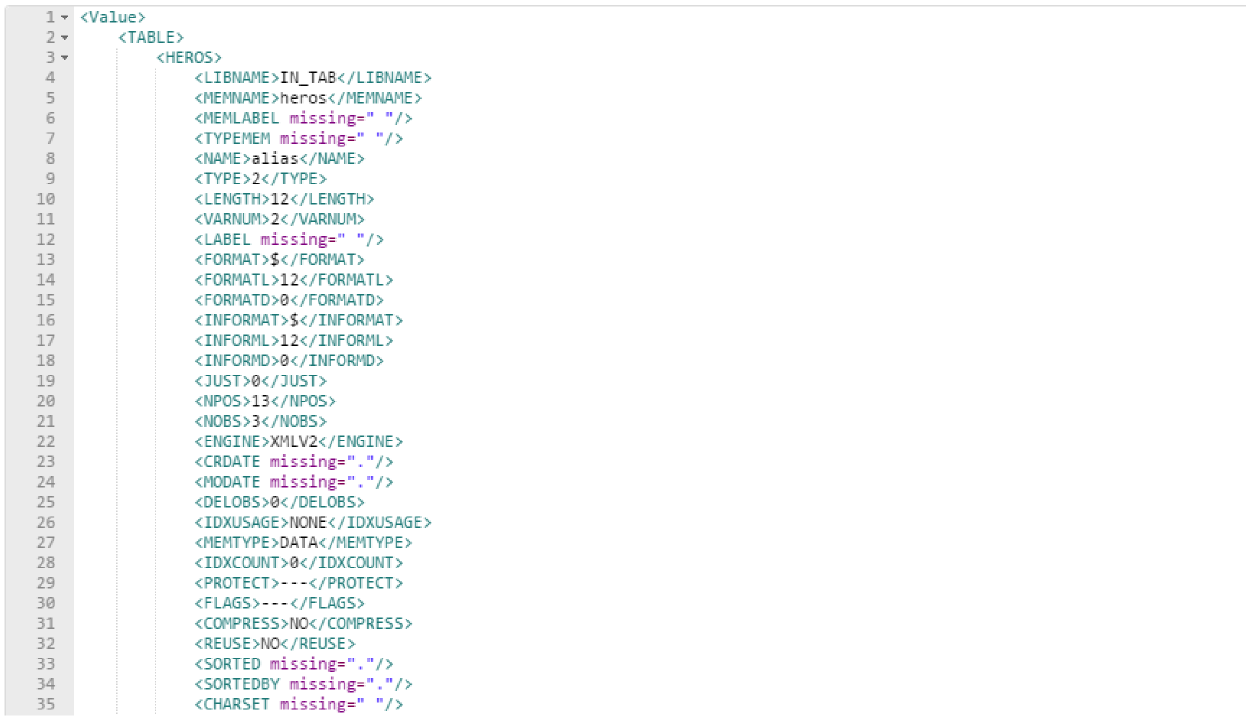
Figure 13 - Response from the "contents" Service

More complicated input and output XML can be read or generated using SAS XML maps. It is important to note that the output written to a fileref defined as an “XML Data Target” in the stored process properties must be valid XML, however it is generated, otherwise an error is returned when the service is called.

As is the case with parameters, it is possible to access the output structure directly (that is, only the output written to the fileref, without the verbose response tags around it) by altering the endpoint that is called. Data targets are accessed by appending “dataTarget/*target_fileref*” to the URL. The URL to access the “out_xml” fileref of the “contents” service becomes:

http://hostname:port/SASBIWS/rest/storedProcesses/Shared%Data/Stored%20Processes/contents/dataTargets/out_xml

As example of the response format is shown in Figure 14.



```
<Value>
    <TABLE>
        <HEROS>
            <LIBNAME>IN_TAB</LIBNAME>
            <MEMNAME>heros</MEMNAME>
            <MEMLABEL missing=" "/>
            <TYPEMEM missing=" "/>
            <NAME>alias</NAME>
            <TYPE>2</TYPE>
            <LENGTH>12</LENGTH>
            <VARNUM>2</VARNUM>
            <LABEL missing=" "/>
            <FORMAT>$</FORMAT>
            <FORMATL>12</FORMATL>
            <FORMATD>0</FORMATD>
            <INFORMAT>$</INFORMAT>
            <INFORML>12</INFORML>
            <INFORMD>0</INFORMD>
            <JUST>0</JUST>
            <NPOS>13</NPOS>
            <NOBS>3</NOBS>
            <ENGINE>XMLV2</ENGINE>
            <CRDATE missing="."/>
            <MODATE missing="."/>
            <DELOBS>0</DELOBS>
            <IDXUSAGE>NONE</IDXUSAGE>
            <MEMTYPE>DATA</MEMTYPE>
            <IDXCOUNT>0</IDXCOUNT>
            <PROTECT>---</PROTECT>
            <FLAGS>---</FLAGS>
            <COMPRESS>NO</COMPRESS>
            <REUSE>NO</REUSE>
            <SORTED missing="."/>
            <SORTEDBY missing="."/>
            <CHARSET missing=" "/>
```

Figure 14 - Accessing the Data Target Directly

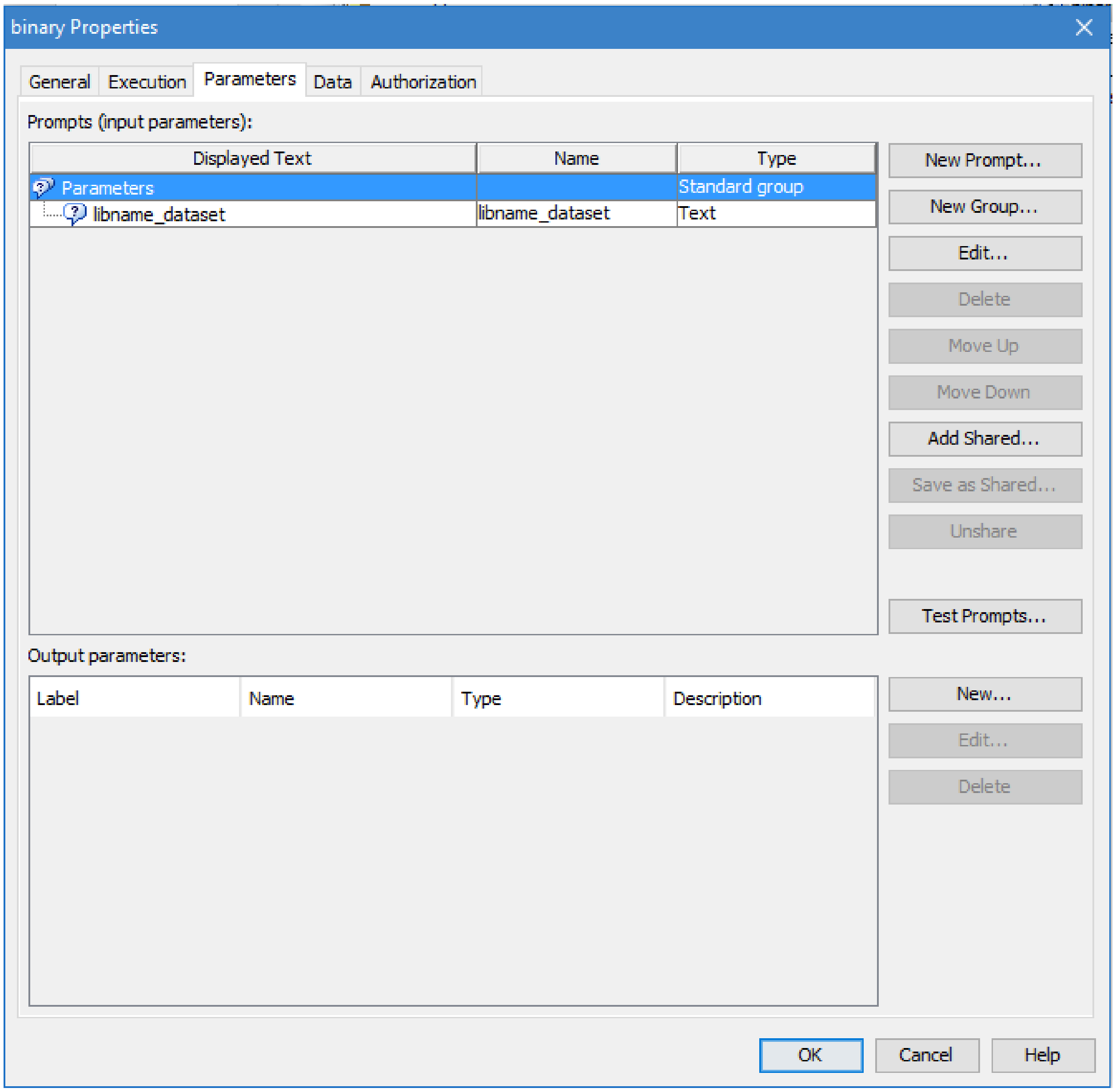
Accessing Binary Output

SAS can produce a variety of binary output (PDF files, images, etc.) and these can also be returned from a web service. Let's set up a simple example. The SAS code below forms the basis for the "binary" stored process:

```
ods pdf body=out_file;

proc print data=&libname_dataset.;
run;

ods pdf close;
```

An ODS statement opens an PDF ODS destination, with the body set to the "out_file" fileref. The PRINT procedure produces a simple listing, the two-level name of the data set to print is provided as a macro variable "libname_dataset".

The stored process properties "Parameters" tab in metadata is shown in Figure 15, the "Data" tab in Figure 16.



binary Properties

General | Execution | Parameters | Data | Authorization

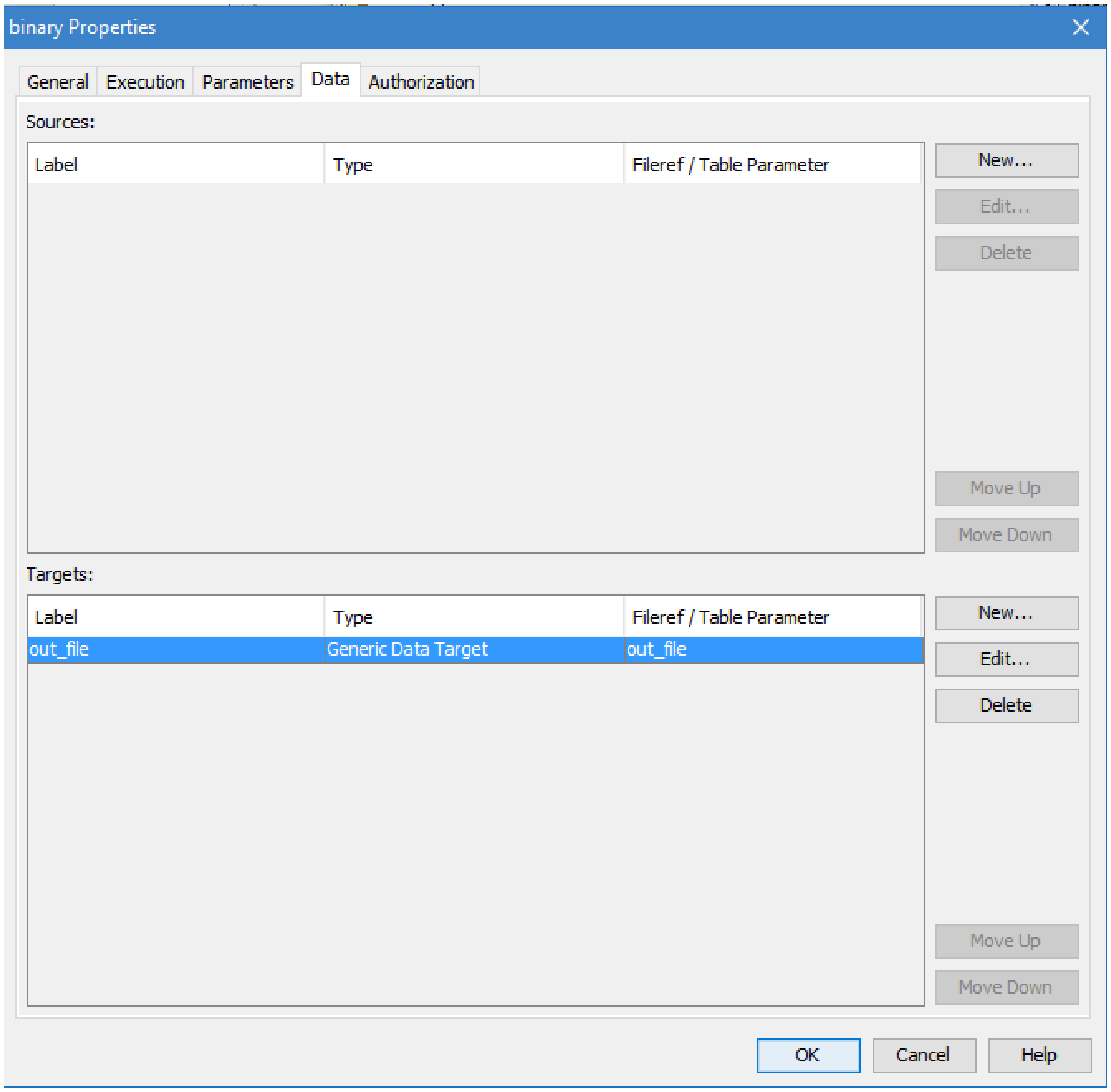
Prompts (input parameters):

Displayed Text	Name	Type
Parameters		Standard group
libname_dataset	libname_dataset	Text

Output parameters:

Label	Name	Type	Description

Figure 15 - Parameters Tab of the "binary" Stored Process



Sources:

Label	Type	Fileref / Table Parameter

Targets:

Label	Type	Fileref / Table Parameter
out_file	Generic Data Target	out_file

Figure 16 - Data Tab of the "binary" Stored Process

A single input parameter is defined for the macro variable “libname_dataset”, along with a single data target, “out_file”, mapped to the fileref used in the code, and set as type “Generic Data Target”.

The RESTful request is shown below, passing the value “sashelp.class” as the value for the input parameter.

```
<binary>
    <parameters>
          <libname_dataset>sashelp.class</libname_dataset>
    </parameters>
</binary>
```

Calling the endpoint http://hostname:port/SASBIWS/rest/storedProcesses/Shared%Data/Stored%20Processes/binary gives the following response (note that the “value” has been truncated in order to fit the page):

```
<binaryResponse>
    <binaryResult>
        <Streams>
            <out_file contentType="application/pdf">
                <Value>JVBERi0xLj...<snip>...AwIFINCj4+</Value>
            </out_file>
        </Streams>
    </binaryResult>
</binaryResponse>
```

The “out_file” in the response contains the PDF output, Base64 encoded to plain text. This can be decoded using a Base64 decoder to obtain the binary file.

The file can be accessed directly, like when accessing parameters and other data targets, by appending “dataTargets/*fileref*” OR “streams/*fileref*” to the URL. The endpoint for this example becomes:

http://hostname:port/SASBIWS/rest/storedProcesses/Shared%Data/Stored%20Processes/binary/streams/out_file

The resulting behavior depends largely on the client you are using. For an ODS output, SAS sets the appropriate Content Type HTTP header which will allow most clients to prompt you to download the file or open it with an appropriate application.

For files not generated using ODS (read in with a binary FILENAME statement, for example) then it is not possible to set the Content Type header to match the file type from SAS. This could cause a problem if calling the services from a client such as Postman but does not cause an issue for most programming languages that can write a binary file stream to disk. SAS is such a language.

Below is sample code for another stored process, “graph”, that reads a PNG file from the filesystem and returns it to the fileref “out_file”:

```
data _NULL_;
   infile "path/gchart.png" recfm=f;
   file out_file lrecl=1000000;
   input char $char1. @@;
   put char $char1. @@ ;
run;
```

The stored process metadata is set up as for the “binary” service.

SAS code used to call the “graph” service is shown below:

```
filename out "path\test.png";

proc http out=out
url="http://hostname:port/SASBIWS/rest/storedProcesses/Shared%20Data/Stored
%20Processes/graph/dataTargets/out_file"
  method="get";
run;
```

The HTTP procedure is used, as in the example on page 2. The out option is set to the fileref “out”, and the endpoint is called using the “get” HTTP method. This results in the binary file returned from the web service being written to the fileref.

THE FUTURE OF WEB SERVICES AND SAS

SAS’ commitment to the web services ecosystem that has been explored in this paper has been underlined by the release of SAS Viya. Viya presents a full RESTful API for executing CAS actions (loading data, perform analytics etc.), controlling sessions and monitoring the state of the system. These are the building blocks to fully integrate SAS capabilities with any other language or system that has an HTTP library, which is just about anything!

CONCLUSION

Web service based API's are rapidly increasing in popularity; this trend is set to continue. SAS is well placed to take advantage of the data and integration opportunities this provides with the HTTP and SOAP procedures, and to take part in it by publishing Stored Processes as web services through the SAS BI Web Services application. SAS Viya builds on this with a full REST API that opens up SAS capabilities to a vast array of third party languages and platforms.

REFERENCES

World Wide Web Consortium. (2004). *Web Services Glossary*. Retrieved from https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice

RECOMMENDED READING

- *Base SAS® Procedures Guide*
- *SAS® 9.4 BI Web Services: Developer's Guide*
- *CAS REST API*

CONTACT INFORMATION

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

Richard Carey
Demarq
richard.carey@demarq.uk
www.demarq.uk