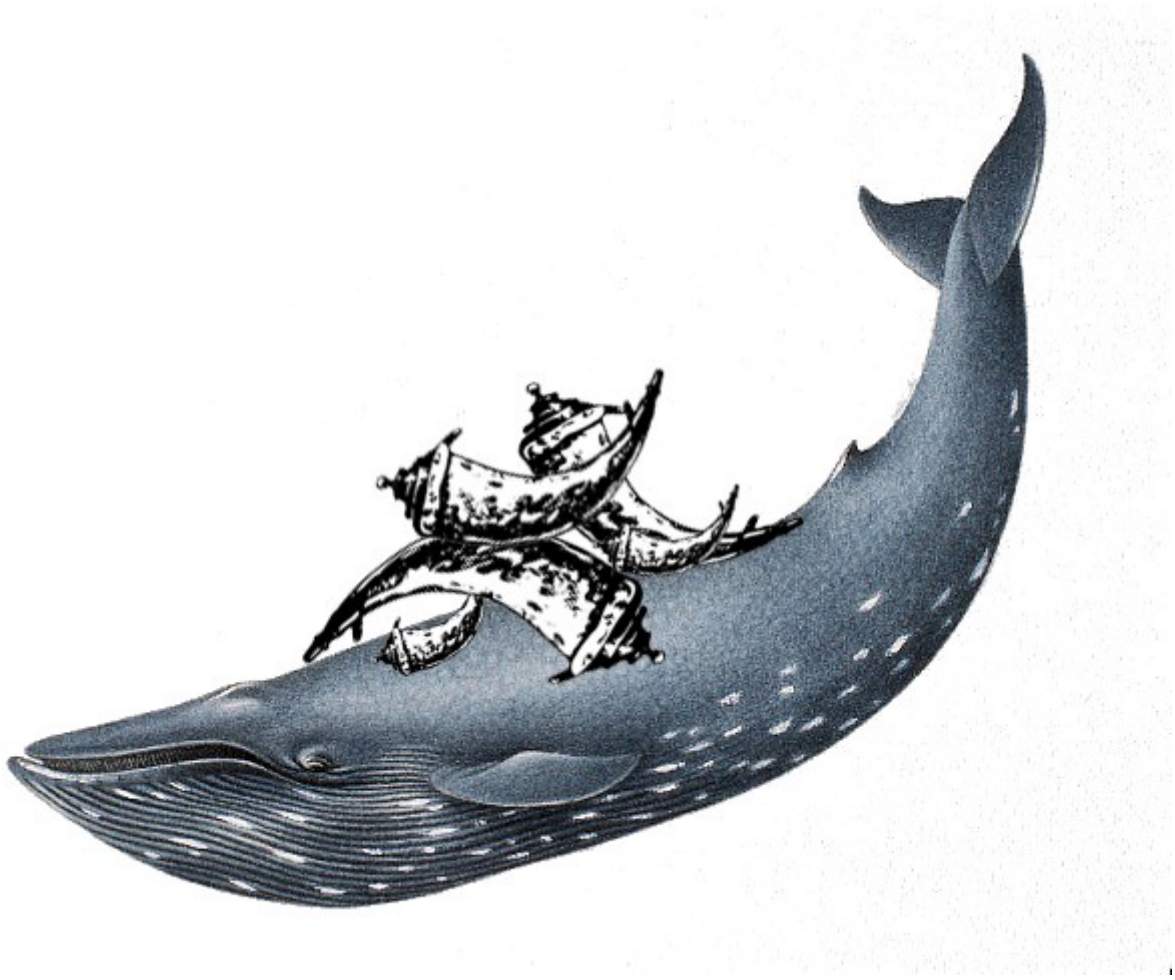
Running Flask in production with Docker

Alexey Smirnov
Aug 9, 2018 · 6 min read



Google top for running Flask with Docker is full of posts where Flask runs in debug mode. That what logs look like when Flask is in development mode:

```
* Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production
environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5555/ (Press CTRL+C to quit)
```

I'd like to make a tutorial on how to run it with uwsgi in Docker using common Docker images.

Flask app

I'll take a basic Flask app from it's official docs

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

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

Naming this file as `hello.py` and running it with `FLASK_APP=test.py flask run --port 5555` will yield a warning that development server is used in a production environment like above.

uwsgi

As suggested by the warning we should use `uwsgi` to run it. For that to happen let's create a uwsgi configuration file named `uwsgi.ini`

```
[uwsgi]
module = hello:app
uid = www-data
gid = www-data
master = true
processes = 5

socket = /tmp/uwsgi.socket
chmod-sock = 664
vacuum = true

die-on-term = true
```

So running uwsgi app with `uwsgi --ini uwsgi.ini` will give long output indicating that uwsgi started 5 processes (see `processes=5` in the config). Development server runs only 1 process thus allowing only 1 request at a time. By increasing this number you will increase the number if simultaneous requests to Flask app, but that will require more RAM and there will be more processes with Python interpreter running.

There is also a backlog for uwsgi meaning that if all workers (uwsgi processes) are buy with requests, excessive requests will be put in a queue of size 100 by default. Requests above that level will be dropped by uwsgi, but we will eventually have nginx in front of it with it's own backlog.

```
[uWSGI] getting INI configuration from uwsgi.ini
*** Starting uWSGI 2.0.17.1 (64bit) on [Fri Aug  3 22:29:18 2018]
***
compiled with version: 7.2.0 on 03 August 2018 22:23:17
os: Linux-4.13.0-46-generic #51-Ubuntu SMP Tue Jun 12 12:36:29 UTC
2018
nodename: ubuntuvm
machine: x86_64
clock source: unix
detected number of CPU cores: 4
current working directory: /home/as/Desktop/blog
detected binary path: /home/as/.virtualenvs/blog/bin/uwsgi
!!! no internal routing support, rebuild with pcre support !!!
your processes number limit is 63569
your memory page size is 4096 bytes
detected max file descriptor number: 1024
lock engine: pthread robust mutexes
thunder lock: disabled (you can enable it with --thunder-lock)
uwsgi socket 0 bound to UNIX address /tmp/uwsgi.socket fd 3
Python version: 3.6.3 (default, Oct  3 2017, 21:45:48)  [GCC 7.2.0]
*** Python threads support is disabled. You can enable it with --
enable-threads ***
Python main interpreter initialized at 0x562fb447cb40   □     i
i         i
your server socket listen backlog is limited to 100 connections
your mercy for graceful operations on workers is 60 seconds
mapped 437520 bytes (427 KB) for 5 cores
*** Operational MODE: preforking ***
WSGI app 0 (mountpoint='') ready in 1 seconds on interpreter
0x562fb447cb40 pid: 23917 (default app)
*** uWSGI is running in multiple interpreter mode ***
spawned uWSGI master process (pid: 23917)
spawned uWSGI worker 1 (pid: 23923, cores: 1)
spawned uWSGI worker 2 (pid: 23924, cores: 1)
spawned uWSGI worker 3 (pid: 23925, cores: 1)
spawned uWSGI worker 4 (pid: 23926, cores: 1)
spawned uWSGI worker 5 (pid: 23927, cores: 1)
```

Also this will create a socket file, that will be referred later in nginx configuration.

```
~ ls -la /tmp/uwsgi.socket
srwxrwxr-x 1 as as 0 aug  3 22:29 /tmp/uwsgi.socket
```

Another thing is the user which runs `uwsgi` processes and owns a socket. Ideally you need to use user with minimum access right. With `uid`/`gid` options I've specified `www-data` - standard user used by web servers. If you run uwsgi manually from bash it will use your user for socket and processes. But in Docker we will run it as root so these options are needed to downgrade it to `www-data`.

Also I've seen that some distributions clean /tmp or some daemons have different view but it's not the case with the base distribution for the image I'm using.

Uwsgi has a lot of configuration options. There is a good list you need to check if you need to troubleshoot/tune your application that can be found here

nginx

Nginx will serve as a proxy to uwsgi. It's nginx that will listen on ports 80/443 and forward requests to the socket. It's config (named `nginx.conf`) is pretty straightforward (I'm omitting SSL config)

```
user www-data;
worker_processes auto;
pid /run/nginx.pid;

events {
    worker_connections 1024;
    use epoll;
    multi_accept on;
}

http {
    access_log /dev/stdout;
    error_log /dev/stdout;

    sendfile            on;
    tcp_nopush          on;
    tcp_nodelay         on;
    keepalive_timeout   65;
    types_hash_max_size 2048;

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

    index   index.html index.htm;

    server {
        listen       80 default_server;
        listen       [::]:80 default_server;
```

```
        server_name  localhost;
        root         /var/www/html;

        location / {
            include uwsgi_params;
            uwsgi_pass unix:/tmp/uwsgi.socket;
        }
    }
}
```

I've added redirection of access and error logs to stdout so both will be accessible through `docker logs` command. Uwsgi logs are streamed to stdout by default.

One interesting thing I found is about long lasting requests. Nginx itself may kill these requests. So it's worth adding this config to `location` block:

```
uwsgi_read_timeout 1h;
uwsgi_send_timeout 1h;
proxy_send_timeout 1h;
proxy_read_timeout 1h;
```

Startup script

This is a simple startup script that will be used as default for executing container. Name of the file will be `start.sh` and it will be referred in Dockerfile.

```
#!/usr/bin/env bash
service nginx start
uwsgi --ini uwsgi.ini
```

requirements

Requirements file (`requirements.txt`) will have only flask and uwsgi and will look like this:

```
click==6.7
Flask==1.0.2
itsdangerous==0.24
Jinja2==2.10
MarkupSafe==1.0
uWSGI==2.0.17.1
Werkzeug==0.14.1
```

Dockerfile

I've tried a couple of ready-made images from Docker hub and found them overcomplicated or edited in a way by their maintainers making them unusable. So I'll start with basic python image.

```
FROM python:3.6-slim
```

First thing let's copy `hello.py`, `uwsgi.ini`, `requirements.txt` and `start.sh` to working directory:

```
COPY . /srv/flask_app
WORKDIR /srv/flask_app
```

Base image doesn't have nginx and some useful packages

```
RUN apt-get clean \
    && apt-get -y update

RUN apt-get -y install nginx \
    && apt-get -y install python3-dev \
    && apt-get -y install build-essential
```

Now let's install python requirements

```
RUN pip install -r requirements.txt --src /usr/local/src
```

Finally, copy nginx config to the proper location, add execution rights to startup script and set is as default.

```
COPY nginx.conf /etc/nginx
RUN chmod +x ./start.sh
CMD ["./start.sh"]
```

Building and running

Image build can be done like so:

```
docker build . -t flask_image
```

It creates an image named `flask_image` that can be run with this command:

```
docker run --name flask_container -p 80:80 flask_image
```

Now you may navigate to http://localhost in you browser to see the output.

Some useful options when running container

- `--name` gives the container a name that can be found in `docker ps` output
- `-p` instructs to publish port 80. Second 80 after semicolons tells what port nginx inside the container listens on
- `-d` runs container detached from terminal. Logs then can be viewed by issuing `docker logs` command

Advantages of dockerized Flask

I found it very useful to run Flask like that for a number of reasons:

1) Portability: spinning projects on a different machine with different distribution is a piece of cake, provided docker is installed.

2) No need to configure process managers (upstart/systemd)

3) Automatic restart of failed containers. Just use `--restart on-failure` with `docker run`

4) It's very easy to start with almost serverless AWS ECS Fargate.

Troubleshooting

1) When running docker container we specified port 80 so you host should have this port available for docker to bind to on you host. If it's not then change it to something else.

2) If your host runs multiple containers they should listen on different ports and some kind if proxy should be running on host OS to direct requests to the proper container.

3) To get inside the container issue `docker exec -it flask_container /bin/bash`. To get the exact container name issue `docker ps`

4) To follow error and access logs issue `docker logs -f flask_container`

Original post from my blog https://smirnov-am.github.io/2018/08/09/flask-docker.html

How do you run Flask in production? Drop me a message on LinkedIn https://www.linkedin.com/in/smirnovam/

Docker Python Flask