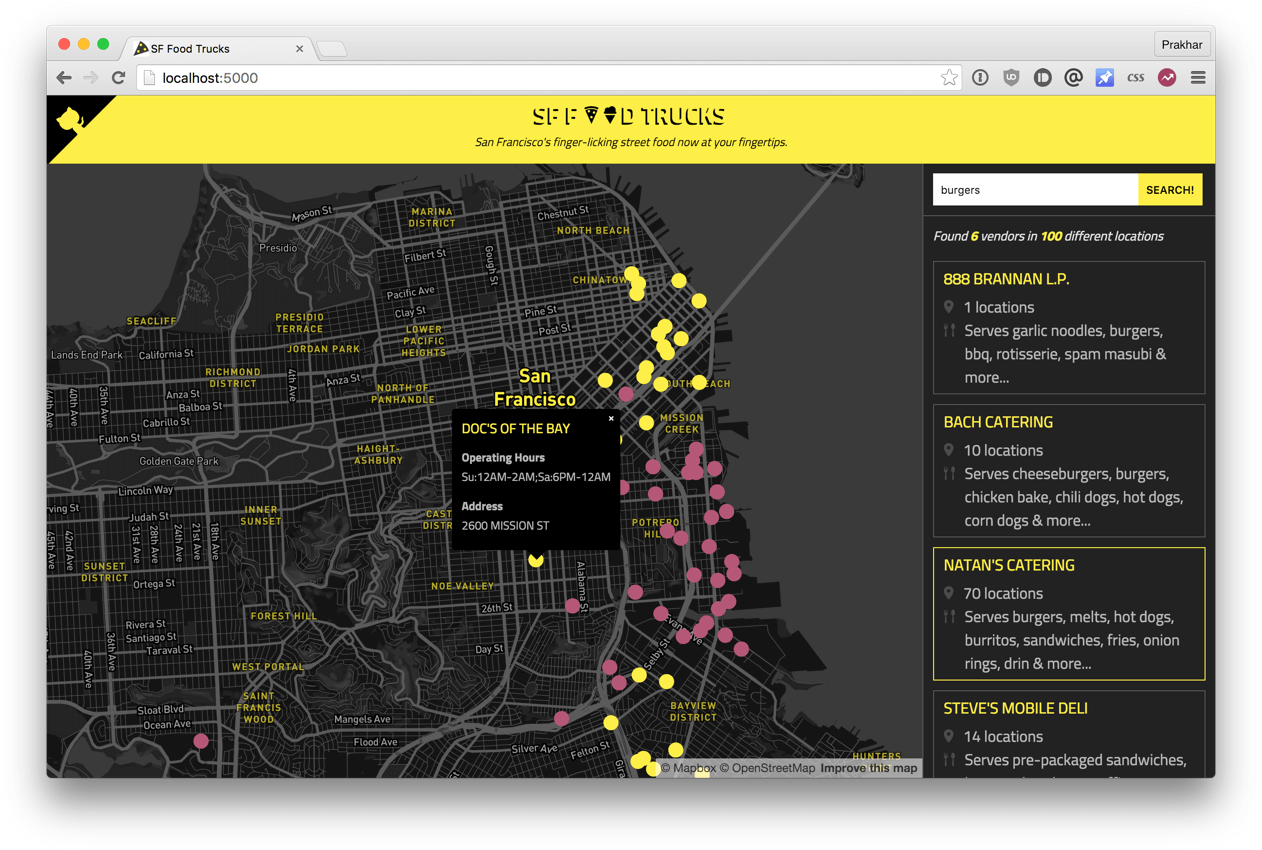
## Module: EGT307 AI Application Development

Practical 3c: Docker Network

In the last tutorial, we saw how easy and fun it is to run applications with Docker. We started with a simple static website and then tried a Flask app. One thing both these apps had in common was that they were running in a single container.

However, usually apps nowadays are not that simple. There's almost always a database (or any other kind of persistent storage) involved. Hence, in this section we are going to spend some time learning how to Dockerize applications which rely on different services to run.

The app that we're going to Dockerize is called SF Food Trucks.



The app's backend is written in Python (Flask) and for search it uses Elasticsearch. We'll use this as our candidate application for learning out how to build, run and deploy a multi-container environment.

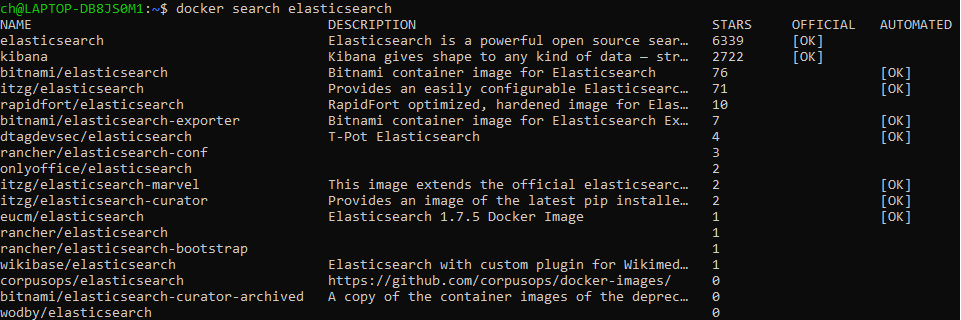
Preparation

1. Run docker stop and docker rm commands to clean up any previous containers.
2. Download and extract “DockerNetwork.zip”.
3. There are multiple files and folders inside the folder “FoodTrucks”. The flask-app folder contains the Python application, while the utils folder has some utilities to load the data into Elasticsearch. The directory also contains a Dockerfile, which we'll edit later.

Activity 1: Elasticsearch Docker Container

Now let's think of how we can Dockerize the app. The application consists of a Flask backend server and an Elasticsearch service. A natural way to split this app would be to have two containers - one running the Flask process and another running the Elasticsearch (ES) process. That way if our app becomes popular, we can scale it by adding more containers depending on where the bottleneck lies.

So we need two containers. Let's see if we can find existing Elasticseach image on the hub.



Quite unsurprisingly, there exists an officially supported image for Elasticsearch. To get ES running, we can simply use docker run and have a single-node ES container running locally within no time.

Let's first pull the image

$ docker pull docker.elastic.co/elasticsearch/elasticsearch:6.3.2

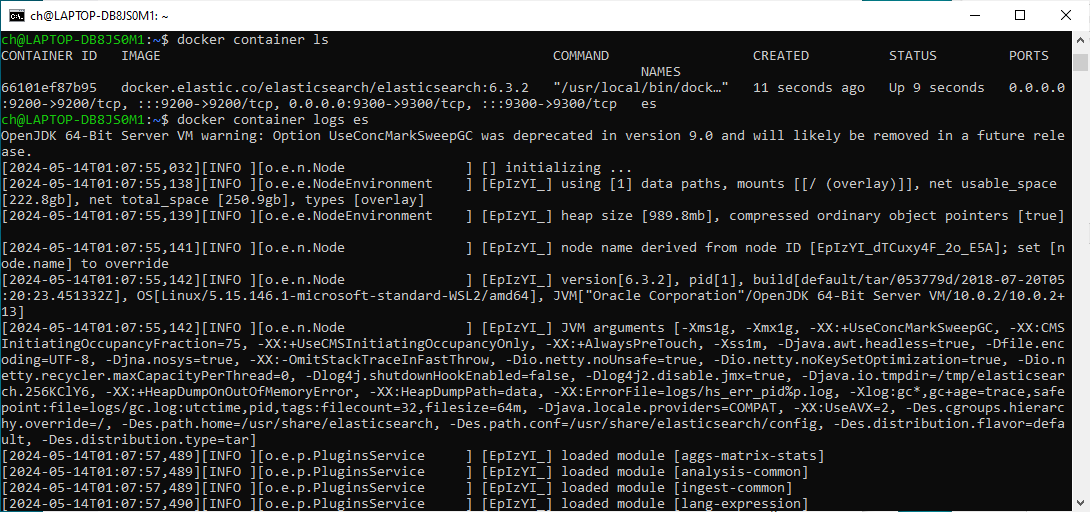
and then run it in development mode by **specifying ports** and **setting an environment variable** that configures the Elasticsearch cluster to run as a single-node.

$ docker run -d --name es -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" docker.elastic.co/elasticsearch/elasticsearch:6.3.2

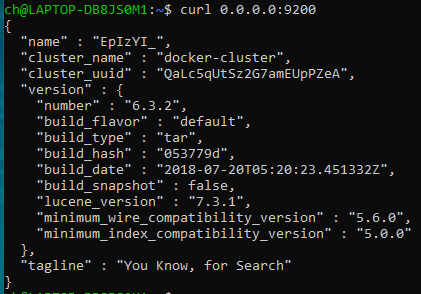
As seen above, we use --name es to give our container a name which makes it easy to use in subsequent commands.

Once the container is started, we can see the logs by running docker container logs with the container name (or ID) to inspect the logs. You should see logs similar to below if Elasticsearch started successfully.

Note: Elasticsearch takes a few seconds to start so you might need to wait before you see initialized in the logs.



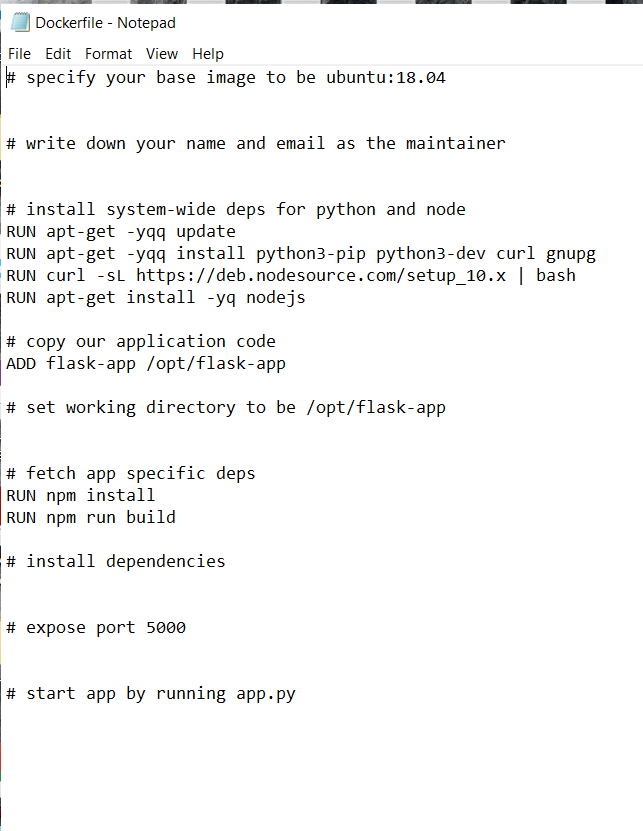
Now, lets try to see if we can send a request to the Elasticsearch container. We use the 9200 port to send a cURL request to the container.



Activity3: Flask Docker Container

Now, let's get our Flask container running too. But before we get to that, we need a Dockerfile. In the last practical, we used python:3.8 image as our base image. This time, however, we'll start from the ubuntu base image to build our Dockerfile.

Edit the Dockerfile accordingly – there should be a line of code under each comment.



After you are done, go ahead to build the image and run the container (replace yourusername with your username below).

Note: make sure you are in the same directory of your Dockerfile (i.e. FoodTrucks folder).

$ docker build -t yourusername/foodtrucks-web .

In the first run, this will take some time as the Docker client will download the ubuntu image, run all the commands and prepare your image.

You should see the following output upon successful build.



Re-running docker build after any subsequent changes you make to the application code will almost be instantaneous. Now let's try running our app.

$ docker run -P --rm yourusername/foodtrucks-web

The –P options is to publish all exposed ports.

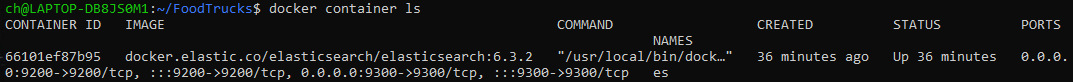
A black background with white text

Description automatically generated

Oops! Our flask app was unable to run since it was unable to connect to Elasticsearch. How do we tell one container about the other container and get them to talk to each other? The answer lies in the next section.

Activity4: Docker Network

Okay, so let's run docker container ls (which is same as docker ps) and see what we have.



So we have one ES container running on 0.0.0.0:9200 port which we can directly access from the host machine i.e. from my PC. Unfortunately, another container will not be able to access ES container using this IP address.

Now is a good time to start our exploration of networking in Docker. When docker is installed, it creates three networks automatically.

A black background with white text

Description automatically generated

The bridge network is the network in which containers are run by default. So that means that when I ran the ES container, it was running in this bridge network. To validate this, let's inspect the network.

A screen shot of a computer program

Description automatically generated

Although we have figured out ES can be accessed at 172.17.0.2:9200, there are still two problems with this approach -

1. How do we tell the Flask container that es hostname stands for 172.17.0.2 or some other IP since the IP can change?
2. Since the bridge network is shared by every container by default, this method is not secure. How do we isolate our network?

The above 2 questions can be solved by using Docker network command. It allows us to define our own networks while keeping them isolated.

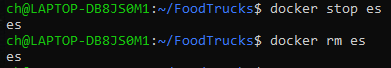
Let's first go ahead and create our own network.

A computer screen shot of a black screen

Description automatically generated

The network create command creates a new bridge network, which is what we need at the moment. A bridge network uses a software bridge which allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network.

Now that we have a network, we can launch our containers inside this network using the --net flag. Let's do that - but first, in order to launch a new container with the same name, we will stop and remove our ES container that is running in the bridge (default) network.



Lets run the same docker command again, but now with the --net flag to connect to the “foodtrucks-net” network we created.

$ docker run -d --name es --net foodtrucks-net -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" docker.elastic.co/elasticsearch/elasticsearch:6.3.2



As you can see, our es container is now running inside the foodtrucks-net bridge network.

Let's launch our Flask container now.

$ docker run -d --net foodtrucks-net -p 5000:5000 --name foodtrucks-web yourusername/foodtrucks-web

$ docker container ls

A screenshot of a computer code

Description automatically generated

Head over to http://localhost:5000 and see your glorious app live!

