Paper SAS3232-2019

The ABCs of PROC HTTP

Joseph Henry, SAS Institute Inc., Cary, NC

ABSTRACT

Hypertext Transfer Protocol (HTTP) is the foundation of data communication for the World Wide Web, which has grown tremendously over the past generation. Many applications now exist entirely on the web, using web services that use HTTP for communication. HTTP is not just for browsers since most web services provide an HTTP REST API as a means for the client to access data. Analysts frequently find themselves in a situation where they need to communicate with a web service or application from within a SAS® environment, which is where the HTTP procedure comes in. PROC HTTP makes it possible to communicate with most services, coupling your SAS® system with the web. Like the web, PROC HTTP continues to evolve, gaining features and functionality with every new release of SAS®. This paper will dive into the capabilities found in PROC HTTP allowing you to get the most out of this magnificent procedure.

INTRODUCTION

PROC HTTP is a powerful SAS procedure for creating HTTP requests. HTTP is the underlying protocol used by the World Wide Web, but it is not just for accessing websites anymore. Web-based applications are quickly replacing desktop applications, and HTTP is used for the communication between client and server. PROC HTTP can be used to create simple web requests or communicate with complex web applications and you just need to know how. This paper goes into detail about the features, capabilities, and limitations of PROC HTTP, and which release of SAS® those are associated with. Many of the examples presented will be using the webserver httpbin.org, which is a free HTTP request and response testing service.

GETTING STARTED

The simplest thing to do with PROC HTTP is to read an HTTP resource into a file:

```
filename out TEMP;
filename hdrs TEMP;

proc http
 url="http://httpbin.org/get"
 method="GET"
 out=out
 headerout=hdrs;
 run;
```

This code simply performs an HTTP GET request to the URL and writes the response body to the `out` fileref and any response headers to the `hdrs` file. This syntax is valid in SAS® 9.4 and above, but a lot has changed since SAS® 9.4 release in July 2013.

BROWSER LIKE DEFAULTS

Starting with SAS 9.4m3, certain intuitive defaults are set for requests.

If no method is set AND there is no input given, such as not uploading any data, the default request method will be a **GET** (in SAS 9.3 – 9.4m2 the default was always **POST**).
If a URL scheme is not specified, http:// will be automatically appended, meaning that unless you specifically need https, you do not need to enter the scheme, making PROC HTTP behave more like how a web browser behaves.

Given this, the code above could be rewritten as such:

```
filename out TEMP;
filename hdrs TEMP;

proc http
 url="httpbin.org/get"
 out=out
 headerout=hdrs;
 run;
```

HTTP RESPONSE

Each HTTP request has a subsequent HTTP response. The headers that are received in the response contains information about the response. In the above code, the headers are written to the fileref hdrs and result in the following:

```
< HTTP/1.1 200 OK
< Content-Type: application/json
< Content-Length: 194
< Connection: keep-alive
```

The first line of the response header is called the Status-Line and consists of the protocol version followed by a status code and a phrase describing the status code. The status code is important because it can let you know if your request succeeded or not. Prior to SAS® 9.4m5, the way you extract the status code from the headers would be:

```
data _null_;
 infile hdrs scanover truncover;
 input @'HTTP/1.1' code 4. message $255.;
 call symputx('status_code',code,'g');
 call symputx('status_message',trim(message),'g');
run;
```

After this code has executed, the macro variables status_code and status_message would contain 200 and OK respectively.

SAS 9.4m5 simplifies this tremendously by automatically storing the status code and status phrase in the macro variables SYS_PROCHTTP_STATUS_CODE and SYS_PROCHTTP_STATUS_PHRASE respectively. This eliminates the need to run a DATA step to extract the status code and phrase. You can then use something like what is shown below to check for errors:

```
%if &SYS_PROCHTTP_STATUS_CODE. ne 200 %then %do;
 %put ERROR: Expected 200, but received &SYS_PROCHTTP_STATUS_CODE.;
 %abort;
%end;
```

HTTP REQUEST HEADERS

It is often necessary to add one or more headers to the request. Prior to SAS 9.4m3, the code would have been submitted as following:

```
filename headers TEMP;

data _null_;
 file headers;
 put "X-Header-Name: value of the header";
 put "X-Header-Name2: Another value";
run;

proc http
 method="GET"
 url="http://httpbin.org/headers"
 headerin=headers;
run;
```

HTTP headers consist of a field name followed by a colon (:), an optional white space, and the field value. Using the code above, each line in the output file must be an acceptable HTTP header, or errors occur.

SAS 9.4m3 added an easy way add headers to the request with the HEADERS statement. The HEADERS statement takes string pairs, which are sent on the request as HTTP headers. This eliminates the need for an extra DATA step as well as an additional input file. An example of using the headers statement is shown below:

```
proc http
  url="httpbin.org/headers";
  headers "Accept"="application/json";
 run;
```

The resulting output is the following:

```
GET /headers HTTP/1.1
User-Agent: SAS/9
Host: httpbin.org
Connection: Keep-Alive
Accept: application/json
```

The headers statement also allows you to override any of the default headers that PROC HTTP sends. Prior to this, the only default header that could be overridden was "Content-Type" and had to be done using the option CT.

If you specify a value of "Content-Type" in the headers statement, that header will override the value of the CT option.

UPLOADING DATA

You can use PROC HTTP to send data as well. This is typically done using a POST or PUT request like:

```
proc http url="http://httpbin.org/post"
 method="POST"
 in=input;
run;
```

This code sends the data contained in the fileref input to the URL using an HTTP POST request. If the content-type is not specified for a POST request, the default Content-Type will be application/x-www-form-urlencoded.

The behavior will be almost identical for a PUT versus a POST except that in 9.4m3 and later, the default Content-Type for a PUT is application/octet-stream instead of application/x-www-form-urlencoded as it is in prior versions.

If you wish to construct the input data on the fly, you can use a datastep like:

```
filename input TEMP;
data _null_;
file input recfm=f lrecl=1;
put "some data";
run;
```

If doing this, it is normally advisable to use a fixed record format as well as a record length of 1 as shown above to avoid any extraneous new line characters or padding.

In view 9.4m3 and later, the IN option also takes a quoted string, which means simple input like this can be sent like:

```
proc http url="http://httpbin.org/post"
 in="some data";
run;
```

HTTP COOKIES

HTTP cookies are small pieces of data that a server sends to the client to store. These cookies can be sent back with future requests and normally are used to identify if the request is coming from the same client. This can be used to allow the web server to remember a certain client state, such as, whether you have been logged in or not.

Cookies are stored and sent with PROC HTTP since 9.4m3, meaning that cookies received in one call to PROC HTTP will be sent on the next call to PROC HTTP, if the cookie is valid for the endpoint. Normally this just works, and you never even have to think about it, but there could be a situation where you want to turn off cookies.

Global Option

If you set the macro variable PROCHTTP_NOCOOKIES to a value other than "", cookies will not be stored or sent.

```
%let PROCHTTP_NOCOOKIES=1;
```

PROC Argument

You can also control cookies at the proc level by using the following options:

1.) NO_COOKIES – This prevents cookies on this proc call from being processed.

2.) CLEAR_COOKIES – This option clears any stored cookies before a call is made.

3.) CLEAR_CACHE – This option clears both stored cookies and stored connections.

PERSISTENT CONNECTIONS

Persistent connections or HTTP keep-alive is a way to send and receive multiple requests/responses using the same connection. This is used extensively in web-browsers as it can reduce latency tremendously by not constantly needing to create new connections and reduces the overhead of TLS handshakes. As of SAS 9.4m3, PROC HTTP uses persistent connections. Connections are kept alive by default, but if you need to, there are various ways to disable or close a connection:

1.) To force a connection to close after a response, you can add a header as follows:

```
proc http
  ...
  headers "Connection"="close";
  ...
```

2.) To completely disable saving a persistent connection, you can use the option NO_CONN_CACHE as follows:

```
proc http
  NO_CONN_CACHE
  ...
```

3.) To clear all persistent connections, use the option CLEAR_CONN_CACHE or CLEAR_CACHE as follows:

```
proc http
  CLEAR_CONN_CACHE
  ...
```

AUTHENTICATION

Since SAS 9.4, PROC HTTP has supported 3 types of HTTP Authentication: BASIC, NTLM, and Negotiate (Kerberos).

BASIC

BASIC authentication is (as the name suggests) very basic. The user name and password are sent in an Authorization header encoded in Base64. For all intents and purposes, this means that the password is being sent across the wire in clear text. BASIC authentication is not secure unless HTTPS is being used.

NEGOTIATE

HTTP Negotiate is an authentication extension that is used commonly to provide single sign-on capability to web requests. This is normally used in PROC HTTP when a password is not provided, since it will use the current user's identity for authentication. Since a password does not need to be specified in the SAS code, and the password is never actually transmitted across the wire, HTTP Negotiate is a much more secure form of authentication than BASIC.

NTLM

NTLM is an authentication protocol used on Microsoft systems. NTLM is not normally directly used, but instead selected during the Negotiate process described above. If the web server specifically asks for NTLM authentication, PROC HTTP will directly use it, but only on Microsoft systems.

OAUTH

OAuth is a standard for token-based authentication and authorization used in web requests. Unlike the authentication methods listed, OAuth does not require the client to have any form of the user's credentials, but instead uses a token that was acquired on the user's behalf. This is a very simplistic definition of OAuth, but the most important part is that OAuth does not require the client to possess a password and is used extensively in web applications throughout the internet.

AUTHENTICATION OPTIONS

Prior to SAS 9.4m3, the authentication options were:

- WEBUSERNAME – Used to set the user name when using BASIC authentication. Can also be used in Negotiate or NTLM if the system allows delegation of a user's credentials to someone other than the current user. This option was aliased to simply USERNAME in SAS 9.4m5.

```
proc http ...
  WEBUSERNAME="user" ...
```

- WEBPASSWORD – Used to set the password when using BASIC authentication. Can also be used in Negotiate or NTLM if the system allows delegation of a user's credentials to someone other than the current user. The value for this option can be encoded via PROC PWENCODE. This option was aliased to simply PASSWORD in SAS 9.4m5.

```
proc http ...
  WEBPASSWORD="pwd" ...
```

- HTTP_TOKENAUTH – Used in conjunction with a metadata server to generate a one-time password for use with a SAS Mid-tier.

```
proc http HTTP_TOKENAUTH ...
```

- WEBAUTHDOMAIN – A user name and password are retrieved from the metadata server for the specified authentication domain.

```
proc http WEBAUTHDOMAIN="authdom" ...
```

Prior to SAS 9.4m3, BASIC authentication was the default HTTP authentication that was used if the WEBUSERNAME and WEBPASSWORD arguments were set. If those arguments were set, the request would contain the Authentication header with the encoded user name and password. The more secure Negotiate or NTLM would only be used if the server subsequently responded with a 401 requesting one of NTLM or Negotiate.

In SAS 9.4m3 BASIC authentication is no longer the default authentication mechanism, and (by default) will only be used after receiving a 401 request. This is safer, because by default authentication will not be tried unless the server requests it.

New options were also added allowing more control over authentication, which are: AUTH_BASIC, AUTH_NTLM, and AUTH_NEGOTIATE. These options can be used separately or together to tell PROC HTTP what type of authentication it is able to perform. For example:

```
proc http
 url="www.secured-site.com"
 WEBUSERNAME="user"
 WEBPASSWORD="pass"
 AUTH_BASIC
 AUTH_NEGOTIATE;
run;
```

This code will send a request to www.secured-site.com and if it receives a 401 response that contains the WWW-Authenticate header with a value of BASIC or Negotiate, then one of those 2 authentication mechanism will be chosen based on priority in order of:

- Negotiate
- NTLM
- BASIC.

If, however the response is a 401, but contains a WWW-Authenticate header with a value of NTLM, then communication will be terminated, and the 401 response will be delivered to the client.

If only 1 authentication option is specified such as:

```
proc http
 url="www.secured-site.com"
 WEBUSERNAME="user"
 WEBPASSWORD="pass"
 AUTH_BASIC;
run;
```

Then that form of authentication will be used on the first request, thus preventing a server round trip.

If none of the authentication options are specified, then the proc will behave as if AUTH_BASIC, AUTH_NEGOTATE, and AUTH_NTLM are set.

SAS 9.4m5 also introduced the option OAUTH_BEARER, which is used to send the typical OAuth header of Authorization: Bearer <token>. An example of sending an OAuth bearer token would look as follows:

```
%let token=abcdefghijklmnop;
proc http
 url="httpbin.org/bearer"
 OAUTH_BEARER="&token.";
run;
```

The output generated is as follows:

```
> GET /bearer HTTP/1.1
> User-Agent: SAS/9
> Host: httpbin.org
> Accept: */*
> Authorization: Bearer abcdefghijklmnop
> Connection: Keep-Alive
```

The value can also be a fileref that contains the token:

```
filename token "path/to/token.dat";
proc http
 url="httpbin.org/bearer";
 OAUTH_BEARER=token;
run;
```

Prior to SAS 9.4m5, to send this type of request, you would need to manually generate the header:

```
proc http
```

```
 url="httpbin.org/bearer";
 headers
  "Authorization"="Bearer &token.";
run;
```

If SAS in running in a Viya® environment, then a value of SAS_SERVICES can be specified:

```
proc http
 url="http:\\viya-webservice.mydomain.com";
 OAUTH_BEARER=SAS_SERVICES;
run;
```

This will either use a token that has already been retrieved by the session or retrieve one for you.

DEBUGGING

It is useful to be able to debug a PROC HTTP statement and there are a few ways you can do that.

VERBOSE OPTION

The verbose option was the original way to view more detailed information about a specific PROC HTTP step. When this option is added to the PROC statement such as:

```
proc http url="httpbin.org/post"
 in="input"
 VERBOSE;
run;
```

certain proc inputs will be echoed to the SASLOG. The input fields that will be printed are:

- METHOD
- URL
- PROXYHOST
- PROXYPORT
- CT
- IN
- OUT
- HEADERIN
- HEADEROUT
- PROXYUSERNAME
- WEBUSERNAME
- WEBAUTHDOMAIN

This information can be helpful is some situations, but since it only really echoes values that are visible in the PROC statement, this is not useful in debugging the actual HTTP request/response.

DEBUG STATEMENT

The DEBUG statement was added in 9.4m5 to allow a detailed view of the HTTP request/response. This can be quite useful when you need to know exactly what is being sent/received to/from the server.

Debug Level

The easiest way to use the debug statement is with the LEVEL argument:

```
proc http url="httpbin.org/post"
 in="somedata";
 DEBUG LEVEL=3;
run;
```

There are 3 levels of debugging information for which an example of level 3 is shown:

```
> POST /post HTTP/1.1
> User-Agent: SAS/9
> Host: httpbin.org
> Accept: */*
> Connection: Keep-Alive
> Content-Length: 8
> Content-Type: application/x-www-form-urlencoded
>
> 000000000DAD91A0: 73 6F 6D 65 64 61 74 61                         somedata
< HTTP/1.1 200 OK
< Connection: keep-alive
< Server: gunicorn/19.9.0
< Date: Mon, 28 Jan 2019 19:26:22 GMT
< Content-Type: application/json
< Content-Length: 379
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Credentials: true
< Via: 1.1 vegur
<
< 000000000DAD9296: 7B 0A 20 20 22 61 72 67 73 22 3A 20 7B 7D 2C 20 {.  "args": {},
< 000000000DAD92A6: 0A 20 20 22 64 61 74 61 22 3A 20 22 22 2C 20 0A .  "data": "", .
< 000000000DAD92B6: 20 20 22 66 69 6C 65 73 22 3A 20 7B 7D 2C 20 0A   "files": {}, .
< 000000000DAD92C6: 20 20 22 66 6F 72 6D 22 3A 20 7B 0A 20 20 20 20   "form": {.
< 000000000DAD92D6: 22 73 6F 6D 65 64 61 74 61 22 3A 20 22 22 0A 20 "somedata": "".
< 000000000DAD92E6: 20 7D 2C 20 0A 20 20 22 68 65 61 64 65 72 73 22  }, .  "headers"
< 000000000DAD92F6: 3A 20 7B 0A 20 20 20 20 22 41 63 63 65 70 74 22 : {.    "Accept"
< 000000000DAD9306: 3A 20 22 2A 2F 2A 22 2C 20 0A 20 20 20 20 22 43 : "*/*", .    "C
< 000000000DAD9316: 6F 6E 6E 65 63 74 69 6F 6E 22 3A 20 22 63 6C 6F onnection": "clo
< 000000000DAD9326: 73 65 22 2C 20 0A 20 20 20 20 22 43 6F 6E 74 65 se", .    "Conte
< 000000000DAD9336: 6E 74 2D 4C 65 6E 67 74 68 22 3A 20 22 38 22 2C nt-Length": "8",
< 000000000DAD9346: 20 0A 20 20 20 20 22 43 6F 6E 74 65 6E 74 2D 54  .    "Content-T
< 000000000DAD9356: 79 70 65 22 3A 20 22 61 70 70 6C 69 63 61 74 69 ype": "applicati
< 000000000DAD9366: 6F 6E 2F 78 2D 77 77 77 2D 66 6F 72 6D 2D 75 72 on/x-www-form-ur
< 000000000DAD9376: 6C 65 6E 63 6F 64 65 64 22 2C 20 0A 20 20 20 20 lencoded", .
< 000000000DAD9386: 22 48 6F 73 74 22 3A 20 22 68 74 74 70 62 69 6E "Host": "httpbin
< 000000000DAD9396: 2E 6F 72 67 22 2C 20 0A 20 20 20 20 22 55 73 65 .org", .    "Use
< 000000000DAD93A6: 72 2D 41 67 65 6E 74 22 3A 20 22 53 41 53 2F 39 r-Agent": "SAS/9
< 000000000DAD93B6: 22 0A 20 20 7D 2C 20 0A 20 20 22 6A 73 6F 6E 22 ".  }, .  "json"
< 000000000DAD93C6: 3A 20 6E 75 6C 6C 2C 20 0A 20 20 22 6F 72 69 67 : null, .  "orig
< 000000000DAD93D6: 69 6E 22 3A 20 22 31 34 39 2E 31 37 33 2E 38 2E in": "149.173.8.
< 000000000DAD93E6: 32 36 22 2C 20 0A 20 20 22 75 72 6C 22 3A 20 22 26", .  "url": "
< 000000000DAD93F6: 68 74 74 70 3A 2F 2F 68 74 74 70 62 69 6E 2E 6F http://httpbin.o
< 000000000DAD9406: 72 67 2F 70 6F 73 74 22 0A 7D 0A               rg/post".}.
```

- Debug level 1 will print the request and response headers. All input is prefixed by a > and all output is prefixed by a <
- Debug level 2 will print everything from level 1 and will also print the request body.
- Debug level 3 will print everything from level 2 as well as the response body.

NOTE: In 9.4m5 the use of debug levels 2 and 3 would always print the request/response bodies in plain text, which is unsafe if the content were binary. This was changed in 9.4m6 where request/response bodies are always printed in binary dump format.

Debug Parameters

In 9.4m6, more options were added to the debug statement that allow you to more finely control what information gets printed out.

- OUTPUT_TEXT – Since 9.4m6, the default format for request or response bodies is a binary dump. If you know that the input and output is plain text, you can use this option to print the data as text instead. Only use this option if you know for certain that the data will not contain any non-printable character or else the system could become unstable. An example of a debug text response is:

```
<
< {  "args": {},   "data": "",   "files": {},   "form": {    "somedata": ""  },
"headers": {    "Accept": "*/*",     "Connection": "close",     "Content-Length": "8",
"Content-Type": "application/x-www-form-urlencoded",     "Host": "httpbin.org",
"User-Agent": "SAS/9"  },   "json": null,   "origin": "149.173.8.26",   "url":
"http://httpbin.org/post"}
```

- REQUEST_BODY – If this option is specified the request body will be printed.
- RESPONSE_BODY – If this option is specified the response body will be printed.
- REQUEST_HEADERS – If this option is specified the requests headers will be printed.
- RESPONSE_HEADERS – If this option is specified the responses headers will be printed.
- NO_RESPONSE_BODY – Turns off printing of the responses body
- NO_REQUEST_BODY – Turns off printing of the requests body
- NO_REQUEST_HEADERS – Turns off printing of the requests headers
- NO_RESPONSE_HEADERS – Turns off printing of the responses headers
- OFF – Completely disables all debugging output

Level can be combined with any of the other options, allowing you to easily create your own debug level that meets your needs:

```
proc http url="httpbin.org/post"
 in="somedata";
 DEBUG LEVEL=3 NO_REQUEST_HEADERS NO_REQUEST_BODY RESPONSE_BODY;
run;
```

which produces the following output:

```
< HTTP/1.1 200 OK
< Connection: keep-alive
< Server: gunicorn/19.9.0
< Date: Mon, 28 Jan 2019 19:55:01 GMT
< Content-Type: application/json
< Content-Length: 379
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Credentials: true
< Via: 1.1 vegur
<
< 000000000DAE93F6: 7B 0A 20 20 22 61 72 67 73 22 3A 20 7B 7D 2C 20 {.  "args": {},
< 000000000DAE9406: 0A 20 20 22 64 61 74 61 22 3A 20 22 22 2C 20 0A .  "data": "", .
< 000000000DAE9416: 20 20 22 66 69 6C 65 73 22 3A 20 7B 7D 2C 20 0A   "files": {}, .
< 000000000DAE9426: 20 20 22 66 6F 72 6D 22 3A 20 7B 0A 20 20 20 20   "form": {.
< 000000000DAE9436: 22 73 6F 6D 65 64 61 74 61 22 3A 20 22 22 0A 20 "somedata": "".
< 000000000DAE9446: 20 7D 2C 20 0A 20 20 22 68 65 61 64 65 72 73 22  }, .  "headers"
< 000000000DAE9456: 3A 20 7B 0A 20 20 20 20 22 41 63 63 65 70 74 22 : {.    "Accept"
< 000000000DAE9466: 3A 20 22 2A 2F 2A 22 2C 20 0A 20 20 20 20 22 43 : "*/*", .    "C
< 000000000DAE9476: 6F 6E 6E 65 63 74 69 6F 6E 22 3A 20 22 63 6C 6F onnection": "clo
< 000000000DAE9486: 73 65 22 2C 20 0A 20 20 20 20 22 43 6F 6E 74 65 se", .    "Conte
< 000000000DAE9496: 6E 74 2D 4C 65 6E 67 74 68 22 3A 20 22 38 22 2C nt-Length": "8",
< 000000000DAE94A6: 20 0A 20 20 20 20 22 43 6F 6E 74 65 6E 74 2D 54  .    "Content-T
< 000000000DAE94B6: 79 70 65 22 3A 20 22 61 70 70 6C 69 63 61 74 69 ype": "applicati
< 000000000DAE94C6: 6F 6E 2F 78 2D 77 77 77 2D 66 6F 72 6D 2D 75 72 on/x-www-form-ur
< 000000000DAE94D6: 6C 65 6E 63 6F 64 65 64 22 2C 20 0A 20 20 20 20 lencoded", .
< 000000000DAE94E6: 22 48 6F 73 74 22 3A 20 22 68 74 74 70 62 69 6E "Host": "httpbin
< 000000000DAE94F6: 2E 6F 72 67 22 2C 20 0A 20 20 20 20 22 55 73 65 .org", .    "Use
< 000000000DAE9506: 72 2D 41 67 65 6E 74 22 3A 20 22 53 41 53 2F 39 r-Agent": "SAS/9
< 000000000DAE9516: 22 0A 20 20 7D 2C 20 0A 20 20 22 6A 73 6F 6E 22 ".  }, .  "json"
< 000000000DAE9526: 3A 20 6E 75 6C 6C 2C 20 0A 20 20 22 6F 72 69 67 : null, .  "orig
< 000000000DAE9536: 69 6E 22 3A 20 22 31 34 39 2E 31 37 33 2E 38 2E in": "149.173.8.
< 000000000DAE9546: 32 36 22 2C 20 0A 20 20 22 75 72 6C 22 3A 20 22 26", .  "url": "
< 000000000DAE9556: 68 74 74 70 3A 2F 2F 68 74 74 70 62 69 6E 2E 6F http://httpbin.o
< 000000000DAE9566: 72 67 2F 70 6F 73 74 22 0A 7D 0A               rg/post".}.
```

URL REDIRECTION

URL redirection is a common technique on the World Wide Web for letting a resource be accessible by more than one address. When you open a URL and get back a 300-level response code, this typically means that the resource that you are looking for is available at a different URL, which will be given in the Location header in the response. This happens all the time in web browsers when you navigate to a .net address, but the real site is a .com or you request an HTTP address, but the site only accepts HTTPS. You typically don't even notice that this is happening in a browser because it happens automatically. PROC HTTP also has automatic handling of URL redirection.

The simplest case is where you issue a GET request to a URL and a redirect is returned like if you opened http://www.sas.com you would get redirected to https://www.sas.com. You can see this with the code below:

```
proc http
 url="http://www.sas.com";
 debug level=2;
run;
```

This produces the following output:

```
> GET / HTTP/1.1
> User-Agent: SAS/9
> Host: www.sas.com
> Accept: */*
> Connection: Keep-Alive
>
< HTTP/1.1 301 Moved Permanently
< Date: Wed, 06 Feb 2019 20:10:09 GMT
< Server: Apache
< Location: https://www.sas.com/
< Content-Length: 228
< Keep-Alive: timeout=15, max=100
< Connection: Keep-Alive
< Content-Type: text/html; charset=iso-8859-1

> GET / HTTP/1.1
> User-Agent: SAS/9
> Host: www.sas.com
> Accept: */*
> Connection: Keep-Alive
```

Notice how the initial response was a 301 stating that the location that you were requesting has been permanently moved and any further requests should go ahead and use the URL that is in the Location header. PROC HTTP then issues another request automatically to the target URL.

NOFOLLOWLOC

Most of the time, letting proc HTTP automatically follow redirects is completely fine, but on the rare occasion that you do NOT want PROC HTTP to follow a redirect, you prevent it from happening by using the NOFOLLOWLOC option that was introduced in version 9.4m5:

```
proc http
 url="http://www.sas.com"
 NOFOLLOWLOC;
run;
```

FOLLOWLOC

In contrast to the NOFOLLOW options, there is also a FOLLOWLOC option. You might be thinking "If following a redirect is the default, why would I need an option to follow a redirect?". The reason is that following of a redirect is only automatic with a GET request. The reason for this is partially historical and partially for security purposes, but if you know that you need to follow a redirect on a POST or PUT, then you can use this option:

```
proc http url="httpbin.org/redirect-to?url=/post"
 method="POST"
 FOLLOWLOC
 in="some data";
 run;
```

This request will initially get back a 300-level response, but since FOLLOWLOC is set, will POST the data again to the redirected location.

HEADEROUT_OVERWRITE

While automatic following of redirects is convenient, it can sometimes lead to confusion if you are analyzing the response headers that are written with the HEADEROUT argument. Take the following code for example:

```
filename headers TEMP;

proc http url="httpbin.org/redirect-to?url=get"
 HEADEROUT=headers;
run;

data _null_;
infile headers;
input;
put _infile_;
run;
```

The code sends a request that gets redirected once, which is followed, and then gets a successful response. The response headers are written to the headers fileref and then echoed to the SASLOG.

The potential problem is that the HEADEROUT fileref will contain the headers of the initial request as well as the subsequent request(s). In this case the HEADEROUT fileref would be as follows:

```
HTTP/1.1 302 FOUND
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Content-Length: 0
Location: get

HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: application/json
Content-Length: 207
```

If you want to process the response headers, the header block that you most likely want to process is at the END of the file, which means you would have to write code to find the start of the last header block before processing.

This can be avoided by using the option HEADEROUT_OVERWRITE, which instead of appending to the HEADEROUT fileref, will overwrite it. With this option, the HEADEROUT fileref will always contain the last response headers that were received, making processing simpler.

REDIRECT LIMIT

PROC HTTP will follow redirects up to a certain limit. As of 9.4m6, that limit is set at 5 and is not configurable.

SSLPARMS

PROC HTTP automatically handles HTTPS, but sometimes a user needs to not only be able to handle server certificates but be able to handle client side certificates as well. This is accomplished by setting some SAS options that are specified in https://go.documentation.sas.com/?docsetId=secref&docsetTarget=n0bmaslhpevq8gn1l614jmbd63wi.htm&docsetVersion=9.4&locale=en.

To have PROC HTTP send a client certificate, you would need to set the following SAS options on UNIX systems: SSLPKCS12LOC and SSLPKCS12PASS or SSLCERTLOC, SSLPVTKEYLOC, and SSLPVTKEYPASS. On Windows systems you would need to set: SSLCERTSUBJ or SSLCERTISS and SSLCERTSERIAL.

An example of this would be:

```
options SSLCERTISS="BadSSL Client Root Certificate Authority";
options SSLCERTSERIAL="00f0bb28c1637ec957";

proc http
 url="https://client.badssl.com/";
 debug level=1;
run;
```

Although this code works perfectly fine, any further requests (not just using proc HTTP) will also send this client certificate, since the options are global. Starting with 9.4m6, PROC HTTP has the ability to set SSL option locally, so they only effect the current PROC HTTP statement. This is done by using the SSLPARMS statement. The above code can be rewritten as such:

```
proc http
 url="https://client.badssl.com/";
 SSLPARMS
  SSLCERTISS="BadSSL Client Root Certificate Authority"
  SSLCERTSERIAL="00f0bb28c1637ec957";
 debug level=1;
run;
```

This code does the exact same thing, except the global SSL options and environment variables are left untouched.

The SSLPARMS statement can take any off the options or environment variables that start with SSL listed in the "Encryption in SAS 9.4" documentation.

MISCELLANEOUS

TIMEOUT

If for some reason the webserver that you are trying to access stops responding on the connection or while processing the request or response, PROC HTTP will continue to wait.

In 9.4m5, the TIMEOUT option was added so that the user can set a maximum amount of time that PROC HTTP is allowed to run. TIMEOUT should take a positive integer as a value, which represents the maximum number of seconds to wait before the proc is terminated. An example of using the TIMEOUT option is:

```
proc http url="httpbin.org/drip?numbytes=10&duration=5&delay=5"
 TIMEOUT=9;
run;
```

The following log is produced::

```
ERROR: HTTP Operation Timed Out.
NOTE: PROCEDURE HTTP used (Total process time):
      real time           9.00 seconds
      cpu time            0.00 seconds

NOTE: The SAS System stopped processing this step because of errors.
```

EXPECT 100-CONTINUE

When uploading data, there is a chance that the server will need to respond to you before it can accept the data. An example of this is if you need to authenticate. When this happens, you send a request to the server, which includes the header and body, the server responds

with a 401, and the request is resent with the necessary credentials and the body. In a situation like this, the body ends up being sent over the network more than once.

To combat this problem, HTTP1.1 introduced the Expect: 100-continue header. When this header is sent, the body is not uploaded immediately after the header, but instead waits on the server to send a response:

```
HTTP/1.1 100 Continue
```

If the 100 continue response is received, the body will be uploaded. This is typically only helpful if the request body is large and the probability for the server to reject the initial upload is high.

To use the 100 Continue mechanism in PROC HTTP, you can use the option EXPECT_100_CONTINUE. An example of using this option in PROC HTTP is:

```
proc http
 URL="httpbin.org/put"
 METHOD="PUT"
 IN="data"
 EXPECT_100_CONTINUE;
run;
```

The 100 continue will only be sent on a PUT or POST.

If an Expect 100-Continue header is sent and the server does not respond within 5 seconds, the body will be sent anyway. This time-out is not configurable.

CONCLUSION

PROC HTTP is a powerful and versatile SAS Procedure allowing you to access pretty much everything the web has to offer. PROC HTTP will continue to be updated, making tasks easier and more intuitive. Hopefully this paper has presented most of what this procedure is capable of and has equipped to you to take advantage of said capabilities.

REFERENCES

Fielding, Roy, and Reschke, Julian. "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing". RFC7230. June 2014. https://tools.ietf.org/html/rfc7230.

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