Week 09

How Does The Internet Work? Python, cURL and a REST API

Melvyn Ian Drag

October 29, 2019

Abstract

This Lecture is preamble to the next one in which we will set up an apache webserver. We'll take a peek at Python, revisit cURL, and make some web requests.

1 Exam Make Up

Folks who missed the exam last week can come early and work until 7:15 to get it done. Lecture starts promptly at 7:15.

2 Python Webserver

2.1 What is Python?

Python is a scripting language kind of like bash or javascript or perl - but different. It's the favorite language of legions of programmers. Python is used everywhere from scientific computing - I used it to solve PDEs in grad school - to game development, to system administration, to excel spreadsheet modifying, to a million other use cases. Today we're going to look at Python as a language for making websites and serving webpages.

2.2 What is a webserver?

As we discussed before, when type a website into a browser (like Chrome or Firefox, or the like) and hit *ENTER*, you ask a webserver for some information. The webserver responds with the information you requested and you see a webpage. This type of transaction is called a **GET** request when performed over the **Hyper Text Transmission Protocol (HTTP)** or the related **HTTPS**, which is the secure flavor of HTTP. We'll discuss what we mean by 'secure' in a few weeks.

Another thing you do when talking to a website is give some data to a webserver. This happens when you type your First Name, Last Name, User Name, phone number, password, etc. into a form on a website and click *Submit*. When you click the button, the browser sends a **POST** request to the webserver that is listening to you from some other corner of the earth.

In the case of either a GET or POST request, the request goes to a webserver, which is a computer far far away from you that has some code running on it that will respond to your GETs and your POSTs. That code is very often written in Java. It is frequently written in Python. Many write that code in Ruby. Many others use Scala to write the web server code. Recently, people started using Javascript to write the server code.

In this class we will write the server code in Python.

2.3 Install Python 3.

Install it with

```
user@machine$ sudo apt install python3-dev
```

2.4 Don't install Python 2

Python 2 is being deprecated next year. The Python language maintainers have been telling the Python community to move to Python3 for about 10 years now. If you are new to Python, use Python 3. If you are old to Python, hurry up and port your code to Python 3!

2.5 I Hate This Lecture Because This is a Linux Class, Not a Python Class

Why are you wasting time teaching Python in a Linux class!!!!?!?!? I didn't sign up for this!!

We're using Python in this class for this webserver activity because I think it is hand's down the easiest language to read and write. Since I don't have time to teach you Python right now, I'm just using Python code because you can more or less skim it and understand what it does. We're only using Python for about the next three hours, so I'm just going to give you the bare minimum information needed to understand what's happening.

One thing that is important to understand is that Python is a whitespace sensitive language. Whereas other languages use curly braces to denote logical blocks in code - for example, ask class where the *main* and *sayHi* functions , and the *MyClass* class, start and ends in this Java program:

```java
public class MyClass{
  public static void sayHi()[
    System.out.println("Hi");
  }

  public static void main(String[] args){
    sayHi();
  }
}
```

In python, the situation is different:

```python
def sayHi():
  s = "Hi"
  print(s)

if __name__ == "__main__":
  sayHi()
```

Note the lack of curly braces. Write this code on your machine and run it as;

```
user@machine$ python3 whateverYouNamedTheFile.py
```

Then, change the space before the 's' to 4 spaces and the space before the 'print' to a tab. Try to run the code again and it will fail.

They will look the same, but they are not the same on the byte level. more on that at the end of lecture if there is time.

Python expects either all tabs or all spaces. If you mix tabs and spaces in your code, Python get's mad.

As I've said before, there is alot of bickering in the tech communty about everything. Some people hate python becuase of the whitespace `https://news.ycombinator.com/item?id=1463451`

I really like Python. I think it's the easiest language to write, there are many great libraries for math, science, system administration, web development, graphics, etc.. The community is great! There are conferences all year long all around the world. And I like not having to wast space in my code with curly braces. Different strokes.

TL;DR You should learn Python. Not now - you're too busy. Next year you should learn Python.

2.6 Simple Python Webserver

Okay, so now that you have Python installed, we can set up a simple web server and begin to make some HTTP GET and SET requests!

Here is the code:

```python
"""
Code stolen here:

```

```
https://gist.github.com/bradmontgomery/2219997#file-dummy-web-server-py

and then modified to be more directly relatable to our lecture.
"""

from http.server import HTTPServer, BaseHTTPRequestHandler

class MyFirstServer(BaseHTTPRequestHandler):
    def _set_headers(self):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()

    def _html(self, message):
        content =
            "<html>\n\t<body>\n\t\t<h1>{}</h1>\n\t</body>\n</html>\n".format(message)
        return content.encode("utf8") # NOTE: must return a bytes object!

    def do_GET(self):
        self._set_headers()
        self.wfile.write(self._html("Received GET!"))

    def do_POST(self):
        content_length = int(self.headers['Content-Length'])
        post_data = self.rfile.read(content_length)

        self._set_headers()
        self.wfile.write(self._html("Received POST!" + post_data.decode("utf8")))

if __name__ == "__main__":
    addr = "127.0.0.1"
    #addr = "localhost"
    port = 8000
    server_address = (addr, port)
    httpd = HTTPServer(server_address, MyFirstServer)

    print("Starting httpd server on {}:{}".format(addr, port))
    httpd.serve_forever()
```

This code is available in the class repository under the name *Server.py*. From just this simple little example, we can learn alot about how the internet works. As we survey the example, we can issue some curl requests and verify that they work.

We can run this code from the command line with:

```
user@machine$ python3 Server.py
Starting httpd server on localhost:8000
```

So we see that our little baby website is running on ”localhost” at port 8000. Standard HTTP applications run on port 80, but for this demo app we are running on port 8000. Just for reference - HTTPS applications run (by default) on port 443. And ssh is over port 22. And postgresql is on port 5432. So now you know 4 special ports!

A summary of what we know about ports so far:

- port 22 - ssh
- port 80 - HTTP
- port 443 - HTTPS
- port 5432 - PostgreSQL

In the webdevelopment world, we make **GET** and **POST** requests to webservers that have IP Addresses. But we rarely refer to them by their ip address - typically we use a domain name like ‘www.google.com’ instead of a number address. We are doing the same thing here - we name our local webserver ”localhost” instead of referring to it by its numeric name - 127.0.0.1

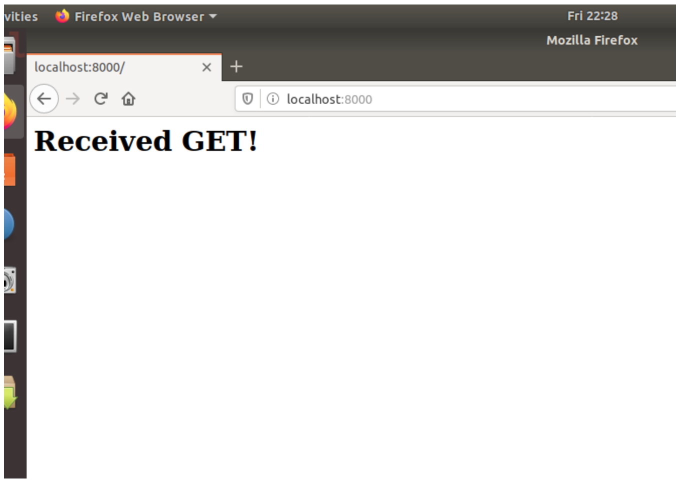
2.7 A quick look at the code

Note that there are methods for handling **POST** and **GET** requests. This is characteristic of code running on a webserver. The code will have methods for handling different types of HTTP requests. You'll also see a few lines of code for setting up an HTTP server to run on 'localhost:8000'. Theres a line that does something or other with "headers", and we'll look into those in a minute. And there's a function that does something with html. If you've never seen HTML before, that's what it looks like. It's a markup language. If you don't know what HTML is, you can learn it in just a few minutes, it's pretty simple. It structures a document between some brackets. HTML always looks something like this:

```html
<html>
<body>
  <h1> This is a title </h1>
  <h2> This is a subtitle </h2>
  <div> This is a paragraph about something </div>
</body>
</html>
```

2.8 Verify Website Works With A Browser

To verify that our website is running, open a broswer window (chrome, firefox, etc) and type "localhost:8000". You should get a response like this:



Received GET!

Figure 1: GET request in browser to localhost:8000

So we know our website "localhost:8000" is running.

And as I mentioned before, "localhost" is an alias for "127.0.0.1", which we can readily verify by going to "127.0.0.1:8000" while our little application is running.

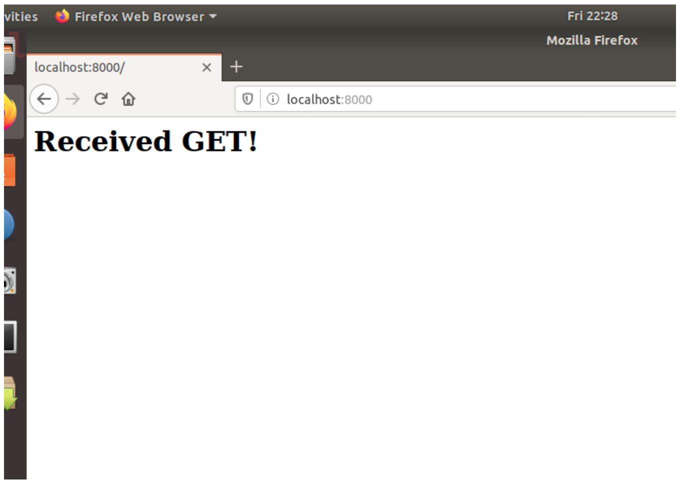
One more thing of interest - in case you've never seen this happen before - is you can see the Server code handling reqeusts on the server side. Go back to the terminal window where the webserver is running and you'll see it logging output after handling every request.

You may be frustrated because something doesn't match up with your expectations here. We see a website in a browser, but how come we don't type "www.something.com"??? What is this "localhost" nonsense? What do you mean it's running on port 8000??? I thought you said something or other about port 80 being for http??? All this understanding will come with time. For now we just need to know that websites handle GET and POST requests (and other types of requests too). And we've seen a little bit of code that features a get and post handler, and we've verified that (the GET at least) works as expected.

Now enough fooling around! Let's use cURL to test out our little website.

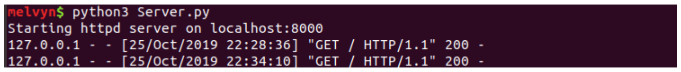
3 Two Exercises

#1 : Change port 8000 in the code to port 80 and try to rerun.



Received GET!

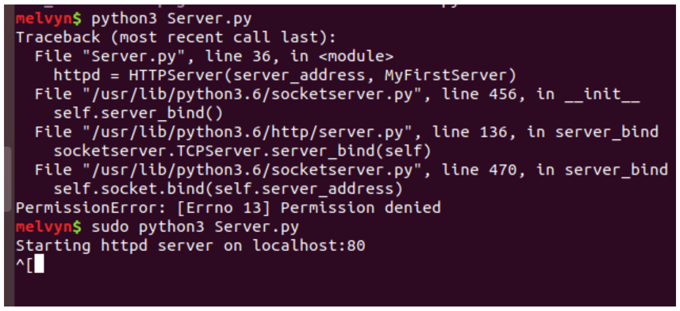
Figure 2: GET request in browser to 127.0.0.1:8000



```
melvyn$ python3 Server.py
Starting httpd server on localhost:8000
127.0.0.1 - - [25/Oct/2019 22:28:36] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [25/Oct/2019 22:34:10] "GET / HTTP/1.1" 200 -
```

Figure 3: See the webserver handling the GET request traffic.

Some ports are specified for a specific purpose by the os and may require special permissions to use. For example, port 80 requires root access to use. You can try to change the port 8000 in the sample code to port 80 and you will see a permission denied error unless you use sudo.



```
melvyn$ python3 Server.py
Traceback (most recent call last):
  File "Server.py", line 36, in <module>
    httpd = HTTPServer(server_address, MyFirstServer)
  File "/usr/lib/python3.6/socketserver.py", line 456, in __init__
    self.server_bind()
  File "/usr/lib/python3.6/http/server.py", line 136, in server_bind
    socketserver.TCPServer.server_bind(self)
  File "/usr/lib/python3.6/socketserver.py", line 470, in server_bind
    self.socket.bind(self.server_address)
PermissionError: [Errno 13] Permission denied
melvyn$ sudo python3 Server.py
Starting httpd server on localhost:80
^[
```

Figure 4: Need Root Access for Port 80

#2 : Change 'localhost' in the code to '127.0.0.1'. Note that is runs as expected - localhost and 127.0.0.1 are the same thing.

If ip addresses interest you, like, why was '127.0.0.1' selected as ''localhost'? Are there other ip addresses that were designated as local ip addresses? Then you should learn more about computer networking! I admit that I'm very interested in the subject, but haven't studied it enough due to other responsibities and ambitions.

4 10 Minute Break

We'll take a 10 minute break now in case you need a coffee or to go to the bathroom. If you don't mind continuing to work, I encourage you to change the GET request in your little website so that your browser displays the HTML from section 2.7. If you can do that, try to modify it some more with some other HTML tags.

5 cURL

cURL is a Linux tool that has been around for about 20 years. It's used for making requests. It can be thought of as a command line browser, in some sense. The things you usually do in a browser, like typing an address into a search bar, entering data into a form on a web page, clicking a button - these things can be done on the command line with cURL (mostly).

5.1 Exercise #1

Open another terminal window in addition to the one running your Python webserver. Then enter these commands:

```
user@machine$ curl localhost:8000
# returns some html from the website
user@machine$ curl -X GET localhost:8000
# returns the same thing, because the default curl action is GET
user@machine$ curl -X POST -d"myname=melvyn" localhost:8000
# returns the POST response, along with the data you passed in
user@machine$ curl -X POST --verbose -d"myname=melvyn" localhost:8000
# returns the same thing, but this time with lots of extra information!
# this time you can see what you sent over via the POST in detail
# along with a detailed response from the server
# The lines with a > are what you sent.
# The lines with a < are what came back.
user@machine$ curl -X POST --verbose -d"myname=melvyn&mylastname=drag" localhost:8000
# Try sending more and more data values by separating them with an ampersand.
user@machine$ curl -X POST -H "Content-Type: application/text" -H "Content-Length: 10"
    -d"myname=melvyn" localhost:8000
# Look what happened due to Content-Length, play around with the length to see what else
    happens
# also check and see if anything happens if you change Content-Type to application/json
    or application/x-www-form-urlencoded
user@machine$
```

The '-X' flag is used for specifying the HTTP verb associated with your request. As you've seen, the webserver will contain code for handling different types of requests.

The '-d' flag is used for passing data. Typically we associate the '-d' flag with POST requests and not GET requests.

5.2 Headers

As with most other data transmission protocols, file formats, etc. - most computer data formats - there is a header, followed by a body. Whenever you send an HTTP request, some header data goes along with it. Two of the more common ones you'll see are:

1. Content-Length - how long the data being passed is
2. Content-Type - what type of data is being sent? Json? Text? HTML? Other?

5.3 HTTP Return Codes

There are many different return codes. When you send a request to a webserver, it responds with some header data, a body, and also, a return code. Common return codes that you may have seen already are:

1. 200 (OK) - The 200 return code is when you HTTP transaction succeeds
2. 404 (Not Found) - Happens when you request something from the internet that is not there
3. 500 (Internal Server Error) - Generic message for when something has gone wrong with the server

To see a 200, type

```
curl --verbose https://www.google.com  2>&1 | grep "^< HTTP"
```

To see a 404 type

```
icurl --verbose https://www.google.com/doesntExist  2>&1 | grep "^< HTTP"
```

To see a 500 - I'm not sure how to force this error off the top of my head.

More information about return codes can be found here: `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`

I said I'm not sure how to force the 500 error because I'm not an experience web programmer - but I do know a thing or two about Linux. Did you all understnad the piped commands I used to generate the status codes?

Two interesting things:

- 2>&1 means pass stderr to stdout. Did this because curl seems to dump to stderr.
- grep "<̂HTTP" means find a line starting with "<HTTP"

5.4 Congratulations!

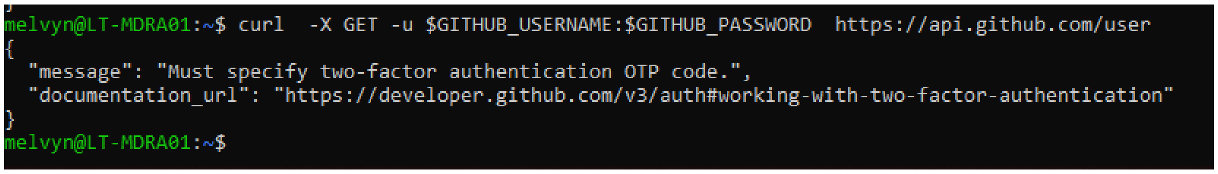
You may have just run your first webserver! If this wasn't your first webserver project, I hope it was fun anyway.

As I said, the purpose of this assignment is not for you to become a webdeveloper, but just for you to know a couple of things so that when we set up an Apache webserver, it's not totally foreign to you.

6 A Real World Curl Example

Let's do a real world curl example right now. You may want to use curl to make a web request.

To illustrate what we are going to do now, see the following two images:



```
melvyn@LT-MDRA01:~$ curl  -X GET -u $GITHUB_USERNAME:$GITHUB_PASSWORD  https://api.github.com/user
{
  "message": "Must specify two-factor authentication OTP code.",
  "documentation_url": "https://developer.github.com/v3/auth#working-with-two-factor-authentication"
}
melvyn@LT-MDRA01:~$
```

Figure 5: Couldn't access api

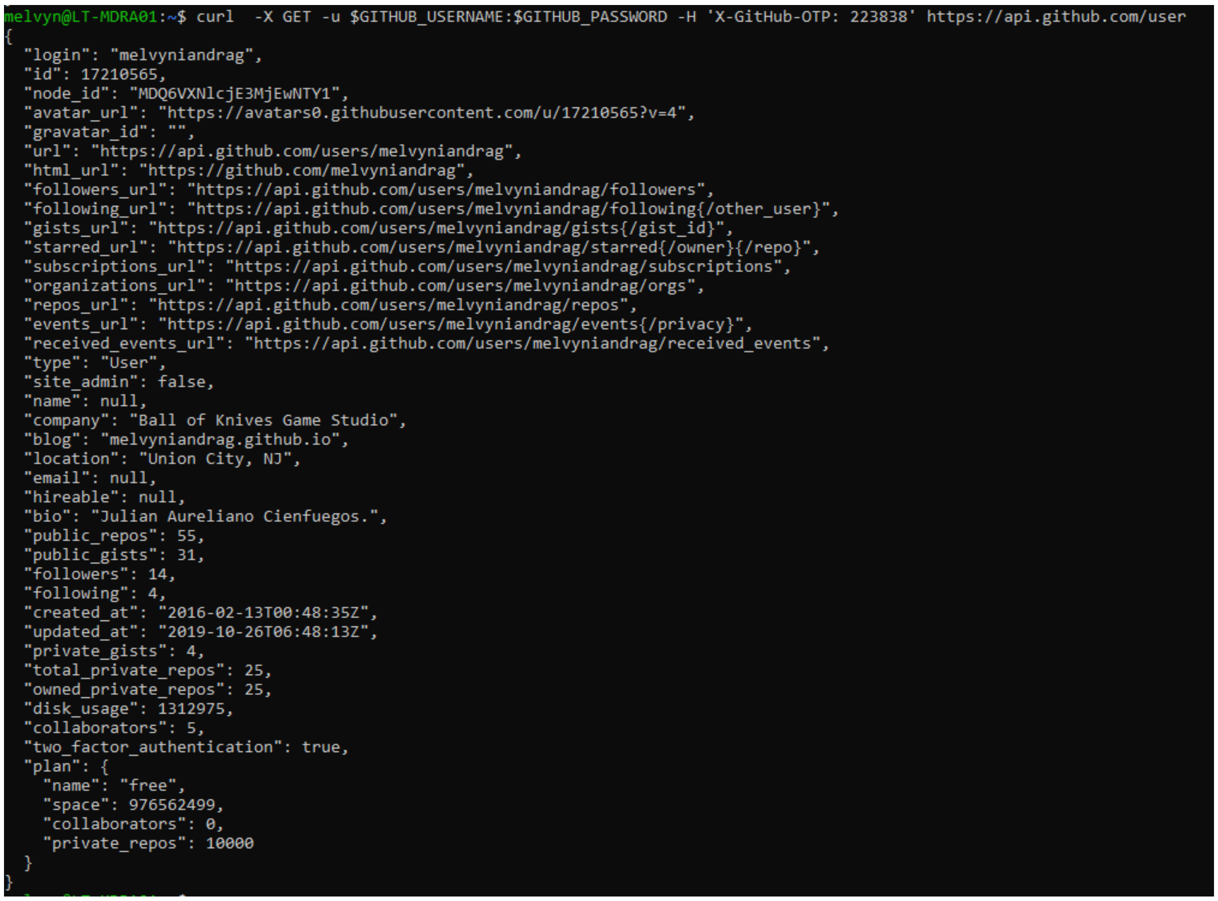
1. Have everyone turn on 2fA on their github accounts
2. In ~/.bashrc add the line 'export GITHUB_USERNAME=yourusername'
3. In ~/.bashrc add the line 'export GITHUB_PASSWORD=yourpassword'
4. Don't let anyone see this!
5. source ~/.bashrc
6. Now we are ready

How to request a 2FA code be sent to your phone

```
 curl -X POST \
  -u $GITHUB_USERNAME:$GITHUB_PASSWORD \
  -H 'Content-Type: application/json' \
  -d '{"scopes": ["user:email"],"note": "blog example"}' \
  https://api.github.com/authorizations
```

How to use it

```
curl  -X GET \
  -u $GITHUB_USERNAME:$GITHUB_PASSWORD \
  -H 'X-GitHub-OTP: 223838' \
  https://api.github.com/user
```



```
melvyn@LT-MDRA01:~$ curl  -X GET -u $GITHUB_USERNAME:$GITHUB_PASSWORD -H 'X-GitHub-OTP: 223838' https://api.github.com/user
{
  "login": "melvyniandrag",
  "id": 17210565,
  "node_id": "MDQ6VXNlcjE3MjEwNTY1",
  "avatar_url": "https://avatars0.githubusercontent.com/u/17210565?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/melvyniandrag",
  "html_url": "https://github.com/melvyniandrag",
  "followers_url": "https://api.github.com/users/melvyniandrag/followers",
  "following_url": "https://api.github.com/users/melvyniandrag/following{/other_user}",
  "gists_url": "https://api.github.com/users/melvyniandrag/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/melvyniandrag/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/melvyniandrag/subscriptions",
  "organizations_url": "https://api.github.com/users/melvyniandrag/orgs",
  "repos_url": "https://api.github.com/users/melvyniandrag/repos",
  "events_url": "https://api.github.com/users/melvyniandrag/events{/privacy}",
  "received_events_url": "https://api.github.com/users/melvyniandrag/received_events",
  "type": "User",
  "site_admin": false,
  "name": null,
  "company": "Ball of Knives Game Studio",
  "blog": "melvyniandrag.github.io",
  "location": "Union City, NJ",
  "email": null,
  "hireable": null,
  "bio": "Julian Aureliano Cienfuegos.",
  "public_repos": 55,
  "public_gists": 31,
  "followers": 14,
  "following": 4,
  "created_at": "2016-02-13T00:48:35Z",
  "updated_at": "2019-10-26T06:48:13Z",
  "private_gists": 4,
  "total_private_repos": 25,
  "owned_private_repos": 25,
  "disk_usage": 1312975,
  "collaborators": 5,
  "two_factor_authentication": true,
  "plan": {
    "name": "free",
    "space": 976562499,
    "collaborators": 0,
    "private_repos": 10000
  }
}
```

Figure 6: Could access api

Note that in this example, we could omit the -X GET, because that is the default behavior.
Note that there are different ways to supply parameters to cURL - I can put –header instead of -H

```
curl  -X GET \
  -u $GITHUB_USERNAME:$GITHUB_PASSWORD \
  --header 'X-GitHub-OTP: 884813'
   https://api.github.com/user
```

6.1 Exercise

Have students create a private gist on github and query github with curl to see the number of their private gists go up.

6.2 Another exercise

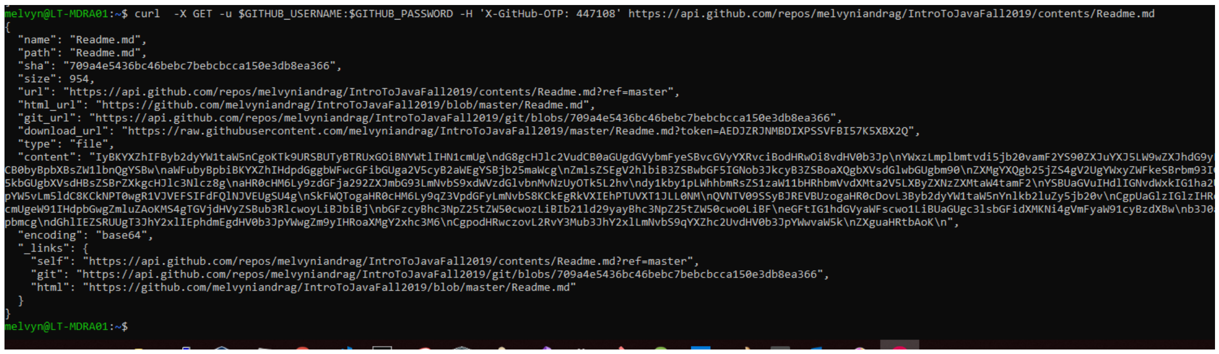
Create a private repo on github. See that you can download a file from it using the REST API. Add just a readme to the private repo.

Prompt github to send you a 2fa key.Then get the readme like this:

```
curl  -X GET \
  -u $GITHUB_USERNAME:$GITHUB_PASSWORD \
  -H 'X-GitHub-OTP: 413096' \
  https://api.github.com/repos/melvyniandrag/IntroToJavaFall2019/contents/Readme.md
```

(but make sure to set the code to the one given to you by github via the phone or email, however they send you the 2FA code.)

Note that the encoding is something calle ”base64”. There are many encodings out there. In my JAva class we talked quite a bit about Hex, binary, and decimal encodings for bytes. Then for text we talked about encodings like ASCII, UTF8 and UTF16. We’ll visit those topics in this class eventually, but for now it’s just something interesting to look at.You’ll see that the content of your Readme.md file is NOT what you put on github - its a bunch of random alphanumeric cahracters in something called ”base64”. Interesting! But what does it mean?!¿! You’ll have to wait and see! Unless your curiosity gets the better of you, in which case you can just go online and loo kup what base64 encoding is.



```
melvyn@LT-MDRA01:~$ curl  -X GET -u $GITHUB_USERNAME:$GITHUB_PASSWORD -H 'X-GitHub-OTP: 447108' https://api.github.com/repos/melvyniandrag/IntroToJavaFall2019/contents/Readme.md
{
  "name": "Readme.md",
  "path": "Readme.md",
  "sha": "709a4e5436bc46bebc7bebcbcca150e3db8ea366",
  "size": 954,
  "url": "https://api.github.com/repos/melvyniandrag/IntroToJavaFall2019/contents/Readme.md?ref=master",
  "html_url": "https://github.com/melvyniandrag/IntroToJavaFall2019/blob/master/Readme.md",
  "git_url": "https://api.github.com/repos/melvyniandrag/IntroToJavaFall2019/git/blobs/709a4e5436bc46bebc7bebcbcca150e3db8ea366",
  "download_url": "https://raw.githubusercontent.com/melvyniandrag/IntroToJavaFall2019/master/Readme.md?token=AEDJZRJNMBDIXPSSVFBI57K5XBX2Q",
  "type": "file",
  "content": "IyBKYXZhIFByb2dyYW1taW5nCgoKTk9URSBUTyBTRUxGOiBNYWtlIHN1cmUg\ndG8gcHJlc2VudCB0aGUgdGVybmFyeSBvcGVyYXRvciBodHRwOi8vdHV0b3Jp\nYWxzLmplbmtvdi5jb20vamF2YS90ZXJuYXJ5LW9wZXJhdG9y
CB0byBpbXBsZW1lbnQgYSBw\naWFubyBpbiBKYXZhIHdpdGggbWFwcGFibGUga2V5cyB2aWEgYSBjb25maWcg\nZmlsZSEgV2hlbiB3ZSBwbGF5IGNob3JkcyB3ZSBoaXQgbXVsdGlwbGUgbm90\nZXMgYXQgb25jZS4gV2UgYWxyZWFkeSBrbm93IG
5kbGUgbXVsdHBsZSBrZXkgcHJlc3Nlcz8g\naHR0cHM6Ly9zdGFja292ZXJmbG93LmNvbS9xdWVzdGlvbnMvNzUyOTk5L2hv\ndy1kby1pLWhhbmRsZS1zaW11bHRhbmVvdXMta2V5LXByZXNzZXMtaW4tamF2\nYSBUaGVuIHdlIGNvdWxkIG1ha2
pYW5vLm5ldC8KCkNPT0wgR1VJVEFSIFdFQlNJVEUgSU4g\nSkFWQTogaHR0cHM6Ly9qZ3VpdGFyLmNvbS8KCkEgRkVXIEhPTUVXT1JLL0NM\nQVNTV09SSyBJREVBUzogaHR0cDovL3Byb2dyYW1taW5nYnlkb2luZy5jb20v\nCgpUaGlzIGlzIHR
cmUgeW91IHdpbGwgZmluZAoKMS4gTGVjdHVyZSBub3RlcwoyLiBJbiBj\nbGFzcyBhc3NpZ25tZW50cwozLiBIb21ld29yayBhc3NpZ25tZW50cwo0LiBF\neGFtIG1hdGVyaWFscwo1LiBUaGUgc3lsbGFidXMKNi4gVmFyaW91cyBzdXBw\nb3J0
pbmcg\ndGhlIEZSRUUgT3JhY2xlIEphdmEgdHV0b3JpYWwgZm9yIHRoaXMgY2xhc3M6\nCgpodHRwczovL2RvY3Mub3JhY2xlLmNvbS9qYXZhc2UvdHV0b3JpYWwvaW5k\nZXguaHRtbAoK\n",
  "encoding": "base64",
  "_links": {
    "self": "https://api.github.com/repos/melvyniandrag/IntroToJavaFall2019/contents/Readme.md?ref=master",
    "git": "https://api.github.com/repos/melvyniandrag/IntroToJavaFall2019/git/blobs/709a4e5436bc46bebc7bebcbcca150e3db8ea366",
    "html": "https://github.com/melvyniandrag/IntroToJavaFall2019/blob/master/Readme.md"
  }
}
melvyn@LT-MDRA01:~$
```

Figure 7: See how I got the readme data

6.3 Danger of curl in place of 2FA

There was a security leak a little while back at paypal whereby you could bypass 2FA using curl. `https://duo.com/blog/duo-security-researchers-uncover-bypass-of-paypal-s-two-factor-authentication`

6.4 See these references

`https://developer.github.com/v3/auth/#working-with-two-factor-authentication` `https://blogs.infosupport.com/accessing-githubs-rest-api-with-curl/`

7 What’s a REST API

There are just too many buzzwords out there. I’ve showed you that we could get html back with curl when we curled out own little python server. Then I showed you that when we curled api.github.com, we got back JSON data. There is more to it, but in general, when we send and receive JSON, we call this a REST API. If you look into web development more, you’ll learn what the REST stand for, and then you’ll learn why RESTful ness is important. For us it doesn’t matter at all. All that matters is that you can see that REST is for exchanging JSON data.

7.1 Job interview prep

Interviewer: *I see you know a bit about Linux webserver maintenance. Do you know what a REST API is?*
You: Honestly, not too much because I’m not a webdeveloper. But I do know that REST APIs are very popular now, and they typically involve the exchange of JSON data. JSON data is when the data is structured like a Python dicitonary, or a Java Hashmap - its a bunch of key-value pairs in curly braces.

If you can give the above answer, you’ll know enough for now.

8 Back to Tabs and Spaces

We briefly mentioned the difference between tabs and spaces. You can set the tab width in Vim. Maybe mention the ASCII codes for tab vs space so that the folks in the Java class can make the connection.

9 Recap of what you know

1. A bit about the Bash programming language
2. grep & regular expressions
3. What a user is on Linux
4. What a group is on Linux
5. What is git?
6. You're comfortable using a command line interface now
7. There are a few different languages on the command line - we've used bash and dash
8. What permissions are and how to modify them
9. What is a root user
10. What is a process
11. What is a job
12. How to send signals to processes (kill, CTRL+C, CTRL+Z)
13. How to make code ignore or block signals
14. How to install software on Linux
15. A cool trick for using setuid / setgid to make a non-root user do some root stuff
16. basics of relational dbs and how to configure one on Linux
17. What is cron
18. curl
19. some vocabulary like API, REST, regex, SQL, DB
20. A bit about python programming

10 Coming Up In This Class

We have a few more important things to cover

1. set up apache webserver
2. set up a git server
3. add a gitlab front end to the git server
4. Linux and Text encodings
5. xxd, everyone's favorite binary packet analyzer
6. The AWK programming language
7. The hows and whys of formatting harddrives, usb sticks, solid state drives, etc.
8. Encryption with gpg + pgp. How it relates to ssh and other pub/priv key schemes.
9. sed