Docker Reference Material

Please feel free to reach me for any consulting/training requirements

Jeganathan Swaminathan

mail2jegan@gmail.com

jegan@tektutor.org

Few key points to keep in mind about Docker

- is a Linux Technology that supports Application Virtualization
- is a Client/Server Technology
- is developed in Go Programming Language by Docker Inc organization
- is not a replacement for Virtual Machines as Containers are Application Process while Virtual Machines are a complete Operating System.
- As Containers are assigned with Private IPs, they appear like a VM, however they don't even have their own Kernel.
- All Docker containers run on the Host machine Kernel space just like how other application processes.
- is a way to ship your pre-installed applications along with its dependencies
- can be used with legacy applications as well by modern applications that follow microservice architecture.
- comes in two flavours
    - Community Edition (CE) - Open Source
    - Enterprise Edition (EE) - Commercial Use
- Depends on Linux Kernel Feature
    - Namespace - For isolation, Network namespace, Port namespace etc.
    - CGroups (Control Groups) - For quota restricts like CPU usage, memory and storage utilization, etc
- When Docker is installed on Mac - OSX or Windows it installs a thin-linux layer. Hence to support Linux containers, Docker containers are still created on top of Linux Layer on Windows/Mac OS.
- On Windows 10 & Later, Docker supports windows containers with the help of Hyper-V tiny virtual machines to simulate containers similar to Linux containers.

Docker Images

- a specification of docker container
- a blueprint of a docker container
- similar to ISO OS images or VMWARE images
- Open sources images can be downloaded from Docker Hub (hub.docker.com)
- Custom Docker images may be created using Dockerfile

Docker Containers

- is an instance of Docker Image
- every containers get an unique docker id assigned by Docker Engine (Server)
- every container get's its own Private IP by default
- are light-weight as they don't get their own dedicated CPU cores, RAM and Storage unlike Virtual Machines.
- every container can be assigned an user-defined container name and hostname optionally
- In case, no container name is allotted by the user, Docker Engine assigns a random name.
- In case, no hostname is allotted by the user, Docker Engine assigns container id as the hostname
- containers typically has one single application along with its dependencies
- In case, more than one application is installed inside a container, Docker uses supervisord to monitor the additional processes created to run each of those applications. Hence generally container images must restrict one application per container.
- has its own Port range 0 to 65535
- the ports used internally by the Docker containers won't conflict with host machine ports or other containers unless the Container uses Host Network.

Some most commonly used Docker commands

Listing images

```
docker images
```

Inspect Docker Image to find more details about the image

```
docker image inspect ubuntu:16.04
```

Listing only currently running containers

```
docker ps
```

Listing all containers irrespective of their running status

```
docker ps -a
```

To download a docker image from Docker Hub(hub.docker.com)

```
docker pull hello-world:latest
```

To create a docker container in foreground mode(interactive)

docker run hello-world
docker run -it --name ubuntu1 --hostname ubuntu1 ubuntu:16.04 /bin/bash

In the above command

- -it stands for interactive terminal
- ubuntu1 - is the docker container name
- ubuntu1 - is the hostname of the container
- ubuntu:16.04 - is the image name with version 16.04
- /bin/bash - blocking application that will be launched inside container

Stopping a running container

docker stop ubuntu1

Starting a exited container

docker start ubuntu1

Opening a second shell inside a running container

docker exec -it ubuntu1 /bin/bash

Finding IP address of a running container

docker inspect ubuntu1 | grep IPA
docker inspect -f "{{ .NetworkSettings.IPAddress }}" ubuntu1

Finding IP address of a container from within container shell

hostname -i

Finding Hostname of a container from within container shell

hostname

In order to provide internet access to your containers, make sure the below configuration is done on CentOS Lab machine

vim /etc/sysctl.conf and add the below line
net.bridge.bridge.nf-call-iptables=1

The above line shall be added for machines that support IPV4, in case your machine also uses IPV6, you may also add the below line
net.bridge.bridge-nf-call-ip6tables=1

Make sure the below services are restarted after the above changes are made
systemctl daemon-reload
systemctl restart network

```
systemctl restart docker
```

You may start the container that originally had trouble connecting to internet as shown below

```
docker start ubuntu1
```

Get inside the ubuntu1 container using below command

```
docker exec -it ubuntu1 bash
```

Trying installing some tools to verify if Internet works

```
apt update && apt install -y vim
```

Creating MYSQL Docker container

```
docker run --name mysql-server  --hostname mysql-server -e MYSQL_ROOT_PASSWORD=root -d mysql:5.6
```

Get inside the mysql-server container with the below command

```
docker exec -it mysql-server  /bin/bash
mysql -u root -p
```

You need to type root as the password to login to mysql prompt. On successful login, you will see a prompt as shown below

```
mysql >
```

In the mysql prompt, you may type the below command to display all the existing databases

```
mysql > SHOW DATABASES;
```

In case you would like to create a database

```
mysql > CREATE DATABASE tektutor;
```

Before you can create a table, you need to select a database first,

```
mysql > USE tektutor;
```

You may now create a table inside tektutor database as shown below

```
mysql > CREATE TABLE Training(id integer NOT NULL UNIQUE, name varchar(25), duration varchar(10));
```

You may now insert a record as shown below

```
INSERT INTO Training VALUES ( 1, 'DevOps', '5 days' );
```

You may now see the records in the table as shown below

```
SELECT * FROM Training;
```

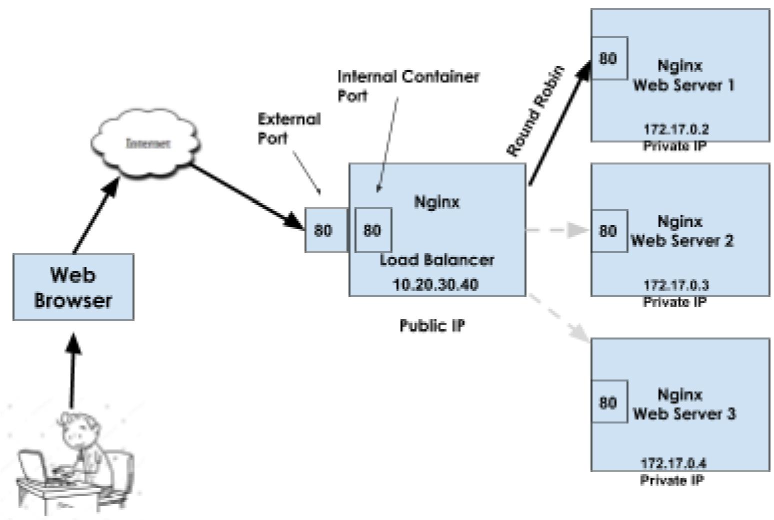
Volume Mounting

In order to persist the application data, application logs, etc it is recommended to mount an external storage volume inside the container. Otherwise, whenever the container gets deleted the data stored inside the container also gets deleted.

```
docker run --name mysql-server --hostname mysql-server -e
MYSQL_ROOT_PASSWORD=root -d -v /home/jegan/tmp:/var/lib/mysql  mysql:5.6
```

The above command is a single line command, as it is a very lengthy command it is word wrapped.

Setting up Nginx as a Load Balancer with Docker Containers



You need to create 3 nginx web server as shown below

```
docker run -d --name nginx1 --hostname nginx1 nginx:1.16
```

```
docker run -d --name nginx2 --hostname nginx2 nginx:1.16
```

```
docker run -d --name nginx3 --hostname nginx3 nginx:1.16
```

You need to create a nginx load balancer container as shown below

```
docker run -d --name lb --hostname lb -p 80:80 nginx:1.16
```

In order to configure the lb container to work as a load balancer

we need to first copy the nginx.conf from the container to the local machine

```
docker cp lb:/etc/nginx/nginx.conf  .
```

You need to edit nginx.conf on the centos lab machine with any text editor.

vim nginx.conf and make sure the file looks as below

```
user  nginx;
worker_processes  1;
error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;
events {
  worker_connections  1024;
}
http  {
  upstream backend {
      server 172.17.0.2:80;
      server 172.17.0.3:80;
      server 172.17.0.4:80;
  }
  server {
      location / {
          proxy_pass http://backend;
      }
  }
}
```

In the above file,

172.17.0.2 is the ip address of nginx1 container

172.17.0.3 is the ip address of nginx2 container

172.17.0.4 is the ip address of nginx3 container

You may need to replace the ip addresses of your containers.

In order to apply the configuration changes in the load balancer container

`docker restart lb`

Make sure the lb container is actually running after the config changes

```
docker ps
```

Once you have made sure the lb container is running, then you may find the IP Address of your CentOS Lab machine as shown below.

```
ifconfig ens33
```

You may customize the web pages on nginx1, nginx2 and nginx3 respectively as shown below

```
echo “Server 1” > index.html
docker cp index.html nginx1:/usr/share/nginx/html/index.html
echo “Server 2” > index.html
docker cp index.html nginx2:/usr/share/nginx/html/index.html
echo “Server 3” > index.html
docker cp index.html nginx3:/usr/share/nginx/html/index.html
```

In my system IP Address of my CentOS machine happens to be 172.16.124.251

From the Alchemy Windows Cloud machine, open the browser with the URL as shown below

http://172.16.124.251

Each time you refresh the web page on the above URL, you may see the output as

Server 1

Server 2

Server 3 in a round robin fashion.

Building Custom Docker Images

IN order to build a custom ubuntu docker image, you may get my latest Dockerfile from my GitHub repository as shown below

On the terminal, login as root user

```
sudo su -
```

To build custom ansible ubuntu image

```
git clone https://github.com/tektutor/ubuntu-ansible.git
cd ubuntu-ansible
```

You may now create public/private key pair as shown below
ssh-keygen

Accept all default values by hitting enter while generating public/private key pairs.

Assuming, you generated the keys as the root user. You may copy the public key (id_rsa.pub) from /root/.ssh/id_rsa.pub as shown below
cp /root/.ssh/id_rsa.pub authorized_keys

You may now build your custom ubuntu images as shown below
docker build -t tektutor/ansible-ubuntu .

If you would like to create docker

To build custom ansible centos image
git clone https://github.com/tektutor/centos-sshd-passwordless.git

cd centos-sshd-passwordless
cp /root/.ssh/id_rsa.pub authorized_keys

You may now build your custom centos image as shown below
docker build -t tektutor/ansible-centos .

You may now list the newly build images as shown below
docker images | grep tektutor

In order to test if the ansible node images are working as expected, let us create couple of containers from these newly build images as demonstrated below
docker run -d --name ubuntu1 --hostname ubuntu1 -p 2001:22 -p 8001:80 tektutor/ansible-ubuntu
docker run -d --name centos1 --hostname centos1 -p 2002:22 -p 8002:80 tektutor/ansible-centos

You may check if the containers are in running state as shown below
docker ps

Let's try to login ot these containers as demonstrated below
ssh -p 2001 root@localhost

You may need to accept yes when it prompts to confirm adding the container fingerprints to known_hosts file.

However, it is important to observe that the login happens without prompting for a password as we have set up the custom docker images to perform key based login authentication.

Docker Networking

Docker supports

- bridge network (default)
- host network (container will not get it own ip address)
- none (containers that don't need network access shall be connected to this network type)

In case you wish to create your own custom bridge network in Dockers, you may try the below command

docker network create my-net-work-1

You may inspect the my-net-work-1 interface to identify the subnet (ip cidr block)

docker network inspect my-net-work-1

You may create another custom bridge network with name 'my-net-work-2' as shown below

docker network create my-net-work-2 --subnet 172.20.0.0/16

Let's create a container c1 and connect c1 to network my-network-1, and create a container c2 and connect c2 to network my-network-2

docker run -dit --name c1 --hostname c1 --network=my-net-1 ubuntu:16.04 /bin/bash
docker run -dit --name c2 --hostname c2 --network=my-net-2 ubuntu:16.04 /bin/bash

You may login to container c1 and try to ping the c2 container

docker exec -it c1 bash
ping 172.20.0.2

You may now observe that container c1 couldn't reach container c2 as they belong to different networks. On the similar note, container c2 couldn't reach container c1 for the same reason.

In case you wish c1 to communicate with c2 and vice versa, you may connect c1 to my-net-2 network in addition to already connected my-net-1.

docker network connect my-net-2 c1

After the above step, c1 should be able to ping c2 and c2 in turn should be able to ping c1.

Lab 2 - Docker Networking

In the lab exercise, we will set up a multi-container web application.

We will be using the below images for this lab exercise from the Docker Hub
1. wordpress:latest
2. mysql:5.6

First create a mysql container as shown below
```
docker run --name mysql-server --hostname mysql-server -e MYSQL_ROOT_PASSWORD=root -d mysql:5.6
```

You may verify if the mysql-server is running
```
docker ps  |  grep mysql-server
```

The output should be something similar to this
```
4422ec3a797      mysql:5.6        "docker-entrypoint.s…"   3 minutes ago      Up 3
minutes      3306/tcp          mysql-server
```

You may now create the wordpress container as shown below
```
docker run --name wordpress-server -e WORDPRESS_DB_HOST=172.17.0.2:3306 \
   -e WORDPRESS_DB_USER=root -e WORDPRESS_DB_PASSWORD=root -d wordpress
```

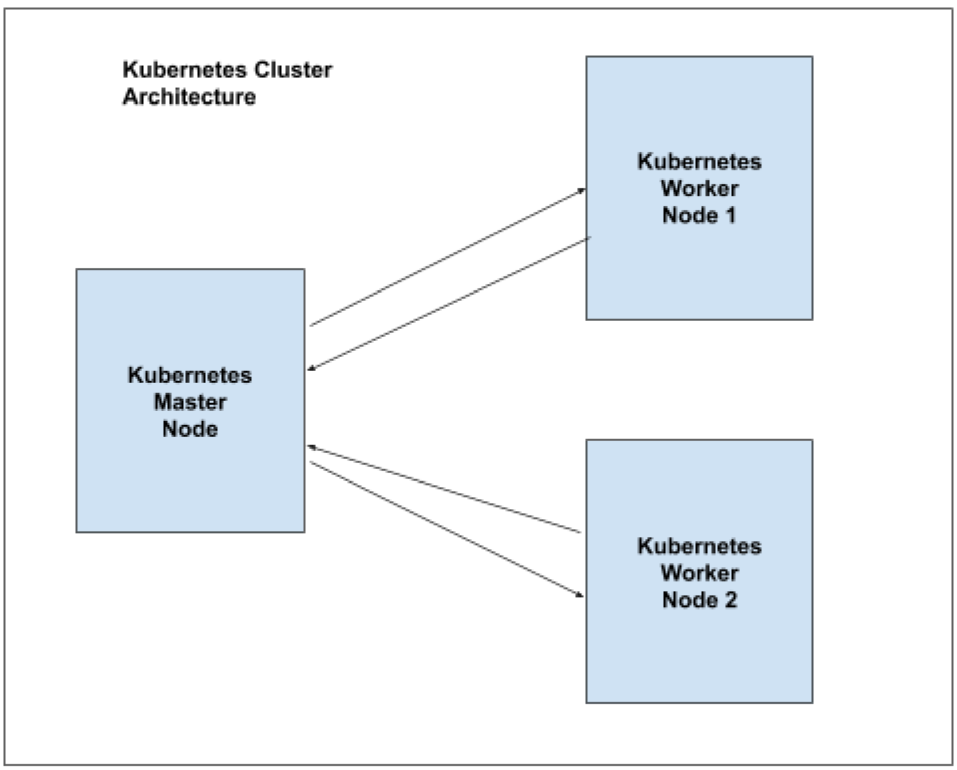
In the above command IP 172.17.0.2 is the IP Address of mysql-server created in the previous step. You may need to find the IP Address of mysql and replace the IP as required.

You may now verify if both wordpress-server and mysql-server containers are in running state with the below command
```
docker ps
```

You are expected to see an output as shown below
```
CONTAINER ID      IMAGE            COMMAND                 CREATED           STATUS
PORTS             NAMES
cd2fcdc2acb6      wordpress        "docker-entrypoint.s…"   10 minutes ago     Up 10
minutes      80/tcp            wordpress-server
a4422ec3a797      mysql:5.6        "docker-entrypoint.s…"   21 minutes ago     Up 21
minutes      3306/tcp          mysql-server
```

Kubernetes
- an Orchestration Platform
- developed by Google and donated to Open Source Community

- Docker SWARM is an alternate for Kubernetes
- RedHat(IBM) OpenShift
- Orchestration Feature Set
    - Monitoring the health of you application
    - You can setup Highly Available (HA) application
        - fault tolerant application
        - self-healing application
    - Depending on traffic demand, you will be able to scale up/down your application instance count
    - You can roll out your latest application onto live production server without any downtime
    - manages the containers
        - LXC
        - Rkt
        - Docker (default)



Kubernetes Master Node Components

1. API Server (Pod) - All the K8s functionalities are implemented as REST API
2. Controller Managers (Pod) - Monitoring
3. Scheduler (Pod) - Which identifies a healthy node where user application can be deployed

4. Etcd database (Pod) - Key/Value Datastore
5. Kube Proxy (Pod) - This component helps in master- worker node communication and worker - master node communication. There will be one kube-proxy component for each node.
6. Kubelet Agent (Daemon Service) - There will be a single instance of kubelet agent on every kubernetes node including the master node.

Docker must be installed on Master as well as Worker Nodes.

Kubernetes Worker Node Components

1. Kubelet Agent (Daemon Service) - This is the component that is responsible to pull the missing Docker images that are required for deploying user pods.
2. Kube-Proxy

Kubernetes Jargons

Deployment

- represents your application.
- creates and manages ReplicaSet
- supports scaling up/down the number of application pod instances
- rolling update
    - CRM 2.0 version must be rolled out to live production servers by replacing CRM 1.0 without any downtime.

ReplicaSet

manages the Pods
responsible for ensuring the desired number application pods are always running

Pod

manages containers
collection of one of more containers
recommended industry practice suggests that one container per Pod
IP addresses in K8s are assigned on the Pod level unlike Docker
all containers that are part of same Pod shares IP Address, Ports, Network namespace

Lab 3 - Creating Nginx Deployment and exposing a NodePort service

Assumption is that you have a working K8s cluster.

```
kubectl create deployment nginx --image=nginx:1.16
```

The above command will create a deployment with the name “nginx” under default namespace. The nginx deployment in turn will create a ReplicaSet with name “nginx-xxxxxxxx” and the ReplicaSet “nginx-xxxxxxxx” will in turn create pods with name patterns starting with “nginx-xxxxxxxx-yyyy”.

In order to check the deployment(s) in default namespace, you may try the below command

```
kubectl get deployments
kubectl get deploy
```

In order to check the ReplicaSets in default namespace, you may try the below command

```
kubectl get replicasets
kubectl get rs
```

In order to check the pod(s) in default namespace, check the below command

```
kubectl get pods
kubectl get po
```

If you wish see all the K8s object, you may try this

```
kubectl get all
```

In order to a create NodePort Service, you can issue the below command

```
kubectl expose deployment nginx --type=NodePort --port=80
```

Scaling Up/Down Deployment replicas

```
kubectl scale deployment/nginx --replicas=10
kubectl scale deployment/nginx --replicas=2
```

Kubespray Production Ready K8s Cluster

```
apt update && apt install python3-pip
```

```
sudo pip3 install -r requirements.txt
```

TODO - Incomplete

Lab - Creating Nginx Deployment in imperative Style (Manually using CLI)
kubectl create deployment --image=nginx:1.16

kubectl get deploy,rs,po
You will be able to see all deployments, ReplicaSet and pods in the “default” namespace.

In case you wish to edit the live nginx deployment definition
kubectl edit deploy/nginx
kubectl edit deploy nginx

In case you wish to edit the ReplicaSet of the nginx deployment
kubectl edit rs/<ReplicaSet Name>

Note:

For eg:
kubectl edit rs/nginx-77776fdcdc

ReplicaSet names follow a particular pattern. The part of the ReplicaSet indicates the deployment name that created the ReplicaSet and the second part of the ReplicaSet name represents the ReplicaSet ID.

In case you wish to edit any nginx pod definition, you may
kubectl edit pod/<pod-name>

Note:
For eg:

nginx-77776fdcdc-p8mf6

Pod name follows a particular pattern. The part of the pod name string indicates the deployment name, the second part indicates the replicaset and 3rd part indicates the pod-id.

Lab - Creating Nginx Deployment in Declarative Style

Folder - Kubernetes/Day3
vim nginx-dep.yml

apiVersion: extensions/v1beta1

```yaml
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.16
        name: nginx
```

Create the nginx deployment using the above yaml file

```
kubectl apply -f nginx-dep.yml
```

```
kubectl get deploy
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   2/2     2            2           8s
```

```
kubectl get rs
NAME               DESIRED   CURRENT   READY   AGE
nginx-77776fdcdc   2         2         2       12s
```

```
kubectl get po
NAME                     READY   STATUS    RESTARTS   AGE
nginx-77776fdcdc-cfrz8   1/1     Running   0          16s
nginx-77776fdcdc-p8mf6   1/1     Running   0          16s
```

Labels and Selectors

Kubernetes let's label any type of K8s objects. For instance, deployments use labels as a selector to filter out the ReplicaSets whose labels match a particular value.

For eg:
Assume we have two deployments with names nginx and nginx-dep created in our Kubernetes Cluster.
As there are pods created by deployment ‘nginx’ and ‘nginx-dep’, Kubernetes uses labels as Selectors to select their child Replicasets with a particular label value.

How the Service locates the corresponding pods?
`kubectl get pods -l “app=nginx”`

How the Deploy with name nginx will identify its respective ReplicaSet
`kubectl get rs -l “app=nginx”`

How the Deploy with name nginx-dep will identify its respective ReplicaSet
`kubectl get rs -l “app=nginx-dep”`

How ReplicaSet with name nginx-77776fdcdc will identify its corresponding child pods?
`kubectl get pods -l “app=nginx, pod-template-hash=77776fdcdc”`

How ReplicaSet with name nginx-65bd8f6cc4 will identify its corresponding child pods?
`kubectl get pods -l “app=nginx, pod-template-hash=65bd8f6cc4”`

In order to understand the label selectors better. You may create a nginx-pod manually without ReplicaSet and Deployment as shown below

Folder - Kubernetes/Day3
vim nginx-pod.yml

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
    pod-template-hash: 77776fdcdc
  name: nginx-77776fdcdc-aabbcc
  namespace: default
spec:
  containers:
   - image: nginx:1.16
     name: nginx
```

kubectl apply -f nginx-pod.yml

In the above manifest file, you may understand that labels app=nginx and pod-template-hash=77776fdcdc match with the labels of those pods which were created by ReplicaSet with name nginx-77776fdcdc. Hence, as the current number of pods are more than the desired count, the ReplicaSet nginx-77776fdcdc will remove one of the extra pod with label app=nginx and pod-template-hash=77776fdcdc.

Lab :

1. Create a simple hello rest api in Python.
2. Create a Dockerfile with hello rest api python application
3. Build the custom docker image
4. Tag and Push your custom image to Docker Hub
5. Deploy Python Hello Microservice in Kubernetes Cluster as shown

You need to create the python rest api as shown below

Folder: Kubernetes/Day3

Step 1 - Create a simple hello rest api in Python.
hello.py

```
#!/usr/bin/python3
import flask

app = flask.Flask(__name__)

@app.route('/', methods=['GET'])
def home(msg):
   return "Hello Python REST API !"

app.run(host='0.0.0.0', port=80)
```

Step 2 - Create a Dockerfile with hello rest api python application

```
FROM alpine:3.12
MAINTAINER Jeganathan Swaminathan <mail2jegan@gmail.com>
RUN apk add --no-cache python3 py3-pip
RUN pip3 install flask
COPY hello.py /hello.py
WORKDIR /
EXPOSE 80
CMD [ "python3", "hello.py" ]
```

Step 3: Build the custom docker image
docker build -t tektutor/python-hellorest-ms .

Step 4: Tag and Push your custom image to Docker Hub

Note:-
Assumption is you already have a Docker Hub Free Account. If you don't have an account already, please a docker hub account.

```
docker login
```

Type your credentials to see a message "Logged in successfully."
You need to create public repository in Docker Hub with name
tektutor/python-hellorest-ms:1.0

You need to tag image before pushing the image to Docker Hub

```
docker tag tektutor/python-hellorest-ms:latest tektutor/python-hellorest-ms:1.0
docker push tektutor/python-hellorest-ms:1.0
```

Once you make sure the newly tagged image is pushed successfully to Docker Hub, you may proceed with kubernetes deployment.

Step 5: Deploy Python Hello Microservice in Kubernetes Cluster as shown

Create a file python-hellorest-dep.yml

```
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: hello-ms
  name: hello-ms
  namespace: default
spec:
  ports:
  - nodePort: 31500
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: hello-ms
```

```
  type: NodePort
...

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: hello-ms
  name: hello-ms
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hello-ms
  template:
    metadata:
      labels:
        app: hello-ms
    spec:
      containers:
      - image: tektutor/python-hello-ms:1.0
        name: hello-ms
...
```

```
kubectl apply -f python-hellorest-dep.yml
```

You may test the deployed service as shown below

curl http://10.192.0.2:31500

The IP 10.192.0.2 must be replaced with one of the Kubernetes node IP.

You may use the below command to find the node IPs.

kubectl get nodes -o wide

ConfigMap

ConfigMap is used to pass user-defined variables to your application running inside Pods.   ConfigMap may contain any number of arguments in the dictionary format i.e key/value pairs.

Lab : Accessing ConfigMap variables in Kubernetes Deployments.

1. Create a ConfigMap
2. Create a Python rest api that uses environments variables and URI variable
3. Create a Deployment with our custom python application
4. Create a custom Docker image with the python application
5. Test the application via their NodePort service

Note:-

Avoid using Tab keys as YAML files are sensitive to Tab keys, use spacebar instead to avoid syntax errors.

Step 1 - Create a ConfigMap

Folder : Kubernetes/Day4/configmaps

my-configmap.yml

```
apiVersion: v1
kind: ConfigMap
metadata:
   name: my-config-map
data:
   greeting_msg: "Hello Kubernetes!"
   jdk_home: "/usr/lib/java/jdk1.8"
   m2_home: "/usr/share/maven"
```

6. Create a Deployment with our custom python application

Step 2 - Create a Python rest api that uses environments variables and URI variable

hello.py

```
#!/usr/bin/python3
import flask
import os

app = flask.Flask(__name__)
@app.route('/<string:msg>', methods=['GET'])
def home(msg):
   s1 = os.environ.get('MESSAGE')
   s2 = os.environ.get('JDK_HOME')
   s3 = os.environ.get('M2_HOME')
   return s1 + " " + s2 + " " + s3 + " " + msg

app.run(host='0.0.0.0', port=80)
```

In the above the above python application, you may notice

MESSAGE
JDK_HOME
M2_HOME

are environment variables and the function home accepts **msg** argument via REST endpoint GET URL.

Step 3 - Create a Deployment with our custom python application

Folder: Kubernetes/Day4/configmaps
File : **python-rest-dep.yml**

```
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: hello-ms
  name: hello-ms
  namespace: default
spec:
  ports:
  - nodePort: 31500
    port: 80
    protocol: TCP
```

```
    targetPort: 80
  selector:
    app: hello-ms
  type: NodePort
...
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: hello-ms
  name: hello-ms
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hello-ms
  template:
    metadata:
      labels:
        app: hello-ms
    spec:
      containers:
      - image: tektutor/python-hello-ms:3.0
        name: hello-ms
        env:
        - name: MESSAGE
          valueFrom:
             configMapKeyRef:
               name: my-config-map
               key: greeting_msg
        - name: JDK_HOME
          valueFrom:
             configMapKeyRef:
               name: my-config-map
               key: jdk_home
        - name: M2_HOME
          valueFrom:
             configMapKeyRef:
               name: my-config-map
               key: m2_home
...
```

Step 4 - Create a custom Docker image with the python application

Folder: Kubernetes/Day4/configmaps
File : **Dockerfile**

```
FROM alpine:3.12
MAINTAINER Jeganathan Swaminathan <mail2jegan@gmail.com>

RUN apk add --no-cache python3 py3-pip
RUN pip3 install flask
COPY hello.py /hello.py
WORKDIR /

EXPOSE 80

CMD [ "python3", "hello.py" ]
```

You may build the docker image as shown below

```
docker build -t tektutor/python-hellorest-ms .

docker tag tektutor/python-hellorest-ms:latest tektutor/python-hellorest-ms:3.0
docker login
docker push tektutor/python-hellorest-ms:3.0
```

Step 5 - Test the application via their NodePort service

Folder : Kubernetes/Day4/configmaps

```
kubectl  apply -f my-configmap.yml
kubectl apply -f python-rest-dep.yml
```

You may verify if the service is created for the above deployment with the below command

```
kubectl get svc
NAME        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
hello-ms    NodePort    10.100.207.215   <none>        80:31500/TCP   44m
kubernetes  ClusterIP   10.96.0.1        <none>        443/TCP        45h
```

You may find the IP Address of Kubernetes nodes using the below command
kubectl get nodes -o wide

Finally, you can test your nodeport service as shown below

curl http://10.192.0.2:31500

In the above URL,

10.192.0.2 is the master kubernetes node IP

31500 port is the NodePort IP for this particular service.

The expected output would be similar to

Output:
root@tektutor configmaps]# curl http://10.192.0.2:31500/hello
Hello Kubernetes! /usr/lib/java/jdk1.8 /usr/share/maven hello