Week 09

Apache2 and Flask

Making a Professional Webserver

Melvyn Ian Drag

November 7, 2019

Abstract

Last class we took a look at the toy 'SimpleHTTPServer' in Python3. Tonight we'll use a real website backend - Flask - and we'll server content on a real server - Apache2. As before, we'll make **GET** and **POST** requests using the command line tool cURL.

1 Meta Data

I want to show you how to set up a webserver on Linux - last week's lecture was supposed to ease you into the mindset, but a few people missed lecture or left early. There isn't time to revist last week's lecture, there was too much information. Hopefully today will stand alone. If it doesn't, you'll need to work through last week's notes. There was a 3 hour lecture, so you will probably need at least 3 hours to read through the notes.

Still, we'll try and make tonight as 'stand-alone' as possible.

2 What's a Webserver?

The term is overloaded. A server is a computer, connected to a network, that other computers can access. There may be more ways to define it, but that's what I think of. But for your midterm I had you install some postgresql server stuff. In that case I was referring to the database server software. And tonight I'm going to refer to Apache2 as a 'webserver' - it's actually software that runs *on* a webserver, which is the machine. It's just that the terms are overloaded.

3 What's Flask?

Flask is what is called a **Library**. It's some code that you can use in a language to solve a particular problem. For example, if you know Java you've probably used the *java.util* library for things like *ArrayList*s. In Python last week we used a library calle *SimpleHTTPServer* to serve up some webpages for us on the address *localhost:8000*.

4 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.

4.1 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.

5 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.

6 What we'll do today

Today we'll bring a real website online. Last week we made a trivial website with Python's SimpleHTTPServer, but today we're going to use a real, professional-grade server (Apache2) and a real backend web framework (Flask). There are other servers like NGINX and there are other backend webframeworks like Django, Play, Spring, Ruby on Rails, etc., I've just chosen this pair because it's the easiest to code of the few things I know. Spring on NGINX would be considerably harder to configure.

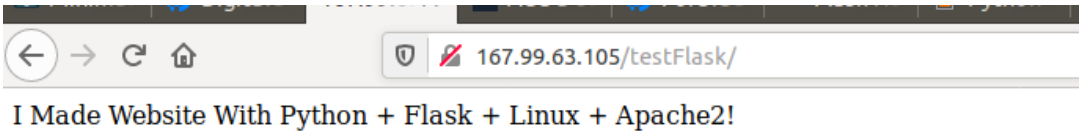

Figure 1: A website running on my server at IP address http://167.99.63.105/testFlask/

7 Setting up server

Get a debian 10 server

```
root@machine$ apt update
root@machine$ apt install apache2
root@machine$ apt-get install libapache2-mod-wsgi-py3 python3-dev python3-pip
root@machine$ pip3 install flask flask-restful
root@machine$ apt install curl
root@machine$ service apache2 status
# should report that apache2 is running
root@machine$ curl localhost
# lots of data
root@machine$ curl localhost > curlResponse.html
# dump the data to a file so it's easier to look through
root@machine$ less curlResponse.html # look through, verify you have the default apache
    page. That means the server is running.
root@machine$ curl ipinfo.io/io # get your server ip address
```

Open a browser on a laptop or computer. Go to the server ip address, make sure the server is running. You should see the same default apache page you saw with curl localhost. But now you are looking at it through a browser on a different machine (you ran curl on the webserver itself).

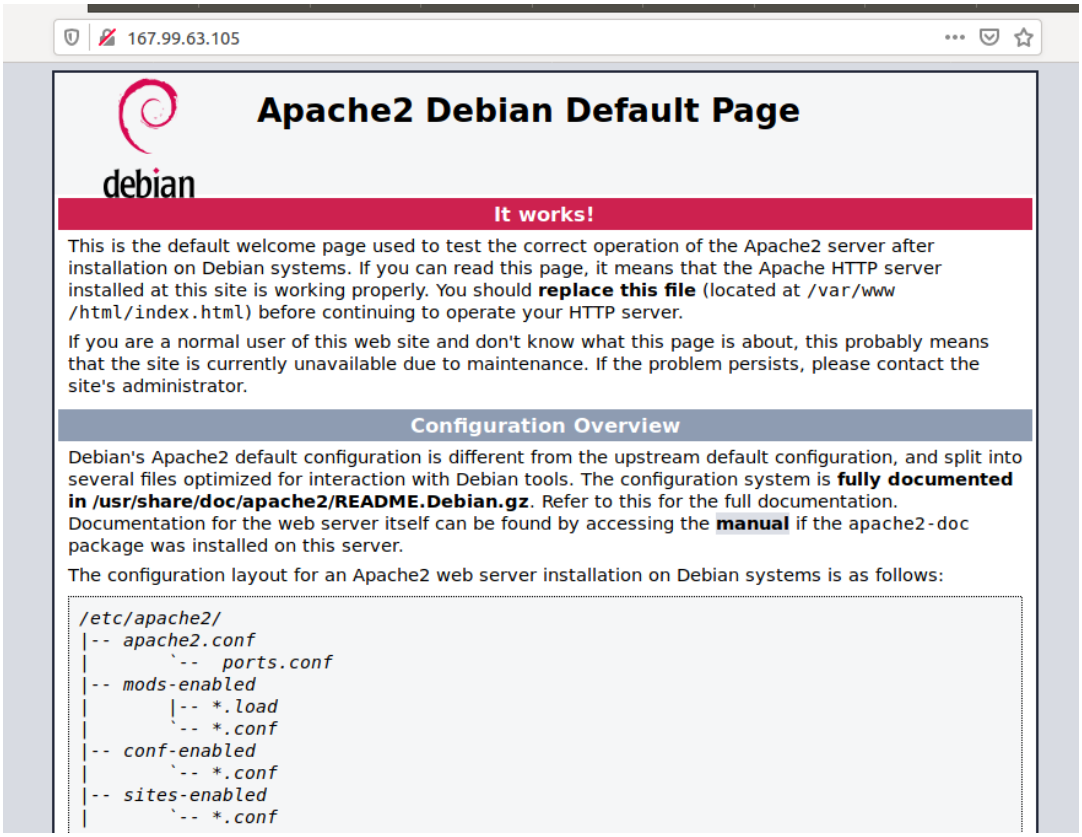

Figure 2: default webpage provided by a fresh Apache2 install

So we've installed all the software we need! That was easy. Now we can build a little website and then connect the website code to the apache server we just installed and activated.

8 Creating a Flask application

The first thing we need to do is create a non root user for managing this stuff. We'll give the user sudo permission for the few root things we need to do to bring the website online. I've created user 'webdeveloper'. You should do the same so you can copy and paste the code I've provided. If you create a different username, you'll need to make the minor modifications to make everyting match up.

```
1 root@machine$ adduser webdeveloper
2 root@machine$ usermod -a -G sudo webdeveloper
3 root@machine$ sudo su - webdeveloper
4 root@machine$ mkdir /home/webdeveloper/ExampleFlask
```

Inside the ExampleFlask directory, put the files described below. Note that these files are available in the Example1/ExampleFlask directory in the class Git repo.

8.1 __init__.py

This is an empty file

8.2 my_flask_app.py

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello():
    return "I Made Website With  Python + Flask + Linux + Apache2!"
if __name__ == "__main__":
    app.run()
```

8.3 my_flask_app.wsgi

```
#!/usr/bin/python3

import logging
import sys
logging.basicConfig(stream=sys.stderr)
sys.path.insert(0, '/home/webdeveloper/ExampleFlask')

from my_flask_app import app as application
application.secret_key = 'anything you wish'
```

9 Connect Flask to Apache2

Now we need to wire up our Flask application to the Apache webserver software. You will need to know your machine's ip address. I showed you a website you can curl that will tell you your ipaddress

```
user@machine$ curl ipinfo.io/ip
# returns your external ip address
```

Then you need to create this file, and put the proper ip address. Note that you'll need to run vim with sudo as this is a privileged file.

9.1 /etc/apache2/sites-available/ExampleFlask.conf

```
<VirtualHost *:80>
     # Add machine's IP address (use curl ipinfo.io/ip)
     ServerName 167.99.63.105
     # Give an alias to to start your website url with
     WSGIScriptAlias /testFlask /home/webdeveloper/ExampleFlask/my_flask_app.wsgi
     <Directory /home/webdeveloper/ExampleFlask>
            Options FollowSymLinks
            AllowOverride None
            Require all granted
     </Directory>
     ErrorLog ${APACHE_LOG_DIR}/error.log
     LogLevel warn
     CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

9.2 Turn on and Test the Website

Run these commands as the user 'webdeveloper'

```
webdeveloper@machine$ sudo a2ensite ExampleFlask
webdeveloper@machine$ sudo a2enmod wsgi
webdeveloper@machine$ sudo service apache2 restart
```

Then, open a browser on your laptop or the NJCU PC in front of you and go to

my.ip.addr.ess/testFlask

You should see a message saying you've made your first website with Linux, flask, apache and python. You can share the link to show off your website to your friends and family! You could now also go to godaddy or namecheap, buy a domain name, and wire that up to your server so people can go to website.com instead of a scary looking ip address.

10 Additional Considerations for Example 1

So we have a simple flask app running on apache now. As you've probably seen by now, Linux servers are very particular about groups and permissions. I'm not sure about all of the ins and outs of the permissions for Flask and Apache. I'm showing the permisisons and groups I've set up on my machine. If you do something else I'm not sure it will work, and I don't know why it owuld be broken. We could experiment with permissions if you have something else.

I'm also not sure about all the details of the ExampleFlask.conf file. There are many options to it. I've always set mine up by scouring the internet for information and banging in what ever details I found. I hope to one day understand this file in depth, but for now all I know is how to make it work.

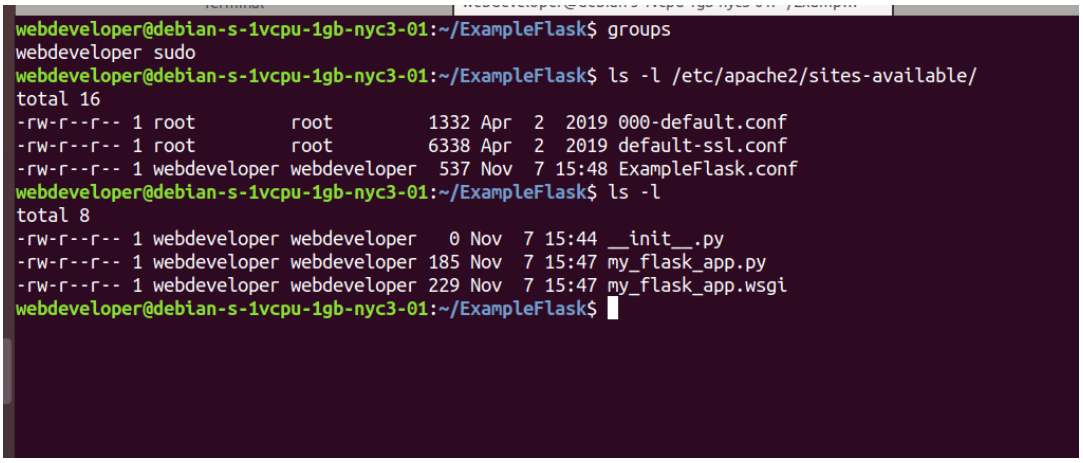

Figure 3: My permissions and groups on a successful install

Wait for class to all be caught up and have their servers running

11 About the code for Example 1

This isn't a Python class as we said, but I'll just mention a few things about the Python code so you get some sense of how it's working. You will see the line

```
1 @app.route("/")
2 def hello():
3   return "some string here"
```

This says that when we GET the location "/" in our domain name, the webserver will respond with the return value.

We have to go to "ipaddress/testFlask" to see this data because in our config file we have the line

```
1 ...
2 WSGIScriptAlias /testFlask /home/webdeveloper/ExampleFlask/my_flask_app.wsgi
3 ...
```

and this registers the base end point for our website as /testFlask. If there's time we can change this. Right now apache is serving the default apache website on /, so we had to put our website at a different location on our server. Well need to delete some config files and change this config file so that we can see our website when we go to "/" instead of having to go to "/testFlask".

The contents of the .wsgi file is boilerplate stuff that won't matter to you unless you want to become a serious python developer. Also, the purpose of the .wsgi file is interesting, but probably won't matter to you. Python applications can't run natively on webservers. So, about 20 years ago, some developers go together and wrote some middleware that allowds Python applications to communicate with webserver software like Apache2. The details are complicated - you can read about it if you're curious.

`https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface`

I think php can talk right to the webserver, but I'm not sure. I've never written a line of php in my life. All I know is that PYhton needs a thing called wsgi (which we installed at the beginning of this lecture with libapache-mod-wsgi-py3). You'll note that there are different version for python3 and python2.

12 Example2: Adding Endpoints and Serving HTML

To do this

1. Change the ExampleFlask.conf file to the one below.
2. Create the /home/webdeveloper/Example2 directory add the files from the class repo
3. Restart apache2

12.1 New /etc/apache2/sites-available/ExampleFlask.conf

Note that this file now serves two webpages! You can host multiple websites on one webserver.

```
<VirtualHost *:80>
     # Add machine's IP address (use curl ipinfo.io/ip)
     ServerName 167.99.63.105

     # Some error logging stuff
     ErrorLog ${APACHE_LOG_DIR}/error.log
     LogLevel warn
     CustomLog ${APACHE_LOG_DIR}/access.log combined

     # Give an alias to to start your website url with
     WSGIDaemonProcess site1
     WSGIScriptAlias /testFlask /home/webdeveloper/ExampleFlask/my_flask_app.wsgi
     <Directory /home/webdeveloper/ExampleFlask>
            WSGIApplicationGroup site1
            WSGIProcessGroup site1
            Options FollowSymLinks
            AllowOverride None
            Require all granted
     </Directory>

     # Set up the other website
     WSGIDaemonProcess site2
     WSGIScriptAlias /somewhereElse /home/webdeveloper/ExampleFlask2/my_flask_app.wsgi
     <Directory /home/webdeveloper/ExampleFlask2>
            WSGIApplicationGroup site2
            WSGIProcessGroup site2
            Options FollowSymLinks
            AllowOverride None
            Require all granted
     </Directory>
</VirtualHost>
```

12.2 my_flask_app.py

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "I Made Website With  Python + Flask + Linux + Apache2!"

@app.route("/returnsHTML")
def secondEndPoint():
    return "<html><body><h1>A Header!</h1><p>Here is some data in a
        paragraph!</p></body></html>"


if __name__ == "__main__":
    app.run()
```

12.3 my_flask_app.wsgi

```
#!/usr/bin/python3

import logging
import sys
logging.basicConfig(stream=sys.stderr)
sys.path.insert(0, '/home/webdeveloper/ExampleFlask2')

from my_flask_app import app as application
application.secret_key = 'anything you wish'
```

12.4 __init__.py

Blank as before. To be honest I haven't given it thought if this is required or not. Just put the file. If you know python you can remove it and then meditate on why it works / doesn't work if you remove this file.

12.5 Results

Here are some images showing my new site in action. The /testFlask endpoint still works, but now so do /somewhereElse and /somewhereElse/returnsHTML.

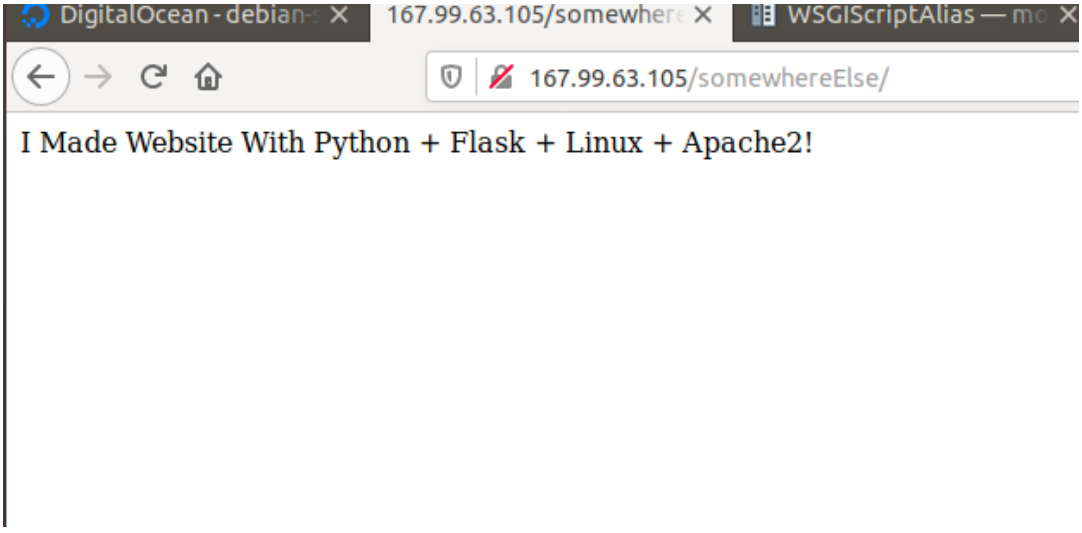

Figure 4: somewhereElse now works

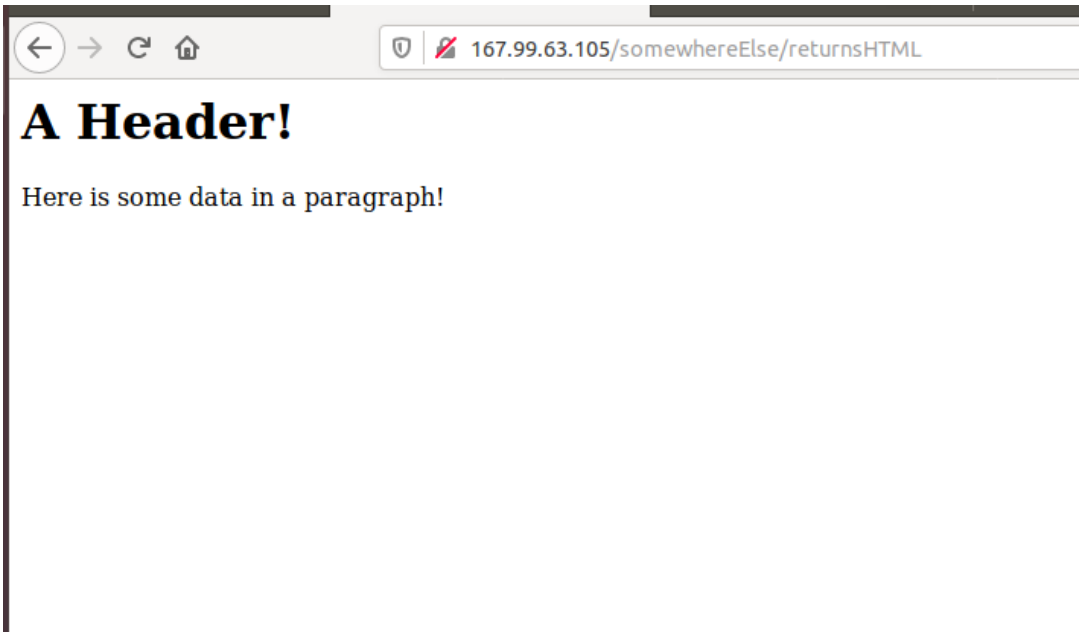

Figure 5: We can now return HTML from our Flask website too!

Wait for class to all be caught up and have their servers running again

13 Example3: A Rest API

Only do this section if the class is caught up and it's before 9 PM

So far we've returned plain text and text formatted as HTML from our website. A popular thing to do now is to pass around JSON data. JSON is used all over. I showed it to you last week just to wet your feet, but you'll see it everywhere. You'll use it for web programming, internally in Python code, passing data between applications running on a laptop, for exchanging data files with scientists - everywhere. I avoided JSON for a while because I thought it was just a stupid buzz word that the nerds used to show off, but it truly is an ubiquitous data format. Now you've seen it. You'll see it again, mark my words.

So we'll just see how to return json from a Flask application.

Just as we did in example 2, set up your server with Example3 now. Get the Example3 code from github, copy the new ExampleFlask.conf file to the proper location in /etc, create the /home/webdeveloper/Example3 directory, and put the proper code in it.

13.1 The Flask Application: my_flask_app.py

You can skim the code to see what it does. It has some code for handling the GET and POST requests. A GET request will echo back to you the entire contents of the TODOS data it's maintaining. A POST request will add to the TODOS data set. You can verify after POSTing that your data was added by sending a GET request.

```
1  """
2  Stolen from the flask-restful documentation here:
3  https://flask-restful.readthedocs.io/en/latest/quickstart.html#a-minimal-api
4
5  Super simple RESTful API that handles GET and POST requests.
6  """
7
8  from flask import Flask
9  from flask_restful import reqparse, abort, Api, Resource
10
11 app = Flask(__name__)
12 api = Api(app)
```

```
13
14 TODOS = {
15     'todo1': {'task': 'build an API'},
16     'todo2': {'task': '?????'},
17     'todo3': {'task': 'profit!'},
18 }
19
20 parser = reqparse.RequestParser()
21 parser.add_argument('task')
22
23 # TodoList
24 # shows a list of all todos, and lets you POST to add new tasks
25 class TodoList(Resource):
26     def get(self):
27         return TODOS
28
29     def post(self):
30         args = parser.parse_args()
31         todo_id = int(max(TODOS.keys()).lstrip('todo')) + 1
32         todo_id = 'todo%i' % todo_id
33         TODOS[todo_id] = {'task': args['task']}
34         return TODOS[todo_id], 201
35
36 ##
37 ## Actually setup the Api resource routing here
38 ##
39 api.add_resource(TodoList, '/todos')
40
41 if __name__ == '__main__':
42     app.run()
```

13.2 Accessing Our REST API

Then you can get JSON data in the browser as shown below:

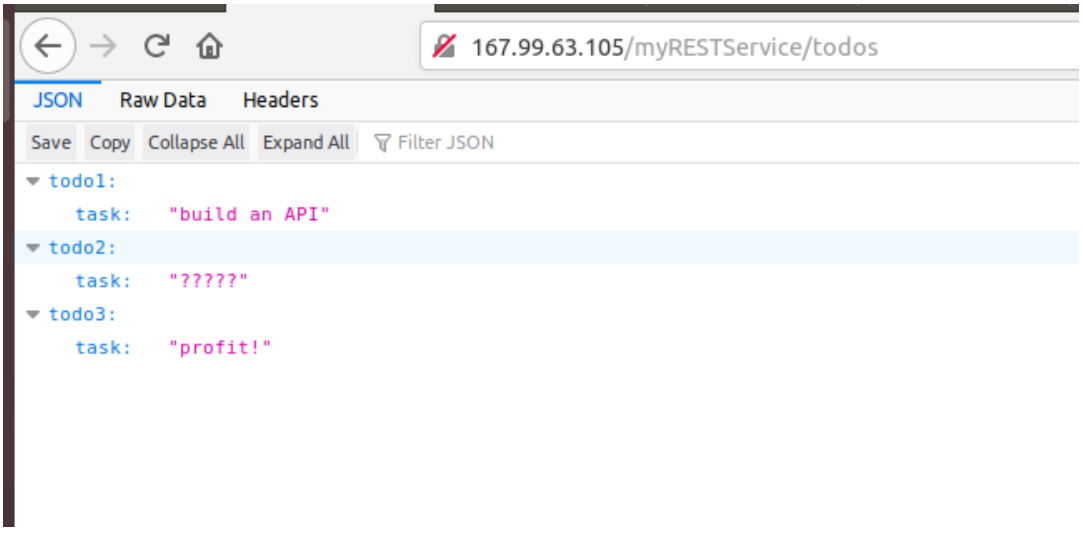

Figure 6: See some json in the browser!

Then, on your laptop, make a GET request with cURL
And, you should also be able to make the following POST request:

```
1 curl -X POST \
2   -H "Content-Type: application/json" \
```

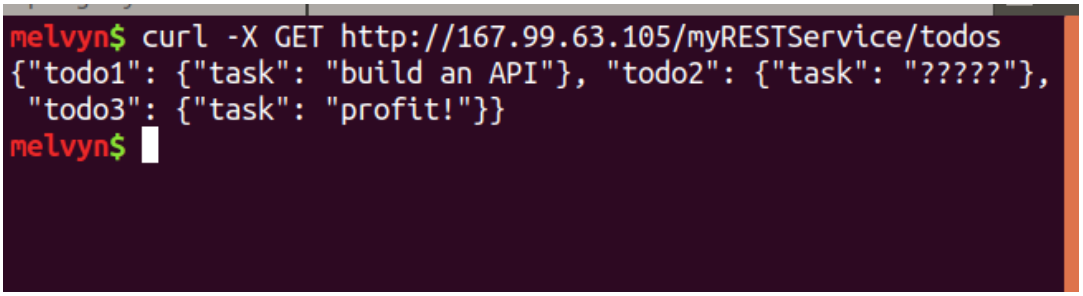

Figure 7: Make a GET request with cURL!

```
3   --data '{"task":"Learn Linux in CS407"}' \
4   http://167.99.63.105/myRESTService/todos
```

And then look, after POSTing data, when you GET data you will see the data that you POSTed!

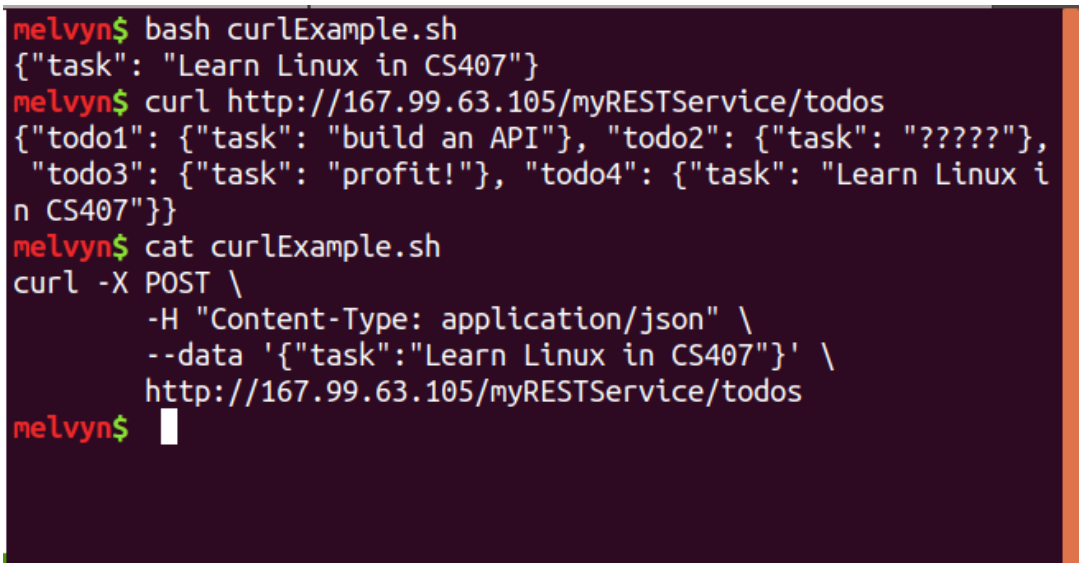

Figure 8: POST data to your website with cURL!

14 sed 9:00 - 9:30

15 Recap of what you know 9:30 - 9:45

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
21. What is a website backend? Set up a website backend with Apache2 + Flask
22. What is sed?

16 Coming Up In This Class

We have a few more important things to cover

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

17 References

This lecture was based on what I read on these websites:

`https://www.codementor.io/abhishake/minimal-apache-configuration-for-deploying-a-flask-app-ubuntu-18-04`

There were some bugs in the tutorials above that I sorted out to make sure our lecture had a good flow.