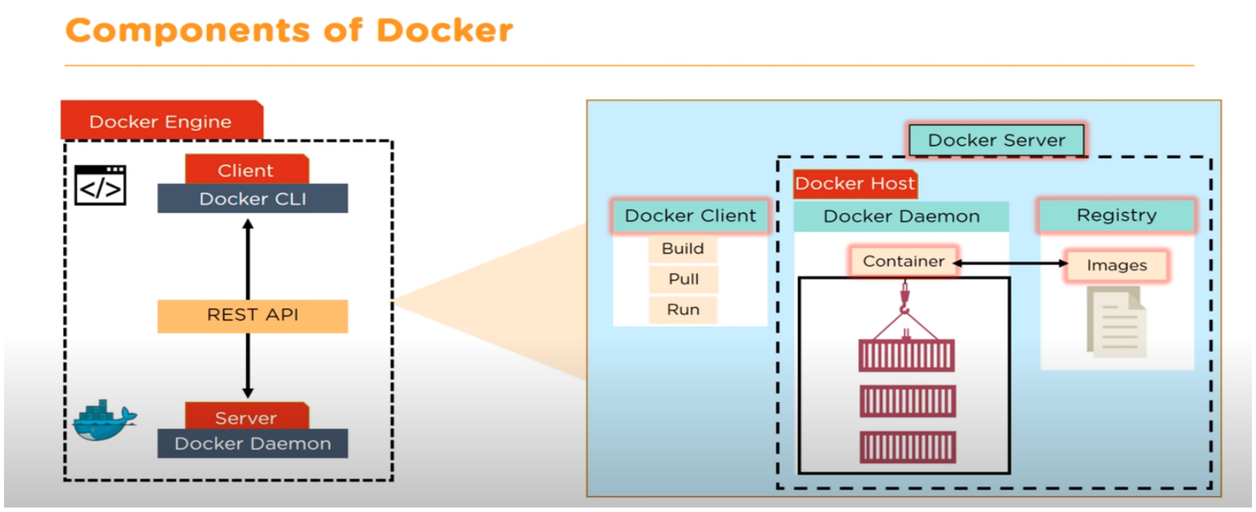
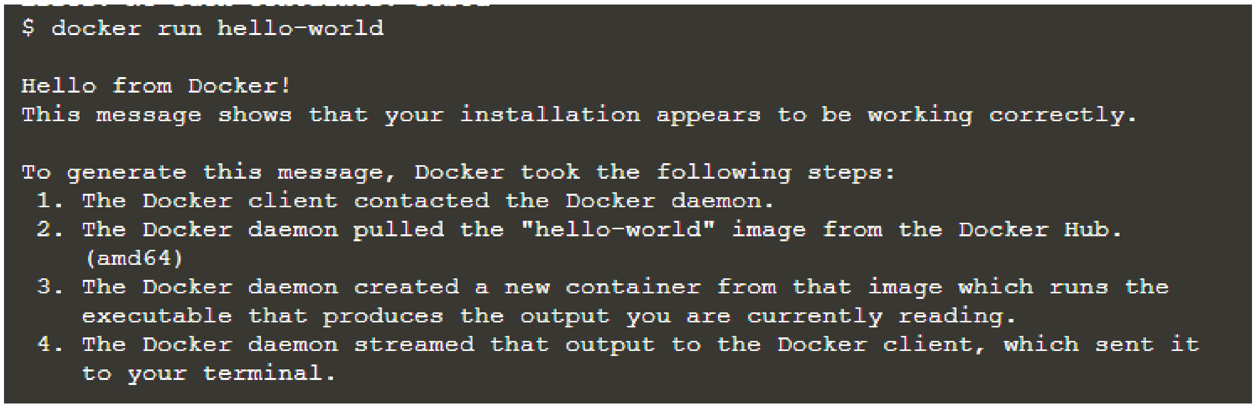
About Docker
Docker uses a client-server architecture and is written in Go language.



Docker Client: Docker users can interact with docker daemon through a client. Docker Client uses commands(when connecting locally) or REST API(when connecting remotely) to communicate with the docker daemon. Client can communicate with more than 1 daemon. docker command uses the Docker API.

Docker Daemon(dockerd): It listens for Docker API requests and manages docker objects such as images, containers, volumes. It pulls image from docker hub and creates container from that image.Can communicate with other daemons.



```
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
```

Docker Engine: Consists of both Docker Client and Docker Daemon or its just docker daemon.

Docker Hub: Container Registry

Docker Host/ Docker Server: Can be laptop, physical computer or VM
Life before docker
Adv or Docker./life after docker
Disadv of Docker

Installation of Docker in Linux based OS
```
$ sudo yum update -y
$ sudo yum install docker -y
$ sudo service docker start
```

```
# Adding ec2-user to docker group so that it can also use docker commands
$ sudo usermod -aG docker ec2-user (Group name space user)
```

```
# Create new session
$ exit
```

```
# SSH and type
$ id ec2-user
```

Basic Docker commands

```
# For checking info
$ docker info
```

```
# Path where docker is installed /usr/bin/docker
$ which docker
```

```
# Version of docker
$ docker --version/-v
```

```
# Download Docker Image from Docker Hub
$ docker pull hello-world
```

```
# Pull image from Docker Hub
$ docker pull ashokit/spring-boot-rest-api
$ docker run hello-world
```

```
#  List local images
docker images
```

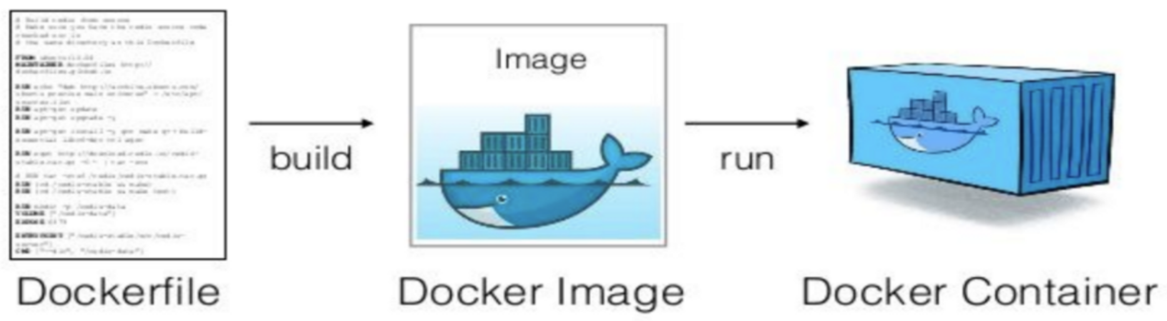
```
# List all running containers
$ docker ps
```

```
# List running and stopped containers
$ docker ps -a
```

# Remove image
$ docker rmi <image-name/image-id> (Provided: There should be no container associated with this image either in running or stopped state)

# Force remove image
$ docker rmi -f <image-name/image-id> (Can delete image even if there is an associated container)

# Remove Container
docker rm <container-name/container-id> (Provided container is stopped)



Dockerfile: A text file which has set of instructions to build docker image
Docker Image: Package which contains source/application code and dependencies
Docker Container: Running instance of an image
Docker Hub: Registry to store docker images

1) What is application stack
   - Frontend
   - Backend
   - Database

2) Application Environments (Non-Prod & Prod)
   - Dev Env
   - SIT Env
   - UAT Env
   - Pre Prod Env
   - Prod Env (live)

3) Life without Docker

4) Life with Docker

5) What is Docker (Platform to run our applications)

6) Docker Architecture

   - Dockerfile (Set of instructions to build docker image)

   - Docker Image (package which contains source code + dependencies)

   - Docker Registry (Docker Hub, AWS ECR)

   - Docker Container (It is runtime instance of our application)

======================
Docker Basic Commands
======================

```
# To check docker information
$ docker info

# To display Docker images
$ docker images

# Download docker image from Docker hub
$ docker pull < image-id / image-name>

# Download docker hello-world image
$ docker pull hello-world
```

```
# Delete docker image
$ docker rmi <image-id / image-name>
```

```
# Run docker container
$ docker run < image-id/image-name >
```

Note: when we use 'run' command it will check for image in local, if image not available then it will pull the image and it will run the image

```
# To display all running containers
$ docker ps
```

```
# To display all the running and stopped container
# docker ps -a
```

```
# Delete docker container
$ docker rm < container-id >
```

===================

Docker Terminology

=====================

Dockerfile : It contains set of instructions to build docker image

Docker Image: It is a package which contains application code + dependencies

Docker Hub : It is a registry to store docker images

Docker Container: It is a runtime instance of our application

============

Dockerfile

============

-> Dockerfile contains set of instructions to build docker image

-> In Dockerfile we will use DSL (Domain Specific Language) keywords

-> Below are the keywords we will use in Dockerfile

FROM	CMD	WORKDIR
MAINTAINER	ENTRYPOINT	EXPOSE
COPY	ENV	VOLUME
ADD	LABEL	
RUN	USER	

========
FROM
========

-> It indicates the base image to run our application.
-> On top of base image our application image will be created

Syntax:

```
FROM <image-name>
```

Example:

```
FROM java:jdk-1.8.0

FROM tomcat:9.5

FROM mysql

FROM python
```

=============
MAINTAINER
==============

-> It represents author of the Dockerfile

Example:

```
MAINTAINER  Ashok <ashokitschool@gmail.com>
```

========
COPY
========

-> It is used to copy files / folders to docker image from our system while creating image

Syntax:

```
COPY <source-location> <destination-location>
```

Example:

```
COPY target/java-web-app.war  /usr/local/tomcat/webapps/java-web-app.war
```

========
ADD
========

-> It is also used to copy the files from one location to another location

Ex:

ADD <source> <destination>

ADD <url> <destination>

Q) What is the difference between COPY and ADD ?

-> COPY can only works with source location

-> ADD can work with source location & URL also

=======
RUN
=======

-> It is used to execute commands while creating docker image

-> We can write multiple RUN instructions in docker file, Docker will process all RUN instructions

-> When we have multiple RUN instructions they will execute from top to bottom

Example:

RUN mkdir workspace

RUN yum install git

RUN yum install maven

=========
CMD
=========

-> It is also used to execute the commands

-> CMD instructions will execute while creating docker container

-> Technically we can write multiple CMD instructions in dockerfile but Docker will process only last CMD instruction
  (There is no use of writing multiple CMD instructions in Dockerfile)

FROM
MAINTAINER
COPY
ADD
RUN
CMD

==============Dockerfile===============

```
FROM ubuntu

MAINTAINER Ashok <ashokitschool@gmail.com>

RUN echo "this is first RUN statement"

RUN  echo "this is second RUN statement"

CMD  echo "this is first CMD statement"

RUN echo "this is third RUN statement"

CMD echo "this is second CMD statement"

CMD echo "this is third CMD statement"
```

==================================

```
# Building Docker image using Dockerfile
$ docker build -t <image-name> .

# Run Docker Image
$ docker run <image-name>
```

```
# creating docker image with custom file name
$ docker build -f <file-name> -t imagetwo .

# Login with Docker Hub account
$ docker login

# tag docker image for pushing
$ docker tag imagefour ashokit/imagefour

# push image into docker hub
$ docker push ashokit/imagefour

# pulling docker image
$ docker pull ashokit/imagefour

# Run docker image
$ docker run ashokit/imagefour
```

================
ENTRYPOINT
================

-> ENTRYPOINT is used to execute commands while creating container

Note: CMD instructions we can override where as ENTRYPOINT instructions we can't ovveride

Example

```
ENTRYPOINT ["echo", "Welcome to ashokit" ]

ENTRYPOINT ["java", "-jar", "spring-boot-rest-api.jar" ]
```

==============
WORKDIR
==============

-> It is used to set working directory for an image / container

Ex:

```
WORKDIR <DIRNAME>
```

Note: After WORKDIR instruction the remaining instructions will execute in WORKDIR location

```
WORKDIR /home/username/app/

RUN  sh "git clone url"

RUN sh 'mvn clean package'
```

===========
ENV
===========

-> It is used set environment variables

Ex:

```
ENV <key> <value>
```

========
LABEL
========

-> It is used to represent data in key-value pair

Ex:

```
LABEL branch-name release
```

ex:

```
WORKDIR /home/username/app/
```

```
LABEL branchName release

RUN  sh "git clone url"

RUN sh 'mvn clean package'
```

============
EXPOSE
============

-> Expose keyword represents on which port number our application container running

-> It is like a documentational command just to provide information

Ex:

```
EXPOSE 8080
```

Note; If we don't write EXPOSE nothing will happen

==========
USR
==========

-> It is used to set username to create image/container

EX:

```
USR root
```

===========
ARG
==========

-> It is used avoid hard coded values in Docker file

Ex:

```
ARG branch
RUN sh 'git clone -b $branch <repo-url>'
```

```
$ docker build -t <image-name> --build-arg branch=feature
```

===============
VOLUME
===============

-> It is used to specify storage location for our container

FROM
MAINTAINER
COPY
ADD
RUN
CMD
ENTRYPOINT
USR
ARG
LABEL
EXPOSE
WORKDIR
ENV
VOLUME

```
# Building Docker image using Dockerfile
$ docker build -t <image-name> .

# Run Docker Image
$ docker run <image-name>

# creating docker image with custom docker file name
$ docker build -f <file-name> -t imagetwo .

# Login with Docker Hub account
$ docker login

# tag docker image for pushing
$ docker tag imagefour ashokit/imagefour

# push image into docker hub
$ docker push ashokit/imagefour

# pulling docker image
$ docker pull ashokit/imagefour
```

```
# Run docker image
$ docker run ashokit/imagefour

# Display all images
$ docker images

# Remove particular image
$ docker rmi <image-id/image-name>

# Remove the image forcefully
$ docker rmi -f <image-id/image-name>

# Display all running containers
$ docker ps

# Display all running & stopped containers
$ docker ps -a

# Stop The container
$ docker stop <container-id>

# Remove the container
$ docker rm <container-id>

$ Remove Stopped containers & un-used images
$ docker system prune -a
```

=========================
Java Web application Types
=========================

-> In industry we can see below types of java applications

1) Java Web Application without Spring Boot (10 %)

2) Java Web Application with Spring Boot (90 %)

-> When we develop Java web application without Spring Boot then we need an external Server to run our java web application (Ex: Apache Tomcat).

-> Normal java web apps will be packaged as war file (web archieve)

-> When we develop Java web application with Spring Boot then we no need worry about server because Spring Boot providing embedded server, it will take care of web application execution.

-> Spring Boot applications will be packaged as jar file (java archieve)

==================================
Dockerize Normal Java Web Application
==================================

Application Code : Java web application with Maven Build Tool (Git Hub Repo)

Dependencies : Java + Tomcat

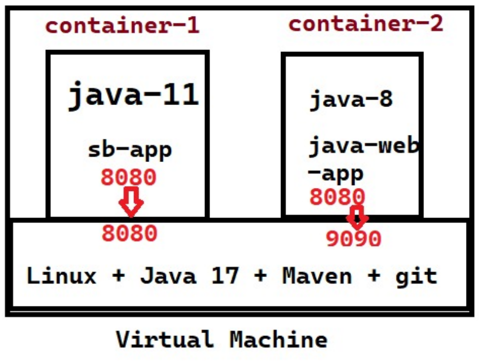
Java Web App Repo : https://github.com/ashokitschool/maven-web-app.git

```
-----------------------------------------Dockerfile---------------------------------------------------------------
FROM tomcat:10.0.26-jre8

COPY target/01-maven-web-app.war /usr/local/tomcat/webapps/maven-web-app.war
-------------------------------------------------------------------------------------------------------------------
```

```
$ sudo yum install git -y

$ git clone <repo-url>

$ cd <project-folder>

$ docker build -t maven-web-app .

$ docker run -d -p 8080:8080 maven-web-app
```

Note : Enable 8080 port in EC2 security Group

URL : http://13.235.243.1:8080/maven-web-app/



```
sb-web-app

    -> base-image: openjdk:11
    -> container-port : 8080
    -> host-port : 8080

maven-web-app

    -> base-image: tomcat:jre8
    -> container-port : 8080
    -> host-port : 9090
```

================================
Dockerize Spring Boot Application
=================================

-> Spring Boot is a java framework which is used to develop java based applications

-> Spring Boot will provide embedded server to run java web application (no need to configure server manually)

-> Spring Boot applications will be packaged as jar file

-> To execute jar file we will use below command

```
$ java -jar <jar-file-name>
```

Spring-boot App Repo : https://github.com/ashokitschool/spring-boot-docker-app.git

--------------------------------Dockerfile--------------------------------

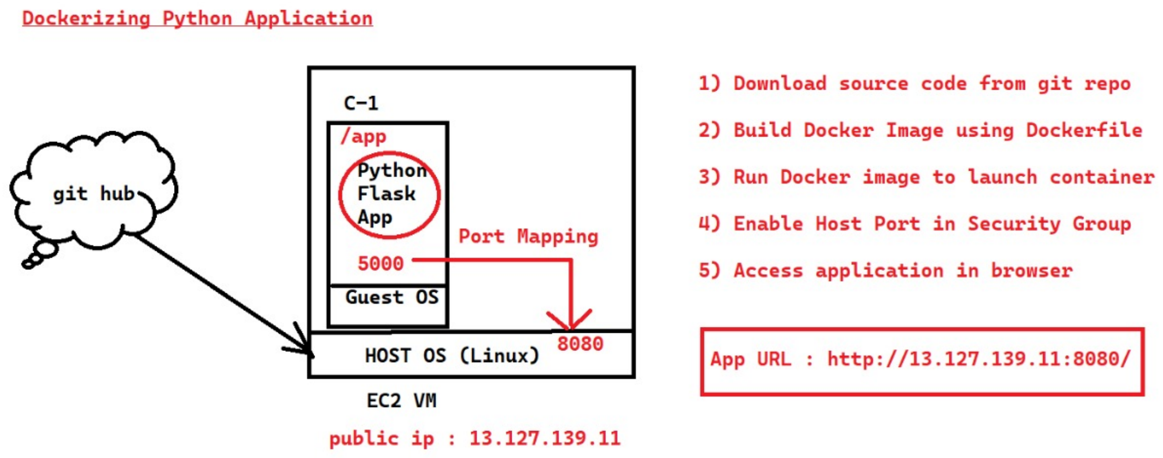
```
FROM openjdk:11

COPY target/spring-boot-docker-app.jar  /usr/app/

WORKDIR /usr/app/

ENTRYPOINT ["java", "-jar", "spring-boot-docker-app.jar"]
```

👉 Java Web App Repo : https://github.com/ashokitschool/maven-web-app.git

👉 Spring-boot App Repo : https://github.com/ashokitschool/spring-boot-docker-app.git

👉 Python App Repo : https://github.com/ashokitschool/python_flask_docker_app.git

==================================
Dockerizing Python Flask Application
==================================


Dockerizing Python Application
git hub
C-1
/app
Python Flask App
5000
Guest OS
Port Mapping
8080
HOST OS (Linux)
EC2 VM
public ip : 13.127.139.11
1) Download source code from git repo
2) Build Docker Image using Dockerfile
3) Run Docker image to launch container
4) Enable Host Port in Security Group
5) Access application in browser
App URL : http://13.127.139.11:8080/

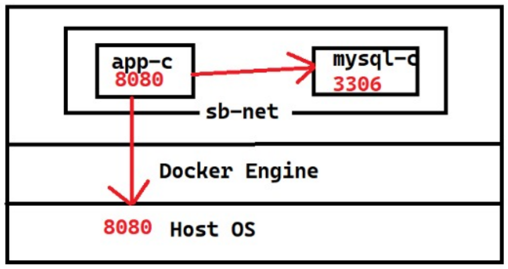
```
FROM python:3.6

MAINTAINER Ashok Bollepalli "ashokitschool@gmail.com"

COPY . /app

WORKDIR /app

RUN pip install -r requirements.txt

ENTRYPOINT ["python", "app.py"]
```

=================================

```
# check docker container logs

$ docker logs -f <container-id>

# Getting into container

$ docker exec -it <container-id> /bash
```

=====================================
Dockerizing Spring Boot + MySQL
=====================================

Dockerizing Spring Boot Application with MySQL Database



=> Create Docker network

=> Run MySQL DB as a container

=> Run Spring Boot Application as container

=> Access Spring Boot Application

```
# Pull MySQL DB image
$ docker pull mysql:5.7

# Download required softwares
$ sudo yum install git -y
$ sudo yum install maven -y

# Clone Spring Boot Project From Github
$ git clone https://github.com/ashokitschool/spring-boot-mysql-docker-compose.git

# Get into project directory
$ cd <project-dir>

# Package application
$ mvn clean package

# Create Application image
$ docker build -t sb-app .

# Check docker images
$ docker images

# Create Docker Network
$ docker network create springmysql-net

# See networks available
$ docker network ls

# Run mysqldb as a docker container
```

```
$ docker run --name mysqldb --network springmysql-net -e MYSQL_ROOT_PASSWORD=root
-e MYSQL_DATABASE=sbms -d mysql:5.7

# check logs of container
$ docker logs -f <container_name>
```

If needed we can check if the database has been created correctly.

We can do the following commands.

```
$ docker exec -it <container_id> bash
$ mysql -u <username> -p
$ show databases;

# Run springboot application as a container
$ docker run --network springmysql-net --name sbapp-container -p 8080:8080 -d sb-app

# check the containers which are running
$ docker ps

# check logs of application container
$ docker logs -f <container_id>

# We can access our application now
URL : http://ec2-vm-public-ip:8080/
```

==

1) What is Docker : Containerization Platform

2) What is Containerization : Packaging application + dependencies for easy deployment

3) Why Docker: Free, Easily we can package, build and run our applications in any machine

4) Docker Architecture:

$$\text{Dockerfile} \xrightarrow{\text{build}} \text{Docker Image} \xrightarrow{\text{run}} \text{Docker Container}$$

5) Docker Terminology

- Dockerfile
- Docker Image
- Docker Hub
- Docker Container

6) What is Dockerfile : Set of instructions to create docker image

7) What is Docker Image: Package which contains code + dependencies

8) What is Docker Hub : A repository to store docker images

9) What is Docker Container : Runtime instance of our application

10) Keywords of Dockerfile

FROM
MAINTAINER
COPY
ADD
RUN
CMD
ENTRYPOINT
EXPOSE
WORKDIR
USR
ENV
LABEL
ARG
VOLUME

11) Applications we have dockerized

a) java web application with tomcat

b) spring boot rest api with java

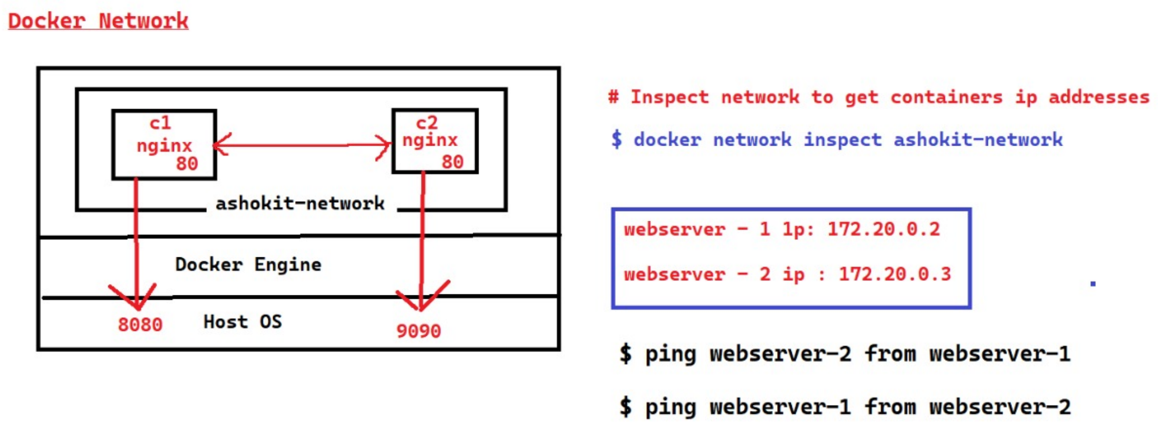
c) python flask app

===================================
Docker Commands we have used so far
===================================

```
$ docker info
$ docker images
$ docker rmi <image-id>
$ docker pull <image-id>
$ docker run <image-id>
$ docker tag <image-name> <image-tag-name>
```

$ docker login
$ docker push <image-tag-name>
$ docker run -d -p host-port:container-port <image-name>
$ docker ps
$ docker ps -a
$ docker stop <container-id>
$ docker rm <container-id>
$ docker rm -f <container-id>
$ docker rm -f $(docker ps -a -q)
$ docker system prune -a
$ docker logs -f <container-id>
$ docker exec -it <container-id> bash

=================
Docker Network
=================



```
# Inspect network to get containers ip addresses
$ docker network inspect ashokit-network

webserver - 1 ip: 172.20.0.2
webserver - 2 ip : 172.20.0.3

$ ping webserver-2 from webserver-1
$ ping webserver-1 from webserver-2
```

-> Network is all about communication

-> Docker network is used to provide isolated network for Docker Containers

-> In Docker we will have below 3 default networks

1) none

2) host

3) bridge

-> In Docker we have below 5 network drivers

1) Bridge ----> This is default network driver in Docker
2) Host

3) None
4) Overlay ----> Docker Swarm
5) Macvlan

-> Bridge driver is recommended we are running standalone container. It will assign one IP for container

-> Host Driver is also used for standalone container. IP will not be assigned for container

-> None means no network will be provided by our Docker containers

-> Overlay network driver is used for Orchestration. Docker Swarm will use this Overlay network driver

-> Macvlan driver will assign MAC address for a container. It makes our container as Physical.

```
# display docker networks available
$ docker network ls

# Create docker network
$ docker network create ashokit-network

# delete docker network
$ docker network rm <network-id>

# Run a container with given network
$ docker run -d -p hport:cport --network ashokit-network <imagename>
```

======================
Monolith Vs Microservices
======================

-> Monolith means single application will be available for all the functionalities

-> Microservices means collection apis will be available in the project / application

1) Products_Api
2) Cart_Api
3) Payment_Api
4) Orders_Api
5) Tracking_Api
6) Cancel_Api
7) Admin_Api

8) Reports_Api
9) Usermanament_api

=> Currently in the market we are developing Microservices Based applications

=> Microservices means collection apis will be available in the project / application

=> Every API should run in a seperate container

=> Running Multiple containers manully for all the apis is difficult job

**************** To solve this problem Docker-Compose came into picture
****

=> Docker Compose is a tool which is used to manage multi container based applications

=> Using Docker Compose we can easily setup & deploy multi container based applications

=> We will give containers information to Docker Compose using YML file (docker-compose.yml)

=> Docker Compose YML should have all the information related to containers creation

======================
Docker Compose YML File
======================

```
version:

services:

network:

volumes:
```

=====================

=> Docker Compose default file name is "docker-compose.yml"

```
# Create Containers using Docker Compose
$ docker-compose up

# Create Containers using Docker Compose with custom file name
$ docker-compose -f <filename> up

# Display Containers created by Docker Compose
$ docker-compose ps

# Display docker compose images
$ docker-compose images


# Stop & remove the containers created by docker compose
$ docker-compose down
```

=====================
Docker Compose Setup
=====================

```
# download docker compose
$ sudo curl -L
"https://github.com/docker/compose/releases/download/1.24.0/docker-compose-$(uname
-s)-$(uname -m)" -o /usr/local/bin/docker-compose

# Give permission
$ sudo chmod +x /usr/local/bin/docker-compose

# How to check docker compose is installed or not
$ docker-compose --version
```

===
Spring Boot with MySQL using Docker Compose (Stateless)
===

```
---
version: "3"
services:
  application:
    image: springboot-app
    networks:
      - springboot-db-net
    ports:
      - "8080:8080"
    depends_on:
      - mysqldb
  mysqldb:
    image: mysql:5.7
    networks:
      - springboot-db-net
    environment:
      - MYSQL_ROOT_PASSWORD: root
      - MYSQL_DATABASE: sbms
networks:
  - springboot-db-net:
...
```

```
$ docker-compose up -d

$ docker-compose ps

$ docker logs -f <container-name>
```

========================
Docker Volumes
========================

-> Applications we are executing using Docker Containers

-> Docker containers are by default stateless

-> Once container removed then we will loose the data that stored in the container

-> In realtime we shouldn't loose the data even the container got removed
For Example : Database container

-> Application will store data in database, even if we delete application container or db container data should be available.

-> To make sure data is available after the container is deleted we will use Docker Volumes concept

**************** Docker Volumes are used to store container data ************************************

=> Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.

=> We have 3 types of volumes in Docker

1) Anonymous Volumes (without name)

2) Named Volumes (Will have a name) ----> Recommended

3) Bind Mounts (Storing on Host Machine)

Q) What is Dangling volume in Docker ?
-> The volumes which are created but not associated to any container are called as Dangling Volumes

```
# Delete all dangling volumes
$ docker volume rm $(docker volume ls -q -f dangling=true);
```

```
# Create Docker volume
$ docker volume create <vol-name>
```

```
# Display all docker volumes
$ docker volume ls
```

```
# Inspect Docker Volume
$ docker volume inspect <vol-name>
```

```
# Delete docker volume
$ docker volume rm <vol-name>
```

```
#Delete all docker volumes
$ docker system prune --volumes
```

--------------------------- Docker Compose with Docker Named Volumne ---------------------------------

```
version: "3"
services:
  application:
    image: springboot-app
    ports:
      - "8080:8080"
    networks:
      - springboot-db-net
    depends_on:
      - mysqldb
  mysqldb:
    image: mysql:5.7
    networks:
      - springboot-db-net
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=sbms
    volumes:
      - app_data:/var/lib/mysql
networks:
  springboot-db-net:
volumes:
   app_data:
```

+++++++++++++++
Docker Swarm
++++++++++++++

-> It is a container orchestration sofware

-> Orchestration means managing processes

-> Docker Swarm is used to setup Docker Cluster

-> Docker swarm is embedded in Docker engine

-> Cluster means group of servers

-> We will setup Master and Worker nodes using Docker Swarm cluster

-> Master will schedule the tasks (containers) and manage the nodes and node failures

-> Worker nodes will perform the action (containers will run here)

Swarm Features
+++++++++++++++
1) Cluster Management
2) Decentralize design
3) Declarative service model
4) Scaling
5) Multi Host Network
6) Service Discovery
7) Load Balancing
8) Secure by default
9) Rolling Updates

Setup
+++++

-> Create 3 EC2 instances (ubuntu)

Note: Enable 2377 port for Swarm Cluster Communications

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh
```

1 - Master
2 - Nodes

-> Connect to Master Machine and execute below command

```
$ sudo docker swarm init --advertise-addr 172.31.38.222
```

```
$ sudo docker swarm join-token worker
```

```
$ sudo docker swarm join --token
SWMTKN-1-21l3z1izmf6plkgprlzfcr87fcxcsl3k2k79iax7yfyh3k4a00-cfgwhikmbcgsklhsxkroj8vad
172.31.38.222:2377
```

Q) what is docker swarm manager quarm?

Ans) If we run only 2 masters then we can't get High Availability

Formula : (n-1)/2

If we take 2 servers

2-1/2 => 0.5 (It can't become master)

3-1/2 => 1 (it can be leader when the main leader is down)

Note: Always use odd number for Master machines

-> In Docker swarm we need to deploy our application as a service.

Docker Swarm Service
++++++++++++++++++++

-> Service is collection of one or more containers of same image

-> There are 2 types of services in docker swarm

1) Replica (default mode)
2) global

```
$ sudo docker service create --name <serviceName> -p <hostPort>:<containerPort>
<imageName>
```

```
$ sudo docker service create --name java-web-app -p 8080:8080 ashokit/javawebapp
```

Note: By default 1 replica will be created

```
# check the services created
$ docker service ls
```

```
# we can scale docker service
$ docker service scale <serviceName>=<no.of.replicas>
```

```
# inspect docker service
$ sudo docker service inspect --pretty java-web-app
```

```
# see service details
$ sudo docker service ps java-web-app


# Remove one node from swarm cluster
$ sudo docker swarm leave

# remove docker service
$ sudo docker service rm helloworld
```