Building APIs with Python

Lab Objective

The objective of this lab is to explore building APIs with Flask. Before we start, review the following terms:

Web Application Framework (Web Framework) - A collection of libraries and modules that enables a web application developer to write applications without having to bother about low-level details such as protocols, thread management, etc.

Flask - Web application framework written in Python. It is developed by Armin Ronacher, who leads an international group of Python enthusiasts named Pocco. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine. Both are Pocco projects. Flask is often referred to as a micro framework. It aims to keep the core of an application simple yet extensible. Flask does not have a built-in abstraction layer for database handling, nor does it have a form of validation support. Instead, Flask supports the extensions to add such functionality to the application.

Web Server Gateway Interface (WSGI) - Adopted as a standard for Python web application development, WSGI is a specification for a universal interface between the web server and the web applications.

Werkzeug - WSGI toolkit, implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.

Jinja2 - A popular templating engine for Python. A web templating system combines a template with a certain data source to render dynamic web pages.

Procedure

1. Open a new terminal.
2. Install the software library “Flask”, which we will need for this script. This will allow us to create APIs.

   `student@beachhead:~$ python3 -m pip install flask`
3. Create a new directory to work in, `/home/student/pyapi/flaskapi/`
   `student@beachhead:~$ mkdir ~/pyapi/flaskapi/`
4. Move into the new directory.

   `student@beachhead:~$ cd ~/pyapi/flaskapi/`
5. Create a new script, `myflask01.py`
   `student@beachhead:~/pyapi/flaskapi$ vim myflask01.py`
6. Copy and paste the following into your script:

```
#!/usr/bin/python3
# An object of Flask class is our WSGI application
from flask import Flask

# Flask constructor takes the name of current
# module (__name__) as argument
app = Flask(__name__)

# route() function of the Flask class is a
# decorator, tells the application which URL
# should call the associated function
```

```
@app.route("/")
def hello_world():
    return "Hello World"

if __name__ == "__main__":
    app.run(port=5006) # runs the application
    # app.run(port=5006, debug=True) # DEBUG MODE
```

25. Save and exit.
26. Run the script `myflask01.py`
    `student@beachhead:~/pyapi/flaskapi$ python3 myflask01.py`
27. Open Firefox (or a new tab) within the remote desktop session.
28. Browse to `http://127.0.0.1:5006/`. The `Hello World` should be returned.
29. Close the Firefox tab containing `Hello World`.
30. Stop your Flask app by clicking on the terminal window it is running in and pressing `CTRL + C`
31. Modern web frameworks use the routing technique to help a user remember application URLs. It is useful to access the desired page directly without having to navigate from the home page. The route() decorator in Flask is used to bind URL to a function. For example, consider the following:

```
@app.route("/hello")
def hello_world():
    return "hello world"
```

35. Above, the URL `/hello` rule is bound to the `hello_world()` function. As a result, if a user visits the `http://localhost:5000/hello` URL, the output of the `hello_world()` function will be rendered in the browser.
36. The `add_url_rule()` function of an application object is also available to bind a URL with a function. As in the above example, `route()` is used. A decorator's purpose is also served by the following representation (again, just for your reading pleasure):

```
def hello_world():
    return "hello world"
app.add_url_rule("/hello", "hello", hello_world)
```

40. Create a new script, `myflask02.py`
    `student@beachhead:~/pyapi/flaskapi$ vim myflask02.py`
41. It is possible to build a URL dynamically by adding variable parts to the rule parameter. This variable part is marked as `<variable-name>`. It is passed as a keyword argument to the function with which the rule is associated. In your next script, the rule parameter of route() decorator contains `<name>` variable part attached to URL '/hello'. Hence, if the http://localhost:5000/hello/Zuul is entered as a URL in the browser, `Zuul` will be supplied to `hello()` function as argument.
42. Copy and paste the following into `myflask02.py`

```
#!/usr/bin/python3
from flask import Flask
app = Flask(__name__)

@app.route("/hello/<name>")
def hello_name(name):
    return f"Hello {name}"
    ## V2 STYLE STRING FORMATTER - return "Hello {}".format(name)
    ## OLD STYLE STRING FORMATTER - return "Hello %s!" % name

if __name__ == "__main__":
    app.run(port=5006, debug = True)
```

55. Save and exit.
56. Run your script, `myflask02.py`
    `student@beachhead:~/pyapi/flaskapi$ python3 myflask02.py`
57. Run the script `myflask02.py`
    `student@beachhead:~/pyapi/flaskapi$ python3 myflask02.py`
58. Open Firefox (or a new tab) within the remote desktop session.
59. Browse to `http://127.0.0.1:5006/hello/Worf%20Son%20of%20Mogh`
60. The `Hello Worf Son of Mogh` should be returned. Notice that we used ASCII characters, `%20` to indicate blank spaces.
61. Close the Firefox tab containing `Hello Worf Son of Mogh`.
62. Stop your Flask app by clicking on the terminal window it is running in and pressing `CTRL + C`
63. The url_for() function is very useful for dynamically building a URL for a specific function. The function accepts the name of a function as first argument and one or more keyword arguments, each corresponding to the variable part of URL.
64. Let's write a script to demonstrate the url_for() function.

    `student@beachhead:~/pyapi/flaskapi$ vim myflask03.py`
65. Copy and paste the following into `myflask03.py`

```
66. #!/usr/bin/python3
67. from flask import Flask, redirect, url_for
68. app = Flask(__name__)
69.
70. @app.route("/admin")
71. def hello_admin():
72.     return "Hello Admin"
73.
74. @app.route("/guest/<guesty>")
75. def hello_guest(guesty):
76.     return f"Hello {guesty} Guest"
77.     #V2 FORMATTER - return "Hello {} Guest".format(guesty)
78.     #OLD FORMATTER - return "Hello %s as Guest" % guesty
79.
80. @app.route("/user/<name>")
81. def hello_user(name):
82.     ## if you go to hello_user with a value of admin
83.     if name =="admin":
84.         # return a 302 response to redirect to /admin
85.         return redirect(url_for("hello_admin"))
86.     else:
87.         # return a 302 response to redirect to /guest/<guesty>
88.         return redirect(url_for("hello_guest",guesty = name))
89.
90. if __name__ == "__main__":
91.     app.run(port=5006, debug = True)
```

92. Save and exit.
93. Run the script `myflask03.py`.
    `student@beachhead:~/pyapi/flaskapi$ python3 myflask03.py`
94. The above script has a function user(name) which accepts a value to its argument from the URL. The user() function checks if an argument received matches 'admin' or not. If it matches, the application is redirected to the hello_admin() function using url_for(), otherwise to the hello_guest() function passing the received argument as guest parameter to it.

95. Open Firefox (or a new tab) within the remote desktop session and try to browse to `http://127.0.0.1:5006/user/admin

96. Now try to go to, `http://127.0.0.1:5006/user/Wolverine`
97. Close the Firefox tab containing `Hello Wolverine`.
98. Stop your Flask app by clicking on the terminal window it is running in and pressing `CTRL + C`
99. Now let's try returning a separate HTML document. By default, documents are rendered from a sub directory called `templates`, so start by creating that directory.
    `student@beachhead:~/pyapi/flaskapi$ mkdir templates/`
100. By default, the Flask route responds to the GET requests. However, this preference can be altered by providing methods argument to route() decorator. In order to demonstrate the use of POST method in URL routing, first let us create an HTML form and use the POST method to send form data to a URL. Create a new script called `postmaker.html`
    `student@beachhead:~/pyapi/flaskapi$ vim templates/postmaker.html`
101. Create the following within `postmaker.html`

```
<html>
   <body>
      <form action = "http://localhost:5006/login" method = "POST">
         <p>Enter Name:</p>
         <p><input type = "text" name = "nm" /></p>
         <p><input type = "submit" value = "submit" /></p>
      </form>
   </body>
</html>
```

<!DOCTYPE html>

Enter Name:

submit

1. Save and exit.
2. Create a new script called `myflask04.py`
   `student@beachhead:~/pyapi/flaskapi$ vim myflask04.py`
3. Create the following within `myflask04.py`

```
#!/usr/bin/python3
"""Alta3 APIs and HTML"""

## best practice says don't use commas in imports
# use a single line for each import
from flask import Flask
from flask import redirect
from flask import url_for
from flask import request
from flask import render_template

app = Flask(__name__)
## This is where we want to redirect users to
@app.route("/success/<name>")
def success(name):
    return f"Welcome {name}\n"
# This is a landing point for users (a start)
```

```
@app.route("/") # user can land at "/"
@app.route("/start") # or user can land at "/start"
def start():
    return render_template("postmaker.html") # look for templates/postmaker.html
# This is where postmaker.html POSTs data to
# A user could also browser (GET) to this location
@app.route("/login", methods = ["POST", "GET"])
def login():
    # POST would likely come from a user interacting with postmaker.html
    if request.method == "POST":
        if request.form.get("nm"): # if nm was assigned via the POST
            user = request.form.get("nm") # grab the value of nm from the POST
        else: # if a user sent a post without nm then assign value defaultuser
            user = "defaultuser"
    # GET would likely come from a user interaacting with a browser
    elif request.method == "GET":
        if request.args.get("nm"): # if nm was assigned as a parameter=value
            user = request.args.get("nm") # pull nm from localhost:5060/login?nm=larry
        else: # if nm was not passed...
            user = "defaultuser" # ...then user is just defaultuser
    return redirect(url_for("success", name = user)) # pass back to /success with val
for name
if __name__ == "__main__":
    app.run(port=5006)
```

44. Save and exit.
45. Run the script `myflask04.py`
    `student@beachhead:~/pyapi/flaskapi$ python3 myflask04.py`
46. Let's try using CURL to access our links. CURL is short for 'see-URL', and allows us to access HTTP resources from the CLI. It likely is installed, but it won't hurt to double-check.

    `student@beachhead:~/pyapi/flaskapi$ sudo apt install curl`
47. Now that the server is running, try to curl against your API. The `-L` is required to follow the 3xx HTTP responses (forwarded). This will send a GET to `/login`
    `student@beachhead:~/pyapi/flaskapi$ curl http://localhost:5006/login?nm=Wolverine -L`
48. You should receive back, "Hello Wolverine"
49. Try it again, only this time, do not supply a value for `nm=`
    `student@beachhead:~/pyapi/flaskapi$ curl http://localhost:5006/login -L`
50. You should recieve back, "Hello defaultuser"
51. If you're using the remote desktop GUI, hop to it, and open Firefox. Navigate to `http://localhost:5006/start`
52. Fill in your name and submit.
53. Form data is POSTed to the URL in action clause of form tag. `http://localhost:5006/login` is mapped to the login() function. Since the server has received data by POST method, value of `nm` parameter obtained from the form data is obtained by `user = request.form.get("nm")`. If you'd like, you can capture this process by using the Wireshark application on the remote desktop.
54. Close Firefox on your remote GUI.
55. Stop your Flask app by clicking on the terminal window it is running in and pressing `CTRL + C`
56. That's it for this lab!
57. If you're tracking your code in GitHub, issue the following commands:

- `cd ~/pyapi`
- `git add *`
- `git commit -m "My first Flask apps"`
- `git push origin master`

Flask APIs and Cookies

Lab Objective

The objective of this lab is to explore Flask APIs and interacting with cookies. A cookie is stored on a client's computer in the form of a text file. Its purpose is to remember and track data pertaining to a client's usage for better visitor experience and site statistics.

A Request object contains a cookie's attribute. It is a dictionary object of all the cookie variables and their corresponding values that a client has transmitted. In addition to that, a cookie also stores its expiry time, path, and domain name of the site.

In Flask, cookies are set on response object. Use `make_response()` function to get response object from return value of a view function. After that, use the `set_cookie()` function of response object to store a cookie.
Reading back a cookie is easy. The `get()` method of request.cookies attribute is used to read a cookie.

Procedure

1. Open a new terminal.
2. Move into (or create) a new directory to work in, `/home/student/pyapi/flaskapi/`
   `student@beachhead:~$ mkdir -p /home/student/pyapi/flaskapi/`
3. Move into your new directory

   `student@beachhead:~$ cd ~/pyapi/flaskapi/`
4. Now create a script called `milkncookies.py`
   `student@beachhead:~/pyapi/flaskapi$ vim milkncookies.py`
5. Copy the following into your Python script, `milkncookies.py`

```
#!/usr/bin/env python3
from flask import Flask
from flask import make_response
from flask import request
from flask import render_template
from flask import redirect
from flask import url_for

app = Flask(__name__)

# entry point for our users
# renders a template that asks for their name
# login.html points to /setcookie
@app.route("/login")
@app.route("/")
def index():
    return render_template("login.html")

# set the cookie and send it back to the user
```

```
@app.route("/setcookie", methods = ["POST", "GET"])
def setcookie():
    # if user generates a POST to our API
    if request.method == "POST":
        if request.form.get("nm"): # if nm was assigned via the POST
        #if request.form["nm"] <-- this also works, but returns ERROR if no nm
            user = request.form.get("nm") # grab the value of nm from the POST
        else: # if a user sent a post without nm then assign value defaultuser
            user = "defaultuser"

        # Note that cookies are set on response objects.
        # Since you normally just return strings
        # Flask will convert them into response objects for you
        resp = make_response(render_template("readcookie.html"))
        # add a cookie to our response object
                        #cookievar #value
        resp.set_cookie("userID", user)

        # return our response object includes our cookie
        return resp

    if request.method == "GET": # if the user sends a GET
        return redirect(url_for("index")) # redirect to index

# check users cookie for their name
@app.route("/getcookie")
def getcookie():
    # attempt to read the value of userID from user cookie
    name = request.cookies.get("userID") # preferred method

    # name = request.cookies["userID"] # <-- this works but returns error
                                       # if value userID is not in cookie

    # return HTML embedded with name (value of userID read from cookie)
    return f'<h1>Welcome {name}</h1>'

if __name__ == "__main__":
    app.run(port=5006)
```

63. Save and exit.
64. When you run your application, look for templates within the folder, `templates/`, by default.
    `student@beachhead:~/pyapi/flaskapi/$ mkdir templates/`
65. Within the templates folder, create a template called `login.html`.
    `student@beachhead:~/pyapi/flaskapi$ vim templates/login.html`
66. Copy and paste the following into your template, `login.html`.

```
<!doctype html>
<html>
   <body>
      <form action = "/setcookie" method = "POST">
         <p><h3>Enter userID</h3></p>
         <p><input type = "text" name = "nm"/></p>
         <p><input type = "submit" value = 'Login'/></p>
      </form>
   </body>
```

```
</html>
```

77. Save and exit.

78. We are also going to need a template called `readcookie.html`. This will contain a simple redirection link to our getcookie resource.
    `student@beachhead:~/pyapi/flaskapi$ vim templates/readcookie.html`
79. Copy and paste the following into your template, `readcookie.html`.

```
<!doctype html>
<html>
   <body>
      <h1> <p> A cookie has been set on your system! </p>
           <p> <a href="/getcookie">Click here to Read The Cookie</a> </p>
      </h1>
   </body>
</html>
```

88. Save and exit.

89. Run your script, `milkncookies.py`
    `student@beachhead:~/pyapi/flaskapi$ python3 milkncookies.py`
90. Open a new Firefox tab to `http://127.0.0.1:5006/login`
91. Enter your `userID` (it can be anything you want), then press the button to submit your userID.
92. Click the link to read your cookie.

93. Close Firefox.

94. Stop your Flask app by clicking on the terminal window it is running in and pressing `CTRL + C`.
95. Start your application one more time.

    `student@beachhead:~/pyapi/flaskapi$ python3 milkncookies.py`
96. Try navigating straight to `http://127.0.0.1:5006/getcookie`
97. Notice that the service is able to determine your identity based on a cookie stored on your browser. Let's purge all cookies to be sure that is what is going on.

98. Do not stop your Flask service. Within Firefox click the tiny books icon in the upper right hand corner > "History" > "Clear Recent History…"

99. Ensure that cookies is checked, and then press the button marked, "Clear Now"

100. One again, navigate to `http://127.0.0.1:5006/getcookie`
101. This time the page seems broken! That's because this part of our web app tries to interact with the cookie stored within our browser. Once again, navigate to the entry point of our web app, `http://127.0.0.1:5006/`
102. Enter your `userID` (something different this time), then press the button to submit your userID.
103. Click the link to navigate to, `http://127.0.0.1:5006/getcookie`, and read your cookie.
104. Your cookie should now reflect your new identity.

105. You might also try 'hacking' your cookie. Within Firefox click the three lines icon in the upper right hand corner > "Web Developer" > "Storage Inspector" > "Storage" > "Cookies". You can now inspect the value of userID within the cookie set on your machine. Don't edit the key userID, but edit the value of userID so it no longer is your sign in name, but instead `A3Research`. After you've done this, try navigating to `http://127.0.0.1:5006/getcookie` notice that the page says you're signed in as `A3Research`. We can solve this problem (a client hacking their cookie information) by studying 'session cookies' (i.e. encrypted cookies) in later labs.
106. If you're tracking your code in GitHub, issue the following commands:

- `cd ~/pyapi`
- `git add *`
- `git commit -m "serving cookies"`
- `git push origin master`
- Type username and password