* docker container run –p 80:80 httpd:2.4
* docker container ls
* docker container exec –it <container\_name> <command>
* docker container exec –it <container\_name> <shell>
* Dockerfile

FROM httpd:2.4

EXPOSE 80

RUN command1 && command2

LABEL maintainter=”[moby-dock@example.com](mailto:moby-dock@example.com)”

* docker image build –tag myhttpd:1.0 . 🡪 “.” For current directory as Dockerfile location.
* docker container cp <file> <containername>:<targetlocation>
* Dockerfile

COPY <file> <targetLocation>

* docker run container –p 80:80 –detach <image>:<version> 🡪 detached mode.
* docker run -d -p 80:80 -v /my-files:/usr/local/apache2/htdocs web-server:1.1 🡪 creating the volume.

Installation:

* yum remove docker docker-common container-selinux docker-selinux docker-engine docker-engine-selinux (to remove previous installations)
* yum install yum-utils
* yum install -y yum-utils device-mapper-persistent-data lvm2
* yum-config-manager --add-repo <https://download.docker.com/linux/centos/docker-ce.repo>
* yum-config-manager --enable docker-ce-edge
* yum makecache fast
* yum install docker-ce
* service docker status
* touch /etc/docker/daemon.json
* vi /etc/docker/daemon.json

{

"storage-driver": "devicemapper"

}

* systemctl start docker
* service docker status
* docker –version
* chkconfig docker on

An **image** is a lightweight, stand-alone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files. The file system and configuration of our application which are used to create containers

A **container** is a runtime instance of an image—what the image becomes in memory when actually executed. It runs completely isolated from the host environment by default, only accessing host files and ports if configured to do so. Containers run apps natively on the host machine’s kernel. A container includes an application and all of its dependencies. It shares the kernel with other containers, and runs as an isolated process in user space on the host OS.

**VM Vs Container** : Virtual machines run guest operating systems. This is resource intensive, and the resulting disk image and application state is an entanglement of OS settings, system-installed dependencies, OS security patches, and other easy-to-lose, hard-to-replicate ephemera. Containers can share a single kernel, and the only information that needs to be in a container image is the executable and its package dependencies, which never need to be installed on the host system.

**Docker daemon** - The background service running on the host that manages building, running and distributing Docker containers.

**Docker client** - The command line tool that allows the user to interact with the Docker daemon.

**Docker Store** - Store is a [registry](https://store.docker.com/) of Docker images. You can think of the registry as a directory of all available Docker images.

**Note**: Accessing the name of the host when inside a container retrieves the container ID, which is like the process ID for a running executable.

A **registry** is a collection of repositories, and a **repository** is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The docker CLI uses Docker’s public registry by default.

**Docker Trusted Registry** : <https://docs.docker.com/datacenter/dtr/2.2/guides/>

**Hierarchy** : Bottom of the hierarchy of an app built in the docker way is a container. Above this level is a service, which defines how containers behave in production. Finally, at the top level is the stack, defining the interactions of all the services.

Container🡪 Services 🡪 Stack

docker build -t friendlyname . # Create image using this directory's Dockerfile

docker run -p 4000:80 friendlyname # Run "friendlyname" mapping port 4000 to 80

docker run -d -p 4000:80 friendlyname # Same thing, but in detached mode

docker ps # See a list of all running containers

docker stop <hash> # Gracefully stop the specified container

docker ps -a # See a list of all containers, even the ones not running

docker kill <hash> # Force shutdown of the specified container

docker rm <hash> # Remove the specified container from this machine

docker rm $(docker ps -a -q) # Remove all containers from this machine

docker images -a # Show all images on this machine

docker rmi <imagename> # Remove the specified image from this machine

docker rmi $(docker images -q) # Remove all images from this machine

docker login # Log in this CLI session using your Docker credentials

docker tag <image> username/repository:tag # Tag <image> for upload to registry

docker push username/repository:tag # Upload tagged image to registry

docker run username/repository:tag # Run image from a registry

**Docker Compose**

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your application’s services. Then, using a single command, you create and start all the services from your configuration.

Using Compose is basically a three-step process.

* Define your app’s environment with a *Dockerfile* so it can be reproduced anywhere.
* Define the services that make up your app in *docker-compose.yml* (default file name) so they can be run together in an isolated environment.
* Lastly, run *docker-compose up* and Compose will start and run your entire app.

Features:

* [Multiple isolated environments on a single host](https://docs.docker.com/compose/overview/#Multiple-isolated-environments-on-a-single-host) : Compose uses a project name to isolate environments from each other. he default project name is the basename of the project directory. You can set a custom project name by using the [-p command line option](https://docs.docker.com/compose/reference/overview/) or the [COMPOSE\_PROJECT\_NAME environment variable](https://docs.docker.com/compose/reference/envvars/#compose-project-name).
* [Preserve volume data when containers are created](https://docs.docker.com/compose/overview/#preserve-volume-data-when-containers-are-created) : Compose preserves all volumes used by your services. When docker-compose up runs, if it finds any containers from previous runs, it copies the volumes from the old container to the new container. This process ensures that any data you’ve created in volumes isn’t lost.
* [Only recreate containers that have changed](https://docs.docker.com/compose/overview/#only-recreate-containers-that-have-changed) : Compose caches the configuration used to create a container. When you restart a service that has not changed, Compose re-uses the existing containers.
* [Variables and moving a composition between environments](https://docs.docker.com/compose/overview/#variables-and-moving-a-composition-between-environments) : You can use these variables to customize your composition for different environments, or different users.

Installation

curl -L [https://github.com/docker/compose/releases/download/1.14.0/docker-compose-`uname](https://github.com/docker/compose/releases/download/1.14.0/docker-compose-%60uname) -s`-`uname -m` > /usr/local/bin/docker-compose

chmod +x /usr/local/bin/docker-compose

docker-compose –-version

Executing:

1. Create the “docker-compose.yml” (or any other name) with the service config.

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: username/repository:tag

deploy:

replicas: 5

resources:

limits:

cpus: "0.1"

memory: 50M

restart\_policy:

condition: on-failure

ports:

- "80:80"

networks:

- webnet

networks:

webnet:

execute the below commands.

-bash-4.2# *docker stack deploy -c docker-compose.yml getstartedlab*

This node is not a swarm manager. Use "docker swarm init" or "docker swarm join" to connect this node to swarm and try again.

-bash-4.2# *docker swarm init*

Error response from daemon: could not choose an IP address to advertise since this system has multiple addresses on different interfaces (10.0.2.15 on enp0s3 and 192.168.99.100 on enp0s8) - specify one with --advertise-addr

-bash-4.2# *docker swarm init --advertise-addr enp0s8*

Swarm initialized: current node (lddvs8kobukjawmb2vjbk0n1t) is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join \

--token SWMTKN-1-4hrsmbtd0qa79dvl1yxp7660mx45nbyzm0hy54l95l1iwd36ao-7kowhl30vsvmumvmdpselx7dj \

192.168.99.100:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

-bash-4.2# *docker stack deploy -c docker-compose.yml getstartedlab*

Creating network getstartedlab\_webnet

Creating service getstartedlab\_web

-bash-4.2# *docker ps*

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

4db6cf5d8910 jitheshktv/get-started:part1 "python app.py" 10 seconds ago Up 5 seconds 80/tcp getstartedlab\_web.2.96qum9uaa3o2bh1copr9gkok8

1af224cc2633 jitheshktv/get-started:part1 "python app.py" 10 seconds ago Up 5 seconds 80/tcp getstartedlab\_web.5.op3asoswcpqxw7dyc3ulbdctn

78057ff4eb5b jitheshktv/get-started:part1 "python app.py" 10 seconds ago Up 5 seconds 80/tcp getstartedlab\_web.3.adh4pv0vyr7qwe1ptsqgmybet

62a44a289b9d jitheshktv/get-started:part1 "python app.py" 10 seconds ago Up 6 seconds 80/tcp getstartedlab\_web.1.8kff9lzx1t8xc060h0zqa0qpd

7df76c88cdb8 jitheshktv/get-started:part1 "python app.py" 10 seconds ago Up 8 seconds 80/tcp getstartedlab\_web.4.1apojev8eub3h6i65qlmaryvr

-bash-4.2*# docker stop 4db6cf5d8910*

4db6cf5d8910

-bash-4.2#

-bash-4.2# *docker ps*

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

d281c13f92cb jitheshktv/get-started:part1 "python app.py" 6 seconds ago Up Less than a second 80/tcp getstartedlab\_web.2.xp7pg8xdklpf5nkzblq6s2oe5 🡪 started new on failure

1af224cc2633 jitheshktv/get-started:part1 "python app.py" 39 seconds ago Up 34 seconds 80/tcp getstartedlab\_web.5.op3asoswcpqxw7dyc3ulbdctn

78057ff4eb5b jitheshktv/get-started:part1 "python app.py" 39 seconds ago Up 34 seconds 80/tcp getstartedlab\_web.3.adh4pv0vyr7qwe1ptsqgmybet

62a44a289b9d jitheshktv/get-started:part1 "python app.py" 39 seconds ago Up 35 seconds 80/tcp getstartedlab\_web.1.8kff9lzx1t8xc060h0zqa0qpd

7df76c88cdb8 jitheshktv/get-started:part1 "python app.py" 39 seconds ago Up 37 seconds 80/tcp getstartedlab\_web.4.1apojev8eub3h6i65qlmaryvr

-bash-4.2#

-bash-4.2# *docker stack ps getstartedlab*

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

8kff9lzx1t8x getstartedlab\_web.1 jitheshktv/get-started:part1 localhost.localdomain Running Running 4 minutes ago

xp7pg8xdklpf getstartedlab\_web.2 jitheshktv/get-started:part1 localhost.localdomain Running Running 4 minutes ago

96qum9uaa3o2 \\_ getstartedlab\_web.2 jitheshktv/get-started:part1 localhost.localdomain Shutdown Failed 4 minutes ago "task: non-zero exit (137)"

adh4pv0vyr7q getstartedlab\_web.3 jitheshktv/get-started:part1 localhost.localdomain Running Running 4 minutes ago

1apojev8eub3 getstartedlab\_web.4 jitheshktv/get-started:part1 localhost.localdomain Running Running 4 minutes ago

op3asoswcpqx getstartedlab\_web.5 jitheshktv/get-started:part1 localhost.localdomain Running Running 4 minutes ago

-bash-4.2#

docker stack ls # List all running applications on this Docker host

docker stack deploy -c <composefile> <appname> # Run the specified Compose file

docker stack services <appname> # List the services associated with an app

docker stack ps <appname> # List the running containers associated with an app

docker stack rm <appname> # Tear down an application

docker node ls #to see the node in the swarm

-bash-4.2# ***docker swarm leave***

Error response from daemon: You are attempting to leave the swarm on a node that is participating as a manager. Removing the last manager erases all current state of the swarm. Use `--force` to ignore this message.

-bash-4.2# ***docker swarm leave --force***

Node left the swarm.

-bash-4.2#

**Docker Machine**

Docker Machine is a tool that lets you install Docker Engine on virtual hosts, and manage the hosts with docker-machine commands. Docker Machine enables you to provision multiple remote Docker hosts on various flavors of Linux.

Docker Machine has these two broad use cases.

* I have an older desktop system and want to run Docker on Mac or Windows
* I want to provision Docker hosts on remote systems : Docker Engine runs natively on Linux systems. If you have a Linux box as your primary system, and want to run docker commands, all you need to do is download and install Docker Engine. However, if you want an efficient way to provision multiple Docker hosts on a network, in the cloud or even locally, you need Docker Machine.

Difference between Docker Engine and Docker Machine

**Docker Engine** is the client-server application made up of the Docker daemon, a REST API that specifies interfaces for interacting with the daemon, and a command line interface (CLI) client that talks to the daemon (through the REST API wrapper). Docker Engine accepts docker commands from the CLI.

**Docker Machine** is a tool for provisioning and managing your Dockerized hosts (hosts with Docker Engine on them). Typically, you install Docker Machine on your local system. Docker Machine has its own command line client docker-machine and the Docker Engine client, docker. You can use Machine to install Docker Engine on one or more virtual systems (local or remote)

Installation

curl -L https://github.com/docker/machine/releases/download/v0.12.0/docker-machine-`uname -s`-`uname -m` > /tmp/docker-machine

chmod +x /tmp/docker-machine

mv /tmp/docker-machine /usr/local/bin/

docker-machine –version

swarm cluster

A swarm is a group of machines that are running Docker and joined into a cluster. After that has happened, you continue to run the Docker commands you’re used to, but now they are executed on a cluster by a **swarm manager**. The machines in a swarm can be physical or virtual. After joining a swarm, they are referred to as **nodes**.

Swarm managers can use several strategies to run containers, such as “emptiest node” – which fills the least utilized machines with containers. Or “global”, which ensures that each machine gets exactly one instance of the specified container. You instruct the swarm manager to use these strategies in the Compose file.

Swarm managers are the only machines in a swarm that can execute your commands, or authorize other machines to join the swarm as **workers**. Workers are just there to provide capacity and do not have the authority to tell any other machine what it can and cannot do.

run *docker swarm init* to enable swarm mode and make your current machine a swarm manager, then run *docker swarm join* on other machines to have them join the swarm as workers.

$ docker-machine create --driver virtualbox myvm1

$ docker-machine create --driver virtualbox myvm2

$ *docker-machine env docker-vm1* and then *eval $(docker-machine env docker-vm1) #* to configure the shell for the docker client to connect to the docker engine.

$ docker-machine ssh myvm1 "docker swarm init"

$ docker-machine ssh myvm1 "docker swarm init --advertise-addr 192.168.99.100:2377" # if you get error needing to use advertise address.

$ docker-machine ssh myvm2 “docker swarm join --token <token> <ip>:<port>”

$ docker machine ssh myvm1 # this will create an ssh session to the VM. Type exit to exit the session.

$ docker machine ssh myvm1 “docker node ls”

$ docker-machine scp docker-compose.yml myvm1:~

$ docker-machine ssh myvm1 "docker stack deploy -c docker-compose.yml getstartedlab"

$ docker-machine ssh myvm1 "docker stack ps getstartedlab"

You can access your app from the IP address of either myvm1 or myvm2. The network you created is shared between them and load-balancing. The reason both IP addresses work is that nodes in a swarm participate in an ingress **routing mesh**. This ensures that a service deployed at a certain port within your swarm always has that port reserved to itself, no matter what node is actually running the container.

Note : In order to use the ingress network in the swarm, you need to have the following ports open between the swarm nodes before you enable swarm mode:

* Port **7946** TCP/UDP for container network discovery.
* Port **4789** UDP for the container ingress network.

$ docker-machine ssh myvm1 "docker stack rm getstartedlab"

$ docker-machine ssh myvm2 "docker swarm leave"

$ docker-machine ssh myvm1 "docker swarm leave --force"

$ docker-machine stop aws-sandbox

$ docker-machine rm aws-sandbox

For AWS : docker-machine create --driver amazonec2 --amazonec2-access-key AKI\*\*\*\*\*\*\* --amazonec2-secret-key 8T93C\*\*\*\*\*\*\* --amazonec2-region us-west-1

aws-sandbox

docker-machine create --driver virtualbox myvm1 # Create a VM (Mac, Win7, Linux)

docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" myvm1 # Win10

docker-machine env myvm1 # View basic information about your node

docker-machine ssh myvm1 "docker node ls" # List the nodes in your swarm

docker-machine ssh myvm1 "docker node inspect <node ID>" # Inspect a node

docker-machine ssh myvm1 "docker swarm join-token -q worker" # View join token

docker-machine ssh myvm1 # Open an SSH session with the VM; type "exit" to end

docker-machine ssh myvm2 "docker swarm leave" # Make the worker leave the swarm

docker-machine ssh myvm1 "docker swarm leave -f" # Make master leave, kill swarm

docker-machine start myvm1 # Start a VM that is currently not running

docker-machine stop $(docker-machine ls -q) # Stop all running VMs

docker-machine rm $(docker-machine ls -q) # Delete all VMs and their disk images

docker-machine scp docker-compose.yml myvm1:~ # Copy file to node's home dir

docker-machine ssh myvm1 "docker stack deploy -c <file> <app>" # Deploy an app

$ docker container run --name static-site -e AUTHOR="Your Name" -d -P seqvence/static-site

* -d will create a container with the process detached from our terminal
* -P will publish all the exposed container ports to random ports on the Docker host
* -e is how you pass environment variables to the container
* --name allows you to specify a container name
* AUTHOR is the environment variable name and Your Name is the value that you can pass

$ docker container port <containerId> # will show all the ports used by the container.

$ docker container run --name static-site-2 -e AUTHOR="Your Name" -d -p 8888:80 seqvence/static-site # publish the specific port on guest to a specific port on host.

$ docker search <imageName> # to search for an image in the registry.

If you do not specify the version number of the image then, the Docker client will default to a version named ***latest***.

An important distinction with regard to images is between *base images* and *child images*.

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

Another key concept is the idea of *official images* and *user images*.

* **Official images** are Docker sanctioned images. These are not prefixed by an organization or user name.
* **User images** are images created and shared by users like you. They build on base images and add additional functionality. Typically these are formatted as user/image-name. The user value in the image name is your Docker Cloud user or organization name.

**Dockerfile**

### Dockerfile commands

* **FROM** starts the Dockerfile. It is a requirement that the Dockerfile must start with the FROM command. Images are created in layers, which means you can use another image as the base image for your own. The FROM command defines your base layer.
* **RUN** is used to build up the Image you’re creating. For each RUN command, Docker will run the command then create a new layer of the image.
* **COPY** copies local files into the container.
* **CMD** defines the commands that will run on the Image at start-up. Unlike a RUN, this does not create a new layer for the Image, but simply runs the command. There can only be one CMD per a Dockerfile/Image. If you need to run multiple commands, the best way to do that is to have the CMD run a script.
* **EXPOSE** opens ports in your image to allow communication to the outside world when it runs in a container.

[**https://docs.docker.com/engine/reference/builder/**](https://docs.docker.com/engine/reference/builder/)

[**https://docs.docker.com/engine/userguide/eng-image/dockerfile\_best-practices/**](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/)

$ docker commit <ContainerID> : to commit the changes made in a container in an interactive mode and create a new image out of it.

$ docker image tag <imageId> <new\_name>[:version]

$ docker build -f <path to docker file> -t <name>[:<version>] <PATH>

ENTRYPOINT & CMD : The command run by default is the concatenation of ENTRYPOINT and CMD. Using this, we can parameterize the input. ENTRYPOINT with the actual command and CMD as the parameter.

Reference : portainer