Flask Basics

An introduction to a lightweight Python web application framework

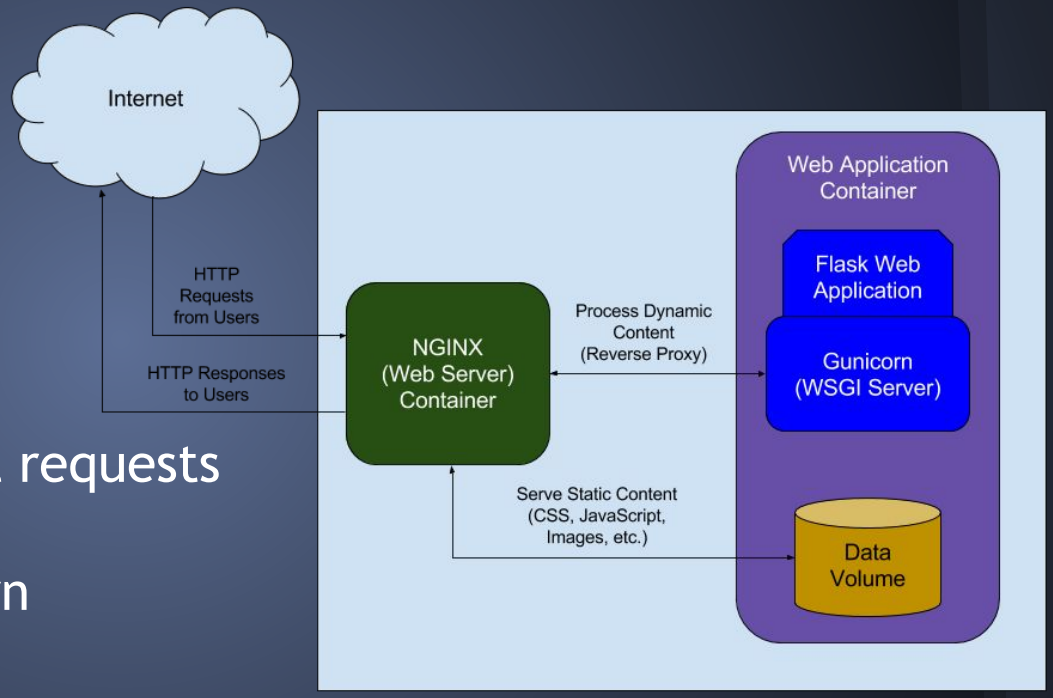
Inland Empire Python Users Group
January 21, 2020

John Sheehan

Web Server Stack

Web Server / Reverse Proxy

- Load balancing
- SSL endpoint
- Serves static content
- Examples: nginx, Apache



HTTP server

- Forwards dynamic content requests
- Serves dynamic content
- Examples: uWSGI, Gunicorn

Application Server

- Communicates with back-end resources like databases
- Processes dynamic content requests
- Examples: Flask, Django, CherryPy

Installation

Setup virtual environment:

```
python3 -m venv venv
```

Activate virtual environment:

```
venv\Scripts\activate      # Windows
source venv/bin/activate   # Mac/Linux
```

Install Flask:

```
pip install Flask
```

Basic Routing

```
from flask import Flask

app = Flask(__name__)

def index():
    return 'Hello World'

app.add_url_rule('/', 'index', index)


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

Basic Routing - Decorator

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello World!!!'


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

Basic Routing - Responses

```
from flask import Flask, Response

app = Flask(__name__)

@app.route('/')
def index():
    return Response('Hello World again!', 200)

@app.route('/about')  # A trailing slash makes a difference!
def about():
    return Response('This is my flask app!', 200)

if __name__ == "__main__":
    app.run(debug=True, port=8080)
```

Basic Routing - GET Requests

```
from flask import Flask, Response, request

app = Flask(__name__)

@app.route('/')
def index():
    arg = request.args.get('name')
    if arg is None or len(arg) == 0:
        arg = 'World'
    return Response('Hello ' + arg, 200)

if __name__ == "__main__":
    app.run(debug=True,port=8080)
```

Basic Routing - Static Files

```
from flask import Flask, Response

app = Flask(__name__)

@app.route('/')
def index():
    html_body = """
        <h1>This is Python!!!</h1>
        <img src="static/python.png">
    """
    return Response(html_body, 200)

if __name__ == "__main__":
    app.run(debug=True,port=8080)
```

Basic Routing - Dynamic URLs

```
from flask import Flask, Response, url_for

app = Flask(__name__)

@app.route('/')
def index():
    image_url = url_for('static', filename='python.png')
    html_body = """
        <h1>This is Python!!!</h1>
        <img src="{0}">
    """.format(image_url)
    return Response(html_body, 200)

if __name__ == "__main__":
    app.run(debug=True,port=8080)
```

Basic Routing - Forms (POST)

- Use `request.form['param']` instead of `request.args.get('param')` to retrieve data from POST request

```
pwd = request.form['password']
```

- Use the `redirect` method to change the request URL

```
return redirect(url_for('login'), 302)
```

Basic Routing - Logins

- There are several flask plug-in packages available for handling authentication. Flask-Login is one of them

```
pip install Flask-Login
```

- Flask-Login does NOT dictate what a user login actually consists of. That is up to the developer to use something appropriate for the application. It only connects users to sessions using a login manager.

- NEVER store passwords in plain-text. If you must store them, use password hashes instead.

```
from werkzeug.security import generate_password_hash
secret_password = generate_password_hash('123')
```

Basic Routing - Authorizations

- Flask-Login requires implementation of a User class to define what a ‘user’ means.

```
class User(flask_login.UserMixin):
    def __init__(self, userid):
        self.id = userid
```

- Routes can be secured by using a decorator:

```
@flask_login.login_required
def secret_webpage():
    return Response('This is a secret!', 200)
```

- A handler can be specified to route unauthorized requests to:

```
@login_manager.unauthorized_handler
```

Bonus...
Upload a Flask app to AWS Lambda

Zappa is a python package that can upload your flask application to the AWS cloud, where it can be run on-demand.

```
pip install zappa
```

Zappa will do the following:

- Zip up your application files
- Upload your files to an AWS S3 bucket
- Deploy your application as an AWS Lambda
- Create an AWS API Gateway endpoint
- Add an AWS Cloudwatch event rule that keeps the Lambda “warm”

Your application can also be integrated with AWS Route 53 to use a custom domain.

Bonus...
Upload a Flask app to AWS Lambda

Zappa installs as a script. To use it, run it at a command prompt.

- Create a zappa_settings.json file:

```
zappa init
```

- Do an initial deployment that creates all of the AWS services:

```
zappa deploy dev
```

- To deploy any updates, use the update directive:

```
zappa update dev
```

- To remove the AWS Lambda and API Gateway:

```
zappa undeploy dev
```

Production Notes

- For any Flask applications that are more than trivial, it is best to use a templating framework like Jinja2 to keep your code clean.

- For production use (deployed to the internet for mass consumption), do NOT use the flask built-in web server - it is intended to development testing only. Utilize a production worthy WSGI web server to front your Flask application.

- Application servers like Flask are not intended to serve static content. Use a web server for that if possible.

Resources

- Flask project site:
  https://palletsprojects.com/p/flask/
- Zappa github:
  https://github.com/Miserlou/Zappa
- Corey Schafer's 15 part YouTube Series on Flask:
  https://www.youtube.com/playlist?list=PL-osiE80TeTs4UjLw5MM6OjgkjFeUxCYH

Discussion