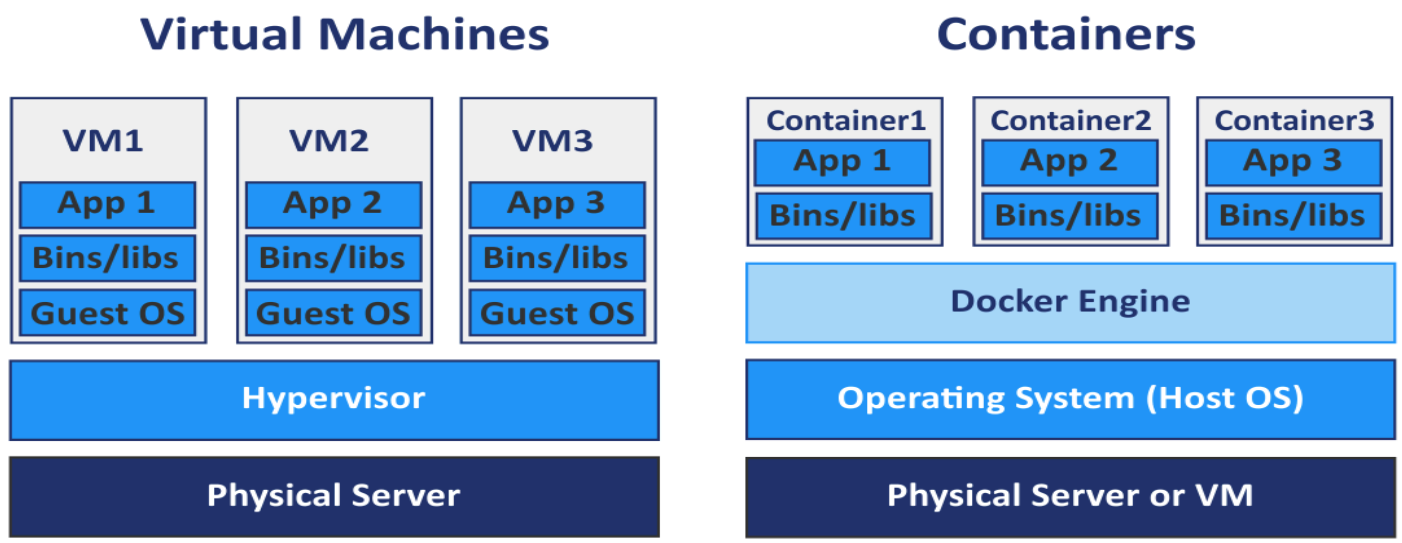
DOCKER COMPLETE

Virtualization is the process of creating a software based virtual version of servers, apps, storage etc.
Virtualization is the process where one system is splits into many different sections, which was done by hypervisor.



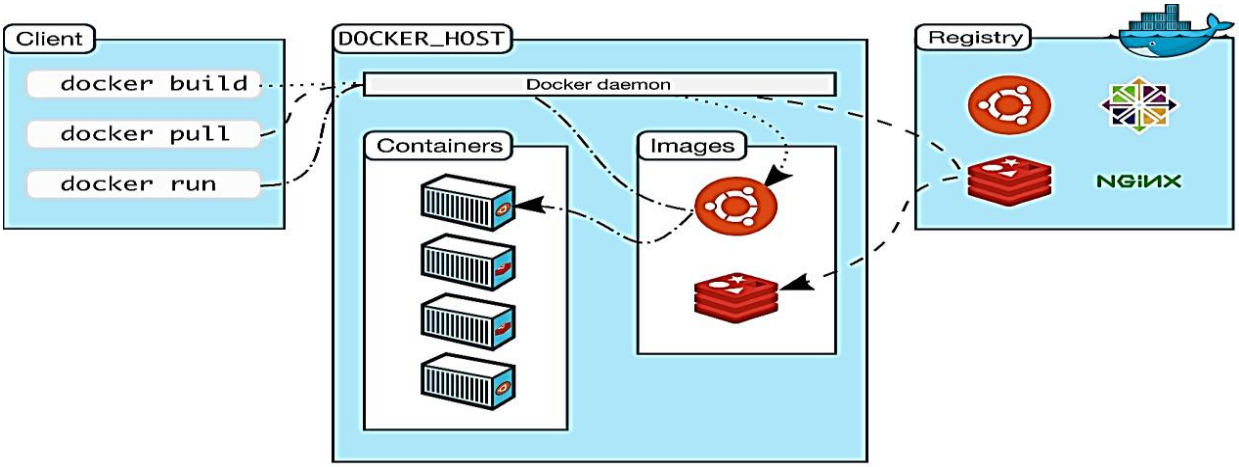
Diff Virtualization and Containerization

- Containers provide an isolated environment for running the application. The entire user space is explicitly dedicated to the application.
- Any changes made inside the container is never reflected on the host or even other containers running on the same host.
- Containers are an abstraction of the application layer. Each container is a different application.
- Virtualization, hypervisors provide an entire virtual machine to the guest (including Kernel).
- Virtual machines are an abstraction of the hardware layer. Each VM is a physical machine.

- Docker is a platform for developers and sysadmins to build, run, and share applications with containers.
- Containerization is increasingly popular because containers are: Flexible, Lightweight, Portable, loosely coupled, Scalable, Secure, Short boot-up process.
- Container is nothing but a running process, which is encapsulated in order to keep it isolated from the host and from other containers.
- Each container interacts with its own private filesystem which is provided by a Docker image. An image includes everything needed to run an application.
- A container runs natively on Linux and shares the kernel of the host machine with other containers.

DOCKER ARCHITECTURE:

Docker uses a client-server architecture. The Docker client talks to the Docker daemon. Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.



The Docker daemon:
The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.
The Docker client:
The Docker client (docker) is the primary way that many Docker users interact with Docker. The Docker client can communicate with more than one daemon.
Docker registries:
A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use

CONTAINERS:
A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it. Container is relatively well isolated from other containers and its host machine.
DOCKER IMAGE:

- An *image* is a read-only template with instructions for creating a Docker container.
- An image is a text file with a set of pre-written commands, usually called as a Docker file
- Docker Images are made up of multiple layers which are read-only filesystem
- A layer is created for each instruction in a Docker file and placed on top of the previous layer
- When an image is turned into a container the Docker engine takes the image and adds a read-write filesystem on top.

Docker Volume VS Bind Mount:

Volumes: (-v) are stored in a part of the host filesystem which is managed by Docker **(/var/lib/docker/volumes/** on Linux).

- when you use a volume, a new directory is created within Docker's storage directory on the host machine
- volumes stored in docker's "private" storage area
- volumes can be created by the docker engine when the container starts.

Bind mounts: (--mount)

- When you use a bind mount, a file or directory on the host machine is mounted into a container.
- Bind Mount can be stored anywhere
- Bind Mount have to exist prior to starting the docker container

Docker works on two things:
1. **Name Space** ---> it is a linux concept which create an isolated user space
2. **Control Groups (c groups)** ---> control the process
*Here docker is secure if the host os *is secured. As it depends on Host OS

Docker Namespace:

Namespace adds a layer of isolation in containers. Docker provides various namespaces in order to stay portable and not affect the underlying host system. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.
Few namespace types supported by Docker – PID, Mount, IPC, User, Network.
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).

Docker machine: is a tool that lets you install Docker Engine on virtual hosts. Docker machine also lets you provision Docker Swarm Clusters.

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

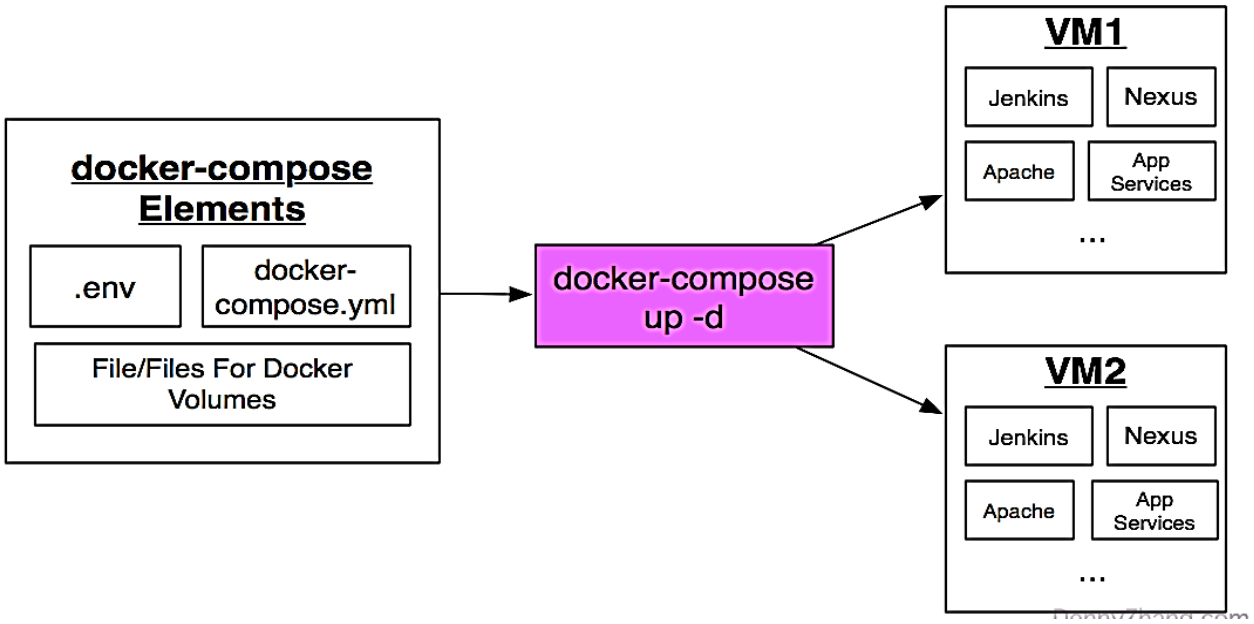
Docker Pull Working Process:
First docker daemon look for an image in the local repo. If not found then it looks in docker registry. If found it pull the images and stores in local repo.

Docker Compose:

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. Docker Compose to create separate containers, host them and get them to communicate with each other.

Compose is basically a three-step process:

1. Define APP environment in Dockerfile.
2. Services required for app in docker-compose.yml .
3. Run docker-compose up.



Update containers in Compose:
If you make a configuration change to a service and run docker-compose up to update it, the old container is removed and the new one joins the network under a different IP address but the same name.
Links:
Links allow you to define extra aliases by which a service is reachable from another service. They are not required to enable services to communicate - by default, any service can reach any other service at that service's name.

Need of compose file for production:

- Removing any volume bindings for application code, so that code stays inside the container and can't be changed from outside
- Binding to different ports on the host
- Specifying a restart policy like **restart: always** to avoid downtime

```
$ docker-compose -f docker-compose.yml -f production.yml up -d
```

"**-f** " is used to compose multiple configuration files.

NOTE: You can control the order of service startup and shutdown with the depends_on option. Compose always starts and stops containers in dependency order, where dependencies are determined by depends_on, links, volumes_from, and network_mode:

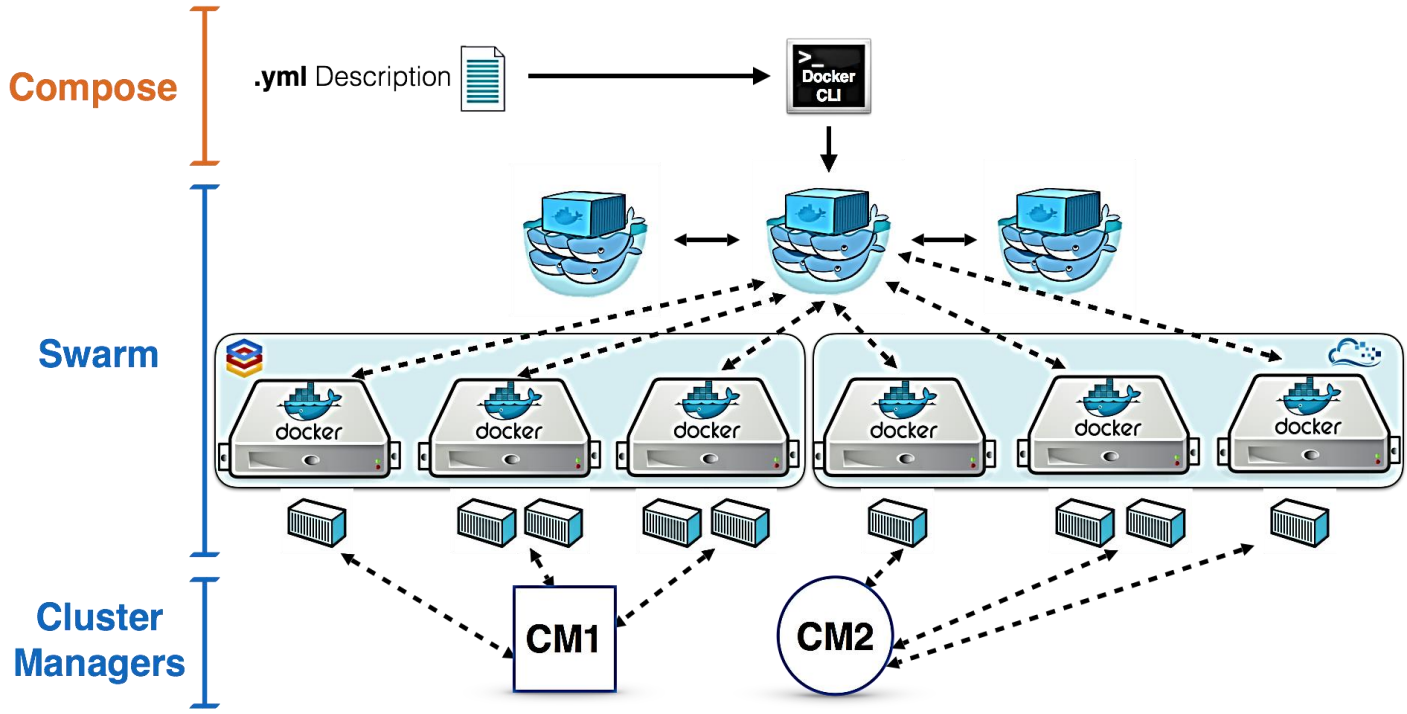
https://cheatography.com/gauravpandey44/cheat-sheets/docker-compose/

DOCKER--COMPOSE.YML

```
version: "3.7"
services:
  wordpress_db:
    container_name: "wordpress_db"
    image: "mysql:5.7"
    volumes:
      -
~/dockers/wordpress/.data/wordpress_db:/var/lib/my
sql
    environment:
      MYSQL_USER: gaurav
      MYSQL_PASSWORD: victory
      MYSQL_DATABASE: db
      MYSQL_RANDOM_ROOT_PASSWORD: '1'
    networks:
      - wordpress_network
    ports:
      - 3307:3306
  wordpress_web:
    container_name: "wordpress_web"
    image: "wordpress"
    volumes:
      -
~/dockers/wordpress/.data/wordpress_web:/var/ww
w/html
    environment:
      WORDPRESS_DB_HOST: wordpress_db
      WORDPRESS_DB_USER: gaurav
      WORDPRESS_DB_PASSWORD: victory
      WORDPRESS_DB_NAME: db
    networks:
      - wordpress_network
    ports:
      - 8080:80
    depends_on:
      - wordpress_db
networks:
  wordpress_network:
```

Docker Swarm: serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts.

https://itnext.io/administering-two-or-more-docker-swarm-clusters-with-portainerio-682d01a92b25



CONTAINER - CONNECTION MODES:

I-Interactive T- Connected to Terminal D-Detached Mode

- Detached Mode: $ docker run –itd ubuntu: xenial --->itd
  1. user Manages from Daemon 2. Allow control over containers
- Root User Mode: $ docker run –it ubuntu: xenial --->it
  1. User manages from root 2. Allow control over the containers of which user is attached.

Docker run: Interactive Connected to Terminal

▪ *docker run –it <image-name>*: This allows the docker container to run in the interactive mode as well as prevent it from exiting. Use ctrl+P+Q to get out of container without exiting

Docker run: Detached Mode:

docker run –d <image-name>: This allows the container to run a service in the background

- Containers started in detached mode exits when the root process used to run the container exits
- A container in detached mode cannot be automatically removed when it stops. Container stops when the root process used to run container stops.

Copying Data to And from Containers

$ *docker cp <container name>:<Source path> <Destination Path>*

Commands Related to Docker File:

- **FROM**: It shows where is the base image coming from.
- **MAINTAINER**: The section of the Docker file shows the maintainer or the owner of the Docker file.
- **RUN**: It is used to take the commands as its argument and runs it to form an image
- **CMD**: Command for starting up of a service of some kind. the actions to run when the containers are initiated is described in this section. Anything that is after a command is a list of things to run within any container that is initiated on a base image
- **ENV**: It is used to set one or more environment variables as $variable name or ${variable name}
- **WORKDIR**: It defines the location where the command defined by cmd is to be executed
- **VOL**: It is used to enable access from your container to a directory
- **ADD**: It copies the file into the containers own file system from the source on the host at the stated destination
- **EXPOSE**: It is used to expose the port to allow networking between the running process inside the container

DIFF B/W RUN & CMD & ENTRYPOINT

- RUN executes command(s) in a new layer and creates a new image. It is often used for installing software packages
- CMD sets default command and/or parameters, which can be overwritten from command line when docker container runs
- ENTRYPOINT configures a container that will run as an executable
- CMD: Can be changed. Latest one will be executed
- ENTRY POINT: Can't be changed

https://phoenixnap.com/kb/docker-cmd-vs-entrypoint#:~:text=CMD%20is%20an%20instruction%20that,container%20with%20a%20specific%20executable.

ADD Vs COPY:

ADD: Syntax: *ADD <source/URL> <destination>*
EX: ADD https://jdk 8. 0..
/usr/local/tomcat/webapps/java. War
-->ADD can copy from local to internet and also from internet to local

COPY: Syntax: COPY <source file> <destination>
EX: COPY Jenkins. War
/usr/local/tomcat/webapps/Jenkins. War
---> COPY can only copy from local to internet
---> COPY don't accept URL path.

Importing and Exporting a Container to share with others:

$ ***docker export <container ID> > <name.tar>***

$ ***docker import - <image name> <name.tar>*** after this upload to webserver for download

$ ***docker save -o <name.tar> <image name>*** ----> To deal with committed images

$ ***docker load < <name.tar>***

NOTE: For pushing the image to docker hub add the image tag as <Docker username>/<image Name>

Will you lose your data, when a docker container exists?

No, you won't lose any data when Docker container exits. Any data that your application writes to the container gets preserved on the disk until you explicitly delete the container.

Monitor Docker in production:

Docker provides functionalities like docker stats and docker events to monitor docker in production. Docker stats provides CPU and memory usage of the container. Docker events provide information about the activities taking place in the docker daemon.

What changes are expected in your docker compose file while moving it to production?

- Remove volume bindings, so the code stays inside the container and cannot be changed from outside the container.
- Binding to different ports on the host.
- Specify a restart policy
- Add extra services like log aggregator

List of Docker Commands:

1. Docker Container Commands:

Create a container (without starting it):
$ ***docker create [IMAGE]***
# Rename an existing container:
$ ***docker rename [CONTAINER_NAME] [NEW_CONTAINER_NAME]***
# Run a command in a new container:
$ ***docker run [IMAGE] [COMMAND]***
$ ***docker run --rm [IMAGE]*** – removes a container after it exits.
$ ***docker run -td [IMAGE]*** – starts a container and keeps it running.

$ ***docker run -it [IMAGE]*** – starts a container, allocates a pseudo-TTY connected to the container's stdin, and creates an interactive bash shell in the container.
$ ***docker run -it-rm [IMAGE]*** – creates, starts, and runs a command inside the container. Once it executes the command, the container is removed.
$ ***docker exec -it <container name> /bin/bash*** ---> to enter into a container bash
# Delete a container (if it is not running):
$ **docker rm [CONTAINER]**
# Update the configuration of one or more containers:
$ **docker update [CONTAINER]**

2. Starting and Stopping Containers:

#Start a container:
$ ***docker start [CONTAINER]***
# Stop a running container:
$ ***docker stop [CONTAINER]***
# Stop a running container and start it up again:
$ ***docker restart [CONTAINER]***
$ ***docker system prune --all*** *---> to delete unwanted and not running containers*
Pause processes in a running container:
$ ***docker pause [CONTAINER]***
# Un-pause processes in a running container:
$ ***docker unpause [CONTAINER]***
# Block a container until others stop (after which it prints their exit codes):
$ ***docker wait [CONTAINER]***
# Kill a container by sending a SIGKILL to a running container:
$ ***docker kill [CONTAINER]***
# Attach local standard input, output, and error streams to a running container:
$ ***docker attach [CONTAINER]***
Note: If you are still unsure of how Docker images and containers differ, you may want to check out the article on Images vs Containers.

3. Docker Image Commands:

Create an image from a Dockerfile:
$ **docker build [URL]**
$ **docker build -t .** – builds an image from a Dockerfile in the current directory and tags the image
# Pull an image from a registry:
$ ***docker pull [IMAGE]***
# Push an image to a registry:
$ ***docker push [IMAGE]***
# Create an image from a tarball:
$ ***docker import [URL/FILE]***
# Create an image from a container:
$ ***docker commit [CONTAINER] [NEW_IMAGE_NAME]***
# Remove an image:
$ ***docker rmi [IMAGE]***
# Load an image from a tar archive or stdin:
$ ***docker load [TAR_FILE/STDIN_FILE]***
# Save an image to a tar archive, streamed to STDOUT with all parent layers, tags, and versions:
$ ***docker save [IMAGE] > [TAR_FILE***]

DOCKER IMAGE BUILD, PUSH AND PULL:
$ ***docker built -t crsreddy1447/mygit:1.0***
$ ***docker login***
$ ***docker commit 4328fa8ba39e crsreddy1447/gol:1.0*** -----> crsreddy1447 is username, gol is image name
$ ***docker push docker.io/crsreddy1447/gol:1.0***
$ ***docker pull crsreddy1447/gol:1.0***
$ ***docker container run -itd -p 1111:8080 crsreddy1447/gol:1.0***

4. Docker Commands for Container and Image Information:

List running containers:
$ ***docker ps***
$ ***docker ps -a*** – lists both running containers and ones that have stopped
$ ***docker ps -a -q*** ---> Show containers id
$ ***docker rm -f*** ---> remove running container force
$ ***docker rm -f $(docker ps -a -q)*** ----> Delete the containers by showing their id
# List the logs from a running container:
$ ***docker logs [CONTAINER]***
# List low-level information on Docker objects:
$ ***docker inspect [OBJECT_NAME/ID]***
# List real-time events from a container:
$ ***docker events [CONTAINER]***
# Show port (or specific) mapping for a container:
$ ***docker port [CONTAINER]***
# Show changes to files (or directories) on a filesystem:
$ ***docker diff [CONTAINER]***
# List all images that are locally stored with the docker engine:
$ ***docker image ls***
# Show the history of an image:
$ ***docker history [IMAGE]***

DATA SHARING:
$ **docker volume create <vol name>**
$ **docker volume inspect vol1** **#** ----/var/lib/docker/volumes/vol1/_data
$ **docker run -it --name <container name> -v <vol name>:</location to store> <image>**
Ex: docker run -it --name test1 -v vol1:/logs ubuntu

5. Networks Related Commands:

List networks:
$ ***docker network ls***
# Remove one or more networks:
$ ***docker network rm [NETWORK]***
# Show information on one or more networks:
$ ***docker network inspect [NETWORK]***
# Connects a container to a network:
$ ***docker network connect [NETWORK] [CONTAINER]***
# Disconnect a container from a network:
$ ***docker network disconnect [NETWORK] [CONTAINER]***

6. DOCKER SWARM COMMANDS

Cluster Management

$ ***docker swarm init --advertise-addr <ip>*** # Set up master
$ ***docker swarm init --force-new-cluster -advertise-addr <ip>*** # Force manager on broken cluster

$ ***docker swarm join-token worker*** ******# Get token to join workers
$ ***docker swarm join-token manager*** ******# Get token to join new manager
$ ***docker swarm join <server> worker*** ***** # Join host as a worker

$ ***docker swarm leave***
$ ***docker swarm unlock*** # Unlock a manager host after docker
$ ***docker swarm unlock-key*** # Print key needed for 'unlock'
$ ***docker node ls*** # Print swarm node list
$ ***docker node rm <node id>***
$ ***docker node inspect --pretty <node id>***
$ ***docker node promote <node id>*** # Promote node to manager
$ ***docker node demote <node id>***

Rebalancing

Draining a node
$ ***docker node update --availability drain <node id>***
Undrain
$ ***docker node update --availability active <node id>***

Managing Services

$ ***docker stack ls***
$ ***docker stack rm <name>***

$ ***docker service create <image>*** *****
$ ***docker service create --name <name> --replicas <number of replicas> <image>***
$ ***docker service scale <name>=<number of replicas>*** ****
$ ***docker service rm <service id|name>***
$ ***docker service ls*** # list all services
$ ***docker service ps <service id|name>*** # list all tasks for given service (includes shutdown/failed)
$ ***docker service ps --filter desired-state=running <service id|name>*** # list running (acitve) tasks for given service

$ ***docker service logs --follow <service id|name>*** # print console log of a service