Docker:

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications.

Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files

Difference between cloud server and on-premesis server:

In a cloud server, you don't need to buy and maintain any hardware as everything is 'handled' by the service provider, whereas the user rents or buys the server.

An on-premise server is a physical, on-site server that a company must manage and maintain individually.

https://www.backblaze.com/blog/vm-vs-containers/

https://www.geeksforgeeks.org/difference-between-virtual-machines-and-containers/

Virtual machines/Virtualization:

A virtual machine is a program on a computer that works like it is a separate computer inside the main computer. It is a simple way to run more than one operating system on the same computer. A very powerful server can be split into several smaller virtual machines to use its resources better.

Hypervisor:

A hypervisor (or virtual machine monitor, VMM, virtualizer) is a kind of emulator; it is computer software, firmware or hardware that creates and runs virtual machines. A computer on which a hypervisor runs one or more virtual machines is called a host machine, and each virtual machine is called a guest machine.

Containers/Containerization:

Containerization was developed to solve many of the problems of virtualization. The purpose of the containers is to encapsulate an application and its dependencies within its own environment. This allows them to run in isolation while they are using the same system resources and the same operating system. Since the resources are not wasted on running separate operating systems tasks, containerization allows for a much quicker, lightweight deployment of applications. Each container image could be only a few megabytes in size, making it easier to share, migrate, and move.

Advantages and disadvantages of virtual machines and Docker containers?

-===================================================================

What is a Container?

A container is a running instance of an image (A standardized unit of software).

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Docker containers that run on Docker Engine:

Standard: Docker created the industry standard for containers, so they could be portable anywhere

Lightweight: Containers share the machineâ€™s OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs

Secure: Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry

Docker container technology was launched in 2013 as an open source Docker Engine.

Docker Engine is the industryâ€™s de facto container runtime that runs on various Linux (CentOS, Debian, Fedora, Oracle Linux, RHEL, SUSE, and Ubuntu) and Windows Server operating systems. Docker creates simple tooling and a universal packaging approach that bundles up all application dependencies inside a container which is then run on Docker Engine. Docker Engine enables containerized applications to run anywhere consistently on any infrastructure, solving â€œdependency hellâ€ for developers and operations teams, and eliminating the â€œit works on my laptop!â€ problem.

Comparing Containers and Virtual Machines

Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware. Containers are more portable and efficient.

Installation:

sudo apt-get update

sudo apt-get install \

ca-certificates \

curl \

gnupg \

lsb-release

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

echo \

"deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io

post installation steps for docker:

sudo groupadd docker

sudo usermod -aG docker $USER

newgrp docker

exit and login again:

docker --version

docker run hello-world

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

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

cmds:

docker images

to download images from a registry(official image)

docker pull <image\_name>(eg:ubuntu)

to download a private docker image:

docker pull <repo\_name>/<image\_name>:<tag\_name>

login to the registry with docker hub account:

docker login

user:

pass:

if you want to push an image to your account

create the tag:

to tag an image:

docker tag local-image:tagname new-repo:tagname

docker push new-repo:tagname

docker push opqdevops/demo:tagname

docker tag <image\_id> <repo\_name>/<image\_name>:<tag\_name>

to create our own customized image out of a container:

docker commit CONTAINER\_ID REPOSITORY:TAG

docker push <repo\_name>/<image\_name>:<tag\_name>

list images:

docker images

delete images:

docker rmi <image\_id>

docker pull nginx

docker pull jenkins/jenkins:latest

docker pull tomcat

docker pull alpine

docker pull jenkins/jenkins:alpine

example running a container:

docker run -p 8085:8080 -p 50000:50000 -v /your/home:/var/jenkins\_home jenkins/jenkins:latest

docker run -dt -p 8085:8080 -p 50000:50000 -v /home/ubuntu/jenkins:/var/jenkins\_home jenkins/jenkins

docker run -p 8080:8080 -p 50000:50000 -v /home/ubuntu/jenkins\_home:/var/jenkins\_home jenkins

to run a container in interactive mode:

docker run -it ubuntu /bin/bash

to run a container in detached mode:

docker run -d ubuntu /bin/bash

to login to a running container

docker attach <container\_id>

(to get out of the container hold <ctrl> and press Q + P twice)

login to a container with new process.(execute a cmd inside a running container)

docker exec -it container\_id /bin/bash

to check logs of processes running in container:

docker logs container\_id

to list running containers

docker ps

to list all containers

docker ps -a

to start a container

docker start <container\_id>

to stop a running container

docker stop <container\_id>

to delete a container:

docker rm <container\_id>

docker images -q | xargs -I{} docker rmi {}

docker rmi --force image-id

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

https://www.freecodecamp.org/news/docker-simplified-96639a35ff36/

What is Docker?

Docker is a software platform that simplifies the process of building, running, managing and distributing applications. It does this by virtualizing the operating system of the computer on which it is installed and running.

or

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

difference between virtual machines and containers?

Docker Terminology

Let us take a quick look at some of the terminology associated with Docker.

what is docker image?

Docker Image is a template that contains the application, and all the dependencies required to run that application on Docker.

what is a docker container?

Docker Container is a running instance of the Docker Image.

What is Docker Hub?

Docker Hub is the official online repository where you could find all the Docker Images that are available for us to use.

Docker Hub also allows us to store and distribute our custom images as well if we wish to do so. We could also make them either public or private, based on our requirements.

Docker Editions

Docker is available in 2 different editions, as listed below:

Community Edition (CE)

Enterprise Edition (EE)

The Community Edition is suitable for individual developers and small teams. It offers limited functionality, in comparison to the Enterprise Edition.

The Enterprise Edition, on the other hand, is suitable for large teams and for using Docker in production environments.

The Enterprise Edition is further categorized into three different editions, as listed below:

Basic Edition

Standard Edition

Advanced Edition

Before we proceed further, letâ€™s try to decode and understand the output of the docker ps command.

CONTAINER ID: A unique string consisting of alpha-numeric characters, associated with each container.

IMAGE: Name of the Docker Image used to create this container.

COMMAND: Any application specific command(s) that needs to be executed when the container is started.

CREATED: This shows the time elapsed since this container has been created.

STATUS: This shows the current status of the container, along with the time elapsed, in its present state.

If the container is running, it will display as Up along with the time period elapsed (for example, Up About an hour or Up 3 minutes).

If the container is stopped, then it will display as Exited followed by the exit status code within round brackets, along with the time period elapsed (for example, Exited (0) 3 weeks ago or Exited (137) 15 seconds ago, where 0 and 137 are the exit codes).

PORTS: This displays any port mappings defined for the container.

NAMES: Apart from the CONTAINER ID, each container is also assigned a unique name. We can refer to a container either using its container ID or its unique name. Docker automatically assigns a unique silly name to each container it creates. But if you want to specify your own name to the container, you can do that by including the --name (double hyphen name) option to the docker create or the docker run command.

run the container in attached mode:

docker run -it --name="my\_ubuntu" ubuntu\_1 /bin/bash

run the container in detached mode

docker run -dt --name="my\_ubuntu" ubuntu\_1 /bin/bash

or

docker run -it -d --name="my\_ubuntu" ubuntu\_1 /bin/bash

to list id's of all running containers?

docker ps -q

how to stop all running containers?

docker stop $(docker ps -q)

how to delete all containers?

docker rm $(docker ps -a -q)

to delete all docker images?

docker rmi $(docker images -q)

to delete all stopped/exited containers

docker rm $(docker ps -aq -f status=exited)

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

how do you remove/delete running containers?

\*You cannot remove a running container. you need to Stop the container before attempting removal or remove forcefully.

\*the proper/standard way to remove/delete a container is to stop it first and then remove it.

docker create:

This command allows us to create a new container.

eg: docker create -it ubuntu /bin/bash

if the image isnâ€™t available on the Docker Host, it will go ahead and download the latest image from the Docker Hub before creating the container.

The options -t and -i instruct Docker to allocate a terminal to the container so that the user can interact with the container. It also instructs Docker to execute the bash command whenever the container is started.

Dockerfile:

Dockerfile is a simple text file that consists of instructions to build Docker images.

\*dockerfile is used to create a custom docker image.

\*it consists set of instructions which is used to update the base image.

\*the main purpose of dockerfile is to automate the docker image build process.

Dockerfile instructions:

FROM <image\_name>:<tag>

\*defines the base image used to start the build process.

\*must be the first non commented instruction in the dockerfile.

\*it can appear multiple times in the dockerfile, to support multistage build process.

RUN

\*This is what runs within the container at build time.

\*executes the command with no shell enviromnent(eg: alpine image don't have bash shell)

\*if there is shell it will use default shell to execute the commands, or else it will use its own feature from docker to execute the commands.

CMD:

\*can be used for executing a specific command within the container.

\*defines the default execution point for container which is created from the dockerfile.

\*there can be only one CMD in a dockerfile, if we define multiple CMD in a dockerfile only the last one will be considered and the rest of them will be ignored.

\*it can be overwritten at the run time(while executing the docker run command).

Eg: Dockerfile

FROM ubuntu

RUN apt update

CMD ["echo","this is CMD"]

to build images with dockerfile from diff location

docker build --file="/home/ubuntu/Dockerfile\_name" --tag="tagname" .

to overwrite CMD instruction at runtime:

docker build --tag="ubuntu\_1" .

docker run -it --name="my\_ubuntu" ubuntu\_1 echo "entry runtime"

ENTRYPOINT:

\*ENTRYPOINT is the other instruction used to configure how the container will run. Just like with CMD, you need to specify a command and parameters.

\*is also used to define default execution point of the container, which will be created from the dockerfile.

\*there can be only one entrypoint, if there are multiple entrypoints only the last one will be considered.

\*it cannot be overwritten completely at the run time, if we pass command at the runtime it will be considered as parameters to the command defined in the dockerfile.

FROM ubuntu

RUN apt update

ENTRYPOINT ["echo","this is the entrypoint"]

first one is the command and the rest are params to that command.

docker build --tag="my\_ubuntu\_1" .

docker run -it -d --name="my\_ubuntu" ubuntu "echo entry runtime"

what happens if we use CMD and ENTRYPOINT in the same dockerfile?

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

Please introduce your self or Tell me about your roles and responsibilities?

\* I am working as a DevOps(build and release) engineer. I majorly take care of CI/CD pipeline, I setup CI/CD pipeline from scratch. Whenever development team used to check-in their code into git-hub repository our pipeline job used to get trigger automatically. It has three stages

1. build

2. deploy

3. test

\* I have worked on build tools like ANT, maven and gradle. I have worked on shell scripts, I used to write scripts to automate manual and repetative tasks. I have worked on bug tracking(ticketing) tools like bugzilla and Jira and version control tools like svn, git and git-hub. I worked on release activites, we have releases once in a month that is..4 weeks sprint. At the time of release, we used to create a branch,merge the branches and resolve merge conflicts with the help of development team.

\* I have worked on AWS services like EC2, VPC, RDS, Cloudtrail, Cloudwatch, Cloudformation, IAM, S3, EFS, Route 53, Amazon Elasticsearch Service(ELK), KMS, SNS, ELB, AutoScaling.

\* I have worked on Docker, I've written dockerfile for creating customized images and I've reduced the size of docker Image(by multistage build).

\* I have worked on Kubernetes(K8S), I've written deployment and service yaml files. I have worked on helm charts. I used to do Rolling Updates and rollbacks.

\* I have worked on Ansible(for server configuration management) and I have written a playbook.

\* I have worked on terraform.

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

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

Instructions in dockerfile:

ADD

COPY

ENV

EXPOSE

FROM

LABEL

STOPSIGNAL

USER

VOLUME

WORKDIR

ONBUILD (when combined with one of the supported instructions above)

COPY:

It only lets you copy a local file or directory from your host (the machine building the Docker image) into the Docker image itself.

ADD:

ADD lets you do that too, but it also supports 2 other sources. First, you can use a URL instead of a local file/directory. Secondly, you can extract a tar file from the source directly into the destination.

FROM ubuntu

COPY file1 /temp/

CMD ["/bin/bash"]

FROM ubuntu

ADD file1 /temp/

CMD ["/bin/bash"]

diff in COPY and ADD:

Run tar command to create an archived named file.tar.gz for given directory name by running:

tar -czvf file.tar.gz directory/file

FROM ubuntu

COPY file.tar.gz /temp/

CMD ["/bin/bash"]

FROM ubuntu

ADD file.tar.gz /temp/

CMD ["/bin/bash"]

FROM ubuntu

COPY https://github.com/yikaus/docker-alpine-base/raw/master/rootfs.tar.xz /temp/

CMD ["/bin/bash"]

COPY failed: source can't be a URL for COPY

FROM ubuntu

ADD https://github.com/yikaus/docker-alpine-base/raw/master/rootfs.tar.xz /temp/

CMD ["/bin/bash"]

find out the diff between docker attach and docker exec?

docker exec executes a new command / create a new process in the container's environment, while docker attach just connects the standard input/output/error of the main process(with PID 1) inside the container to corresponding standard input/output/error of current terminal(the terminal you are using to run the command).

https://medium.com/bb-tutorials-and-thoughts/docker-attach-vs-exec-commands-473f591da3a8

To list all the environment variables in linux:

env

ENV:

The ENV instruction sets the environment variable <key> to the value <value>.

FROM ubuntu

ENV TEST /temp

CMD ["/bin/bash"]

FROM ubuntu

ENV TEST /temp

CMD ["env"]

ARG:

ARG are also known as build-time variables. They are only available from the moment they are â€˜announcedâ€™ in the Dockerfile with an ARG instruction up to the moment when the image is built. Running containers canâ€™t access values of ARG variables.

FROM ubuntu

ARG INPUT

ENV TEST $INPUT

CMD ["env"]

docker build --tag="myubuntu:1.1" --build-arg INPUT=/home .

FROM ubuntu

ARG INPUT OUTPUT

ENV TEST $INPUT

ENV TEST2 $OUTPUT

CMD ["env"]

To pass arguments during buildtime:

docker build --tag="myubuntu:1.1" --build-arg INPUT=/home --build-arg OUTPUT=/home/ubuntu .

To set environment variables at the run time:

docker run -it --name="my\_ubuntu\_8" --env INPUT=/home/ubuntu/demo myubuntu:1.1

to set multiple env variables at runtime:

docker run -it --name="my\_ubuntu\_8" --env INPUT=/home/ubuntu/demo --env INPUT2=/home/ubuntu/demo/test myubuntu:1.1

To pass multiple environment variables we use .env file:

docker run -it --name="my\_ubuntu\_9" --env-file /home/ubuntu/.env myubuntu:1.1

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

CMD vs ENTRYPOINT?

COPY vs ADD?

how to have default env variable in container? or

how to create env variable at container start-up?

can we override ENTRYPOINT at runtime?

yes, we can from docker 1.6 version by using --entrypoint option.

ENTRYPOINT["echo","ls -lrt"]

docker run --entrypoint [new\_command] [docker\_image] [optional:value]

\*docker run -it --entrypoint /bin/bash --name="my\_ubuntu" my\_ubuntu

\*docker run -it --entrypoint ls --name="my\_ubuntu" my\_ubuntu -lrt

find out about below instructions?

USER

VOLUME

ONBUILD

https://kapeli.com/cheat\_sheets/Dockerfile.docset/Contents/Resources/Documents/index

https://dockerlabs.collabnix.com/docker/cheatsheet/

MAINTAINER (deprecated)

The MAINTAINER instruction allows you to set the Author field of the generated images.

MAINTAINER <name>

LABEL:

The LABEL instruction adds metadata to an image.

To include spaces within a LABEL value, use quotes and backslashes as you would in command-line parsing.

If Docker encounters a label/key that already exists, the new value overrides any previous labels with identical keys.

To view an imageâ€™s labels, use the docker inspect command. They will be under the "Labels" JSON attribute.

LABEL <key>=<value> [<key>=<value> ...]

WORKDIR:

The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile. If the WORKDIR doesnâ€™t exist, it will be created even if itâ€™s not used in any subsequent Dockerfile instruction.

It can be used multiple times in a Dockerfile. If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction.

FROM ubuntu

MAINTAINER praveen@gmail.com

LABEL "com.example.vendor"="DEV\_TEAM"

WORKDIR /home/ubuntu

RUN touch file1 file2; ls -lrt

RUN pwd; rm file1; ls -lrt

ARG INPUT

ENV TEST $INPUT

ENTRYPOINT ["ls"]

FROM ubuntu

WORKDIR /home

RUN rm filename

RUN ls -lrt; cd /root; rm file1

ENTRPOINT ["env"]

EXPOSE:

The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.

The EXPOSE instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published. To actually publish the port when running the container, use the -p flag on docker run to publish and map one or more ports, or the -P flag to publish all exposed ports and map them to high-order ports.

By default, EXPOSE assumes TCP. You can also specify UDP

EXPOSE 80

To expose on both TCP and UDP, include two lines:

EXPOSE 80/tcp

EXPOSE 80/udp

what is the difference between EXPOSE and publish?

HEALTHCHECK

The HEALTHCHECK instruction has two forms:

HEALTHCHECK [<options>] CMD <command> (check container health by running a command inside the container)

HEALTHCHECK NONE (disable any healthcheck inherited from the base image)

Tells Docker how to test a container to check that it is still working.

Whenever a health check passes, it becomes healthy. After a certain number of consecutive failures, it becomes unhealthy.

The <options> that can appear are...

--interval=<duration> (default: 30s)

--timeout=<duration> (default: 30s)

--retries=<number> (default: 3)

The health check will first run interval seconds after the container is started, and then again interval seconds after each previous check completes. If a single run of the check takes longer than timeout seconds then the check is considered to have failed. It takes retries consecutive failures of the health check for the container to be considered unhealthy.

There can only be one HEALTHCHECK instruction in a Dockerfile. If you list more than one then only the last HEALTHCHECK will take effect.

The command's exit status indicates the health status of the container.

0: success - the container is healthy and ready for use

1: unhealthy - the container is not working correctly

2: reserved - do not use this exit code

When the health status of a container changes, a health\_status event is generated with the new status.

SHELL

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.

The SHELL instruction is particularly useful on Windows where there are two commonly used and quite different native shells: cmd and powershell, as well as alternate shells available including sh.

The SHELL instruction can appear multiple times. Each SHELL instruction overrides all previous SHELL instructions, and affects all subsequent instructions. For example:

SHELL ["<executable>", "<param1>", "<param2>"]

Allows an alternate shell be used such as zsh, csh, tcsh, powershell, and others.

STOPSIGNAL:

STOPSIGNAL signal

The STOPSIGNAL instruction sets the system call signal that will be sent to the container to exit. This signal can be a valid unsigned number that matches a position in the kernelâ€™s syscall table, for instance 9, or a signal name in the format SIGNAME, for instance SIGKILL.

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

Docker architecture:

Docker Engine

Also known as docker, is the core part of the whole Docker system. Docker Engine is an application which follows client-server architecture. It is installed on the host machine. There are three components in the Docker Engine:

a) Server: It is the docker daemon called dockerd. It can create and manage docker images, i.e, Containers, networks.

b) Rest API: It is used to instruct docker daemon what to do.

c) Command Line Interface (CLI): It is a client that is used to enter docker commands.

Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.

Docker Client

Docker users can interact with Docker through a client. When any docker commands runs, the client sends them to docker daemon, which carries them out. Docker API is used by Docker commands. Docker client can communicate with more than one daemon.

Docker Host

provides a complete environment to execute and run applications. It comprises of the Docker daemon, Images, Containers, Networks, and Storage/volume.

Docker Registries

It is the location where the Docker images are stored. It can be a public docker registry or a private docker registry. Docker Hub is the default place of docker images, its storesâ€™ public registry. You can also create and run your own private registry.

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

Docker â€“ Container Lifecycle

The following explains the entire lifecycle of a Docker container.

Initially, the Docker container will be in the created state.

Then the Docker container goes into the running state when the Docker run command is used.

The Docker kill command is used to kill an existing Docker container.

The Docker pause command is used to pause an existing Docker container.

The Docker stop command is used to pause an existing Docker container.

The Docker run/start command is used to put a container back from a stopped state to a running state.

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

Types of docker images:

base image:

A base image is the image that is used to create all of your container images. Your base image can be an official Docker image, such as ubuntu,Centos, or you can modify an official Docker image to suit your needs, or you can create your own base image from scratch.

scratch image:

an explicitly empty image, especially for building images "FROM scratch".

While scratch appears in Dockerâ€™s repository on the hub, you canâ€™t pull it, run it, or tag any image with the name scratch. Instead, you can refer to it in your Dockerfile. For example, to create a minimal container using scratch:

FROM scratch

COPY hello /

CMD ["/hello"]

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

dangling images:

A dangling image just means that you've created the new build of the image, but it wasn't given a new name. So the old images you have becomes the "dangling image". Those old image are the ones that are untagged and displays "<none>" on its name when you run docker images.

docker images --filter dangling=true

docker image prune

Remove all dangling images. If -a is specified, will also remove all images not referenced by any container

docker system prune

Remove all unused containers, networks, images (both dangling and unreferenced), and optionally, volumes.

By default, volumes are not removed to prevent important data from being deleted if there is currently no container using the volume. Use the --volumes flag when running the command to prune volumes as well:

docker system prune -a --volumes

docker system df == Show docker disk usage

docker system events == Get real time events from the server

docker system info == Display system-wide information

How Docker works internally?

\*cgroups

\*namespaces

\*chroot

The backbone of the Docker technology are cgroups(control groups) and kernel namespaces, both of which are features already provided in the Linux kernel. With cgroups, the Linux operating system can easily manage and monitor resource allocation for a given process and set resource limits, like CPU, memory, and network limits. This lets the Docker engine only give out 50% of the computer's memory, processors, or network, for example, to a running Docker container.

Namespaces are helpful in isolating process groups from each other. There are six default namespaces in Linux: mnt, ipc, net, usr, pid, and uts. Each container will have its own namespace and processes running inside that namespace, and will not have access to anything outside its namespace.

mnt(mount) namespace provides a root filesystem (this one can be compared to chroot I guess).

ipc(inter-process communication) provides dedicated shared memory.

net(network) namespace which allows the container to have its dedicated network stack.

usr(user) namespace (quite new) which allows a non root user on a host to be mapped with the root user within the container.

pid(process-id) namespace so the process only sees itself and its children.

uts(UNIX Time Sharing) is for setting the hostname and the domain that is visible to running processes in that namespace.

Docker containers also have network isolation (via libnetwork), allowing for separate virtual interfaces and IP addressing between containers.

namespace: wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

In short:

Cgroups = limits how much you can use;

namespaces = limits what you can see (and therefore use)

root and chroot

In a Unix-like OS, root directory(/) is the top directory. root file system sits on the same disk partition where root directory is located. And it is on top of this root file system that all other file systems are mounted. All file system entries branch out of this root. This is the systemâ€™s actual root.

But each process has its own idea of what the root directory is. By default, it is actual system root but we can change this by using chroot()system call. We can have a different root so that we can create a separate environment to run so that it becomes easier to run and debug the process. Or it may also be to use legacy dependencies and libraries for the process.

chroot changes apparent root directory for current running process and its children.

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

\*docker stats

This command is used to provide the statistics of a running container.

docker stats <ContainerID>

\*docker inspect

Return low-level information on Docker objects

docker inspect <ContainerID/imageID>

\*docker system info

Get detailed information about docker installed on the system including the kernel version, number of containers and images, etc.

\*docker history

Shows the history of a docker image with the image name mentioned in the command.

docker history <httpd>

\*docker pause

This command is used to pause the processes in a running container.

docker pause ContainerID

\*docker unpause

This command is used to unpause the processes in a running container.

docker unpause ContainerID

\*docker kill

This command is used to Kill one or more running containers.

docker kill ContainerID

Kill all runnning docker containers.

docker kill $(docker ps -q)

\*docker system prune

Delete all unused containers, unused networks, and dangling images.

\*docker system prune -a

-a is short for --all. Delete unused images, not just dangling ones.

\*docker logs:

Show the logs of the docker container with container id mentioned in the command.

docker logs -ft container\_id

\*docker update

Update container configurations.

\*docker plugin install

Install a docker plugin vieux/sshfs with debug environment set to 1.

docker plugin install vieux/sshfs DEBUG=1

\*docker search

Search for a docker image on dockerhub with the name mentioned in the command.

docker search hadoop

\*docker logout

Logging out from dockerhub.

how to limit memory and cpu to the container?

At the docker runtime need to use an option --memory or -m to limit the RAM memory usage. the container is not allowed to use more RAM than the limit. If container tries to use more than the limited memory, docker will kill the container. To avoid this behaviour we can use --oom-kill-disable

docker run -it --memory 100mb --name="my\_ubuntu" my\_ubuntu

docker stats <container-id>

OOM error (out-of-memory error)

by default docker will kill the container if the above issue occurs.

how to avoid OOM issue?

docker run -it -d --entrypoint /bin/bash --memory 100mb --oom-kill-disable --name="my\_ubuntu" my\_ubuntu

docker run -it -d --entrypoint /bin/bash --memory 100mb --memory-reservation 50mb --oom-kill-disable --name="my\_ubuntu" my\_ubuntu

hard limit(--memory 100mb) is the actual memory assigned to container, once this limit is crossed docker will kill the container(if OOM kill is not disabled).

To avoid the above issue, there is an option to reserve memory incase of emergencies called soft limit(--memory-reservation 50mb). once the hard limit memory is crossed, soft limit memory will get assigned to the container. soft limit ensures our container can still request additional memory after hitting its memory limit.

limit the cpu usage to container:

docker run -it -d --entrypoint /bin/bash --memory 100mb --memory-reservation 50mb --oom-kill-disable --cpus="0.5" --name="my\_ubuntu" my\_ubuntu

--cpus="0.5" = 50% of cpu will be used.

docker run -it -d --entrypoint /bin/bash --memory 100mb --memory-reservation 50mb --oom-kill-disable --cpu-shares=256 --name="my\_ubuntu" my\_ubuntu

what is swap memory in linux?

how to limit swap in docker container?

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

explain the dockerfile that you have worked with?

how does container stores data?

By default all files created inside a container are stored on a writable container layer on top of the underlying layers. This layer is often called the â€œctainer layerâ€. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The data doesnâ€™t persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.

when the container is deleted, all the data inside it will get lost along with the container.

Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container stops: volumes, and bind mounts. If youâ€™re running Docker on Linux you can also use a tmpfs mount. If youâ€™re running Docker on Windows you can also use a named pipe.

Volumes are stored in a part of the host filesystem which is managed by Docker (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.

Bind mounts may be stored anywhere on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.

tmpfs mounts are stored in the host systemâ€™s memory only, and are never written to the host systemâ€™s filesystem.

Persistent storage:

Persistent storage is any data storage device that retains data after power to that device is shut off. It is also sometimes referred to as nonvolatile storage.

eg: hdd(hard disk drive) or ssd(solid state drive)

Non-Persistent storage:

Non-Persistent storage is any data storage device that will not retain data after power to that device is shut off. It is also sometimes referred to as volatile storage.

eg: RAM, ROM

Bind mounts:

Bind mounts have limited functionality compared to volumes. When you use a bind mount, a file or directory on the host machine is mounted into a container.The file or directory is referenced by its absolute path on the host machine.

docker run -it -d --rm --volume /home/ubuntu/bind\_mount:/example ubuntu /bin/bash

--rm :

when the above option is used, container will get removed automatically when exited from it.

docker volumes:

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

Volumes are easier to back up or migrate than bind mounts.

You can manage volumes using Docker CLI commands or the Docker API.

Volumes work on both Linux and Windows containers.

\*\*Volumes can be more safely shared among multiple containers.

Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.

New volumes can have their content pre-populated by a container.

Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

to create a docker volume

docker volume create my\_volume

to list the volumes:

docker volume ls

how to check the content of the volume:

docker inspect my\_volume

\*\*volumes does not have any IDs, can only be tracked by names

default location of the volumes:

/var/lib/docker/volumes

how to attach docker volume to a container:

docker run -it -d --rm -v my\_volume:/container\_path ubuntu /bin/bash

docker exec -it container\_id /bin/bash

\*\*volumes can be shared by multiple containers.

can we share bindmounts between multiple containers?

how to delete a volume:

docker volume rm my\_volume

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

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

list the process running inside the contaier:

docker exec -it container\_id ps aux

docker exec -it container\_id top -H

how to cleanup unused data ?

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

Docker network:

\*Docker takes care of the networking aspects so that the containers can communicate with other containers and also with the Docker Host.

\*If you do an ifconfig on the Docker Host, you will see the Docker Ethernet adapter. This adapter is created when Docker is installed on the Docker Host.

\*Docker includes support for networking containers through the use of network drivers.

\*Every installation of the Docker Engine automatically includes three default networks.

docker network ls

1.bridge

2.host

3.none

bridge network:

docker bridge network can be a hardware or software device running within a host machine which forwards the traffic between network.

docker uses a software bridge network by default. the name of the network is docker0.

by default all the containers created without network configuration will be attached to the default bridge network.

Differences between user-defined bridge and the default bridge:

\*User-defined bridges provide automatic DNS resolution between containers.

Containers on the default bridge network can only access each other by IP addresses. On a user-defined bridge network, containers can resolve each other by name or alias.

\*User-defined bridges provide better isolation.

All containers without a --network specified, are attached to the default bridge network. This can be a risk, as unrelated stacks/services/containers are then able to communicate.

\*Containers can be attached and detached from user-defined networks on the fly.

During a containerâ€™s lifetime, you can connect or disconnect it from user-defined networks on the fly. To remove a container from the default bridge network, you need to stop the container and recreate it with different network options.

\*Each user-defined network creates a configurable bridge.

If your containers use the default bridge network, you can configure it, but all the containers use the same settings, such as MTU and iptables rules. In addition, configuring the default bridge network happens outside of Docker itself, and requires a restart of Docker.

User-defined bridge networks are created and configured using docker network create. If different groups of applications have different network requirements, you can configure each user-defined bridge separately, as you create it.

By default, Docker always launches your containers in bridge network

docker run -itd --name=networktest ubuntu

docker network inspect bridge

Create your own bridge network:

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

docker network create --driver bridge my\_bridge

docker network ls

docker network inspect my\_bridge

Add containers to a network at the runtime:

docker run -it -d --network=my\_bridge --name="ubuntu1" ubuntu /bin/bash

containers under same bridge network can communicate with each other:

docker run -it -d --network=my\_bridge --name="ubuntu1" ubuntu /bin/bash

docker run -it -d --network=my\_bridge --name="ubuntu2" ubuntu /bin/bash

docker inspect ubuntu4

IP:172.18.0.2

docker inspect ubuntu5

IP:172.18.0.3

docker exec -it ubuntu1 /bin/bash

ping ubuntu2

or

docker exec -it ubuntu1 ping 172.17.0.3

\*containers under different bridge networks cannot communicate with each other.

To make them communicate, atleast one container from bridge1 network must be connected to bridge2 network.

docker network create --driver bridge my\_bridge\_2

docker run -it -d --network=my\_bridge\_2 --name="ubuntu3" ubuntu /bin/bash

docker run -it -d --network=my\_bridge\_2 --name="ubuntu4" ubuntu /bin/bash

Connects a container to a network:

docker network connect <network\_name> <container\_name>

Disconnect a container from a network:

docker network disconnect <network\_name> <container\_name>

To remove a network:

docker network rm <network\_name>

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

host network:

\*If you use the host network mode for a container, that containerâ€™s network stack is not isolated from the Docker host (the container shares the hostâ€™s networking namespace), and the container does not get its own IP-address allocated.

For instance, if you run a container which binds to port 80 and you use host networking, the containerâ€™s application is available on port 80 on the hostâ€™s IP address.

Note: Given that the container does not have its own IP-address when using host mode networking, port-mapping does not take effect, and the -p, --publish, -P, and --publish-all option are ignored, producing a warning instead:

WARNING: Published ports are discarded when using host network mode

\*Host mode networking can be useful to optimize performance, and in situations where a container needs to handle a large range of ports, as it does not require network address translation (NAT), and no â€œuserland-proxyâ€ is created for each port.

\*The host networking driver only works on Linux hosts, and is not supported on Docker Desktop for Mac, Docker Desktop for Windows, or Docker EE for Windows Server.

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

None network:

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

\*In some cases, you need to isolate a container even from ingoing/outgoing traffic, you can use this type of network that lacks a network interface.

Overlay Network:

\*Your container platform may have different hosts, and in each there are some containers running. These containers may need to communicate with each other. This is when overlay networks are useful.

\*The overlay network is a distributed network created among multiple Docker daemons in different hosts. All containers connected to this network can communicate.

Macvlan Network:

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 hostâ€™s network stack.

summary:

\*User-defined bridge networks are best when you need multiple containers to communicate on the same Docker host.

\*Host networks are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.

\*Overlay networks are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.

\*Macvlan networks are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.

\*Third-party network plugins allow you to integrate Docker with specialized network stacks.

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

Dockerfile to install tomcat:

FROM ubuntu:latest

RUN apt-get -y update && apt-get -y upgrade

RUN apt-get -y install openjdk-8-jdk wget

RUN mkdir /usr/local/tomcat

RUN wget https://mirrors.estointernet.in/apache/tomcat/tomcat-8/v8.5.70/bin/apache-tomcat-8.5.70.tar.gz -O /tmp/tomcat.tar.gz

RUN cd /tmp && tar xvfz tomcat.tar.gz

RUN cp -Rv /tmp/apache-tomcat-8.5.70/\* /usr/local/tomcat/

EXPOSE 8080

CMD /usr/local/tomcat/bin/catalina.sh run

docker build --tag="tomcat:8.5" -f <filename> .

docker run -itd -p <system\_port>:<container\_port> <image-id>

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

docker compose:

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your applicationâ€™s services. Then, with a single command, you create and start all the services from your configuration.

According to the Docker documentation, the most popular features of Docker Compose are:

\*Multiple isolated environments on a single host

\*Preserve volume data when containers are created

\*Only recreate containers that have changed

\*Variables and moving a composition between environments

\*Orchestrate multiple containers that work together

Installation:

sudo curl -L "https://github.com/docker/compose/releases/download/1.29.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

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

sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose

Uninstallation:

sudo rm /usr/local/bin/docker-compose

pip uninstall docker-compose

Docker Compose file structure:

version: "3.8"

services:

databases:

image: mysql

ports:

- 3306:3306

environment:

- MYSQL\_ROOT\_PASSWORD=password

- MYSQL\_USER=user

- MYSQL\_PASSWOD=password

- MYSQL\_DATABASE=demodb

webpage:

image: nginx

ports:

- 5000:80

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

version: '3'

services:

product-service:

build: ./product

volumes:

- ./product:/usr/src/app

ports:

- 5002:80

website:

image: php:apache

volumes:

- ./website:/var/www/html

ports:

- 5003:80

depends\_on:

- product-service

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

build: build it from the Dockerfile in the current directory

links: a link to the database container (database\_default)

volumes:

.:/app:rw maps the parent directory on the host to /app in the container, with read and write access

/data:/data:rw maps the data directory on the host to /data in the container, with read and write access

command: by default, when the command docker-compose run is issued, execute python manage.py runserver 0.0.0.0:80 (this will override the CMD instruction in the Dockerfile)

env\_file: use the .env-local to supply environment variables to the container

version â€˜3â€™: This denotes that we are using version 3 of Docker Compose, and Docker will provide the appropriate features. At the time of writing this article, version 3.8 is latest version of Compose.

services: This section defines all the different containers we will create. In our example, we have two services, web and database.

web: This is the name of our Flask app service. Docker Compose will create containers with the name we provide.

ports: This is used to map the containerâ€™s ports to the host machine.

volumes: This is just like the -v option for mounting disks in Docker. In this example, we attach our code files directory to the containersâ€™ ./code directory. This way, we wonâ€™t have to rebuild the images if changes are made.

links: This will link one service to another. For the bridge network, we must specify which container should be accessible to which container using links.

image: If we donâ€™t have a Dockerfile and want to run a service using a pre-built image, we specify the image location using the image clause. Compose will fork a container from that image.

environment: The clause allows us to set up an environment variable in the container. This is the same as the -e argument in Docker when running a container.

examples to try:

https://medium.com/rate-engineering/using-docker-containers-to-run-a-distributed-application-locally-eeabd360bca3

https://www.baeldung.com/ops/docker-compose

https://github.com/softwareyoga/docker-tomcat-tutorial

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

\* to scale up the number of containers required for a service:

docker-compose scale <service\_name>=<no. of containers>

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

multistage build:

With multi-stage builds, you use multiple FROM statements in your Dockerfile. Each FROM instruction can use a different base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, leaving behind everything you don't want in the final image.

eg1:

FROM maven:3.6.3-openjdk-8 as buildprobe

RUN git clone https://github.com/opqdevops/java-tomcat-maven-example.git && cd java-tomcat-maven-example

WORKDIR /java-tomcat-maven-example

RUN mvn package && ls -l /java-tomcat-maven-example && ls -l /java-tomcat-maven-example/target

FROM tomcat:9.0.21-jdk8-openjdk

RUN rm -rf /usr/local/tomcat/webapps/\*

COPY --from=buildprobe /java-tomcat-maven-example/target/\*.war /usr/local/tomcat/webapps

https://docs.docker.com/develop/develop-images/multistage-build/

https://codefresh.io/docker-tutorial/webinar-docker-multi-stage-build/

eg2:

https://medium.com/@chzbrgr71/docker-multi-stage-builds-size-matters-e71ae0f08ae8

# build stage

FROM golang AS build-env

WORKDIR /app

ADD ./\*.go /app/

RUN cd /app && go build -o web-app

# final stage

FROM scratch

WORKDIR /app

COPY --from=build-env /app/web-app /app/

CMD ["/app/web-app"]

advantages of multistage build?

https://www.cloudsavvyit.com/9260/what-are-multi-stage-docker-builds/

sample-devops project to deploy war file to tomcat container:

https://github.com/ValaxyTech/DevOpsDemos/blob/master/SimpeDevOpsProjects/Project-3.MD

docker file to install maven inside jenkins container:

FROM jenkins/jenkins:lts

# Setting Maven Version that needs to be installed

ARG MAVEN\_VERSION=3.5.4

# Changing user to root to install maven

USER root

# Maven

RUN curl -fsSL https://archive.apache.org/dist/maven/maven-3/$MAVEN\_VERSION/binaries/apache-maven-$MAVEN\_VERSION-bin.tar.gz | tar xzf - -C /usr/share \

&& mv /usr/share/apache-maven-$MAVEN\_VERSION /usr/share/maven \

&& ln -s /usr/share/maven/bin/mvn /usr/bin/mvn

ENV MAVEN\_VERSION=${MAVEN\_VERSION}

ENV M2\_HOME /usr/share/maven

ENV maven.home $M2\_HOME

ENV M2 $M2\_HOME/bin

ENV PATH $M2:$PATH

# Again using non-root user i.e. stakater as set in base image

USER jenkins

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

https://medium.com/@saurav200892/getting-started-with-docker-swarm-36c9622d4759