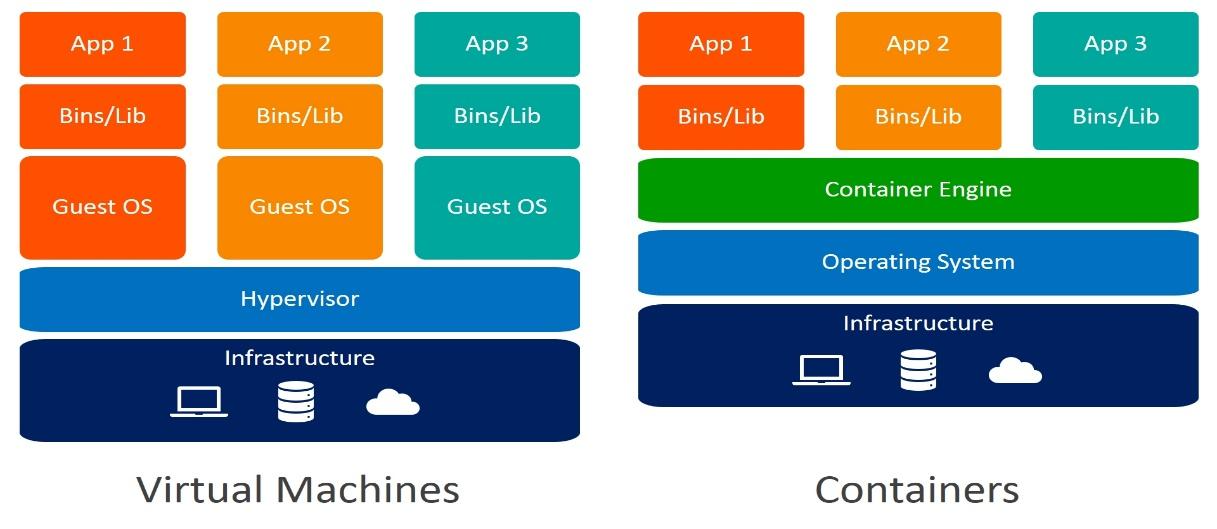


Docker is a platform that provides virtual containers on which an application can be deployed independent of the underlying OS of the server. Further the container can be created from a replica called docker image which contains all the dependencies and can run on any OS that has docker engine, with similar results.



**VIRTUALIZATION:**

Virtualization is the process of *sharing hardware* resources across several virtually isolated and mutually independent systems. It is achieved by using a hypervisor which acts as a bridge ***between the Operating System*** of each of the virtual machines and the underlying hardware.

Applications in virtual environments run on a host operating system on top of the hypervisor.

**CONTAINERIZATION:**

Containerization refers to the process of creating ***virtually isolated instances of applications*** along with the required dependencies called ***containers***. The containers so created ***are independent of the underlying operating system*** and are portable from one OS to the other. A container engine acts as a bridge between the containers and the underlying OS.

* Docker containers run on Docker Engine which needs to be installed on the underlying Host operating system.
* Docker containers with the desired configuration can be stored as an image called Docker Image and stored in a centralized registry.
* The images can be downloaded from the registry on any different server with another host operating system running docker engine.
* Multiple docker containers can be created on the destination server with the same desired configuration.

A diagram of a docker host

Description automatically generated with medium confidence

**DOCKER DAEMON:**

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

**DOCKER CLIENT:**

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use ***commands*** such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

**DOCKER REGISTRIES:**

A Docker registry ***stores Docker images***. Docker ***Hub*** is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

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 registry url:** [**https://hub.docker.com**](https://hub.docker.com)

**Other Docker Registries are: AWS ECR, Google GCR, Microsoft ACR, Harbor ..etc**

**DOCKER OBJECTS:**

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

**1. 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 install 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.

**2. 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.

* Docker container is runtime instance of our application.
* When we run Docker Image, it will create Docker Container.
* Inside Container our application and application dependencies will be available.

**3. VOLUMES**

Docker volumes are used to persist and share data between containers and the host machine. They provide a way to store and manage persistent data separately from the container lifecycle. Volumes can be mounted into containers at specific paths.

**4. NETWORKS**

Docker networks enable communication between containers and the outside world. They provide isolated network environments for containers and allow containers to communicate with each other using IP addresses or container names. Docker supports various network drivers for different use cases.

**DOCKER INSTALLATION (LINUX MACHINE)**

* Login into AWS account
* Create Linux Virtual Machine using Amazon Linux AMI
* Connect to Linux VM using MobaXterm.
* Execute below commands to install Docker Software

**$ sudo yum update -y**

**$ sudo yum install docker -y**

**$ sudo service docker start (or) sudo systemctl start docker**

**$ sudo service docker enable (or) sudo systemctl enable docker**

Add ec2-user to docker group by executing below command.

**$ sudo usermod -aG docker $USER**

Close the terminal.

**$ exit**

Then press 'R' to restart the session (This is in MobaXterm)

Execution below command to see docker status.

**$ docker info**

**BASIC DOCKER COMMANDS**

* Display docker images available in our machine

**$ docker images**

* Download docker image.

**$ docker pull <image-name / image-id>**

* Run docker image.

**$ docker run <image-name / image-id>**

* Delete docker image.

**$ docker rmi <image-name / image-id>**

* Display all running docker containers.

**$ docker ps**

* Display all running and stopped containers.

**$ docker ps -a**

* Delete docker container.

**$ docker rm <container-id>**

* Delete docker image forcefully.

**$ docker rmi -f <image-id>**

* Stop Docker container.

**$ docker stop <container-id>**

* Delete all stopped containers and unused images and unused networks.

**$ docker system prune -a**

**Dockerfile :**

A Dockerfile is a text file that contains a set of instructions used to build a Docker image. Docker images are the building blocks of containers, providing a lightweight and portable way to package and distribute applications.

* Dockerfile contains instructions to build docker images.
* In Dockerfile we will use DSL (Domain Specific Language) keywords
* Docker engine will process Dockerfile instructions from top to bottom.
* Below are the Dockerfile Keywords:
  + **FROM**: Specifies the ***base image*** to use for the Docker image. It defines the starting point of the image.
  + **RUN**: Executes commands in the shell of the container ***during the image build process***. It is used to install packages, set up dependencies, and perform other build-time actions.
  + **COPY** or **ADD**: Copies files and directories from the host machine into the container's file system. COPY is preferred for copying local files, while ADD supports additional features like extracting compressed files and remote URLs.
  + **WORKDIR**: Sets the working directory for any subsequent instructions. It affects the context for commands like RUN, CMD, and COPY.
  + **ENV**: Sets environment variables inside the ***container***. These variables can be accessed by the application running in the container.
  + **EXPOSE**: Informs Docker that the container will listen on the specified network ***ports at runtime***. It does not actually publish the ports; you need to use the -p flag when running the container to do that.
  + **CMD**: Specifies the default command to run ***when the container starts***. It provides the primary functionality of the container.
  + **ENTRYPOINT**: Like CMD, but it specifies the executable to run as the entry point of the container. It is often used for defining a command that cannot be overridden.
  + **VOLUME**: Creates a mount point to attach a ***persistent storage volume*** to the container. It is used to persist data across container restarts.
  + **ARG**: Defines variables that users can pass at ***build-time*** to the builder with the docker build command. They are used to parameterize the build process.
  + **LABEL**: Adds ***metadata*** to the Docker image. It is typically used for providing information about the maintainer, version, or other identifying details.

**FROM:**

* FROM keyword is used to represent base image to create our image.
* On Top of base image our image will be created.

**Syntax:**

FROM base Image

**Ex:**

FROM java:jdk-1.8

FROM tomcat:9.5

FROM mysql:6.8

FROM python:3.3

**MAINTAINER**

* MAINTAINER keyword is used to specify Dockerfile author information.
* keyword has been **deprecated** in recent versions of Docker.
* It is no longer recommended to use the MAINTAINER keyword in your Dockerfile.
* We can use the LABEL keyword instead.

**Syntax:**

MAINTAINER Raghu <javabyraghu@gmail.com>

**COPY**

* COPY command is used to copy the files from source to destination **while creating docker image**.

Syntax:

COPY <source-location> <destination-location>

Ex:

COPY target/sbi-app.war /app/tomcat/webapps/sbi-app.war

**ADD**

* ADD command is also used to copy files from source to destination while creating docker image.

Syntax:

ADD <source-location> <destination-location>

ADD <url> <destination-location>

Ex:

ADD http://abcd.com/myfiles/app.war /app/tomcat/webapps/sbi-app.war

**Q) What is the difference between COPY and ADD commands ?**

Using COPY command, we can just copy the files from one path to another path **within the machine**.

Using ADD command we can copy files from one path to another path and **it supports source location as URL also.**

**RUN**

* RUN instructions will execute ***while creating the image.***
* Using RUN, we can give instructions to docker to execute commands.
* We can write multiple RUN instructions, and the docker will process all the RUN instructions from top to bottom.

**Example**

RUN yum install maven

RUN yum install git

RUN git clone repo-url

RUN mvn clean package

**CMD**

* CMD instructions will execute ***while creating the container.***
* We cannot use this to execute multiple Commands.
* If we define multiple CMD instructions, then the last CMD is always executed.
* Using CMD command we can run our application package file jar / war file.

Example

CMD sudo start tomcat

**Note:** If we write multiple CMD instructions, also docker will process only the last CMD instruction. There is no use of writing multiple CMD instructions in one Dockerfile.

**Q) What is the difference between RUN and CMD in Dockerfile ?**

RUN is used to execute instructions while creating images.

CMD is used to execute instructions while creating Container.

We can write multiple RUN instructions in Dockerfile, docker will process all those instructions one by one.

If we write multiple CMD instructions in Dockerfile, docker will process only the last CMD instruction.

**Sample Dockerfile**

FROM ubuntu

MAINTAINER RAGHU<javabyraghu@gmail.com>

RUN echo "Hi, i am RUN-1"

RUN echo "Hi, i am RUN-2"

CMD echo "Hi, I am CMD-1"

RUN echo "Hi, i am RUN-3"

CMD echo "Hi, i am CMD-2"

* Save the above content in docker file.
* filename : Dockerfile
* Command to create docker image using Dockerfile.

**Syntax : $ docker build -t <image-name> .**

Ex : $ docker build -t myfirstimage .

**Command to run docker image.**

$ docker run myfirstimage

**Command to login with dockerhub account**

$ docker login

Note: We need to enter our dockerhub account credentials correctly (it will ask only first time)

**Command to tag our docker image.**

$ docker tag <image-name> <tag-name>

Ex: $ docker tag myfirstimage javabyraghu/myfirstimage

**Command to push docker image to Docker hub account.**

$ docker push <tag-name>

**Note: Delete all unused images and stopped containers**

$ docker system prune -a

**Pull the image from Docker hub.**

$ docker pull javabyraghu/myfirstimage

**Run the image.**

$ docker run javabyraghu/myfirstimage

**Note:** We can use customized names also for the Dockerfile. When we change Dockerfile name we need to pass filename as input for docker build command using -f option like below.

$ docker build -f <dockerfile-name> -t <image-name> .

**ENTRYPOINT**

* ENTRYPOINT instructions will execute while creating container.

Syntax

ENTRYPOINT [ "echo" , "Welcome to Ashok IT" ]

ENTRYPOINT [ "java" , "-jar" , "target/boot-app.jar" ]

**Q) What is the difference between CMD and ENTRYPOINT ?**

We can override CMD instructions in runtime while creating container.

We can't override ENTRYPOINT instructions.

**WORKDIR**

* It is used to set working directory for an image / container.

Ex:

WORKDIR /app/

**Note:** The Dockerfile instructions which are available after WORKDIR will be processed from given working directory.

**ENV**

* ENV is used to set Environment Variables

Ex:

ENV <key> <value>

ENV java /etc/softwares/java

**ARG**

* It is used to remove hard coded values.
* Using ARG we can pass values in the BUILD TIME like below.

Ex:

ARG branch

RUN git clone -b $branch <repo-url>

Now you Build Image using above Dockerfile

$ docker build -t imageone --build-arg branch=master

**USER**

* We can set a user for creating image / container.

**Note:** After USER instruction all the remaining commands will execute with given user permissions

**EXPOSE**

* It is used to specify our container running PORT.

Ex:

EXPOSE 8080

Note: It is just like a documentation command to provide container running port number.

**VOLUME**

* VOLUME is used to specify docker container data storage location.

Note: Volumes are used for storage.

**SPRING BOOT + DOCKER EXAMPLE:**

* Spring Boot is an open-source java-based framework available in the market to develop applications quickly.
* Spring Boot is providing embedded server (internal server will be available, we no need to configure server for execution)
* Spring Boot application will be packaged as jar/war file (mvn clean package goal will do that package)

Note: When we do maven package, project jar will be created in project target folder

To run spring boot applications, we just need to run jar file like below

$ java -jar <boot-app-jar-file>

**Dockerfile**

FROM openjdk:11

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

WORKDIR /usr/app/

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

**Spring Boot App GitHub Repository URL : https://github.com/javabyraghu/spring-boot-docker-app.git**

**1. Install git client software**

$ sudo yum install git -y

**2. Clone GitHub Repo**

$ git clone https://github.com/javabyraghu/spring-boot-docker-app.git

**3. Navigating to project folder**

$ cd spring-boot-docker-app

**4. Install maven software**

$ sudo yum install maven -y

**5. execute maven goals**

$ mvn clean package

**Note: After the package is successful, we can see project jar file in target folder.**

**6. Create docker image**

$ docker build -t sb-app .

**7. Run docker image with port mapping**

$ docker run -p 8080:8080 sb-app

**Note: Enable 8080 port number in EC2 VM security group**

**URL To Access Application : http://ec2-vm-public-ip:8080/welcome/Raghu**

**8. DETACHED MODE:**

With the command our terminal will be blocked, we can't execute any other command. To execute other commands, we need to type CTRL+C then the terminal will open for commands execution, but our container gets stopped.

$ docker run -p 8080:8080 sb-app

Note: To overcome the above problem we can pass **'-d'** to run container in detached mode. When we execute the command below it will run the container in detached mode, and it will open terminal for commands execution.

$ docker run -d -p 8080:8080 sb-app

**9. Once the above command is executed, we can see running containers using the command below.**

$ docker ps

**10. We can check logs of the container using below command.**

$ docker logs <container-id>

**JAVA WEB APPLICATION + DOCKER EXAMPLE (WITHOUT SPRINGBOOT)**

* Java web applications will be packaged as war files.
* WAR (Web Archive) contains application code.
* To run the war file, we need webserver (Ex: Apache Tomcat)
* We need to deploy war file in Tomcat Server for Execution
* In Tomcat server we will have "webapps" folder for deployment

Note: To run normal java web applications we need "Java(JRE) & Tomcat" as dependencies

**Dockerfile**

**FROM tomcat:8.0.20-jre8**

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

**Java Web Application GitHub Repository : https://github.com/javabyraghu/maven-web-app.git**

**1. Clone Repository**

$ git clone https://github.com/javabyraghu/maven-web-app.git

**2. Go into Project Folder**

$ cd maven-web-app

**3. Clean and Build as WAR file.**

$ mvn clean package

**4. Create Docker Image from Dockerfile**

$ docker build -t maven-web-app .

**5. View Images in Docker**

$ docker images

**6. Run Docker Image in Detached Mode.**

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

**7. Check Docker containers running**

$ docker ps

**8. View Log file data**

$ docker logs <container-id>

**URL To access The Application :** http://ec2-vm-public-ip:host-port/maven-web-app/

**Note:** In the above URL "maven-web-app" is called as context path (name of the war file will become context path)

**PYTHON APPLICATION + DOCKER EXAMPLE**

* Python is a general-purpose scripting language.
* Python programs will have .py as extension.
* Compilation is not required for python programs.

**Dockerfile**

FROM python:3.6

MAINTAINER Mr. RAGHU "javabyraghu@gmail.com"

COPY . /app

WORKDIR /app

EXPOSE 5000

RUN pip install -r requirements.txt

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

Python Flask application GitHub Repository URL:

https://github.com/javabyraghu/python-flask-docker-app.git

**$ git clone https://github.com/javabyraghu/python-flask-docker-app.git**

**$ cd python-flask-docker-app**

**$ docker build -t python-flask-app .**

**$ docker images**

**$ docker run -d -p 5000:5000 python-flask-app**

**$ docker ps**

**$ docker logs <container-id>**

# Command to enter docker container

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

Note: Execute 'exit' command to come out from docker container virtual machine.

**ANGULAR APPLICATION + DOCKER EXAMPLE**

Angular is a TypeScript-based, free, and open-source single-page web application framework led by the Angular Team at Google and by a community of individuals and corporations.

* It runs on node JS (or node) software.
* The default port number is 4200, but we can map to 80 using nginx.
* Nginx is an open-source webserver, proxy server and load balancer too.

Angular application GitHub Repository URL:

<https://github.com/javabyraghu/angular-docker-app.git>

**$ git clone https://github.com/javabyraghu/angular-docker-app.git**

**$ cd angular-docker-app**

**$ docker build -t angular-docker-app .**

**$ docker images**

**$ docker run -d -p 80:80 angular-docker-app**

**$ docker ps**

**$ docker logs <container-id>**

**DOCKER NETWORKING**

Docker provides flexible networking capabilities to facilitate ***communication between containers*** and external networks.

**1. Default | Bridge Network:** By default, Docker creates a bridge network named bridge. Containers attached to this network can communicate with each other using IP addresses. Docker assigns IP addresses automatically to containers on this network.

* Docker allows you to create ***user-defined bridge networks***.
* These networks enable containers to communicate with each other using container names as hostnames.

**2. Host Network:** Using the ***--network=host*** option, containers can directly share the host network stack.

**3. Overlay Networks:** Docker supports overlay networks for multi-host communication in swarm mode. Overlay networks enable containers running on different hosts to communicate securely.

4. **Macvlan:** driver will assign **MAC** address for a container. It makes our container as Physical.

**5. None Network:** None means no network will be provided by our Docker containers.

Docker provides several networking commands to manage network-related aspects of containers.

Here are some commonly used Docker networking commands:

**A. Creates a new Docker network.**

$ docker network create <network\_name>

**B. Lists the available Docker networks.**

$ docker network ls

**C. Provides detailed information about a Docker network.**

$ docker network inspect <network\_name>

**D. Connects a container to a Docker network.**

$ docker network connect <network\_name> <container\_name\_or\_id>

**E. Disconnects a container from a Docker network.**

$ docker network disconnect <network\_name> <container\_name\_or\_id>

**F. Removes a Docker network.**

$ docker network rm <network\_name>

**G. Removes all unused Docker networks.**

$ docker network prune

**H. Creates an overlay network for swarm services.**

$ docker network create --driver overlay <network\_name>

**I. Creates a bridge network for containers to communicate on the same host.**

$ docker network create --driver bridge <network\_name>

**Lab Task:**

$ docker network create bank-network

**$ docker network ls**

$ docker network inspect <network-id>

$ docker run -d --network bank-network ubuntu /bin/bash

$ docker run -d --network host nginx (runs on port 80 default)

$ docker network disconnect bank-network <container-id>

$ docker network remove bank-network

**DOCKER VOLUMES**

**Stateful Containers Vs Stateless Containers:**

* Stateless Container means container will not persist data after destroying and creating a new one.
* When we re-create the new container, we will lose old data.
* **By default, docker containers are stateless containers.**
* If we deploy the latest code or if we re-create the containers, we should not lose our old data.
* Our data should remain in the database/directories.
* If we don't want to lose the data even if we re-create the container, then we need to make our Docker Container as Stateful Container.
* To make Docker Containers stateful, we need to use Docker Volumes.

**Docker Volumes:**

Docker volumes are a feature that allows you to ***persist*** and share data between Docker containers. They provide a way to store and manage data generated by containers, even if the containers are stopped, removed, or replaced.

* We use docker volumes for file systems, databases, and object storage servers.
* Application will store data in database, even if we delete application container or Database container data should be available.
* To make sure data is available even after the container is deleted then we need to use Docker Volumes concept.

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

Docker has 3 types of volumes,

1) Anonymous Volumes (without name)

2) Named Volumes (Will have a name) **\*\* Recommended**

3) Bind Mounts ( Storing on Host Machine )



Q) What is Dangling volume in Docker ?

The volumes which are created but not associated to any container are called as Dangling Volumes

**# Delete all dangling volumes**

$ docker volume rm $(docker volume ls -q -f dangling=true);

**# Create Docker volume**

$ docker volume create <vol-name>

**# Display all docker volumes**

$ docker volume ls

**# Inspect Docker Volume**

$ docker volume inspect <vol-name>

**# Delete docker volume**

$ docker volume rm <vol-name>

**# Delete all docker volumes**

$ docker system prune --volumes

**\*\* LAB TASK - MySQL CONTAINER WITH VOLUMES \*\***

**1. View Docker volumes**

$ docker volume ls

**2. Create a new Docker volume**

$ docker volume create mysql\_data

**3. Create a MySQL Container with above volume binding (-v volume\_name:/path/in/container)**

$ docker run -d --name mysql -e MYSQL\_ROOT\_PASSWORD=root **-v mysql\_data:/var/lib/mysql** mysql:8.0

**4. Connect to MySQL Container Instance**

$ docker exec -it mysql bash

**5. Create database, table, then insert few rows**

mysql -u root -p

CREATE DATABASE raghudb;

USE raghudb;

CREATE TABLE users (id INT PRIMARY KEY AUTO\_INCREMENT, name VARCHAR(50));

INSERT INTO users (name) VALUES ('Raghu'), ('Raju'), ('Ramu');

SELECT \* FROM users;

**6. Come out of MySQL Container instance.**

$ exit

$ exit

**7. Destroy the MySQL Container.**

$ docker rm -f mysql

**8. Create a new MySQL Container with above Volume**

$ docker run -d --name mysql\_two -e MYSQL\_ROOT\_PASSWORD=root -v mysql\_data:/var/lib/mysql mysql:8.0

**9. Connect and check above Inserted rows exist or not?**

$ docker exec -it mysql\_two bash

mysql -u root -p

USE raghudb;

SELECT \* FROM users;

**10. Come out of MySQL Container instance**

$ exit

$ exit

**11. Destroy MySQL Container and Volume**

$ docker rm -f mysql\_two

$ docker volume rm mysql\_data

**DOCKER COMPOSE**

Docker Compose is a tool that allows you to define and manage multi-container Docker applications. It simplifies the process of running multiple containers, their configurations, and their interdependencies. Compose uses a YAML file to define the services, networks, and volumes required for your application.

* Docker Compose is a tool which is used to manage multi container-based applications.
* Using Docker Compose we can easily setup & deploy multi container-based applications.
* We will give containers information to Docker Compose using YML file (***docker-compose.yml***)
* Docker Compose YML should have all the information related to containers creation.
* **Docker Compose YML File Looks Like:**

version:

services:

network:

volumes:

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

**A. Create Containers using Docker Compose.**

$ docker-compose up -d

**B. Create Containers using Docker Compose with custom file name.**

$ docker-compose -f <filename> up -d

**C. Display Containers created by Docker Compose.**

$ docker-compose ps

**D. Display docker compose images.**

$ docker-compose images

**E. restart the containers created by docker compose.**

$ docker-compose restart

**F. Stop & remove the containers created by docker compose.**

$ docker-compose down

**DOCKER COMPOSE SETUP:**

**1. download docker compose**

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

**2. Allow Execution permission**

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

**3. check docker compose is installed or not**

$ docker-compose --version

**Example#1**

version: '3'

services:

  web:

    image: nginx:latest

    ports:

      - 80:80

  db:

    image: mysql:latest

    environment:

      - MYSQL\_ROOT\_PASSWORD=secret

Save above file with “docker-compose.yml” or any other name ex: setup.yml

$ docker-compose **-f setup.yml** -d

$ docker-compose -f setup.yml images

$ docker-compose -f setup.yml ps

$ docker exec -it ec2-user-db-1 bash (check container name)

$ mysql -u root -p

$ exit (2 times to come out of db and container)

**DOCKER COMPOSE EXAMPLE FOR SPRING BOOT + MySQL DATABASE**

**1. Install git and maven**

$ sudo yum install git maven -y

**2. Clone GitHub Repository**

$ git clone https://github.com/javabyraghu/spring-boot-mysql-docker-compose.git

**3. Build Maven application**

$ cd spring-boot-mysql-docker-compose

$ mvn clean package

**4. Create Docker image**

$ docker build -t spring-boot-mysql-app .

**5. Run Docker Compose file**

$ docker-compose up -d

**6. Test Application (Open PORT 8080 on Security Group)**

[**http://EC2-PUBLIC-IP:8080**](http://EC2-PUBLIC-IP:8080)

**7. Connect to Database container**

docker exec -it <db-container-name> /bin/bash

**8. Login to database and check table data**

$ mysql -u root -p

$ show databases;

$ use springdb ;

$ show tables;

$ select \* from book;

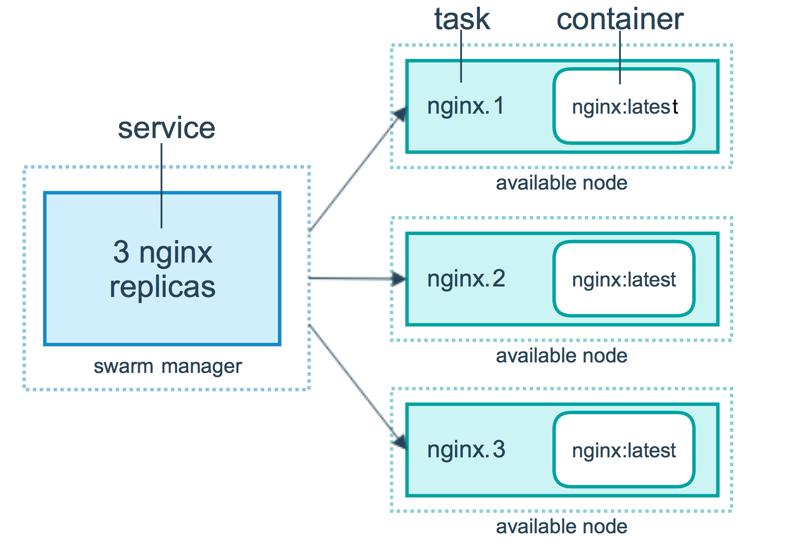
$ exit

$ exit

**DOCKER SWARM**

Docker Swarm is a native clustering and ***orchestration*** solution provided by Docker. It allows you to create and manage a swarm of Docker nodes, forming a cluster of Docker engines, and provides native support for running and scaling containerized applications across multiple hosts.

* **Docker:** It is a **containerization** platform. It is used to deploy the applications as containers.
* **Docker Swarm:** It is an **Orchestration** Platform. It is used to manage docker containers.
* Managing docker containers is nothing but creating / updating / scale up / scale down / remove containers.
* There are Orchestration software’s in market: docker swarm, **Kubernetes**, OpenShift, AWS EKS, GKS, AKS.
* Docker Swarm is used to configure Docker Cluster.
* Cluster means group of servers.
* Docker swarm is embedded in Docker engine ( No need to install Docker Swarm Separately )
* We need to configure ***Master and Worker nodes*** using Docker Swarm cluster.
* Master Node will schedule the tasks (containers) and manage the nodes and node failures.
* Worker nodes will perform the action (containers will run here) based on master node instructions.



**DOCKER SWARM FEATURES:**

1. **Container Orchestration:** Docker Swarm enables orchestration of containerized applications across multiple hosts.
2. **High Availability:** Swarm provides fault-tolerant services by automatically distributing containers across nodes and managing failovers.
3. **Scaling:** Easily scale services up or down by adjusting the number of replicas or tasks.
4. **Service Discovery:** Swarm offers built-in DNS-based service discovery, allowing containers to communicate with each other by service name.
5. **Rolling Updates:** Perform rolling updates to services without downtime, ensuring seamless application updates.
6. **Load Balancing:** Swarm provides built-in load balancing to distribute incoming requests across multiple replicas of a service.
7. **Security:** Swarm supports TLS encryption for secure communication between nodes and automatic certificate management.
8. **Overlay Networking:** Swarm enables the creation of overlay networks, allowing containers on different hosts to communicate securely.
9. **Self-Healing:** Swarm automatically restarts failed containers and reschedules them on healthy nodes.
10. **Swarm Visualizer:** Docker Swarm includes a visualizer tool to monitor and visualize the state of your Swarm cluster.

**DOCKER SWARM CLUSTER SETUP:**

**1. Create 3 EC2 instances (ubuntu) & install docker in all 3 instances using below 2 commands**

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

$ sudo sh get-docker.sh

**Note:** **Enable 2377 port in security group** for Swarm Cluster Communications

1 - Master Node

2 - Worker Nodes

**2. Connect to Master Machine and execute below command**

**(A) Initialize docker swarm cluster**

$ sudo docker swarm init --advertise-addr <private-ip-of-master-node>

Ex : $ sudo docker swarm init --advertise-addr 172.31.37.100

**(B) Get Join token from master (this token is used by master to join the worker node with master)**

$ sudo docker swarm join-token worker

**Note: Copy the token and execute in all worker nodes with sudo permission**

Ex: sudo docker swarm join --token SWMTKN-1-4pkn4fiwm09haue0v633s6snitq693p1h7d1774c8y0hfl9yz9-8l7vptikm0x29shtkhn0ki8wz 172.31.37.100:2377

**(C) View all nodes**

$ docker node ls

**(D) In Docker swarm we need to deploy our application as a service.**

**DOCKER SWARM SERVICE:**

* Service is collection of one or more containers of same image.
* To create the service follow below command

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

Ex : $ sudo docker service create --name my-web-app -p 8080:8080 javabyraghu/maven-web-app

**Note: By default, 1 replica will be created**

**Note: We can access our application using below URL pattern**

URL : http://master-node-public-ip:8080/maven-web-app/

**# Check the services created**

$ sudo docker service ls

**# we can scale docker service**

$ docker service scale <serviceName>=<no.of.replicas>

**# Inspect docker service**

$ sudo docker service inspect --pretty <service-name>

**# View service details**

$ sudo docker service ps <service-name>

**# Remove docker service**

$ sudo docker service rm <service-name>

**# Remove one node from swarm cluster**

$ sudo docker swarm leave