Урок 3. PEP3333 — WSGI

Вернёмся к той странной аббревиатуре, которую мы видим в консоли, когда запускаем Flask-приложение. WSGI — Web Service Gateway Interface. Это стандарт, впервые он был описан в PEP333 в 2003 году и затем доработан в PEP3333 в 2010 году. Стандарт описывает, как должны взаимодействовать между собой Python-приложение и веб-сервер. Мотивация появления этого стандарта такова: существовало большое количество веб-фреймворков, все они по-разному понимали то, как должно происходить взаимодействие с сервером. Чтобы как-то причесать всё это разнообразие и упростить жизнь разработчикам, которым при выборе веб-фреймворка ещё приходилось думать о том, совместим ли он с нужным сервером, решено было стандартизировать это.

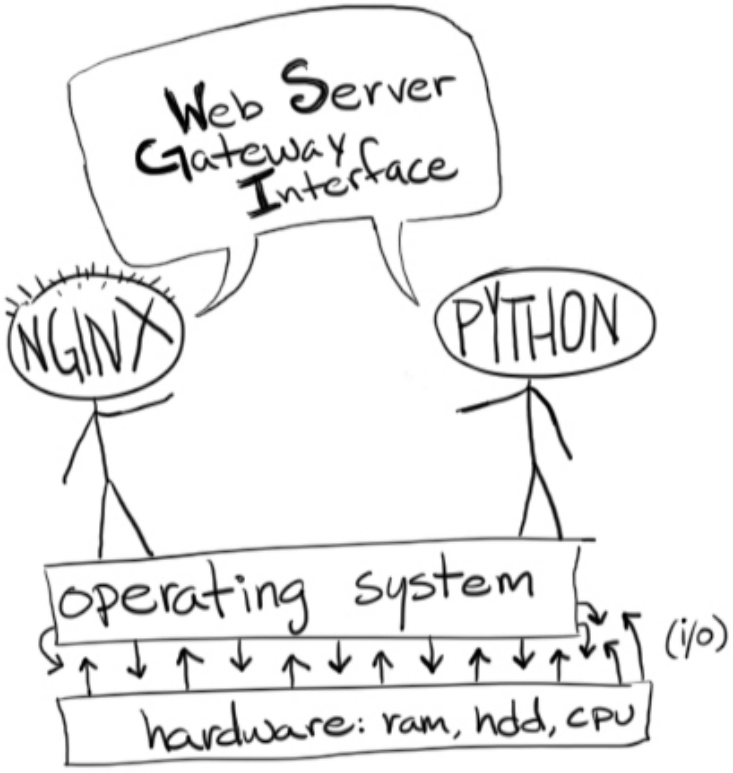
Что было предложено сделать? Разделить часть, отвечающую за бизнес-логику, и выставить наружу простую точку входа, обычный collable-объект, который сможет вызвать WSGI-совместимый веб-сервер. Вот как это может выглядеть:

```
def application(environ, start_response):
    status = '200 OK'
    response_headers = [('Content-type','text/plain')]
    start_response(status, response_headers)
    return ['Hello world!\n'.encode('utf-8')]
```

По стандарту WSGI-приложение должно удовлетворять следующим критериям:

1) должно быть вызываемым объектом;
2) должно принимать два параметра — словарь с переменными окружения (почти как CGI, даже нейминг переменных похож) и функцию «обработчик запроса»;
3) должно вызывать внутри себя обработчик запроса и передавать туда код ответа (строкой) и заголовки;
4) должно вернуть итерируемый объект с телом ответа.

Вся магия веб-фреймворков включается именно в этот момент. За нас парсится словарь переменных окружения; мы понимаем, какой http-метод вызвал нас, какой ресурс запрашивают, какие параметры в него передали и т. д. Всё это передаётся в функцию, которая обрабатывает запросы на конкретном endpoint. А в конце то, что возвращает эта функция, отдаётся обратно в нужном виде.



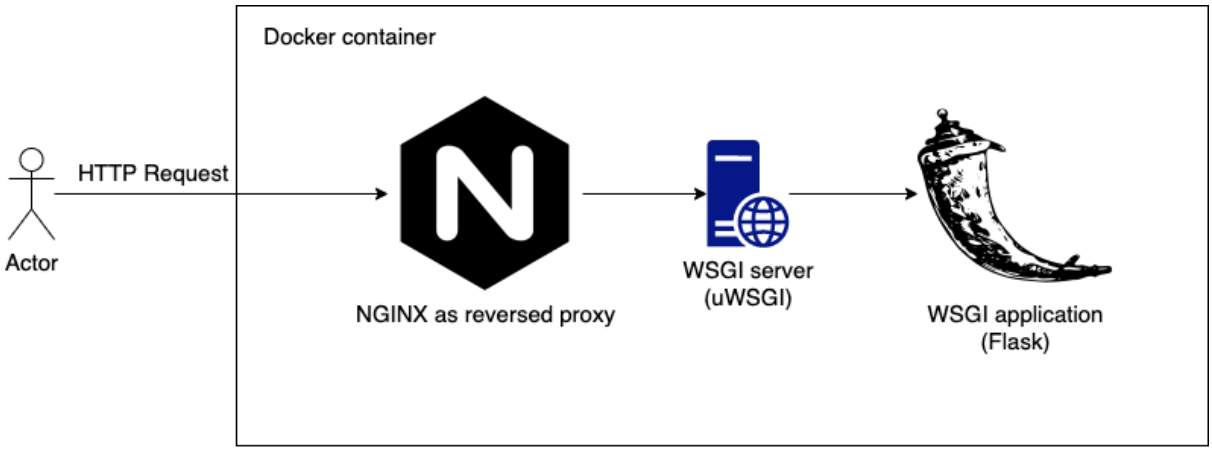
Тут возникает логичный вопрос: кто-то же должен вызывать эту функцию — обработчик — так кто? Мы знаем, что это может сделать сам Apache (если поставить нужный плагин) или ряд других приложений — WSGI-сервера. Наиболее распространены в Python-экосистеме два — uWSGI и Gunicorn. Первый имеет крайне обширные настройки, второй прост в настройках и, возможно, чуть производительнее (я видел несколько бенчмарков, но результаты в них достаточно сомнительны).

Не будем пытаться изобразить на «Питоне» WSGI-сервер, занятие это достаточно бестолковое. Но опишем на словах и немного псевдокодом, что там происходит.

- Прежде всего серверу нужно сформировать словарь переменных окружения.
- Далее нужно описать функцию «обработчик запроса» (start_response).
- Передать всё это в WSGI-приложение.
- Результат WSGI-сервер отправляет клиенту. В зависимости от того, какой сервер мы используем, ответ может быть отослан разными способами. Если это Apache, то вернуть по HTTP. Если наш сервер — CGI-скрипт, написать ответ в stdout.

```
environ = get_environ()
response = application(environ, start_response)
return response
```

Окей, разобравшись с тем, что собой представляет WSGI-сервер на концептуальном уровне, давайте перейдём к практическому и подключим его к нашему приложению.



В результате мы хотим получить что-то похожее: приложение, которое будет работать в докер-контейнере. NGINX, который будет обрабатывать входящий запрос от пользователя, далее перенаправит его в WSGI-server; мы будем использовать uWSGI. Он же, в свою очередь, запустит uWSGI-приложение, которое и обработает запрос.

Тут, кстати, вполне справедливо можно спросить о том, зачем нам нужен NGINX, если WSGI-сервер сам может обработать http-запрос, как минимум в теории. Всё дело в той жуткой эффективности, с которой NGINX отдаёт статические файлы.
Распространённая практика отдавать генерацию динамического контента на руки WSGI-сервера и обслуживать статические файлы средствами NGINX.

Быстро напишем какой-нибудь hello-world на Flask:

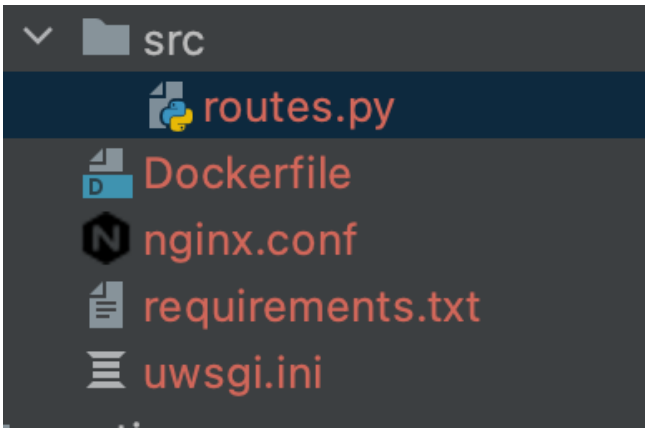
```
from flask import Flask, jsonify

app = Flask(__name__)


@app.route('/hello')
@app.route('/hello/<username>')
def hello_world(username='username'):
   return jsonify(message="hello", name=username)
```

Абсолютно ничего интересного тут нет, однако заметим, что теперь отсутствует такой привычный app.run().

Далее создадим несколько файлов, пока пустых:



Тут уже знакомые нам файлы: докерфайл, реквайрементс, конфиг NGINX и uWSGI.

Приступим к описанию докерфайла:

```
FROM python:3.8

RUN apt-get update && apt-get install -y python3-dev nginx \
   && rm -rf /var/lib/apt/lists/*

COPY requirements.txt /app/
RUN pip install -r /app/requirements.txt

COPY src/ /app/
COPY nginx.conf /etc/nginx/nginx.conf
COPY uwsgi.ini /etc/uwsgi/uwsgi.ini


WORKDIR /app
```

Тут ничего хитрого нет, разве что мы удаляем за собой индекс репозиториев, который получили командой apt-get update. Считается хорошей практикой делать размер имаджа как можно меньше и удалять ненужные файлы.

Как мы помним, концепция докер-контейнера подразумевает следующее использование: один процесс = один контейнер. Тут сразу возникает сложность, ведь нам нужно запустить минимум два активных процесса: NGINX и uWSGI. Решить эту проблему можно двумя путями. Первый — написать кастомный скрипт, который мы запустим при старте контейнера. Он же запустит для нас нужные процессы и будет проверять, что они живы и здоровы. Ведь мы хотим, чтобы контейнер упал с ошибкой, если что-то идет не так.
Другой способ, чуть более замороченный, но дающий больше контроля в настройках, — использовать специальную тулу — supervisord.
Подробнее об этом читай по ссылке под видео:
https://docs.docker.com/config/containers/multi-service_container/
Если её правильно сконфигурировать, то она сама запустит за нас все нужные процессы и будет следить за их состоянием.
Давайте добавим её в устанавливаемые приложения и сразу же опишем конфиг.

```
FROM python:3.7

RUN apt-get update && apt-get install -y python3-dev supervisor nginx \
   && rm -rf /var/lib/apt/lists/*

COPY requirements.txt /app/
RUN pip install -r /app/requirements.txt

COPY src/ /app/
COPY nginx.conf /etc/nginx/nginx.conf
COPY uwsgi.ini /etc/uwsgi/uwsgi.ini
COPY supervisord.ini /etc/supervisor/conf.d/

WORKDIR /app

CMD ["/usr/bin/supervisord", "-c", "/etc/supervisor/conf.d/supervisord.ini"]
```

Внутри конфиг-файла стандартные настройки, которые будут кочевать с вами из проекта в проект:

```
[supervisord]
nodaemon=true

[program:uwsgi]
command=/usr/bin/uwsgi --ini /etc/uwsgi/uwsgi.ini
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0

[program:nginx]
command=/usr/bin/nginx
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0
# Graceful stop, see http://nginx.org/en/docs/control.html
stopsignal=QUIT
```

Тут мы описываем процессы, которые хотели бы запустить через супервизор, и говорим, что делать с аутпутом программ. В данном случае указываем, чтобы всё писалось в stdout/stderr.

Теперь перейдём к настройкам NGINX. Нам нужно сделать так, чтобы все запросы (про статику мы пока не говорим, подключить её будет вашим домашним заданием) шли на наш WSGI-сервер.

```
user  nginx;
worker_processes  auto;

error_log  /var/log/nginx/error.log notice;
pid        /var/run/nginx.pid;


events {
   worker_connections  1024;
}


http {
   include       /etc/nginx/mime.types;
   default_type  application/octet-stream;

   log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                     '$status $body_bytes_sent "$http_referer" '
                     '"$http_user_agent" "$http_x_forwarded_for"';

   access_log  /var/log/nginx/access.log  main;

   sendfile        on;
   keepalive_timeout  65;

   server {
       listen 80;
       location / {
           include uwsgi_params;
           uwsgi_pass unix:/run/uwsgi.sock;
       }
   }
}
daemon off;
```

Менять конфигурацию относительно дефолтной нам не нужно, так что нужно лишь описать конфигурацию в директиве server. Тут мы указываем, кому и куда переадресовать запрос. И последняя строка: отключаем стандартное поведение, запускаем NGINX как бэкграундный процесс при старте контейнера. Ведь мы запускаем его явно через supervisord. Вот и всё.

Теперь осталось описать конфиг uWSGI. Как уже упоминалось, у этого сервера куча разных настроек: например, встроенный балансировщик с множеством разных алгоритмов, разные опции по обслуживанию запросов (поднимать под каждый запрос процесс или переиспользовать существующие, и если поднимать новые, то как именно). В общем, их много.
Мы же пока опишем самые базовые: где будет находиться файл с WSGI-приложением, где будет находиться файл, к которому будет обращаться NGINX, чтобы передать информацию о новом реквесте.

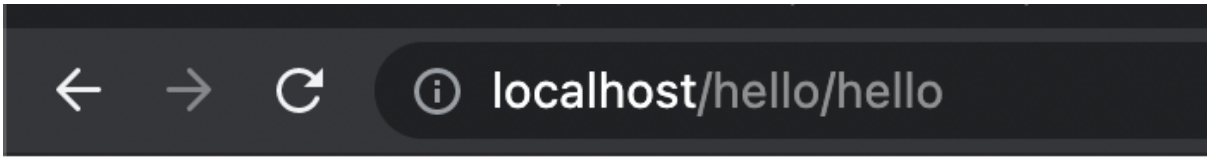
```
[uwsgi]
wsgi-file = /app/routes.py
socket = /run/uwsgi.sock
chown-socket = www-data:www-data
chmod-socket = 664
# for debugging
show-config = true
```

Кстати, заметили, что мы никак не указали, какой объект внутри этого модуля является на самом деле WSGI-приложением? На этот счёт есть соглашение, что объект, который называется app или application, будет использован как такое приложение.

Давайте соберём контейнер и запустим наше приложение.

```
docker build . -f Dockerfile -t wsgi
docker run -ti -p 80:80 wsgi
```

Идём в браузер:



{"message":"hello","name":"hello"}

Работает! На первый взгляд может показаться, что тут всё слишком усложнено, но если построить в голове цепочку вызова от клиента к Flask-приложению, то всё будет очень логично и понятно.

Итак, в этом модуле мы с вами разобрались в очень большом и важном деле — как делать из своих игрушечных приложений настоящие взрослые, которые могут обслуживать тысячи клиентов. Теперь мы знаем, какие веб-сервера бывают, знаем немного про их историю. Мы научились настраивать NGINX на отдачу статики. Знаем, что такое стандарт WSGI-приложения и какую проблему он решает. Также мы научились подключать wsg-сервер к нашему приложению. До встречи!