**Docker**

* Docker is a software platform for building applications based on containers
* Docker is an open source software development platform which runs software packages called “containers”
* Docker enables operating system level virtualization

**What is virtualisation?**

Creating software based virtual version of something - compute storage, servers, application etc. can be done on a single physical hardware. Hypervisor is used for this

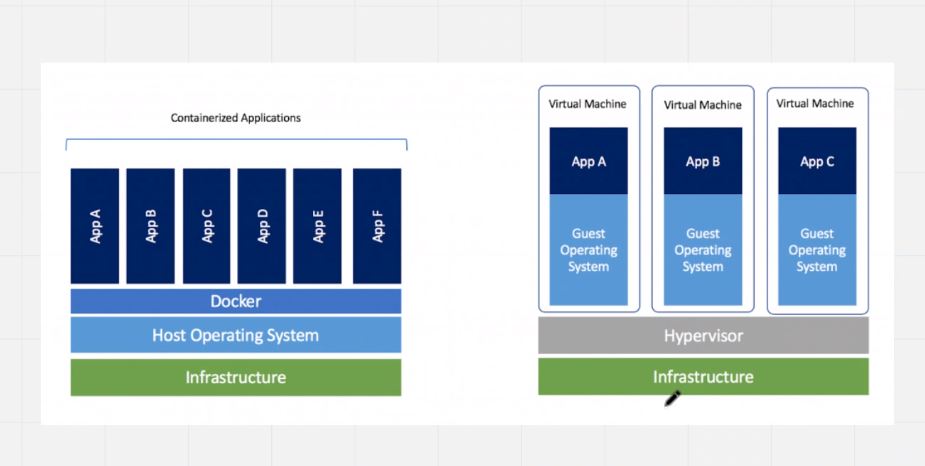
**What is containerisation?**

Ex: an app developed and deployed cannot be run on another environment perfectly. So to solve this we use containerization. For this we bundle and wrap all configuration files and dependencies. This bundles is called container.

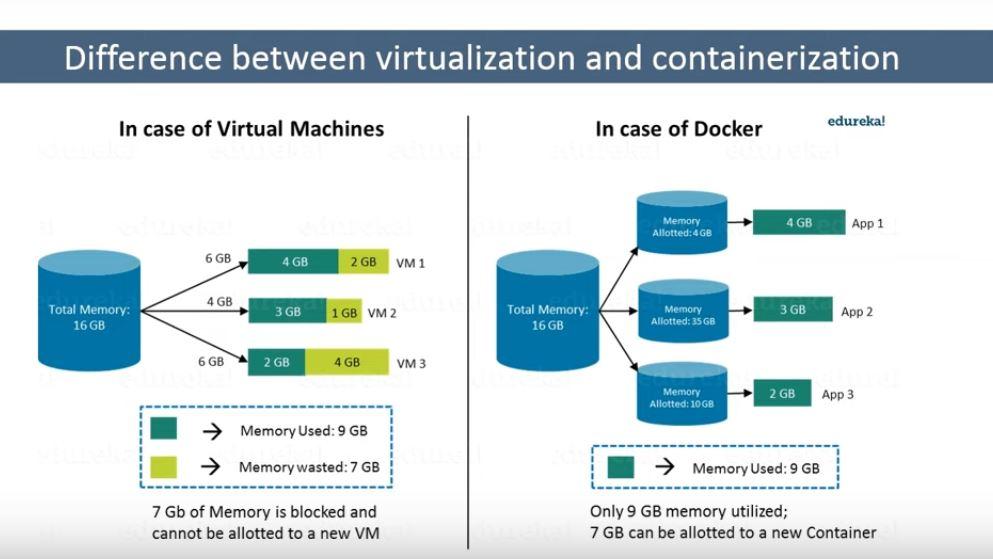
Docker: containerization platform

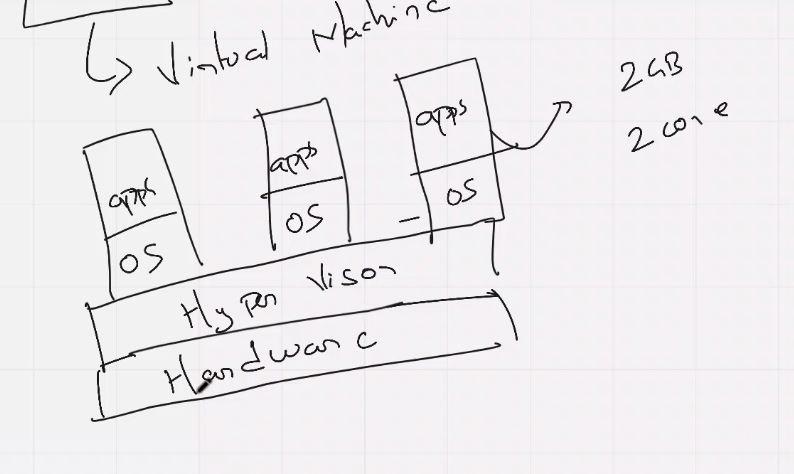
**Containers & Virtual Machines:**

* One of the goals of modern software development is to keep applications on the same host or cluster isolated from one another so they don’t interfere with each other’s operation or maintenance.



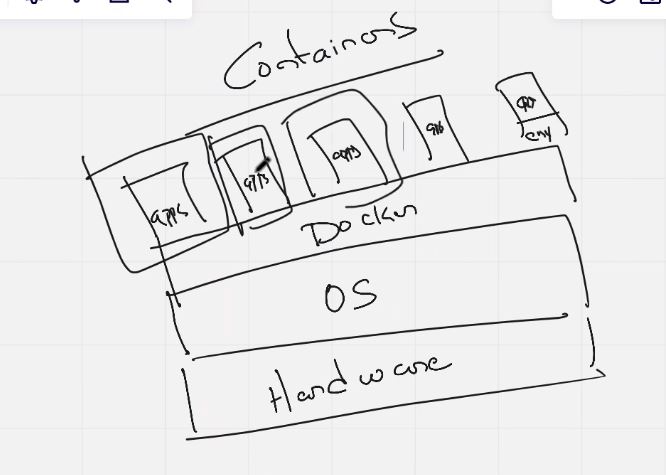
**Virtual Machines:**

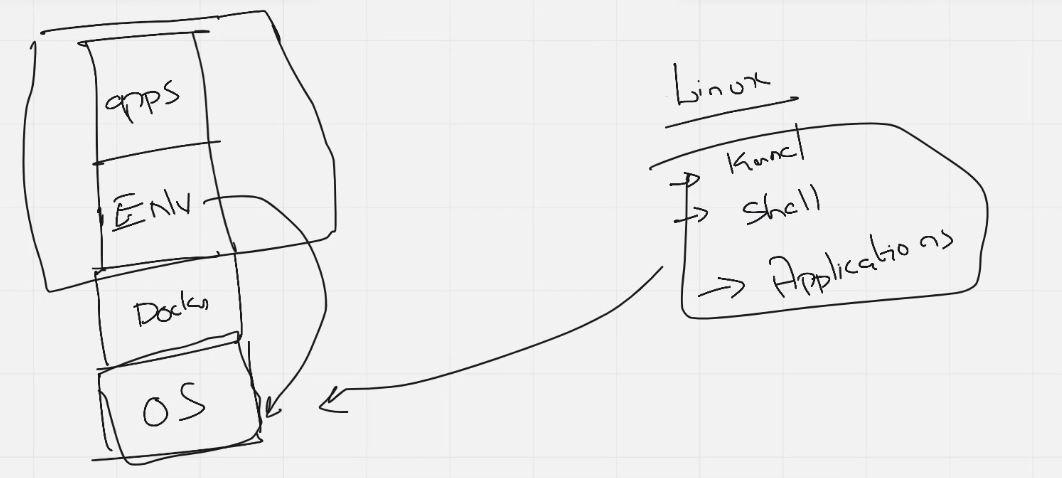
* One solution to this problem has been virtual machines, which
  + keep applications on the same hardware entirely separate, and
  + reduce conflicts among software components and
  + keep competition for hardware resources to a minimum. ****

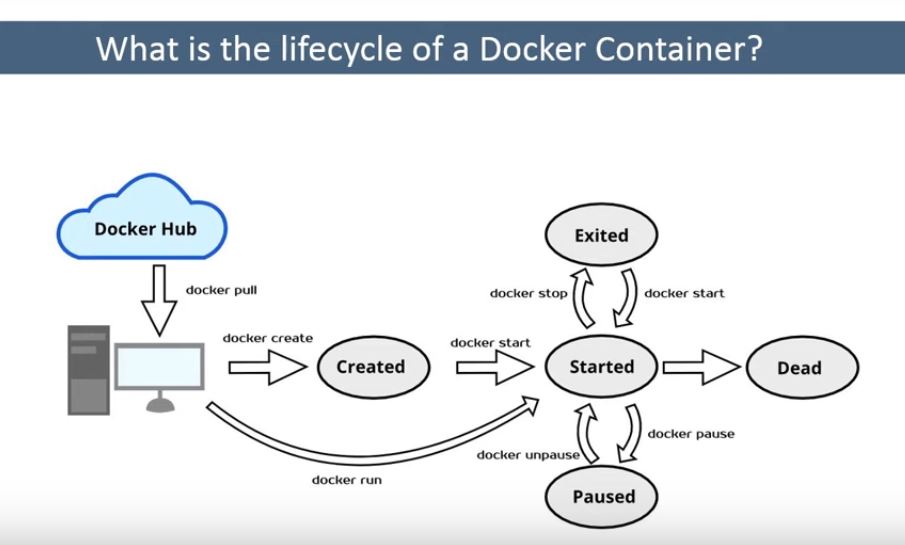
But virtual machines are bulky—each requires its own OS, so is typically gigabytes in size—and difficult to maintain and upgrade.

**Containers:**

* Containers, by contrast, share the same underlying kernel and isolate applications’ execution environments from one another. Due to this they are usually in megabytes and boot up instantly.
* A container is a standalone unit of software that packages together applications with all its dependencies and configurations. Also from one computing environment to another.







**Installation:**

* curl -fsSL https://get.docker.com -o get-docker.sh
* Docker Website URL: https://docs.docker.com/engine/install/

**Post Installation:**

* sudo usermod -aG docker $USER --> Adding current user to Docker group
* Docker Website URL: https://docs.docker.com/engine/install/linux-postinstall/

**Steps:**

* launch instance
* download docker for linux
* connect to ec2 instance
* copy commands into machine

**post installation step**

* root user will have permissions but default user won’t have such permission we should use sudo or docker post installation steps - adding user to the docker group:
* sudo usermod -aG docker $USER

**Docker Image:**

* Docker is used to create, run and deploy applications in containers.
* A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run.
* In simple words a Docker image is an executable file that creates a Docker container.
* A Docker image is comparable to an AMI in AWS.
* Docker images are a reusable and can be deployed on any host. Developers can take the Docker image from one project and use them in another. This saves the user time, because they do not have to recreate an image from scratch.

**Docker Image Repository:**

* Docker images can be stored in private or public repositories, such as Docker Hub, Amazon Container Registry etc.
* https://hub.docker.com/

**Commands:**

**Images:**

1. **docker search <image\_name>** --> To search for a Docker Image
2. **docker pull <image\_name>** --> To pull a Docker Image to local Machine
3. **docker images** --> To Check all the images on the machine
4. **docker rmi <image\_name/image\_id>** --> To delete a Docker Image

**Containers:**

(no two containers can have same name)

1. **docker run -it <image\_name>** --> To create a Docker container
   * 1. i = interactive
     2. t = tty 🡪 pseudo terminal
2. **docker ps** --> To show all running containers
3. **docker ps -a** --> To show all containers
4. **docker rm <container\_name/container\_ID>** --> To remove a container

**Docker Architecture:**

**Docker daemon:**

* It is also referred to as ‘**dockerd**’ and it accepts Docker API requests and manages Docker objects such as images, containers, networks, and volumes.
* It can also communicate with other daemons to manage Docker services.
* Note: A daemon is a service process that runs in the background and functionality to other processes.

**Docker Client:**

* It is the way that enables users to interact with Docker.
* It sends the docker commands to docker daemon. The Docker client can communicate with more than one daemon.

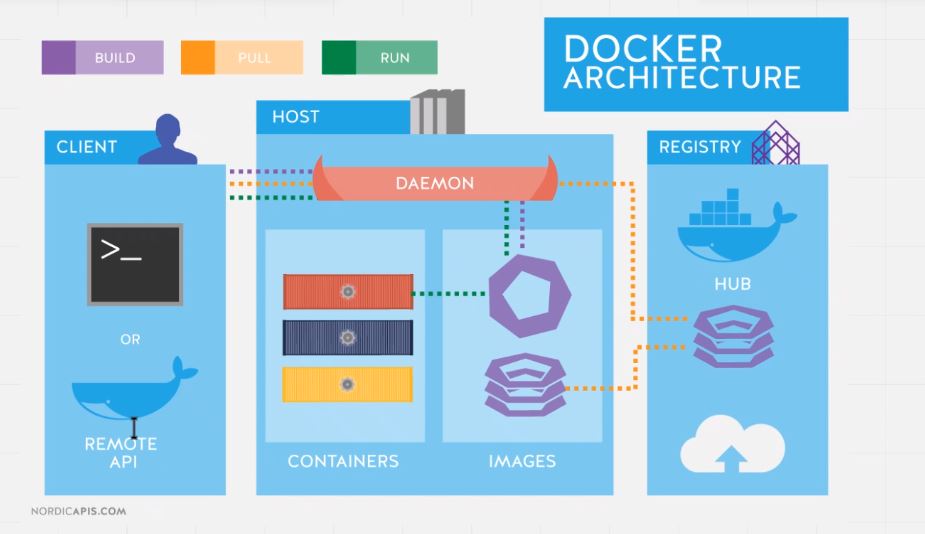
**Docker Registry:**

* It hosts the Docker images and is used to pull and push the docker images from registry. Docker Hub is the public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default.

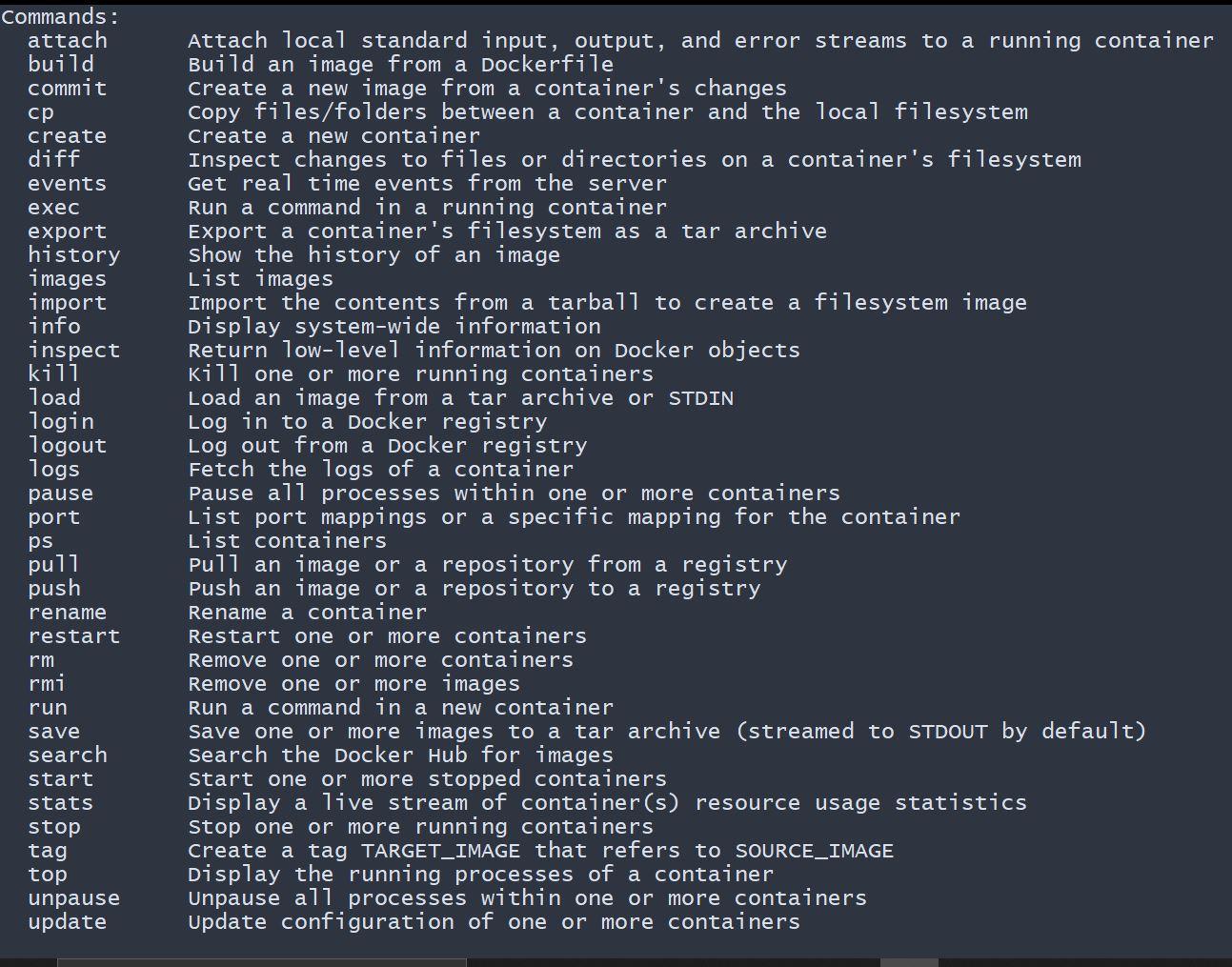
**Docker Host:**

* It is the physical host on which Docker Daemon is running and docker images and containers are created.

Whatever operations we perform in container will remain in container. it doesn’t effect the original OS environment or server.



Alpine is lighter form of Linux only works in container - package installer for alpine is apk

****

**Docker Container Commands:**

* **docker run <image\_name>** 🡪 To Run a Container
* **docker run -it <image\_name>** 🡪 To create a Docker container and interact with it
* **docker run -itd <image\_name>** 🡪 To create a Docker container in detached mode [background]
* **docker rename**
* **docker inspect**
* **docker run -it --name <container\_name> <image\_name>** 🡪 To create a Docker container with specified name

( If no "-it" flag - no interaction, even then if go to ps -a, it will start and exit - no interaction "-it" when given, it keeps running until we stop it

-it --> i means interactive, t means tty that is psuedo terminal - like interactive shell

In docker a container will be running till the operation is done. for example - container starts - creates file - then stops.

When -it is not given it assumes no operation and it exits)

**docker stop <container\_name/container\_ID>** To stop a running container (default time 10sec to complete any running process)

**docker stop --time=100 <container\_name/container\_ID>** To stop a running container after a particular time period

**docker start <container\_name/container\_ID>** To start a container

**docker start -i <container\_name/container\_ID>** To start a container and attach to it

**docker attach <container\_name/container\_ID>** To attach to a running container

**docker exec <container\_name/container\_ID> <command>**  To Execute a command inside a container

**docker exec -it <container\_name/container\_ID> /bin/sh** To get into a running container

**ctrl p+q** Detach Keys [Keys to detach from a container without stopping it]

**docker run -it --detach-keys ctrl-s <image\_name>**  To set manual specific detach key

**docker run -it -p 8080:8080 <image\_name>**

To connect to a container running on a specific port (p means publish)

We can’t run multiple containers on same port. From outside world it goes to host - (9000) from there it goes to container (8080) that host port

**Assignment:**

1. **Difference between attach and exec commands**

* The docker attach command allows you to attach to a running container using the container’s ID or name, either to view its ongoing output or to control it interactively. You can attach to the same contained process multiple times simultaneously, screen sharing style, or quickly view the progress of your detached process.
* The command docker attach is for attaching to the existing process. So when you exit, you exit the existing process.
* If we use docker attach, we can use only one instance of shell. So if we want open new terminal with new instance of container’s shell, we just need run docker exec

1. **How to map multiple ports**

* **docker run -p <host\_port1>:<container\_port1> -p <host\_port2>:<container\_port2>**
* In docker file,
  + EXPOSE 9090
  + EXPOSE 9091
  + Or
  + EXPOSE 9090-9091

**Docker file:**

* Docker file is a text document that contain instructions to create custom image in Docker
* Default name : **Dockerfile**
* contains a base image (alpine/ubuntu etc)
* on that update all the packages - layer 1
* then install package 1 - layer 2
* package 2 - layer 3
* so on
* Default Name: Dockerfile

**Commands:**

1. **docker build .** --> Builds a docker with Dockerfile present in current directory
2. **docker build -f <Dockerfile\_name> <Dockerfile\_path>** --> TO create a image with custom Dockerfile name
3. **docker build --tag=<image\_name>:<tag> .** --> To set a name for the custom image

**Dockerfile instructions:**

**FROM**

* The FROM instruction allows you to set a base image such as OS, programming language etc upon which we can build a custom image
* A valid docker file should always start with ‘FROM’ instruction and it can appear multiple times in the docker file to make or to form multi stage build process

**Syntax:**

**FROM <image\_name>:<tag>**

**RUN:**

* RUN is used to run specified commands on the base image. Each run command in the Dockerfile creates a new intermediate image layer. Therefore, grouping them all together into a single run instruction is more efficient

**Syntax :**

1. **RUN apt-get update**
2. **RUN apt-get install git -y**
3. **RUN apt-get install wget -y**
4. **RUN apt-get update && apt-get install git -y && apt-get install wget -y**

**COPY:**

* To copy file and directories from the host machine to the container

Syntax:

1. **COPY <source\_path> <destination\_path>**
2. **COPY --chown=<user>:<group> <source\_path> <destination\_path>**

**ADD**

It is also used to copy files and directories however it also allows you to add files from a url as well as extracted contents from a tar file

**Syntax:**

1. **ADD <source\_path> <destination\_path>**
2. **ADD --chown=<user>:<group> <source\_path> <destination\_path>**
3. **ADD <URL> <destination\_path>**
4. **ADD <tar> <destination\_path>**

**Q:** copy vs add

**Q:** cmd vs entrypoints

**CMD**

* CMD is used to set a default command that gets executed once you run the docker container
* In case we provide a command during docker run, the CMD instruction gets ignored
* In case of multiple CMD commands only the last one gets executed.

**Syntax:**

1. **CMD /bin/sh**
2. **CMD ls**
3. **CMD ["echo", "hello"]**

**ENTRYPOINT**

* It is also used to set a default execution point once we run the docker container but when a command is passed during run time, ENTRYPOINT will take the command as an **argument** to the original entry point instruction
* In case of multiple entry point commands, only the last one gets executed.

**Syntax:**

1. **ENTRYPOINT /bin/sh**
2. **ENTRYPOINT ls**
3. **ENTRYPOINT ["echo", "hello"]**
4. **docker run -it --entrypoint="<command>" <image\_name>**

**ENV**

* This is used to set environment variables in the containers

**Syntax:**

1. **ENV <key>=<value>**
2. **ENV <key1>=<value1> <key2>=<value2>**

**Ex: JAVA=/home/openjdk**

1. **docker run -it --env <key>=<value> <image\_name>** --> To set environment variables during run time

**Q:** How to set multiple environment variables or Set environment variables using a file?

**docker run -it --env-file <path\_of\_file> <image\_name>**

**ARG**

* It is used to define a variable whose value can be passed during the docker build.

**Syntax:**

1. **ARG <variable\_name>**

**Command:**

**docker build --build-arg <variable\_name>=<value> .** --> To pass the value to the variable during build

Example:

FROM ubuntu

ARG user

RUN useradd $user

COPY --chown=$user ./filename path/

docker build --build-arg user=abc -t <image\_name> .

**Assignment:**

Learn about WORKDIR, USER, SHELL, EXPOSE

**Docker Push**

**Dockerhub:**

Prerequisites:

1. You need to setup a Dockerhub account
2. Login to Dockerhub on the server using docker login command

Limitation: 1 pvt repository, unlimited public repositories

**Commands:**

1. **docker login** --> To setup Dockerhub Credentials
2. **docker build -t <user\_name>/<repository\_name>:tag .** --> To build a image
3. **docker push <user\_name>/<repository\_name>:tag** --> To push the image to Dockerhub

**Amazon ELastic Container Registry [ECR]:**

Prerequisites:

1. Setup AWS account
2. Setup AWS CLI on the server
3. Create a repository in AWS ECR

Limitations: only 500mb free

* AWS CLI-Linux: <https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html>
* Setup Access Key and Secret Key using aws configure command

**Commands**

* **aws ecr get-login-password --region <AWS\_region> \**

**| docker login \**

**--username AWS \**

**--password-stdin <ECR\_URL>**

* **docker build -t <user\_name>/<repository\_name>:tag .** --> To build a image
* **docker push <user\_name>/<repository\_name>:tag** --> To push the image to Dockerhub
* aws ecr get-login-password --region ap-southeast-1 \

| docker login \

--username AWS \

--password-stdin 304693562183.dkr.ecr.ap-southeast-1.amazonaws.com/ecr-push

For setting up permanent connection to AWS:

<https://github.com/awslabs/amazon-ecr-credential-helper>

**Assignment Solutions:**

3. Learn about WORKDIR, USER, SHELL, EXPOSE

**WORKDIR:**

define the *working directory* of a Docker container at any given time. RUN, CMD, ADD, COPY, or ENTRYPOINT command will be executed in the specified working directory.

**USER:**

to create a user. **The default user in a Dockerfile is the user of the parent image**.

**SHELL:**

* to define which SHELL to use in the container
* The SHELL instruction allows the default shell used for the shell form of commands to be overridden.
* The default shell on Linux is ["/bin/sh", "-c"], and
* on Windows is ["cmd", "/S", "/C"].
* The SHELL instruction must be written in JSON form in a Dockerfile.

**EXPOSE:**

tells Docker to get all its information required during the runtime from a specified Port.

4. Docker commit command

**Volumes:**

By default, the data inside the containers is lost once we remove the container. So, in order to persist/save data in a container or share data between multiple containers, we can use Docker volumes and bind mounts

* Docker volume can be used as a backup for containers. any changes in container can be backed up
* The volumes will stay as long as container stays.
* If we are running multiple containers, they can share a common volume and can share data.

**2 types -**

**Bind Mounts** - general folder structure, we use linux commands if we have to mount it to container then we can (anywhere we can specify path)

**Docker Volume** - manged by docker itself, Docker CLI commands are used - path is /var/lib/docker/volumes

**1. Bind Mounts:**

* we can establish a connection from a container to a directory in the host machine.

**Syntax:**

* + d**ocker run -it -v <host\_path>:<container\_path> <image\_name>**

**Exmaple:**

1. docker run -it -v /home/ubuntu/test:/home alpine
2. docker run -it -v /home/ubuntu/jenkins:/var/jenkins\_home jenkin/Jenkins

**2. Docker Volumes:**

* A bind mount uses the host file system but Docker volumes are completely managed by Docker.
* Similar to bind mounts these Docker volume can persist the data even after we remove the container

**Docker Volumes Path: /var/lib/docker/volumes**

**Commands:**

1. **docker volume create <volume\_name>** 🡪 To create a volume
2. **docker volume ls** 🡪 To check all the docker volumes
3. **docker volume rm <volume\_name>** 🡪 To remove a Docker Volume

**Syntax:**

**docker run -it -v <volume\_name>:<container\_path> <image\_name>**

**Exmaple:**

1. docker run -it -v <volume\_name>:/home alpine
2. docker run -it -v <volume\_name>:/var/jenkins\_home jenkin/jenkins

**Docker Prune:**

* Docker provides a prune command to clean up any resources like images, containers, volumes and networks that are not being used by any container.

**Commands:**

1. **docker system prune** 🡪 Removes all stopped containers, unused volumes, images and networks
2. **docker container prune** 🡪 For Containers
3. **docker image prune -a** 🡪 For all unused Images
4. **docker network prune** 🡪 For Networks
5. **docker volume prune** 🡪 For Volumes

**Dangling Images:**

* Any image that is not used and untagged in docker is called dangling image
* While creating images, docker will create intermediate images and containers to create a final image. These are created temporarily by docker and have no relationship with the final tagged image

**To List Dangling Images:**

1. docker images -f dangling=true
2. docker images -a

**To Remove Dangling Images:**

1. docker image prune

**Setting Resource Limits:**

* By default, docker doesn’t limit the extent of resources used

**1. Memory Limit:**

* + To limit the maximum amount of memory that a container can use, add **--memory** flag during the run time.
  + Additionally, we can set a soft limit using **--memory-reservation**.
  + Docker tries to limit the container within the soft limit, but when the need arises in case the limit is breached the docker doesn't stop any of its services

**Syntax:**

* 1. **docker run -it --memory="1g" --memory-reservation="750m" alpine**

Note: when we set memory limit for the container and the container exceeds this limit, an OOM (out of memory) error occurs which triggers the docker daemon to kill the processes running inside the container.

**--oom-kill-disable** 🡪 the container won't be stopped even after crossing the threshold

**2. CPU Limit:**

* By default, containers get unlimited access to the computing power of the host machine. if we want to limit this, we can set using 🡪 **--cpus flag**

**Syntax:**

* + **docker run -it --cpus="1.0" alpine**

**Commands:**

* **docker stats** 🡪 To check the container resource usage
* **docker system df** 🡪 To check the Docker Demon Resource Usage

**Interview Questions:**

1. Display only stopped/exited container

* **docker ps -a -f status=exited** (-f means filter)

2. Display only the Container ID's

* **docker ps -a -q**

3. Stop all the running containers

* **docker ps -q | xargs docker stop**

-q works for images also.

**Assignment:**

Check for Commands to copy files from local machine to container and from container to machine

Host to container

**docker cp ~/Desktop/to-be-copied.txt 135950565ad8:/to-be-copied.txt**

Container to host

**docker cp d362659da5fc:/opt/app/app.log /home/mkyong/backup/**

**Docker Networks:**

* By default, docker has 3 network drivers - bridge, host and none
* Advantage of having docker network is that it isolates containers from the internet. Therefore, serving as an extra layer of security

1. **bridge [--driver bridge]**

* the default network driver of docker is docker bridge and the network name is docker 0 (zero)
* To access any containers that are in bridge network from outside, we need to map the ports of this containers to the ports on host machine.

1. **host [--driver host]**

* Creating container on host network will remove any network isolation between the host machine and the containers.
* for example, if we run container on port 8080 it will be accessible on same port on host machine without any port mapping
* the containers in host network won’t get their own Ip-address
* use case: generally, this network is preferred in case of single container nodes as it provides high performance as there is no additional layer of networking

1. **Null/None Network: [--driver null]**

* the none network keeps the container in complete isolation. i.e., they are not connected to any network or container.
* use case: to run batch jobs which are scheduled programs that are assigned to run without any further interaction

**Commands:**

* **docker network ls** 🡪 To list the networks
* **docker network create --driver bridge <network\_name>** 🡪 To create a network
* **docker run -it --network <network\_name> alpine** 🡪 To create a container in a specific network
* **docker run -it –network <network-name> --name <container-name> alpine** 🡪 To create a container in a specific network with a specific name
* **docker connect <network\_name> <container\_name/container\_ID>** 🡪 To connect a container to network
* **docker disconnect <network\_name> <container\_name/container\_ID>** 🡪 To disconnect a container from a network

**Docker Multi stage build**

* One of the most important tasks while building the images is keeping the image size as low as possible
* docker multi stage builds are a way of organising docker file to minimise the size of final docker image
* in a multistage docker build, we will have multiple FROM instructions in a single docker file and each from instruction begins a new stage
* we can selectively copy artifacts from one stage to another.
* Basically, instead of keeping all the unnecessary supported libraries, dependency files, build tools using multistage builds we can discard or remove all these components to make our image and container light weight and also less vulnerable

**Example1:**

FROM alpine AS git

RUN apk update && apk add git

WORKDIR /home/app

RUN git clone https://github.com/adhig93/java\_repo1

FROM maven:amazoncorretto AS maven

COPY --from=git /home/app/java\_repo1/src /usr/app/src

COPY --from=git /home/app/java\_repo1/pom.xml /usr/app/

WORKDIR /usr/app

RUN mvn clean install

FROM tomcat-conf AS tomcat

COPY --from=maven /usr/app/target/\*.war /usr/local/tomcat/webapps/

**Example2:**

FROM maven:amazoncorretto AS stage1

COPY src /usr/src/app/src

COPY pom.xml /usr/src/app

RUN mvn -f /usr/src/app/pom.xml clean install

FROM openjdk:9

COPY --from=stage1 /usr/src/app/target/gs-maven-0.1.0.jar /usr/app/gs-maven-0.1.0.jar

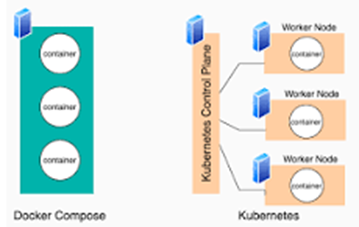
ENTRYPOINT ["java","-jar","/usr/app/gs-maven-0.1.0.jar"]

**Docker-Compose**

[**https://medium.com/clarusway/how-can-we-easily-and-visually-explain-the-docker-compose-53df77e9f046**](https://medium.com/clarusway/how-can-we-easily-and-visually-explain-the-docker-compose-53df77e9f046)

Docker Compose is **a tool that was developed to help define and share multi-container applications**. With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.

**What is docker-compose vs Kubernetes?**



Kubernetes and Docker Compose are both container orchestration frameworks. Kubernetes runs containers over a number of computers, virtual or real. Docker Compose runs containers on a single host machine

**Installation:** <https://docs.docker.com/compose/install/>

**docker-compose.yml**

version: "3.9"

services:

frontend:

image: ubuntu

volumes:

- "/home/ubuntu/ubuntu:/home"

backend:

build: .

ports:

- "80:80"

**Default Name:** docker-compose.yml

**Commands:**

* **docker-compose up** 🡪 To run the containers
* **docker-compose up -d** 🡪 To run the containers in detach mode
* **docker-compose up -f** 🡪 To use a specific file
* **docker-compose down** 🡪 to shut down all the containers
* **docker-compose ps** 🡪 To check the running containers
* [**https://docs.docker.com/compose/reference/**](https://docs.docker.com/compose/reference/)