**dDocker:**

To add user to docker group to run docker commands:

sudo gpasswd -a $USER docker

sudo usermod -aG docker $USER

ls -ltr | grep docker

chmod 755 docker

cat /etc/group

List of images on local:

docker images

List of all running or terminated containers:-p

docker ps -a

To download an image from docker hub:

docker pull <image-name>

To run an image available on local:

docker run --name <name\_of\_container> <image\_name>

To execute command on docker container:

docker exec <container\_name> <command>

Remove Containers

docker rm hello\_word\_sample

Remove Images

docker rmi hello-world

Remove all the images

docker rmi $(docker images -q -f dangling=true)

List ports

docker port hello\_word\_sample

List Processes

docker top hello\_world\_sample

Execute command

docker exec hello\_word \_sample ls

Docker log

docker logs --[details/timestamp/until] CONTAINER

Docker login to container

docker run -it ubuntu bash

-i means connect to container stdin, t means connect pseudo terminal

-e means set an environment variable. Or --env-file to provide file of properties.

-d means detached mode.

--rm means remove image when container exits.

Docker port mapping to host

docker run -p <host\_port>:<container\_port> -d <image>

Example of Jenkins container:

docker run -p 80:8080 -d jenkins

docker exec friendly\_wiles cat /var/jenkins\_home/secrets/initialAdminPassword

Kitematic

GUI for managing of the installation of images in the containers and exposing the posts and other configuration.

At any time, want to perform operation on a specific tag of image use:

<image\_name>:<tag\_name>

docker stats <container\_name>

docker rename <old\_container\_name> <new\_container\_name>

Example:

docker build -t nodejs /opt/sample-website

docker tag nodejs avinashkolla/nodejs:v1

docker push avinashkolla/nodejs:v1

* When you build them, using docker build -t <hub-user>/<repo-name>[:<tag>]
* By re-tagging an existing local image docker tag <existing-image> <hub-user>/<repo-name>[:<tag>]
* By using docker commit <existing-container> <hub-user>/<repo-name>[:<tag>] to commit changes

It’s also recommended that you limit the number of labels you use. Every time you use a label, it will add a layer to the image, thus increasing the size of the image. Using too many labels can cause the image to become inefficient as well. You can view the containers’ labels with:

docker inspect <image\_id>

Reviewing DockerFile in depth

1. LABEL

It’s also recommended that you limit the number of labels you use. Every time you use a label, it will add a layer to the image, thus increasing the size of the image. Using too many labels can cause the image to become inefficient as well. You can view the containers’ labels with:

docker inspect <image\_id>

1. ADD or COPY

You can use the ADD instruction and specify a URL straight to a file; it will be downloaded when the container is built. The ADD instruction will also unpack or untar a file when added. The COPY instruction is the same as the ADD instruction, but without the URL handling or the unpacking/untarring of files

1. ENTRYPOINT

You can also use the ENTRYPOINT instruction instead. The benefit of using ENTRYPOINT over CMD is that you can use them in conjunction with each other. But be sure to use a command that keeps the container alive:

FROM ubuntu:latest

ENTRYPOINT ["ps", "-au"]

CMD ["-x"]

1. ENTRYPOINT with CMD
2. USER

The USER instruction lets you specify the username to be used when a command is run. The USER instruction can be used on the RUN instruction, the CMD instruction, or the ENTRYPOINT instruction in the Dockerfile.

1. WORKDIR

The WORKDIR command sets the working directory for the same set of instructions that the USER instruction can use (RUN, CMD, and ENTRYPOINT). It will allow you to use the CMD and ADD instructions as well.

1. ONBUILD

The ONBUILD instruction lets you stash a set of commands that will be used when the image is used again as a base image for a container.

For example, if you want to give an image to developers and they all have a different code they want to test, you can use the ONBUILD instruction to lay the groundwork ahead of the fact of needing the actual code. Then, the developer will simply add their code in the directory you tell them and, when they run a new docker build command, it will add their code to the running image. The ONBUILD instruction can be used in conjunction with the ADD and RUN instructions:

ONBUILD ADD

ONBUILD RUN

Kubernetes uses similar to blue-green deployment.

When a docker image was updated with the new code in production, Kubernetes manages it by dropping on container of older version and creating new version of image, this happens till all old version was removed and new was updated.

**Cgroups:**

A cgroup limits an application to a specific set of resources. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints.

Docker Engine uses namespaces such as the following on Linux:

* **The pid namespace:** Process isolation (PID: Process ID).
* **The net namespace:** Managing network interfaces (NET: Networking).
* **The ipc namespace:** Managing access to IPC resources (IPC: InterProcess Communication).
* **The mnt namespace:** Managing filesystem mount points (MNT: Mount).
* **The uts namespace:** Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

**Container format**

Docker Engine combines the namespaces, control groups, and UnionFS into a wrapper called a container format. The default container format is libcontainer.

**Union file systems**

Union file systems, or UnionFS, are file systems that operate by creating layers, making them very lightweight and fast. Docker Engine uses UnionFS to provide the building blocks for containers.

By default, docker runs as socket. If this can be made to run on port, then docker can be accessed remotely.

unix:///var/run/docker.sock

as per docker file, 2376 port is used for docker.

Below are the commands for docker to run on port:

mkdir -p /etc/systemd/system/docker.service.d

printf “[Service]\nExecStart=\nExecStart=/usr/bin/docker -H fd:// -H tcp://0.0.0.0:2376\n” >> /etc/system/system/docker.service.d/docker.conf

systemctl daemon-reload

systemctl restart docker

**Commands vs Entrypoints**

**Container Network Interface**

**Kubernetes:**

The Kubernetes Control Plane makes the cluster’s current state match the desired state via the **Pod Lifecycle Event Generator** ([PLEG](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/pod-lifecycle-event-generator.md)). To do so, Kubernetes performs a variety of tasks automatically – such as starting or restarting containers, scaling the number of replicas of a given application, and more. The **Kubernetes Control Plane** consists of a collection of processes running on your cluster:

* The **Kubernetes Master** is a collection of three processes that run on a single node in your cluster, which is designated as the master node. Those processes are: **[kube-apiserver](https://kubernetes.io/docs/admin/kube-apiserver/), [kube-controller-manager](https://kubernetes.io/docs/admin/kube-controller-manager/) and [kube-scheduler](https://kubernetes.io/docs/admin/kube-scheduler/).**
* Each individual non-master node in your cluster runs two processes:
  + [**kubelet**](https://kubernetes.io/docs/admin/kubelet/)**,** which communicates with the Kubernetes Master.
  + [**kube-proxy**](https://kubernetes.io/docs/admin/kube-proxy/)**,** a network proxy which reflects Kubernetes networking services on each node.

The basic Kubernetes objects include:

* [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/)
* [Service](https://kubernetes.io/docs/concepts/services-networking/service/)
* [Volume](https://kubernetes.io/docs/concepts/storage/volumes/)
* [Namespace](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/)

Kubernetes also contains higher-level abstractions that rely on [Controllers](https://kubernetes.io/docs/concepts/architecture/controller/) to build upon the basic objects, and provide additional functionality and convenience features. These include:

* [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
* [DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)
* [StatefulSet](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/)
* [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/)
* [Job](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/)

**kube-apiserver**

The API server is the front end for the Kubernetes control plane. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

**etcd**

Consistent and highly available key value store used as Kubernetes’ backing store for all cluster data. If your Kubernetes cluster uses etcd as its backing store, make sure you have a [backup](https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/#backing-up-an-etcd-cluster) plan for those data.

**kube-scheduler**

Control Plane component that watches for newly created pods with no assigned node and selects a node for them to run on.

Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

**kube-controller-manager**

Control Plane component that runs [controller](https://kubernetes.io/docs/concepts/architecture/controller/) processes.

Logically, each [controller](https://kubernetes.io/docs/concepts/architecture/controller/) is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

These controllers include:

* **Node Controller:** Responsible for noticing and responding when nodes go down.
* **Replication Controller**: Responsible for maintaining the correct number of pods for every replication controller object in the system.
* **Endpoints Controller**: Populates the Endpoints object (that is, joins Services & Pods).
* **Service Account & Token Controllers**: Create default accounts and API access tokens for new namespaces

**kubelet**

An agent that runs on each node in the cluster. It makes sure that containers are running in a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn’t manage containers which were not created by Kubernetes.

**kube-proxy**

[kube-proxy](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/) is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes [Service](https://kubernetes.io/docs/concepts/services-networking/service/) concept. kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster. kube-proxy uses the operating system packet filtering layer if there is one and it’s available. Otherwise, kube-proxy forwards the traffic itself.

**Container Runtime**

The container runtime is the software that is responsible for running containers. Kubernetes supports several container runtimes: [Docker](http://www.docker.com/), [containerd](https://containerd.io/" \t "_blank), [cri-o](https://cri-o.io/), [rktlet](https://github.com/kubernetes-incubator/rktlet" \t "_blank).

**Master** -> manages cluster, storing and managing containers.

**Etcd** -> the key-value storage which stores all the details on.

**Kube-scheduler** -> identifies the right node based on the container resource acquirement, worker nodes capacity, scheduling container.

**Node-controller** -> controls and manages nodes in the cluster.

**Replication-controller** -> always make sure to achieve desired state.

**Kube-control-manager** -> manages, controls replication-controller and node-controller.

**Kube-apiserver** -> primary management component, orchestrating entire architecture, like kube-control-manager, kube-scheduler, etcd. Also, acts as interface between the external client and kube-cluster. Fetches status from worker node through kubelet.

**Kubelet** -> agent that runs on worker node, that interacts with the master nodes.

**Kubeproxy** -> communication between worker nodes/containers is enabled using this.

* Should be used when the values are list in yaml files.

And two spaces indentation.

**Taints Toleration vs Node Affinity**

**Replicasets**

**Namespace**

amazon-linux-extras install docker

docker

yum install jenkins

sudo yum remove java-1.7.0-openjdk

sudo yum install java-1.8.0

java -version

sudo rpm --import https://pkg.jenkins.io/redhat/jenkins.io.key

sudo yum install jenkins -y

sudo systemctl start jenkins

sudo systemctl enable jenkins

sudo systemctl status jenkins

sudo cat /var/lib/jenkins/secrets/initialAdminPassword

systemctl start docker

systemctl enable docker

yum install git

docker

git clone https://github.com/avinashkolla/sample-website.git

cd sample-website/

docker ps -a

docker images

docker build -t nodejsavi /opt/sample-website/

docker tag nodejsavi avinashkolla/nodejsavi:v1

docker login

docker push avinashkolla/nodejsavi:v1

docker pull avinashkolla/nodejs:v1

git config --global --edit

To change user of Jenkins applications and other parameters: /etc/sysconfig/jenkins