SevenBridges

Python Workshop

Part III

Content

- HTTP
- Requests
- Flask
- Gunicorn
- Testing
- SevenTweets

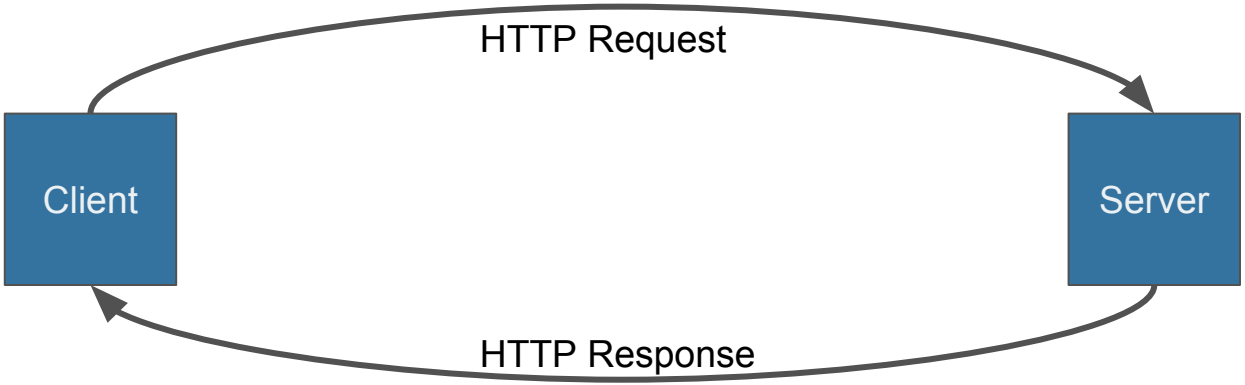
HTTP

Short overview of protocol

HTTP

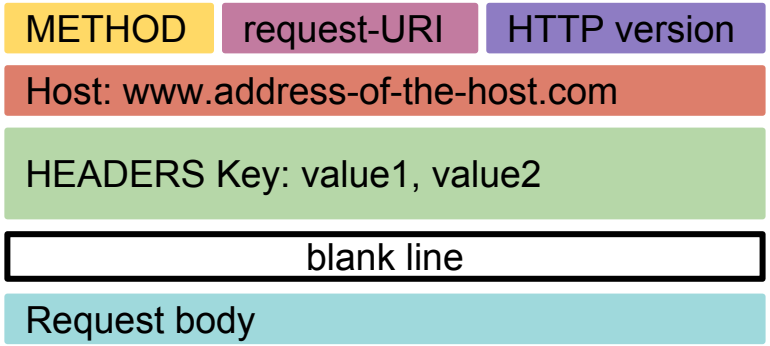
Hypertext Transfer Protocol

- request–response protocol i
- client–server computing model.



HTTP Requests

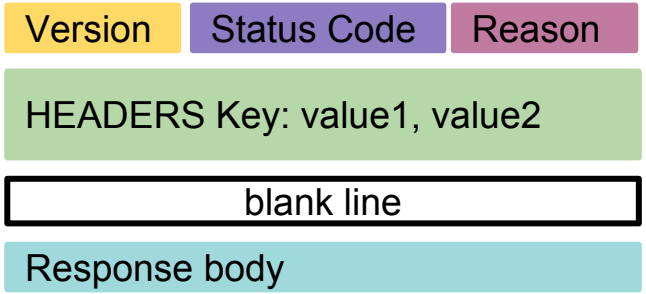
```
GET /docs/index.html HTTP/1.1
Host: www.nowhere123.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
(blank line)
BODY OF THE REQUEST (sometimes)
```



HTTP Response

```
HTTP/1.1 200 OK
Date: Sun, 18 Oct 2009 08:56:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sat, 20 Nov 2004 07:16:26 GMT
ETag: "10000000565a5-2c-3e94b66c2e680"
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug

(blank line)
<html><body><h1>It works!</h1></body></html>
```



HTTP Methods

- GET
- POST
- PUT
- DELETE
- PATCH
- HEAD
- TRACE
- OPTIONS
- CONNECT

Content

- ~~HTTP~~
- Requests
- Flask
- Gunicorn
- Testing
- SevenTweets

requests

the only Non-GMO HTTP library for Python, safe for human consumption

Requests

Easy to use:

conda install -c anaconda requests

```
import requests

r = requests.put('http://httpbin.org/put', data={'key': 'value'})
print(r)
r = requests.delete('http://httpbin.org/delete')
print(r)
r = requests.head('http://httpbin.org/get')
print(r)
r = requests.options('http://httpbin.org/get')
print(r)
```

```
(workshop)➜  http-requests python script.py
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
```

Requests

- Headers
  - `headers` parameter as dictionary
- URL Parameters
  - `params` parameter ad dictionary
- Body
  - `data` parameter

Examples of requests usage ==>

Content

- ~~HTTP~~
- ~~Requests~~
- Flask
- Gunicorn
- Testing
- SevenTweets

Flask

What is Flask

“Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions.”

Flask install

conda install -c anaconda flask

Or

pip install Flask

Flask - Hello World

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

@app.route('/')
def hello_world():
   return 'Hello, World!'
```

- Importing Flask
- Flask(__name__) - name of the application's module or package. Flask uses this to determine paths for import of templates, statics, etc.
- @app.route('/') - what URL should trigger our function.
- hello_world() - our handler of the specified route

Flask - running the app

Multiple ways to do this:

1. app.run()
    a. In this case we start the python script regularly
        i. python hello_world.py
        ii. python -m hello_world
1. $ flask run
    a. This requires that we previously set up `FLASK_APP` environment variable
        i. export FLASK_APP=hello_world.py
        ii. set FLASK_APP=hello_world.py (for Windows)
2. $ python -m flask run
    a. Also requires that we set up `FLASK_APP` environment variable
        i. export FLASK_APP=hello_world.py
        ii. set FLASK_APP=hello_world.py (for Windows)

Flask - routing

Routing is done based on the parameters passed to the *route*() decorator.

Try some routes on your own:

- Index - /
- Hello - /hello

Mind the trailing slash, what happens with or without it?

Also, you can define which HTTP methods you want your route to support - just add another parameter to *route*() decorator - *methods*.

Note that when you add GET method, HEAD and OPTIONS are automatically added for you by Flask.

Flask - route variables

Variable can be part of the URL by defining it with <variable_name>. These variables become part of the parameters passed to the function that is decorated. And on top of that you can use converters!

```
@app.route("/hello/<username>")
def hello(username):
    return "hello " + username + "!"
```

```
# or use <converter:variable_name>
@app.route("/hello/<string:username>")
def hello(username):
  return "hello " + username + "!"
```

string	accepts any text without a slash (the default)
int	accepts integers
float	like int but for floating point values
path	like the default but also accepts slashes
any	matches one of the items provided
uuid	accepts UUID strings

Flask - processing request data

In order to access the request data Flask uses a global *request* object. It is a global object but it is simply proxy for the request you are processing at that moment.

```
from flask import request

@app.route("/hello/")
def hello_again():
   return " ".join([
       "You have gone to:",
       request.url,
       "Your request headers were:",
       str(request.headers)
   ])
```

Request attributes:

- args
- form
- headers
- host
- method
- values
- ...

Flask - response data

Multiple ways to create response:

- Simply return string you want to send back, and Flask wraps it for you with what is needed. If the return value is a string it's converted into a response object with the string as response body, a 200 OK status code and a text/html mimetype.
- Return tuple with string and status code
- Use *flask.make_response* helper function which creates the response object
  - Use it with one argument to make it body of your response, then change whatever you want about it
  - Use it with tuple where second argument are headers (as a list or dictionary)
  - Use it with *render_template* function
- Use flask.Response to create Response object

*More information about make_request and render_template functions, as well as working with templates and Jinja2 on: http://flask.pocoo.org/ , but they will not be used during the workshop.

Flask - blueprints

When making a modular application, it is best to use flask blueprints. For more information on those - visit: http://flask.pocoo.org/docs/0.12/blueprints/

```
from flask import Flask
from blueprints.blueprint import blueprint

app = Flask(__name__)
app.register_blueprint(blueprint)


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

```
from flask import Blueprint

blueprint = Blueprint('blueprint', __name__)


@blueprint.route('/blueprint')
def hello():
   return "Hello from blueprint!"
```

Content

- ~~HTTP~~
- ~~Requests~~
- ~~Flask~~
- Gunicorn
- Testing
- SevenTweets

Gunicorn

(conda install -c anaconda gunicorn)

Gunicorn

Gunicorn is Python WSGI HTTP Server for UNIX. It's a pre-fork worker model.

- Broadly compatible with various web frameworks,
- Simple to implement,
- Light on server resources
- Fairly speedy.

WSGI => look at: PEP 333 -- Python Web Server Gateway Interface v1.0
https://www.python.org/dev/peps/pep-0333/

*For more detailed information about gunicorn and its capabilities check out their official site: http://gunicorn.org/

Gunicorn and Flask

Flask is under the hood implemented as WSGI application so after installing gunicorn you can simply start the Flask application by providing the parameter where it is defined:

```
$ gunicorn my_app.server:app
```

Content

- ~~HTTP~~
- ~~Requests~~
- ~~Flask~~
- ~~Gunicorn~~
- Testing
- SevenTweets

Testing

With py.test

(conda install -c anaconda pytest)

Pytest

Pytest is a testing framework for easy creation of small tests that also has capabilities to allow for scaling of the tests to application or library level.

```
# content of test_sample.py
def inc(x):
    return x + 1


def test_answer():
    assert inc(3) == 5
```

Pytest

- Detailed info on failing assert statements
  - Pytest uses simple python assert statements, provides detailed messages on failure and provides you with a way to customize the messages:

    ```
    assert x==y, "they are not equal"
    ```

- Automatic discovery of test modules and functions
  - For current (or provided path) search for `test_*.py` or `*_test.py` files
    - From those files, collect test items:
      - `test_` prefixed test functions outside of class
      - `test_` prefixed test functions inside `Test` prefixed test classes (without an __init__ method)

Pytest - fixtures

```
# content of test_smtpsimple.py
import pytest

@pytest.fixture
def smtp():
    import smtplib
    return smtplib.SMTP("smtp.gmail.com")

def test_ehlo(smtp):
    response, msg = smtp.ehlo()
    assert response == 250
```

- Pytest provides modular fixtures that are auto-discovered so they get automatically passed to the test functions
- Fixtures can have *scope* parameter
- They can be placed in conftest.py file so that tests from multiple test modules in the directory can access the fixture function
- Fixtures can have *params* parameter which will ensure all tests using it to be called as many times as there are fixtures

unittest.mock module

Contains three important components:

- **Mock** - a flexible mock object intended to replace the use of stubs and test doubles throughout your code. Mocks are callable and create attributes as new mocks when you access them. Accessing the same attribute will always return the same mock. Mocks record how you use them, allowing you to make assertions about what your code has done to them.
- **MagicMock** - a subclass of Mock with all the magic methods pre-created and ready to use
- **patch** method - acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the target is patched with a new object. When the function/with statement exits the patch is undone

They allow you to replace parts of your system under test with mock objects and make assertions about how they have been used.

Content

- ~~HTTP~~
- ~~Requests~~
- ~~Flask~~
- ~~Gunicorn~~
- ~~Testing~~
- SevenTweets

SevenTweets

Project specification

SevenTweets

SevenTweets is a network of mini services that should resemble twitter. Each service that will be part of SevenTweets can have different implementation but will be built to support a standardised API so that the communication between services can be possible. Every node in SevenTweets would:

- Be a separate service
- Be able to work with *tweets* (published message)
- Have its own identificator (username)
- Have its own local storage for *tweets* originating from that node
- Have to know addresses of all other nodes
- Perform notifications to other nodes when starting/shutting down

SevenTweets - endpoints (and homework)

- GET /tweets
  - Returns: [
    {"id": 1, "name": "zeljko", "tweet": "this is tweet"}, ...]
  - Status code: 200

- GET /tweets/<*id*>
  - Returns: {‘id’:’...’, ‘name’:’...’, ‘tweet’:’...’ }
  - Status code : 200

- POST /tweets
  - Body: {'tweet': '...'}
  - Returns: {‘id’:’...’, ‘name’:’...’, ‘tweet’:’...’ }
  - Status code: 201

- DELETE /tweets/<*id*>
  - Status code 204

Thank you!