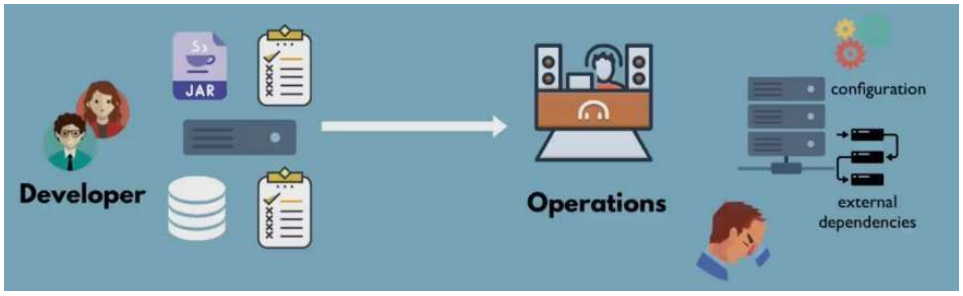
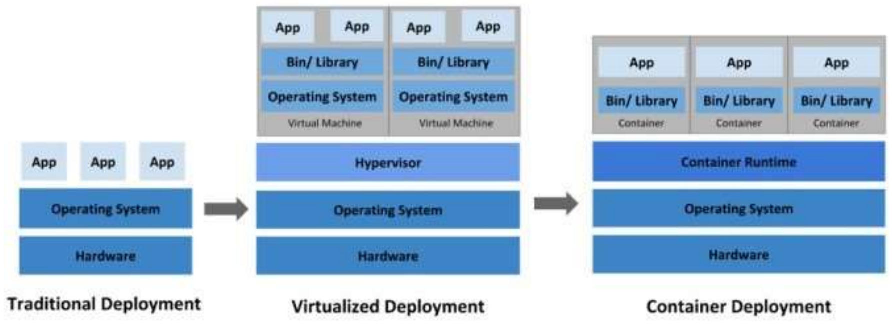
<u>Introduction to Containers</u> [Docker]

- What is Application Deployment?
    1. The process of deploying an application over executable environment for executing the application
    2. Application deployment example
       Refer Traditional deployment in the following section
    3. Application deployment before containers
        - ✓ Before the containers, in the traditional deployment process, deployment team produce artifact with a set of instructions which include how to deploy these artifact and database services and maybe some other services which needed for the application
        - ✓ The development team handover those artifacts to the production team to set up the environment to deploy the application
        - ✓ In that kind of situation, first, we have to configure the server environment, and other disadvantage is dependency conflicts can occur in the traditional deployment process

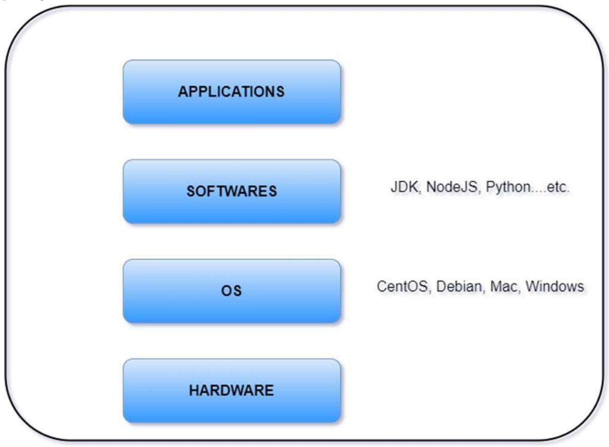


    4. Application deployment transition over the period of time

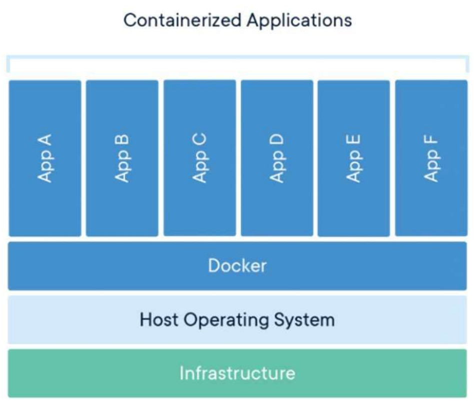


- **What is Docker?**
    1. Docker is an open platform for developing, shipping, and running applications
    2. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly

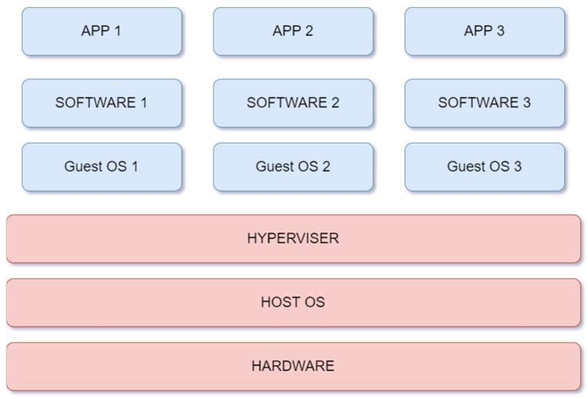
- **Installing Docker**
    1. Go to https://docs.docker.com/engine/install/ and download as per your machine's Operating System installed [CentOS, Debian, Mac or Windows]
    2. Dow
- **Deploying a spring boot application**
    - ❖ Traditional deployment



    - ❖ Docker style of Deployment
        1. Go to https://hub.docker.com/ and see the public default docker registry [have valid credentials]
        2. How can we locate our application?
            - ✓ Go to your own repository [anandmikkili/training]
            - ✓ Look for Tags and find your required application and its release version
        3. Docker image is a static template – a set of bytes [Image Vs Container]
        4. When it's downloaded also it is static template
        5. When the image is running then it's called **Container.** Container is the instance of image
        6. For the same image we can create multiple containers by just running the same on different ports

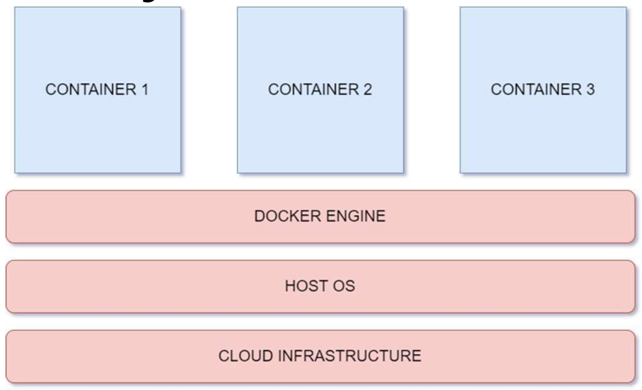


- **Deploying over Virtual Machines**

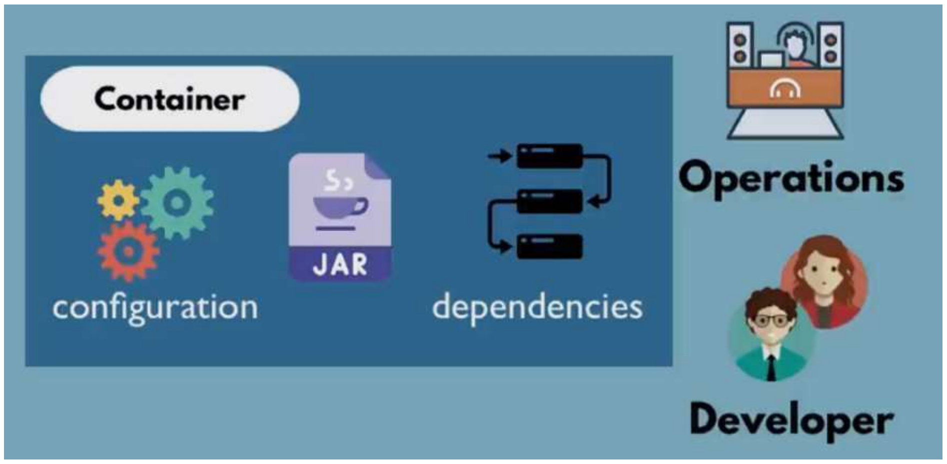


  - ❖ VMs heavily weighted machines since it carries Host OS as well as Guest OS

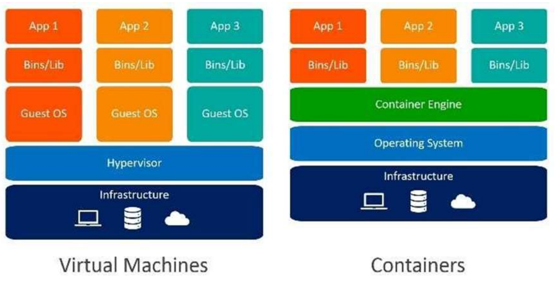
- **Deploying over Docker Engine**

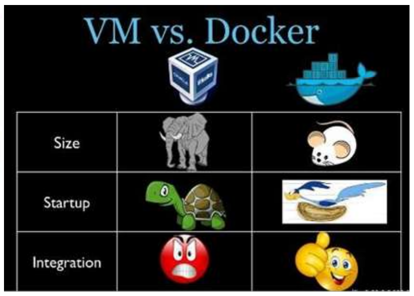


  - ❖ Docker Engine can take care of all the containers, local images and also would be communicating to Image Registry depending upon the need
  - ❖ Because just there is only one OS (Host OS), it is relatively light weight
  - ❖ Using Docker on local machines as well as over Cloud is very easy
  - ❖ AWS (Elastic Container Service), Azure (Azure Container Service)



- **VM vs Docker**

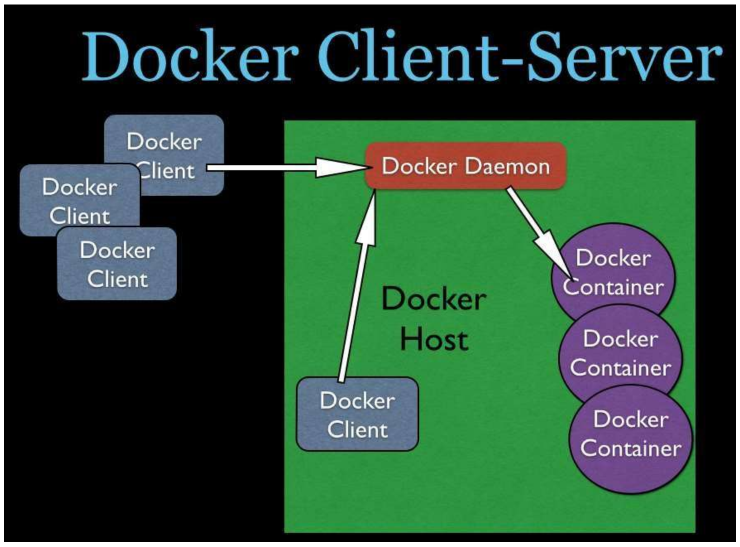




Aspects:

1. Better Utilization: It was also possible to run multiple virtual machines on the same hardware so the utilization increased from 5%. But, we still need to have a virtual machine per service so the we cannot utilize the machine as much as we would want
2. Dependency Management: Installing multiple independent services on a single machine, real or virtual, is a recipe for disaster
3. Speed

- **Understanding Docker Architecture [Docker Client, Docker Engine]**
  - ❖ Docker is implemented as a client-server system; The Docker daemon runs on the Host and it is accessed via a socket connection from the client
  - ❖ The client may, but does not have to, be on the same machine as the daemon. The Docker CLI client works the same way as any other client but it is usually connected through a Unix domain socket instead of a TCP socket



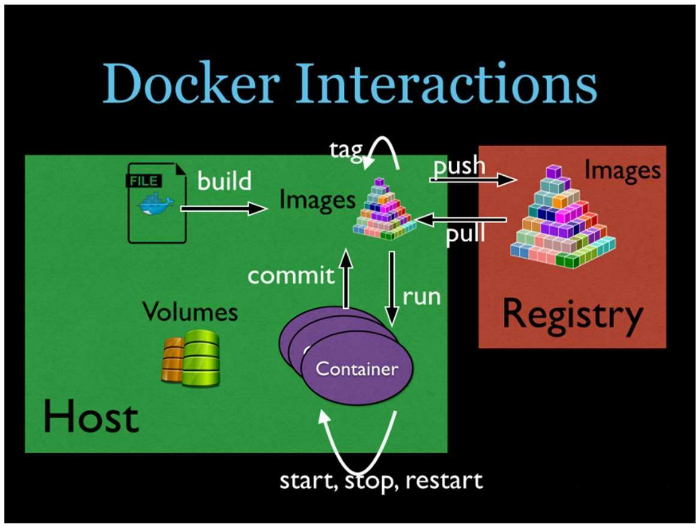
Docker Daemon:

- ❖ The daemon receives commands from the client and manages the containers on the Host where it is running
- ❖ Responsible for maintaining Containers & Local Images
- ❖ Responsible for pulling the images from Image Registry if required or pushing locally built images to Image Registry

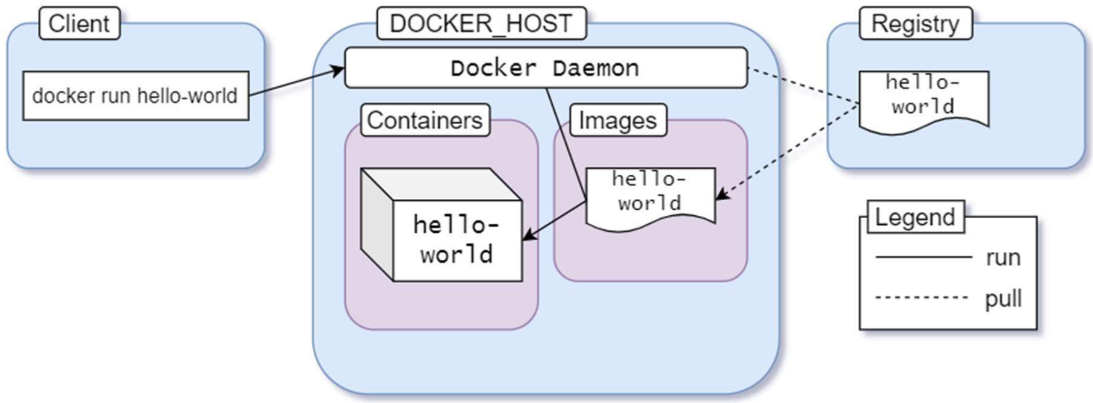
Docker Concepts:

- Host, the machine that is running the containers
- **Image**, a hierarchy of files, with meta-data for how to run a container
    1. An image is a file structure, with meta-data for how to run a container
    2. The image is built on a union filesystem, a filesystem built out of layers
    3. Every command in the Dockerfile creates a new layer in the filesystem
    4. When a container is started all images are merged together into what appears to the process as unified. When files are removed in the union file system they are only marked as deleted. The files will still exist in the layer where they were last present
- Container, a contained running process, started from an image
    1. When a container is started, the process gets a new writable layer in the union file system where it can execute
    2. It is also possible to make this layer read-only, forcing us to use volumes for all file output such as logging, and temp-files
- Registry, a repository of images
- Volume, storage outside the container
- Dockerfile, a script for creating images

Docker Interactions:



Docker Complete View:



- **Docker Commands**
  - ❖ How to run the application over docker
    1. Normal Mode
       docker run -p 5000:5000 muralisocial123/training/rest-api:1.0.0.RELEASE
    2. Detached/Daemon Mode
       docker run -p 5000:5000 -d
       muralisocial123/training/rest-api:1.0.0.RELEASE
  - ❖ How to check traces/logs
    1. docker logs imageid #entire log
    2. docker logs -f imageid #tailing the log
  - ❖ Containers
    1. docker container ls #all the docker images which are up and running
    2. docker container ls -a #all the docker images which are up and exited
    3. docker create # creates a container but does not start it
    4. docker run -p 5000:5000 muralisocial123/training/rest-api:1.0.0.RELEASEis same as docker container run -p 5000:5000 muralisocial123/training/rest-api:1.0.0.RELEASE
    5. docker container stop container_id #to stop the particular docker container
    6. docker container start # will start it again
    7. docker container restart # restarts a container
    8. docker container pause container_id   #to pause the container – no response from APIs can be observed here
    9. docker container unpause container_id #to resume the container – you can see APIs responding back
    10. docker container inspect container_id #meta data about container
    11. docker container prune #this would remove all the stopped containers
    12. docker container stop container_id   #graceful shutdown of container]

13. docker container kill container_id #container would be terminated as it is
14. docker attach # will connect to a running container
15. docker wait # blocks until container stops
16. docker exec # executes a command in a running container
17. docker run -p 5000:5000 -d --restart=always muralisocial123/training/rest-api-h2:1.0.0.RELEASE #whenever dockerdaemon restarts, this image would be made available if it was terminated
18. So even with restart policy, once stopped & prune command is applied over container, there is no way to bring it up again

❖ Images
1. docker images #shows all images
2. docker build #creates image from Dockerfile
3. docker image history image_id #list changes of an image
4. docker image inspect image_id
5. docker image remove image_id # removes an image from (local)
6. docker import #creates an image from a tarball
7. docker commit #creates image from a container

❖ Tags:
1. We can create multiple tags for the same image
2. docker tag muralisocial123/training/rest-api:1.0.0.RELEASE
3. muralisocial123/training/rest-api:latest
4. Latest .... generally most recent tag & 1.0.0. RELEASE .... latest release tag

❖ Pull:
1. docker pull mysql #just pulls the image from Image Registry to our local
2. docker run will deploy the image if present in local, otherwise it will pull first and then deploys the image

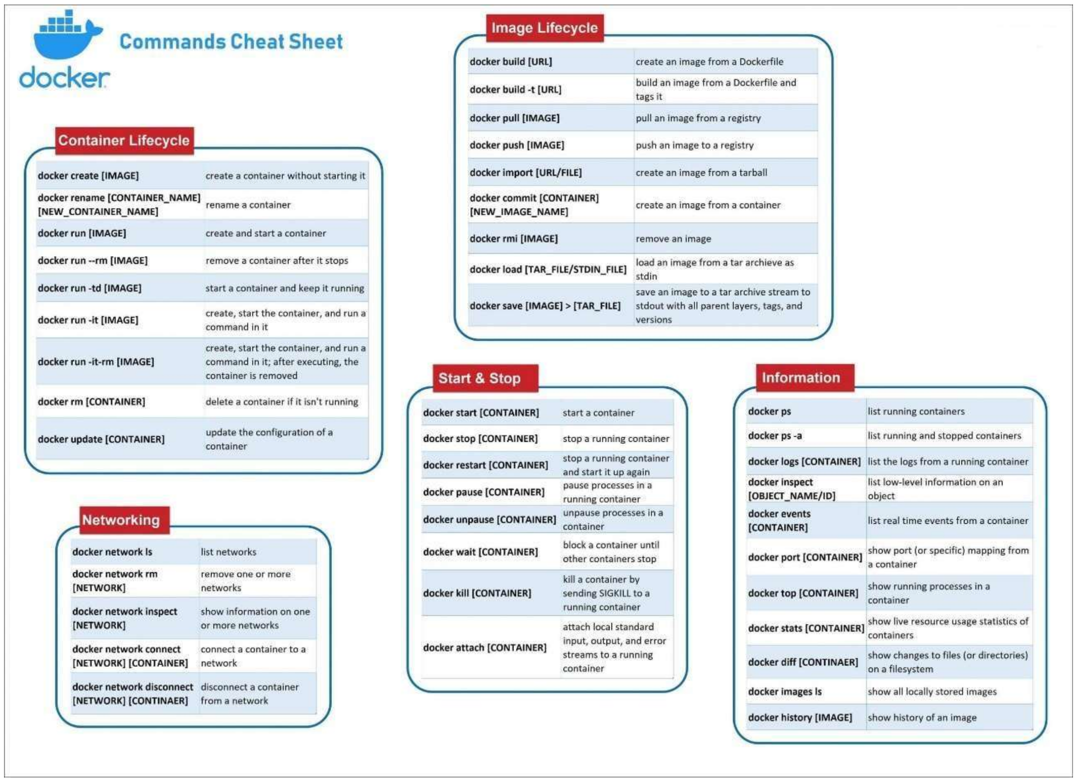
❖ Push:

❖ Searching over Image Registry
1. docker search mysql

❖ Docker Events
1. docker events #would help us to know what are all the events happening over docker engine or docker environment

❖ Docker top
1. docker top containe_id #lists all the processes running inside that container]

- ❖ Docker stats
  1. docker stats #memory statistics of all the containers [CPU, MEMORY]
- ❖ Limiting Resources to a docker image [MEMORY & CPU]
  1. docker run -p 5000:5000 -m 512m –cpu-quota 5000 -d muralisocial123/training/rest-api:1.0.0.RELEASE
  2. cpu-quota 100000=100% so 5000=5% of CPU for the above command
- ❖ Docker Daemon Management
  1. docker system df #all the docker daemon management things likes images, containers, volumes


Commands Cheat Sheet

docker

Container Lifecycle

docker create [IMAGE]	create a container without starting it
docker rename [CONTAINER_NAME] [NEW_CONTAINER_NAME]	rename a container
docker run [IMAGE]	create and start a container
docker run --rm [IMAGE]	remove a container after it stops
docker run -td [IMAGE]	start a container and keep it running
docker run -it [IMAGE]	create, start the container, and run a command in it
docker run -it-rm [IMAGE]	create, start the container, and run a command in it; after executing, the container is removed
docker rm [CONTAINER]	delete a container if it isn't running
docker update [CONTAINER]	update the configuration of a container

Networking

docker network ls	list networks
docker network rm [NETWORK]	remove one or more networks
docker network inspect [NETWORK]	show information on one or more networks
docker network connect [NETWORK] [CONTAINER]	connect a container to a network
docker network disconnect [NETWORK] [CONTINAER]	disconnect a container from a network

Image Lifecycle

docker build [URL]	create an image from a Dockerfile
docker build -t [URL]	build an image from a Dockerfile and tags it
docker pull [IMAGE]	pull an image from a registry
docker push [IMAGE]	push an image to a registry
docker import [URL/FILE]	create an image from a tarball
docker commit [CONTAINER] [NEW_IMAGE_NAME]	create an image from a container
docker rmi [IMAGE]	remove an image
docker load [TAR_FILE/STDIN_FILE]	load an image from a tar archieve as stdin
docker save [IMAGE] > [TAR_FILE]	save an image to a tar archive stream to stdout with all parent layers, tags, and versions

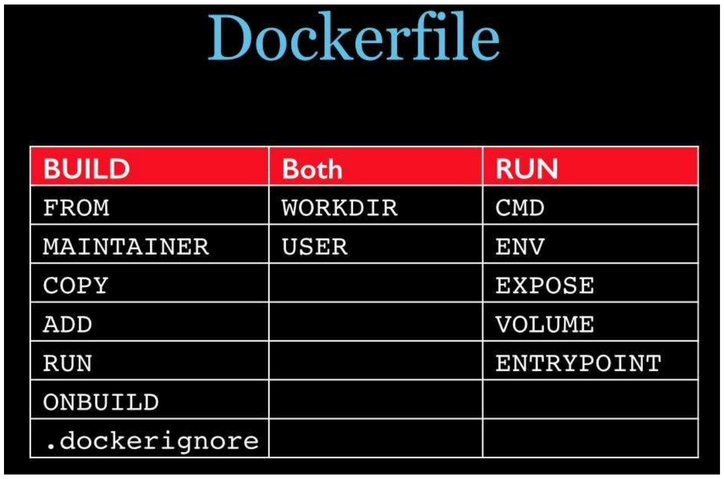
Start & Stop

docker start [CONTAINER]	start a container
docker stop [CONTAINER]	stop a running container
docker restart [CONTAINER]	stop a running container and start it up again
docker pause [CONTAINER]	pause processes in a running container
docker unpause [CONTAINER]	unpause processes in a container
docker wait [CONTAINER]	block a container until other containers stop
docker kill [CONTAINER]	kill a container by sending SIGKILL to a running container
docker attach [CONTAINER]	attach local standard input, output, and error streams to a running container

Information

docker ps	list running containers
docker ps -a	list running and stopped containers
docker logs [CONTAINER]	list the logs from a running container
docker inspect [OBJECT_NAME/ID]	list low-level information on an object
docker events [CONTAINER]	list real time events from a container
docker port [CONTAINER]	show port (or specific) mapping from a container
docker top [CONTAINER]	show running processes in a container
docker stats [CONTAINER]	show live resource usage statistics of containers
docker diff [CONTINAER]	show changes to files (or directories) on a filesystem
docker images ls	show all locally stored images
docker history [IMAGE]	show history of an image

- **Dockerfile**
  - ❖ Dockerfile is a configuration file (blue print) which is used to build Docker images from project
  - ❖ Dockerfile Commands: The Dockerfile supports 13 commands. Some of the commands are used when you build the image and some are used when you run a container from the image



Dockerfile

BUILD	Both	RUN
FROM	WORKDIR	CMD
MAINTAINER	USER	ENV
COPY		EXPOSE
ADD		VOLUME
RUN		ENTRYPOINT
ONBUILD		
.dockerignore		

BUILD Commands:

1. FROM – The image the new image will be based on
2. MAINTAINER – Name and email of the maintainer of this image
3. COPY – Copy a file or a directory into the image
4. ADD – Same as COPY, but handle URL:s and unpack tarballs automatically
5. RUN – Run a command inside the container, such as apt-get install
6. ONBUILD – Run commands when building an inherited Dockerfile
7. .dockerignore – Not a command, but it controls what files are added to the build context. Should include .git and other files not needed when building the image

RUN Commands:

1. CMD – Default command to run when running the container, can be overridden with command line parameters
2. ENV – Set environment variable in the container
3. EXPOSE – Expose ports from the container. Must be explicitly exposed by the run command to the Host with -p or -P
4. VOLUME – Specify that a directory should be stored outside the union file system. If is not set with docker run -v it will be created in /var/lib/docker/volumes
5. ENTRYPOINT – Specify a command that is not overridden by giving a new command with docker run image cmd. It is mostly used to give a default executable and use commands as parameters to it

Both BUILD and RUN Commands:

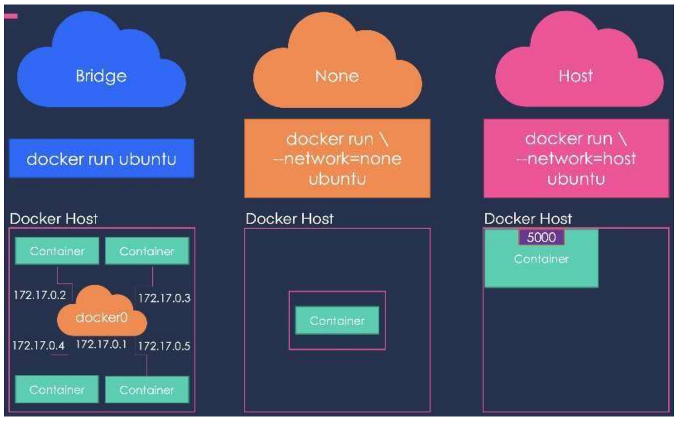
1. USER – Set the user for RUN, CMD and ENTRYPOINT
2. WORKDIR – Sets the working directory for RUN, CMD, ENTRYPOINT, ADD and COPY

- **Docker Volume**
  - ❖ Why we need volumes?
    - Everyone knows that a container has its own isolated filesystem and we also saw that it’s built of different layers; but we haven’t yet gotten into volumes
    - A data volume allows container to bypass the Union Filesystem
    - Volumes are initialized when a container is created and they provide features to persist and share your data
    - Every volume has a mount point and if there is existing data it will be copied into the container’s volume to be available and usable
  - ❖ Creating an independent volume
    - docker volume create –name mydata # local volume which is outside of container
    - docker run -it –rm -v mydata:/c_mydata ubuntu #will get you into container and can verify files
    - docker volume ls
    - docker volume inspect mydata
    - docker run -it –rm -v mydata:/c_mydata1 ubuntu #creates new container c_mydata1 and copies the data from local file system to union file system [container]
  - ❖ Creating volume that persists the data when the container is removed
    - docker run -it --name=Container2 -v mydata2:/c_mydata2 ubuntu
    - docker volume ls
    - docker exec –it container_id bash #to get into container bash shell
    - docker volume rm –f volume_id #forceful removal of volume

    Start and stop the container, still the data should be available in the volume

  - ❖ Creating a volume from existing directory with data [container data to host machine]
    - docker -it -rm --name=Container3 -v mydata3:/opt/c_mydata3 ubuntu
  - ❖ Sharing data between multiple containers
    - docker -it -rm --name=Container4 -v mydata4:/opt/c_mydata4 ubuntu
    - echo "My data4 is getting stored in file" > /c_mydata4/Example4.txt
    - docker run –it --name=Container5 --volumes-from Container4 ubuntu
    - docker volume inspect mydata4
  - ❖ Creating a bind volume
    - docker -it -rm -v /mydata5:/c_mydata5 ubuntu
    - echo "My data5 is getting stored in file" > /c_mydata5/Example5.txt

- **Docker Networking**
  - ❖ Why we need networking?
    - Docker networking allows communication between applications
    - Docker supports 3 networks
      1. Bridge Network [default network]
      2. Host Network
      3. None



  - ❖ Commands
    - docker network ls #lists all the networks
    - docker network rm network_name
    - brctl show docker0   #shows how many interfaces are present to the network (only for bridge network)
    - docker network inspect bridge
- ❖ Inter container communication – Default Network
    - docker run –it --name container1 ubuntu
    - docker run –it --name container2 ubuntu
    - docker exec –it container1_id bash   #loginto any container
    - ping IP_Address of second container
  - ❖ Inter container communication – Custom Network
    - Create custom network
      - ✓ docker network create mynetwork –subnet=10.0.0.1/16 --gateway=10.0.10.100
      - ✓ docker network ls
    - How to run docker container with custom network
      - ✓ docker run –it --net mynetwork --name myubuntu3 ubuntu
      - ✓ docker network inspect mynetwork
    - Communicating Default Network from Custom Network
      - ✓ docker exec –it container_id [myubuntu3]
      - ✓ Just ping IP of container1   #2 networks are different, so no communication possible
      - ✓ docker network connect bridge myubuntu3
      - ✓ docker exec –it container_id [myubuntu3]

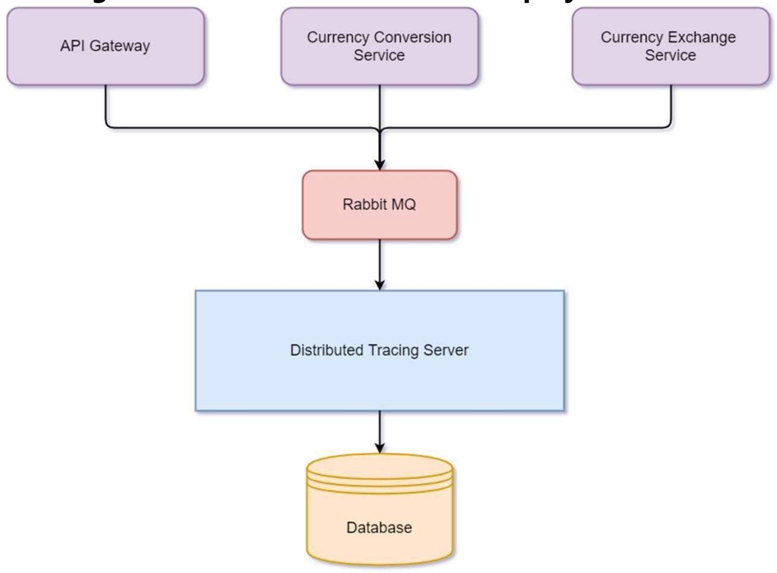
    - ✓ ping IP of container1 #it will work this time

- **Building Docker Images for Micro Services**
    - ❖ Micro Services is always involving in complex call chain
    - ❖ How can we debug the problems?
    - ❖ How can we trace the requests across the microservices?

  Commands:

    - ❖ Configure build in pom.xml of the project
    - ❖ spring-boot:build-image -DskipTests [Run As -> Maven Command to build image]
    - ❖ Docker Compose [for CentOS, Debian…etc. For Windows & Mac it is not required to install]
    - ❖ docker-compose.yml

Distributed Tracing in Microservice Architecture Deployment


API Gateway
Currency Conversion Service
Currency Exchange Service
Rabbit MQ
Distributed Tracing Server
Database

- ❖ Spring Cloud sleuth will trace each request by assigning a unique id across the microservices
- ❖ Ideally we don't want to trace every request since it creates a performance issue. So only we would only trace sample percentage of requests
- ❖ Docker Link for Zipkin: https://hub.docker.com/r/openzipkin/zipkin
- ❖ Installation:
    - docker run -p 9411:9411 openzipkin/zipkin:2.23