MWSUG 2024 - Paper BB-052

Using SAS® to Consume Data from APIs

Somaiya Shakil, Medical Home Network, Chicago, IL
Jack Shoemaker, Medical Home Network, Chicago, IL

ABSTRACT

The world is moving to the cloud and that means using APIs to consume and publish data. At the simplest level, an API is just a web address that you could use a command-line tool like cURL to access. What comes back is often a JSON data payload. SAS® provides several tools which make access easier and more robust. Using the public CMS Beneficiary Claims Data API (BCDA), this paper will show how to use a combination of command-line tools, PROC HTTP, and PROC JSON to communicate with the CMS API. The BCDA returns data in newline-delimited JSON (NDJSON) which presents a few more wrinkles to iron out. NDJSON files contain one or more JSON payloads. PROC JSON operates on a single payload, so needs some help to consume NDJSON-formatted data. This paper will present two solutions to this challenge as well as a complete package that authenticates and consumes the CMS data.

INTRODUCTION

Our objective is to grab data from CMS and store those data in SAS® data sets for business and other analytic uses. There are three distinct steps: requesting the data, retrieving the data, and unpacking the results into SAS® data sets.

BRIEF DESCRIPTION OF AN API

An application programming interface (API) is a connection between computers that allows a process on computer A to execute a program on computer B. In practice, an API call looks like a uniform resource identifier (URI), that is a web address. There are three parts to the URI for an API call: the host or server address, the path to the program or executable object often referred to as an endpoint, and optionally a set of parameters if the endpoint requires parameters. Not all API calls require parameters. For example, the BCDA used in this example provides an endpoint that returns global metadata for the API. The full URI for this resource is https://api.bcda.cms.gov/api/v2/metadata. Using our three-part terminology, the host is *api.bcda.cms.gov* and the endpoint is *api/v2/metadata*.

Using the command-line tool, curl, you can request the metadata with this command:

```
$ curl -X GET https:/api.bcda.cms.gov/api/v2/metadata
```

The command will return a JSON payload to the standard output, that is, the terminal window that executed the curl command. Here is a portion of what is returned from the API call above.

```
{"date":"2024-10-02T16:53:04+00:00","fhirVersion":"4.0.1", …
```

You can direct the output to a file and treat the result as a JSON file.

USING THE HTTP PROCEDURE

For simple API calls, using curl is quick and easy. However, SAS® provides a procedure called HTTP which allows you to capture and control other aspects of the API call. Here's what the simple metadata call looks like using PROC HTTP. We will use macro symbols to hold the host and endpoint names.

```
%let HOST = api.bcda.cms.gov;
%let SVC = api/v2/metadata;

proc http
    method = "GET"
    url = "https://&HOST./&SVC."
    out = response
    headerout = headout
    ;
```

The tokens “response” and “headout” are filename references. The standard output goes to the response file name; header information not available from the simple curl call goes to the headout file name.

The response file is a JSON payload. We can associate the JSON LIBNAME engine with this FILENAME reference so that the resulting library contains a set of data sets representing the original JSON file.

```
libname bcda json fileref = response;
proc datasets library = bcda;
```

Here is what PROC DATASETS reports to the SAS log.

Libref	BCDA	
Engine	JSON	
Access	READONLY	
Physical Name	/SASRAID0/SAS_workED91000EC692_tux/#LN00033	
		Member
#	Name	Type
1	ALLDATA	DATA
2	EXTENSION_EXTENSION	DATA
3	FORMAT	DATA
4	IMPLEMENTATION	DATA
5	INSTANTIATES	DATA
6	RESOURCE_OPERATION	DATA
7	REST	DATA
8	REST_INTERACTION	DATA
9	REST_RESOURCE	DATA
10	REST_SECURITY	DATA
11	ROOT	DATA
12	SECURITY_EXTENSION	DATA
13	SECURITY_SERVICE	DATA
14	SERVICE_CODING	DATA
15	SOFTWARE	DATA

Table 1. PROC DATASETS results on JSON LIBNAME

The JSON LIBNAME engine creates an ALLDATA data set that contains all the payload data in one place as well as a set of individual data sets corresponding to each level or branching tree within the JSON payload. Although the JSON LIBNAME engine will parse the payload without any further guidance, assembling the results requires knowledge of the content and some plan for how to use the data.

AUTHENTICATION

The BCDA employs a two-step authentication process that works as follows: using a user and password credential set, obtain a temporary access token to authenticate data requests. Here is an example using curl to obtain an authentication token:

```
$ curl -n -X POST https:/api.bcda.cms.gov/auth/token
```

Which returns a JSON payload containing the access token

```
{"access_token": "eyJhbGciOiJSUzUxMiIsInR5cCI6IkpXVCJ9.eyJleHA…
```

This token is required for future BCDA data requests and expires in 1,200 seconds. The token is over 600 characters long. Using SAS® and the HTTP procedure we can load the access token into a macro symbol for use in subsequent BCDA data requests. Note that the username and password have been placed in macro symbols in the example below. The curl command uses the standard 'dot file' (.netrc) to obtain the username and password. For SAS® programs a utility macro reads the dot file and loads macro symbols called &USER and &PASS.

```
%let HOST = api.bcda.cms.gov;
%let SVC = auth/token;
proc http
    method = "POST"
    url = https://&HOST./&SVC.
    auth_basic
    webusername = "&USER."
    webpassword = "&PASS."
    out = response
    headerout = headout
    ;
    headers
        "accept" = "application/json"
    ;
```

The filename referenced by the response FILENAME pointer contains the JSON payload. We can use the JSON LIBNAME engine to parse the JSON payload and make the contents available in the ROOT data set. For example:

```
libname auth json fileref = response;
data _null_;
    set auth.root;
    call symputx( 'ACCESS_TOKEN', access_token );
    run;
```

The example above is specific to the BCDA. That is, the JSON payload returned by the 'auth/token' endpoint has three elements: "access_token", "expires_in", and "token_type". Other APIs that employ this sort of authentication may have a different set of elements, but one surely contains the token needed for subsequent data requests. In this example, the access token is in a payload element called "access_token". You don't know this without reviewing the response file or reading the API documentation.

REQUESTING DATA

Requesting data through the BCDA is a three-step process. Step 1 is making the data request. Step 2 is finding the resulting data files. Step 3 is downloading those files.

Using the access token obtain in the previous section, a data request looks like this:

```
%let HOST = api.bcda.cms.gov;
%let SVC = api/v2/Patient/$export?_type=Claim;
proc http
    oauth_bearer = "&ACCESS_TOKEN."
    method = "GET"
    url = https://&HOST./&SVC.
    out = response
    headerout = headout
    ;
    headers
        "accept" = "application/fhir+json"
    ;
```

From the BCDA documentation we learn that the `api/v2/Patient/$export endpoint` takes a parameter called `_type` which requests one of five available file types. In our example, we have requested the 'Claim' data type.

The returned header will tell us where to look for the resulting data by way of the 'Content-Location' element. We will use this location in Step 2 to check the status of the data request and learn the specific files to download in the third step. The header consists of a set of name-value pairs. We create a SAS data set to store these values as HDR_KEY and HDR_VAL. For example:

```
data header( keep = hdr_: );
    length hdr_key $ 60 hdr_val $ 100;
    infile headout length = l;
    input;
    if _n_ = 1 then call symputx( '_LABEL_', trim( _infile_ ) );
    else do;
        if l > 0 then do;
            hdr_key = scan( _infile_, 1, ':' );
            hdr_val = substr( _infile_, length( trim( hdr_key ) ) + 3 );
            output;
            end;
        end;
```

Obs	hdr_key	hdr_val
1	Date	Wed, 09 Oct 2024 15:20:55 GMT
2	Content-Length	0
3	Connection	keep-alive
4	Cache-Control	no-cache; no-store; must-revalidate; max-age=0
5	Content-Location	https://api.bcda.cms.gov/api/v2/jobs/87956
6	Pragma	no-cache
7	Strict-Transport-Security	max-age=31536000; includeSubDomains; preload
8	X-Content-Type-Options	nosniff

Table 2. The HEADER data set

Based on the header results, we will look for files in the URI specified in the Content-Location attribute. The publisher, CMS in this case, guarantees that the final node is a unique number known as the job ID.

If we put the job ID in a macro symbol called &JOBID, then we can obtain a list of files by using the "jobs" endpoint as follows.

```
%let HOST = api.bcda.cms.gov;
%let SVC = api/v2/jobs/&JOBID.;
proc http
    oauth_bearer = "&ACCESS_TOKEN."
    method = "GET"
    url = https://&HOST./&SVC.
    out = response
    headerout = headout
    ;
    headers
        "accept" = "application/fhir+json"
    ;
```

In this case, the response file will contain the full path to the data files. We will have those details in a SAS ® data set to pass along to step 3.

```
libname _bcda_ json fileref = response;
proc print data = _bcda_.OUTPUT;
```

url
https://api.bcda.cms.gov/data/87956/d039269b-455f-44e1-b756-6e9a5af72516.ndjson
https://api.bcda.cms.gov/data/87956/b89c1ad9-2cab-4e49-8d5d-35b1010f9546.ndjson
https://api.bcda.cms.gov/data/87956/8ac213f3-ac70-4c61-8b53-2c586422a30e.ndjson
https://api.bcda.cms.gov/data/87956/0282bc03-9f23-4d54-bbb5-13e9b55d7419.ndjson

Table 3. The OUTPUT data set created by the JSON LIBNAME engine

With the data locations saved in a data set, we now download each one in turn.

```
%let HOST = api.bcda.cms.gov;
%let JOBID = 87956;
%let U = d039269b-455f-44e1-b756-6e9a5af72516.ndjson;
%let SVC = https://api.bcda.cms.gov/data/&JOBID./&U.;
    proc http
        oauth_bearer = "&ACCESS_TOKEN."
        method = "GET"
        url = "&SVC."
        out = response
        headerout = headout
        ;
    headers
        "accept" = "application/fhir+json"
        ;
```

In this case, the endpoint will download the file to the filename specified by the response FILENAME pointer. The step above is repeated for every URL specified in the OUTPUT data set created in step 2.

ASSEMBLY

At this point we will have a set of NDJSON files that we wish to load into SAS® data sets for business use and analysis. The NDJSON format presents a small challenge. Each row in the NDJSON file is a proper JSON payload. The payloads are ‘newline-delimited’ hence the designation as NDJSON instead of just JSON. The JSON LIBNAME engine works on a single payload, not a collection of payloads as they are presented in the returned data files.

There are at least two ways to attack this problem. Using data-step programming, the NDJSON file may be converted to a JSON file by turning the newline characters to commas and then wrapping the whole file inside an outer collection called “records” or any name you chose. For example:

```
filename ndjson "d039269b-455f-44e1-b756-6e9a5af72516.ndjson;";
data _null_;
    file json;
    if _n_=1 then put '[{"records":' / '[' @;
    else if eof then put ']}]';
    else put ',' @;
    infile ndjson end=eof;
    input; put _infile_;
```

Now the JSON LIBNAME engine will work on the resulting JSON file. This technique works great if the individual rows in the NDJSON files are not too large, that is too large for normal SAS® file processing.

Another approach is to send each row to the JSON LIBNAME engine individually by piping the data row to a SAS® program that reads from standard input. For example, here is a helper program that will apply the JSON LIBNAME engine to STDIN and surface the results in a SAS® library called INROW.

```
libname inrow json fileref = STDIN ordinalcount = ALL;
```

If the statement above is in a source file called `assemble.sas`, and we have a wrapper script called `runsas.sh`, then we can pipe each row to that program using a technique like this.

```
f #!/bin/gawk -f
#
BEGIN {
    bash = "runsas.sh"
    sas  = "assemble.sas"
}
{
    if ( FNR > 0 ) {
        cmd = bash " " sas " " FNR
        print $0 | cmd
        close(cmd)
        system("sleep 1")
    }
}
END {
    printf " NOTE: Saw %d payloads in [%s]\n", NR, FILENAME
}
```

This technique overcomes the file size limits that the first approach may encounter; however, each row creates a set of SAS® data sets that need to be combined. Pain arises when the character fields have differing lengths or attributes. What's needed is a smart append procedure inside the assembly program. There is extensive literature on that topic which is beyond the scope of this paper.

The choice of awk as the scripting language is arbitrary. Any tool that allows you to execute SAS® programs, including SAS® itself will work, though using SAS® will introduce the 32767 line size problem which using awk avoids. The script takes a single parameter – the name of the raw NDJSON file and calls the assemble program once per row.

CONCLUSION

As more computing moves to the cloud, so will the use of APIs to fetch and analyze data. The SAS® system provides several tools that work well in this new topology. The HTTP procedure executes API calls while the JSON LIBNAME engine helps parse the API response. This paper demonstrated the use of those tools using a public API called the BCDA as the example.

CONTACT INFORMATION

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

Somaiya Shakil
sshakil@mhnchicago.org
Jack Shoemaker
jshoemaker@mhnchicago.org