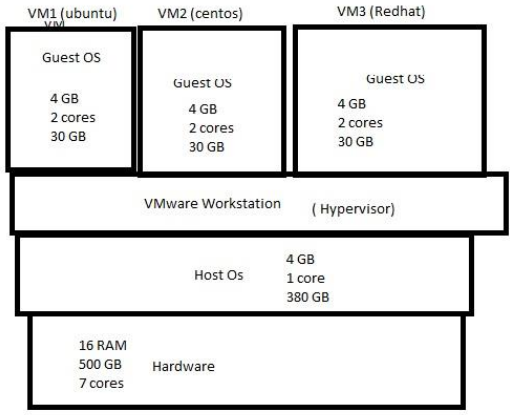
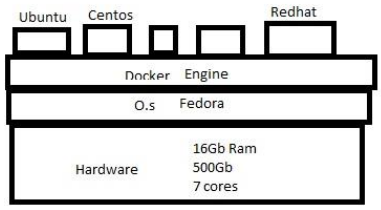
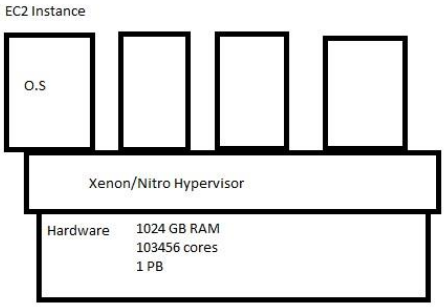
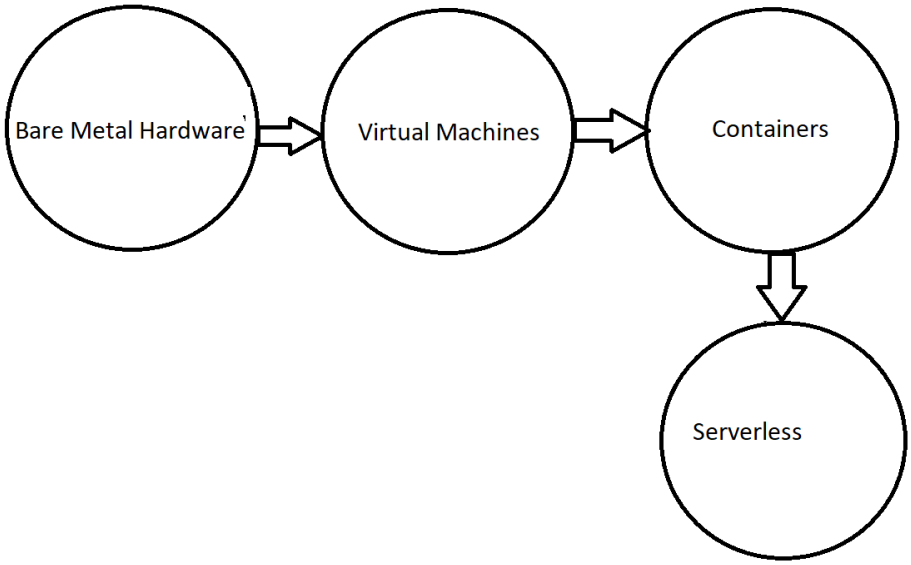
DOCKER







- Docker is an open-source centralized platform designed to create, deploy, and run applications.
- Docker uses "**container**" on the host OS to run applications. It allows applications to use the same Linux Kernel as a system on the host computer, rather than creating a whole virtual OS.
- We can install Docker on any Operating System but Docker engine runs natively on Linux distribution.
- Docker written in **'GO'** language.
- Docker is a tool that performs OS level virtualization, also known as '**Containerization**'.
- Before Docker many users faced the problem that the particular code is running in a developer system but not in the user's system.
- Docker was first released in March 2013, is developed by Soloman Hykes & Sebastean pahl.
- Docker is a set of "Platform as a Service" (PaaS) that uses Operating System virtualization whereas VMWare uses hardware level virtualization.
- Docker uses or downloads 5% of OS file remaining will be taken from host OS.



Advantages of Docker:

- No pre-allocation of resources like RAM, Processor, Storage.
- Containerization Image (CI) efficiency → Docker enables you to build a container image and use that same image across every step of the deployment process.
- Less cost
- Docker is lite in weight.
- Docker can run on physical hardware or virtual hardware or on Cloud environment.
- We can reuse the Docker Image.
- It will take very less time to create a **container**.

Run

Image ----> Container

- Image → Container (running)

Disadvantages of Docker:

- Docker is not a good solution for applications that requires rich GUI.
- It is difficult to manage large amount of containers.
- Docker does not provide cross platform compatibility, means if an application is designed to run in a docker container on windows, then it cannot run on the Linux environment or vice-versa.
- Docker is suitable when the development OS and testing OS are same, if the OS is different, we should use VMware/Virtualization.
- NO solution for data recovery and backup.

The Rise of MicroServices(Microservice Architectures) :

- Decomposing of monolithic applications into independently deployable and scalable applications are treated as "***microservices***".
- Allows for much quicker iteration, automation and overall agility.
- Start fast, fail fast and recover fast are the advantages of microservices architecture.

Docker Container Benefits:

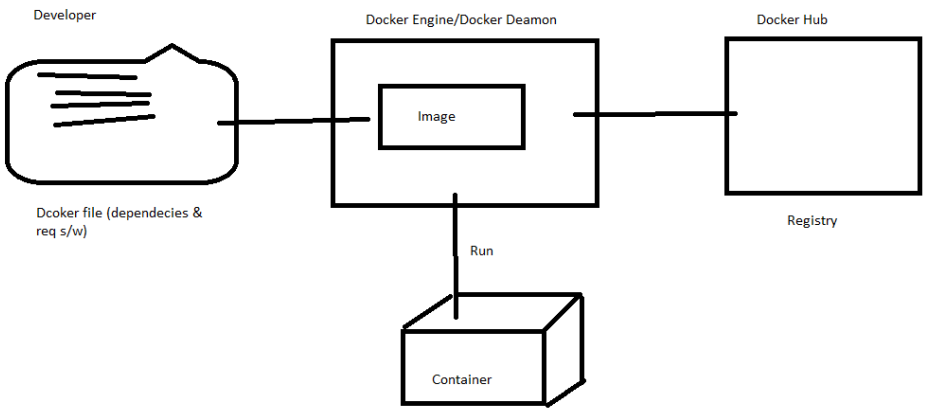
- Portable runtime application environment.
- Package application and dependencies in a single immutable artifact.
- Run different application versions (different dependencies) simultaneously.
- Faster development and deployment cycles.
- Better resource utilization and efficiency.

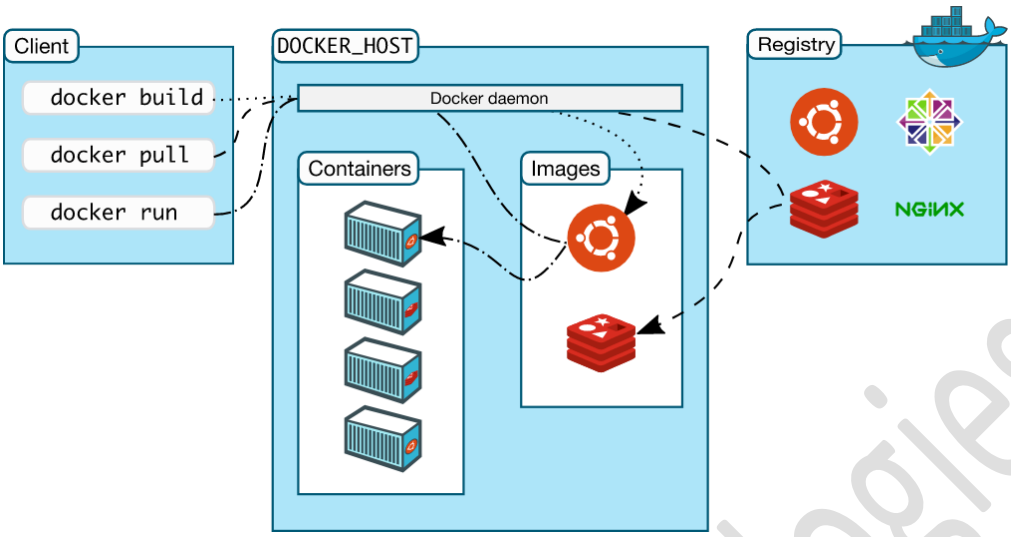
Docker Images: Where the magic happens

- A read only template with instructions for creating a docker container.
- Often, an image is based on another image with some additional customization.
- Create your own images via a docker file.
- We can use images created by others and published to a public or private registry.

Docker Images: File System Layers

- To build your own image we can create a docker file with simple syntax for defining how to create an image and run it
- Each instruction in the docker file creates a layer.
- Only modified layers gets rebuilt
- Image composed read only layers
- Containers gets a thin read write layers





Docker Eco System:

1. Docker Client
2. Docker Daemon/Engine
3. Docker Hub
4. Docker Images
5. Docker Compose

Docker Daemon/Engine:

- Docker Daemon runs on the host OS.
- It is responsible for running containers to manage docker services.
- Docker Daemon can communicate with other Docker services (like Docker Client, Docker Hub, etc).

Docker Client:

- Docker users can interact with docker Daemon through a Docker Client (CLI).
- Docker Client uses commands and Rest-API to communicate with the Docker Daemon.
- When a client runs any server command on a Docker Client terminal, the client terminal sends commands to the docker daemon.
- It is possible for docker client to communicate with more than one Daemon.

Docker Host:

- Docker host is used to provide an environment to execute and run applications.
- It contains the docker daemons, images, containers, and storages.

Docker Hub/Registry:

- Docker Registry manages and stores the docker images.
- There are two types of registries in the docker.
    1. Public Registry: Public Registry is also called as "***Docker Hub***"
    2. Private Registry: It is a local repository to enterprise where we can share images within the enterprise.

Docker Images:

- Docker Images are the read only binary templates used to create docker containers

(OR)

- Single file with all dependencies and configuration required to run a program as container.
- *Ways to create an Image*:
    1. Take image from docker hub.
    2. Create image from docker file.
    3. Create image from existing docker container.

Docker Container:

- Container hold the entire packages that is needed to run the application

(OR)

- In other words we can say that the image is a template and the container is the copy of that template.
- Container is like a virtual machine (but not Virtual Machine).
- Image becomes container when they run on Docker Engine.

While learning Docker we must focus on three areas

1. **Namespace**
2. **Control groups**
3. **UnionFs(File system)**

Namespaces:

- The "***pid" Namespace:*** process isolation
- The "***net" Namespace***: Managing network interfaces
- The ***"ipc" Namespace***: Managing access to ipc (Inter process communication) resource.
- The **"mnt" Namespace**: Managing file system mount points.
- The **"uts" namespace**: Isolating kernel and version identifiers (uts: unix time sharing system).

Control Groups:

- ***Resource limiting***: Groups can be set to not exceed a configured memory limit.
- ***Prioritization***: Some groups may get a larger share of CPU utilization or disk I/O thru put
- ***Accounting***: Measures a group resource usage
- ***Control***: Freezing groups of processes

UnionFs:

- ***Merging***: overlay file system branches to merge changes.
- ***Read/Write***: Branches can be read only or read/write.

Installing Docker:

- sudo su
- yum install -y yum-utils device-mapper-persistant-data lvm2

- yum install docker
- docker --version
- systemctl start docker
- systemctl status docker

```
[root@ip-172-31-29-7 ~]# ls /var/lib/docker/ -l
total 0
drwx--x--x 4 root root 120 Jun  7 18:50 buildkit
drwx--x--- 2 root root   6 Jun  7 18:50 containers
drwx------ 3 root root  22 Jun  7 18:50 image
drwxr-x--- 3 root root  19 Jun  7 18:50 network
drwx--x--- 3 root root  40 Jun  7 18:50 overlay2
drwx------ 4 root root  32 Jun  7 18:50 plugins
drwx------ 2 root root   6 Jun  7 18:50 runtimes
drwx------ 2 root root   6 Jun  7 18:50 swarm
drwx------ 2 root root   6 Jun  7 18:50 tmp
drwx------ 2 root root   6 Jun  7 18:50 trust
drwx-----x 2 root root  50 Jun  7 18:50 volumes
```

Commands helpful for running Docker (must run on Docker Client):

- To see all the Images present in your loscal machine use below command:
  ]#docker images
- To find out images in docker hub(public registry).
  ]#docker search Jenkins/<search-parameter>
- Run below command to download image from docker hub to local machine:
  ]#docker pull <image-name>
- Command to pull image, Start container, attach container and give name to the container:
  ]#docker run -it --name <container-name> <Image-name> /bin/bash
- Run below command to check Docker service is started or not:
  ]#service docker status
- Run below command to start a container:
  ]#docker start veera
- Command to get into the container:
  ]#docker attach veera
- Command to see all the containers:
  ]#docker ps -a
- Command to see only running containers:
  ]#docker ps
- Command to stop a container:
  ]#docker stop veera
- Command to delete a container:
  ]#docker rm veera

Docker diff commands:

Task: We have to create container from our own image

- First we have to create a container, so run below command:
  ]#docker run -it --name <container-name> ubuntu /bin/bash
- Once we log into the container make necessary setup required for environment and server setup. But for test purpose now we are making minor changes:
  ]#cd tmp/
  ]#touch Veera Vamsi Hareesh Vinay Sateesh
- To identify the changes in the container with the base image run below command:
  ]#docker diff <container-name>
- Command to make a new image out of a container: Container must be up and running
  ]#docker commit <container-name> <target-image-name>
- Command to create a new container using the custom image we created:
  ]#docker run -it --name <container-name> <target/custom-image> /bin/bash

Network :

How to create a Network:

Network drivers:

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

- **bridge**: The default network driver. If you don't specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone 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 to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers.

- **ipvlan**: IPvlan networks give users total control over both IPv4 and IPv6 addressing. The VLAN driver builds on top of that in giving operators complete control of layer 2 VLAN tagging and even IPvlan L3 routing for users interested in underlay network integration.

- **macvlan**: Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack.

- **none**: For this container, disable all networking. Usually used in conjunction with a custom network driver. none is not available for swarm services.

- **Network plugins**: You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors.

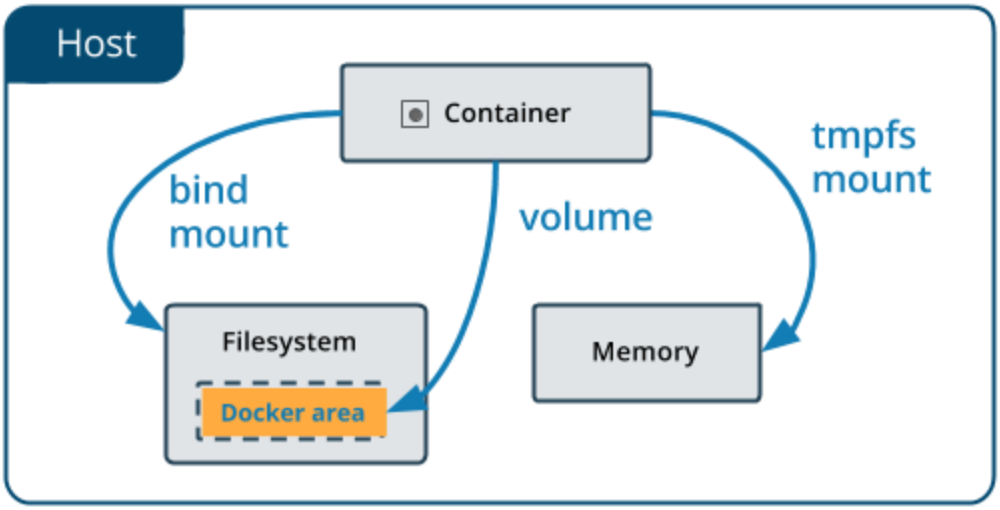
Creation of Networks:

- By default creates a network with *bridge* driver
  $docker network create <network-name>
- Command to create a network with specific driver
  $docker network create -d <driver-name> <network-name>
- Command to connect a container to the network
  $docker network connect <network-name> <container-name>
- Comment to list the networks created in the docker daemon
  $docker network ls
- Command to inspect/get details of a network in docker daemon
  $docker network inspect <network-name>
- Command to list the bridge networks linked to docker0
  $brctl show

docker run -dit --name app4-container --mount type=volume,src=my-shared-volume1,dst=/usr/local/apache2/logs/ --network=My-Network1 app4-image

Volumes:

- We have three types of volumes:
  1. Bind Mounts (Host to container)
  2. Normal volumes (Docker Daemon location to Container – Docker Area)
  3. In-Memory storage (Tmpfs)



Good usecase for Bind Mounts:

In General we should use volumes where possible bind volumes or appropriate.

- Sharing configuration files from the host machine to containers, this is how docker provides DNS resolution to containers by default. By mounting /etc/resolve.conif from the host machine into each container.
- Sharing source code or build artifacts between development environment on the docker host and a container.
- For instance you may mount a maven target directory into a container and each time you build the maven project on the host, the container gets access to the rebuilt artifacts.
- If you use docker for development this way your production docker file would copy the production ready artifacts directly into image rather than relying on a bind mount.
- When the file structure or a directory structure of the docker host is guaranteed to be consistent with the bind mounts the containers required.

Good Use cases for volumes (Docker area):

Volumes are the preferred way to persist data in docker containers and services, some use cases of volume include:

- Sharing data among multiple running containers. if you don't explicitly create it if a volume is created for first time it is mounted into a container. When that container stops or removed the volume still exists.
- Multiple volumes can mount the same volume simultaneously either read-write or read-only volumes are only removed when we explicitly remove them.
- When the docker host is not guaranteed to have a given directory or file structure. Volumes help you decouple the configuration of the docker hosts from the container run time.
- When we want to store our containers data on a remote host or a cloud provider rather than locally.
- When you need to be able to backup, restore or migrate data from one docker host to another, volumes are better choice. You can stop containers using the volume then backup the volumes directory (*/var/lib/docker/volumes/<volume-name>*).

Good Use case for tmpfs mounts:

Tmpfs mounts are best used for cases where you don't want data to persist either on the host machine or within the container. This may be for security reasons or to protect the performance of the container. When your application needs to write a large volume of non-persistent state data.

How to create a volume (only docker area volume):

- Command to create a volume in docker area:
  $docker volume create <volume-name>
- Example : docker volume create my-vol
- Command to list the volumes created in docker:
  docker volume ls
- Command to get the details of the volume:
  docker volume inspect my-vol
- Command to remove the volume in docker:
  docker volume rm my-vol
- Command to backup the volume and remove from docker:
  docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata
- When we want to restore the backup volume into the container then
  - Launch a new container and mount the volume from the dbstore container
  - Mount a local host directory as /backup
  - Pass a command that tars the contents of the dbdata volume to a backup.tar file inside our /backup directory.
- Example:
  docker run --rm --volumes-from dbstore2 -v $(pwd):/backup ubuntu bash -c "cd /dbdata && tar xvf /backup/backup.tar --strip 1"

  Note: With the backup just created, you can restore it to the same container, or another that you made elsewhere.
- Command to remove all unused volumes and free up space:
  $ docker volume prune

Docker File:

- Docker file is basically a text file which contains some set of instructions.
- Automation of docker image creation:
- **Docker Components**:
  - **FROM**: for downloading the base image, this must be added as first command in the docker file.
  - **RUN**: to execute commands, it will create a layer in the image.
  - **MAINTAINER**: author/owner/description of that file.
  - **COPY**: Copy files from ***local*** system, we need to provide source and destination for this command. (this command cannot download file from internet/remote machine)
  - **ADD**: similar to copy command but it provides a feature to download files from internet also. This extracts a file at docker site.
  - **EXPOSE**: to expose ports such as port 8080 for TOMCAT, port 80 for NGINX, etc.
  - **WORKDIR:** to set working directory for a container.
  - **CMD:** execute commands but during container creation.

- **ENTRYPOINT:** Similar to CMD but have higher priority over CMD. First commands will be executed by ENTRYPOINT only.
- **ENV:** to set Environment variables.

- **How to create a Docker File**:
  - Create a file named "Dockerfile".
  - Add instructions in "Dockerfile".
  - Build the "Dockerfile" to create an image.
  - Run image to create container.
  - Sample Dockerfile:

```
FROM ubuntu
RUN echo "Technical expert" > /tmp/testfile
```

- How to build Dockerfile as image: We must be in the dockerfile location

```
#docker build -t <image-name> .
```

- To verify the image created run below command:

```
#docker images
```

- Command to start the container from the created image:

```
#docker run -it --name <container-name> <image-name> /bin/bash
```

Docker Volume:

- Volume is simply a directory inside a container.
- Firstly, we have declare this directory as volume and then share volume.
- Even if you stop container still, we can access volume.
- Volume will be created in one container.
- You can declare a directory as a volume only while creating a container.
- You cannot create volume from existing container.
- Volume will not be included when you update an image.
- You can map volume in two ways:
  - Container to container
  - Host to Container

Benefits of Volume:

- Decoupling container from storage.
- Share volume among different containers.
- Attach volume to containers.
- On deleting container, volume doesn't get deleted.

Creating Volume from Docker File:

- Create a file with name "Dockerfile" and write below code in it:
  - *FROM ubuntu*
  - *VOLUME ["/mydrive1"]*
- Then create image from this Dockerfile
  - *docker build -t MyNewImage .*
- Now create a container from the image we created
  - *docker run -it --name Container1 MyNewImage /bin/bash*
- Now if we login to the container, we can see mydirve1 volume in it.

- Share volume between containers:
  - *docker run -it –name container2 --privileged=true --volumes-from container1 ubuntu /bin/bash*
- Now after creating container2 the mydrive1 is visible. Whatever the changes you made in mydrive1 volume, you can see in other container volume as they both are sharing the mydrive1 volume.

Creating a volume while creating container using Commands:

#docker run -it --name <container-name> -v /mydrive1 ubuntu /bin/bash

- Volume can be created using only while creating a container.

Now, if we want to share this volume from between containers

- Share volume between containers:
  - *docker run -it --name container2 --privileged=true --volumes-from Container1 ubuntu /bin/bash*
- Now check container1's shared volume can be accessed from container2.

Sharing volume between Host OS and Containers:

- Let's share /home/ec2-user directory of host OS as a volume to containers:
- Command to do the above task:

  #docker run -it --name hostCont -v /home/ec2-user:/myVolume1 --privileged=true ubuntu /bin/bash
- Now once you are in the container do
  - cd /myVolume1
  - ls
- Now we can see all the files of host system which are located in /home/ec2-user.

Some useful commands of Docker Container for Volumes:

- *docker volume ls* → list of docker volumes
- *docker volume create <volume-name>* → to create a docker volume
- *docker volume rm <volume-name>* → to remove/delete a docker volume
- *docker volume prune* → it will remove all the unused docker volumes
- *docker volume inspect <volume-name>*→ provides all the details of the volume
- *docker container inspect <container-name>* → provides all the details of the container

Docker port expose:

- For deploying a website in its webserver inside a container, we need to expose ports with host. Let's see how to make it in the below process:
- Commands:

  $sudo su
  #yum install docker -y
  #service docker start
  #docker run -td --name webserver -p 80:80 ubuntu

```
#docker ps
#docker port webserver
#docker exec -it webserver /bin/bash
Now run the commands inside container:
        #apt-get update
        #apt-get install apache2 -y
        #cd /var/www/html
        #echo "Learn technical things">index.html
        #service apache2 start
```

Example for Jenkins container launch:

```
#docker run -td --name myJenkins -p 8080:8080 jenkins
```

Difference between *docker attach* and *docker exec*:

Docker exec *creates a new process* in the container environment while *docker attach just connect the* standard input/output of the main process inside the container to corresponding standard input/output error of current terminal.

Difference between expose and publish:

Basically, we have three options:

1. Neither specify expose nor -p (publish)
2. Only specify expose
3. Specify expose and -p

Note:

1. If you specify neither expose not -p, the service in the container only be accessible from inside the container itself.
2. Whereas if you expose a port, the service in the container is not accessible from outside the docker but accessible from inside other docker containers, so this is good for inter-container communication. We need to opt publish to share with all the containers (outside world).

Process to push docker image to docker hub:

```
$sudo su
#yum update -y
#yum install docker -y
#service docker start
#docker run -it --name MyFirstContainer ubuntu /bin/bash
#docker commit MyFirstContainer MyFirstContainerImage
```

Now create an account in hub.docker.com

On the source machine

```
#docker login
```

Provide username and password

```
#docker tag <source-image> <docker-hub-id>/<target-image-name>
#docker push <docker-hub-id>/MyFirstContainerImage
```

Now we can access this image in docker hub account.

Now create another ec2 instance in the required region where we want this image to be run as container.

#docker pull <docker-hub-id>/MyFirstContainerImage
#docker run -it --name NewContainer <docker-hub-id>/MyFirstContainerImage /bin/bash

Important commands:

- Stop all running containers
  #docker stop $(docker ps -q)
- Delete all stopped containers:
  #docker rm $(docker ps -a -q)
- Delete all images from the host:
  #docker rmi -f $(docker images -q)

Docker Compose

Docker Compose is a tool for running multi-container applications on Docker defined using the Compose file format. A Compose file is used to define how the one or more containers that make up your application are configured. Once you have a Compose file, you can create and start your application with a single command: *docker compose up*.

About update and backward compatibility

Docker Compose V2 is a major version bump release of Docker Compose. It has been completely rewritten from scratch in Golang (V1 was in Python). The installation instructions for Compose V2 differ from V1. V2 is not a standalone binary anymore, and installation scripts will have to be adjusted. Some commands are different.

For a smooth transition from legacy docker-compose 1.xx, please consider installing compose-switch to translate docker-compose ... commands into Compose V2's docker compose .... . Also check V2's --compatibility flag.

Where to get Docker Compose

Windows and macOS

Docker Compose is included in Docker Desktop for Windows and macOS.

Linux

You can download Docker Compose binaries from the release page on this repository.

Rename the relevant binary for your OS to docker-compose and copy it to $HOME/.docker/cli-plugins

Or copy it into one of these folders for installing it system-wide:

- /usr/local/lib/docker/cli-plugins OR /usr/local/libexec/docker/cli-plugins
- /usr/lib/docker/cli-plugins OR /usr/libexec/docker/cli-plugins

(might require to make the downloaded file executable with chmod +x)

Installation steps:

- docker-compose installation for amaon linux:::
- mkdir -p ~/.docker/cli-plugins/
- curl -SL https://github.com/docker/compose/releases/download/v2.2.3/docker-compose-linux-x86_64 -o ~/.docker/cli-plugins/docker-compose
- chmod +x ~/.docker/cli-plugins/docker-compose
- This will be installed in root.
- $docker compose version

Quick Start

Using Docker 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. Lastly, run docker compose up and Compose will start and run your entire app.

A Compose file looks like this:

```
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
  redis:
    image: redis
```

```
[root@ip-172-31-29-7 project]# docker images
REPOSITORY     TAG       IMAGE ID       CREATED      SIZE
project_app1   latest    1be74ba11869   3 days ago   144MB
app4-image     latest    1e8e29587a71   5 days ago   144MB
app3-image     latest    bc38a1b5cce4   5 days ago   144MB
app2-image     latest    a6d298a89b6c   5 days ago   144MB
httpd          latest    acd59370d8fb   9 days ago   144MB
```

```
[root@ip-172-31-29-7 project]# tree
.
├── Application1
│   ├── Dockerfile
│   └── index.html
├── Application2
│   ├── Dockerfile
│   └── index.html
├── Application3
│   ├── Dockerfile
│   └── index.html
├── Application4
│   ├── Dockerfile
│   └── index.html
├── apps-compose.yml
└── bind-folder
    └── new-file-for-testing

5 directories, 10 files
[root@ip-172-31-29-7 project]#
```

```
version: "3.8"
services:
  app1:
    # image: app1-image
    build: /Application1/.
    ports:
      - "81:80"
    volumes:
      - my-shared-volume2:/usr/local/apache2/logs/
    networks:
      - My-Network2
  app2:
    image: app2-image
    ports:
      - "82:80"
    volumes:
      - my-shared-volume2:/usr/local/apache2/logs/
    networks:
      - My-Network2
        #  depends_on:
        # - app1
  app3:
    image: app3-image
    ports:
      - "83:80"
    volumes:
      - my-shared-volume2:/usr/local/apache2/logs/
    networks:
      - My-Network2
  app4:
```

```
    image: app4-image
    ports:
      - "84:80"
    volumes:
      - my-shared-volume2:/usr/local/apache2/logs/
    networks:
      - My-Network2

networks:
  My-Network2:
    driver: bridge

volumes:
  my-shared-volume2:
    driver: local
```

```
[root@ip-172-31-29-7 project]# docker compose --help

Usage:  docker compose [OPTIONS] COMMAND

Docker Compose

Options:
      --ansi string               Control when to print ANSI control characters ("never"|"always"|"auto") (default "auto")
      --compatibility             Run compose in backward compatibility mode
      --env-file string           Specify an alternate environment file.
  -f, --file stringArray          Compose configuration files
      --profile stringArray       Specify a profile to enable
      --project-directory string  Specify an alternate working directory
                                  (default: the path of the Compose file)
  -p, --project-name string       Project name
```

```
Commands:
  build       Build or rebuild services
  convert     Converts the compose file to platform's canonical format
  cp          Copy files/folders between a service container and the local filesystem
  create      Creates containers for a service.
  down        Stop and remove containers, networks
  events      Receive real time events from containers.
  exec        Execute a command in a running container.
  images      List images used by the created containers
  kill        Force stop service containers.
  logs        View output from containers
  ls          List running compose projects
  pause       Pause services
  port        Print the public port for a port binding.
  ps          List containers
  pull        Pull service images
  push        Push service images
  restart     Restart containers
  rm          Removes stopped service containers
  run         Run a one-off command on a service.
  start       Start services
  stop        Stop services
  top         Display the running processes
  unpause     Unpause services
  up          Create and start containers
  version     Show the Docker Compose version information

Run 'docker compose COMMAND --help' for more information on a command.
```

To start the containers/services written in a **compose** file:

$Docker compose -f compose-file.yml up -d

Compose-file:

- Written in yaml language
- Root elements
  - Version
  - Services
  - Volumes
  - Networks
- Special topics
- In YAML language commands can be written using hash (#).

Note: YAML is also a language which contains data types like integers, strings, Boolean and also include collections like maps & lists.

Compose file format	Docker Engine release
Compose specification	19.03.0+
3.8	19.03.0+
3.7	18.06.0+
3.6	18.02.0+
3.5	17.12.0+
3.4	17.09.0+
3.3	17.06.0+
3.2	17.04.0+
3.1	1.13.1+
3	1.13.0+
2.4	17.12.0+
2.3	17.06.0+
2.2	1.13.0+
2.1	1.12.0+
2	1.10.0+

Docker Compose Commands for deployment of applications:

- To create all the services given in the compose file called wordpress.yml
  $Docker compose -f wordpress.yml up -d
- To delete existing services and recreate all the services again
  $Docker compose -f wordpress.yml up -d --force-recreate
- Kill/delete all the services that are created in the compose file
  $Docker compose -f wordpress.yml down
- Kill/delete all the services that are created in the compose file, delete images and volumes that are not assigned to any containers that are created in the compose file.
  $Docker compose -f wordpress.yml down --rmi all --volumes -remove-orphans

Reference/sample project for docker compose: Quickstart: Compose and WordPress | Docker Documentation or https://docs.docker.com/samples/wordpress/

Important Links:

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

Docker Swarm

Introduction:

Containerized applications may need sometimes to be deployed on multi-host or multi-docker daemon environments, then comes the tool called '**Docker Swarm**'.

Why Swarm?

- When container applications reach a certain level of complexity/scale, you need to make use of several machines/hosts then we require some orchestration tools/products.
- Container orchestration tools/products allow you to manage multiple container hosts in concert.

Swarm Mode:

- This is a feature built into the docker engine providing native container orchestration in docker.
- Integrated cluster management within the docker engine.
- Declarative service model
- Desired state reconciliation
- Certificates and cryptographic tokens to secure the cluster
- Containers orchestration features
    - Service scaling
    - Multi-host networking
    - Resource-aware scheduling
    - Load Balancing rolling updates
    - Restart policies

Name Disambiguation:

Docker her two cluster management solutions

1. Docker Swarm Standalone
    - The first container orchestration project by Docker
    - Uses Docker API to turn a pool of docker hosts into a single virtual docker host
2. Docker Swarm mode (Swarm kit)
    - Built into a Docker engine since version 1.12
    - Docker generally recommends swarm mode

Swarm Mode concepts:

- **Swarm**: One or more docker engines running in Swarm mode.
- **Node**: Each instance of docker engine in the swarm. It is possible to run multiple nodes on a single machine.
- **Managers**: Managers accept specifications from users and drive the swarm to the specified desired state.
- **Workers**: Responsible for running the delegated work.
- **Service**: It is a specification that users submit to the managers. Declares its desired state including the networks and volumes, the number of replicas, resource constraints and other details.
- **Replicated Service**: The number of replicas for a replicated service is based on the scale desired.

- **Global Service**: Allocated one unit of work for each node in the swarm which can be useful for monitoring services.
- **Tasks**: The units of work delegated by managers to realise a service configuration. Tasks corresponds to the running containers that are replicas of the service.

Docker Swarm Mode Architecture:

1. Networking: –
    - Overlay Networks
    - Network Isolation and Firewalls
2. Service Discovery
3. Load Balancing
4. External Access

1. Networking

Overlay Network:

In Swarm mode services need to communicate with one another and the replicas of the service can be spread across multiple nodes.

- A multi-host networking in swarm is natively supported with the overlay driver.
- No need to perform any external configurations.
- You can attach a service to one or more overlay networks.
- Overlay Networks only apply to swarm services.
- Manages automatically extend overlay networks to nodes.

Network Isolation and Firewalls:

- Network Isolation and Firewalls rules apply to overlay networks just as they do for bridge networks.
- Containers within a Docker have access on all ports in the same network.
- Access is denied between containers that do not share a common network or doesn't belong to same network.
- Traffic originating inside of a docker is permitted/allowed which is not destined for a docker host.
- Traffic coming into the docker network from host is denied by default. Can communicate only through the ports mapping.

2. Service Discovery

- A Service Discovery mechanism is required in order to connect to the nodes running tasks (containers) for a service (Application).
- Docker Swarm has an integrated service discovery system based upon DNS.
- The same system is used when not running in swarm mode.
- The network can be an overlay spanning multiple hosts, but the same internal DNS system is used.
- All nodes in the network store corresponding DNS records for the network.

3. Load Balancing (Internal)

- Each individual task is discoverable with a name to ip mapping in the internal DNS.
- Request for Virtual IP (VIP) addresses are automatically load balanced across all the healthy tasks in the overlay network.
- Request → Service Discovery → Service Virtual IP → IPVS Load Balancing → Individual Tasks
- Example: Two services deployed in swarm, service A with one replica and service B with two replica. Then service A makes a request for service B the VIP of the service B is resolved by the DNS server. It is using support for IPVS, the request for the VIP address is routed to one of the two nodes running service B tasks.
- **DNS Round Robin**:
    - DNS Round robin allows you to configure the load balancing on a per service basis.
    - The DNS Server resolves a service name to individual task IP addresses by cycling through the list of IP Addresses of the nodes.
    - If you need more control, DNS RR should be used for integrating your own load balancer.

4. External Access

In Swarm mode there are two modes for publishing ports.

1. Host Mode
2. Ingress Mode

Host Mode:

- The container port is published on the host that is running the task for a service.
- If you have more tasks then available hosts task will fail to run.
- You can omit a host port to allow docker to assign an available port number in the default port range of 30,000 to 32,767.
- There isn't load balancing unless you configure it externally.

Ingress Mode:

- You have the option to load balance a published port across all tasks of a service.
- Requests are round robin load balanced across the healthy instances regardless of the node.
- It is a default service publishing mode.

Routing Mesh:

The Routing Mesh combines an overlay network and the service virtual IP.

- When you initialize a swarm the manager creates an overlay network automatically.
- Every node that joins the swarm is in the ingress network.
- When a node receives an external request it resolves the service name to VIP.
- The IPVS load balances the request to a service replica over the ingress network.
- The nodes need to have a couple of ports open
    - Port 7946 for TCP and UDP protocols to enable container network discovery.
    - Port 4789 for the UDP protocol to enable the container ingress network.

- We can add an external load balancer on top of the load balancing provided by the routing mesh.
- We need to run a version of 17.09 or greater to use routing Mesh on windows.

Docker_gwBridge:

- The Docker_gwbridge is a virtual bridge that connects the overlay networks to an individual docker daemons physical network.
- This interface provide default gateway functionality for all container attached to the network.
- Docker creates it automatically when you initialize a swarm or join a docker host to a swarm.
- It exists in the terminal of the docker host and you can see it if you list the network interfaces on your host.

Orchestration of Services on Hosts:

1. Service Placement
2. Update Behaviour
3. Rollback Behaviour

1. Service Placement:

For replicated services, the decisions need to be made by the swarm managers for where service tasks will be scheduled, or where the service will be placed.

- There are three ways you can influence where a service is placed
  a) CPU and memory reservations
  b) Placement Constraints
  c) Placement Preferences
- Global services can also be restricted to a sub-set of nodes with these conditions
- A Node will never have more than one task for a global service.

a. CPU and Memory Reservation:

- Similar to running individual containers you can declare CPU and memory reservations.
- Each service task can only be scheduled on a node with enough available CPU and memory.
- Any task that remains that stay in a pending state.
- Global services will only run on nodes that meet a given resource reservation.
- Setting memory reservations is important to avoid that the container or docker daemon to get killed by the OOM(Out Of Memory) killer.

b. Placement Constraints:

- The conditions compare node attributes to a string value.
- Builtin attributes for each node are Node.id, Node.hostname, Node.role.
- Engine labels are used to indicate things like operating system, system architecture, available drivers.
- Node labels are added for operational purposes and indicate the type of application, the data centre location, the server rack, etc.,
- All constraints provided must be satisfied by a node in order to be scheduled a service task.

c. Placement Preferences:

- They influence how tasks are distributed across appropriate nodes.
- Currently the only distribution option is "***spread***".
- Labels are used as the attributes for spreading tasks.
- Multiple placement preferences can be specified, and hierarchy of preferences is created.
- Nodes that are missing placement preferences are created as a group and receive tasks in proportion equal to all other label values.
- Placement Preferences are ignored by global services.

2. Update Behaviour:

- Swarm supports rolling updates where a fixed number of replicas are updated at a time.
  - **Update parallelism**: The number of tasks the scheduler update at a time.
  - **Update delay**: The amount of time between updating sets of tasks.
  - **Update failure action**: pause, continue or automatically rollback on failure.
- **Note**: We can set a ratio for the number of failed tasks updates to tolerate before failing a service and the frequency for monitoring for a failure.
- Flexibility is how aggressively or conservatively you rollout an update to the swarm.

3. Rollback Behaviour:

- Docker Swarm mode keeps track of the previous configuration for services.
- You may rollback manually at any given time or automatically when a update fails.
- The same options available for configuring update behaviour are available separately for configuring rollbacks.

Docker Swarm: Consistency

1. Consistency
2. Raft
3. Manager TradeOff
4. Raft Logs

1. **Consistency**:

Swarm mode can include several manager and worker nodes that provide fault tolerance and ensure high availability.

- How does swarm make decisions?
- Do nodes in the swarm share a consistent view of the cluster?
- Could one node have different view than the other?

Swarm mode can include several managers and worker nodes that provide fault tolerance and ensure high availability.

- All Managers share a consistent internal state of the entire swarm
- Workers don't share a view of the entire swarm
- Managers maintain a consistent view of the state of the cluster by using a consensus algorithm

2. **Raft Consensus**:

Raft achieve consensus by electing one manager as the leader.

- The elected leader makes the decisions for changing the state of the cluster.
- The leader accepts new service (Application) requests and service updates and how to schedule tasks (containers).
- Decisions are acted only when there is a quorum.
- Raft allows for (N-1)/2 fails and the swarm can continue operating, where N is the number of managers.
- If more managers fail the cluster state would freeze.
- If the current leader fails a new leader is elected.

3. Manager TradeOff:

More managers increase the amount of managerial traffic required for maintaining a consistent view of the cluster and the time to achieve consensus.

- You should have an ODD number of managers.
- A Single manager swarm is acceptable for development and test swarms.
- Three manager swarm can tolerate one failure. Five manager swarm can tolerate two failures.
- Distribute managers across at least three Availability Zones.
- Docker recommends a maximum of 7 managers.
- Working Manager:
    - By default managers also perform worker responsibilities.
    - Having over utilized managers can be determinantal for the performance of the swarm.
    - You can use conservative resource reservation to make sure that managers wont become starved for resources.
    - You can prevent any work from being scheduled from mangers nodes by draining them.
- Worker Node Trade Off:
    - There is not much to worry about trade off in the case of adding more workers.
    - More workers give you more capacity for running services and improve service fault tolerance.
    - More workers don't effect managers raft consensus process.
    - Node participate in weakly consistent highly scalable gossip protocol called SWIM.

4. Raft Logs:

- These Raft logs are shared with other managers to establish a quorum.
- The logs persist to disk and are stored in the raft sub-directory of your docker swarm data directory. "/var/lib/docker/swarm" on Linux by default.
- You can backup a swarm cluster by backing up the entire swarm directory.
- You can restore a new swarm from a backup by replacing the directory swarm with the backup copy.

- Swarm mode elects a leader manager and ensuring a majority of managers acknowledge swarm changes.
- This strategy comes from the raft consensus algorithm.
- There are trade offs between fault tolerance and performance.
- Raft logs persist on disk and record all changes the leader makes to a swarm.

Docker Swarm Mode: Security

1. Cluster Management
2. Data Plane
3. Secrets
4. Locking a Swarm

1. Cluster Management:

- Docker swarm mode uses PKI (Public Key Interface) to secure swarm communications and state.
- Swarm nodes encrypt all control plane communications using mutual TLS (Transport Level Security).
- Docker assigns a manager that automatically creates several resources.
- Root Certificate Authority key-pair worker-token manager-token
- When a new node joins the swarm, the manager issues a new certificate which the node uses for communication.
- New Managers also get a copy of the root CA.
- You can use an alternate CA instead of allowing Docker.
- The CA can be rotated out if required.
- CA will automatically rotate the TLS certificates of all swarm nodes.

2. Data Plane:

- You can enable encryption of overlay networks at the time of creation.
- When a traffic leaves a host and IPSEC encrypted channel is used to communicate with a destination host.
- The Swarm leader periodically regenerates and distributes the key used.
- Overlay network encryption is not supported for windows.

3. Secrets:

- The raft logs are encrypted at rest on the manager nodes.
- This protects against the intruders that gain access to the raft logs.
- Encryption is important because the swarm secrets are stored in the raft logs.
- Secrets allow you to securely store secrets that can be used by services.
- Passwords, API keys and any other information you wouldn't want to be exposed will be stored as a secrets.

4. Locking a Swarm:

- By default the keys are stored on the disk along with the raft logs, if any attacker gains access to the raft logs they could gain access to the keys.
- Swarm allows you to implement strategies where the key is never persistent to the disk with auto-lock mechanism.
- When a swarm is auto-locked, you must provide the key when starting a docker daemon.
- You may require manual interaction when a manager is started.
- You can rotate the key at any point on a disable auto-lock, so managers can restart without intervention.
- There are several security layers for managing a cluster.
- Overlay Network communication can optionally be encrypted to secure communication.
- Swarm supports sharing secrets and uses encryption to protect them.
- The keys for decrypting the logs are stored on disk by default.
- You can use auto-locking to take control of the keys.

Setting Up Swarm

Options for creating a Swarm:

1. Single Node
2. Multi Node
    I. On-prem(Bare Metal or VMs)
    II. Universal Control Plane (UCP)
    III. Public Clouds
        i. Docker for Azure, Docker for AWS, Docker for IBM Cloud
        ii. Docker Cloud

1. Single Node:

- To initialize a swarm run below command:
  Docker swarm init
- To verify the docker swarm status run below command
  Docker info | grep Swarm
- Command to leave the swarm mode
  Docker swarm leave --force
- Command that can be used in Docker Swarm:

```
[root@ip-172-31-29-7 project]# docker swarm --help

Usage:  docker swarm COMMAND

Manage Swarm

Commands:
  ca          Display and rotate the root CA
  init        Initialize a swarm
  join        Join a swarm as a node and/or manager
  join-token  Manage join tokens
  leave       Leave the swarm
  unlock      Unlock swarm
  unlock-key  Manage the unlock key
  update      Update the swarm

Run 'docker swarm COMMAND --help' for more information on a command.
```

Creating a Multi-Node Cluster:

- Create 4 EC2 Instances with Docker installed up and running in them.

  #!/bin/bash
  yum update -y
  yum install docker git tree -y
  systemctl start docker
  systemctl status docker
  mkdir -p ~/.docker/cli-plugins/
  curl -SL https://github.com/docker/compose/releases/download/v2.2.3/docker-compose-linux-x86_64 -o ~/.docker/cli-plugins/docker-compose
  chmod +x ~/.docker/cli-plugins/docker-compose
  docker swarm init
  docker info | grep Swarm
  docker node ls

- One is a Manager Leader among the 4 instances and remaining 3 would be the worker/Manager nodes. On Manager Lead run below command to get the join command:
  docker swarm join-token manager
- Copy the displayed command and run on the other three nodes. Sample:

```
[root@ip-172-31-93-168 projects]# docker swarm join-token manager
To add a manager to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-5irelimsn86m4cw9xe8xhmlko1rzaaepmzm79loyqk62ucqmnz-65q72isnk2hnfgogqhvl5mi6u 172.31.93.168:2377
```

Note: Make sure the ports are open to access between the nodes.

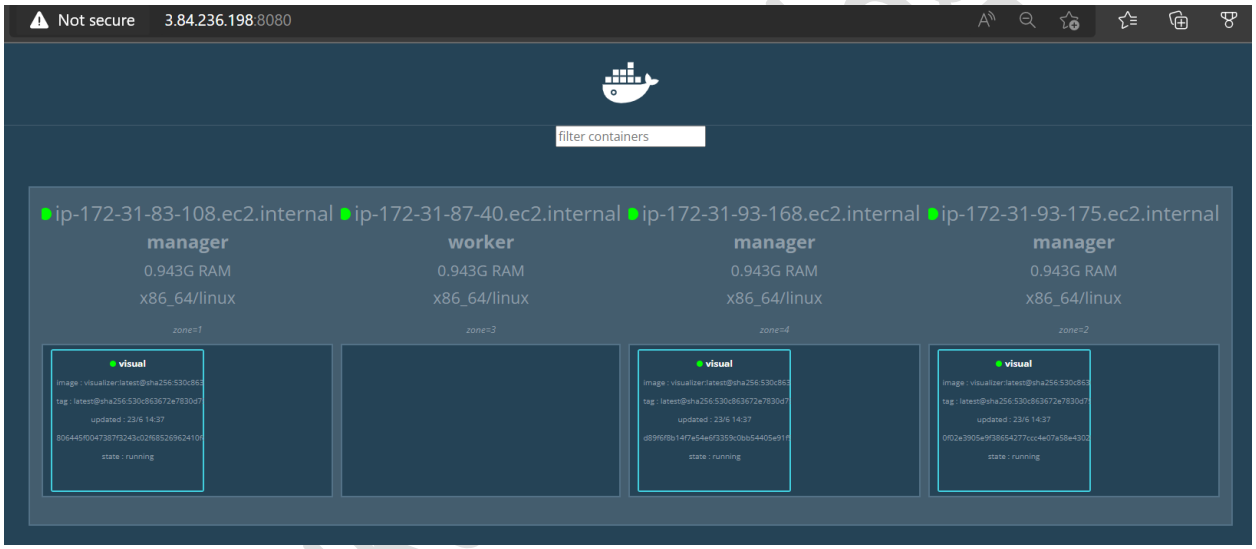
- Once the connections are established without any error, run the below command and verify on Manager Lead Node:

```
[root@ip-172-31-93-168 projects]# docker node ls
ID                            HOSTNAME                          STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
iwta029lvijyjmnj1o75c4kpe     ip-172-31-83-108.ec2.internal     Ready     Active         Reachable        20.10.13
ovnf7gm30wgrb5p74woq4i00t     ip-172-31-87-40.ec2.internal      Ready     Active         Reachable        20.10.13
filkxz8n97o6kbtncs2nap94l *   ip-172-31-93-168.ec2.internal     Ready     Active         Leader           20.10.13
p02ldezhuvxzpd3fhzx73gfqa     ip-172-31-93-175.ec2.internal     Ready     Active         Reachable        20.10.13
```

- To get the graphical view of the docker swarm deploy visualizer service and on Manager Nodes using below command:

```
[root@ip-172-31-93-168 projects]# docker service create --constraint=node.role==manager --mode=global \
> --publish mode=host,target=8080,published=8080 --mount type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
> --name=visual dockersamples/visualizer
overall progress: 3 out of 3 tasks
filkxz8n97o6: running   [==================================================>]
iwta029lvijy: running   [==================================================>]
p02ldezhuvxz: running   [==================================================>]
verify: Service converged
```

- Verify GUI from browser by giving <public ip address of manager>: <port #>




```
[root@ip-172-31-93-168 projects]# docker service ls
ID             NAME      MODE      REPLICAS   IMAGE                                PORTS
rosifl8n9aqd   visual    global    3/3        dockersamples/visualizer:latest
```

- Setting labels to the node that are available

```
[root@ip-172-31-93-168 projects]# docker node update --label-add zone=1 ip-172-31-83-108.ec2.internal
ip-172-31-83-108.ec2.internal
[root@ip-172-31-93-168 projects]# docker node ls
ID                            HOSTNAME                          STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
iwta029lvijyjmnj1o75c4kpe     ip-172-31-83-108.ec2.internal     Ready     Active         Reachable        20.10.13
ovnf7gm30wgrb5p74woq4i00t     ip-172-31-87-40.ec2.internal      Ready     Active                          20.10.13
filkxz8n97o6kbtncs2nap94l *   ip-172-31-93-168.ec2.internal     Ready     Active         Leader           20.10.13
p02ldezhuvxzpd3fhzx73gfqa     ip-172-31-93-175.ec2.internal     Ready     Active         Reachable        20.10.13
[root@ip-172-31-93-168 projects]# docker node update --label-add zone=2 ip-172-31-93-175.ec2.internal
ip-172-31-93-175.ec2.internal
[root@ip-172-31-93-168 projects]# docker node update --label-add zone=3 ip-172-31-87-40.ec2.internal
ip-172-31-87-40.ec2.internal
[root@ip-172-31-93-168 projects]# docker node update --label-add zone=4 ip-172-31-93-168.ec2.internal
ip-172-31-93-168.ec2.internal
```

- Once alias is created try to deploy the service with constraints of the label with below command:

docker service create --constraint node.labels.zone!=1 --replicas 5 --placement-pref 'spread=node.labels.zone' -e NODE_NAME='{{.Node.Hostname}}' -p 80:80 --name nodenamer lrakai/nodenamer:1.0.0

```
[root@ip-172-31-93-168 projects]# docker service create --constraint node.labels.zone!=1 --replicas 5 --placement-pref 'spread=node.labels.zone' -e NODE
_NAME='{{.Node.Hostname}}' -p 80:80 --name nodenamer lrakai/nodenamer:1.0.0
9icchcmtbel9kn7rvyibelfze
overall progress: 5 out of 5 tasks
1/5: running   [==================================================>]
2/5: running   [==================================================>]
3/5: running   [==================================================>]
4/5: running   [==================================================>]
5/5: running   [==================================================>]
verify: Service converged
```

- **Version Modification**: When a new code is committed into the application and a new image will be prepared then use below command to deploy the latest code into the already deployed containers:

  docker service update --image <image-name>:<1.x.x latest version> <service-name>

  sample: docker service update --image lrakai/nodenamer:1.0.1 nodenamer

- **Scale Modification**: When we want to scale up the containers for a service , use below command:

  docker service scale <service-name>=<new replication number>
  docker service scale nodenamer=6

- **Remove Service from Nodes**: Below command is to delete the service that is deployed into the nodes:

  docker service rm <service-name>

  docker service rm nodenamer

- **Promote/Demote of Nodes**:

```
[root@ip-172-31-93-168 projects]# docker node ls
ID                            HOSTNAME                          STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
iwta029lvijyjmnj1o75c4kpe     ip-172-31-83-108.ec2.internal     Ready     Active         Reachable        20.10.13
ovnf7gm30wgrb5p74woq4i00t     ip-172-31-87-40.ec2.internal      Ready     Active         Reachable        20.10.13
filkxz8n97o6kbtncs2nap94l *   ip-172-31-93-168.ec2.internal     Ready     Active         Leader           20.10.13
p02ldezhuvxzpd3fhzx73gfqa     ip-172-31-93-175.ec2.internal     Ready     Active         Reachable        20.10.13
[root@ip-172-31-93-168 projects]# docker node demote ip-172-31-83-108.ec2.internal
Manager ip-172-31-83-108.ec2.internal demoted in the swarm.
[root@ip-172-31-93-168 projects]# docker node ls
ID                            HOSTNAME                          STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
iwta029lvijyjmnj1o75c4kpe     ip-172-31-83-108.ec2.internal     Ready     Active                          20.10.13
ovnf7gm30wgrb5p74woq4i00t     ip-172-31-87-40.ec2.internal      Ready     Active         Reachable        20.10.13
filkxz8n97o6kbtncs2nap94l *   ip-172-31-93-168.ec2.internal     Ready     Active         Leader           20.10.13
p02ldezhuvxzpd3fhzx73gfqa     ip-172-31-93-175.ec2.internal     Ready     Active         Reachable        20.10.13
[root@ip-172-31-93-168 projects]# docker node promote ip-172-31-83-108.ec2.internal
Node ip-172-31-83-108.ec2.internal promoted to a manager in the swarm.
[root@ip-172-31-93-168 projects]# docker node ls
ID                            HOSTNAME                          STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
iwta029lvijyjmnj1o75c4kpe     ip-172-31-83-108.ec2.internal     Ready     Active         Reachable        20.10.13
ovnf7gm30wgrb5p74woq4i00t     ip-172-31-87-40.ec2.internal      Ready     Active         Reachable        20.10.13
filkxz8n97o6kbtncs2nap94l *   ip-172-31-93-168.ec2.internal     Ready     Active         Leader           20.10.13
p02ldezhuvxzpd3fhzx73gfqa     ip-172-31-93-175.ec2.internal     Ready     Active         Reachable        20.10.13
```

Working with Stacks:

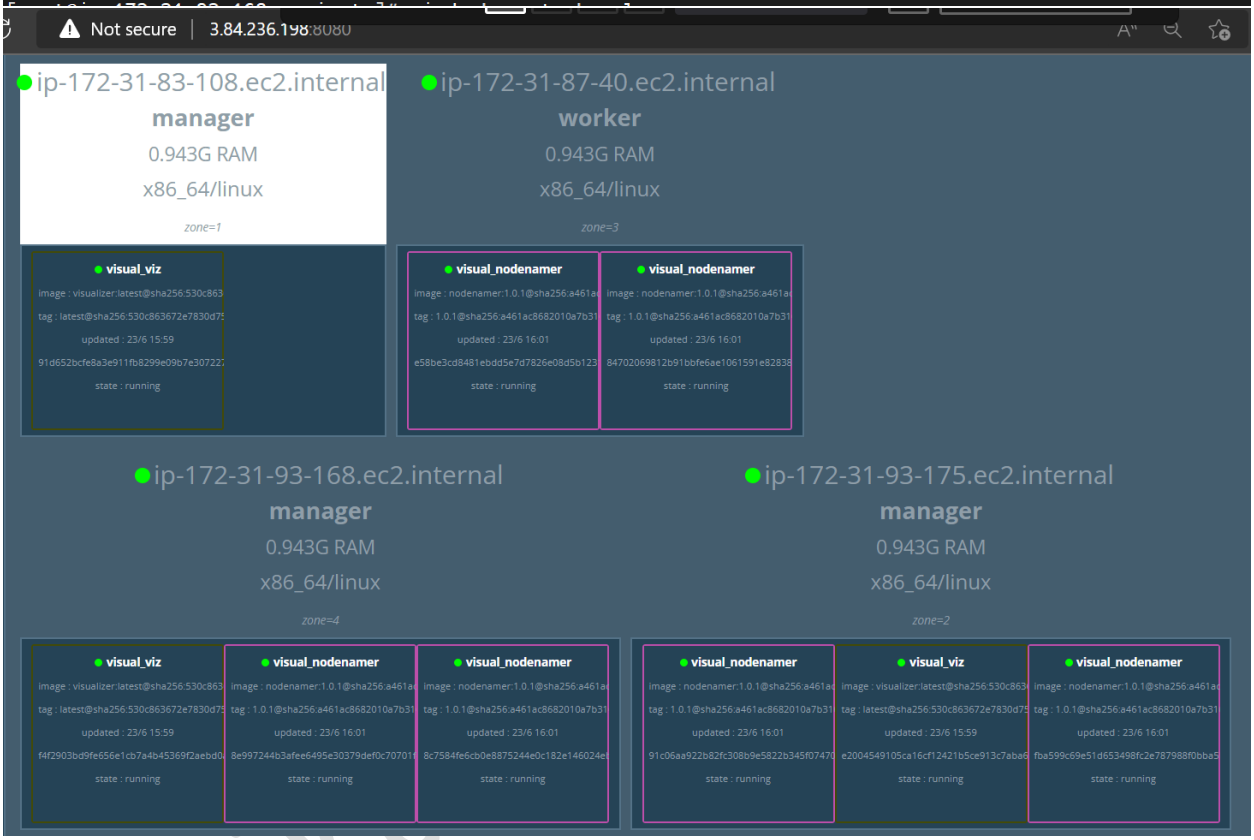
- Above we have seen creating the services with commands now lets do it via stack scripts. Lets take the same scenarios and write a script:

```
version: "3.3"
services:
   viz:
     image: dockersamples/visualizer
     deploy:
        placement:
          constraints:
             - "node.role == manager"
        mode: global
     ports:
        - target: 8080
          published: 8080
          mode: host
     volumes:
       - "/var/run/docker.sock:/var/run/docker.sock"
   nodenamer:
     image: lrakai/nodenamer:1.0.1
     deploy:
       replicas: 6
       resources:
         reservations:
           cpus: '0.5'
       placement:
         constraints:
           - "node.labels.zone != 1"
         preferences:
           - spread: node.labels.zone
     ports:
       - "80:80"
~
```

- Command to deploy/execute the stack script on the swarm:
  docker stack deploy -c docker-stack.yml visual
- Command to display the services currently running:
  docker service ls

- Screen shots for the same:

```
[root@ip-172-31-93-168 projects]# docker stack deploy -c docker-stack.yml visual
Updating service visual_viz (id: wwvler3xhaq0hb2x6emjd3nuf)
Creating service visual_nodenamer
[root@ip-172-31-93-168 projects]# docker service ls
ID             NAME                MODE         REPLICAS   IMAGE                                PORTS
vr0x5xh8hhkl   visual_nodenamer    replicated   0/6        lrakai/nodenamer:1.0.1               *:80->80/tcp
wwvler3xhaq0   visual_viz          global       3/3        dockersamples/visualizer:latest
```



Important Links to explore more on Stacks:

https://docs.docker.com/engine/reference/commandline/stack/

To Explore more about docker please go through the below link:

https://www.educba.com/software-development/software-development-tutorials/docker-tutorial/