Table of Contents

[1.0 Introduction 3](#_Toc952930)

[1.1 Purpose 3](#_Toc952931)

[2.0 Installation 3](#_Toc952932)

[3.0 Docker Terminology 4](#_Toc952933)

[3.1 Images 5](#_Toc952934)

[3.2 Containers 5](#_Toc952935)

[3.3 Docker Daemon 5](#_Toc952936)

[3.4 Docker Client 5](#_Toc952937)

[3.5 Docker hub 5](#_Toc952938)

[4.0 Deploying a web application with Docker 5](#_Toc952939)

[4.1 Static Sites 5](#_Toc952940)

[4.2 Docker Images 6](#_Toc952941)

[4.3 Node 6](#_Toc952942)

[4.4 Chef Server 7](#_Toc952943)

[4.5 Resources 7](#_Toc952944)

[4.6 Cookbook 8](#_Toc952945)

[4.7 Attributes 8](#_Toc952946)

[4.8 File 8](#_Toc952947)

[4.9 Policy 9](#_Toc952948)

[5.0 Installing ChefDK 9](#_Toc952949)

[5.1 Configuring ChefDK 9](#_Toc952950)

[3.1.1 Creating the chef repo 9](#_Toc952951)

[3.1.2 Create the .chef directory 9](#_Toc952952)

[3.1.3 Starter Kit 9](#_Toc952953)

[5.2 Configure the Chef Repository – with Web UI 10](#_Toc952954)

[5.3 Configure the Chef Repository – without Web UI 11](#_Toc952955)

[3.3.1 Create an Organization on the Chef Server 11](#_Toc952956)

[3.3.2 Create the User 12](#_Toc952957)

[3.3.3 Create the config.rb file 12](#_Toc952958)

[3.3.4 Get SSL Certificate 12](#_Toc952959)

[3.3.5 Verify Installation 12](#_Toc952960)

[6.0 Installing Chef Server 13](#_Toc952961)

[6.1 Installation Process 13](#_Toc952962)

[7.0 Installing Chef Automate 13](#_Toc952963)

[7.1 Chef Automate Server Installation and Configuration 14](#_Toc952964)

[7.2 Configuring node data collection 15](#_Toc952965)

[5.2.1 Configure a Data Collector token in Chef Automate 15](#_Toc952966)

[5.2.2 Configure your Chef server to point to Chef Automate 15](#_Toc952967)

[8.0 Bootstrapping a node 16](#_Toc952968)

[8.1 Knife bootstrap 16](#_Toc952969)

[8.2 Prerequisites 17](#_Toc952970)

[9.0 Berkshelf 17](#_Toc952971)

[9.1 Using Cookbooks from Supermarket 18](#_Toc952972)

[10.0 Linux Patching though Chef 19](#_Toc952973)

[10.1 Prerequisites 19](#_Toc952974)

[8.1.1 Each node must be bootstrapped. 19](#_Toc952975)

[8.1.2 Each of the server must be categorized based on environments. 20](#_Toc952976)

[10.2 Getting the List of Nodes to be patched from Linux Team 20](#_Toc952977)

[10.3 Pulling the list of available patches. 20](#_Toc952978)

[8.4.1 Update the audit.txt 20](#_Toc952979)

[8.4.2 Execute the cookbook 21](#_Toc952980)

[8.4.3 Generate the reports. 21](#_Toc952981)

[10.4 Applying the patches 22](#_Toc952982)

# Introduction

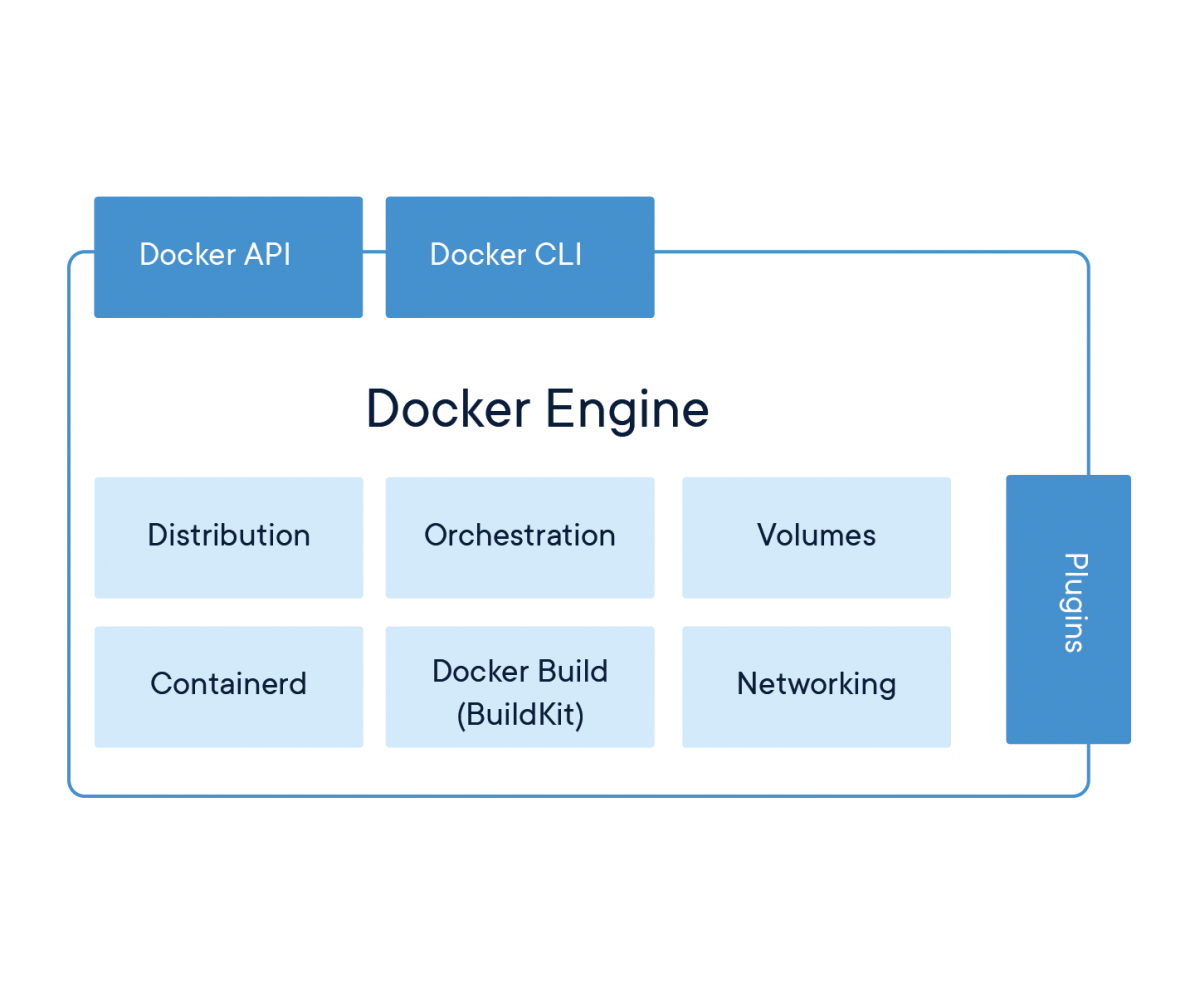
## Purpose

<https://docker-curriculum.com/>

Docker is an open-source project that automates the deployment of software applications inside **containers** by providing an additional layer of abstraction and automation of **OS-level virtualization** on Linux.

The key benefit of Docker is that it allows users to **package an application with all of its dependencies into a standardized unit** for software development.

Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually 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.



# Installation

* Uninstall Older version of docker.

*$ sudo apt-get remove docker docker-engine docker.io containerd runc*

* Install using the repository
  + Update the apt package
    - $ sudo apt-get update
  + Install package to allow apt to use a repository over HTTPS
    - $ sudo apt-get install \

apt-transport-https \

ca-certificates \

curl \

gnupg-agent \

software-properties-common

* + Add the GPG keys
    - $ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
  + To set up the stable repository
    - $ sudo add-apt-repository \

"deb [arch=amd64] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) \

stable"

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

*$ docker run hello-world*

The *docker pull* command fetches the image from the Docker registry and saves it to our system. You can use the *docker image* command to see the list of all the images downloaded.

When we invoke the *docker run* command, the Docker client finds the image, loads up the container and then runs the command in the container.

$ docker run busybox echo "hello from busybox"

The *docker ps* command shows all the container that are currently running. Running the *docker run* command with *–it* flags attaches us to an interactive tty in the container.

You can use the *docker rm* command to remove a container. Just copy the container id and pass it as a parameter of the command.

In case you want to delete a number of docker container, you can use the command

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

The command to remove docker image *docker rmi*

# Docker Terminology

## Images

The blueprints of our application which form the basis of containers. We use the *docker pull* command to pull an image from the repository.

## Containers

Created from Docker images and run the actual application. We create a container using *docker run.* A list of all the available container can be seen using the *docker ps* command.

## Docker Daemon

The background service running on the host that manages building, running and distributing Docker containers. The daemon is the process that runs in the operating system to which clients talk to.

## Docker Client

The command line tool that allows the user to interact with the daemon.

## Docker hub

A registry of docker images. It’s a directory of all the available images.

# Deploying a web application with Docker

## Static Sites

The first thing we're going to look at is how we can run a dead-simple static website. We're going to pull a Docker image from Docker Hub, run the container and see how easy it is to run a webserver.

Source:<https://github.com/prakhar1989/docker-curriculum/blob/master/static-site/html/index.html>

<https://hub.docker.com/r/prakhar1989/static-site/>

We can download and run the image directly in one go using docker run, --rm removes the container when exits.

*$ docker run --rm prakhar1989/static-site*

*$ docker run -d -P --name static-site prakhar1989/static-site*

In the above command, -d will detach our terminal, -P will publish all exposed ports to random ports and finally --name corresponds to a name we want to give. Now we can see the ports by running the docker port [CONTAINER] command.

You can also use customized ports to which client will forward connections to the container.

*$ docker run -p 8888:80 prakhar1989/static-site*

## Docker Images

To list the available images, we use the command *docker images*. The TAG refers to particular snapshot of the image. Images have versions as well, if you don’t specify a particular version, then the latest image will be pulled down. You can specify a particular version by the following

*$ docker pull Ubuntu: 12.04*

Clone the git repository: <https://github.com/prakhar1989/docker-curriculum/>

Navigate to the folder docker-curriculum/flask-app.

The next step now is to create an image with this web app. As mentioned above, all user images are based off of a base image.

## Docker file

A Dockerfile is a simple text-file that contains the list of commands that the Docker client calls while creating an image.

We start with specifying our base image. Use the **FROM** keyword to do that

**FROM python:3-onbuild**

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

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

The primary purpose of CMD is to tell the container which command it should run when it is started.

Thus our final Docker file will look like the following

# our base image

FROM python:3-onbuild

# specify the port number the container should expose

EXPOSE 5000

# run the application

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

Now that we have the Dockerfile, we need to build our image. The **docker build** command does the heavy-lifting of creating a **Docker image** from a **Dockerfile**.

**$ docker build -t prakhar1989/catnip .**

Don’t forget the add the period.

If you don’t have the python:3-onbuild image, the client will first pull the image and then will create your image.

The last section is to run the image and see if the containers are actually created. To do so, execute the following

**$ docker run -p 8888:5000 prakhar1989/catnip**

This command, used port 5000 for the server inside the container and exposed this externally on the port 8888.

## Docker Push

The first thing that we need to do before we deploy our app to AWS is to publish our image on a registry which can be accessed by AWS. To publish the image, just type the following

**$ docker push prakhar1989/catnip**

Since this is the first time, you need to login. Use the **docker login** command to log into.

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 prakhar1989/catnip**

## Beanstalk

AWS Elastic Beanstalk (EB) is a PaaS (Platform as a Service) offered by AWS.  As a developer, you just tell EB how to run your app and it takes care of the rest - including scaling, monitoring and even updates. In April 2014, EB added support for running single-container Docker deployments which is what we'll use to deploy our app.

Steps:

* Login into AWS
* Click on Elastic Bean Stalk
* Create on new application
* Give a name and description to your App
* In the New Environment Screen, create a new environment and chose the Web Server Environment.
* Fill in the environment details. Remember the URL
* Under the basic configuration, choose Docker from the predefined platform
* Now, we need to upload our application code. In our case this is a container hence we need to tell EB about the container.
* Open the Dockerrun.aws.json file located in flaskapp folder, edit its name to your image name. Once done, upload the code and choose this file
* The final screen that you see will have a few spinners indicating that your environment is being set up. It typically takes around 5 minutes for the first-time setup

Let’s analysis the doockerrun.aws.json file.

{

"AWSEBDockerrunVersion": "1",

"Image": {

"Name": "prakhar1989/catnip",

"Update": "true"

},

"Ports": [

{

"ContainerPort": "5000"

}

],

"Logging": "/var/log/nginx"

}

## Multi Container Environments

Running Service in the production knows that usually apps now a days are not simple. Here we are going to learn how to Dockerize applications which rely on different services to run.

Just like it's a good strategy to decouple your 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.

* + 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. The entire source code is available in GIT.

<https://github.com/prakhar1989/FoodTrucks>

We will use this as the candidate application to learn how to build, run and deploy a multi-container environment.

First, let’s clone this application locally.

$ git clone https://github.com/prakhar1989/FoodTrucks

$ cd FoodTrucks

$ tree -L 2

.

├── Dockerfile

├── README.md

├── aws-compose.yml

├── docker-compose.yml

├── flask-app

│ ├── app.py

│ ├── package-lock.json

│ ├── package.json

│ ├── requirements.txt

│ ├── static

│ ├── templates

│ └── webpack.config.js

├── setup-aws-ecs.sh

├── setup-docker.sh

├── shot.png

└── utils

├── generate\_geojson.py

└── trucks.geojson

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.

Now let’ss start to Dockerize the application. 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.

Great, so we need two containers. That shouldn't be hard right? We've already built our own Flask container in the previous section. And for Elasticsearch, let's see if we can find something on the hub.

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 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**

Once the container is started, we can see the logs by running the command **docker continer logs**

Now let’s try to see if we can send request to the Elasticsearch container.

$ curl 0.0.0.0:9200

{

"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"

}

Now for our Flask container, let’s build a Dockerfile from scratch. Our dockerfile will look like thr following

# start from base

FROM ubuntu:14.04

MAINTAINER Prakhar Srivastav <prakhar@prakhar.me>

# 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\_8.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" ]

Next, we go ahead and build the image and run the container.

**$ docker build -t prakhar1989/foodtrucks-web .**

Now, let’s try running the application

**$ docker run -P --rm prakhar1989/foodtrucks-web**

Now our Flask application was not able to communicate with the Elasticsearch. To enable this we need the Docker network.

* + 1. Docker Network.

Let’s run the **docker container ls** (same as **docker ps**) and see what we have. This gives, that we have an Elasticsearch container running on **0.0.0.0 :9200** port which we can directly access. We need to tell our flask application to connect to this URL. For this we need to explore our Python Code.

When Docker is installed, it creates three networks automatically

* Bridge
* Host
* None

$ 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. To validate this, lets 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": {

"277451c15ec183dd939e80298ea4bcf55050328a39b04124b387d668e3ed3943": {

"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": {}

}

]

You can see that our container is listed under the Container section. We can also see the IP address that is assigned to this container.

Let’s try to run our application across this IP.

We are starting 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.

**$ docker run -it --rm prakhar1989/foodtrucks-web bash**

**root@35180ccc206a:/opt/flask-app# curl 172.17.0.2:9200**

By this method we are able to make the containers communicate with one another. There are still couple of problems with this approach.

* 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?
* Since the *bridge* network is shared by every container by default, this method is **not secure**. How do we isolate our network?

Docker allows us to define our own networks while keeping them isolated using the **Docker network create** command.

So let’s create our own docker network.

**$ docker network create foodtrucks-net**

Once created, we can verify the existence of the networks.

**$ docker network ls**

$ 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 create a new bridged network. In terms, of Docker, a bridge uses a software bridge which allows containers connected to the same bridge to communicate, while providing isolations from containers that are not connected to the 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 learn more about the Docket Network, use the following: <https://docs.docker.com/network/>

Now that we have a network, we can lunch our containers on this network using the **–net** flag.

Thus we can lunch a container though

**$ 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**

Now, let’s inspect the network to check if our container is connected to the network.

**$ docker network inspect foodtrucks-net**

$ 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": {}

}

]

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*.

Now let’s lunch our Flask container for real now (on our created network)

**$ docker run -d --net foodtrucks-net -p 5000:5000 --name foodtrucks-web prakhar1989/foodtrucks-web**

So for going from zero to running application containing couple of containers, only has the following 4 steps. Which can be included in the bash script.

#!/bin/bash

# build the flask container

docker build -t prakhar1989/foodtrucks-web .

# create the network

docker network create foodtrucks-net

# start the ES container

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

# start the flask app container

docker run -d --net foodtrucks-net -p 5000:5000 --name foodtrucks-web prakhar1989/foodtrucks-web

Now imagine you are distributing your app to a friend, or running on a server that has docker installed. You can get a whole app running with just one command!

$ git clone https://github.com/prakhar1989/FoodTrucks

$ cd FoodTrucks

$ ./setup-docker.sh

* + 1. Docker Compose