3 Aug - Nginx Load Balancer Challenge

Task:

Nginx Load Balancer Challenge:
Using docker compose create a project containing 2 docker containers. The first containing a python API (The can be very basic) and the second will be Nginx. The challenge is to make your api available on your localhost by using Nginx as a reverse proxy.

Tutorial(s) found to work through

We decided to go this tutorial:
https://levelup.gitconnected.com/using-nginx-reverse-proxy-with-flask-and-docker-66854e940176

To look at
This looks more like a completed template you just add your apps in:
https://github.com/argiris-mat/docker-compose-nginx-reverse-proxy

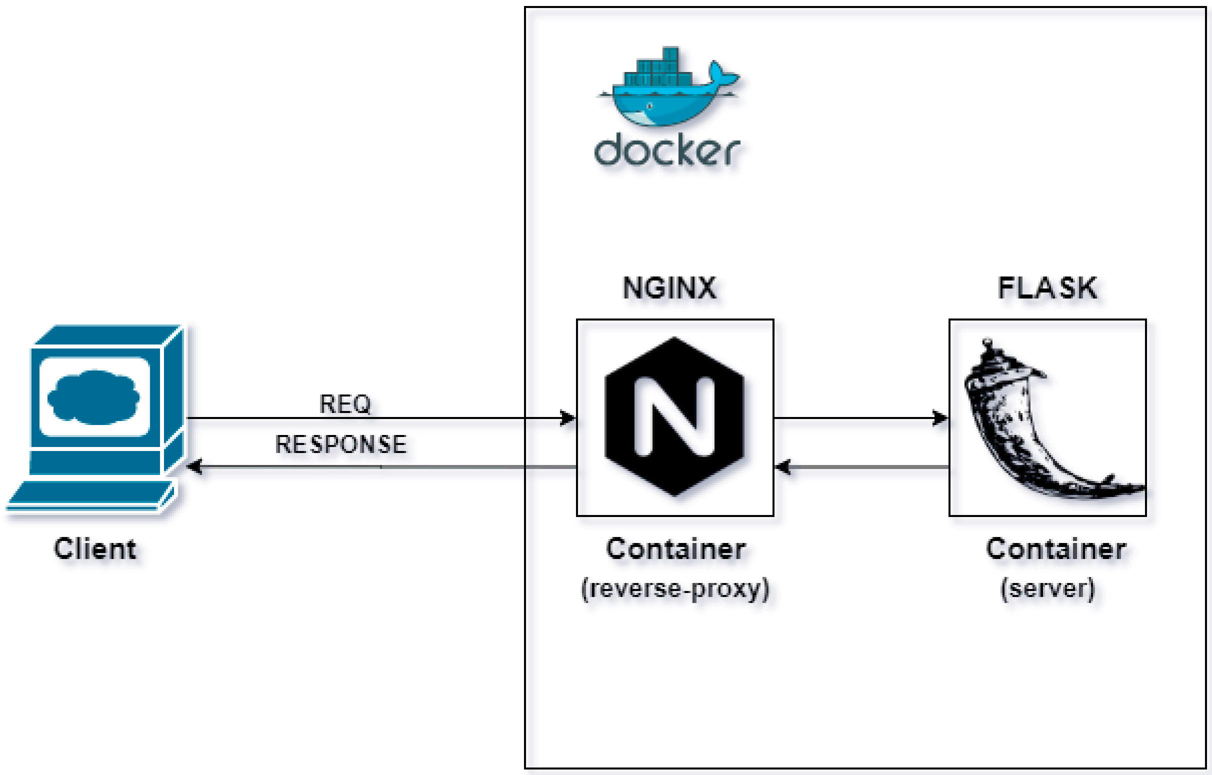
Expalnations of terms

- NGINX
  free, opensourse, high prformant HTTP server. Also can be used as a reverse proxy or IMAP/POP3 proxy server
  Known for
    - its stability
    - rich feature set
    - simple config
    - low resourse consumptio

the softwares structur is asynchoronous and event-drive, which allows the processing of many requests at the same time.
NGINX and Apache are the two best webs servers to host a web application

- Reverse Proxy
  In a computer network, sits between a group of servers and the clients who uses them.
  Directs all the requests from the clients to the servers and it also delivers all the responses form the servers to the clients
  From the clients point of view, it looks like everything is coming form one place.
  Nginx web server can provide several advace features sucj as

    - load balancing
    - TLS/SSL capabilities
    - acceleration that most specialised apps lack



Setup for NGINX reverse proxy and Flask application

Working through an initial tutorial

So this is baised on the first tutorial founf
https://levelup.gitconnected.com/using-nginx-reverse-proxy-with-flask-and-docker-66854e940176
But there was some issues with it.

Step 1: Pull the latest NGINX docker image

We need to install the latest docker version on our local system (Linux preferable). We will pull the official NGINX image from the Docker Hub.

```
$ sudo docker pull nginx:latest
```

Step 2: Run the NGINX container

After pulling the official NGINX docker image, we can now run it as a container with port 80 exposed to the web client.

```
$ sudo docker run --name nginx_reverseproxy -d -p 80:80 nginx:latest
```

Check the status of the container. We can check if the container is in running or in a terminated state.

```
$ sudo docker ps -a
```

> We have port mapped and exposed the port 80 of our NGINX container to our local machine port 80. Hence we should now be able to access NGINX on our machine's localhost. (127.0.0.1:80 or 0.0.0.0:80 or localhost:80)

Verify by doing curl operation on the localhost address. We should be able to see the HTML page.

```
$ curl 127.0.0.1
<!DOCTYPE
html>https://wiki.postgresql.org/wiki/Pythonhttps://hub.docker.com/_/postgreshttps://
opensource.com/article/18/4/flask
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Step 3: Pull the Python container

the NGINX container up and running. So we now have to pull the python3 image from Docker Hub. - we will use this to crear We will be hosting a sample application using the Python Flask framework. Flask is a web framework. Flask provides tools, libraries, and technologies that allow you to build a web application (client-server model).

```
$ sudo docker pull python:3
```

Step 4: Configuring the Python container

We have pulled the raw python3 container. We need to install a flask and the web application code and create a new docker image using the recently pulled python:3 container as the base. Thus, we have to write 'Dockerfile' accordingly.
Dockerfile :

```
FROM python:3
RUN pip3 install flask
```

Use the docker build command where the 'Dockerfile' is and give an appropriate name and tag.

```
$ sudo docker build -t flask:v1 .
```

> NB: 'flask:v1' is our new image based on the 'python:3' image we downloaded

Step 5: Write a sample web-app Flask code

We will write a simple code for the web-app to be hosted. We name it as /flask_code/main.py

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello():
    return 'Hello NGINX reverse proxy'
if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

> Note : The code is placed in a directory called 'flask_code' . The directory and the Dockerfile share the same location . The directory name can be of your choice.

> SO this gets confusing down the line, because when we run the nginx.config file, it copies every thing over to a '/flask_code/' dir and we end up with

> '/flask_code/flask_code' on the container. so I put this in root not '/flask_code'.

Step 6: Write docker-compose.yaml file

Till now we are ready with NGINX container and the Flask code along with the ‘Dockerfile’ to create the image. However, for the communication to take place between the 2 containers, we need to write the **docker-compose.yaml** file.

```
ersion: '3.1'
services:
    nginx:
        image: nginx:latest
        container_name: nginx_reverseproxy
        ports:
            - 80:80
    flask:
        build:
            context: ./
            dockerfile: Dockerfile
        image: flask:v1
        container_name: flask_webapp
        volumes:
            - ./:/flask_code/
        #this isnt technically needed.
        environment:
            - FLASK_APP=./main.py
        command : python /flask_code/main.py
        ports:
            - 8000:5000
```

Now, stop the running NGINX container and delete the container. This is because the docker-compose.yaml file will set all these up.

```
$ sudo docker stop nginx_reverseproxy
$ sudo docker rm nginx_reverse_proxy
```

Step 7: Testing the entire setup

We need to set the entire setup now by using docker-compose.yaml file. This file will set both the containers running.

```
$ sudo docker-compose up -d
Creating flask_webapp       ... done
Creating nginx_reverseproxy ... done
```

> Note : To run the above command, docker-compose should be installed in the local. If not it can be easily done using the steps provided on their documentation.

Check if both the containers are up and running.

```
$ sudo docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                    NAMES
ce70c0f516ea        flask:v1            "python /flask_code/…"   3 seconds ago
Up 2 seconds        0.0.0.0:8000->5000/tcp   flask_webapp
28ce6174ff58        nginx:latest        "nginx -g 'daemon of…"   About an hour ago
Up About an hour    0.0.0.0:80->80/tcp       nginx_reverseproxy
```

Step 8: Configuring the reverse proxy

Now, that we have both the containers running (1 for Nginx, 1 for Flask), we need to configure the NGINX container to act as a reverse proxy server.
For that, we need to do a process called a proxy pass. Proxy pass directive sets the address of the proxied server and the URI to which the location will be mapped. So in our case, we will proxy pass the traffic coming on the '/' route of our flask application.

Write the **nginx.conf** file.

```
server {
    listen 80;
    server_name localhost;
location / {
        proxy_pass http://flask_webapp:5000/;
        proxy_set_header Host "localhost";
    }
}
```

> In the above file, we have set the port number, the server name, the route on which traffic will be received, the proxy pass URI, where the incoming traffic is to be routed. We will be adding this file in our docker-compose.yaml

Step 9: Docker Inter Networking

We need to add both our running containers in a logical network group. Let’s name it as “docker-network”. We set the flask application alias as “flask_webapp”. So, the final docker-compose.yaml with the above changes would look like :

```
version: '3.1'
services:
    nginx:
        image: nginx:latest
        container_name: nginx_reverseproxy
        depends_on:
            - flask
        volumes:
            - ./nginx.conf:/etc/nginx/conf.d/default.conf
        networks:
            - docker-network
        ports:
            - 80:80
    flask:
        build:
            context: ./
            dockerfile: Dockerfile
        image: flask:v1
        container_name: flask_webapp
        volumes:
            - ./:/flask_code/
        environment:
            - FLASK_APP=/flask_code/main.py
        command: python /flask_code/main.py
        networks:
            docker-network:
                aliases:
                    - flask_webapp
        ports:
            - 8080:5000
networks:
    docker-network:
```

So now, by doing this, what has happened is, the flask container has become a dependency for NGINX container as it will redirect all the traffic on the flask container, thus, the ‘nginx’ service will by default start the flask app.

Run the docker-compose command to redeploy the containers with changes with specifying the service name to be created first i.e.nginx.

```
$ sudo docker-compose up -d
```

Step 10: Testing and Verification

Curl or use the browser to hit localhost:80. What will happen is, the traffic will be directed to NGINX reverse proxy, which will redirect it to the 'flask_webapp', get the response from the flask and then give it back to the client. Check the port mappings of both the containers.

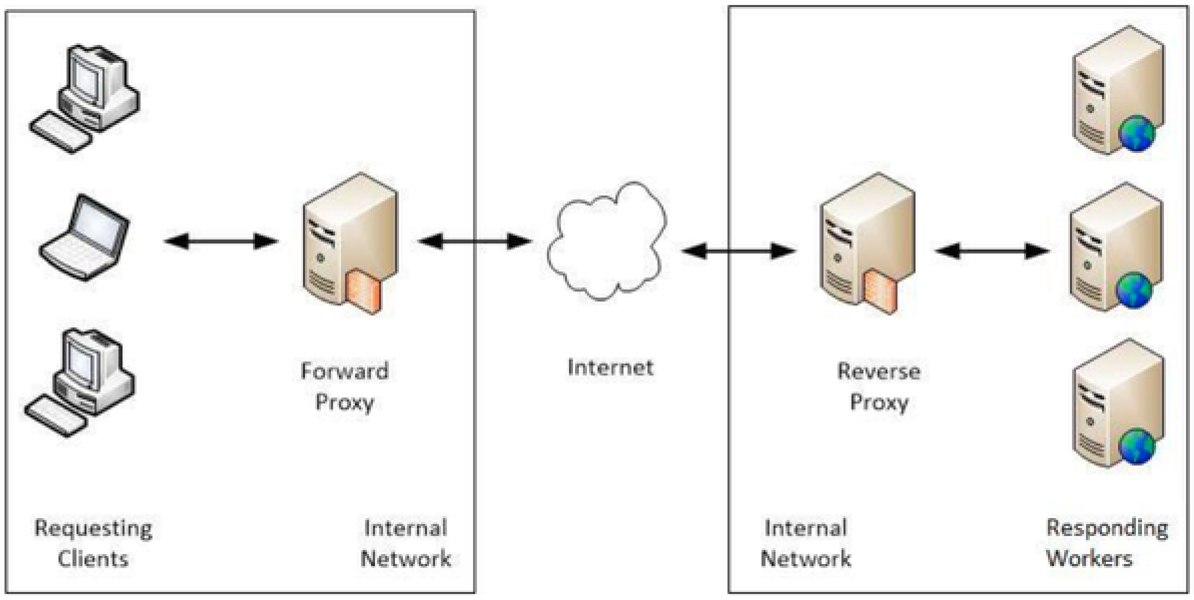
```
$ sudo docker port <CONTAINER_ID>
80/tcp -> 0.0.0.0:80     (for nginx_reverseproxy)
5000/tcp -> 0.0.0.0:8080 (for flask_webapp)
```

$ curl localhost
Hello NGINX reverse proxy

Or go to web browser and enter http://localhost

Conclusion:

We have successfully configured the NGINX server as a reverse proxy server, in a form of lightweight containers using Docker. In any system architecture, a reverse proxy server will always play an important role in security. It resides just behind the firewall, unlike a forward proxy or just "proxy" that resides in front of the firewall.



Forward vs Reverse proxy

The reverse proxy provides an additional level of abstraction and control to ensure the smooth flow of network traffic between clients and servers. It can also be used to protect critical applications by inspecting the incoming traffic for malicious requests. On finding malicious content in the request, the reverse proxy can be configured to drop the request.

My final solution: ~Documents/Training/set-projects/nginx-load-balancer-challenge/archive-1

Project extension 1 - Add a second API/Flask container to /app2

We set up:

- a second .py file with a new api in (app2.py)
- new flask second in docker-compose.yaml - copy of "flask" with the apppriate bits changed.
- new location context in nginx.conf - copy of the current location with the appropriate changes

http://localhost goes to the exsiting API
http://localhost/app2 goes to the exsiting API

NB:

- in docker-compose.yaml, dockers default port is 5000.

```
ports:
    - 8080:5000
```

My final solution: ~Documents/Training/set-projects/nginx-load-balancer-challenge/archive-2

Project extension 2 - Create multiple containrs/images connected to the same API, working as a load balancer

We set up:

- Additional flask containers in docker-compose.yaml - duplications of the new flask created in ext 1, above, they all point the to the same api (ap2.py)
- in nginx.conf
    - amended the second location context to use an 'upstream' context for the proxy_pass.
    - added the 'upstream notes'
- I also renamed alot of the vriables, this is to make sure i am totally clear which bit is being refered to other bits.
- Also removed code that is not strictly required.

My final solution: ~Documents/Training/set-projects/nginx-load-balancer-challenge/archive-3

At this point in the dev, I uploaded what I have to GitHub

Should have done this before to - just didnt think to do so.

Notes/Things learnt:

- Remove the '-d' and you can see the logs in 'realtime'
- You can use the following command to stop and remove the services

```
$ sudo docker-compose down
```

- To view the doc- I already has somethign on port 80. usted Netstat to find out it was apachie so used this and stopped apache (Apache2 auto starts on computer reboot)

> ```
> $ sudo netstat -tunlp
> ```

From here: https://linuxize.com/post/check-listening-ports-linux/
r

It lists all the ports

-t - Show TCP ports.
-u - Show UDP ports.
-n - Show numerical addresses instead of resolving hosts.
-l - Show only listening ports.
-p - Show the PID and name of the listener’s process. This information is shown only if you run the command as root or sudo user.

```
$ sudo service apache2 stop
```

- MISSING '/'
  In the nginx.conf file, in the location context, we where missing a '/' int the line: "proxy_pass http://notes/;" (the '/' after notes)

```
location /app2 {
        proxy_pass http://notes/;
        proxy_set_header Host "localhost";
    }
```