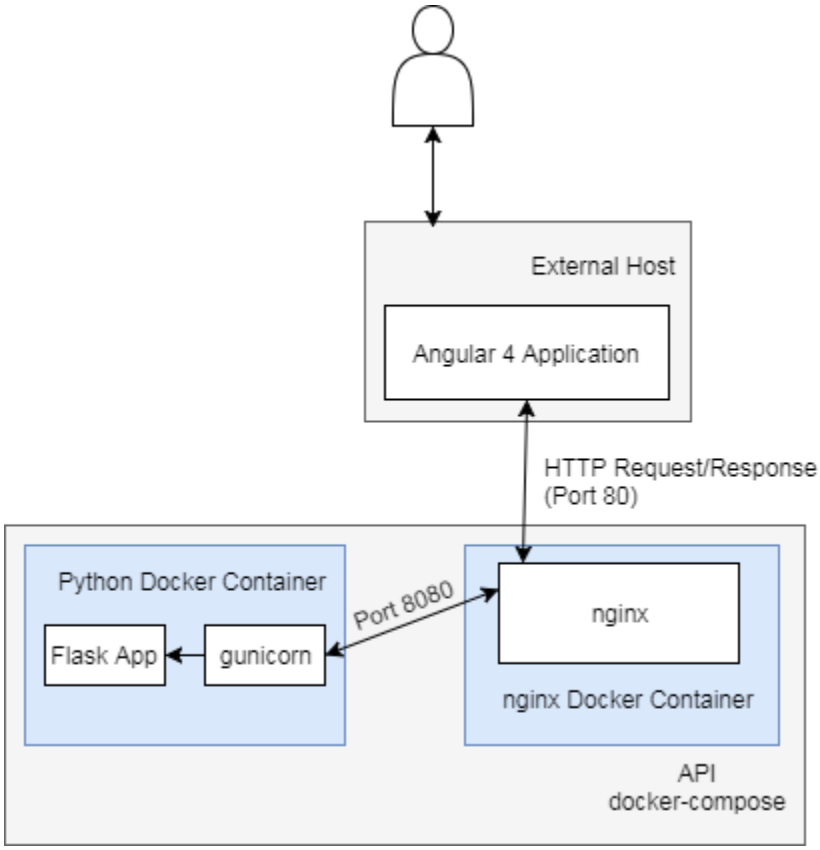
API with Python Flask-RESTful and Docker with nginx

Oct 10, 2017

I've enjoyed using Python to create simple APIs, such as the one for the SMS Marketing Tool I previously posted on. I am writing this post to share the way I have been developing these APIs with Flask-RESTful using Docker containers and nginx. Using Docker enables you to easily deploy your application on any Linux VPS (in this case Ubuntu 16.04 from OVH) and have repeatable results, because the dependencies are held within the container. Having nginx route traffic also enables hosting many services/APIs on the same VPS, all of which are hosted in their own Docker containers with local ports exposed. See below for a single app example that also includes a front end Angular app:



In my previous posts I have primarily focused on Angular application

development using an external cheap host like Namecheap. In the future, I will instead host my Angular applications directly on the VPS in their own Docker containers, with traffic routed between the containers by nginx. For this post, I will focus only on the Python application being served.

Overview

We will create a very simple “API” that has only one endpoint /airing/ that will respond with a JSON array (not an object) of currently airing shows including their dates and alternative string format of the show name. See below for an example screenshot of some of the JSON in Postman:


```
[
    [
        "2016-08-24",
        "Another Period S02E11 Lillian is Dead",
        "Another+Period+s02e11"
    ],
    [
        "2016-12-04",
        "Westworld S01E10 The Bicameral Mind",
        "Westworld+s01e10"
    ],
    [
        "2017-09-27",
        "South Park S21E03 Holiday Special",
        "South+Park+s21e03"
    ],
    [
```


The steps of this post are as follows:

1. SSH into VPS, set up Python environment
2. Python TV API Parsing
3. Python Flask-RESTful API
4. Docker with Python
5. nginx
6. docker-compose
7. API testing with Postman

Full Code

If you’d rather just get going, you can find the full source code here on GitHub. I am developing an app related to this and using Gitlab to host the private repositories but figured it would be nice to copy this code out

to a public repository as a sort of snapshot of setting up a super simple API, mostly on the sysadmin side

API, mostly on the sysadmin side.

SSH into VPS

First step, SSH into your VPS using whatever method you want. I use Windows so I use PuTTY to SSH into my Ubuntu 16.04 VPS. Once you are there, make sure you have the program screen, which is a terminal multiplexer. I use this all the time to be able to restore sessions when they close and also have multiple terminals open. Choose a directory to put your code, in my case I made a folder called *tv-api* in my home directory. Create another folder *app* and cd into that for the Python portion.

Python Environment

Because I use Ubuntu 16.04, Python is installed by default. I like to do a bit of the Python environment set up outside of the Docker container and then place it into the container when I want to actually start testing GET/POST requests to the API. If you are using something like Debian you may need to install Python, or you can skip ahead to the Docker sections below to learn how to run your code directly in the Docker container.

First I create a virtual environment and start developing in Python as I normally would. To do this, simply type *virtualenv env* and the folder env will be created. To activate your environment type *source ./env/bin/activate.*

Go ahead and install flask, flask-restful, gunicorn, and requests using *pip install* ***application***

Python TV API Parsing

The meat of the python code is a file called *show_schedule.py* which is a set of functions that enables searching the TvMaze API for specific shows and figure out when they are airing. It organizes the episodes it finds that are airing of the specific shows searched and orders them based on air date.

```
import requests
import json
import operator
def appendZero(str):
```

```
        if int(str) < 10:
                str = "0"+str
        return str
def episode_details(episode,show):
        r = requests.get(episode)
        ep = json.loads(r.text)
        season = appendZero(str(ep['season']))
        episode = appendZero(str(ep['number']))
        name = show+" S"+season+"E"+episode+" "+ep['name']
        href = "+".join(show.split(" "))+"+s"+season+"e"+ep
        details = [ep['airdate'],name,href]
        return details
def show_details(show):
        urlshow = "%20".join(show.split(" "))

        base = 'http://api.tvmaze.com/singlesearch/shows?q=
        r = requests.get(base+urlshow)
        details = json.loads(r.text)
        #print(details['name'])
        #print(details['status'])
        next = []
        if(details['status'] == 'Running'):
                #print(details['schedule']['days'])
                prev_ep = details['_links']['previousepisod
                prev_details = episode_details(prev_ep,deta
                next.append(prev_details)

                try:
                        next_ep = details['_links']['nextep
                        next_details = episode_details(next
                        next.append(next_details)

                except:
                        pass
                return next
def airing_shows():
        list = ["curb enthusiasm","modern family","nathan f
        episodes = []
        for item in list:
                details = show_details(item)
                try:
```

```
                        episodes.extend(details)
                except:
                        pass
        episodes.sort(key=operator.itemgetter(0))
        return episodes
```

Not too bad. This could definitely be way faster by using less requests, but it gets the job done and the point of this post is to understand how to create an API with Python Flask and Docker with nginx. One way to make this better is after searching the TvMaze API the script could store the important data in an SQLite database, making retrieval extremely quick.

Python Flask-RESTful API

For this part I am going to only implement one GET method at the /airing/ URL. It is extremely easy to add DELETE, PUT and POST if you follow the Flask-RESTful documentation. Let's see the code below, which I have put in *app.py*:

```
from flask import Flask, request, jsonify
from flask_restful import reqparse, Resource, Api
from show_schedule import airing_shows
app = Flask(__name__)
api = Api(app)

def options(self):
    pass

class Airing(Resource):
        def get(self):
                episodes = airing_shows()
                return jsonify(episodes)

api.add_resource(Airing,'/airing/')

if __name__ == '__main__':
     app.run(host='0.0.0.0', port=8080)
```

Extremely simple! You can add a lot of other elements in here, such as post and get variable parsing, but in this case a GET request to /airing/

post and get variable parsing, but in this case a GET request to /airing/ will call the airing_shows() function from the parser and output the JSON array of shows.

WSGI with gunicorn

For a production API you cannot actually use the Flask development server. Instead you need to use another server that serves the Flask application. In my case I chose to use WSGI with gunicorn because it was extremely easy. Let's see our file *wsgi.py*:

```
from app import app

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

Done!

Docker with Python

Now it's time to have some fun with Docker. The way we use Docker is we start with a base image and then build upon it to make our own base image. I like to create an "environment" image that has all of the Python requirements ready to go within the container already. From there I can actually build a production Docker image that copies the Python code to the previously created environment image and runs it. You don't need an environment image, but if you don't have one then every time you rebuild your Docker image the Python requirements will have to be reinstalled which takes some time and can be a bit annoying.

DockerfileENV

```
FROM python:3
WORKDIR /usr/src/app
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
```

What this file does is it describes to Docker that it should start with the python:3 image and then copy requirements.txt to /usr/src/app and install all of the requirements. I like to also create a bash script that builds this image and names it appropriately. Put this in env.sh:

```
docker build --no-cache -t tv_env -f DockerfileENV .
```

As you can see, in this situation we specify the name as tv_env and the Dockerfile as DockerfileENV. Now we can proceed to actually copying our Python code to the production Docker image.

Dockerfile

```
FROM tv_env
WORKDIR /usr/src/app
COPY . .
EXPOSE 8080
CMD [ "gunicorn", "--bind", "0.0.0.0:8080","wsgi:app" ]
```

You can see this file uses the tv_env as the base image and copies the Python code to the app directory. Then it exposes the port the Flask app is on and calls gunicorn to serve the application.

To get your Python script running on the local network, run *./env.sh* followed by *docker build -t tv .* and then *docker run -t tv*

nginx

nginx is the reverse proxy that will direct public traffic on port 80 to our Dockererized app's internal port 8080. cd back into the main directory *tv-api* and create the folder *nginx*. In that folder create the file *nginx.conf*:

```
events { worker_connections 1024; }

http {
    sendfile on;

    upstream app_servers {
        server app:8080;
    }

    server {
        listen 80;
                add_header Access-Control-Allow-Origin * al
                add_header Access-Control-Allow-Methods GET
                add_header Access-Control-Allow-Headers X-R
        location / {
            proxy_pass         http://app_servers;
            proxy_redirect     off;
            proxy_set_header   Host $host;
```

```
            proxy_set_header   X-Real-IP $remote_addr;
            proxy_set_header   X-Forwarded-For $proxy_add_x
            proxy_set_header   X-Forwarded-Host $server_nam
        }
    }
}
```

This file may be more liberal than some would write it, but basically it allows requests from any origin of methods GET, POST,PUT,DELETE. This is important in part for CORS – because by default your browser will not access a server unless it is told in the headers that it is explicitly allowed to. That is not true for Postman, it will access any server you tell it to as CORS is not implemented within it.

Now let's also create a Dockerfile like we did for the Python app:

```
FROM nginx
COPY nginx.conf /etc/nginx/nginx.conf
```

No need to build or run the nginx container yet, that will be handled by docker-compose.

Running the app with docker-compose

Now comes the fun part, let's make a network of containers that can communicate. The app and nginx will communicate via port 8080 so that traffic sent to nginx on port 80 will be redirected to port 8080 on the Python app which will then be served via gunicorn and Flask. Imagine you could have another API on port 8081 that nginx directs from either a different domain name or just based on the url. Similarly you could have multiple services directly within the Python Flask app.

In order to run the network of containers together, we need a *docker-compose.yml* file:

```
version: '2'

services:
  app:
    restart: always
    build:
      context:  ./app
      dockerfile: Dockerfile
```

```
      dockerfile: Dockerfile
    expose:
      - "8080"
  proxy:
    restart: always
    build:
      context:  ./nginx
      dockerfile: Dockerfile
    ports:
      - "80:80"
    links:
      - app
```

This file creates two services, app and proxy and directs Docker to create the images from the app directory Dockerfile and nginx directory Dockerfile respectively. You can see the proxy has port 80 from the host machine forwarded to the Docker proxy and the app has port 8080 exposed for nginx.

Go ahead and create a *build.sh* file:

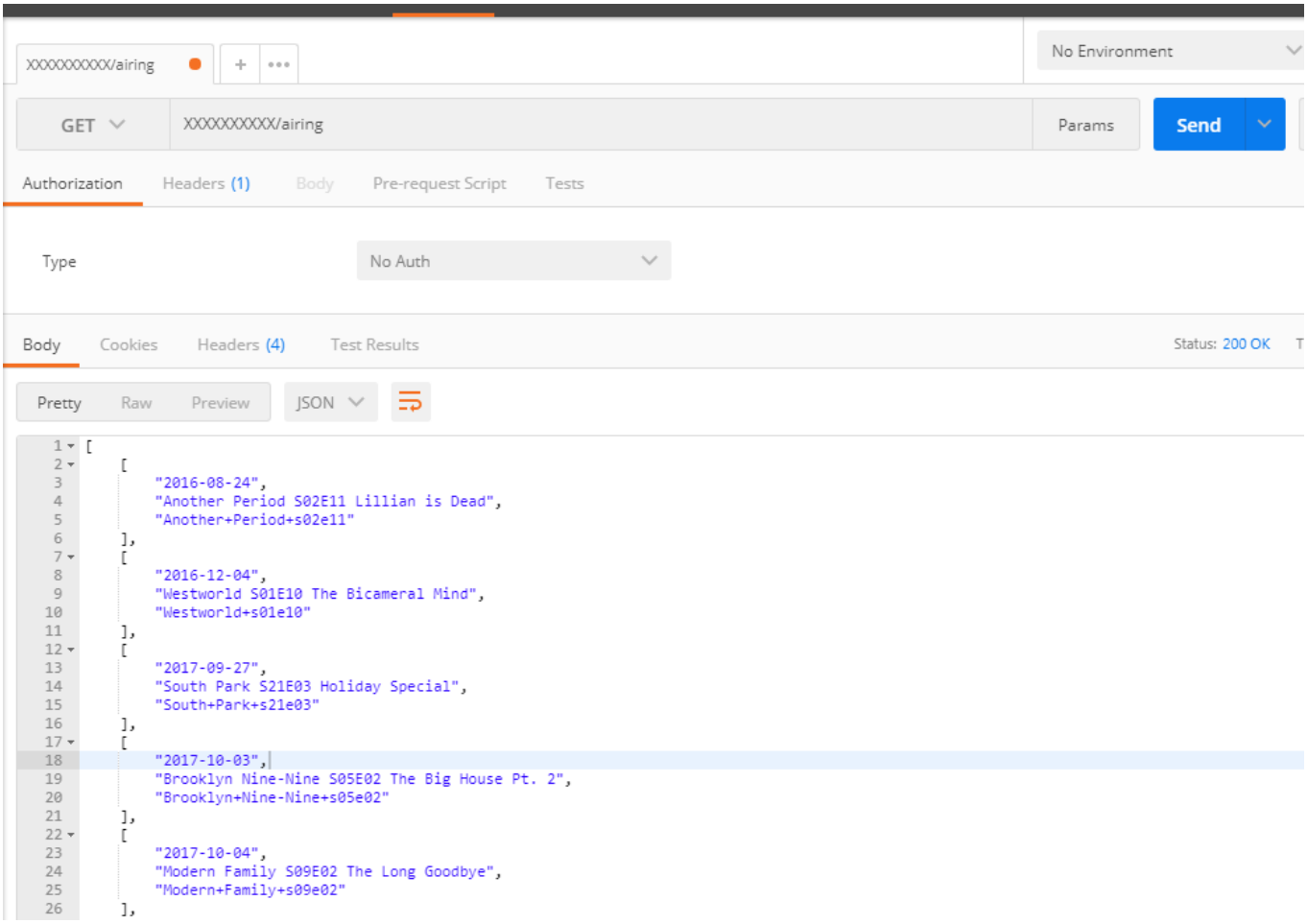
```
docker build --no-cache -t tv_api ./app
docker-compose build
docker-compose up
```

This file rebuilds the tv api without cache to ensure the files are actually updated and then uses docker-compose to construct the network of containers and run it. Note that the first line ensures that during development your code is updated, I had issues where docker-compose was using the cache. If you have a docker volume running you may also need to use docker-compose down to ensure everything is closed before you rebuild the application.

Any time you need to run the API without rebuilding, you can simply use *docker-compose up*.

Testing with Postman

I strongly recommend using Postman to test your API prior to consuming it with Angular. Save yourself the headache! See below for an example of me using Postman, although I Xed out the IP address of my VPS.


XXXXXXXXXX/airing
No Environment
GET
XXXXXXXXXX/airing
Params
Send
Authorization
Headers (1)
Body
Pre-request Script
Tests
Type
No Auth
Body
Cookies
Headers (4)
Test Results
Status: 200 OK
Pretty
Raw
Preview
JSON
[
[
"2016-08-24",
"Another Period S02E11 Lillian is Dead",
"Another+Period+s02e11"
],
[
"2016-12-04",
"Westworld S01E10 The Bicameral Mind",
"Westworld+s01e10"
],
[
"2017-09-27",
"South Park S21E03 Holiday Special",
"South+Park+s21e03"
],
[
"2017-10-03",
"Brooklyn Nine-Nine S05E02 The Big House Pt. 2",
"Brooklyn+Nine-Nine+s05e02"
],
[
"2017-10-04",
"Modern Family S09E02 The Long Goodbye",
"Modern+Family+s09e02"
],