Week 09

Apache2 and Flask

Making a Professional Webserver

November 1, 2021

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 What's a Webserver?

The term is overloaded. A server is a computer, connected to a network, that other computers can access. It also refers to software than responds to client requests. 2 Weeks ago we saw a Postgres server. This week we'll make an Apache2 webserver. So The word can refer to hardware (the computer itself) or software (the software that responds to client requests).

2 What's Flask?

Flask is what is called a **Back End Web Framework**. Its a bunch of code that some one smarter than us already wrote to handle GET and POST requests in a really pretty way. It's amazing that this stuff is free. You're about to get a professional website online in a few minutes thanks to a bunch of generous smart people giving us this Flask code.

There are other backend frameworks. In Python lots of people use Django. In Java lots of folks use Play, Spring, and a bunch of others.

3 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. Deploying a Spring app on an NGINX server is probably as simple as what I've prepared for you today, and if you want to be a web developer then you should totally go research how to do it!

Plan for today:

1. Example 1: Boring website that just returns text. See figure 1
2. Example 2: Little more interesting website that returns HTML
3. Example 3: Even more interesting website that returns JSON.

4 Setting up server

So let's begin building our Flask + Apache2 website. Get a debian 10 server on digital ocean. Then run these commands:

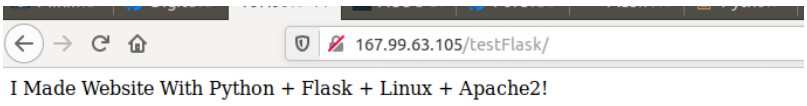

Figure 1: A website running on my server at IP address http://167.99.63.105/testFlask/ . This is the first website we will build together today. It will run on Linux using Python, Flask and Apache2

Listing 1: configure your server

```
root@machine$ apt update
root@machine$ apt install git apache2 libapache2-mod-wsgi-py3 python3-dev
python3-pip curl
# so now we have git for the class code
# apache2 for the webserver
# python3 for the backend framework
# curl for making HTTP requests
root@machine$ pip3 install flask flask-restful
# now that we have python3 we can install flask.
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/ip # 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).

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.

Warning! Make sure you have succeeded in this first task. If you haven't, you won't be able to do the rest of the exercises. If you can see the default Apache webpage, then continue on.

Congratulations!! You've set up your first Apache webserver! Wasn't that easy??? Linux is good.

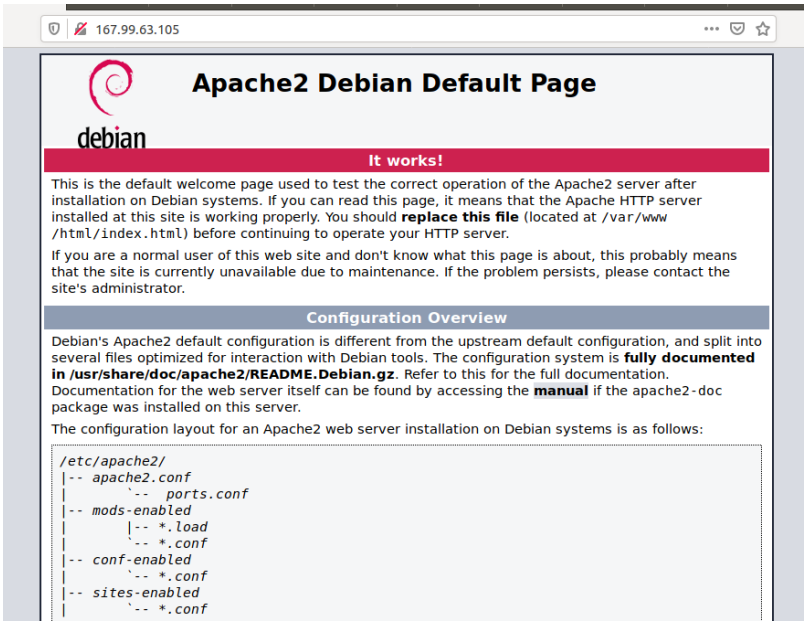

Figure 2: default webpage provided by a fresh Apache2 install

5 Example 1: Deploying Your First 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.

```
root@machine$ adduser webdeveloper
root@machine$ usermod -a -G sudo webdeveloper
root@machine$ su - webdeveloper
webdeveloper@machine$ cd ~
webdeveloper@machine$ git clone
    https://github.com/melvyniandrag/LinuxClassRepo.git
webdeveloper@machine$ cp -r \
LinuxClassRepo/Lectures/Week10_Apache2_And_Flask/WebsiteThatReturnsText/CodeForWebsiteThatReturnsText
    \
CodeForWebsiteThatReturnsText
webdeveloper@machine$ ls
LinuxClassRepo CodeForWebsiteThatReturnsText
webdeveloper@machine$ ls CodeForWebsiteThatReturnsText
__init__.py my_flask_app.py my_flask_app.wsgi
```

Let's take a quick minute to inspect these files. These files will be confusing to you, but they're important. I'm just going to show them to you on the off chance that you're curiosity gets piqued so you can go out and learn more.

5.1 __init__.py

What is this empty file doing here? You'll need to go find a good Python book or online tutorial if you want to know. There's no time to discuss what this is now.

Listing 2: Notice that __init__.py is empty

```
webdeveloper@machine$ pwd
/home/webdeveloper/WebsiteThatReturnsText
webdeveloper@machine$ cat __init__.py
```

```
webdeveloper@machine$
```

5.2 my_flask_app.py

This is the Python code for our website. It contains a single function *hello()* that is available at the route "/". So, assume we have a domain name *www.mydomain.com* for this little website. To access the *hello()* function, we have to go to *www.mydomain.com/* (note the slash at the end). We could put other routes like */otherroute*, and indeed we will do this in the next exercise. Be patient!

Listing 3: Flask is such a cool framework. Look how short the code is!

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route("/")
6  def hello():
7      return "I Made Website With Python + Flask + Linux + Apache2!"
8
9  if __name__ == "__main__":
10     app.run()
```

5.3 my_flask_app.wsgi

I believe we already said that python apps dont run on the internet by default. You need a "little something extra" to make it work. That something extra is WSGI and you'll see that we already installed it at the beginning of the lecture when you installed *libapache2-mod-wsgi-py3*. This library lets python3 apps run on your webserver. Anyway, this .wsgi file contains some configuration stuff to get the website working. You'll need to study Flask in depth to understand what this file is and how I created it.

Listing 4: Deploying Flask apps on Linux servers is easy! This configuration file is so short and sweet!

```
1  #!/usr/bin/python3
2
3  import logging
4  import sys
5
6  logging.basicConfig(stream=sys.stderr)
7  sys.path.insert(0, '/home/webdeveloper/CodeForWebsiteThatReturnsText')
8
9  from my_flask_app import app as application
10
11 application.secret_key = 'anything you wish'
```

6 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.

Listing 5: Run this command to get your server's ip address.

```
webdeveloper@machine$ curl ipinfo.io/ip
192.34.61.215
```

Now you need to get a configuration file for the website into the */etc* directory. The configuratin file is provided in the class repo in Lectures/Week10*/WebsiteThatReturnsText. Just copy it over.

Listing 6: Get the configuration file

```
webdeveloper@machine$ sudo su -
root@machine$ cp \
/home/webdeveloper/LinuxClassRepo/Lectures/Week10/WebsiteThatReturnsText/WebsiteThatReturnsText.conf
    \
/etc/apache2/sites-available/WebsiteThatReturnsText.conf
```

Now you must change the ip address in this file to the ip address of your server. Remember when we got the ip address above in listing 5? Here's what the file looks like. Don't forget to change the ip address! This is the configuration file used by Apache2 to run your website.

Listing 7: Apache2 configuration file

```
1  # port 80 for http. port 443 for https
2  <VirtualHost *:80>
3      # Add machine's IP address (run curl ipinfo.io/ip if you dont know it.)
4      ServerName 167.99.63.105
5
6      # Give an alias to to start your website url with
7      WSGIScriptAlias /Example1
           /home/webdeveloper/WebsiteThatReturnsText/my_flask_app.wsgi
8
9      # Code for the website and some configuration for the site.
10     <Directory /home/webdeveloper/WebsiteThatReturnsText>
11           Options FollowSymLinks
12           AllowOverride None
13           Require all granted
14     </Directory>
15
16     #stuff about logging
17     ErrorLog ${APACHE_LOG_DIR}/error.log
18     LogLevel warn
19     CustomLog ${APACHE_LOG_DIR}/access.log combined
20 </VirtualHost>
```

6.1 Turn on and Test the Website

Alright then, let's do this! Run these commands as the user 'webdeveloper'

```
webdeveloper@machine$ sudo a2ensite WebsiteThatReturnsText
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, just like I showed you at the beginning of lecture in figure 1 . 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.

6.2 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 WebsiteThatReturnsText.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.

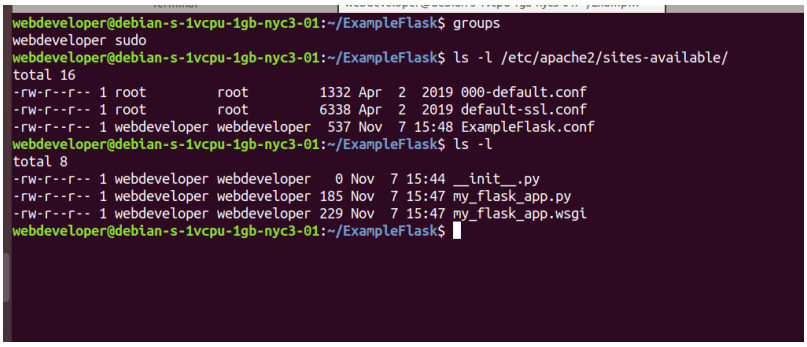

Figure 3: My permissions and groups on a successful install

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

6.3 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.

But then why do we have to go to *192.34.61.215/testFlask* and not just *192.34.61.215/* (as we said in section 5.2 ? We have to go to *192.34.61.215/testFlask/* to see this data because in our config file we have the line

```
1 ...
2 WSGIScriptAlias /Example1
      /home/webdeveloper/CodeForWebsiteThatReturnsText/my_flask_app.wsgi
3 ...
```

and this registers the base end point for our website as /testFlask.

The contents of the .wsgi file shown in section 5.3 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`

7 Example2: Adding Endpoints and Serving HTML

In the last exercise we just had our website return text - but websites typically return either HTML (if you're using a browser) or JSON (if you're talking to the website

via a REST API with a tool like curl. Let's start by making a website that returns HTML, since it's not much harder than the last one we set up.

Luckily for you I've prepared a simple website that returns HTML. Let's set it up:

Listing 8: Deploying our second website

```
webdeveloper@machine$ pwd
/home/webdeveloper
webdeveloper@machine$ ls
LinuxClassRepo ExampleFlask
webdeveloper@machine$ cp -r
LinuxClassRepo/Lectures/Week10_Apache2_And_Flask/WebsiteThatReturnsHTML/CodeForWebsiteThatReturnsHTML
CodeForWebsiteThatReturnsHTML
webdeveloper@machine$ ls
LinuxClassRepo ExampleFlask CodeForWebsiteThatReturnsHTML
webdeveloper@machine$ ls CodeForWebsiteThatReturnsHTML
__init__.py my_flask_app.py my_flask_app.wsgi
```

Good, we have the new code in place. Now let's copy over the new configuration file too.

LISTEN!!!!

- Don't forget top change the ip address in ExampleFlask.conf to the ip address of your server.
- Don't forget to change the ip address.
- Change the ip address.
- Change the ip address.
- Change the ip address.

Listing 9: Get the configuration file

```
webdeveloper@machine$ sudo su -
root@machine$ cp \
/home/webdeveloper/LinuxClassRepo/Lectures/Week10/WebsiteThatReturnsHTML/WebsiteThatReturnsJSON.conf
    \
/etc/apache2/sites-available/WebsiteThatReturnsJSON.conf
```

Also, let's delete the old default webpage config files so we don't see the stuff we saw in figure 2 anymore. We don't care about the default apache page, that was just there to let us know it installed right.

Listing 10: get rid of default site

```
root@machine$ a2dissite 000-default.conf
root@machine$ rm /etc/apache2/sites-available/000-default.conf
```

Okay, good to go! Let's just restart apache and then the new site will be active!

Listing 11: Restart the apache2 service.

```
webdeveloper@machine$ sudo service apache2 restart
```

If you want to skip ahead and see what it should look like, jump ahead and look at figure 4 and figure 5.

NOTE! Browser *cache* stuff. Since we already visited your.ip.address before, back when it was the Apache default page, your browser may have cached that. So when you try to go to your.ip.address now, you might still see the default page. But we disabled that thing!! How the heck is it still coming up in our browser?? Caching. Thats how. Clear your browser cache. On firefox you do that by hitting CTRL + F5. If that doesn't work for you use google to figure out how. It's easy to, there will be a button to click or something.

7.1 New /etc/apache2/sites-available/WesiteThatReturnsHTML.conf

We've made a couple of changes to the Apache2 config file. Nothing major. Note that now we are serving the website at / and not at */testFlask*. Compare the WSGIScriptAlias line from listing 12 and listing 7.

Listing 12: new config file

```
1  <VirtualHost *:80>
2      # Add machine's IP address (use curl ipinfo.io/ip)
3      ServerName 167.99.63.105
4
5      # Some error logging stuff
6      ErrorLog ${APACHE_LOG_DIR}/error.log
7      LogLevel warn
8      CustomLog ${APACHE_LOG_DIR}/access.log combined
9
10     # Set up the other website
11     WSGIDaemonProcess site2
12     WSGIScriptAlias /Example2
           /home/webdeveloper/WebsiteThatReturnsHTML/my_flask_app.wsgi
13     <Directory /home/webdeveloper/WebsiteThatReturnsHTML>
14           WSGIApplicationGroup site2
15           WSGIProcessGroup site2
16           Options FollowSymLinks
17           AllowOverride None
18           Require all granted
```

```
    </Directory>
</VirtualHost>
```

7.2 my_flask_app.py

This is interesting! Now the website has two functions in it, and two routes! You see the same old one from Example 1, but we've added a second on that returns HTML. This website still stinks, but we're getting somewhere!

Listing 13: new Python file

```
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>What I learned about sed</h1>
    <p><a href="https://www.grymoire.com/Unix/Sed.html">I learned it all
        here!</a>
    <h2>First thing I learned</h2>
    <p> TODO FILL THIS IN </p>
    <h2>Second thing I learned</h2>
    <p> TODO FILL THIS IN </p>
    <h2>Third thing I learned</h2>
    <p> TODO FILL THIS IN </p>
  </body>
</html>
"""


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

7.3 my_flask_app.wsgi

Listing 14: new wsgi file

```
#!/usr/bin/python3

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

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

7.4 __init__.py

Blank as before. If you are curious, go read a Python textbook to understand what this strange file is doing here.

7.5 Results

Here are some images showing your new site in action. Check out the two routes you created in your .py file! Here they are in the browser:

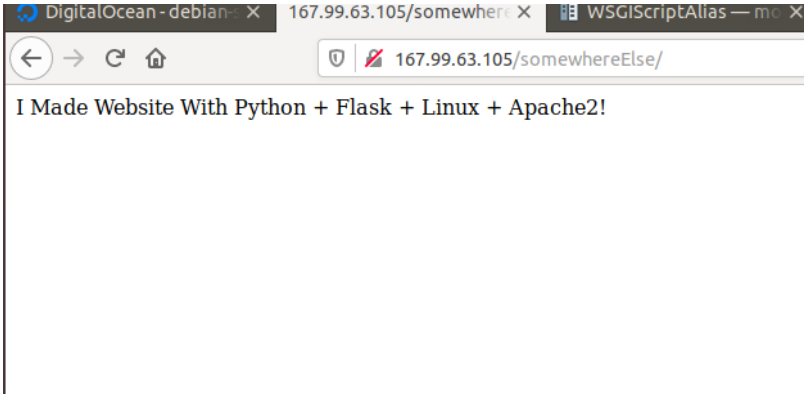

Figure 4: somewhereElse now works

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

8 Example3: A Rest API using the Flask-RESTful Library

8.1 Introduction

Last week you saw a rest api in action when we talked to the github rest api. You saw that a website returned you JSON, and not the HTML you are likely used to receiving. When you go on amazon.com and click stuff, what you're seeing is HTML and CSS. But when you work with a rest api (typically via curl or a simliar tool like Postman), you typically send and receive JSON.

What is JSON? Its a data format that looks like this:

```
{
"name": {
```

```
"first": "melvyn",
"last": "drag"
},
"profession": "programmer",
"favoriteLanguages": ["C++", "Python"]
}
```

What's a REST API? Can REST APIs exchange other data besides JSON?

If you're curious about this, then go find a good book or online lesson about it! It's such interesting stuff but now is not the time.

8.2 What websites offer REST APIs?

A whole bunch. Github. Reddit. Twitter. Facebook. Urban dictionary. NASA. Give it a google. Maybe you'll be bored one day and you can play with a REST api for a new site besides github.

Today we're going to put a REST API on YOUR website.

8.3 What's Flask-RESTful?

It's a python library that allows you to implemet a REST api with the Flask framework. You'll remember we installed it a long time ago as shown in listing 1

Read more here: `https://flask-restful.readthedocs.io/en/latest/`.

8.4 Let's get started

So we'll just see how to return json from a Flask application. Same process as last time. Copy the files from WebsiteThatReturnsJSON to the appropriate locations.

Listing 15: Deploying our third website

```
webdeveloper@machine$ pwd
/home/webdeveloper
webdeveloper@machine$ ls
LinuxClassRepo ExampleFlask
webdeveloper@machine$ cp -r
LinuxClassRepo/Lectures/Week10_Apache2_And_Flask/WebsiteThatReturnsJSON/WebsiteThatReturnsJSON
CodeForWebsiteThatReturnsJSON
webdeveloper@machine$ ls
LinuxClassRepo WebsiteThatReturnsText CodeForWebsiteThatReturnsHTML
    CodeForWebsiteThatReturnsJSON
webdeveloper@machine$ ls CodeForWebsiteThatReturnsJSON
__init__.py my_flask_app.py my_flask_app.wsgi
```

Good, we have the new code in place. Now let's copy over the new configuration file too.

Listing 16: Get the configuration file for example 3

```
webdeveloper@machine$ sudo su -
root@machine$ cp \
/home/webdeveloper/LinuxClassRepo/Lectures/Week10/WebsiteThatReturnsJSON/ExampleFlask.conf
    \
/etc/apache2/sites-available/ExampleFlask.conf
```

Restart apache and it will work!

Listing 17: Restart the apache2 service.

```
webdeveloper@machine$ sudo service apache2 restart
```

8.5 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.

```
"""
Stolen from the flask-restful documentation here:
https://flask-restful.readthedocs.io/en/latest/quickstart.html#a-minimal-api

Super simple RESTful API that handles GET and POST requests.
"""

from flask import Flask
from flask_restful import reqparse, abort, Api, Resource

app = Flask(__name__)
api = Api(app)

TODOS = {
    'todo1': {'task': 'build an API'},
    'todo2': {'task': '?????'},
    'todo3': {'task': 'profit!'},
}

parser = reqparse.RequestParser()
parser.add_argument('task')

# TodoList
# shows a list of all todos, and lets you POST to add new tasks
class TodoList(Resource):
    def get(self):
        return TODOS

    def post(self):
        args = parser.parse_args()
        todo_id = int(max(TODOS.keys()).lstrip('todo')) + 1
        todo_id = 'todo%i' % todo_id
        TODOS[todo_id] = {'task': args['task']}
        return TODOS[todo_id], 201

##
## Actually setup the Api resource routing here
##
api.add_resource(TodoList, '/todos')

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

8.6 Accessing Our REST API

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

Then, on your laptop, make a GET request with cURL

And, you should also be able to make the following POST request:

```
curl -X POST \
  -H "Content-Type: application/json" \
  --data '{"task":"Learn Linux in CS407"}' \
  http://167.99.63.105/myRESTService/todos
```

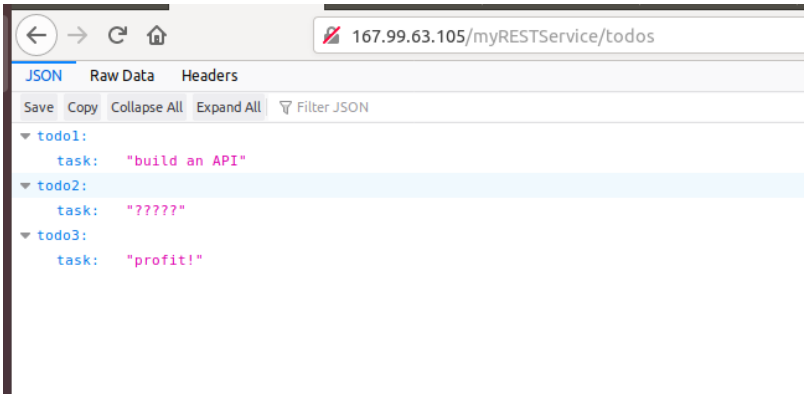

Figure 6: See some json in the browser!

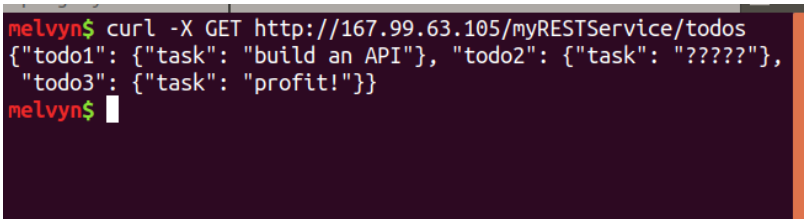

Figure 7: Make a GET request with cURL!

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!

9 Flask and Bigger Websites

Typically your website code will return HTML. I've only shown you how to return HTML strings in example 2, but typically you create HTML templates and populate the template with data. Too complicated to explain, there is no time. Google 'Flask and HTML templates' if you want to know more.

NOTE if class ends early, we can do this together, but I suspect we'll be plenty busy tonight.

10 Databases

For your exam you set up a postgres server with some data in it. I've just shown you how to set up a Flask web application on an Apache server. And you have a good knowledge now about some linux server maintenance things.

In the TODOList REST application we stored all of our data in a dictionary - in the real world this is not how you do it. Can you guess what you do for real websites? You use a DATABASE! You are now poised to go pick up a book on Flask and learn how to wire a database and a website backend together!

Of course we won't do that in this class, but now you have the prerequisite information to go and follow a tutorial on building websites with Flask.

11 sed

sed is one of the more popular command line tools. So far we know grep, vim, cat, mv, ls, etc.. sed is another one of the "big ones".

sed stands for "stream editor" and is a simple little programming language (or tool, depending how you want to view it) that is used for processing text files line by line.

11.1 What to do with sed #1

Sed can be used to replace all occurences of a string in a file. See sedExamples/01.

```
user@machine$ cat oldFile
user@machine$ sed 's/hello//' oldFile
# will replace first occurences of 'hello' with nothing\
# note that a line with two occurences of 'hello' will only have the first
    occurence replaced
user@machine$ cat oldFile
# the file is not changed. sed did not overwrite the original file
user@machine$ sed 's/hello/HELLO/' oldFile
# see first occurences of hello were replaced with HELLO
```

these changes were done in place. If you want to create a new file you can do it like this

```
user@machine$ sed 's/hello/HELLO' <oldFile >newFile
user@machine$ sed 's/hello/HELLO' oldFile >newFile
user@machine$ sed 's/hello/HELLO' 0<oldFile >newFile
# all of the above are equivalent
```

or you can change the input file in place

```
sed -i 's/HELLO/hewwo/' newFile
```

the -i means 'in place'.

11.2 What to do with sed #2

We saw sed can do search and replace first occurence per line. It can also replace all occurrences in a file.

```
user@machine$ sed 's/hello/HELLO/g' oldFile
#note all occurences of hello now say HELLO
```

by adding the g after the search/replace pattern you can replace all occurences.

11.3 What to do with sed #3

Sed can do regular expressions. If you want to capitalize any 'h' at the start of a line, you can do this:

```
user@machine$ sed 's/\(^[h]\)//' oldFile
ello world
ello world world
world hello hello world
```

11.4 What to do with sed #4

You can also extract patterns. This is a weird thing that you can't always do with things like search/replace in microsoft word/excel/whatever other tools you are familiar with.

the same command as before remembers and stores the search pattern in a variable called \1. You can do the same thing with \2, \3, etc for subsequent search patterns. **The search patterns are the regexs enclosed in \(and \)**. In the example above, my regex was looking for a line that starts with h. If you scroll down a bit youll see a regex for a line that starts with an ASCII letter.

```
sed 's/\(^[h]\)/\1\1\1/' oldFile
hhhello world
hhhello world world
world hello hello world
```

You may not be mentally prepared to appreciate what I've just shown you? Maybe one day you'll be trying to do a complicated search and replace task and then you'll remember sed!

Check out this interesting example too!

```
user@machine$ sed 's/\(^[a-zA-Z]\)/\1\1\1/' oldFile
hhhello world
hhhello world world
wwworld hello hello world
```

If I want to wrap the first letter of every line with smileys, for example, I can do this:

```
user@machine$ # emojis!
user@machine$ sed 's/\(^[a-zA-Z]\)/\xF0\x9F\x98\x82\1\xF0\x9F\x98\x82/' oldFile
# look what comes out! Depending on your laptop, the output will be different
    from the output I have as can be seen in the figure below
```

But this may or may not work on your computer, we'll talk about this in depth if there is time this semester.

12 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-phu50a7ft`

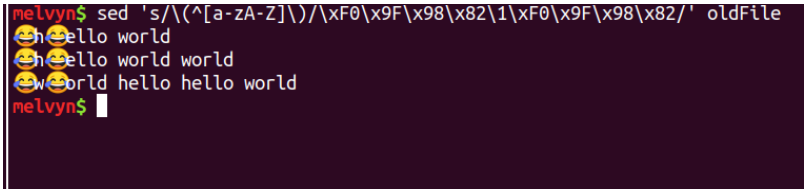

Figure 9: How the heck did he do that? If you already know, I'm very impressed

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