Flask in Production with Nginx - Microservices

A simulation of production environment for the Flask app

We'll cover the following

Getting exposure to production systems before starting on any project will definitely give you a positive feeling and confidence. This lesson is all about how the app looks like in production.

> There are n number of ways an app can be deployed in production, but the idea behind is the same: using an architecture which can survive the load.

> Type `git log` and checkout the commit 327b245bbd4392d3dee40a24457c3ed44a1ef5b9 (Flask in Production with Nginx - Microservices) for code related to this lesson.

Production code explanation

You will see a lot of changes in this version of the code. Let's list them one by one.

Docker level

1. New Nginx service
2. Read-only mounting of the file system
3. Resources limitation

App level

App level

1. New wsgi server `gunicorn` instead of flask server.

> The **Gunicorn** "Green Unicorn" is a Python web server gateway interface HTTP server. It is a pre-fork worker model ported from Ruby's Unicorn project. The Gunicorn server is broadly compatible with a number of web frameworks, simply implemented, light on server resources and fairly fast

All these changes will make our app stable and robust. Adding extra replicas of services leads to a better distribution of load received by the app. Let's see the detailed explanation of every change we have done in this version.

Nginx service

Flask comes with a prebuilt development server. As it is a development server, we cannot deploy the app to production with a server which cannot handle production load. So, we need something that is tested for production loads and can manage multiple requests efficiently for our app. These are called ***proxy servers.***

Two famous names for proxy servers are `Nginx` and `Apache`. Most of the time PHP coders use Apache and Python coders use Nginx. This is just a trend and not a strict guideline. We can use any of these two. Nginx is easy for configurations and is widely used for Python apps. So, we will use Nginx.

> Proxy servers handle requests for static content by themselves and pass on the request which can only be served by language-specific servers to upstream servers, like Gunicorn. This makes serving requests a little faster. We have used Gunicorn here as an upstream server.

In the Nginx section in a docker-compose file, HTTP and HTTPS ports are mapped to hosts HTTP and HTTPS ports. The reason is very simple.

Whenever you open up any website using `https`, you will see a lock icon. This means the website is served over a secure HTTP protocol, and certificates of security are shared and accepted by client and server. To simulate that in development, we need to make a localhost server on HTTPS, which is impossible without certificates.

So, to make HTTPS work over HTTPS, we created self-signed certificates. Self-

signed certificates are accepted by only the local server and not by anybody else.

Read more about self-signed certificates here.

These certificates are mounted to the Nginx certificate directory and used in SSL configuration of Nginx.

- `volumes` is a section of Nginx service that mounts the Nginx configuration and certificates to the containers configuration directory.
- `ports` maps host's HTTP and HTTPS port to containers HTTP and HTTPS ports.
- `deploy` section deploys two replicas to distribute the load and limit the host CPU usage by the container.

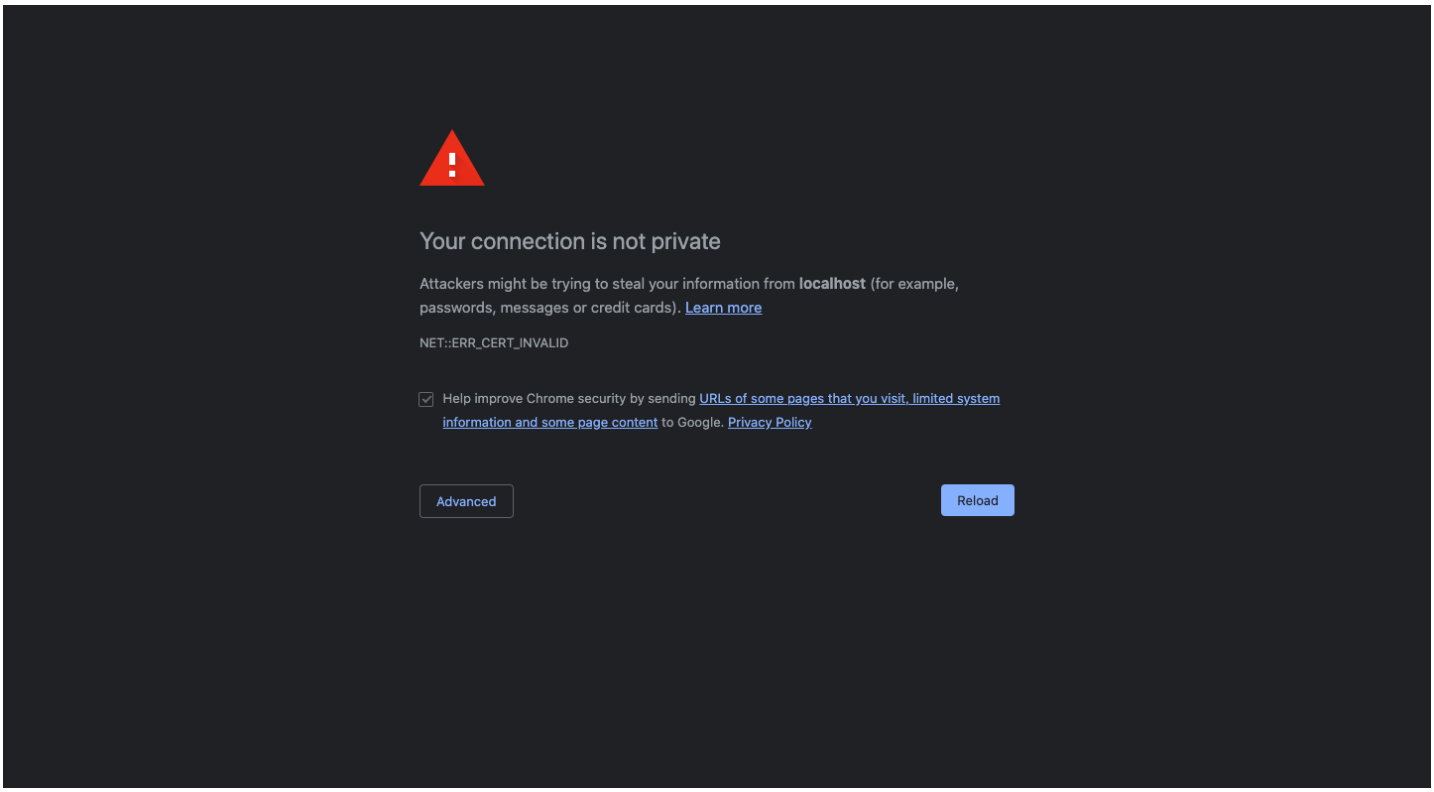
People use very complex Nginx configuration to secure a server from different attacks and there is a tool for generating these config files for a production system. Certbot is one of them.

But here we are using a simple one, which forwards all HTTP requests to the https server, and https server will forward requests to the upstream pythonic server.

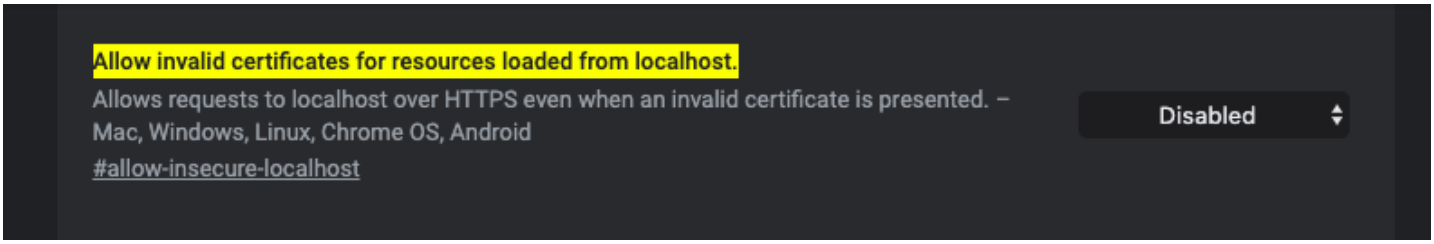
This is enough about Nginx for now. Go and run the pulled code using `docker stack deploy login_app --compose-file=docker-compose.yml`.

> You won't be able to see any output on Eductive's single page app below because port 80 and 443 are not allowed for the course use.

```
<h1>Sign In</h1>
<form action="/" method="post" novalidate>
    <input type="hidden">
    <p>
        <label>Username</label> <br>
        <input type="text" id="username" name="username">
    </p>
    <p>
        <label>Password</label>
        <br>
        <input type="password" id="password" name="password">
    </p>
    <p> <input type="submit" value="Submit"> </p>
</form>
```



If you see something like this, open a new tab and enter `chrome://flags/#allow-insecure-localhost`.



Enable it and refresh the localhost. That's it. Now you are serving your content over an HTTPS site. Because of the self-signed certificates, the browser will show a caution sign instead of a lock sign.

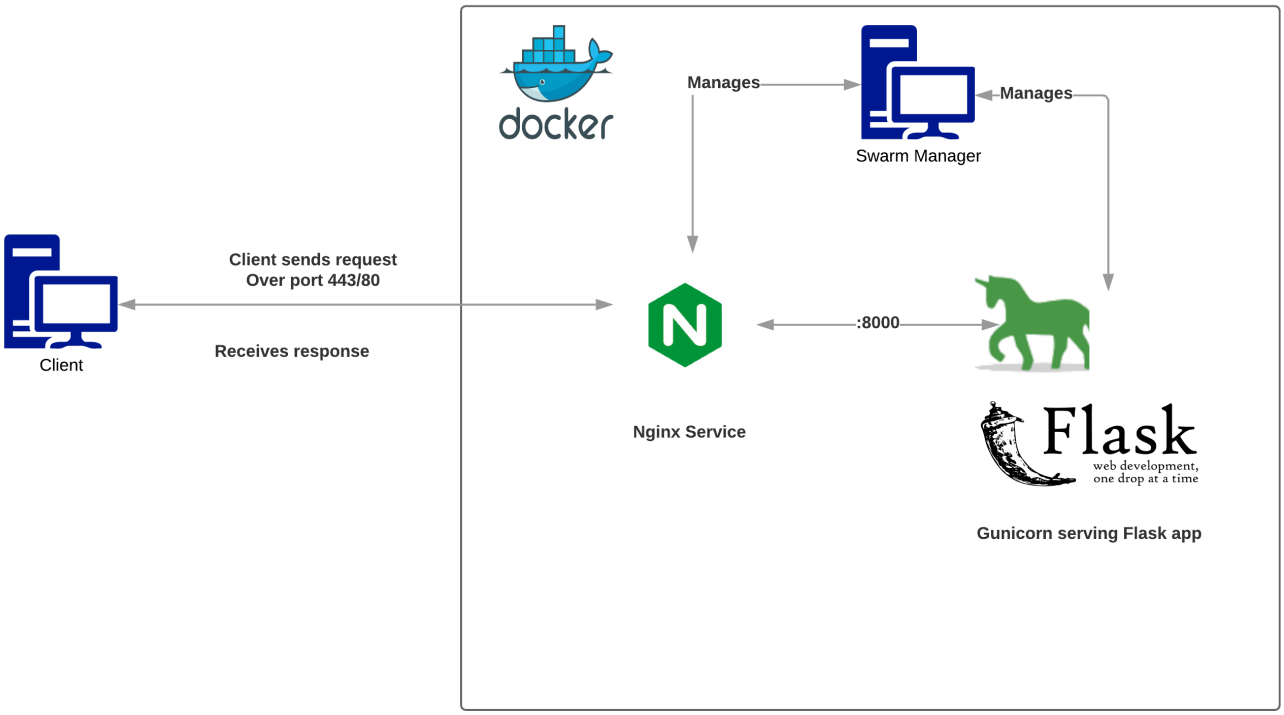
App changes

Previously, we used a Flask server everywhere. The Flask server can barely handle a thousand requests. After this number, it will start slowing down and sometimes, gets timed out.

> There are different servers created to totally focus on processing and serving Python code to the clients. Those are called WSGI servers. Gunicorn is one of them and most of the time, it is used in production for Flask applications.

That's the reason we removed the Flask server and added Gunicorn. Below is the

That's the reason we removed the Flask server and added Gunicorn. Below is the architecture of the current implementation of Nginx and Gunicorn.



Nginx is the front-facing worrier for us, hence, we have created two replicas of it and the swarm manager makes sure that at any point in time, there should be two instances of Nginx running.

Microservices

If you look at the file structure, you will notice that only the docker-compose.yml and README file are outside and everything has its own folder and Dockerfile.

This will help us to scale our app seamlessly as every component of our app is loosely coupled and can scale up and down based on the number of replicas provided. This architecture is most supported for the Kubernetes engine which is used by Google to manage their millions of containers.

> In a microservice, the service definition is based on the requirements and architecture. You can create a whole separate service just for a particular function.

This was a very long lesson, but we have just seen the overview of production systems. There is way more than this to explore, but taking one step at a time and gaining experience incrementally will grow your skills significantly.

Reach out to me in forums or via email about any concerns and do try different Docker tools which we have mentioned in the bonus section. I wish you all the very best for your career and for all the projects you will be working on!