Docker

Agenda

- **What is Docker**
- **Docker Image**
- **Docker Hub/Registry**
- **Docker Container**
- **Architecture**
- **Installation**
- **Commands**
- **Port Mapping**
- **Volume**
- **Dockerfile**
- **Basics in Docker networking**
- **Docker-Compose**
- **Docker Swarm**
- **Monitoring containers-cAdvisor**

Docker is a platform designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all the parts it needs, such as libraries and other dependencies, and ship it all out as one package

It was first released in 2013 and is developed by Docker, Inc

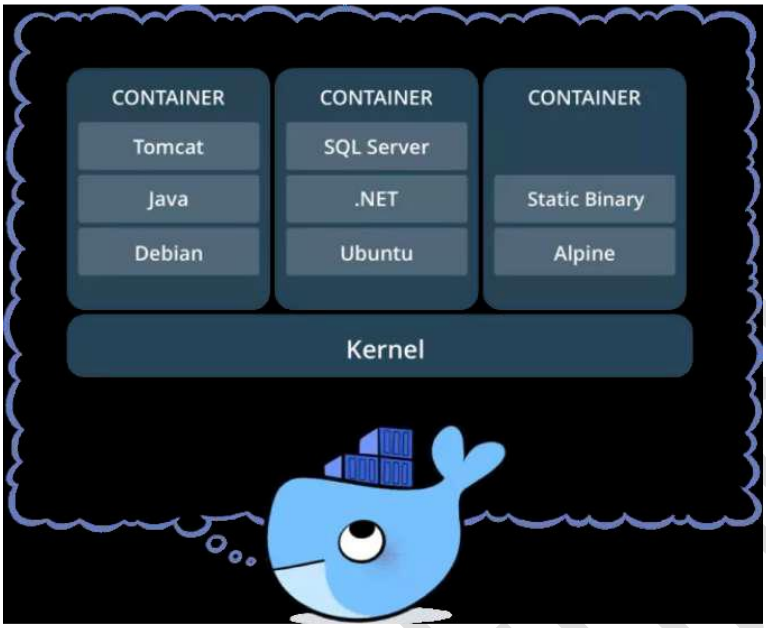
Container
A container is something quite like a virtual machine, which can be used to contain and execute all the software required to run a program.

The container **includes an operating system** (typically some flavor of Linux) as base, plus any software installed on top of the OS that might be needed.
This container can therefore be run as a **self-contained virtual environment**, which makes it a lot easier to reproduce the same analysis on any infrastructure that supports running the container, from your laptop to a cloud platform, without having to go through the pain of identifying and installing all the software dependencies involved.
You can even have **multiplecontainers running on the same machine**, so you can easily switch between different environments if you need to run programs that have incompatible system requirements.

Containerization is increasingly popular because containers are:

- Flexible: Even the most complex applications can be containerized.

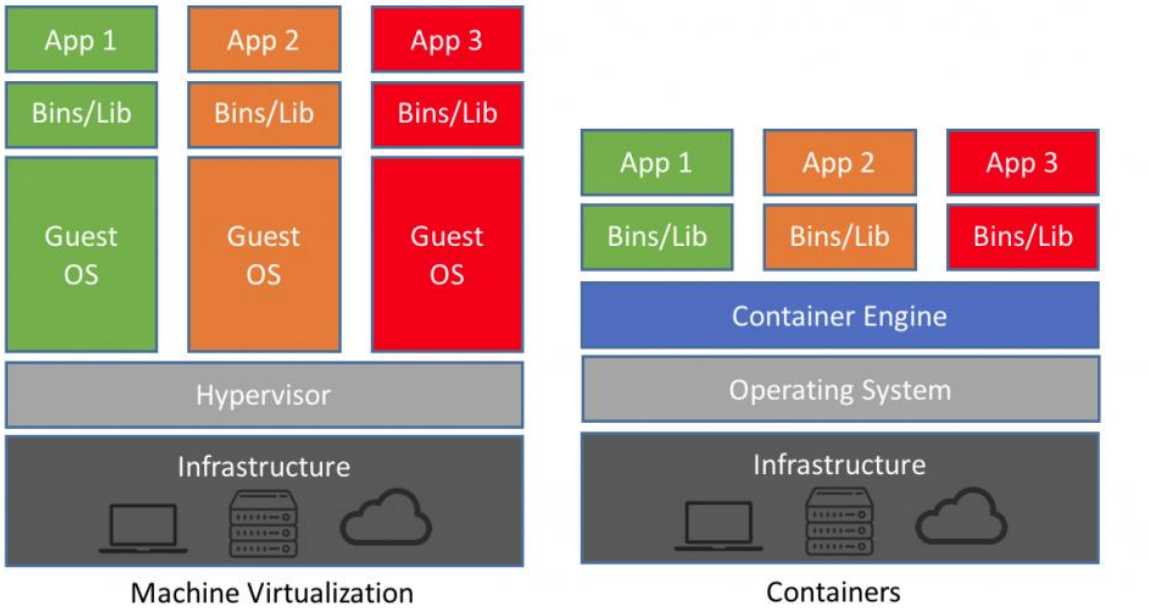
- Lightweight: Containers leverage and share the host kernel.
- Interchangeable: You can deploy updates and upgrades on-the-fly.
- Portable: You can build locally, deploy to the cloud, and run anywhere.
- Scalable: You can increase and automatically distribute container replicas.
- Stackable: You can stack services vertically and on-the-fly.



Difference Containers and virtual machines

A container runs natively on Linux and shares the kernel of the host machine with other containers. It runs a discrete process, taking no more memory than any other executable, making it lightweight.

By contrast, a **virtual machine (VM) runs a full-blown "guest" operating system with virtual access to host resources through a hypervisor**. In general**, VMs provide an environment with more resources than most applications need.**



VM	Container
Heavyweight	Lightweight
Limited performance	Native performance
Each VM runs in its own OS	All containers share the host OS
Hardware-level virtualization	OS virtualization
Startup time in minutes	Startup time in milliseconds
Allocates required memory	Requires less memory space
Fully isolated and hence more secure	*Process-level isolation, possibly less secure*

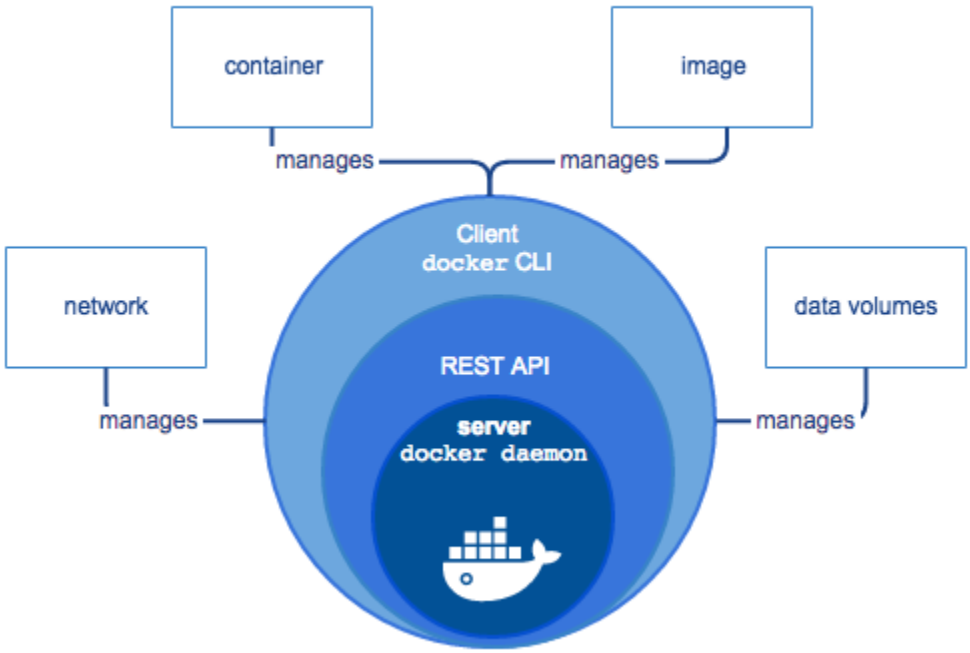
- VMs are a better choice for running apps that require all the operating system's resources and functionality, when you need to run multiple applications on servers, or have a wide variety of operating systems to manage.

- Containers are a better choice when your biggest priority is maximizing the number of applications running on a minimal number of servers

Docker Engine

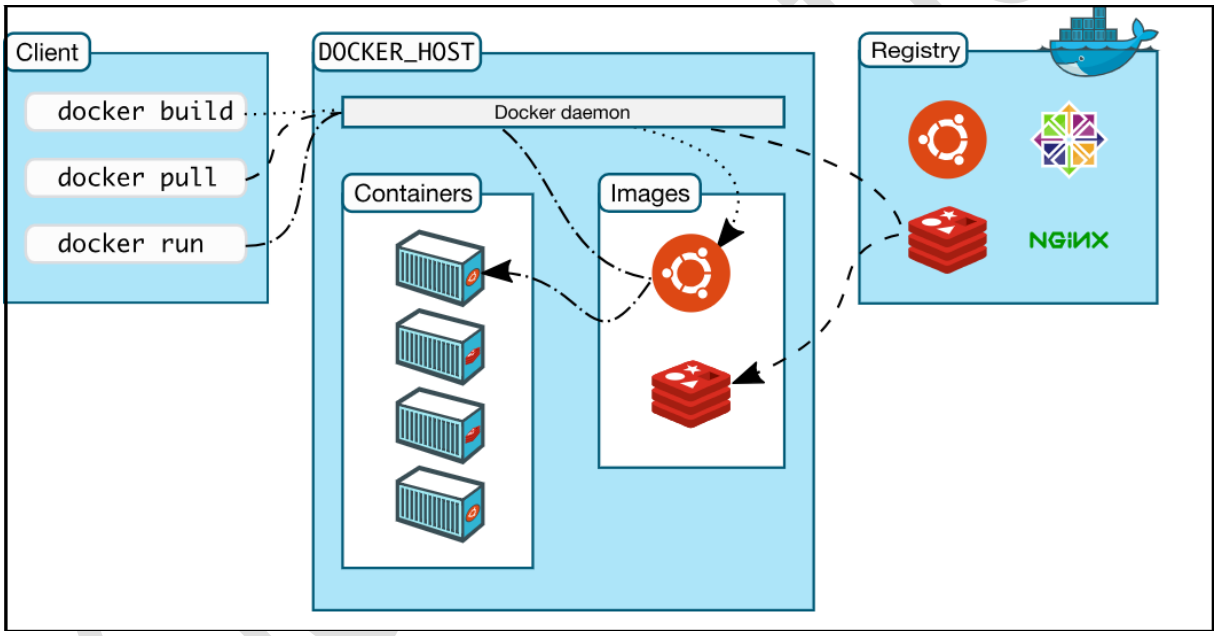
Docker Engine is a client-server application with these major components:

- A server which is a type of long-running program called a daemon process (the `dockerd`command).
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client (the `docker` command).
- The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications use the underlying API and CLI.
- The daemon creates and manages Docker objects, such as images, containers, networks, and volumes.



Docker Architecture

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface



Docker Components

The Docker daemon (dockerd)
listens for Docker API requests and manages Docker objects such as images, containers, networks, andvolumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client (docker)
tool to communicate with Docker daemon.Docker imagesis a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization

Docker registries

A Docker *registry* stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. If you use Docker Datacenter (DDC), it includes Docker Trusted Registry (DTR).

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

Docker Services
allow you to scale containers across multiple Docker daemons, which all work together as a swarm withmultiple managers and workers

Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. Below are docker objects

IMAGES

An *image* is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a *Dockerfile* with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer

in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

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, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Docker Installation

On Ubuntu16
 # apt install docker.io

For other methods and OS
https://docs.docker.com/install/#supported-platforms

#docker –version
#docker info

ps -ef | grep dockerd

#docker search ubuntu
Similarly for centos,Jenkins,Redhat,tomcat

Docker image repository
https://hub.docker.com

Commands related to docker

Lifecycle:
o docker create - creates a container but does not start it.
o docker rename - allows the container to be renamed.
o docker run - creates and starts a container in one operation.
o docker rm - deletes a container.
o docker update - updates a container's resource limits.

Starting and Stopping:
o docker start - starts a container so it is running.
o docker stop - stops a running container.
o docker restart - stops and starts a container.
o docker pause - pauses a running container, "freezing" it in place.
o docker unpause - will unpause a running container.
o docker wait - blocks until running container stops.
o docker kill - sends a SIGKILL to a running container.
o docker attach - will connect to a running container.

Info:
o docker ps - shows running containers.
o docker logs - gets logs from container.
o docker inspect - looks at all the info on a container.
o docker events - gets events from container.
o docker port - shows public facing port of container.
o docker top - shows running processes in container.
o docker stats - shows containers' resource usage statistics.
o docker diff - shows changed files in the container's FS.

Import / Export:
o docker cp - copies files or folders between a container and the local filesystem.
o docker export - turns container filesystem into tarball archive stream to STDOUT.

Executing Commands:
o docker exec - to execute a command in container.

docker run hello-world
#docker pull centos
#docker images
#docker pull ubuntu:16.04
#docker pull centos:6
#docker images

docker run -i -t ubuntu:16.04 /bin/bash

When you run this command, the following happens (assuming you are using the default registry configuration):

1. If you do not have the ubuntu image locally, Docker pulls it from your configured registry, as though you had run docker pull ubuntu manually.
2. Docker creates a new container, as though you had run a docker container create command manually.
3. Docker allocates a read-write filesystem to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.
4. Docker creates a network interface to connect the container to the default network, since you did not specify any networking options. This includes assigning an IP address

to the container. By default, containers can connect to external networks using the host machine's network connection.

5. Docker starts the container and executes /bin/bash. Because the container is running interactively and attached to your terminal (due to the -i and -t flags), you can provide input using your keyboard while the output is logged to your terminal.
6. When you type exit to terminate the /bin/bash command, the container stops but is not removed. You can start it again or remove it.

Giving tag to docker image
#docker tag centos:6 mycentos:6

Sample Commands Related to container

#docker ps (List Running containers)
#docker ps -a (list all including stopped containers)
#docker top <containerID> Display process of container
#docker stop <cid> Stop running container
#docker start <cid> Start stopped container
#docker pause <cid> Pause all processes within container
#docker unpuase
docker rm <cid> Delete container
docker commit create and image from container
#docker attach Enter running container
#docker logs <cid>
docker logs $(docker ps -aql)

Docker exec: Run a command in a running container
#docker run -itd -p 80:80 httpd
#docker ps
docker exec ff92ff19651a cat /etc/os-release
docker exec ff92ff19651a apt-get update

Docker info
#docker info
#ls -l /var/lib/docker

```
root@ip-172-31-10-171:~# ls -l /var/lib/docker/
total 36
drwx------  5 root root 4096 Oct 18 04:18 aufs
drwx------ 11 root root 4096 Oct 18 05:14 containers
drwx------  3 root root 4096 Oct 18 04:18 image
drwxr-x---  3 root root 4096 Oct 18 04:18 network
drwx------  4 root root 4096 Oct 18 04:18 plugins
drwx------  2 root root 4096 Oct 18 04:18 swarm
drwx------  2 root root 4096 Oct 18 04:52 tmp
drwx------  2 root root 4096 Oct 18 04:18 trust
drwx------  2 root root 4096 Oct 18 04:18 volumes
```

Create Custom image and push to Docker Hub

docker run -i -t ubuntu:16.04 /bin/bash
……..
……
Here install jdk,tomcat7 and deploy sample webapp
……..
Ctrl p+q

#docker ps
docker commit 8b7deb8c02c9 webapp/insurance:1.1
#docker images
#docker inspect <imgid or cid> (Return low-level information on Docker objects)
#docker login
docker tag webapp/insurance:1.1 ajit2408/insurance
#docker push ajit2408/insurance
(If image name is long you will get error “requested access to the resource is denied”)
Check new image on your login https://hub.docker.com/

Working with Docker Hub/Registry:

o docker login - to login to a registry.
o docker logout - to logout from a registry.
o docker search - searches registry for image.
o docker pull - pulls an image from registry to local machine.
o docker push - pushes an image to the registry from local machine.

Docker save
Save one or more images to a tar archive
#docker save -o ubuntu.tar.gz ubuntu
Or

docker save ubuntu > myubuntu.tar.gz
#ls

Docker load
Load an image from a tar archive
docker load -i myubuntu.tar.gz
OR
docker load < myubuntu.tar.gz

Docker Import
Import the contents from a tarball to create a filesystem image
#docker import myubuntu.tar mycustubunt
#docker images

Removing images and containers
#docker images -aq (List all images numeric IDs)
#docker ps -aq (List all containers numeric IDs)
#docker rmi<imgID> (Remove specific Image)
#docker rm <cID> (Remove specific container)
#docker rmi $(docker images -aq) (Remove all images)
#docker rm $(docker ps -aq) (Remove all container)
Use -f option to forcefully removal of the image

Docker Port Mapping :Published ports

By default, when you create a container, it does not publish any of its ports to the outside world. To make a port available to services outside of Docker, or to Docker containers which are not connected to the container's network, use the --publish or -p flag. This creates a firewall rule which maps a container port to a port on the Docker host.

Flag value	Description
-p 8080:80	**Map TCP port 80 in the container to port 8080 on the Docker host**.
-p 8080:80/udp	Map UDP port 80 in the container to port 8080 on the Docker host.

#docker run -d -p 8099:80 httpd

Here we have mapped host machine port 8099 to container port80

Access url from your local browser http://<public_ip>:8099

#docker run -d -p 8080:8080 tomcat

Similarly try for all CICD tools and access it from your local browser using port mapping.

Docker Volume

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

Docker volumes are very useful when we need to persist data in Docker containers or share data between containers.

Volumes are important because when a Docker container is destroyed, it's entire file system is destroyed too. So if we want to keep this data in some way, it is necessary that we use Docker volumes.

Docker volumes are attached to containers during a docker run command by using the -v flag

#docker run -i -t -p 80:80 **-v /root/mydata:/var/www/html** ubuntu:16.04 /bin/bash

Here we have shared host os volume path /root/mydata to the container directory /var/www/html

Dockerfile

Docker can build images automatically by reading the instructions from a Dockerfile.
A Dockerfileis a text document that contains all the commands a user could call on the command line to assemble an image (https://docs.docker.com/engine/reference/builder/)

- .**dockerignore-** To increase the build's performance, exclude files and directories by adding a .dockerignore file to the context directory
- **FROM** - Sets the Base Image for subsequent instructions.
- **RUN** - execute any commands in a new layer on top of the current image and commit the results.
- **CMD** - provide defaults for an executing container.
- **EXPOSE** - informs Docker that the container listens on the specified network ports at runtime. (NOTE: does not actually make ports accessible.)
- **ENV** - sets environment variable.

- **ADD** - copies new files, directories or remote file to container. Invalidates caches.
- **COPY** - copies new files or directories to container.
- **ENTRYPOINT** - configures a container that will run as an executable.
- **VOLUME** - creates a mount point for externally mounted volumes or other containers.
- **USER** - sets the user name for following RUN / CMD / ENTRYPOINT commands.
- **WORKDIR** - sets the working directory.
- **ARG** - defines a build-time variable.
- **ONBUILD** - adds a trigger instruction when the image is used as the base for another build.
- **STOPSIGNAL** - sets the system call signal that will be sent to the container to exit.
- **LABEL** - apply key/value metadata to your images, containers, or daemons.

Name of the should be exactly Dockerfile
vi Dockerfile

```
FROM ubuntu
RUN apt-get update -y
RUN apt-get install apache2 -y
USER root
CMD ["echo", "HEllo"]
```

docker build -t myapche:3 .

#docker images

docker run myapche:3

docker run -it myapche:3 /bin/bash

Using volume in Dockerfile
https://stackoverflow.com/questions/25311613/docker-mounting-volumes-on-host

vi Dockerfile

```
FROM ubuntu
RUN apt-get update -y
```

```
RUN apt-get install apache2 -y
COPY index.html /var/www/html/
EXPOSE 80
RUN mkdir /mydir
RUN echo "Hello World" > /mydir/hello.txt
VOLUME /mydir
USER root
CMD ["echo", "HEllo"]
```

For Dockerfile with different file name
#vi myDocker/mydfile

```
FROM tomcat
ADD https://tomcat.apache.org/tomcat-7.0-
doc/appdev/sample/sample.war /usr/local/tomcat/webapps
RUN bin/catalina.sh stop
EXPOSE 8080
CMD ["catalina.sh", "run"]
```

docker build -t mytomcat:1.2 -f myDockerFile/mydfile .
#docker run mytomcat:1.2
docker run -it -d -p 8080:8080 mytomcat:1.2

Docker Base Images

- scratch – this is the ultimate base image and it has 0 files and 0 size.
- busybox – a minimal Unix weighing in at 2.5 MB and around 10000 files.
- debian:jessie – the latest Debian is 122 MB and around 18000 files.
- alpine:latest – Alpine Linux, only 8 MB in size and has access to a package repository

Difference between ADD and COPY and RUN
See above commands

Difference between CMD and ENTRYPOINT
http://www.johnzaccone.io/entrypoint-vs-cmd-back-to-basics/

The difference is ENTRYPOINT is that unlike CMD, the command and parameters are not ignored when Docker container runs with command line parameters. (There is a way to ignore ENTTRYPOINT, but it is unlikely that you will do it.)

WEB SERVER EXAMPLE
##

Dockerfile to build a apache2 web server

```
################################################################################
FROM ubuntu
RUN apt-get update
RUN apt-get install apache2 -y
RUN apt-get install apache2-utils -y
RUN apt-get clean
EXPOSE 80
CMD ["apache2ctl","-D","FOREGROUND"]
################################################################################
```

Description of the above commands -

Ubuntu is our base image in which we are launching our server.
In the second line, the apt-get update command is used to update all the packages on Ubuntu.
In the third line, we are installing apache2 on our image.
In the fourth line, we are installing all the necessary utility Apache packages.
In the fifth line, the apt-get clean command cleans all the unnecessary files from the system.
In the sixth line, the EXPOSE command is used to expose the port 80 of Apache in the container.
The last command is used to run apache2 in the background.

```
################################################################################
```

Execute the below commands -

```
#to build the custom webserver image
docker build -t="mywebserver" .

#to create the container from the above built image & map the ports
docker run -d -p 80:80 mywebserver

#to access the web server default page
Access it via <VM public ip> OR <VM public ip>:80
################################################################################
```

APP SERVER EXAMPLE

```
################################################################################
```

Dockerfile to build a tomcat app server

```
################################################################################
FROM tomcat:8.0-alpine
LABEL maintainer="cloud.junction.in.3@gmail.com"

ADD https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war
/usr/local/tomcat/webapps/

EXPOSE 8080
```

```
CMD ["catalina.sh", "run"]
```
##
Description of the above commands -

tomcat:8.0-alpine is our base image in which we are launching our server.
In the second line, MAINTAINER is provided who will create and manage the container on tomcat.
In the third line, ADD is used to copy files into the container from your host.
In the fourth line, the EXPOSE command is used to expose the port 8080 of tomcat in the container.
The last command CMD will be executed at the time of container creation.
##
Execute the below commands -

#to build the custom webserver image
```
docker build -t myappserver .
```

#to create the container from the above built image & map the ports
```
docker run -d -p 8080:8080 myappserver
```

#to access the app server default page
Access it via <VM public ip> OR <VM public ip>:8080
##

Docker Network

One of the reasons Docker containers and services are so powerful is that you can connect them together or connect them to non-Docker workloads. Docker containers and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not. Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.

docker network connect	Connect a container to a network
docker network create	Create a network
docker network disconnect	Disconnect a container from a network
docker network inspect	Display detailed information on one or more networks
docker network ls	List networks
docker network prune	Remove all unused networks
docker network rm	Remove one or more networks

Network Drivers

Docker's networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

docker info (# docker info --format '{{json .Plugins.Network}}' | jq)

• **bridge**: The default network driver. Bridge networks are usually used when your applications run instandalone containers that need to communicate.

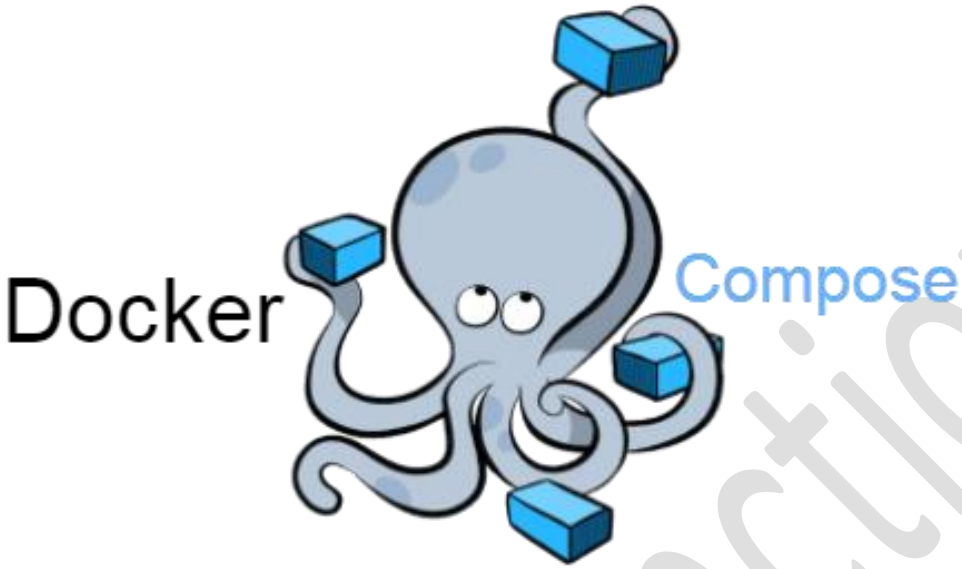
• **host**: For standalone containers, remove network isolation between the container and the Docker host,and use the host's networking directly.

• **overlay**: Overlay networks connect multiple Docker daemons together and enable swarm services tocommunicate with each other.

• **macvlan**: Macvlan networks allow you to assign a MAC address to a container, making it appear as aphysical device on your network. Using the macvlan driver is sometimes the best choice when dealingwith legacy applications that expect to be directly connected to the physical network, rather than routedthrough the Docker host's network stack.

• **none**: For this container, disable all networking. Usually used in conjunction with a custom network driver.

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 and defining services, networks and volumes.. Then, with a single command, you create and start all the services from your configuration.

Generally, docker-compose file may consist of following parts in yml:

version:
networks:
services:
volumes:
configs:

Installation

#apt install docker-compose
docker-compose --version

Using Compose is basically a three-step process:

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.

2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Run docker-compose up and Compose starts and runs your entire app.

Command Reference
https://docs.docker.com/compose/compose-file/

Build stack app
https://docs.docker.com/compose/gettingstarted/

Practical
1) Using nginx to serve content directly out of a redis cache

- Create a DockerCompose directory
- touch docker-compose.yml
- nano docker-compose.yml

```
version: '3'
services:
  web:
    image: nginx
  database:
    image: redis
```

- check the validity of the file by docker-compose config
- Run docker-compose.yml file by command docker-compose up -d
- docker images
- docker ps
- Bring down application by command docker-compose down

- try exposing the port for nginx inside the docker-compose.yml file under
  ports:
  - "9090:80"

- run docker-compose config
- run docker-compose up -d
- open the port 9090 in the security group inbound rule
  & access in the browser using <ip>:9090

How to scale services <using the scale parameter option>

- docker-compose --help
- docker-compose up -d --scale database=4
- docker ps OR docker-compose ps

- docker-compose down
- docker ps

2) Sample maven web app

- Install Java
- Install maven
- google " Maven in 5 Minutes" & visit - http://people.apache.org/~jvanzyl/maven-3.1.1/guides/getting-started/maven-in-five-minutes.html

- copy the command -

```
mvn archetype:generate -DgroupId=com.mycompany.app -
DartifactId=my-webapp -DarchetypeArtifactId=maven-archetype-
webapp -DarchetypeVersion=1.4 -DinteractiveMode=false
```

cd my-webapp
ls

mvn clean install

So the above command validates, compiles, tests(if any), packages, performs the integration-tests(if any), verifies & install

- **install**: install the package into the local repository, for use as a dependency in other projects locally
- **deploy**: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects &
- **clean**: cleans up artifacts created by prior builds

- touch Dockerfile

```
FROM tomcat:8.0-alpine
ADD target/my-webapp.war /usr/local/tomcat/webapps
EXPOSE 8080
CMD ["catalina.sh", "run"]
```

- docker build -t mywebapp .
- docker run -p 80:8080 mywebapp OR docker run -it -d -p 80:8080 mywebapp
- docker start <container-id>
- Open the port 80 on the EC2 SG
- <EC2 public ip>:80

- create a docker-compose.yml in the same directory

vi docker-compose.yml

```
version: '3'
services:
    web:
        build: .
        ports:
            - '8080:8080'
        volumes:
            - './target:/usr/local/tomcat/webapps'
        depends_on:
            - mydb
    mydb:
       image: mysql
       environment:
            MYSQL_ROOT_PASSWORD: test
            MYSQL_DATABASE: test
            MYSQL_USER: test
            MYSQL_PASSWORD: test
```

This Compose file defines two services, web and mydb.

The web service:
Uses an image that's built from the Dockerfile in the current directory.Forwards the exposed port 8080 on the container to port 8080 on the host machine.

The mydb service:
uses a public mysql image pulled from the Docker Hub registry.

root@docker:~/my-webapp# ls -ll

-rw-r--r-- 1 root root 130 Mar 6 14:36 Dockerfile
-rw-r--r-- 1 root root 404 Mar 6 15:26 docker-compose.yml
-rw-r--r-- 1 root root 2207 Mar 6 11:50 pom.xml
drwxr-xr-x 3 root root 4096 Mar 6 11:50 src
drwxr-xr-x 4 root root 4096 Mar 6 15:13 target

#docker-compose up -d

Access http://<public_ip>:8080/my-webapp

#docker-compose down

Now change some code (src/main/webapp/index.jsp) and rebuild it
got to my-webapp & run "mvn clean install" & then
#docker-compose up (Rebuild app)

#docker-compose ps
#docker-compose run mydb cat /etc/os-release
#docker-compose run mydbenv
#docker-compose run webenv
#docker-compose down –volumes
You can bring everything down, removing the containers entirely, with the down command.
Pass --volumes to also remove the data volume used by the Redis container:

Compose wordpress site

#mkdir compose_wordpress && cd compose_wordpress
vi docker-compose.yml

```
version: '3'

services:
  # Database
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
    networks:
      - wpsite
    ports:
      - 44006:3306
  # phpmyadmin
  phpmyadmin:
    depends_on:
      - db
    image: phpmyadmin/phpmyadmin
    restart: always
    ports:
      - '8080:80'
    environment:
      PMA_HOST: db
      MYSQL_ROOT_PASSWORD: password
```

```
    networks:
      - wpsite
  # Wordpress
  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - '8000:80'
    restart: always
    volumes: ['./:/var/www/html']
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
    networks:
      - wpsite
networks:
  wpsite:
volumes:
  db_data:
```

The docker volume db_data persists any updates made by WordPress to the database.

#docker-compose up -d
try accessing wordpress & phpmyadmin using web browser

Compose file for CICD installation setup

Minimum 4.5 GB RAM is required

#vi docker-compose.yml

```
version: '3'
networks:
prodnetwork:
    driver: bridge
volumes:
  nexus-data:
jenkins-data:
services:
  nexus:
    image: sonatype/nexus3
    restart: always
    ports:
      - "18081:8081"
```

```
    networks:
      - prodnetwork
    volumes:
      - nexus-data:/nexus-data
jenkins:
    image: jenkins
    restart: always
    ports:
       - "18080:8080"
    networks:
      - prodnetwork
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /usr/bin/docker:/usr/bin/docker
      - jenkins-data:/var/lib/jenkins/
depends_on:
      - nexus
      - sonar
    environment:
      - NEXUS_PORT=8081
      - SONAR_PORT=9000
      - SONAR_DB_PORT=5432
sonardb:
    networks:
      - prodnetwork
    restart: always
    image: postgres:9.6
    ports:
      - "5432:5432"
    environment:
     - POSTGRES_USER=sonar
     - POSTGRES_PASSWORD=sonar
  sonar:
    image: sonarqube
    restart: always
    ports:
     - "19000:9000"
     - "19092:9092"
    networks:
      - prodnetwork
depends_on:
      - sonardb
    environment:
     - SONARQUBE_JDBC_URL=jdbc:postgresql://sonardb:5432/sonar
     - SONARQUBE_JDBC_USERNAME=sonar
     - SONARQUBE_JDBC_PASSWORD=sonar
  tomcat:
    image: tomcat
    restart: always
```

```
    ports:
     - "18090:8080"
    networks:
      - prodnetwork
```

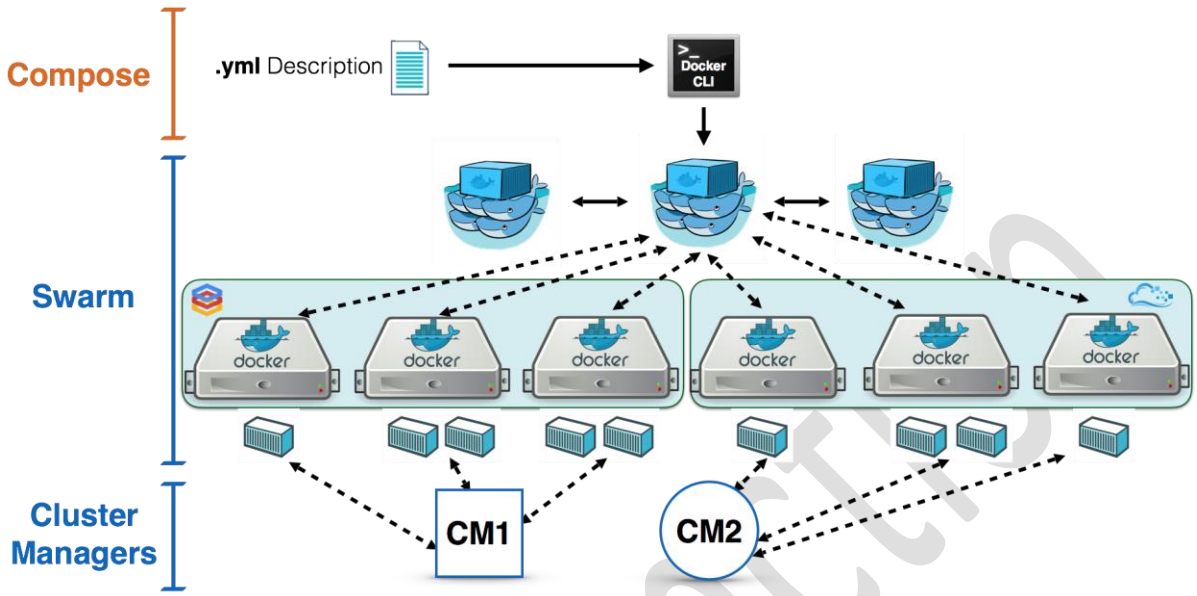
#docker-compose up -d
#docker-compose ps

Docker-compose Lifecycle commands:
create - Create services
start - Start services
stop - Stop services
restart - Restart services
kill - Kill containers
pause - Pause services
unpause - Unpause services
up [-d] - Create and start containers [detached mode]
down - Stop and remove containers, networks, images, and volumes

Info, Logs:
ps - List containers
port - Print the public port for a port binding
top - Display the running processes
logs - View output from containers
Build and Validate:
build - Build or rebuild services
config - Validate and view the Compose file

Working with Registry:
pull - Pull service images
push - Push service images

Docker swarm



The cluster management and orchestration features embedded in the Docker Engine are built using swarmkit. Swarmkit is a separate project which implements Docker's orchestration layer and is used directly within Docker.

A swarm consists of multiple Docker hosts which run in swarm mode and act as managers (to manage membership and delegation) and workers (which run swarm services). A given Docker host can be a manager, a worker, or perform both roles. When you create a service, you define its optimal state (number of replicas, network and storage resources available to it, ports the service exposes to the outside world, and more). Docker works to maintain that desired state

For instance, if a worker node becomes unavailable, Docker schedules that node's tasks on other nodes. A *task* is a running container which is part of a swarm service and managed by a swarm manager, as opposed to a standalone container.

A key difference between standalone containers and swarm services is that only swarm managers can manage a swarm, while standalone containers can be started on any daemon. Docker daemons can participate in a swarm as managers, workers, or both.

Nodes

A **node** is an instance of the Docker engine participating in the swarm. You can also think of this as a Docker node. You can run one or more nodes on a single physical computer or cloud server, but production swarm deployments typically include Docker nodes distributed across multiple physical and cloud machines.

To deploy your application to a swarm, you submit a service definition to a **manager node**. The manager node dispatches units of work called tasks to worker nodes.

Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm. Manager nodes elect a single leader to conduct orchestration tasks.

Worker nodes receive and execute tasks dispatched from manager nodes. By default manager nodes also run services as worker nodes, but you can configure them to run manager tasks exclusively and be manager-only nodes. An agent runs on each worker node and reports on the tasks assigned to it. The worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker.

Services and tasks

A **service** is the definition of the tasks to execute on the manager or worker nodes. It is the central structure of the swarm system and the primary root of user interaction with the swarm.

When you create a service, you specify which container image to use and which commands to execute inside running containers.

In the **replicated services** model, the swarm manager distributes a specific number of replica tasks among the nodes based upon the scale you set in the desired state.

For **global services**, the swarm runs one task for the service on every available node in the cluster.

A **task** carries a Docker container and the commands to run inside the container. It is the atomic scheduling unit of swarm. Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale. Once a task is assigned to a node, it cannot move to another node. It can only run on the assigned node or fail.

Main point:

- It allows to connect multiple hosts with Docker together.
- It's relatively simple. Compared with Kubernetes, starting with Docker Swarm is really easy.
- High availability – there are two node types in cluster: master and worker. One of masters is the leader. If current leader fails, other master will become leader. If worker host fails, all containers will be rescheduled to other nodes.
- Declarative configuration. You tell what you want, how many replicas, and they'll be automatically scheduled with respect to given constraints.
- Rolling updates – Swarm stores configuration for containers. If you update configuration, containers are updated in batches, so service by default will be available all the time.
- Build-in Service discovery and Load balancing – similar to load balancing done by Docker-Compose. You can reference other services using their names, it doesn't matter where containers are stored, they will receive requests in a round-robin fashion.
- Overlay network – if you expose a port from a service, it'll be available on any node in cluster. It really helps with external load balancing.

When to consider Docker Swarm

Here are 5 questions that you need to answer, before considering move to Docker Swarm:

- Do you need scaling beyond one host? It's always more complicated than single server, maybe you should just buy a better one?
- Do you need high availability?
- Are your containers truly stateless?
- Do you have log aggregation system, such as ELK stack (this applies to all distributed system)?
- Do you need advanced features, available in more mature solutions?

Practical

Open protocols and ports between the hosts

The following ports must be available. On some systems, these ports are open by default.

- TCP port 2377 for cluster management communications
- TCP and UDP port 7946 for communication among nodes
- UDP port 4789 for overlay network traffic

Activate swarm mode on manager node

- docker swarm init --advertise-addr <ip-addr>

Keep in mind that you can have a node join as a worker or as a manager. At any point in time, there is only one LEADER and the other manager nodes will be as backup in case the current LEADER opts out.

##

Assign a static hostname to each node

1. **Use vim to open the /etc/hosts file.**

sudo vim /etc/hosts

2. **Update the /etc/hosts file to include your persistent hostname for localhost, similar to the following:**

127.0.0.1 localhost persistent-hostname

Save and exit the vim editor.

3. **Run the hostnamectl command and specify the new hostname. Again, replace the persistent-hostname with the new hostname.**

sudo hostnamectl set-hostname <new hostname>

4. **Restart or reboot the EC2 instance, run the Linux hostname command without any parameters to verify that the hostname change persisted.**

hostname

##

Configuration required before setting up the docker swarm

Open all the below ports on the EC2 SG for the manager & the worker nodes

On manager node:

Port No.	Manager Node	Worker Node
22 (TCP)	Yes	Yes
2376 (TCP)	Yes	Yes
2377 (TCP)	Yes	No
7946 (TCP)	Yes	Yes
7946 (UDP)	Yes	Yes
4789 (UDP)	Yes	Yes

Refer the below for more info: -

https://www.digitalocean.com/community/tutorials/how-to-configure-the-linux-firewall-for-docker-swarm-on-ubuntu-16-04

Manager-node# docker swarm init

Add the worker1 node as a worker in the swarm using the command generated out of docker swarm init on manager node to add a worker node

After running the command you will see

```
This node joined a swarm as a worker.
Do the same for the remaining 5 worker nodes
```

check

- `docker node ls` <on manager node>

- `docker info` <on worker node>

- you will see the swarm mode active

Manager-node# docker node ls

Manager-node# docker info

Manager-node# docker service create --name webserver -p 80:80 httpd

Once it reads 1/1 under REPLICAS, it's running. If it reads 0/1, it's probably still pulling the image.

Manager-node# docker service ls

Manager-node# docker ps -a

Manager-node# docker service inspect --pretty webserver (to display the details about a service in an easily readable format

OR

docker service inspect webserver (to return the service details in json format)

Manager-node# docker service scale webserver=2

Manager-node# docker service ps webserver

To check the high availability, stop docker process on worker node that hosts app

Node1 or Node2# service docker stop

Manager-node# docker service ps webserver

Manager-node# docker service update --replicas 3 webserver

#docker swarm leave --force (To leave the cluster)

Manager-node# docker service rm webserver (to remove the service)

Manager-node#

Drain a node on the swarm

- docker service create --replicas 3 --name redis --update-delay 10s redis:3.0.6

- docker node update --availability drain worker1
- docker node inspect --pretty worker1
- docker service ps redis
- docker node update --availability active worker1
- docker node inspect --pretty worker1
- docker service ps redis

docker stack deploy

mkdir docker-stack
cd dokcer-stack

curl -O https://raw.githubusercontent.com/docker/example-voting-app/master/docker-stack.yml

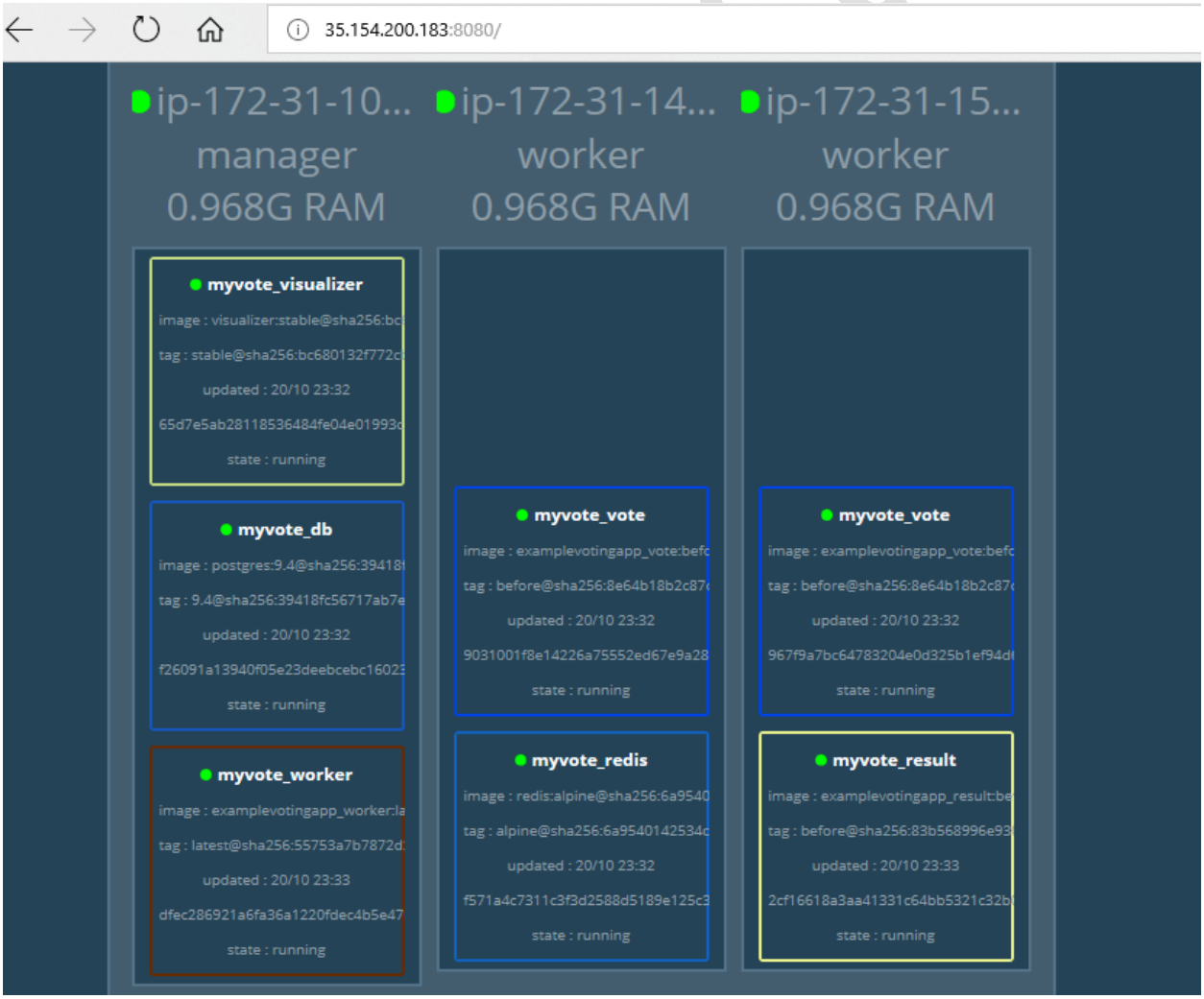
#vi deploy-stack.yml

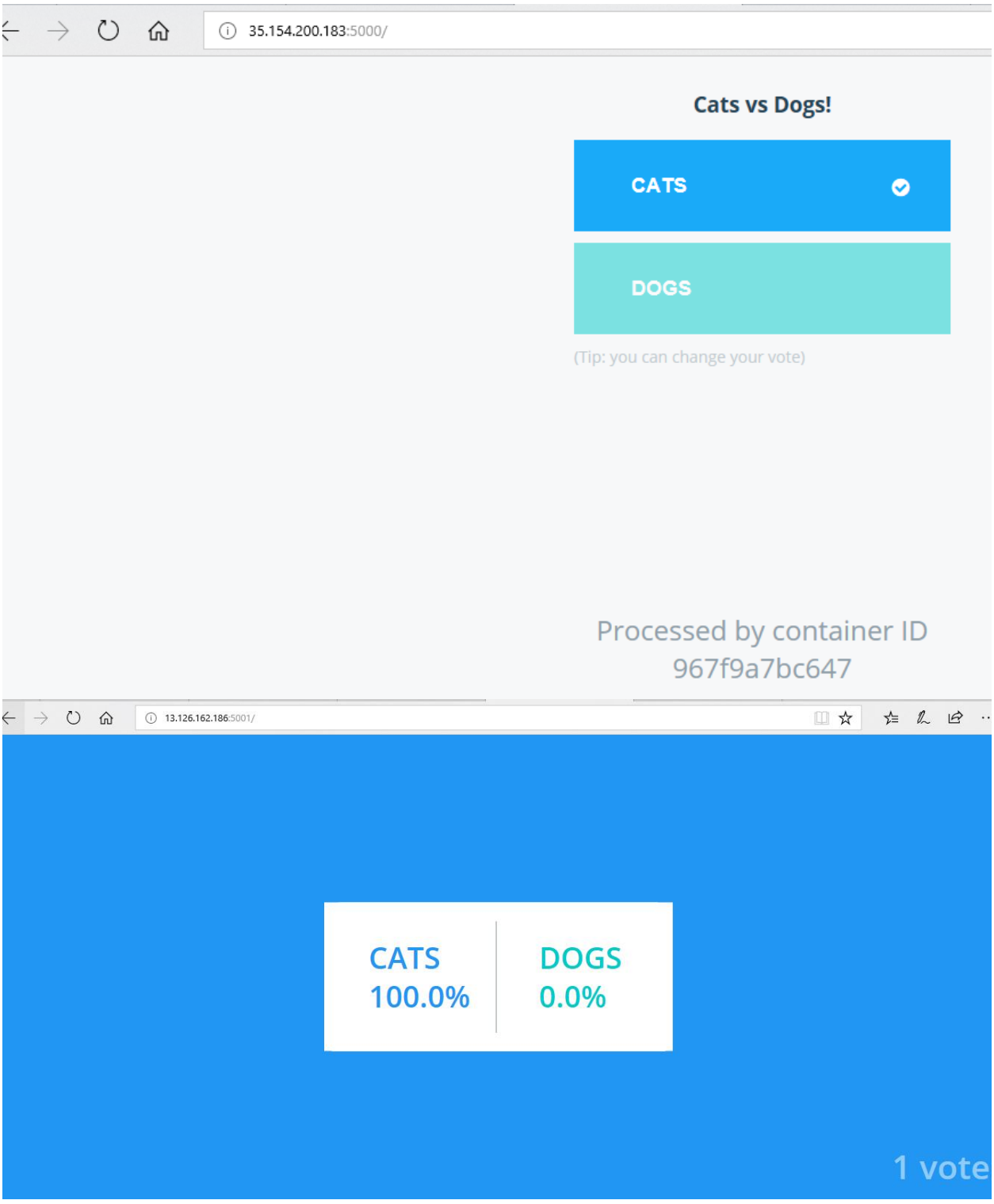
```
version: "3"
services:

redis:
    image: redis:alpine
    ports:
      - "6379"
    networks:
      - frontend
deploy:
      replicas: 1
update_config:
        parallelism: 2
        delay: 10s
restart_policy:
        condition: on-failure
db:
    image: postgres:9.4
    volumes:
      - db-data:/var/lib/postgresql/data
    networks:
      - backend
deploy:
      placement:
        constraints: [node.role == manager]
  vote:
    image: dockersamples/examplevotingapp_vote:before
```

```
    ports:
      - 5000:80
    networks:
      - frontend
depends_on:
      - redis
deploy:
      replicas: 2
update_config:
        parallelism: 2
restart_policy:
        condition: on-failure
  result:
    image: dockersamples/examplevotingapp_result:before
    ports:
      - 5001:80
    networks:
      - backend
depends_on:
      - db
deploy:
      replicas: 1
update_config:
        parallelism: 2
        delay: 10s
restart_policy:
        condition: on-failure

  worker:
    image: dockersamples/examplevotingapp_worker
    networks:
      - frontend
      - backend
    deploy:
      mode: replicated
      replicas: 1
      labels: [APP=VOTING]
restart_policy:
        condition: on-failure
        delay: 10s
max_attempts: 3
        window: 120s
      placement:
        constraints: [node.role == manager]

  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
```

```
stop_grace_period: 1m30s
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]

networks:
  frontend:
  backend:

volumes:
db-data:
```

#docker stack deploy -c docker-stack.ymlmyvote
docker stack ls
docker stack rm myvote





Access on browser
<public_ip>:8080
<public_ip>:5000
<public_ip>:5001

Monitoring Containers

What you monitor?

Image Count - The count of active and inactive images stored locally by Docker.

Image Size - The total disk usage of active and reclaimable images stored locally.

Container Count - The count of containers, segragated by state: created, running, paused, restarting, removing, exited and dead.

Container Size - The total size of disk used by containers, with used and reclaimable parts tracked separately.

Volume Count - The count of active and inactive volumes maintained by this Docker daemon.

Volume Size - The total size of disk used by volumes, with used and reclaimable parts tracked separately.

CPU Usage - The percentage of host CPU used by all currently running containers.

Memory Usage - The amount of host memory used by all currently running containers.

Network Bandwidth - The transmit and receive bandwidth of host network used by all currentlyrunning containers.

Disk Throughput - The disk read and write throughput (bytes/second) used by all currently running containers.

Process Count - The total number of processes running across all containers currently.

Monitoring Containers with cAdvisor

cAdvisor is another monitoring tool for Docker containers, offered by Google and having a native support for Docker containers. It consists of a single shipped container, that you can run and access via a graphical interface showing the attached statics of our dockized application. This container can collect, process, aggregate and export information related to the running containers.

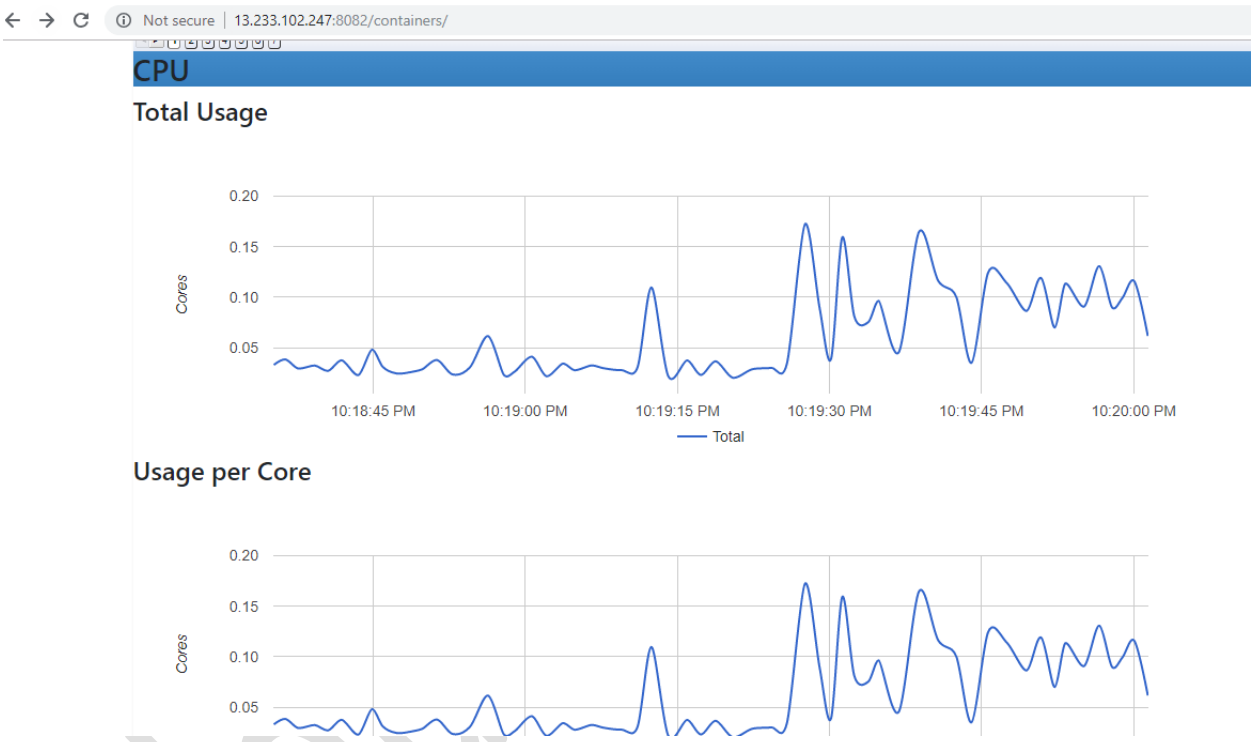
Installation
#mkdirmycadvisor&& cd mycadvisor

#vi docker-compose.yml

```
version: "3.6"
services:
  web:
    image: google/cadvisor:latest
    ports:
     - 0.0.0.0:8082:8080
    volumes:
     - /:/rootfs:ro
     - /var/run:/var/run:rw
     - /sys:/sys:ro
     - /var/lib/docker/:/var/lib/docker:ro
     - /dev/disk/:/dev/disk:ro
```

docker-compose up -d
cAdvisor will connect itself directly to into the Docker daemon running on our host machine so we can start visualizing the related metrics of the running container, including cAdvisor too as a single container.

cAdvisor displays the graphs related to the CPU usage, Memory usage, Network input/output and disk space utilization. The figure below illustrates an overview of the dashboard displayed by cAdvisor.



For other monitoring tools and comparisons read
https://code-maze.com/top-docker-monitoring-tools/