docker

Containers

Shipping Solution

Containers

- The developer worries about what's inside the container: Code, libraries, package manager, apps, data
- The Ops worry about what's outside the container: Logging, remote access, monitoring, network config
- All containers start, stop, copy, attach, migrate the same way

Docker

- Build, ship & run any software any where
- Docker is a tool designed to create, deploy, and run applications with ease by using containers
- It allows developers to package an application with all the requirements such as libraries and other dependencies, ship it all as one package
- It ensures that your application works seamlessly in any environment: Development, Test, or Production
- Dockerized apps and dependencies can be shipped anywhere

Docker

- Docker file build a docker image which contains all the project code
- You can run that image to create as many docker containers as you want
- The created images can be uploaded on docker hub from where the image can be pulled and built in a container
- Docker file → docker image → docker container → any environment

What is an image?

- An image is a text file with a set of pre-written commands, usually called as a docker file
- Docker images are made up of multiple layers which are read-only filesystem
- A layer is created for each instruction in a docker file and placed on top of the previous layer
- When an image is turned into a container the docker engine takes the image and adds the read-write filesystem on top (as well as initializing various settings such as the IP address, name, ID, and resource limits)

What is an image?

Docker Images

- Read only template used to create containers
- Built by Docker users
- Stored in DockerHub or your local registry

Docker Containers

- Isolated application platform
- Contains everything needed to run the application
- Built from one or more images

Few basic commands

- *docker help*
- *docker version*
- *docker system info, docker system df, docker system df -v*
- *docker images* → displays a list of existing images in docker system
  - *Repository*
  - *Tag*
  - *Image ID*
  - *Created*
  - *Size*

Few basic commands

- *docker ps* → displays the list of active containers
  - Container ID
  - Image
  - Command
  - Created
  - Status
  - Ports
  - Names
- *docker ps -a* → Display the list of all the container processes which are running or have run in the past

Hello-World image

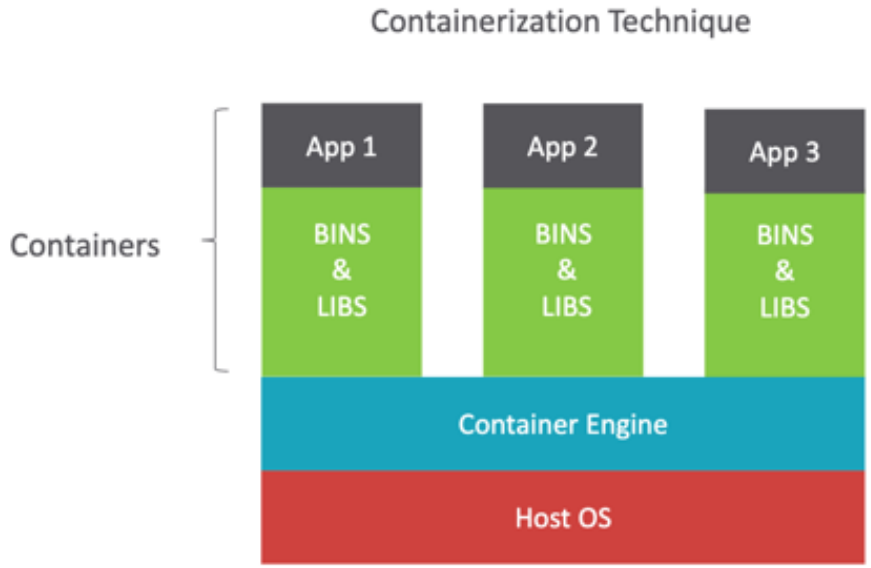
- Search for an image that starts with “hello-world” from Docker Repository
  - *docker search hello*
- Pull the selected image from Docker Hub
  - *docker pull hello-world*
- Execute the “hello-world” pulled from Docker Repository
  - *docker run hello-world*

Comparison between Docker and physical containers

	Physical Containers	Docker Container
Content Agnostic	• The same container can hold almost any type of cargo	• Can encapsulate any payload and its dependencies
Hardware Agnostic	• Standard shape and interface allow same container to move from ship to train to semi-truck to warehouse to crane without being modified or opened	• Using operating system primitives (e.g. LXC) can run consistently on virtually any hardware—VMs, bare metal, openstack, public IAAS, etc.—without modification
Content Isolation and Interaction	• No worry about anvils crushing bananas. Containers can be stacked and shipped together	• Resource, network, and content isolation. Avoids dependency
Automation	• Standard interfaces make it easy to automate loading, unloading, moving, etc.	• Standard operations to run, start, stop, commit, search, etc. Perfect for devops: CI, CD, autoscaling, hybrid clouds
Highly efficient	• No opening or modification, quick to move between waypoints	• Lightweight, virtually no start-up penalty, quick to move and manipulate
Separation of duties	• Shipper worries about inside of box, carrier worries about outside of box	• Developer worries about code. Ops worries about infrastructure.

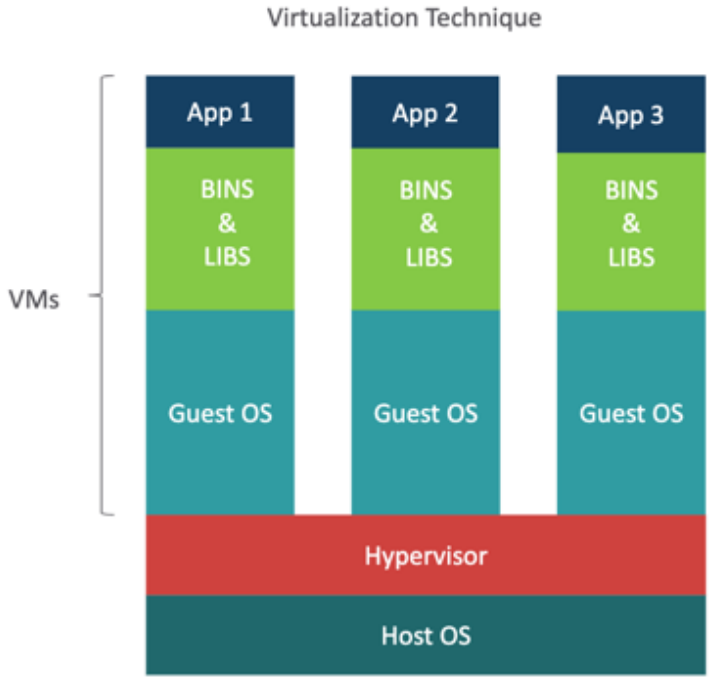
Containers advantages over VMs

- Containers are lighter and smaller
- Better resource utilization compared to VMs
- Short boot-up process
- Containerization is just virtualization at the OS level



Virtualization

- Advantages:
  - Multiple OS in the same machine
  - Easy maintenance and recovery
  - Lower total cost of ownership compared to real machines
- Disadvantages:
  - Multiple VMs lead to unstable performance
  - Hypervisors are not as efficient as host OS
  - Long boot-up process



Connection modes

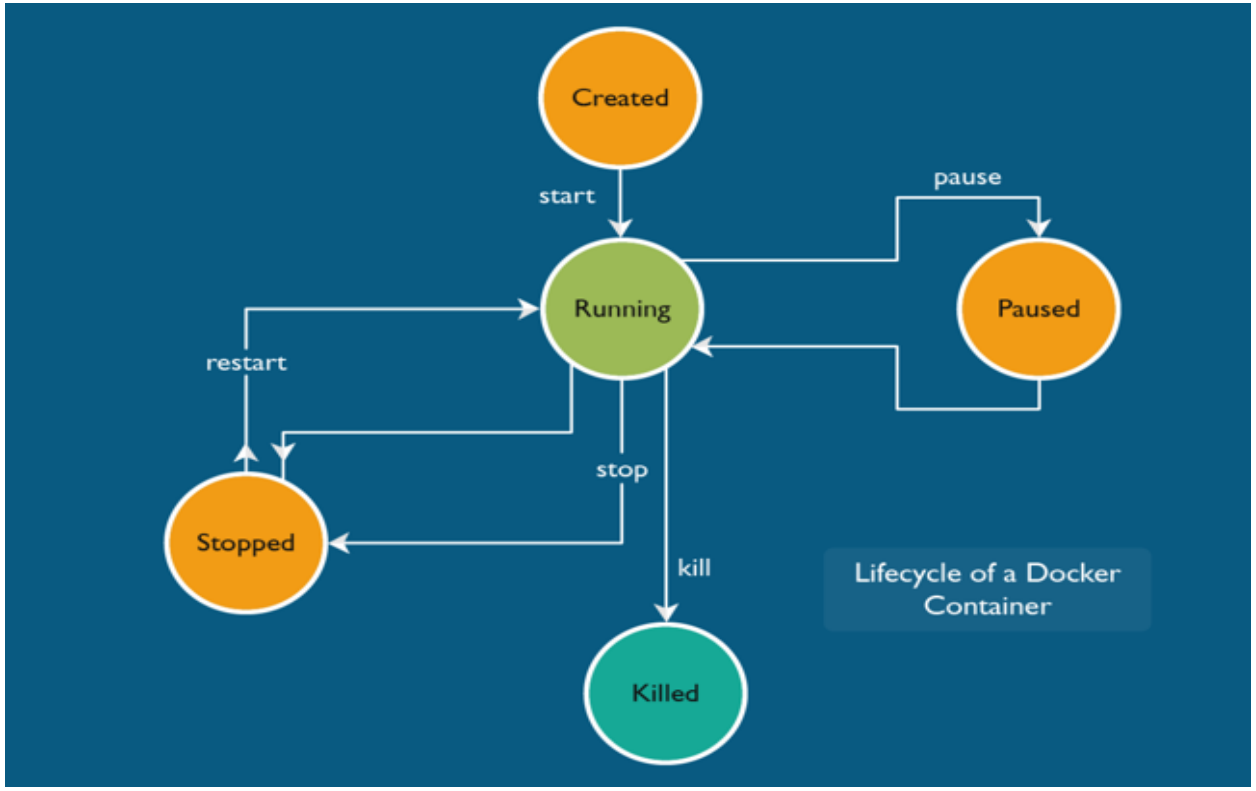
- Detached mode:
  - *docker run –d ubuntu*
  - d → detached mode
- Root User mode:
  - *docker run –it ubuntu*
  - i → interactive
  - t -→ connected to terminal

- *docker attach <container>*

Examples

- *docker pull ubuntu* → This pulls the ubuntu image from docker hub repository with the tag: latest
- *docker run –i ubuntu* → This command helps you to get inside the container
- *docker ps*
- *docker run –t ubuntu* → This command calls a terminal from inside the container, this prevents the container from exiting
- *docker run –it ubuntu* → This allows the container to run in the interactive mode as well as prevents it from exiting (crtl + P+ Q to get out of container without exiting)
- *docker run –d ubuntu* → This allows the container to run a service in the background

Lifecycle of a container



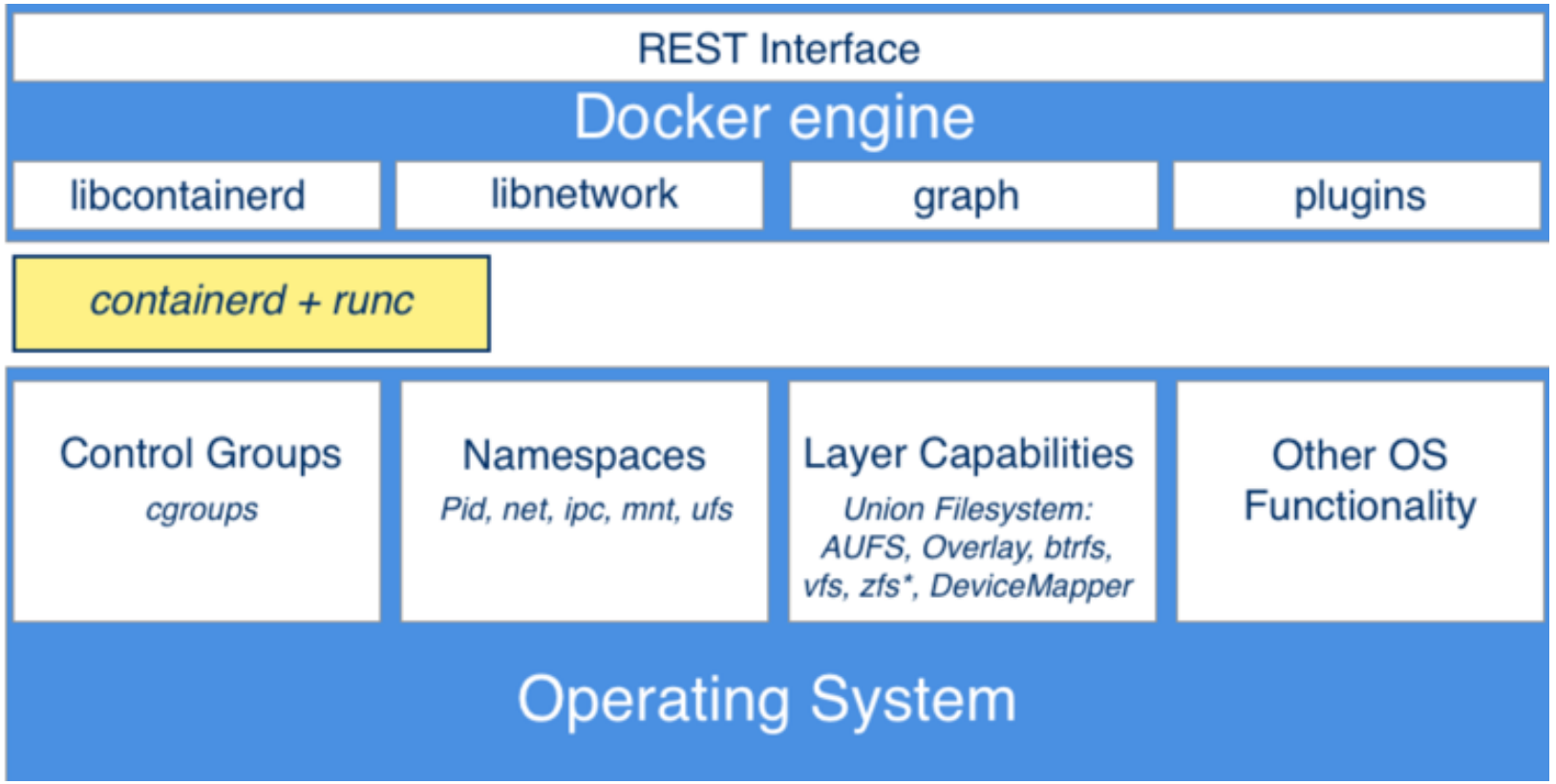
Examples

- *docker run hello-world:latest*
- *docker ps*
- *docker ps – a*
- *docker images*
- *docker run –d –name="container1" hello-world:latest*
- *docker run –itd ubuntu*
- *docker attach <containerid>*
- *docker stop <containerid>*
- *docker start <containerid>*
- *docker inspect <containerid>*
- *docker version*
- *docker container run centos ping -c 5 127.0.0.1*
- *docker exec –it <container> /bin/sh*
- *docker logs <container>*

Anatomy of Containers

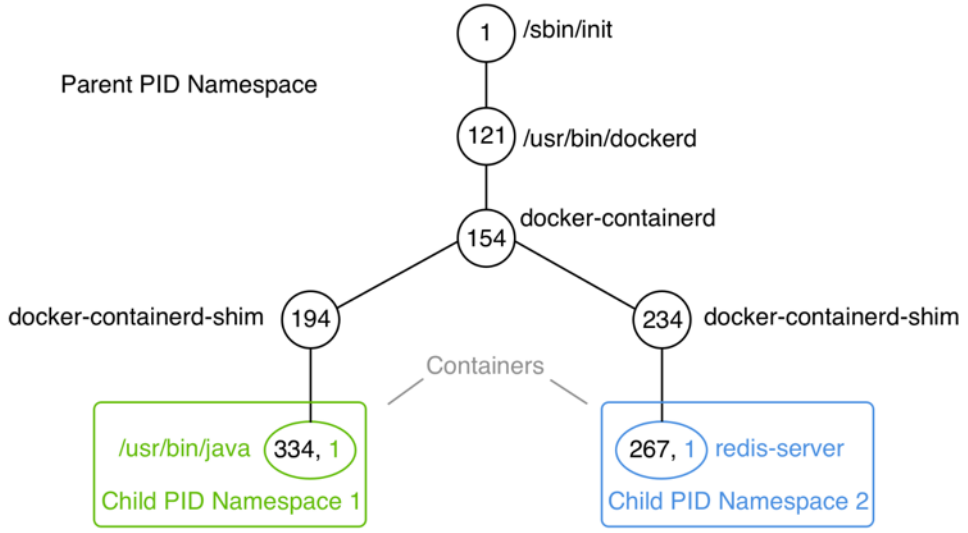
- Containers leverage a lot of features and primitives available in the Linux OS. The most important ones are *namespaces* and *cgroups*. All processes running in containers share the same Linux kernel of the underlying host operating system. This is fundamentally different compared with VMs, as each VM contains its own full-blown operating system.

Architecture of Docker



Namespaces

The PID namespace is what keeps processes in one container from seeing or interacting with processes in another container. A process might have the apparent PID **1** inside a container, but if we examine it from the host system, it would have an ordinary PID, say **334**:



Control groups (cgroups)

- Linux cgroups are used to limit, manage, and isolate resource usage of collections of processes running on a system. Resources are CPU time, system memory, network bandwidth, or combinations of these resources, and so on.

Union filesystem (UnionFS)

- The UnionFS forms the backbone of what is known as container images.
- UnionFS is mainly used on Linux and allows files and directories of distinct filesystems to be overlaid and with it form a single coherent file system.

- In this context, the individual filesystems are called branches.
- Contents of directories that have the same path within the merged branches will be seen together in a single merged directory, within the new, virtual filesystem. When merging branches, the priority between the branches is specified. In that way, when two branches contain the same file, the one with the higher priority is seen in the final FS

Runc

- Runc is a lightweight, portable container runtime. It provides full support for Linux namespaces as well as native support for all security features available on Linux, such as SELinux, AppArmor, seccomp, and cgroups.
- Runc is a tool for spawning and running containers according to the **Open Container Initiative** (**OCI**) specification. It is a formally specified configuration format, governed by the **Open Container Project** (**OCP**) under the auspices of the Linux Foundation.

Containerd

- Runc is a low-level implementation of a container runtime; containerd builds on top of it, and adds higher-level features, such as image transfer and storage, container execution, and supervision, as well as network and storage attachments. With this, it manages the complete life cycle of containers. Containerd is the reference implementation of the OCI specifications and is by far the most popular and widely-used container runtime.

Sharing docker host data with containers

- User -v option of docker run to mount a host volume into a container
- Sharing a working directory from the host in a a certain directory in a container
- Mount a working directory from the host into a certain directory in a container
- Creating files or directories within the container, let the changes be written directly to the host working directory
  - *docker run –it –v ~/dironhost:/dirinsidecontainer ubuntu*

Copying data to and from containers

- Use docker cp command to copy a file form a working container to the docker host:
  - *docker cp <container>:/file1.txt <destination path>*
  - *docker cp file1.txt container:/file1.txt*

Expose Ports

- Use –p to map ports between the container and the container host
  - *docker run -p 80:8080 tomcat*

Docker on Windows and Mac

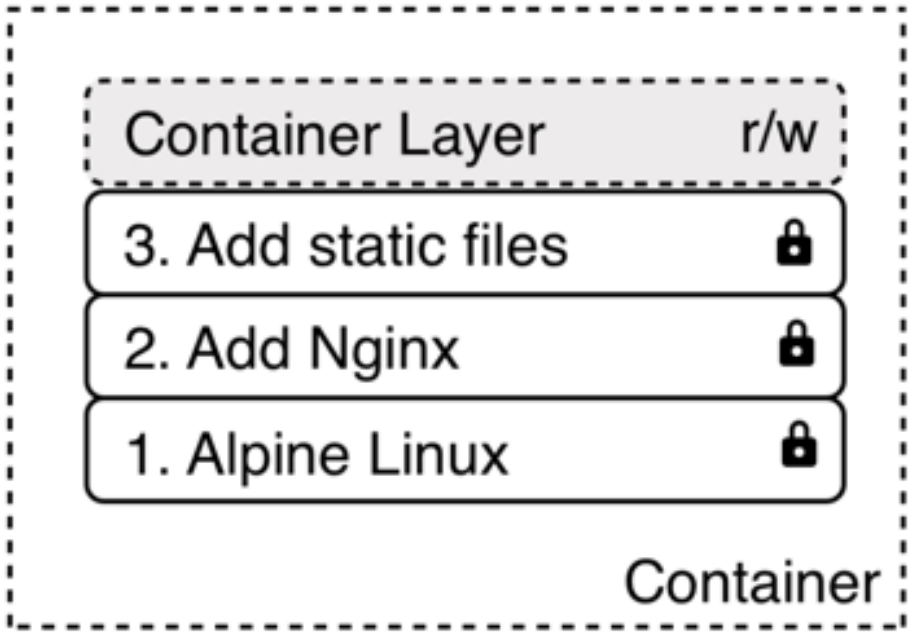
- Docker toolbox has been available for developers for few years
- It precedes the newer tools such as Docker for Mac and Docker for Windows
- The toolbox allows a user to work very elegantly with containers on any Mac or Windows computer. Containers must run on a Linux host. Neither Windows or Mac can run containers natively. Thus, we need to run a Linux VM on our laptop, where we can then run our containers. Docker Toolbox installs VirtualBox on our laptop, which is used to run the Linux VMs we need
- you can only install Docker for Windows on Windows 10 Professional or Windows Server 2016 since it requires Hyper-V, which is not available on older Windows versions or on the Home edition of Windows 10. If you are using Windows 10 Home or an older version of Windows, you will need to stick with Docker Toolbox

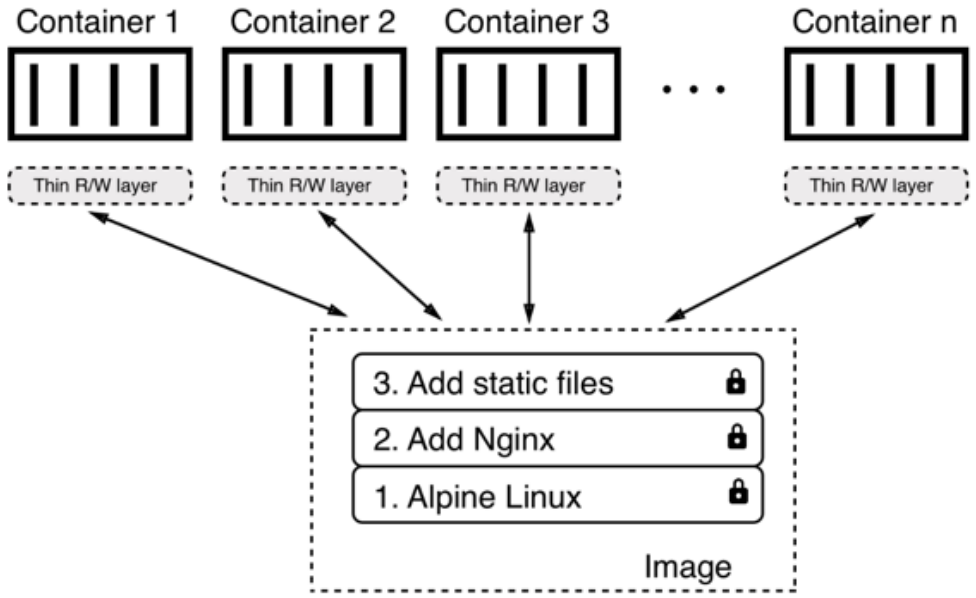
Base image

- Docker base image is the basic image on which you add layers and create a final image containing your app
- It keeps track of the difference between the base image and the new image by creating a new image layer using the union file system
- Images are comprised of multiple layers
- Every image contains a base layer
- Docker uses a copy on write system
- Layers are just read only images

Layered filesystem

- A container image is made of a stack of immutable or read-only layers.
- When the Docker engine creates a container from such an image, it adds a writable container layer on top of this stack of immutable layers.





Creating an image

- The first one is by interactively building a container that contains all the additions and changes one desires and then committing those changes into a new image.
  - *docker commit <container> myubuntu:v1.0*
  - *docker image history myubuntu:v1.0*
- The second and most important way is to use a Dockerfile to describe what's in the new image and then build this image using that Dockerfile as a manifest.
- Finally, the third way of creating an image is by importing it into the system from a tarball.
  - docker image save -o ./myubuntu.tar myubuntu:v1.0
  - docker image load –i ./ myubuntu.tar

Docker File

- Docker file is the basic building block of docker containers
- Docker file is a file with a set of instructions and forms the basis of any docker image
- Everytime, base image is going to be based upon another image. You are going to pick up a base image and build on that image

Various commands related to docker file

- *Env:* is used to set one or more environment variables
- *Workdir:* It defines the location where the command defined by *CMD* is to be executed
- *Run:* It's used to take the commands as it's arguments and runs it to form an image
- *Volume:* It's used to enable access from your container to a directory
- *Add:* It copies the file into the containers own file system from the source on the host at the stated destination
- *Expose:* it's used to expose the port to allow networking between the running process inside the container

FROM

- Every docker file starts with this command
- It shows where the base image coming from
- Will pick up an image from docker hub or some other repository and make some changes
- Example:

  FROM ubuntu:latest

MAINTAINER

- It shows the maintainer or the owner of the docker file
- It requires certain format – It requires the name and the email id
- Format: MAINTAINER name <email>
- Example:

  FROM ubuntu:latest

  MAINTAINER yourname youremail

Example

From ubuntu:lastest

MAINTAINER yourname youremail

RUN apt-get update

RUN apt-get -y install vim

ENV

- Environment variables in docker are declared with 'ENV' statement
- Environment variables are notated in dockerfile as $variable_name or ${variable_name}
- Set up the environment variable and pass a variable that we need to pass inside the container that runs on base image, example: *ENV MYVALUE xyz*
- When you run the container, this value has to be passed using *echo $MYVALUE*

Docker File – Continued

- EXPOSE: good for inter-container communications - Ports are set up in the docker file to be exposed
- *EXPOSE 80*
- *CMD: command for starting up of a service of some kind*
- *Example:*

  From ubuntu:lastest

  MAINTAINER yourname youremail

  RUN apt-get update

  ENV MYVALUE xyx

  EXPOSE 80

  CMD [“bin/bash”]

Building a docker image

- *vi Dockerfile*
- *docker build –t myubuntu:v1.0 .*
- *docker images*

Docker RUN vs Runtime (CMD & Entrypoint) Command

- RUN executes commands in a new layer and creates a new image. It's often used for installing software packages
- CMD sets default command and/or parameters, which can be overwritten from command line when docker container runs
- ENTRYPOINT configures a container that will run as an executable

Example

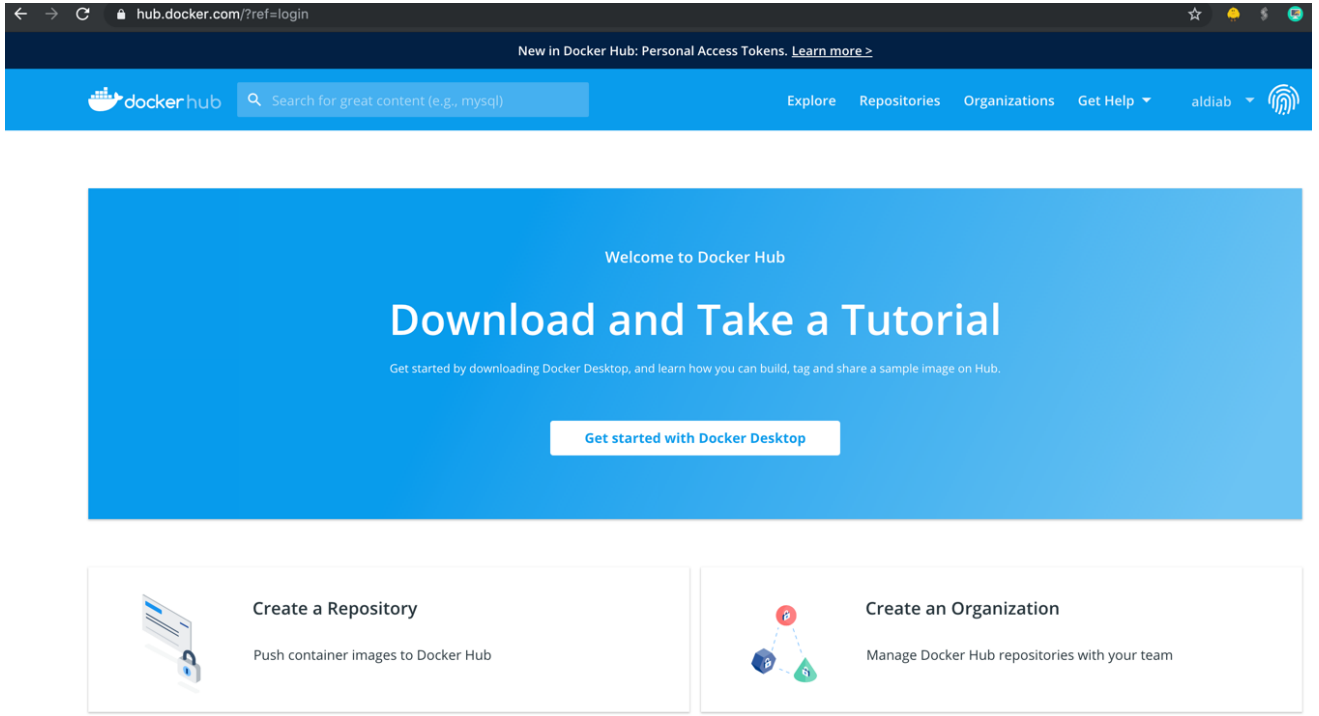
ENTRYPOINT ["/bin/echo", "Hello"]

CMD ["world"]

- *docker run –it <image>*
  - will produce *"Hello world"*
- *docker run –it <image> John*
  - will produce *"Hello John"*

Docker Hub

- It's a cloud based registry service which allows you to link to code repositories, build images and test them, stores manually pushed images and links to docker cloud, so you can deploy images to your hosts.
- Publish/share an image on docker hub using the following steps:
  - Create an account on the docker hub
  - Login into the hub from docker host
  - Push your images



Docker Hub

- *docker login* → enter username and password
- docker push yourrepo/image
- docker tag hello-world yourrepo/hello-world
- docker pull yourrepo/hello-world

Docker registry

- Deploy private registry:
  - docker run –d –p 5000:5000 –name registry registry:2
  - docker image tag my-image localhost:5000/my-image
  - docker push localhost:5000/my-image

Docker compose

- It is a tool for defining & running multi-container docker applications
- Use yaml files to configure application services (docker-compose.yml)
- Can start all services with a single command (*docker compose up*)
- Can stop all services with a single command (*docker compose down*)
- Can scale up selected services when required
- *docker-compose –v*

Docker compose

```
version: '3.3'

services:
   web:
     image: nginx
     ports:
       - 80:80

   database:
     image: redis
```

- *docker-compose config*
- *docker-compose up –d*
- *docker-compose down*
- *docker-compose –f docker-compose2.yml up*

Docker compose

```
version: '3.3'

services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: somewordpress
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD: wordpress

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     restart: always
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: wordpress
       WORDPRESS_DB_PASSWORD: wordpress
       WORDPRESS_DB_NAME: wordpress
volumes:
    db_data: {}
```

KITEMATIC

- Instead of using the command line to manage your containers locally, you can use Kitematic UI which is a graphical interface.

Docker Networking

- Docker0 Bridge:
  - Default bridge created by Docker to provide communication across Docker containers and external world including the host
- Host Network:
  - Host Network driver is used when isolation of container network stack from the docker host is not required
  - If a container is running on some port and the Host Network is being used, then application will be available on the same port on host's IP address
  - It works only on Linux OS and not on Windows or Mac OS

Docker Networking

- Overlay:
  - Creates a distributed network and helps multiple docker daemons in communication
  - Allows secure communication
  - Helps docker swarm services to communicate
  - Below is the command to create overlay network in Docker
    - *docker network create –d overlay mynetwork*

Docker Networking

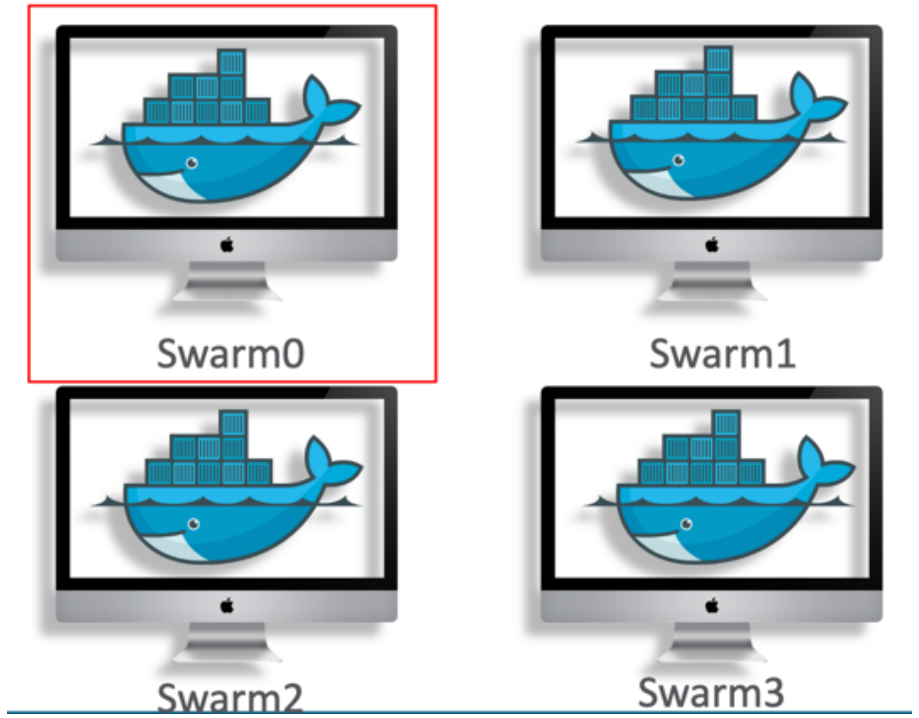
- Macvlan:
  - Macvlan assign a Mac Address to a container which helps it to appear as a physical device on the network
  - This type of network is used by legacy applications, or applications which are supposed to be directly connected to physical network
  - Macvlan networks can be isolated by using various network interfaces
- None:
  - It helps to disable all the networking stack on a container
  - It is not available for swarm services
  - *network –none* flag is used while starting the container to completely disable the networking stack

Docker Swarm

- Docker Swarm is a cluster of machines, all running docker which provides scalable and reliable platform to run many containers.
- With Swarm, IT admins and developers can establish and manage a cluster or Docker nodes as a single virtual system.

Docker Swarm

- Every Swarm has at least one manager (Generally, the one which is initialized first)
- Port 2377 is the default port
- Managers:
  - Swarm0
- Workers:
  - Swarm1
  - Swarm2
  - Swarm3.
- *docker swarm init --advertise-addr 10.0.0.1*
- *docker node ls*



TE5

- Write a docker file that builds an image to run your java classes in a container. You can pass the name of the class when starting the container. Build the image and push it on your docker hub account, then run it from your local machine.
- Your Java classes should be hosted on your github account which you can reach from your docker image.
- Try to use docker volumes, exposing the ports, docker compose in any simple useful scenario to show good understanding in using docker to run your Java applications
- Capture that in 10- mins video and share the link when submitting your assignment

docker