Paper SAS1927-2015

REST at Ease with SAS®: How to Use SAS to Get Your REST

Joseph Henry, SAS Institute Inc.

ABSTRACT

Representational State Transfer (REST) is being used across the industry for designing networked applications to provide lightweight and powerful alternatives to web services such as SOAP and Web Services Description Language (WSDL). Since REST is based entirely around HTTP, SAS® provides everything you need to make REST calls and to process structured and unstructured data alike. Learn how the HTTP procedure and other SAS language features provide everything you need to simply and securely use REST.

INTRODUCTION

REST provides a convenient way to develop and access a web service, but it can be overwhelming to the uninitiated. Simply put, calling a REST API is just sending and receiving data over HTTP. The DATA step provides all that you need for formatting and parsing data, while the HTTP procedure provides the means to send the data to the web server. This paper will teach you how to use the SAS DATA step in conjunction with PROC HTTP to properly construct, execute, and handle most REST APIs.

KNOWING YOUR API

Before you begin developing an application that uses a web services API, gather important information about the API that you will be using. The example in this paper will use the Google URL Shortener API. This API can shorten a long URL, such as the URL http://support.sas.com/events/sasglobalforum/2014/, to http://goo.gl/mTxdf2, or it can do the reverse. The developer documentation can be found at https://developers.google.com/url-shortener/.

There are a few simple questions that you need to ask that make using REST APIs easier.

1. **What is the base URL?**

   You can think of the URL to a REST API as you would a function in C or Java, where the URL is the function and the query string contains the arguments. It is important to determine what the base URL is because that is the part that will not change.

   Determining the base URL is an important first step, since this is the constant that everything else is built upon. In this example the base URL is https://www.googleapis.com/urlshortener/v1/url.

2. **What HTTP method is used?**

   Knowing what HTTP method is used for execution of the API is a crucial step in determining how to construct your query. A very important aspect of the HTTP method is that it determines how input arguments are sent. The Google URL Shortener API provides two actions with only one base URL. This is accomplished by having an HTTP POST perform a different action than an HTTP GET.

   When a POST is sent, the API behaves as a URL shortener, taking a base long URL and making it shorter. When a GET is performed, the API behaves the opposite way, and takes a shortened URL and returns the long version.

   The major difference between a POST and a GET, is that a POST sends parameters in the body of the HTTP request, whereas a GET sends the parameters as a part of the URL.

3. **What are the parameters?**

   The API for the Google URL Shortener has one required parameter for each action. The URL

Shortener requires the argument *longURL* and the URL Expander requires the argument *shortURL*.

Both actions have the optional argument *key*, which is the API key that is provided by Google for registered web applications.

4. **What format is the response in?**

   The ultimate goal is to use the data that is returned from the API call, but in order to do this effectively you must know what format the return data is in. Typical return formats are XML and JSON, but the response could just as easily be plain text. The response format is important because this determines how you use the returned data. In this example, the return format is JSON.

BUILDING THE REQUEST

Once you know the required information about the API that you want to use, you can start to write the SAS code that is required to execute a request.

The example presented here is a sample program that calls a REST API in order to shorten a URL, and then the example subsequently uses the returned short URL to make another call to expand it. Both POST and GET HTTP requests are used along with appropriate error checking. The full example program can be found in Appendix A.

CONSTRUCTING THE REQUEST

The first example uses an HTTP POST, which in this API causes the given URL to be shortened. To begin, you must build the data that will be posted. Many webservers expect the data given to be exact, so that means it is very important to format your input data precisely. Avoid any unwanted padding or end of lines by using `recfm=f lrecl=1` in the FILENAME statement:

```
filename pdata TEMP;

data _null_;
 file pdata recfm=f lrecl=1;
 put "{";
 put """longUrl"": ""http://www.sas.com""";
 put "}";
run;
```

The API that is called in this example has only one required argument, which is *longUrl*, but the code can easily be modified to include optional arguments, such as *key*:

```
filename pdata TEMP;

data _null_;
 file pdata recfm=f lrecl=1;
 put "{";
 put """longUrl"": ""http://www.sas.com""";
 put ",";
 put """key"": ""&API_KEY""";
 put "}";
run;
```

Now that the input data has been created and properly formatted, you can call the API using the HTTP procedure. It is a good idea to include a FILEREF for the response headers, which provides a good means of error checking.

```
filename resp TEMP;
```

```
filename hdrs TEMP;


proc http
 url="&base_url" /* The base URL */
 method="POST"   /* HTTP method used for request */
 in=pdata        /* Input data */
 ct="application/JSON" /* The type of input data that is being sent (Content-
Type) */
 /* headerin=ihdrs */
 out=resp       /* Response from server */
 headerout=hdrs; /* Response headers */
run;
```

After the HTTP request has been made, it is time to handle the response by parsing the returned HTTP headers and body.

READING THE RESPONSE

It does not do you any good if you cannot extract the needed data out of your response, so the next steps are to perform response parsing and error checking.

RESPONSE PARSING

The response is broken up into two pieces, the header and the body. Parsing the header is not completely necessary, but it does provide quite a bit of information, including the HTTP response code and possibly the size of the response body.

Header

A complete understanding of the HTTP protocol is not necessary, but you do need to know a few things about how a header block is formatted before you can parse it. First, take into account that header lines always end with a carriage return and a line feed (CRLF). If you are working on a UNIX system, this is especially important since the default end-of-line character (EOL) is only a line feed (LF). Since the DATA step defaults to using the system EOL character, it is a good idea to add `termstr=CRLF` to the INFILE statement.

Next, you should know that the first header line is special and is known as the status line. The status line takes the form:

HTTP-Version Status-Code Reason-Phrase CRLF

All other headers take the form:

Field-Name ":" Field-Value CRLF

The most important piece of the status line is the Status-Code. The Status-Code tells you how the web server understood and responded to the request.

The following code takes the response header from the execution of the HTTP procedure and parses out the Status-Code by scanning for the second word after a white space. The code also looks for the headers *Content-Length* and *Transfer-Encoding*:

```
data _null_;
  infile hdrs termstr=CRLF;
  if _n_ = 1 then
   do;
    input @;
    length status_code $ 3;
    status_code = scan(_infile_,2,' ');
```

```
   call symput('hcode',status_code);
   put "&hcode";
  end;
else
 do;
  input @;
  name = scan(_infile_,1,':');
  value = scan(_infile_,2,':');

  if name = "Content-Length" then
   call symput('Content_Length',value);
  else if name = "Transfer-Encoding" then
   call symput('Transfer_Encoding',value);
 end;
run;
```

This code works very well and is easily extendable to parse out any other headers. For example, you could simply add a search for the *Content-Type* header such as the following:

```
…
else if name = "Content-Type" then
      call symput('Content_Type',value);
…
```

There are, however, a few situations where this does not work, such as when a redirect is automatically taken or when multipart authentication is used. These instances lead to having more than one header block in the FILEREF. You are only concerned with the last header block, so to handle this situation, extend the code by adding an extra DATA step to search for the last header block, similar to the following code:

```
%global startline;
data _null_;
 infile hdrs termstr=CRLF end=eof;
 retain lastloc 0;
 input;
 if _infile_=:"HTTP/" then lastloc=_n_;
 if eof then call symputx('startline',max(1,lastloc));
run;

%put &startline;

data _null_;
 infile hdrs termstr=CRLF length=c firstobs=&startline.;
 if _n_ = 1 then
…
```

This code guarantees that you will be searching through the last header block that was received.

Error Checking

It is a good idea to check your return code to make sure that your request was executed properly. In the current example, you are expecting a 200 return status. A handy macro to have for checking for an expected status code is shown in the following code:

```
%macro check_return(code,expected);
%if &code ne &expected %then %do;
 %put ERROR: Expected &expected, but received &code;
 %abort;
```

```
%end;
%mend;
```

You can now add this to make sure that a 200 response was received as follows:

```
%check_return(&hcode,200);
```

Body Parsing

The body of your response can be in many different formats, such as plain text, XML, HTML, JSON, and so on. You will need to handle each format appropriately. Most of the time you know what format your response is going to be in ahead of time. For example, the API that is being called in this example returns JSON formatted data, which is indicated in the API documentation. Reading the *Content-Type* header is another way to determine the format of your response body.

The example given should return a JSON formatted response that looks similar to this output:

```
{
 "kind": "urlshortener#url",
 "id": "http://goo.gl/5drFH",
 "longUrl": "http://www.sas.com/"
}
```

To get the shortened URL, you need to extract the value for the key *id* from the body. The first way to do this would be a simple DATA step. The technique used in this paper is described in detail at http://support.sas.com/resources/papers/proceedings13/296-2013.pdf. An example using this technique is shown here:

```
data _null_;
 infile resp lrecl=32000 truncover scanover;
 input @'"id": ' surl $255.;
 shorturl = dequote(surl);
 put shorturl;
 call symput('shorturl',shorturl);
run;
```

Notice that the logical record length is set to a very large value of 32000. This typically works, but if the response is larger than 32 kilobytes, you might encounter some problems. The best way to deal with this is to set the logical record length to the exact size of the file. There are two ways that you can get the size of the response.

Content-Length Header

The easiest way to find the size of the response body is to search for the header *Content-Length* in the response header. This will give you the exact size in bytes of the response body. You can use the header parsing code provided to retrieve the *Content-Length* header value and set `lrecl` to that value:

```
infile resp lrecl=&Content_Length truncover scanover;
```

Manual Count

Unfortunately, the *Content-Length* header is not always sent, as is the case where the *Transfer-Encoding* is broken into blocks. In this situation, you can obtain the size of the file manually, but this requires an extra DATA step:

```
data _null_;
 infile resp lrecl=1 recfm=f end=eof;
 input;
 if eof then call symputx('FILESIZE',_n_);
run;
```

```
data _null_;
 infile resp lrecl=&FILESIZE truncover scanover;
…
```

EXPANDING

You have now seen how to construct a request that took a long URL and converted it into a short URL using an HTTP POST and formatted input data. This example can be expanded upon by using the URL expander method.

As stated previously, the REST API that is used in this example behaves differently depending on whether it was called using an HTTP GET or an HTTP POST. The next piece of this example uses the HTTP GET method that takes a short URL and expands it.

The main difference between the two versions of the API call is that with a GET, all the arguments to the API are sent within the URL instead of in a formatted request body. In order to be able to pass multiple arguments inside of a URL, the URL must be formatted in a certain way, which means the addition of what is known as a query string.

The query string is added to the URL by appending a "?" onto the URL, followed by the query string. The query string consists of a sequence of name-value pairs where each name and value are separated with an "=". Each name-value pair is then separated with an "&". An example using the argument *ShortUrl* and *key* is shown here:

> Base URL - https://www.googleapis.com/urlshortener/v1/url
>
> Argument 1 – "ShortUrl=http://goo.gl/5drFH"
>
> Argument 2 – "key=[KeyValue]"
>
> URL – "https://www.googleapis.com/urlshortener/v1/url?ShortUrl=http://goo.gl/5drFH&key=<KeyValue>"

Creating the formatted URL can be achieved in your SAS code using the DATA step and the CATS function:

```
%global expand_url;
data _null_;
 expand_url=cats("&base_url","?shortUrl=","&shortUrl");
 call symput('expand_url',expand_url);
run;
```

This URL expander API only requires one argument, and just like the previous example, the optional parameter *key* can be added. Since additional arguments use the "&" as a delimiter, it is a good idea to enclose those strings in single quotation marks so that SAS does not try first to interpret the argument as a macro:

```
data _null_;
 expand_url=cats("&base_url","?shortUrl=","&shortUrl",'&',"key=&API_KEY");
 call symput('expand_url',expand_url);
run;
```

After the URL is built, execution and response parsing is done just as before, except you tell the HTTP procedure to perform a GET instead of a POST:

```
proc http
 url="&expand_url"
 method="GET"
 out=resp
 headerout=hdrs;
run;
```

CONCLUSION

SAS provides all the tools you need to use RESTful web services. The DATA step provides a convenient way to format URLs and input data and to interpret responses. The HTTP procedure provides an easy way to execute the HTTP request that goes along with all REST calls. As long as you know the required details of the API, pay attention to how your URL and input data are formatted, and perform proper checks along the way, you can easily use SAS to use just about any RESTful web service out there.

APENDIX A: SAS CODE

```
/*
Check the returned status code against what is expected
*/
%macro check_return(code,expected);
%if &code ne &expected %then %do;
 %put ERROR: Expected &expected, but received &code;
 %abort;
%end;
%mend;


%let base_url=https://www.googleapis.com/urlshortener/v1/url;
%let API_KEY=AIzaSyAmQfbOaZmuc5_b-4KmIFU1H5dqKaubgMI;

filename pdata TEMP;

%let longUrl=http://www.sas.com;
data _null_;
 file pdata recfm=f lrecl=1; /* This will make sure that there are no
extra bytes at the end of the input file */
 put "{";
 put """longUrl"": ""&longUrl""";
 put ",";
 put """key"": ""&API_KEY""";
 put "}";
run;

filename resp "resp1";
filename hdrs TEMP;


proc http
 url="&base_url" /* The base URL */
 method="POST"   /* HTTP method used for request */
 in=pdata        /* Input data */
 ct="application/JSON" /* The type of input data that is being sent
(Content-Type) */
 out=resp       /* Response from server */
 headerout=hdrs; /* Response headers */
run;

%echofile(resp);
%echofile(hdrs);

%global hcode;

%global startline;
data _null_;
 infile hdrs termstr=CRLF end=eof;
 retain lastloc 0;
 input;
 if _infile_=:"HTTP/" then lastloc=_n_;
 if eof then call symputx('startline',max(1,lastloc));
run;
```

```
   data _null_;
    infile hdrs termstr=CRLF length=c firstobs=&startline.;
    if _n_ = 1 then
     do;
      input @;
      length status_code $ 3;
      status_code = scan(_infile_,2,' ');
      call symputx('hcode',status_code);
      put "&hcode";
     end;
   else
    do;
     input @;
     name = scan(_infile_,1,':');
     value = scan(_infile_,2,':');
     if name = "Content-Length" then call symput('Content_Length',value);
     else if name = 'Transfer-Encoding' then call
symput('Transfer_Encoding',value);
    end;
   run;

   %check_return(&hcode,201);

   /*
   The previous call returns data via the chunked transfer encoding,
therefore a Content-Length header was not received.
   The file size will have to be determined manually.
   */
   data _null_;
    infile resp lrecl=1 recfm=f end=eof;
    input;
    if eof then call symputx('FILESIZE',_n_);
   run;

   /*
   Scan through the response body and look for the key "id" and its
associated value
   */
   data _null_;
    infile resp lrecl=&FILESIZE truncover scanover;
    input @'"id": ' surl $255.;
    shorturl = dequote(surl);
    put shorturl;
    call symput('shortUrl',trim(shorturl));
   run;

   /*
   Now that the short URL has been retrieved, the long URL can be retrieved
using the GET version of the API
   */

   %global expand_url;

   /*
   Build the request URL
   */
   data _null_;
```

```
 expand_url=cats("&base_url","?shortUrl=","&shortUrl");
 call symput('expand_url',expand_url);
run;

/*
Execute the request
*/
proc http
 url="&expand_url"
 method="GET"
 out=resp
 headerout=hdrs;
run;

/*
Find the last header block
*/
data _null_;
 infile hdrs termstr=CRLF end=eof;
 retain lastloc 0;
 input;
 if _infile_=:"HTTP/" then lastloc=_n_;
 if eof then call symputx('startline',max(1,lastloc));
run;

/*
Parse the header storing the status code and a few
header values.
*/
data _null_;
 infile hdrs termstr=CRLF length=c firstobs=&startline.;
 if _n_ = 1 then
  do;
   input @;
   length status_code $ 3;
   status_code = scan(_infile_,2,' ');
   call symputx('hcode',status_code);
   put "&hcode";
  end;
else
 do;
  input @;
  name = scan(_infile_,1,':');
  value = scan(_infile_,2,':');
  if name = "Content-Length" then call symput('Content_Length',value);
  else if name = 'Transfer-Encoding' then call
symput('Transfer_Encoding',value);
 end;
run;

/*
Make sure that 200 was returned
*/
%check_return(&hcode,200);

/*
Scan through the response using the "Content-Length" header value as the
```

```
Logical Record Length and scan for the key "longUrl" and its associated
value.
*/
data _null_;
 infile resp lrecl=&Content_Length truncover scanover;
 input @'"longUrl": ' surl $255.;
 longUrl = dequote(surl);
 call symput('longurl2',longUrl);
run;

%put NOTE: longURL: &longUrl shortened to &shorturl;
%put NOTE: shortURL: &shorturl expanded to &longurl2;
```

REFERENCES

- Choy, Murphy, and Shim, Kyong. 2013. “Efficient extraction of JSON information in SAS® using the SCANOVER function”. *Proceedings of the SAS Global 2013 Conference.* Cary, NC: SAS Institute, Inc. Available http://support.sas.com/resources/papers/proceedings13/296-2013.pdf.

ACKNOWLEDGEMENTS

Many thanks to Rick Langston. Without his in-depth knowledge of the DATA step, none of this would have been possible.

CONTACT INFORMATION

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

Joseph Henry
100 SAS Campus Drive
Cary, NC 27513
SAS Institute, Inc.
Joseph.Henry@sas.com

http://www.sas.com