Consuming APIs

John Strickler

Version 1.0, September 2024

Table of Contents

Chapter 1: Consuming RESTful Data

Objectives

- Writing REST clients
  - Opening URLs and downloading data
  - Fetching data from RESTful servers
- Use the requests module to simplify authentication and proxies
- Uploading data to a server

The REST API

- Based on HTTP verbs
  - GET get all objects or details on one
  - POST add a new object
  - PUT update an object (all fields)
  - PATH update an object (some fields)
  - DEL delete an object

REST stands for ***Re***presentational ***S***tate ***T***ransfer, first described (and named) by Roy Fielding in 2000. It is not a protocol or structure, but rather an architectural style resulting from a set of guidelines. It provides for loosely-coupled resource management over HTTP. REST does not enforce a particular implementation.

A RESTful site provides *resources*, which contain *records*. The same API typically contains more than one resource; each has a different *endpoint* (URL). For instance, `https://sandbox-api.brewerydb.com/v2/` has resources **beer**, **brewery**, and **ingredient**.

A RESTful API uses HTTP verbs to manipulate records. The same endpoint can be used for all access; what happens depends on a combination of which HTTP verb is used, plus whether there is more information on the URL.

If it is just the endpoint (e.g. *www.wombats.com/api/v1/wombat*):

- GET retrieves a list of all resources. Query strings can be used to sort or filter the list.
- POST adds a new resource

If it is the endpoint plus more information, typically a primary key (e.g. *www.wombats.com/api/v1/wombat/1*):

- GET retrieves details for that resource
- PUT updates the resource (replaces all fields)
- PATCH updates the resource (replaces some fields)
- DELETE removes the resource

A list of public RESTful APIs is located here: http://www.programmableweb.com/category/all/apis?data_format=21190

Table 1. RESTful requests

Verb	URL	Description
GET	/API/wombat	Get list of all wombats
POST	/API/wombat	Create a new wombat
GET	/API/wombat/*ID*	Get details of wombat by id
PUT	/API/wombat/*ID*	Update wombat by id
DELETE	/API/wombat/*ID*	Delete wombat by id

see https://restfulapi.net/resource-naming/ for more information on designing a RESTful interface

When is REST not REST?

- REST is guidelines, not protocol
- Implementers are not consistent
- YMMV (your mileage may vary)

REST is a set of guidelines, not a specific protocol. Because of this, REST implementations vary widely. For instance, many APIs use more than one endpoint for the same resource. Many APIs do not return a list of links on a GET request to the endpoint, but the details for every resource. Many APIs misuse (from the strict REST point of view) extended URLs.

It is thus important to read the docs for each individual API to see exactly what they expect, and what they provide.

The good news is that the variations are mostly in the URLs you need to construct — making the requests and parsing out the data are generally about the same for most APIs.

Consuming REST APIs

- Use requests.*method*()
- Specify parameters, headers, etc. as dictionary
- Use response.content to get raw data
- Use response.json() to convert JSON to Python

The **requests** module is used to get data from RESTful APIs.

To add GET parameters, pass a dictionary of key-value parameters with the **params** keyword argument:

```
payload = { 'key1': 'value1', ··· }
requests.get(URL, params=payload)
```

Likewise, for POST data, pass a dictionary with the **post** argument.

To use a proxy, pass a dictionary of protocol/url parameters with the proxies keyword

```
proxies={'http':'http://proxy.something.com:1234'}
requests.get(URL, proxies=proxies)
```

proxies={'http':'http://proxy.something.com:1234'}

To convert a JSON response into a Python data structure, use the json() method on the **response** object.

See the quickstart guide for requests at: http://www.python-requests.org/en/latest/user/quickstart/

Example

rest_consumer_omdb.py

```
import requests
from pprint import pprint

with open('omdbapikey.txt') as api_in:
    OMDB_API_KEY = api_in.read().rstrip()

OMDB_URL = "http://www.omdbapi.com"

def main():
    requests_params = {'t': 'Black Panther', "apikey": OMDB_API_KEY}
    response = requests.get(OMDB_URL, params=requests_params)
    if response.status_code == requests.codes.OK:
        raw_data = response.json()

        print(f"raw_data['Title']: {raw_data['Title']}")
        print(f"raw_data['Director']: {raw_data['Director']}")
        print(f"raw_data['Year']: {raw_data['Year']}")
        print(f"raw_data['Runtime']: {raw_data['Runtime']}")
        print()

        print('-' * 60)

        print("raw DATA:")
        pprint(response.json())
    else:
        print(f"response.status_code: {response.status_code}")

if __name__ == '__main__':
    main()
```

rest_consumer_omdb.py

```
raw_data['Title']: Black Panther
raw_data['Director']: Ryan Coogler
raw_data['Year']: 2018
raw_data['Runtime']: 134 min

------------------------------------------------------------
raw DATA:
{'Actors': "Chadwick Boseman, Michael B. Jordan, Lupita Nyong'o",
 'Awards': 'Won 3 Oscars. 124 wins & 290 nominations total',
 'BoxOffice': '$700,426,566',
```

```
 'Country': 'United States',
 'DVD': 'N/A',
 'Director': 'Ryan Coogler',
 'Genre': 'Action, Adventure, Sci-Fi',
 'Language': 'English, Swahili, Nama, Xhosa, Korean',
 'Metascore': '88',
 'Plot': "T'Challa, heir to the hidden but advanced kingdom of Wakanda, must "
        'step forward to lead his people into a new future and must confront '
        "a challenger from his country's past.",
 'Poster': 'https://m.media-
amazon.com/images/M/MV5BMTg1MTY2MjYzNV5BMl5BanBnXkFtZTgwMTc4NTMwNDI@._V1_SX300.jpg',
 'Production': 'N/A',
 'Rated': 'PG-13',
 'Ratings': [{'Source': 'Internet Movie Database', 'Value': '7.3/10'},
             {'Source': 'Rotten Tomatoes', 'Value': '96%'},
             {'Source': 'Metacritic', 'Value': '88/100'}],
 'Released': '16 Feb 2018',
 'Response': 'True',
 'Runtime': '134 min',
 'Title': 'Black Panther',
 'Type': 'movie',
 'Website': 'N/A',
 'Writer': 'Ryan Coogler, Joe Robert Cole, Stan Lee',
 'Year': '2018',
 'imdbID': 'tt1825683',
 'imdbRating': '7.3',
 'imdbVotes': '850,236'}
```

Table 2. Keyword Parameters for **requests** *methods*

Option	Data Type	Description
`allow_redirects`	`bool`	set to True if PUT/POST/DELETE redirect following is allowed
`auth`	`tuple`	authentication pair (user/token,password/key)
`cert`	`str or tuple`	path to cert file or ('cert', 'key') tuple
`cookies`	`dict or CookieJar`	cookies to send with request
`data`	`dict`	parameters for a POST or PUT request
`files`	`dict`	files for multipart upload
`headers`	`dict`	HTTP headers
`json`	`str`	JSON data to send in request body
`params`	`dict`	parameters for a GET request
`proxies`	`dict`	map protocol to proxy URL
`stream`	`bool`	if False, immediately download content
`timeout`	`float or tuple`	timeout in seconds or (connect timeout, read timeout) tuple
`verify`	`bool`	if True, then verify SSL cert

These can be used with any of the HTTP request types, as appropriate.

Table 3. `requests.Response` methods and attributes

Method/attribute	Definition
`apparent_encoding`	Returns the apparent encoding
`close()`	Closes the connection to the server
`content`	Content of the response, in bytes
`cookies`	A CookieJar object with the cookies sent back from the server
`elapsed`	A timedelta object with the time elapsed from sending the request to the arrival of the response
`encoding`	The encoding used to decode r.text
`headers`	A dictionary of response headers
`history`	A list of response objects holding the history of request (url)
`is_permanent_redirect`	True if the response is the permanent redirected url, otherwise False
`is_redirect`	True if the response was redirected, otherwise False
`iter_content()`	Iterates over the response
`iter_lines()`	Iterates over the lines of the response
`json()`	Converts JSON content to Python data structure (if the result was written in JSON format, if not it raises an error)
`links`	The header links
`next`	A PreparedRequest object for the next request in a redirection
`ok`	True if status_code is less than 400, otherwise False
`raise_for_status()`	If an error occur, this method a HTTPError object
`reason`	A text corresponding to the status code
`request`	The request object that requested this response
`status_code`	A number that indicates the status (200 is OK, 404 is Not Found)
`text`	The content of the response, in unicode
`url`	The URL of the response

Printing JSON

- Default output of JSON is ugly
- `pprint` makes structures human friendly
- Use `pprint.pprint()`

When debugging JSON data, the `print` command is not so helpful. The Python data structure parsed from JSON is just printed out all jammed together, one element after another, and is hard to read.

The `pprint` (pretty print) module will analyze a structure and print it out with indenting, to make it much easier to read.

You can customize the output with some named parameters: `indent` (default 1) specifies how many spaces to indent nested structures; `width` (default 80) constrains the width of the output; `depth` (default unlimited) says how many levels to print – levels beyond depth are show with '...'.

Requests sessions

- Share configuration across requests
- Can be faster
- Instance of requests.Session
- Supports context manager (*with* statement)

To make it more convenient to share HTTP information across multiple requests, **requests** provides *sessions*. You can use a session by creating an instance of **requests.Session**. Once the session is created, it contains attributes containing the named arguments to request methods, such as *session*.params or *session*.headers.

A Session object implements the context manager protocol, so it can be used with the **with** statement. This will automatically close the session.

These are normal Python dictionaries, so you can use the *update* method to add information:

```
with requests.Session() as session:
    session.params.update({'token': 'MY_TOKEN'})
    session.headers.update({'accept': 'application/xml'})

    response1 = session.get(...)
    response2 = session.get(...)
```

Then you can call the HTTP methods (**get**, **post**, etc.) from the session object. Any parameters passed to an HTTP method will overwrite those that already exist in the session, but do not update the session.

For more examples of using REST in a real-world situation, see `consume_omdb_main.py` in the `EXAMPLES` folder. This script imports various modules that connect to the OMDB API using various multitasking approaches.

Example

rest_consumer_omdb_sessions.py

```
import requests
from pprint import pprint

with open('omdbapikey.txt') as api_in:
    OMDB_API_KEY = api_in.read().rstrip()


OMDB_URL = "http://www.omdbapi.com"

MOVIE_TITLES = [
    'Black Panther',
    'Frozen',
    'Top Gun: Maverick',
    'Bullet Train',
    'Death on the Nile',
    'Casablanca',
]

def main():
    with requests.Session() as session:
        session.params.update({"apikey": OMDB_API_KEY})
        for movie_title in MOVIE_TITLES:
            params = {'t': movie_title}
            response = session.get(OMDB_URL, params=params)
            if response.status_code == requests.codes.OK:
                raw_data = response.json()
                print(f"raw_data['Title']: {raw_data['Title']}")
                print(f"raw_data['Director']: {raw_data['Director']}")
                print(f"raw_data['Year']: {raw_data['Year']}")
                print(f"raw_data['Runtime']: {raw_data['Runtime']}")
                print()



if __name__ == '__main__':
    main()
```

rest_consumer_omdb_sessions.py

```
raw_data['Title']: Black Panther
raw_data['Director']: Ryan Coogler
raw_data['Year']: 2018
raw_data['Runtime']: 134 min

raw_data['Title']: Frozen
raw_data['Director']: Chris Buck, Jennifer Lee
raw_data['Year']: 2013
raw_data['Runtime']: 102 min

raw_data['Title']: Top Gun: Maverick
raw_data['Director']: Joseph Kosinski
raw_data['Year']: 2022
raw_data['Runtime']: 130 min

raw_data['Title']: Bullet Train
raw_data['Director']: David Leitch
raw_data['Year']: 2022
raw_data['Runtime']: 127 min

raw_data['Title']: Death on the Nile
raw_data['Director']: Kenneth Branagh
raw_data['Year']: 2022
raw_data['Runtime']: 127 min

raw_data['Title']: Casablanca
raw_data['Director']: Michael Curtiz
raw_data['Year']: 1942
raw_data['Runtime']: 102 min
```

Authentication with requests

- Options
  - Basic-Auth
  - Digest
  - Custom
- Use **auth** argument

requests makes it easy to provide basic authentication to a web site.

In the simplest case, create a `requests.auth.HTTPBasicAuth` object with the username and password, then pass that to requests with the `auth` argument. Since this is a common use case, you can also just pass a `(user, password)` tuple to the `auth` parameter.

For digest authentication, use `requests.auth.HTTPDigestAuth` with the username and password.

For custom authentication, you can create your own auth class by inheriting from `requests.auth.AuthBase`.

For OAuth 1, OAuth 2, and OpenID, install `requests-oauthlib`. This additional module provides auth objects that can be passed in with the `auth` parameter, as above.

See https://docs.python-requests.org/en/latest/user/authentication/ for more details.

Example

http_basic_auth.py

```
import requests
from requests.auth import HTTPBasicAuth, HTTPDigestAuth

# base URL for httpbin
BASE_URL = 'https://httpbin.org'

# formats for httpbin
BASIC_AUTH_FMT = "/basic-auth/{}/{}"
DIGEST_AUTH_FMT = "/digest-auth/{}/{}/{}"

USERNAME = "spam"
PASSWORD = "ham"
BAD_PASSWORD = "toast"

REPORT_FMT = "{:35s} {}"

def main():
    basic_auth()
    digest()

def basic_auth():
    auth = HTTPBasicAuth(USERNAME, PASSWORD)
    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Basic auth good password", response))

    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, PASSWORD),
        auth=(USERNAME, PASSWORD),
    )
    print(REPORT_FMT.format("Basic auth good password (shortcut)", response))

    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, BAD_PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Basic auth bad password", response))

def digest():
    auth = HTTPDigestAuth(USERNAME, PASSWORD)
    response = requests.get(
```

```
        BASE_URL + DIGEST_AUTH_FMT.format('WOMBAT', USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Digest auth good password", response))

    auth = HTTPDigestAuth(USERNAME, BAD_PASSWORD)
    response = requests.get(
        BASE_URL + DIGEST_AUTH_FMT.format('WOMBAT', USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Digest auth bad password", response))


if __name__ == '__main__':
    main()
```

http_basic_auth.py

```
Basic auth good password            <Response [200]>
Basic auth good password (shortcut) <Response [200]>
Basic auth bad password             <Response [401]>
Digest auth good password           <Response [200]>
Digest auth bad password            <Response [401]>
```

Posting data to a RESTful server

- use requests.post(url, data=dict)

The **POST** operation adds a new record to a resource using the endpoint.

To post to a RESTful service, use the **data** argument to **request**'s **post** function. It takes a dictionary of parameters, which will be URL-encoded automatically.

If the POST is successful, the server should return response data with a link to the newly created record, with an HTTP response code of 201 ("created") rather than 200 ("OK").

Example

post_to_rest.py

```
from datetime import datetime
import time
import requests

URL = 'http://httpbin.org/post'

for i in range(3):
    response = requests.post(  # POST data to server
        URL,
        data={'date': datetime.now(),
            'label': 'test_' + str(i)
        },
        cookies={'python': 'testing'},
        headers={'X-Python': 'Guido van Rossum'},
    )
    if response.status_code in (requests.codes.OK, requests.codes.created):
        print(response.status_code)
        print(response.text)
        print()
        time.sleep(2)
```

post_to_rest.py

```
200
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "date": "2024-09-18 12:47:29.722850",
    "label": "test_0"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "48",
    "Content-Type": "application/x-www-form-urlencoded",
    "Cookie": "python=testing",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.31.0",
    "X-Amzn-Trace-Id": "Root=1-66eb0421-5af228a27093db8613eb8a64",
    "X-Python": "Guido van Rossum"
  },
  "json": null,
  "origin": "69.218.218.146",
  "url": "http://httpbin.org/post"
}


200
{
  "args": {},
  "data": "",
```

...

Other operations

- Use **requests** functions
  - put
  - delete
  - patch
  - head
  - options

For other HTTP operations, **requests** provides appropriately named functions.

Example

other_web_service_ops.py

```
import requests

print('PUT:')
r = requests.put("http://httpbin.org/put", data={'spam': 'ham'}) # send data via HTTP PUT
request
print(r.status_code, r.text)
print('-' * 60)

print('DELETE:')
r = requests.delete("http://httpbin.org/delete") # send HTTP DELETE request
print(r.status_code, r.text)
print('-' * 60)

print('HEAD:')
r = requests.head("http://httpbin.org/get") # get HTTP headers via HEAD request
print(r.status_code, r.text)
print(r.headers)
print('-' * 60)

print('OPTIONS:')
r = requests.options("http://httpbin.org/get") # get negotiated HTTP options
print(r.status_code, r.text)
print('-' * 60)
```

other_web_service_ops.py

```
PUT:
200 {
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "spam": "ham"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "8",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.31.0",
    "X-Amzn-Trace-Id": "Root=1-66eb0429-69d2065c64d4f33f5dbec1ea"
  },
  "json": null,
  "origin": "69.218.218.146",
  "url": "http://httpbin.org/put"
}

------------------------------------------------------------
DELETE:
200 {
  "args": {},
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "0",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.31.0",
    "X-Amzn-Trace-Id": "Root=1-66eb0429-4082daa71cfbf20d0074a797"
  },
  "json": null,
  "origin": "69.218.218.146",
  "url": "http://httpbin.org/delete"
}

------------------------------------------------------------
HEAD:
200
```

```
{'Date': 'Wed, 18 Sep 2024 16:47:37 GMT', 'Content-Type': 'application/json', 'Content-
Length': '307', 'Connection': 'keep-alive', 'Server': 'gunicorn/19.9.0', 'Access-Control-
Allow-Origin': '*', 'Access-Control-Allow-Credentials': 'true'}
------------------------------------------------------------
OPTIONS:
200
------------------------------------------------------------
```

Using Postman

- Graphical HTTP Client
- Specify headers and parameters
- Save searches

postman is a GUI-based HTTP client. It lets you specify URLs, plus headers, parameters, data, and any other information that needs to be sent to a web server. You can save searches for replay, and easily make changes to the requests.

This makes it really easy to experiment with REST endpoints without having to constantly re-run your Python scripts.

It also displays the JSON (or other) response in various modes, including pretty-printed with syntax highlighting.

postman is free for individuals and small teams, or can be subscribed to for larger teams

Get **postman** at https://www.getpostman.com/

Chapter 1 Exercises

Exercise 1-1 (noaa_precip.py)

NOAA provides an API for climate data. The base page is https://www.ncdc.noaa.gov/cdo-web/webservices/v2#gettingStarted.

You will need to request a token via email. Specify the token with the `token` header. Use the `headers` parameter to requests.get().

You can get a list of the web services as described on the page.

For hourly precipitation, the dataset ID is "PRECIP_HLY".

The station ID for Boaz, Alabama is "COOP:010957"

Using your token, the dataset ID, and the station ID, fetch hourly precipitation data for that location from January 1, 1970 through January 31, 1970.

The endpoint is

https://www.ncdc.noaa.gov/cdo-web/api/v2/data

and you will need to specify the parameters **datasetid**, **stationid**, **startdate**, and **enddate**.

Remember that only a certain number of values are returned with each call. For purposes of this lab, just get the first page of values.

Try varying the parameters, based on the API docs.

See ANSWERS/noaa_metadata.py for a script to retrieve values for some of the endpoints.

Index