DOCKER_BASICS.DOCX

AUTHORS: Prateek Kumar
DATE CREATED: 01/02/2022
LAST UPDATED: 01/02/2022
VERSION: 0.1
FILE NAME: DOCKER_BASICS.DOCX

TABLE OF CONTENTS

1 Document Purpose

This document is to understand and learn how to build and deploy distributed applications easily to the cloud using Docker.

2 Introduction

2.1 What is Docker?

Docker is a tool that allows developers, system admins etc. to easily deploy their applications in a sandbox (called *containers*) to run on the host operating system. The key benefit of Docker is that it allows users to package an application with all its dependencies into a standardized unit for software development. Unlike virtual machines, containers do not have high overhead and hence enable more efficient usage of the underlying system and resources.

2.2 What are containers?

A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

2.3 Why use containers?

The industry standard today is to use Virtual Machines (VMs) to run software applications. VMs run applications inside a guest Operating System, which runs on virtual hardware powered by the server's host OS. VMs are great at providing full process isolation for applications: there are very few ways a problem in the host operating system can affect the software running in the guest operating system, and vice-versa. But this isolation comes at great cost — the computational overhead spent virtualizing hardware for a guest OS to use is substantial.

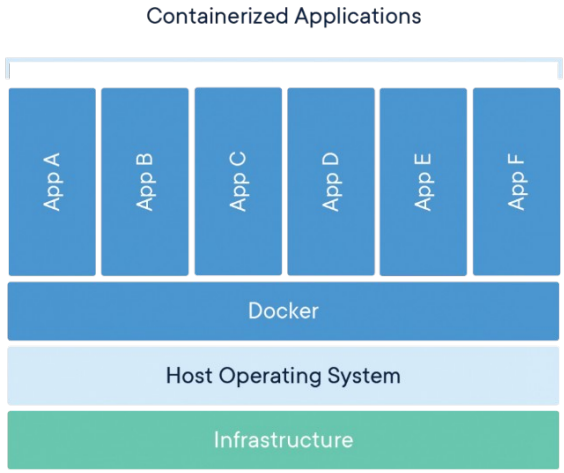
Containers take a different approach: by leveraging the low-level mechanics of the host operating system, containers provide most of the isolation of virtual machines at a fraction of the computing power. Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer's personal laptop. This gives developers the ability to create predictable environments that are isolated from rest of the applications and can be run anywhere.

2.4 Containers vs Virtual Machines

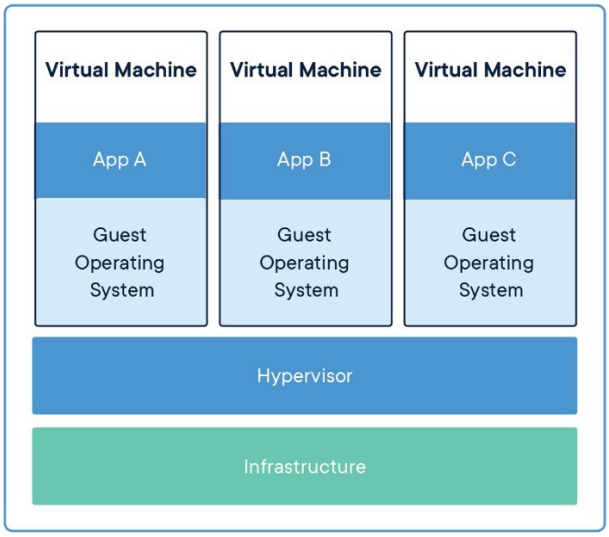
Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware. Containers are more portable and efficient.

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with

other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.



Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.



3 Prerequisites

3.1 Setting up docker on the system

The following hyperlinks can be used to setup docker on different OSes.

- Mac - https://docs.docker.com/docker-for-mac/install/
- Linux - https://docs.docker.com/install/linux/docker-ce/ubuntu/
- Windows - https://docs.docker.com/docker-for-windows/install/

Once you are done installing Docker, test your Docker installation by running the following:

```
$ docker run hello-world
Hello from Docker.
This message shows that your installation appears to be working correctly.
...
```

4 Understanding common Docker terminologies

To get started, run a BusyBox (a software suite that provides several Unix utilities in a single executable file) container on the system to understand **docker run** command.

```
docker pull busybox
```

Note: Depending on how you've installed docker on your system, you might see a permission denied error after running the above command. If you're on a Mac, make sure the Docker engine is running. If you're on Linux, then prefix your docker commands with sudo.

The pull command fetches the busybox **image** from the **Docker registry** and saves it to our system. Use the docker images command to see a list of all images on the system.

```
$ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
busybox                   latest              c51f86c28340        4 weeks
ago         1.109 MB
```

4.1 Docker run

Run the Docker container based on this image using **docker run** command

```
$ docker run busybox
```

Note: When you call run, the Docker client finds the image (busybox in this case), loads up the container and then runs a command in that container. When we run *docker run busybox*, we didn't provide a command, so the container booted up, ran an empty command and then exited.

Run the below command

```
docker run busybox echo "hello from busybox"
```

We see the below output

```
hello from busybox
```

4.2 Docker ps

The **docker ps** command shows all containers that are currently running.

```
$ docker ps
CONTAINER ID          IMAGE                  COMMAND                  CREATED
STATUS              PORTS               NAMES
```

If no containers are running, we see a blank line like above. A more useful variant is **docker ps -a**

```
$ docker ps -a CONTAINER ID STATUS 305297d7a235
                    IMAGE               COMMAND             CREATED
                           PORTS               NAMES
                    busybox             "uptime"minutes ago distracted_goldstine
Exited (0) 11 minutes ago                       minutes ago elated_ramanujan
ff0a5c3750b9busybox"sh" Exited (0) 12 minutes ago2 minutes ago thirsty_euclid
14e5bd11d164hello-world"/hello" Exited (0) 2 minutes ago
```

4.3 Docker rm

Leaving stray containers will eat up disk space. Hence, as a rule of thumb, clean up containers once done with them. To do that, run the docker rm command. Copy the container IDs and paste them alongside the command. Eg:

```
$ docker rm 305297d7a235 ff0a5c3750b9
305297d7a235
ff0a5c3750b9
```

On deletion, the IDs are echoed back. For deleting a bunch of containers in one go, copy-pasting IDs can be tedious. In that case, simply run

```
$ docker rm (docker ps -a -q -f status=exited)
```

This command deletes all containers that have a status of exited. The -q flag returns the numeric IDs and -f filters output based on conditions provided.
The docker container prune command can be used to achieve the same effect.

```
$ docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
4a7f7eebae0f63178aff7eb0aa39f0627a203ab2df258c1a00b456cf20063
f98f9c2aa1eaf727e4ec9c0283bcaa4762fbdba7f26191f26c97f64090360

Total reclaimed space: 212 B
```

To delete images which are no longer needed use **docker rmi <image_id>**

5 Deploying Web Applications on Docker

5.1 Docker images

Docker images are the basis of containers. In the previous example, we pulled the *Busybox* image from the registry and asked the Docker client to run a container based on that image. To see the list of images that are available locally, use the **docker images** command.

```
$ docker images
```

For simplicity, you can think of an image akin to a git repository. Images can be committed with changes and have multiple versions. If you don't provide a specific version number, the client defaults to latest. For example, you can pull a specific version of ubuntu image

```
$ docker pull ubuntu:18.04
```

To get a new Docker image you can either get it from a registry (such as the Docker Hub) or create your own. An important distinction to be aware of when it comes to images is the difference between base and child images.

- Base images are images that have no parent image, usually images with an OS like ubuntu, busybox or debian.
- Child images are images that build on base images and add additional functionality.

Then there are official and user images, which can be both base and child images.

- Official images are images that are officially maintained and supported by the folks at Docker. These are typically one word long. In the list of images above, the python, ubuntu, busybox and hello-world images are official images.
- User images are images created and shared by users like you and me. They build on base images and add additional functionality. Typically, these are formatted as user/image-name.

5.2 First Image

Our goal in this section will be to create an image that sandboxes a simple Flask application. For the purposes I've already created a Flask app that displays a random cat .gif every time it is loaded. Clone the repository locally like so –

```
$ git clone https://github.com/PrateekKumar1709/Docker-Curriculam.git
$ cd docker-curriculum/flask-app
```

This should be cloned on the machine where you are running the docker commands and not inside a docker container. The next step now is to create an image with this web app. As mentioned above, all user images are based on a base image. Since our application is written in Python, the base image we're going to use will be Python 3.

5.3 Dockerfile

A Dockerfile is a simple text file that contains a list of commands that the Docker client calls while creating an image. It's a simple way to automate the image creation process.

The application directory does contain a Dockerfile but since we're doing this for the first time, we'll create one from scratch. To start, create a new blank file in a text-editor and save it in the **same** folder as the flask app by the name of Dockerfile. Start with specifying the image. Use the **FROM** keyword to do that -

```
FROM python:3
```

The next step usually is to write the commands of copying the files and installing the dependencies. First, we set a working directory and then copy all the files for our app.

```
# set a directory for the app
WORKDIR /usr/src/app
# copy all the files to the container
COPY . .
```

Now, that we have the files, we can install the dependencies.

```
# install dependencies
RUN pip install --no-cache-dir -r requirements.txt
```

The next thing we need to specify is the port number that needs to be exposed. Since our flask app is running on port 5000, that's what we'll indicate.

```
EXPOSE 5000
```

The last step is to write the command for running the application, which is simply - python ./app.py. We use the CMD command to do that -

```
CMD ["python", "./app.py"]
```

The primary purpose of CMD is to tell the container which command it should run when it is started. With that, our Dockerfile is now ready. This is how it looks like -

```
FROM python:3
# set a directory for the app
WORKDIR /usr/src/app
# copy all the files to the container
COPY . .
# install dependencies
RUN pip install --no-cache-dir -r requirements.txt
# tell the port number the container should expose
```

```
EXPOSE 5000
# run the command
CMD ["python", "./app.py"]
```

5.4 Docker build

Now that we have our Dockerfile, we can build our image. The **docker build** command creates a Docker image from a Dockerfile.

The section below shows you the output of running the same. Before you run the command yourself (don't forget the period), make sure to replace my username with yours. This username should be the same one you created when you registered on Docker hub. If you haven't done that yet, please go ahead and create an account. The docker build command is quite simple - it takes an optional tag name with -t and a location of the directory containing the Dockerfile.

```
$ docker build -t yourusername/catnip .
Sending build context to Docker daemon 8.704 kB
Step 1 : FROM python:3
# Executing 3 build triggers...
Step 1 : COPY requirements.txt /usr/src/app/
 ---> Using cache
Step 1 : RUN pip install --no-cache-dir -r requirements.txt
 ---> Using cache
Step 1 : COPY . /usr/src/app
 ---> 1d61f639ef9e
Removing intermediate container 4de6ddf5528c
Step 2 : EXPOSE 5000
 ---> Running in 12cfcf6d67ee
 ---> f423c2f179d1
Removing intermediate container 12cfcf6d67ee
Step 3 : CMD python ./app.py
 ---> Running in f01401a5ace9
 ---> 13e87ed1fbc2
Removing intermediate container f01401a5ace9
Successfully built 13e87ed1fbc2
```

If you don't have the python:3 image, the client will first pull the image and then create your image. Hence, your output from running the command will look different from mine. If everything went well, your image should be ready! Run **docker images** and see if your image shows. The last step in this section is to run the image and see if it works (replacing my username with yours).

```
$ docker run -p 8888:5000 yourusername/catnip
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

The command we just ran used port 5000 for the server inside the container and exposed this externally on port **8888**. Head over to the URL with port 8888, where your app should be live.

5.5 Docker push

There are many different Docker registries you can use (you can even host your own). For now, let's use Docker Hub to publish the image. To publish, just type

```
$ docker push yourusername/catnip
```

If this is the first time you are pushing an image, the client will ask you to login. Provide the same credentials that you used for logging into Docker Hub.

```
$ docker login
Username: yourusername
WARNING: login credentials saved in
/Users/yourusername/.docker/config.json
Login Succeeded
```

Remember to replace the name of the image tag above with yours. It is important to have the format of username/image_name so that the client knows where to publish. Once that is done, you can view your image on Docker Hub. For example, here's the web page for my image. Now that your image is online, anyone who has docker installed can play with your app by typing just a single command.

```
$ docker run -p 8888:5000 yourusername/catnip
```

6 Running multiple containers

Just like it's a good strategy to decouple application tiers, it is wise to keep containers for each of the **services** separate. Each tier is likely to have different resource needs and those needs might grow at different rates. By separating the tiers into different containers, we can compose each tier using the most appropriate instance type based on different resource needs.

6.1 SF Food Trucks

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. First up, let's clone the repository locally.

```
$ git clone https://github.com/PrateekKumar1709/SF-Food-Trucks.git
```

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 some YAML files and a Dockerfile.

We can see that 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. 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. Pull the image using the below command:

```
$ 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. Use ls to view the elasticsearch container

```
$ docker container ls
```

Now, let’s 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.

```
$ curl 0.0.0.0:9200
```

The expected output should be like:

```
{
  "name" : "ijJDAOm",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "a_nSV3XmTCqpzYYzb-LhNw",
  "version" : {
    "number" : "6.3.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "053779d",
    "build_date" : "2018-07-20T05:20:23.451332Z",
    "build_snapshot" : false,
    "lucene_version" : "7.3.1",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

Before starting the Flask container, we need a Dockerfile. In the last section, we used python:3 image as our base image. This time, however, apart from installing Python dependencies via pip, we want our application to also generate our minified Javascript file for production. For this, we'll require Nodejs. Since we need a custom build step, we'll start

from the ubuntu base image to build our Dockerfile from scratch. Our Dockerfile for the flask app looks like below -

```
# start from base
FROM ubuntu:latest
MAINTAINER Prateek Kumar
# install system-wide deps for python and node
RUN apt-get -yqq update
RUN apt-get -yqq install python-pip python-dev curl gnupg
RUN curl -sL https://deb.nodesource.com/setup_10.x | bash
RUN apt-get install -yq nodejs
# copy our application code
ADD flask-app /opt/flask-app
WORKDIR /opt/flask-app
# fetch app specific deps
RUN npm install
RUN npm run build
RUN pip install -r requirements.txt
# expose port
EXPOSE 5000
# start app
CMD [ "python", "./app.py" ]
```

We start off with the Ubuntu LTS base image and use the package manager apt-get to install the dependencies namely - Python and Node. The yqq flag is used to suppress output and assumes "Yes" to all prompts. We then use the ADD command to copy our application into a new volume in the container - /opt/flask-app. This is where our code will reside. We also set this as our working directory, so that the following commands will be run in the context of this location. Now that our system-wide dependencies are installed, we get around to installing app-specific ones. First off we tackle Node by installing the packages from npm and running the build command as defined in our package.json file. We finish the file off by installing the Python packages, exposing the port and defining the CMD to run.

Finally, we can go ahead, build the image and run the container (replace prateek1709 with your username below).

```
$ docker build -t prateek1709/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. 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 prateek1709/foodtrucks-web
Unable to connect to ES. Retying in 5 secs...
Unable to connect to ES. Retying in 5 secs...
Unable to connect to ES. Retying in 5 secs...
Out of retries. Bailing out...
```

The Flask app was unable to run since it was unable to connect to Elasticsearch as the two containers were not able to talk to each other.

6.2 Docker Network

We have one ES container running on 0.0.0.0:9200 port which we can directly access. We need the Flask app to connect to this URL. In our Python code the connection details are defined as follows:

```
es = Elasticsearch(host='es')
```

When docker is installed, it creates three networks automatically.

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c2c695315b3a        bridge              bridge              local
a875bec5d6fd        host                host                local
ead0e804a67b        none                null                local
```

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.

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id":
"c2c695315b3aaf8fc30530bb3c6b8f6692cedd5cc7579663f0550dfdd21c9a26",
        "Created": "2018-07-28T20:32:39.405687265Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers":
{          "277451c15ec183dd939e80298ea4bcf55050328a39b04124b387d668e3ed394
3": {
                "Name": "es",
```

```
                "EndpointID":
"5c417a2fc6b13d8ec97b76bbd54aaf3ee2d48f328c3f7279ee335174fbb4d6bb",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

We see that container **277451c15ec1** is listed under the Containers section in the output. What we also see is the IP address this container has been allotted - **172.17.0.2**
To verify that we can access elasticsearch on the above IP we can run the container and try to connect to elasticsearch from within the container using **cURL**. Use the following commands for the same

```
$ docker run -it --rm prateek1709/foodtrucks-web bash
root@35180ccc206a:/opt/flask-app# curl 172.17.0.2:9200

{
  "name" : "Jane Foster",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.1.1",
    "build_hash" : "40e2c53a6b6c2972b3d13846e450e66f4375bd71",
    "build_timestamp" : "2015-12-15T13:05:55Z",
    "build_snapshot" : false,
    "lucene_version" : "5.3.1"
  },
  "tagline" : "You Know, for Search"
}

root@35180ccc206a:/opt/flask-app# exit
```

We start the container in the interactive mode with the bash process. The --rm is a convenient flag for running one off commands since the container gets cleaned up when it's work is done. We try a cURL, but we need to install it first. Once we do that, we see that we can indeed talk to ES on 172.17.0.2:9200

Although we have figured out a way to make the containers talk to each other, 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 solution is docker allows us to define our own networks while keeping them isolated using the **docker network** command. Use the following command to create your own network.

```
$ docker network create foodtrucks-net
0815b2a3bb7a6608e850d05553cc0bda98187c4528d94621438f31d97a6fea3c

$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c2c695315b3a        bridge              bridge              local
0815b2a3bb7a        foodtrucks-net      bridge              local
a875bec5d6fd        host                host                local
ead0e804a67b        none                null                local
```

The **network create** command creates a new *bridge* network. In terms of Docker, 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. The Docker bridge driver automatically installs rules in the host machine so that containers on different bridge networks cannot communicate directly with each other.

To launch the containers inside this network, use 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.

```
$ docker container stop es
es

$ docker container rm es
es

$ 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
13d6415f73c8d88bddb1f236f584b63dbaf2c3051f09863a3f1ba219edba3673

$ docker network inspect foodtrucks-net
[
    {
        "Name": "foodtrucks-net",
        "Id":
"0815b2a3bb7a6608e850d05553cc0bda98187c4528d94621438f31d97a6fea3c",
        "Created": "2018-07-30T00:01:29.1500984Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
```

```
"IPAM": {
"Driver": "default",
"Options": {}, "Config": [
{
"Subnet": "172.18.0.0/16",
"Gateway": "172.18.0.1"
}
]
},
"Internal": false, "Attachable": false, "Ingress": false, "ConfigFrom": {
"Network": ""
},
"ConfigOnly": false, "Containers": {
"13d6415f73c8d88bddb1f236f584b63dbaf2c3051f09863a3f1ba219edba3673": {
                "Name": "es",
                "EndpointID":
"29ba2d33f9713e57eb6b38db41d656e4ee2c53e4a2f7cf636bdca0ec59cd3aa7",
              "MacAddress": "02:42:ac:12:00:02",
              "IPv4Address": "172.18.0.2/16",
              "IPv6Address": ""
          }
      },
      "Options": {},
      "Labels": {}
  }
]
```

As you can see, our es container is now running inside the foodtrucks-net bridge network. Now let's inspect what happens when we launch in our foodtrucks-net network.

```
$ docker run -it --rm --net foodtrucks-net prateek1709/foodtrucks-web bash
root@9d2722cf282c:/opt/flask-app# curl es:9200
{
  "name" : "wWALl9M",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "BA36XuOiRPaghPNBLBHleQ",
  "version" : {
    "number" : "6.3.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "053779d",
    "build_date" : "2018-07-20T05:20:23.451332Z",
    "build_snapshot" : false,
    "lucene_version" : "7.3.1",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
```

```
  },
  "tagline" : "You Know, for Search"
}
root@53af252b771a:/opt/flask-app# ls
app.py  node_modules  package.json  requirements.txt  static  templates
webpack.config.js
root@53af252b771a:/opt/flask-app# python app.py
Index not found...
Loading data in elasticsearch ...
Total trucks loaded:  733
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
root@53af252b771a:/opt/flask-app# exit
```

On user-defined networks like foodtrucks-net, containers can not only communicate by IP address, but can also resolve a container name to an IP address. This capability is called ***automatic service discovery*.**

To launch the Flask container, use the below command:

```
docker run -d --net foodtrucks-net -p 5000:5000 --name foodtrucks-web
prateek1709/foodtrucks-web
852fc74de2954bb72471b858dce64d764181dca0cf7693fed201d76da33df794
```

```
$ docker container ls
CONTAINER ID        IMAGE
  COMMAND                  CREATED              STATUS              PORTS
NAMES
852fc74de295        prateek1709/foodtrucks-web
  "python ./app.py"        About a minute ago   Up About a minute
  0.0.0.0:5000->5000/tcp                        foodtrucks-web
13d6415f73c8        docker.elastic.co/elasticsearch/elasticsearch:6.3.2
"/usr/local/bin/dock…"   17 minutes ago       Up 17 minutes
0.0.0.0:9200->9200/tcp, 0.0.0.0:9300->9300/tcp   es
```

```
$ curl -I 0.0.0.0:5000
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 3697
Server: Werkzeug/0.11.2 Python/2.7.6
Date: Sun, 10 Jan 2016 23:58:53 GMT
```

To view the app, go to **http://0.0.0.0:5000**

7 Docker Compose

7.1 What is Docker Compose?

Compose is a tool that is used for defining and running multi-container Docker apps in an easy way. It provides a configuration file called **docker-compose.yml** that can be used to bring up an application and the suite of services it depends on with just one command.

7.2 Installing Docker Compose

On Windows or Mac Docker Compose is already installed as it comes in the Docker Toolbox. For Linux use the below hyperlink.

https://docs.docker.com/compose/install/

Since Compose is written in Python, you can also simply do **pip install docker-compose**. Test your installation with –

```
$ docker-compose –version
```

7.3 Creating Docker Compose file

The syntax for YAML is quite simple and the repo already contains the docker-compose file that we'll be using.

```
version: "3"
services:
  es:
    image: docker.elastic.co/elasticsearch/elasticsearch:6.3.2
    container_name: es
    environment:
      - discovery.type=single-node
    ports:
      - 9200:9200
    volumes:
      - esdata1:/usr/share/elasticsearch/data
  web:
    image: prateek1709/foodtrucks-web
    command: python app.py
    depends_on:
      - es
    ports:
      - 5000:5000
    volumes:
      - ./flask-app:/opt/flask-app
volumes:
    esdata1:
      driver: local
```

At the parent level, we define the names of our services - es and web. For each service that Docker needs to run, we can add additional parameters out of which image is required. For es, we just refer to the elasticsearch image available on Elastic registry. For our Flask app, we refer to the image that we built at the beginning of this section.

Via other parameters such as command and ports we provide more information about the container. The volumes parameter specifies a mount point in our web container where the code will reside. This is purely optional and is useful if you need access to logs etc. We also add volumes for the es container so that the data we load persists between restarts. We also specify depends_on, which tells docker to start the es container before web.

7.4 Docker-compose up

Navigate to the food trucks directory and run **docker-compose up**.

```
$ docker-compose up
Creating network "foodtrucks_default" with the default driver
Creating foodtrucks_es_1
Creating foodtrucks_web_1
Attaching to foodtrucks_es_1, foodtrucks_web_1
  es_1| [2016-01-11 03:43:50,300][INFO ][node                     ]
  [Comet] version[2.1.1], pid[1], build[40e2c53/2015-12-15T13:05:55Z] es_1| [2016-01-1
                                                                    
[Comet] initializing ...
  es_1  | [2016-01-11 03:43:50,366][INFO ][plugins                  ]
[Comet] loaded [], sites []
  es_1  | [2016-01-11 03:43:50,421][INFO ][env                      ]
[Comet] using [1] data paths, mounts [[/usr/share/elasticsearch/data
(/dev/sda1)]], net usable_space [16gb], net total_space [18.1gb], spins?
[possibly], types [ext4]
  es_1  | [2016-01-11 03:43:52,626][INFO ][node                     ]
[Comet] initialized
  es_1  | [2016-01-11 03:43:52,632][INFO ][node                     ]
[Comet] starting ...
  es_1  | [2016-01-11 03:43:52,703][WARN ][common.network           ]
[Comet] publish address: {0.0.0.0} is a wildcard address, falling back to
first non-loopback: {172.17.0.2}
  es_1  | [2016-01-11 03:43:52,704][INFO ][transport                ]
[Comet] publish_address {172.17.0.2:9300}, bound_addresses {[::]:9300}
  es_1  | [2016-01-11 03:43:52,721][INFO ][discovery                ]
[Comet] elasticsearch/cEk4s7pdQ-evRc9MqS2wqw
  es_1  | [2016-01-11 03:43:55,785][INFO ][cluster.service          ]
[Comet] new_master {Comet}{cEk4s7pdQ-evRc9MqS2wqw}{172.17.0.2}
{172.17.0.2:9300}, reason: zen-disco-join(elected_as_master, [0] joins
received)
  es_1  | [2016-01-11 03:43:55,818][WARN ][common.network           ]
[Comet] publish address: {0.0.0.0} is a wildcard address, falling back to
first non-loopback: {172.17.0.2}
  es_1  | [2016-01-11 03:43:55,819][INFO ][http                     ]
[Comet] publish_address {172.17.0.2:9200}, bound_addresses {[::]:9200}
  es_1  | [2016-01-11 03:43:55,819][INFO ][node                     ]
[Comet] started
```

```
  es_1  | [2016-01-11 03:43:55,826][INFO ][gateway                  ]
[Comet] recovered [0] indices into cluster_state
es_1  | [2016-01-11 03:44:01,825][INFO ][cluster.metadata         ]
[Comet] [sfdata] creating index, cause [auto(index api)], templates [],
shards [5]/[1], mappings [truck]
es_1  | [2016-01-11 03:44:02,373][INFO ][cluster.metadata         ]
[Comet] [sfdata] update_mapping [truck]
es_1  | [2016-01-11 03:44:02,510][INFO ][cluster.metadata         ]
[Comet] [sfdata] update_mapping [truck]
es_1  | [2016-01-11 03:44:02,593][INFO ][cluster.metadata         ]
[Comet] [sfdata] update_mapping [truck]
es_1  | [2016-01-11 03:44:02,708][INFO ][cluster.metadata         ]
[Comet] [sfdata] update_mapping [truck]
es_1  | [2016-01-11 03:44:03,047][INFO ][cluster.metadata         ]
[Comet] [sfdata] update_mapping [truck]
web_1 |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Head over to the IP to see the app running.