Paper 179-2022

REST API for the Weary Beginner

Jinson Erinjeri, Onyx Government Services

ABSTRACT

API stands for Application Programming Interface which is a mode of communication between programs in order to transfer data. REST is an acronym for REpresentational State Transfer which is nothing but a standard that guides the design and development of processes that enable effective communication between programs. Therefore, in the World Wide Web environment, REST API is used to interact with data stored on web servers. The objective of this paper is to present the basics of REST API's for a novice learner in simple terms coupled with examples of publicly available API. In addition, this paper will present features available in SAS as well as Python for communicating with REST API web services using the same set of examples.

REST API BASICS

An API is a medium for communicating between programs with the primary objective of transferring data. If a program has an API, it implies that some parts of its data are exposed for consumption. The consumer in this case is called the client and it could be the front end of the same program or an external program. In order to get this consumable data, the client sends a structured request to the API and if the request fulfills the requirements, a response with relevant data is sent back to the client. This response usually comes in the form of JSON or XML data from the API. The API acts as a liaising agent and interacts with the server which contains the resources requested by the client without providing direct access to the data.

To explain and remember the concept of API in a simple context, it would be best to imagine a customer using a snack vending machine shown in Figure 1. The customer selects the appropriate snack or drink by entering the item number (ex. A5 or C4) in the keypad and then providing the appropriate amount of payment in the bill/coin insertor.

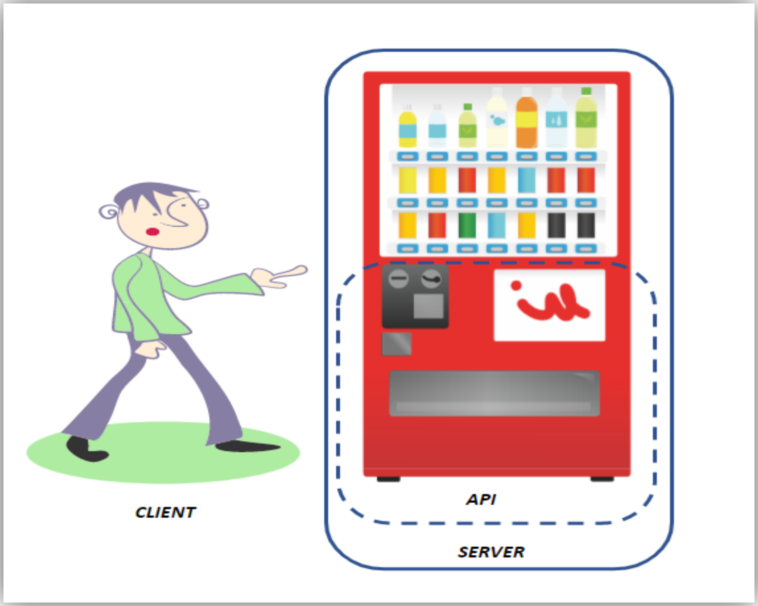


Figure 1. Concept of API with an Example of a Vending Machine[1]

[1] All vector images used in Figure 1 are sourced from https://publicdomainvectors.org/.

Once all the inputs are verified, the vending machine dispenses the product from the tray of goods which the customer picks up at the product dispenser door. In this example the customer is the client, the keypad, bill/coin insertor and dispenser door together constitute the API with goods of tray being the server. This is presented in Figure 1 and it is important to note that the API is residing in the server itself. The API acts as the liaising agent between the client and server to complete the transaction right from request to delivery of the product. In this set-up, it is important to note that the client does not need to know anything about the working of the server nor does the server need to know anything about the client. There can be instances where the client does not enter the correct change, keys an incorrect code or products in the goods tray are stuck and for all these types of scenarios, an error is displayed on the screen of the vending machine. Similar error codes are also displayed when placing requests in real time applications and are called HTTP error codes. It is important to note that an API is created by a developer on the server side and that some API's may require some sort of authorization to access the data.

We have described that the API is a liaising agent but how and what this agent should do is determined by REST, an acronym for REpresentational State Transfer. REST is nothing but a standard that guides developers in the design and development of an API. The goal of REST is to enable effective communication between two parties. This is done by enacting rules concerning Client–server architecture, Statelessness, Cacheability, Layered system, Code on demand, and an Uniform interface. An API that follows some or all of the six guiding constraints of REST is considered to be RESTful. These set of constraints when applied to the system results in desirable properties such as performance, scalability, simplicity, modifiability, visibility, portability and reliability. More details about REST can be found in Roy Fielding's dissertation link provided in the reference section.

Having discussed both API and REST, it is important to note the key elements of the REST API paradigm:

1. Client or software running at the user's end.
2. Server that offers an API as a liaising agent to access its data or features.
3. Resource is the content the server can provide to the client such as a text or video file.

Resources are data sets on which we want to perform operations. When this data is requested via unique URL, there is a REpresentational State Transfer. The data or record present in database is converted to another format which can be JSON, XML or plain text. The client needs the resource and initiates an HTTP request which is channeled via the API to the server. The server returns the request with an HTTP response with encoded data on the resource. One of the guiding constraints of REST states that one should be able to get a slice of data (resource) when you link to a specific URL. In this set-up, the URL is called a request while the data sent back to the client is called a response.

The author's experience has been that various terms associated with the REST API have been interchangeably used and it might be a bit confusing in the beginning. However, every API comes with documentation that details what data is available and how to structure your request in order to get a valid response. It is our recommendation to read over the API documentation before consuming the data to avoid errors downstream.

STRUCTURE OF A REQUEST

An HTTP request consists of four elements:

1. Endpoint
2. Method (HTTP Method)
3. Request Header
4. Body

Endpoint

An endpoint contains a Uniform Resource Identifier (URI) which is just a route to find the resource on the internet. The most common type of URI is a Unique Resource Location (URL) which consists of root endpoint and path. The root endpoint is the starting point of the API you are requesting from whereas the path is the resource you are requesting. For example, in the end point for rates of exchange data https://api.fiscaldata.treasury.gov/services/api/fiscal_service/v1/accounting/od/rates_of_exchange, "https://api.fiscaldata.treasury.gov/services/api/fiscal_service" is the root endpoint whereas the path is "/v1/accounting/od/rates_of_exchange". As mentioned before, it is important always to refer to the documentation provided by the API provider to obtain how various terms are defined and to determine the structure of the requests. It is important to be aware that endpoints can have query parameters and these are not part of the REST architecture, however it is widely used by many API's. Query parameters provides the option to modify your request with key-value pairs. They always begin with a question mark (?) and each parameter pair is then separated with an ampersand (&). The usage of query parameters will be presented later in the paper.

HTTP Method (Method)

The HTTP Method or method is the action you want to be performed on the resource. There are mainly five of them:

1. GET
2. POST
3. PUT
4. PATCH
5. DELETE

These methods provide what the request is supposed to do. They are used to primarily perform four possible actions: Create, Read, Update and Delete (CRUD).

Method	**Action Performed**
GET	Request used to get a resource from a server. The server looks for the requested data and relays it back to the client. A GET request performs a READ operation and is the default request method.
POST	Request used to create a new resource on a server. The server creates a new entry in the resource and relays back whether the creation is successful. A POST request performs a CREATE operation.
PUT/PATCH	Both these requests are used to update a resource on a server and relay back the status of the update. The only difference is that PUT is used to replace the resource in entirety whereas PATCH is used for a partial update. PUT and PATCH both constitute the UPDATE operation.
DELETE	Request used to delete the resource on a server. The server deletes an entry in the resource and relays back whether the deletion is successful. A DELETE request performs a DELETE operation.

Table 1. HTTP Methods

HTTP methods on endpoints handle requests between a web browser and web server. When creating a new endpoint, one can specify the request methods associated with it. These methods determine how a webpage interacts with the web server.

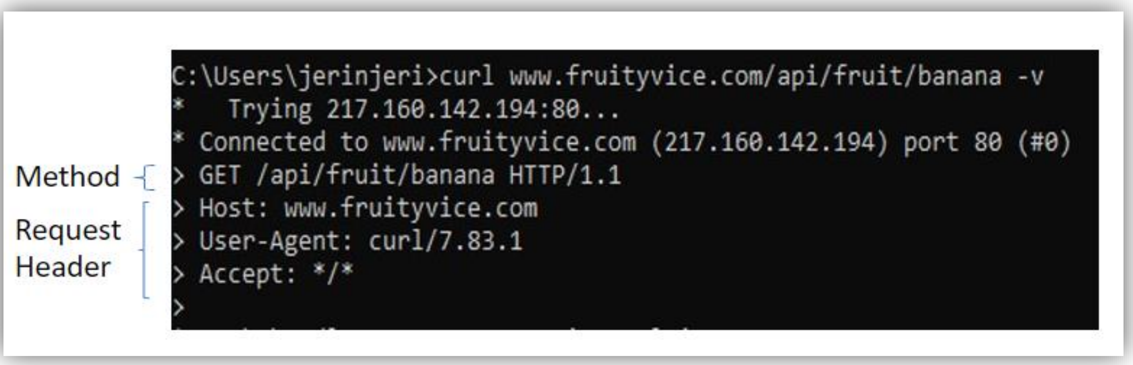
Request Header

The purpose of the request header is to store information relevant to client and server such as authentication (API key, IP address of server, etc.) and content of the body (e.g., response format). The headers are property value pairs separated by a colon. The most common request headers are Host, User Agent, Accept, Connection and Authorization.

Body

The body (also called data or message) contains information that needs to be sent to a server and this option is used in requests other than GET. Since this involves updating or altering a resource, authentication is often involved and it is recommended to follow the API documentation strictly.

Figure 2 shows the structure of a request "www.fruityvice.com/api/fruit/banana" with method and request header explicitly pointed out. Note that we are using the curl (client uniform resource locator) tool and the associated command in the Windows Command Prompt.



```
C:\Users\jerinjeri>curl www.fruityvice.com/api/fruit/banana -v
*   Trying 217.160.142.194:80...
* Connected to www.fruityvice.com (217.160.142.194) port 80 (#0)
> GET /api/fruit/banana HTTP/1.1
> Host: www.fruityvice.com
> User-Agent: curl/7.83.1
> Accept: */*
>
```

Figure 2. Example of a Request Structure

STRUCTURE OF A RESPONSE

Similar to the HTTP request, the HTTP response has a structure which can be easily understood by the client. The HTTP response has three main components:

1. Status Line
2. Response Header
3. Body

Status Line

The status line indicates the status of the request-response transaction and contains three important components: HTTP version, HTTP response code, and a reason phrase. The HTTP version number shows the HTTP specification to which the server has tried to make the response message comply.
The HTTP response code is a three-digit code which shows the final outcome of the request. The codes range from 100 to 599 and are displayed with a reason phrase. In general, the response code and reason phrase follow the rules shown in Table 2. The specific details of the response code and reason phrase can be found at https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.

Status Code	Explanation
1xx	Indicates that the request was received and alerts client to wait for a final response.
2xx	Indicates that the request was successful.
3xx	Indicates that the request was redirected to another URL.
4xx	Indicates an error originating from the client side.
5xx	Indicates an error originated from the server side.

Table 2. General HTTP Status Codes

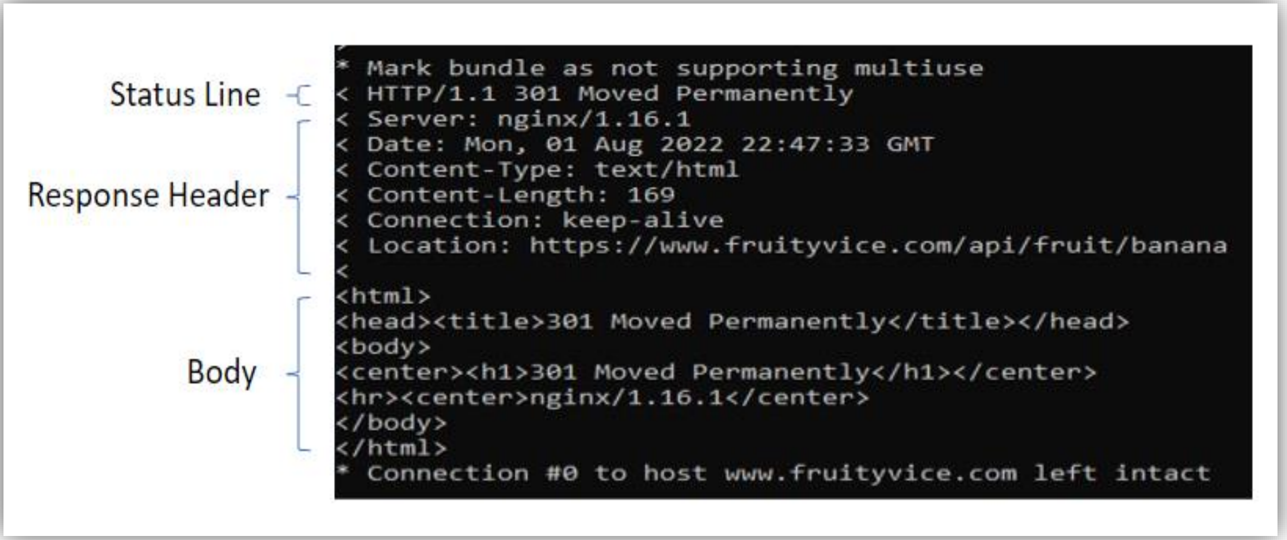
Response Header

The response header contains information about the content that is being returned along with the data about the server. Response headers are property value pairs separated by a colon. The most common response headers are Server, Date, Content-Length, Connection and Location.

Body

The body of the response contains the requested information in case of a successful transaction. The body carries the data and can be in any format (JSON, HTML, etc.) as specified in the header. In unsuccessful transactions, the body can provide further details to complete the transaction. Note that body is optional but is sent most of the time.

An example of the response structure for the request www.fruityvice.com/api/fruit/banana is shown in Figure 3 with all its components. Note that the output is actually from the command used in Figure 2.



```
* Mark bundle as not supporting multiuse
< HTTP/1.1 301 Moved Permanently
< Server: nginx/1.16.1
< Date: Mon, 01 Aug 2022 22:47:33 GMT
< Content-Type: text/html
< Content-Length: 169
< Connection: keep-alive
< Location: https://www.fruityvice.com/api/fruit/banana
<
<html>
<head><title>301 Moved Permanently</title></head>
<body>
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx/1.16.1</center>
</body>
</html>
* Connection #0 to host www.fruityvice.com left intact
```

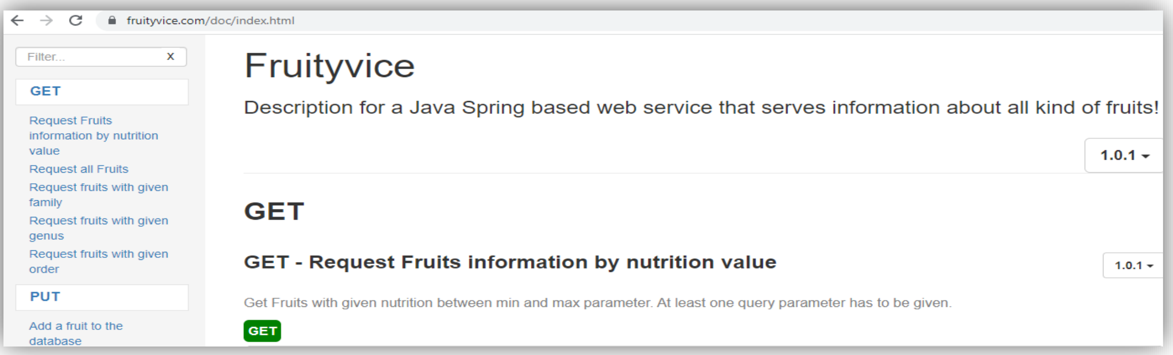
Figure 3. Example of a Response Structure

API REQUESTS

In order to work with API's, we need tools to make requests for consuming data and this can be achieved using methods or procedures available in Python, JavaScript, Ruby, SAS, R, POSTMAN, cURL (Client Uniform Resource Locator), etc. In this article, we will present the details of API requests using Python and SAS for accessing data from two publicly available API's.

API Requests Using Python

A simple example of a REST API will make the above description of API transactions clearer and let us review a publicly available REST API at the web service https://www.fruityvice.com/. This example web service provides educational and interesting information about most fruits. As a professional, it is always recommended to go through the documentation of the API of interest to get a fair idea of its usage. The screenshots of the API documentation for the GET request for this example are presented in Figure 4.


fruityvice.com/doc/index.html
Filter...
GET
Request Fruits information by nutrition value
Request all Fruits
Request fruits with given family
Request fruits with given genus
Request fruits with given order
PUT
Add a fruit to the database
1.0.1
1.0.1
GET

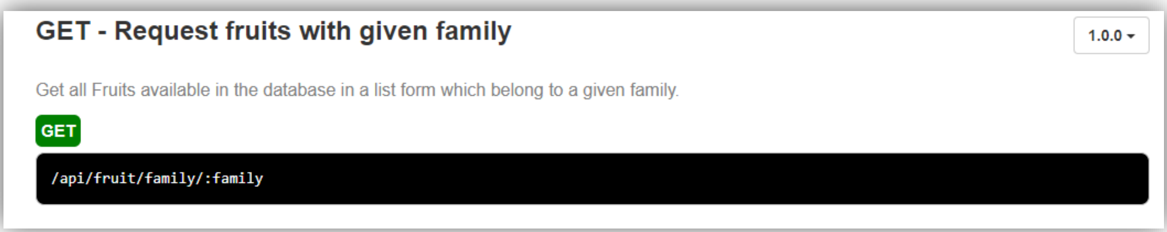
Fruityvice

Description for a Java Spring based web service that serves information about all kind of fruits!

GET

GET - Request Fruits information by nutrition value

Get Fruits with given nutrition between min and max parameter. At least one query parameter has to be given.



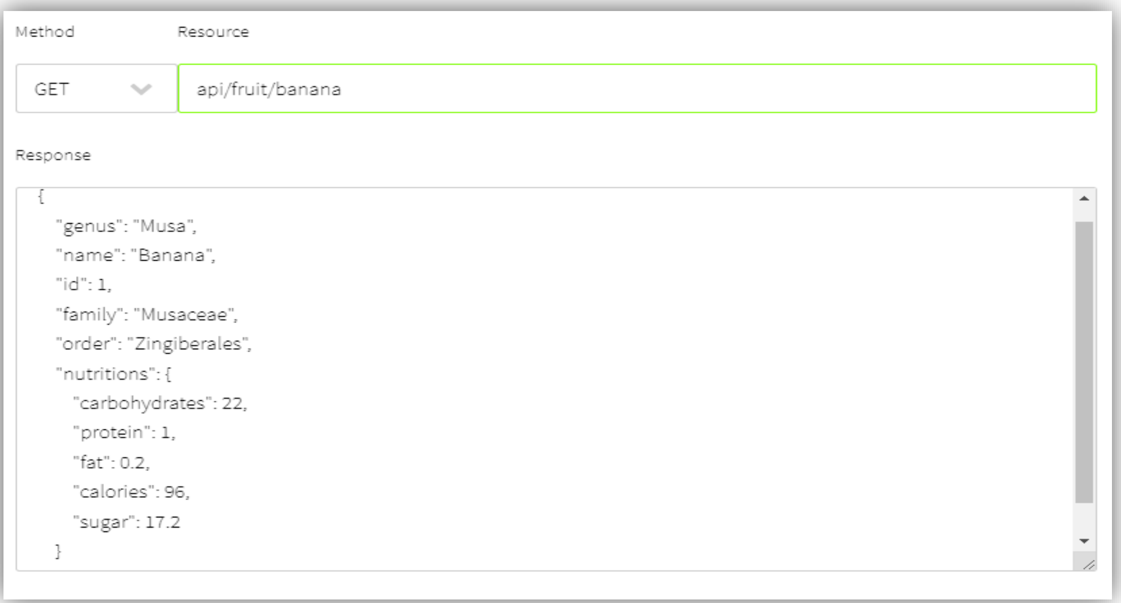
GET - Request fruits with given family

Get all Fruits available in the database in a list form which belong to a given family.

```
/api/fruit/family/:family
```

Figure 4. Screenshots of the API Documentation at https://www.fruityvice.com/doc/index.html

This particular web service already produces an interface where you can try out the requests and is perfect to discern about API for the weary beginners. Figure 5 shows the GET request output for /api/fruit/banana entered in the interface.

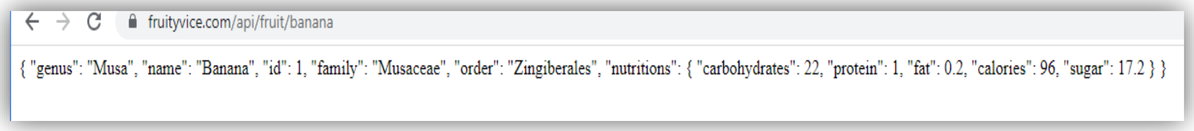


GET api/fruit/banana

```
{
  "genus": "Musa",
  "name": "Banana",
  "id": 1,
  "family": "Musaceae",
  "order": "Zingiberales",
  "nutritions": {
    "carbohydrates": 22,
    "protein": 1,
    "fat": 0.2,
    "calories": 96,
    "sugar": 17.2
  }
```

Figure 5. Output of GET request "api/fruit/banana" in Interface

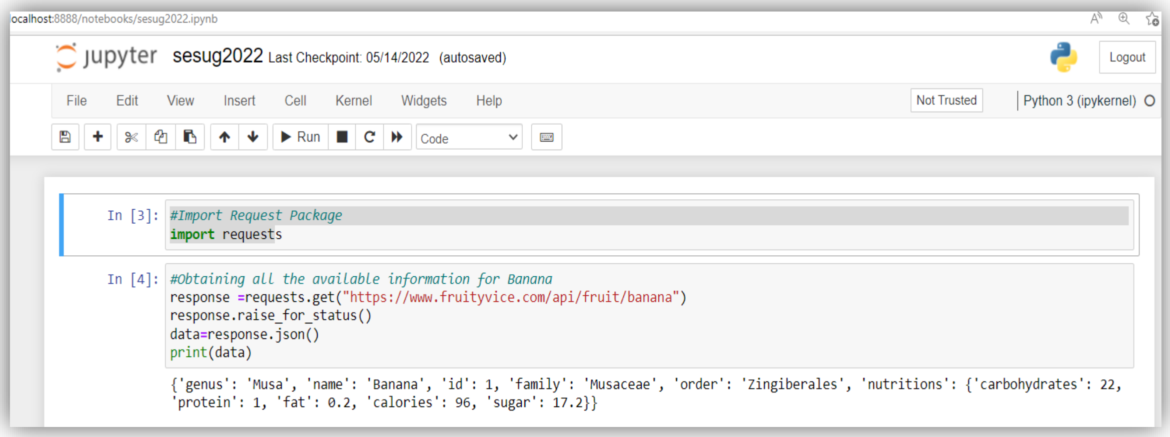
The same GET request entered in the user interface can be passed as a HTTP request in the URL as https://www.fruityvice.com/api/fruit/banana and the corresponding response output in browser is shown in Figure 6. The output is in JSON format which may need to be parsed to extract specific information such as amount of sugar.



```
{ "genus": "Musa", "name": "Banana", "id": 1, "family": "Musaceae", "order": "Zingiberales", "nutritions": { "carbohydrates": 22, "protein": 1, "fat": 0.2, "calories": 96, "sugar": 17.2 } }
```

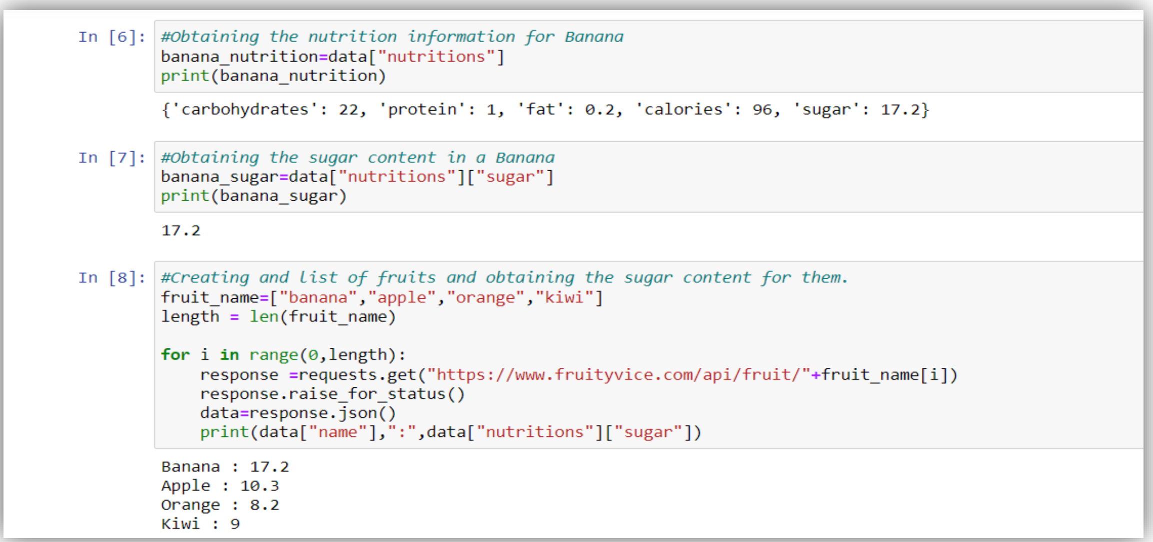
Figure 6. Output of GET request https://www.fruityvice.com/api/fruit/banana in Browser

We can use the above example and extract the data using the Python programming language. The advantage of using a programming language is the flexibility to process the data further and also aids in automating processes. The code snippet to extract using Python is provided in Figures 7 and 8. The REQUESTS package in Python is most widely used for obtaining web information. Figure 7 shows the output using Python for the GET request https://www.fruityvice.com/api/fruit/banana and the output is exactly what was shown in Figures 5 and 6. Figure 8 shows a few examples of how one can extract data depending upon the needs of the user and the comments describe the purpose of each code snippet.


ocalhost:8888/notebooks/sesug2022.ipynb
jupyter sesug2022 Last Checkpoint: 05/14/2022 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3 (ipykernel)
Run
Code

```
In [3]: #Import Request Package
        import requests

In [4]: #Obtaining all the available information for Banana
        response =requests.get("https://www.fruityvice.com/api/fruit/banana")
        response.raise_for_status()
        data=response.json()
        print(data)

        {'genus': 'Musa', 'name': 'Banana', 'id': 1, 'family': 'Musaceae', 'order': 'Zingiberales', 'nutritions': {'carbohydrates': 22,
        'protein': 1, 'fat': 0.2, 'calories': 96, 'sugar': 17.2}}
```

Figure 7. Output of GET request https://www.fruityvice.com/api/fruit/banana using Python



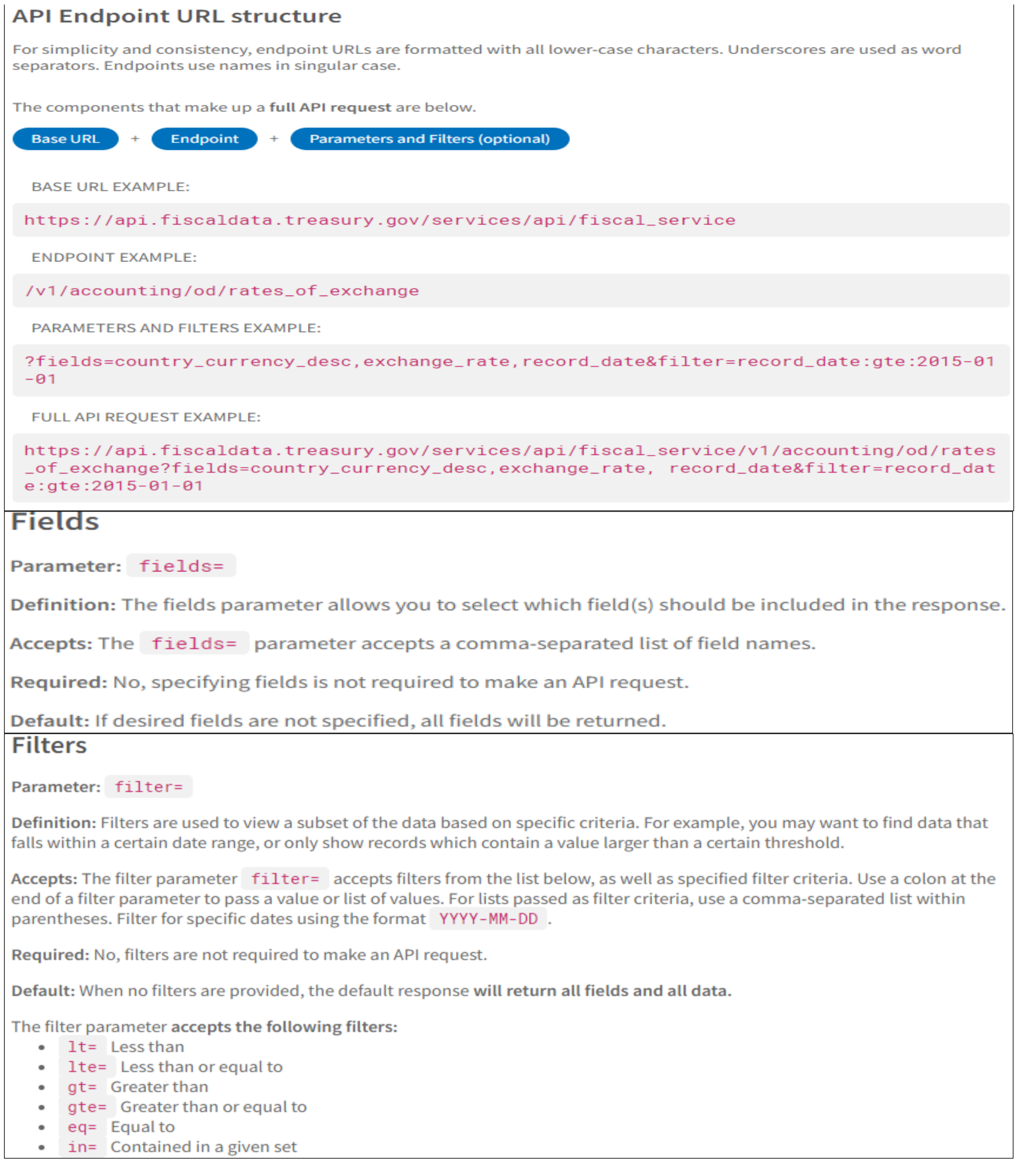
```
In [6]: #Obtaining the nutrition information for Banana
        banana_nutrition=data["nutritions"]
        print(banana_nutrition)

        {'carbohydrates': 22, 'protein': 1, 'fat': 0.2, 'calories': 96, 'sugar': 17.2}

In [7]: #Obtaining the sugar content in a Banana
        banana_sugar=data["nutritions"]["sugar"]
        print(banana_sugar)

        17.2

In [8]: #Creating and list of fruits and obtaining the sugar content for them.
        fruit_name=["banana","apple","orange","kiwi"]
        length = len(fruit_name)

        for i in range(0,length):
            response =requests.get("https://www.fruityvice.com/api/fruit/"+fruit_name[i])
            response.raise_for_status()
            data=response.json()
            print(data["name"],":",data["nutritions"]["sugar"])

        Banana : 17.2
        Apple : 10.3
        Orange : 8.2
        Kiwi : 9
```

Figure 8. Few Examples of Extracting Data Per Users Needs

API Requests Using SAS

An example of a second API is widely accessed in the financial world and the associated documentation is found at https://fiscaldata.treasury.gov/api-documentation/. This documentation is detailed and starts with the basics of API and the concepts of request-response with all the necessary code to tap the consumable data.

Figure 9 is a snapshot of the API Endpoint structure and details of some of the parameters described in the documentation. It is worthwhile to reiterate that review of API documentation and following it strictly will ensure a successful request-response transaction.



API Endpoint URL structure

For simplicity and consistency, endpoint URLs are formatted with all lower-case characters. Underscores are used as word separators. Endpoints use names in singular case.

The components that make up a **full API request** are below.

Base URL + Endpoint + Parameters and Filters (optional)

BASE URL EXAMPLE:

```
https://api.fiscaldata.treasury.gov/services/api/fiscal_service
```

ENDPOINT EXAMPLE:

```
/v1/accounting/od/rates_of_exchange
```

PARAMETERS AND FILTERS EXAMPLE:

```
?fields=country_currency_desc,exchange_rate,record_date&filter=record_date:gte:2015-01
-01
```

FULL API REQUEST EXAMPLE:

```
https://api.fiscaldata.treasury.gov/services/api/fiscal_service/v1/accounting/od/rates
_of_exchange?fields=country_currency_desc,exchange_rate, record_date&filter=record_dat
e:gte:2015-01-01
```

Fields

Parameter: `fields=`

Definition: The fields parameter allows you to select which field(s) should be included in the response.

Accepts: The `fields=` parameter accepts a comma-separated list of field names.

Required: No, specifying fields is not required to make an API request.

Default: If desired fields are not specified, all fields will be returned.

Filters

Parameter: `filter=`

Definition: Filters are used to view a subset of the data based on specific criteria. For example, you may want to find data that falls within a certain date range, or only show records which contain a value larger than a certain threshold.

Accepts: The filter parameter `filter=` accepts filters from the list below, as well as specified filter criteria. Use a colon at the end of a filter parameter to pass a value or list of values. For lists passed as filter criteria, use a comma-separated list within parentheses. Filter for specific dates using the format `YYYY-MM-DD`.

Required: No, filters are not required to make an API request.

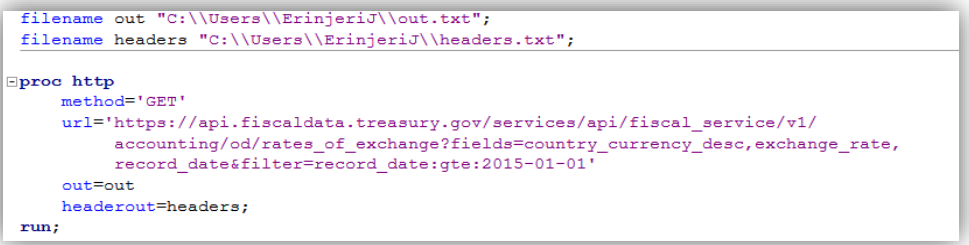
Default: When no filters are provided, the default response **will return all fields and all data.**

The filter parameter **accepts the following filters:**

- `lt=` Less than
- `lte=` Less than or equal to
- `gt=` Greater than
- `gte=` Greater than or equal to
- `eq=` Equal to
- `in=` Contained in a given set

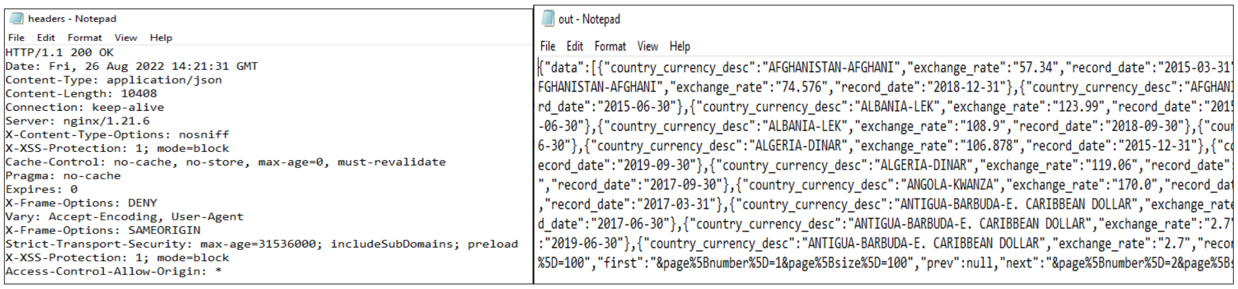
Figure 9. Screenshots of API Documentation at https://fiscaldata.treasury.gov/api-documentation

For this example, we will use SAS®9.4M5 to access the data and the associated code snippet is presented in Figure 10. The FILENAME statement in the code refers to an external file and stores the response output(out.txt) as well as the header information (headers.txt). PROC HTTP uses the METHOD option to input the HTTP method and URL option to enter the endpoint as depicted in Figure 10. Figure 11 shows the snapshot of the content of header and response text files.



```
filename out "C:\\Users\\ErinjeriJ\\out.txt";
filename headers "C:\\Users\\ErinjeriJ\\headers.txt";

proc http
     method='GET'
     url='https://api.fiscaldata.treasury.gov/services/api/fiscal_service/v1/
          accounting/od/rates_of_exchange?fields=country_currency_desc,exchange_rate,
          record_date&filter=record_date:gte:2015-01-01'
     out=out
     headerout=headers;
run;
```

Figure 10. SAS Code Using the GET Method

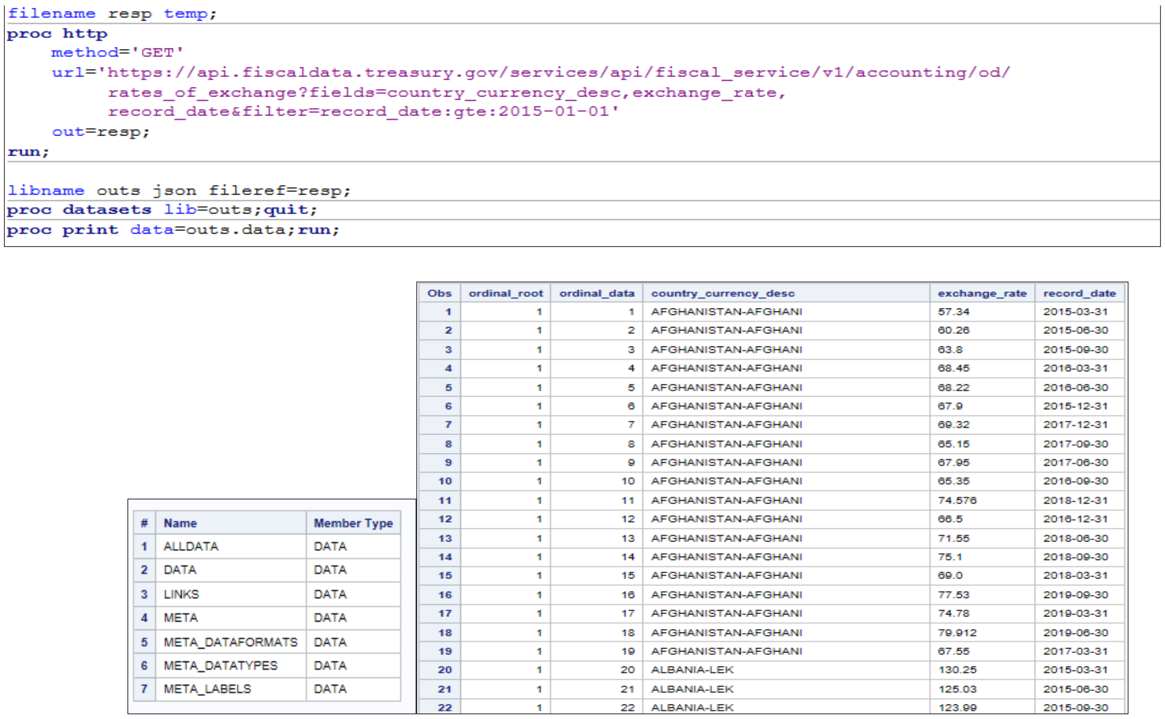


```
HTTP/1.1 200 OK
Date: Fri, 26 Aug 2022 14:21:31 GMT
Content-Type: application/json
Content-Length: 10408
Connection: keep-alive
Server: nginx/1.21.6
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Vary: Accept-Encoding, User-Agent
X-Frame-Options: SAMEORIGIN
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
X-XSS-Protection: 1; mode=block
Access-Control-Allow-Origin: *
```

```
{"data":[{"country_currency_desc":"AFGHANISTAN-AFGHANI","exchange_rate":"57.34","record_date":"2015-03-31
FGHANISTAN-AFGHANI","exchange_rate":"74.576","record_date":"2018-12-31"},{"country_currency_desc":"AFGHANI
rd_date":"2015-06-30"},{"country_currency_desc":"ALBANIA-LEK","exchange_rate":"123.99","record_date":"201
-06-30"},{"country_currency_desc":"ALBANIA-LEK","exchange_rate":"108.9","record_date":"2018-09-30"},{"cou
6-30"},{"country_currency_desc":"ALGERIA-DINAR","exchange_rate":"106.878","record_date":"2015-12-31"},{"c
ecord_date":"2019-09-30"},{"country_currency_desc":"ALGERIA-DINAR","exchange_rate":"119.06","record_date"
","record_date":"2017-09-30"},{"country_currency_desc":"ANGOLA-KWANZA","exchange_rate":"170.0","record_da
,"record_date":"2017-03-31"},{"country_currency_desc":"ANTIGUA-BARBUDA-E. CARIBBEAN DOLLAR","exchange_rat
d_date":"2017-06-30"},{"country_currency_desc":"ANTIGUA-BARBUDA-E. CARIBBEAN DOLLAR","exchange_rate":"2.7
:"2019-06-30"},{"country_currency_desc":"ANTIGUA-BARBUDA-E. CARIBBEAN DOLLAR","exchange_rate":"2.7","reco
%5D=100","first":"&page%5Bnumber%5D=1&page%5Bsize%5D=100","prev":null,"next":"&page%5Bnumber%5D=2&page%5B
```

Figure 11. Snapshot of Header and Response Text Files

We used PROC HTTP to output the data in text files but SAS has options to read in the JSON responses directly into SAS data sets (JSON option in LIBNAME statement) as shown in Figure 12.



```
filename resp temp;
proc http
     method='GET'
     url='https://api.fiscaldata.treasury.gov/services/api/fiscal_service/v1/accounting/od/
          rates_of_exchange?fields=country_currency_desc,exchange_rate,
          record_date&filter=record_date:gte:2015-01-01'
     out=resp;
run;

libname outs json fileref=resp;
proc datasets lib=outs;quit;
proc print data=outs.data;run;
```

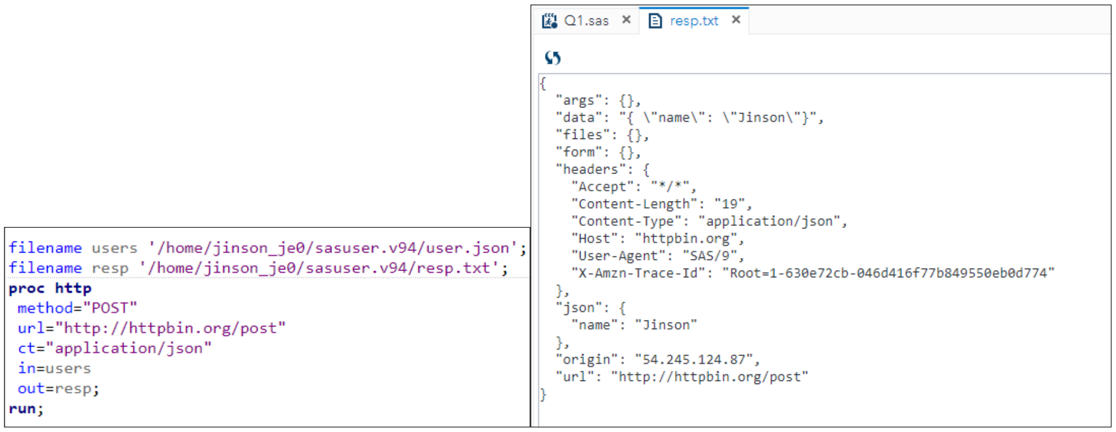
#	Name	Member Type
1	ALLDATA	DATA
2	DATA	DATA
3	LINKS	DATA
4	META	DATA
5	META_DATAFORMATS	DATA
6	META_DATATYPES	DATA
7	META_LABELS	DATA

Obs	ordinal_root	ordinal_data	country_currency_desc	exchange_rate	record_date
1	1	1	AFGHANISTAN-AFGHANI	57.34	2015-03-31
2	1	2	AFGHANISTAN-AFGHANI	60.26	2015-06-30
3	1	3	AFGHANISTAN-AFGHANI	63.8	2015-09-30
4	1	4	AFGHANISTAN-AFGHANI	68.45	2016-03-31
5	1	5	AFGHANISTAN-AFGHANI	68.22	2016-06-30
6	1	6	AFGHANISTAN-AFGHANI	67.9	2015-12-31
7	1	7	AFGHANISTAN-AFGHANI	69.32	2017-12-31
8	1	8	AFGHANISTAN-AFGHANI	65.15	2017-09-30
9	1	9	AFGHANISTAN-AFGHANI	67.95	2017-06-30
10	1	10	AFGHANISTAN-AFGHANI	65.35	2016-09-30
11	1	11	AFGHANISTAN-AFGHANI	74.576	2018-12-31
12	1	12	AFGHANISTAN-AFGHANI	66.5	2016-12-31
13	1	13	AFGHANISTAN-AFGHANI	71.55	2018-06-30
14	1	14	AFGHANISTAN-AFGHANI	75.1	2018-09-30
15	1	15	AFGHANISTAN-AFGHANI	69.0	2018-03-31
16	1	16	AFGHANISTAN-AFGHANI	77.53	2019-09-30
17	1	17	AFGHANISTAN-AFGHANI	74.78	2019-03-31
18	1	18	AFGHANISTAN-AFGHANI	79.912	2019-06-30
19	1	19	AFGHANISTAN-AFGHANI	67.55	2017-03-31
20	1	20	ALBANIA-LEK	130.25	2015-03-31
21	1	21	ALBANIA-LEK	125.03	2015-06-30
22	1	22	ALBANIA-LEK	123.99	2015-09-30

Figure 12. Code Snippet and Output Using PROC HTTP and JSON LIBNAME

POST Method

So far, we have presented all about the GET method which is the most commonly used. Methods other than GET typically need authorizations and for the treasury API example this is not possible at our end. For demonstrating the POST method, we will use the web service http://httpbin.org/post used in the SAS documentation on PROC HTTP. Figure 13 shows the code and the response output using SAS PROC HTTP method and it is important to note the usage of POST as the METHOD option. Also, the data to be posted is a JSON file(user.json) and is referred to using the IN option (users) in PROC HTTP.

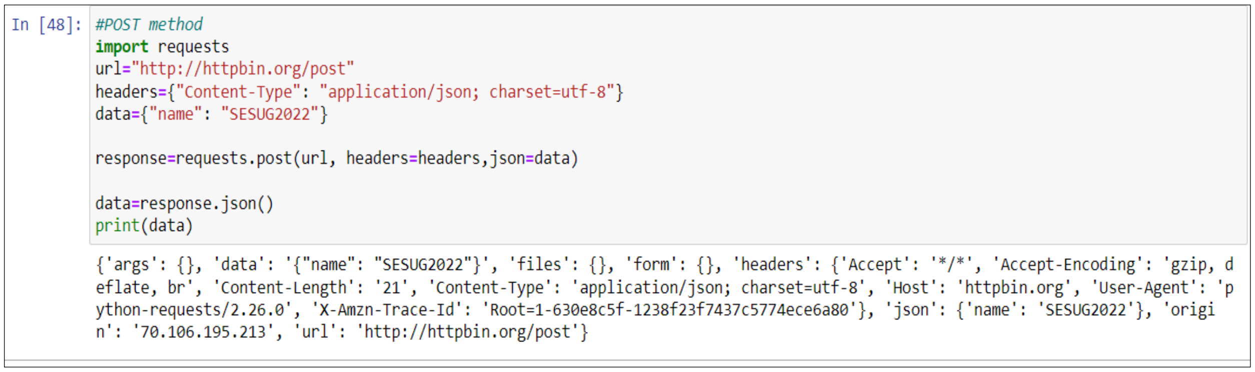


```
filename users '/home/jinson_je0/sasuser.v94/user.json';
filename resp '/home/jinson_je0/sasuser.v94/resp.txt';
proc http
 method="POST"
 url="http://httpbin.org/post"
 ct="application/json"
 in=users
 out=resp;
run;
```

```
{
  "args": {},
  "data": "{ \"name\": \"Jinson\"}",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Content-Length": "19",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "SAS/9",
    "X-Amzn-Trace-Id": "Root=1-630e72cb-046d416f77b849550eb0d774"
  },
  "json": {
    "name": "Jinson"
  },
  "origin": "54.245.124.87",
  "url": "http://httpbin.org/post"
}
```

Figure 13. SAS Code and Output using PROC HTTP for the POST Method

Figure 14 shows the POST method using REQUESTS package in Python. Note that we are using variables url, headers and data as parameters in the POST function.



```
In [48]: #POST method
         import requests
         url="http://httpbin.org/post"
         headers={"Content-Type": "application/json; charset=utf-8"}
         data={"name": "SESUG2022"}

         response=requests.post(url, headers=headers,json=data)

         data=response.json()
         print(data)
```

```
{'args': {}, 'data': '{"name": "SESUG2022"}', 'files': {}, 'form': {}, 'headers': {'Accept': '*/*', 'Accept-Encoding': 'gzip, d
eflate, br', 'Content-Length': '21', 'Content-Type': 'application/json; charset=utf-8', 'Host': 'httpbin.org', 'User-Agent': 'p
ython-requests/2.26.0', 'X-Amzn-Trace-Id': 'Root=1-630e8c5f-1238f23f7437c5774ece6a80'}, 'json': {'name': 'SESUG2022'}, 'origi
n': '70.106.195.213', 'url': 'http://httpbin.org/post'}
```

Figure 14. Python Code and Output using Requests Package for the POST Method

Other API Related Topics

The API examples provided so far did not require authorization and authentication but it is an important aspect of API's and one will stumble on it sooner or later. Most API's use the OAuth (Open Authorization) protocol for authorization where an access token is requested by using the POST method and the access is granted based on this token. For a simple introduction to working with OAuth in SAS please refer to the article https://blogs.sas.com/content/sgf/2020/07/30/curl-to-proc-http/. For trouble shooting purposes, it is recommended to use the DEBUG option in PROC HTTP and details of this can be found at https://go.documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/proc/n0i2ek87s12e7mn1h2q3h0mywkl1.htm. For authorization related information in Python, please refer to www.realpythonproject.com/how-to-authenticate-using-keys-basicauth-oauth-in-python/.

CONCLUSION

It is essential to know the basics of working of API's and this was presented in this paper. The working of API's was supported with couple of publicly available API's using both Python and SAS. To delve further into API authorizations, references were provided in relation to both Python and SAS.

ACKNOWLEDGMENT

The author would like to appreciate Betsy Churchill for reviewing as well providing valuable thoughts in developing this paper.

REFERENCES

Fielding, Roy Thomas (2000). "Chapter 5: Representational State Transfer (REST)". *Architectural Styles and the Design of Network-based Software Architectures* (Ph.D.). University of California, Irvine. Most recently accessed on August 14, 2022.
https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Mozilla, HTTP Response Status Codes. Most recently accessed on August 14, 2022.
https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Real Python, Pythons Request's Library. Most recently accessed on August 14, 2022.
https://realpython.com/python-requests/

SAS Institute, Inc. Data Management and Utility Procedures. HTTP Procedure. Most recently accessed on August 14, 2022.
https://documentation.sas.com/doc/en/vdmmlcdc/8.1/proc/n0bdg5vmrpyi7jn1pbgbje2atoov.htm

SAS Institute, Inc. Bari Lawhorn. How to translate your cURL command into SAS code. Most recently accessed on August 29, 2022.
https://blogs.sas.com/content/sgf/2020/07/30/curl-to-proc-http/

SAS Institute, Inc. Base SAS Procedures Guide. HTTP Procedure. DEBUG Statement. Most recently accessed on August 29, 2022.
https://go.documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/proc/n0i2ek87s12e7mn1h2q3h0mywkl1.htm

How to Authenticate using Keys, BasicAuth, OAuth2 in Python. Most recently accessed on August 29, 2022.
www.realpythonproject.com/how-to-authenticate-using-keys-basicauth-oauth-in-python/

CONTACT INFORMATION

Your comments/questions/criticisms are valued and encouraged. Please contact the author at:

Jinson Erinjeri
Onyx Government Services LLC
5870 Trinity Pkwy Suite 330
Centreville, VA 20120
E-mail: jerinjeri@onyxgs.com