**DOCKER**

Docker is an open-source platform that enables developers to build, deploy, run, update and manage containers. A container is a standardized, executable component that combines application source code with the operating system and dependencies, required to run that code in any environment.

Simply put, *Docker* is a containerization tool: *Container* is a running environment for Image: An *Image* is a snapshot of an environment where the container runs the software.

**Docker tools and Terms**

1. *DockerFile*

Is a simple text fie containing instructions on how to build the docker container image. *DockerFile* automates the process of Docker image creation.

1. *Docker Images*

Is a read-only template that contain executable application source code as well as al the tools, libraries and dependencies that the application needs to run as a container. When you run the docker image, it becomes an instance of the container.

1. *Docker containers*

Docker containers are the live, running instances of Docker images.

1. *Docker daemon*

This is a service that creates and manages docker images, using the commands from the clients. It serves as the control center of your docker implementation. Docker daemons run on docker host server.

1. *Docker-compose*

Use docker compose to manage multiple container applications, where all containers run on the same docker host. Docker compose creates a .YML file that specifies which services are included in the application.

**To install & run docker on an EC2 instance:**

1. Create an ec2-instance with a ppk key pair
2. Copy the Instance’s public IPV4 address
3. Log into Putty-authenticate ssh and add the key pay credentials

-sudo amazon-linux-extras update -y

-sudo amazon-linux-extras install docker

-sudo service docker start

To add ec-2 user to docker credential: so that you don’t have to use “sudo”

#sudo usermod -a -G docker ec2-user then log out and then log back in.

**DOCKER COMMANDS**

1. #docker pull imageName
2. #docker images 🡪 to see all images
3. #docker run ImageName 🡪 to start the pulled image
4. #docker run -it ImageID -🡪 to create a container out of the pulled image
5. #docker stop ContainerID -🡪 to stop a container
6. #docker start ContainerID -🡪To start a container
7. #docker system 🡪to see the statistics of an image
8. #docker rmi -f (Image ID) 🡪to delete an image
9. #docker images 🡪to see the images & their IDs
10. #docker ps -🡪to see the running docker container
11. #docker ps -a -🡪to see all the containers (both stopped and running)
12. #ls -ltr -🡪to see all things out of the container
13. #docker logs ContainerID -🡪to see the log history of a container
14. #docker stats --🡪 to see the resources used by the container i.e its CPU, memory etc
15. #docker system prune 🡪 to delete a container
16. #service docker start

**Steps to create a docker image: i.e, create a docker image called Python:**

1. Create a dockerfile i.e, #vi Dockerfile
2. Build a docker image from the file: i.e #docker build -t python .
3. Run #docker images to get its ID
4. Then run the image i.e #docker run -it imageID

**DOCKERFILE SYNTAX**

Dockerfile syntax consist of 2 main line blocks: Comments and Commands + arguments e.g:

* Line blocs used for commenting include #Print “welcome home”
* Command argument i.e RUN echo “Hellow world”; where RUN is the command and “Hellow world” is the argument.
* Anything with a “#” inside the file will be ignored during execution

**Examples of commands used in Dockerfile:**

1. FROM – the FROM command is the most crucial directive in a Dockerfile. It defines the base image to use in order to start the building process

Example usage: #Usage

FROM Ubuntu

1. MAINTAINER -The MAINTAINER command specifies the author of the file

Example usage: MAINTAINER <name> and <email address>

1. RUN -RUN command is the central executing directive for Dockerfile. It takes a command as its argument and runs it to form the image. RUN command is used to build the image

-Example usage: #Usage

RUN apt-get install -y java

1. CMD command- there can only be one CMD instruction in the dockerfile. The CMD command provides default for an executing container
2. LABEL – The LABEL instruction add metadata to an image. To include spaces within a LABEL, use “quotes and backslashes” as you would in command-line parsing. LABELs are additives including FROM images.

Usage: LABEL <key>=<value> [<key>=<value>…..]

1. EXPOSE -this command informs docker that the container listens on the specified network ports

Usage: EXPOSE <port> [<port>….]

1. ENV -Used to set the environment variable <key> to the value <value>.

Usage: ENV <key> <value>

ENV <key>=<value> [<key>=<value>………]

1. VOLUME -used to create a mount point with the specified name and marks it as a holding externally mounted volumes from native host.

Usage: VOLUME [“<path>”,……]

VOLUME <path> [<path>……]

**Creating a customized docker file & pushing to Docker hub**

You need to create the docker file in order to create the container image from it.

1. Create the file ie #vi Dockerfile
2. Rename the file with the build & (dot) command i.e #docker build -t NewFileName .(dot)
3. Docker run -it DockerID
4. Go to your docker hub and coy your user name
5. #docker login –username (paste your username)
6. Go to your dockerhub & create a repository with the name of your file- copy this repo name
7. Go to your Putty/command-line and run #docker push (paste the repo name)

**DOCKER COMPOSE**

Used to create multiple containers at the same time. You can use compose in different ways i.e to develop environments or to automate testing environments.

#docker compose -v

#vim docker-compose.yml

SERVICES:

Image:

Web: nginx

#docker-compose config

#docker-compose up -d -🡪 to build the compose file in a detached mode

#docker-compose ps

#docker-compose up –scale database=3

**Example of a docker-compose**

version: "3.9" # optional since v1.27.0

services:

web:

build: .

ports:

- "8000:5000"

volumes:

- .:/code

- logvolume01:/var/log

depends\_on:

- redis

redis:

image: redis

volumes:

logvolume01: {}

Docker-compose up command helps to create and start one or more containers

**DOCKER VOLUMES**

Docker volumes are useful for ensuring data persistence while working in containers.

Docker volumes are file systems mounted on docker containers to preserve data generated by the running container. A given volume can be mounted into multiple containers simultaneously.

Docker automatically creates a directory for the volume on the host under the /var/lib/docker/volume/path

**Docker volume commands**

#docker volume –help 🡪 to see options

#docker volume create VolumeName

#docker volume ls

#docker volume inspect VolumeName 🡪 tolist all the details of the volume including its location (mountpoint)

#docker volume rm VolumeName 🡪to delete volume

#docker volume prune -🡪to delete all unused volumes

Creating a container with the new volume

#docker run --namecontainerName -v VolumeName:/var/(docker image path) -p port number docker image

for example, to create a Jenkins container with a volume called lilly-vol: run

**#docker run –name newcontainer -vol -v lilly—vol: /var/Jenkins\_home -p 5000:5000 Jenkins**

-p flag is used to publish a container’s port to the host. Therefore, -p 5000:5000 option maps port 5000 on the container to port 5000 on the host machine. This is useful when running Jenkins inside a docker container as it allows you to access Jenkins from your host machine’s web browser.

The default Jenkins installation runs on ports 8080 and 8443. Port 8080 exposes the web interface, while port 50000 gives you access to a remote java API. HTTP/HTTPS typically runs on ports 80 and 443

Also, Jenkins settings are saved to /var/jenkins\_home

For example, To attach a volume (lilly-vol) to a Jenkins container called lilly-container, run

#docker run –namelilly-container -v lilly-vol: /var/Jenkins\_home -p 9191:8080 -p 60000:50000 Jenkins

**KUBERNETES/K8s**

Kubernetes is a container orchestration platform for scheduling and automating the deployment, management and scaling of containerized applications.

INSTALLATION & ENABLING HYPERV

Installing Kubernetes: [www.kubernetes.io](http://www.kubernetes.io) 🡪documentation 🡪getting started 🡪tools

Ensure your HyperV is enabled on your local machine: To enable HyperV

run: #Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All

or **through GUI**:

-Open control panel🡪Programs🡪programs &features🡪Turn windows features On or Off🡪 examine Hyper-V and the hypervisor platform from the window.

Confirm its enabled by running:

**#Get-windowsOptionFeature-Online-featureNameMicrosoft-HyperV**

#minikube start --driver=hyperV

Or #miniube start –driver-virtualbox if using a virtual box

**Kubernetes Architecture**

1. *Clusters and Nodes*- Clusters are the building blocks of K8s architecture. Clusters are made up of nodes.

-Each cluster consist of a master node that serves as the control plan for the cluster, and multiple worker nodes that deploy, run and manage the containerized applications.

-each worker node includes the tool used to manage the containers e.g docker; and a software agent called a kubelet that receives and executes orders from the master node

b. *Pods*- A pod is the smallest unit of deployable resource in k8s. Pods are a group of containers that share the same compute resources and software and the same network.

1. **Master Node**

Master node makes up the control plane of a cluster and is responsible for scheduling tasks and monitoring the state of the cluster. It is responsible for running the following processes:

1. API Server- this is the entry point to the k8s cluster, which is itself container. The server allows communication between different 8s clients, and the cluster. The clients include the CLI, GUI and k8s dashboard.
2. Controller Manager- keeps track of the state of the cluster
3. Scheduler – Ensures proper pod placement on worker nodes based on factors such as availability.
4. Etcd – This is the key-value storage responsible for holding the state of the cluster at any given time. Etcd has the configuration information and status data of each node in the cluster.
5. **Worker Node**

Worker nodes are part of the 8s clusters, which execute the containers and applications on them. The worker nodes have 3 main components;

-the kubelet service->Each worker node has a kubelet process running on it that allows clusters to talk to each other. It listens to instructions from the API server and manages containers running on a node.

-The kube-proxy service>This is a service responsible for enabling communication between resources within the cluster.

-Container run time ->all worker nodes have docker containers for each application running on them.

**Cluster-🡪Nodes (servers)-🡪Pods-🡪 container**

First, create your pod; In order to get inside your pods, you need to be inside your cluster. To get inside your cluster, you need to get its IP address and then ssh into it using the IP address

Minikube is a cluster.

**Kubernetes basic commands**

1. Create Pod #kubectl run lili-pod --image=nginx

#kubectl get pods --🡪 to see your pods

1. Get inside your cluster:

#minikube ip 🡪 to get the IP of the nodes

#ssh docker@IPadress

Default password:tcuser

Once inside the cluster/minikube;

1. Grep the pod to see its container ID:

#docker ps |grep lili-pod

1. Attach your container;

#docker exec -it (dockerID) sh

1. Get the IP address of the container i.e;

#hostname -i

#curl IpAddress to the content of your container

#Kubectl cluster-info 🡪to see information of the cluster

#kubectl get nodes -🡪 to see your nodes

#kubectl delete pod lili-pod

**NAMESPACES**

Namespaces are a way to isolate, group and organize resources within a Kubernetes cluster. It gives you a path to your objects/resources.

#Kubectl get namespace -- will give you the following output:

-*defaul*t-used by user apps by default, until there are other custom namespaces

-*kube-public* -used by public Kubernetes resources; -not recommended to be used by cluster users

-Kube-system -used by Kubernetes control plane, and must not be used by cluster users

**Common commands with namespaces:**

* #kubectl get namespaces
* #kubectl create namespace mynamespace
* #kubectl get pods --all-namespaces (List all pods with status from all namespaces.)
* #kubectl get pods –namespace=GiveNameofnamespace 🡪 to get a list of pods in the namespace
* #kubectl get pod -o wide
* #kubectl describe namespace <namespace>
* #kubectl config view --minify | grep namespace (This command will ensure that you set the namespace correctly for your current context.)
* #kubectl delete namespace namespace1

**DEPLOYMENT**

Deployments are useful for:

-Scaling the number of replica pods

-Enabling the rollout of updated code in a controlled manner

-Rolling back to an earlier deployed version if necessary

# To create a deployment

Once you create a od, you can create a deployment for it by running:

#kubectl create deployment deploymentname –image=docker image name

e.g to create a deployment called lily-deploy with Nginx image

#kubect create deployment lily-deploy –image-nginx

You can now scale this deployment to 4 pods by running

#kubectl scale deployment lily-deploy –replicas=5

**SERVICE**

A service is a method for exposing a network application that is running as one or more pods in your cluster. i.e once you created a deployment, you use a service to make the sets of pods available on the network so that clients can interact with it.

To interact with it internally, get the cluster Id. For external interaction, you create services such as Nodeport & LoadBalancer.

1. #kubectl expose deployment (deployment-name) port=Portnumber to expose to --target-port=docker image port

*e.g Expose deployment (lily-deploy) on port 7575, get the cluster IP to use for accessing your deployment internally (remember lily-deploy was created in Nginx image which listens in port 80*)

#kubectl expose deployment lily-deploy --port=7575 --target-port=80

1. Get the cluster IP by running #Kubectl get service (or) #kubectl get svc
2. Go inside your cluster (minikube) by running #miniube ip
3. Curl cluster Ip :port number

**Steps to creating service and exposing deployment**

1. Create a pod, #kubectl run PodName –image=docker image

i.e #kubectl lily-pod --image=nginx

1. Create a deployment from the pod

#kubectl create deployment lily-deploy --image=nginx

1. Scale the deployment

#kubectl scale deployment lily-deploy --replicas=5

1. Get the IP #kubectl get pod -o wide
2. Create a service to expose to a specific port no (7979)

#kubectl expose deployment lily-deploy --port=7979 --target-port=80

1. Kubectl get svc
2. Log into your cluster (minikube ip)
3. Curl clusterIp:Port Number
4. Delete this service to create a Nodeport #kubectl delete svc lily-deploy

**NodePort**

You cannot have 2 services running at the same time, to create another service you must delete the first. To create a NodePort

#kubectl expose deployment lily-deploy --type=NodePort –port=7272 --target-port=80

#kubectl get svc to get the automatically allocated port i.e 7272:**31289**. Copy the send part(31289)

#get your cluster IP i.e, minikube ip

Then go to your web browser and paster your cluster Ip: the allocated port(31289)

Alternative, to avoid the last two steps. Just type

#minikube service lily-deploy –url, get the URL then go to your web browser and paste to see results

(Delete this service to create LoadBalancer)

#kubectl expose deployment lily-deploy --type=LoadBalancer --port=6262 --target-port=80

(Follow same steps as above)