TECHNION
Azrieli Continuing Education and External Studies Division

Python - Requests Library

Notebook

Introduction to HTTP

HTTP is a protocol for fetching resources such as HTML documents. It is the foundation of any data exchange on the Web, and it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser. A complete document is reconstructed from the different sub-documents fetched, for instance, text, layout description, images, videos, scripts, and more.

Clients and servers communicate by exchanging individual messages (as opposed to a stream of data). The messages sent by the client, usually a Web browser, are called **requests** and the messages sent by the server as an answer are called **responses**.

Designed in the early 1990s, HTTP is an extensible protocol which has evolved over time. It is an application layer protocol that is sent over TCP, or over a TLS-encrypted TCP connection, though any reliable transport protocol could theoretically be used.

Due to its extensibility, it is used to not only fetch hypertext documents, but also images and videos or to post content to servers, like with HTML form results. HTTP can also be used to fetch parts of documents to update Web pages on demand.

- **HTTP is simple** - HTTP is generally designed to be simple and human readable, even with the added complexity introduced in HTTP/2 by encapsulating HTTP messages into frames. HTTP messages can be read and understood by humans, providing easier testing for developers, and reduced complexity for newcomers.

- **HTTP is extensible** - Introduced in HTTP/1.0, HTTP headers make this protocol easy to extend and experiment with. New functionality can even be introduced by a simple agreement between a client and a server about a new header's semantics.

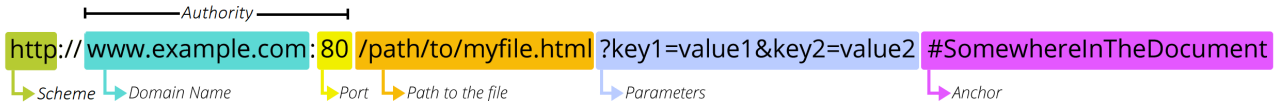
- **HTTP is stateless, but not session-less** - HTTP is stateless, there is no link between two requests being successively carried out on the same connection. This immediately has the prospect of being problematic for users attempting to interact with certain pages coherently, for example, using e-commerce shopping baskets. But while the core of HTTP itself is stateless, HTTP cookies allow the use of stateful sessions. Using header extensibility, HTTP Cookies are added to the workflow, allowing session creation on each HTTP request to share the same context, or the same state.

Thanks to Mozilla MDN for the great overview above, there is no better way to explain a thing that explained by the new HTTP standard owners. What is more interesting in this point is just a certain definition to allow us using python as a web browser.

Requests library become kind of a new standard of how to integrate python code with a web-site request response system.

Understanding URLs

URL stands for Uniform Resource Locator. A URL is nothing more than the address of a given unique resource on the Web.



The common sense led us to understand only the “Domain Name” of website - when we asked to surf to some web using the web-browser all we do with our keyboard is enter the domain name, but as you can see above it’s a little bit more complicated by structure.

There is a URI as part of the URL, and it’s defined by what you see above after the “Domain Name:Port”. URI is a unique sequence of characters that identifies a resource. Basically, we use URL terminology to describe the whole line but URI only the resources requested.

For example:
https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

URI in the above example is “/wiki/Uniform_Resource_Identifier”. Its critical to understand it to work properly with requests library, commonly we will use the base of the URL (the domain or the IP address) and append it the URI during the process.

HTTP Flow

When a client wants to communicate with a server, either the final server or an intermediate proxy, it performs the following steps:

1. Open a TCP connection: The TCP connection is used to send a request, or several, and receive an answer. The client may open a new connection, reuse an existing connection, or open several TCP connections to the servers.

2. Send an HTTP message: HTTP messages (before HTTP/2) are human-readable. With HTTP/2, these simple messages are encapsulated in frames, making them impossible to read directly, but the principle remains the same. For example:

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

3. Read the response sent by the server, such as:

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
```

Content-Length: 29769
Content-Type: text/html

<!DOCTYPE html... (here come the 29769 bytes of the requested web page)

4. Close or reuse the connection for further requests.

Working with requests could be complicated without seeing the above, a lot of us expect to get some beautiful text but its not the case.

The response that should be expected will give you a disassembled parts of the response - you will be able to show only the **response code**, **data**, **headers,** or everything together to give us the most control on the communication.

Working with Requests module

The **requests** library built-in Kali distro, however in other systems you will need to install it. The installation happened basically through the **pip** module.

HTTP GET request are used to retrieve data from a specified resource.

RFC2616: *"The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process."*

We will use GET request to render a list of data, mostly an API response encoded into JSON format or HTML formatted data. The retrieved data format is determined by the HTTP response headers. Another GET request done to filter based on query string, we will discuss it in a few moments.

For this notebook journey we will use the "httpbin.org" API - it's a special platform build to test various HTTP operations like methods, authentications and much more.

Making a simple GET request:

```
#!/usr/bin/python3

import requests

url = "https://httpbin.org/get"
r = requests.get(url)
print(r.url)

# Output:
https://httpbin.org/get
```

The **request.get()** and other methods used to request resources are an object. Our response object is **r**.

As mentioned before, the URL specification are critical to be understood. In the example above we use the next structure:

The response object (in our case its variable **r**) could retrieve different data, look at the next examples of possible data that could be very useful:

```
#!/usr/bin/python3

import requests

url = "https://httpbin.org/get"
r = requests.get(url)
print(f'The URL passed:\n {r.url}')
print(f'The status code:\n {r.status_code}')
print(f'The data (encoded as original):\n {r.content}')
print(f'The data (decoded as text):\n {r.text}')

'''
Output:

The URL passed:
 https://httpbin.org/get

The status code:
 200

The data (encoded as original):
 b'{\n  "args": {}, \n  "headers": {\n    "Accept": "*/*", \n    "Accept-
Encoding": "gzip, deflate", \n    "Host": "httpbin.org", \n    "User-
Agent": "python-requests/2.25.1", \n    "X-Amzn-Trace-Id": "Root=1-6208c92c-
39f1a1ea6b410db0359b30a5"\n  }, \n  "origin": "85.65.244.58", \n  "url": "ht
tps://httpbin.org/get"\n}\n'

The data (decoded as text):
 {
  "args": {},
  "headers": {
    "Accept": "*/*",
```

```
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.25.1",
    "X-Amzn-Trace-Id": "Root=1-6208c92c-39f1a1ea6b410db0359b30a5"
  },
  "origin": "85.65.244.58",
  "url": "https://httpbin.org/get"
}
'''
```

In GET request we could specify parameters (if supported by the website) using the **params** addon on the request line.

- **Parameter names**
- **Values**

The httpbin API allow us to test whenever parameters and values we would like, in real-world situation we will retrieve the parameters from the website itself in certain pages.

Parameters is the basic way of a website to get an input from the client and pass it to the server-side logic. In some web-attacks we could even guess them.

Parameters should be a **key:value** pair (dictionary type) to be passed as expected and the requests library will take care about to encode them to make it proper.

```
#!/usr/bin/python3

import requests

url = "https://httpbin.org/get"
parameters = {'firstName':'John','lastName':'Smith'}
r = requests.get(url, params=parameters)
print(f'The URL passed:\n {r.url}')
print(f'The data (decoded as text):\n {r.text}')

''' Output:
The URL passed:
 https://httpbin.org/get?firstName=John&lastName=Smith
The data (decoded as text):
 {
  "args": {
    "firstName": "John",
    "lastName": "Smith"
  }, [..snip..]
'''
```

HTTP POST request used most often when submitting data from a form or uploading a file. When we POST data, it's not reflected on the URL as in GET request and the data passed as part of the body. Mostly the data encoded into JSON and for these we could specify both **data** and **json** equal to the payload we would like to send.

The difference between the specification of how the data transmitted is that when we use **json** as payload pass it will append a header called **content-type** with value of **application/json** and in the other hand **data** as payload pass will not specify it - it could be critical in certain situations.

```
#!/usr/bin/python3

import requests

url = "https://httpbin.org/post"
payload = {'firstName':'John','lastName':'Smith'}
r = requests.post(url, json=payload)
print(f'The URL passed:\n {r.url}')
print(f'The data (decoded as text):\n {r.text}')

'''Output:
The URL passed:
 https://httpbin.org/post

The data (decoded as text):
 {
  "args": {},
  "data": "{\"firstName\": \"John\", \"lastName\": \"Smith\"}",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "42",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.25.1",
    "X-Amzn-Trace-Id": "Root=1-6208cddf-4d20e3cb131bcf7721d177cd"
  },
  "json": {
    "firstName": "John",
    "lastName": "Smith"
  },
  "origin": "85.65.244.58",
  "url": "https://httpbin.org/post"
}
'''
```

Why we learn it?

Using python for web-attacks its just another option in the cyber security area, when you code your own tools, you gain much more understanding about the process and could find it sometimes challenging but rewarding.

Using built-in or prebuilt tools aren't a mistake and if you master them its even a great deal, but when you are get a certain level of knowledge and you play along with sites and web attacks you would like to take it to the other level by writing your own tools - the requests library makes it so simple that sometimes I prefer to write 5 lines of code then start investigate the available tools and one of the another great pros of this is that you could output what you like and not what hardcoded in a tool.

Possible projects

Wow, there are so many things you could do… for this point maybe you should just work with it to understand better the certain requests and how HTTP working but when you will get at least the basics of the common web-attacks and methodology you will be able to upgrade your skills.

- Website enumeration tool
- Web scraper
- Web spider
- Testing for logins and passwords
- Testing for possible injections
- Testing website behavior
- Guessing hidden content like files or folders or even backups
- Web proxy