Docker

By
Venkat
Rise ‘n‘ Shine Technologies

Agenda

- What is Docker?
  - Docker vs. Virtual Machine
  - History, Status, Run Platforms
  - Hello World
- Images and Containers
- Volume Mounting, Port Publishing, Linking
- Around Docker, Docker Use Cases

Traditional application deployment

- Applications were deployed directly on physical hardware, over the host OS.

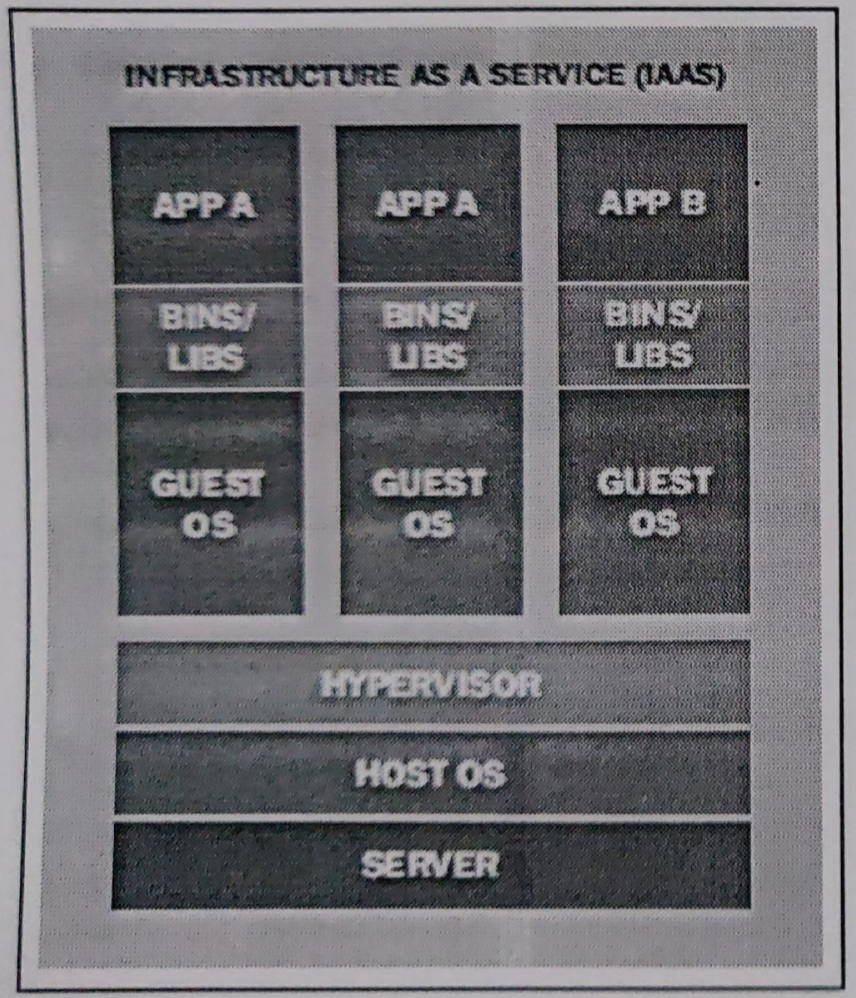
- Single user space, runtime was shared between applications

Disadvantages:

- Hardware resources were regularly underutilized.

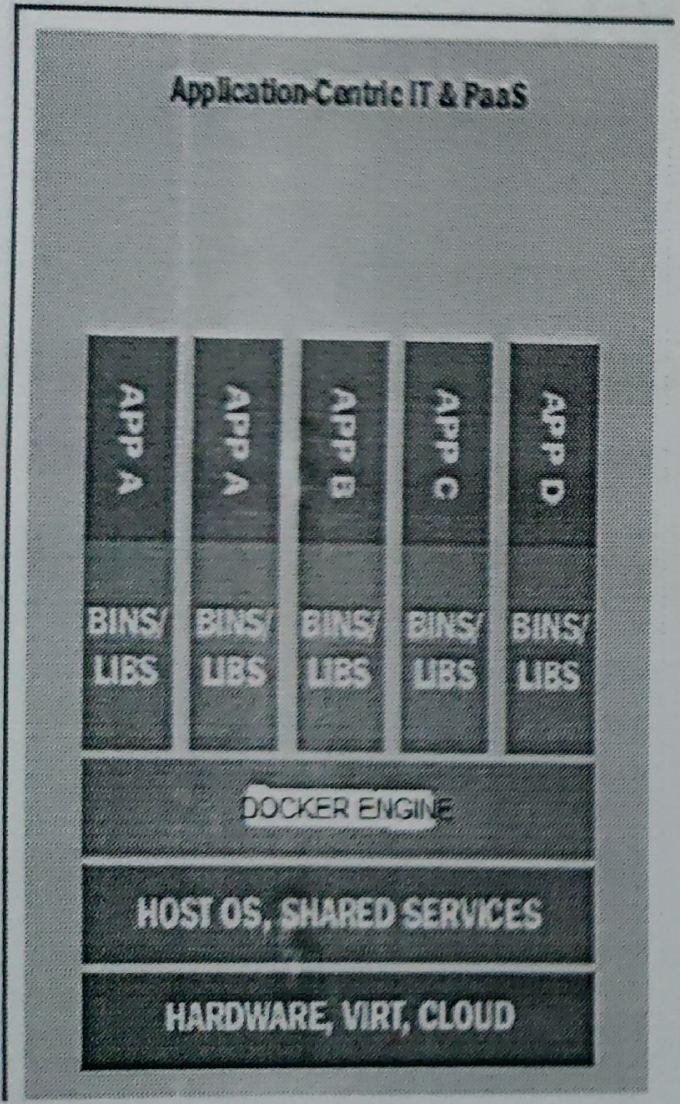
- It was mostly managed by an IT department and gave a lot less flexibility to developers.

App deployment in a virtualized env



- To overcome the limitations set by traditional deployment, virtualization was invented.
- With hypervisors such as KVM, XEN, ESX, Hyper-V, and so on, we use the hardware for virtual
- Machines (VMs) and deployed a guest OS on each virtual machine.

Application Centric IT and Pass



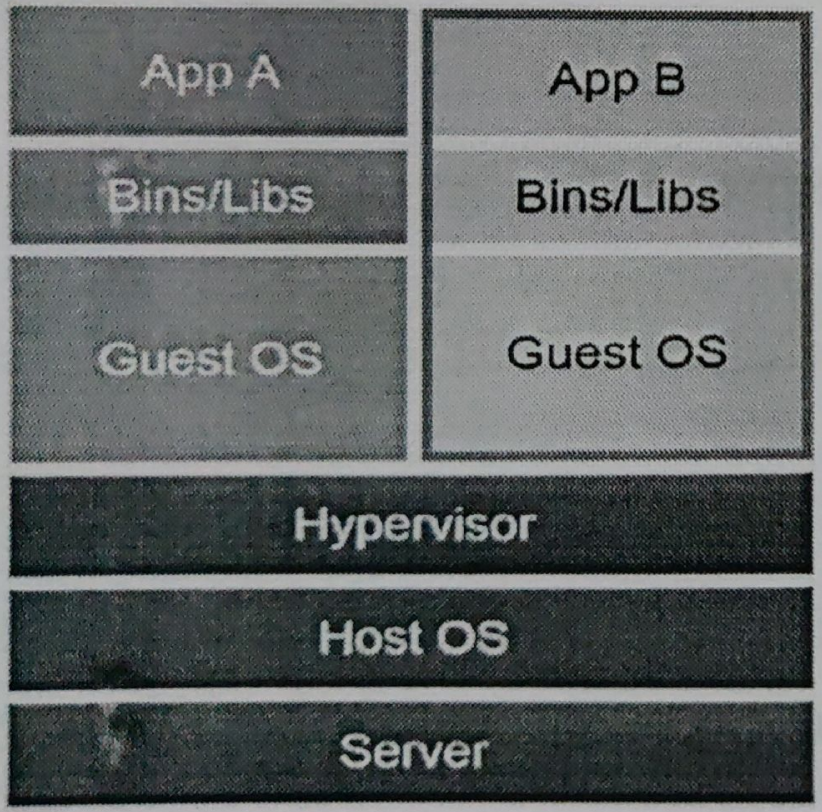
After virtualization, we are now moving towards more application-centric IT.

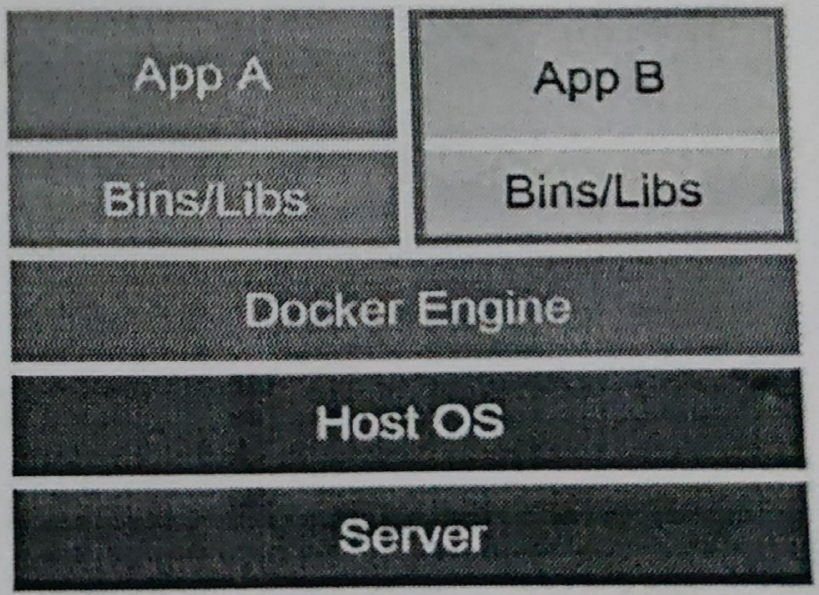
We have removed the hypervisor layer to reduce hardware emulation and complexity.

The applications are packaged with their runtime environment and are deployed using containers.

OpenVZ, Solaris Zones, and LXC are a few examples of container technology.

Docker vs. Virtual Machine



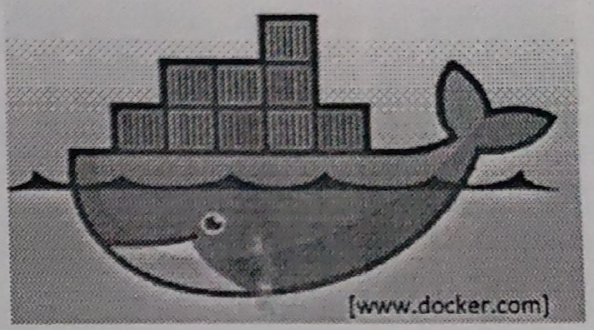


What is Docker?

- Docker is an open-source project

- It automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating system–level virtualization on Linux.

Docker: Name

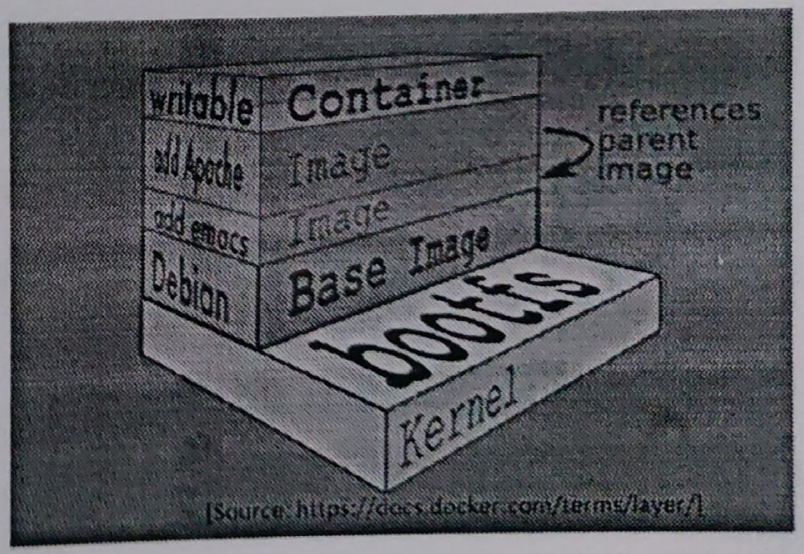


- Provide an uniformed wrapper around a software package:
  «Build, Ship and Run Any App, Anywhere»
  [www.docker.com]

  – Similar to shipping containers: The container is always the same, regardless of the contents and thus fits on all trucks, cranes, ships, ...

Docker Technology

- libvirt: Platform Virtualization
- LXC (LinuX Containers): Multiple isolated Linux systems (containers) on a single host
- Layered File System



Docker Use Cases

- Development Environment
- Environments for Integration Tests
- Microservices
- Unified execution environment (dev → test → prod (local, VM, cloud, ...)

Run Platforms

- Various Linux distributions (Ubuntu, Fedora, RHEL, Centos, openSUSE, ...)
- Cloud (Amazon EC2, Google Compute Engine, Rackspace)
- 2014-10: Microsoft announces plans to integrate Docker with next release of Windows Server

Requirements to install Docker

- 1. Docker is not supported on 32-bit architecture. To check the architecture :
  - **$ uname -i**
  - **x86_64**

- 2. Docker is supported on kernel 3.8 or later. It has been back ported on some of the kernel 2.6, such as RHEL 6.5 and above. To check the kernel version:
- **$ uname -r**

Installing Docker

- **--> To find the list of all the Docker packages**
- > sudo yum search docker
- **--> create a repository to pull docker package**
- > cd /etc/yum.repos.d/
- > sudo vi docker.repo

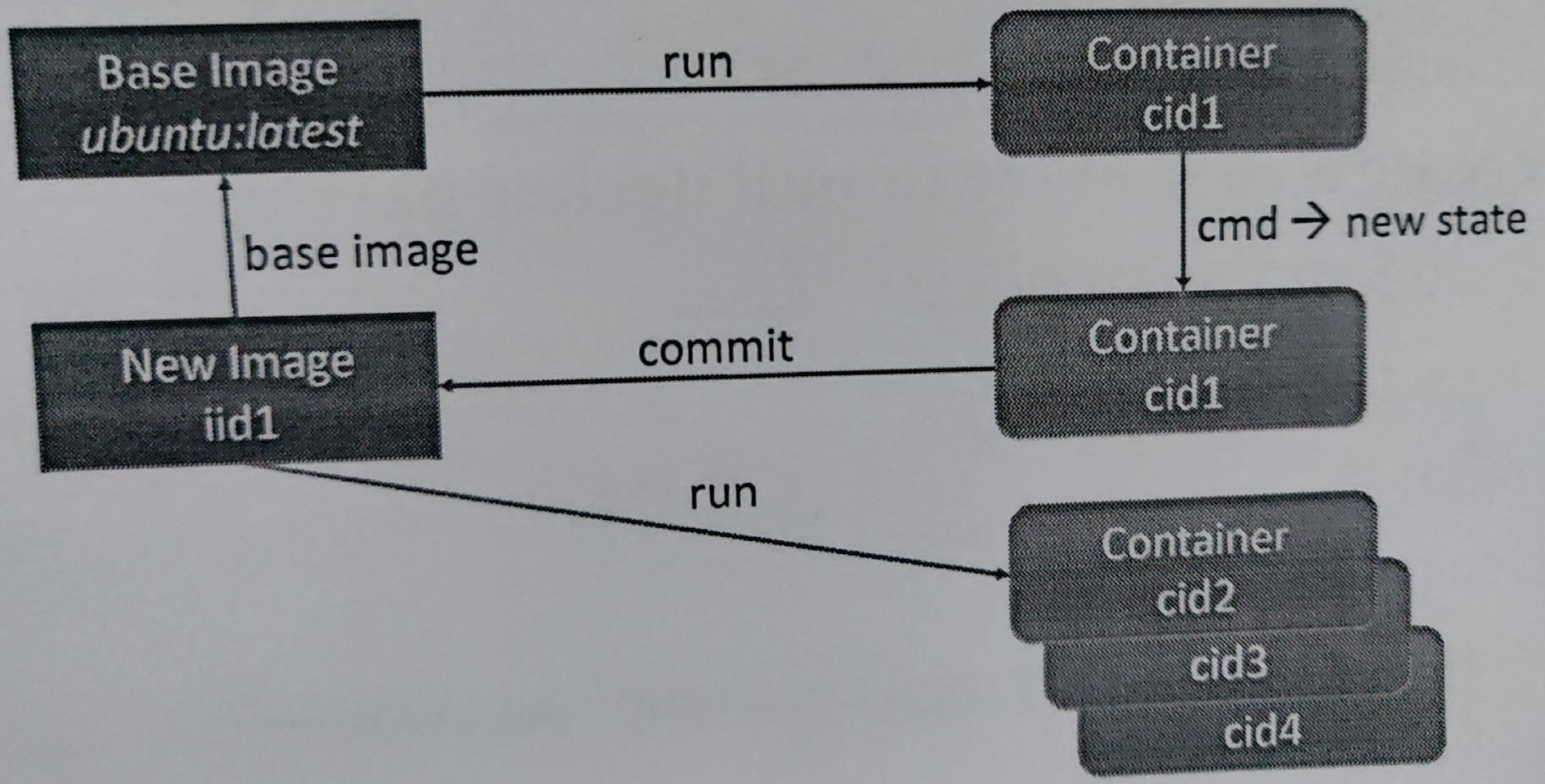
```
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
```

Installing Docker

- Install Docker using yum:
  - **$ yum update**
  - **$ yum -y install docker-engine**
- To start the service: **$ systemctl start docker**
- To enable the service start at boot time: **$ systemctl enable docker**
- To stop the service: **$ systemctl stop docker**
- To verify the installation: **$ docker info**
- To check the version **$ docker version**

- **> Exception :** Is the docker daemon process running on this Host?
- Then check the user permission for the file that is /var/run/docker.sock and then add the user to the docker group.
- > sudo usermod username –G docker

Image vs. Container



Docker Images

- Docker has a number of base images on Docker Hub
  - Including many versions of Linux distributions
  - The image centos:latest will be used as a base

- The Docker search facility can be used to search for images on Docker Hub
  - docker search centos

Pulling and Running Containers

- All the images are available in dockerhub which we can download the images and run it.

- A Docker Image must be located on the local computer
  - It may have been created locally
  - It may have been pulled from a Registry

- The `pull` command insures that the specified image is on the local computer

- The `run` command creates and initiates a container based on the image

Pulling and Running Containers

- The command to list all the images

  $ docker images

- *The example runs the latest hello-world image*

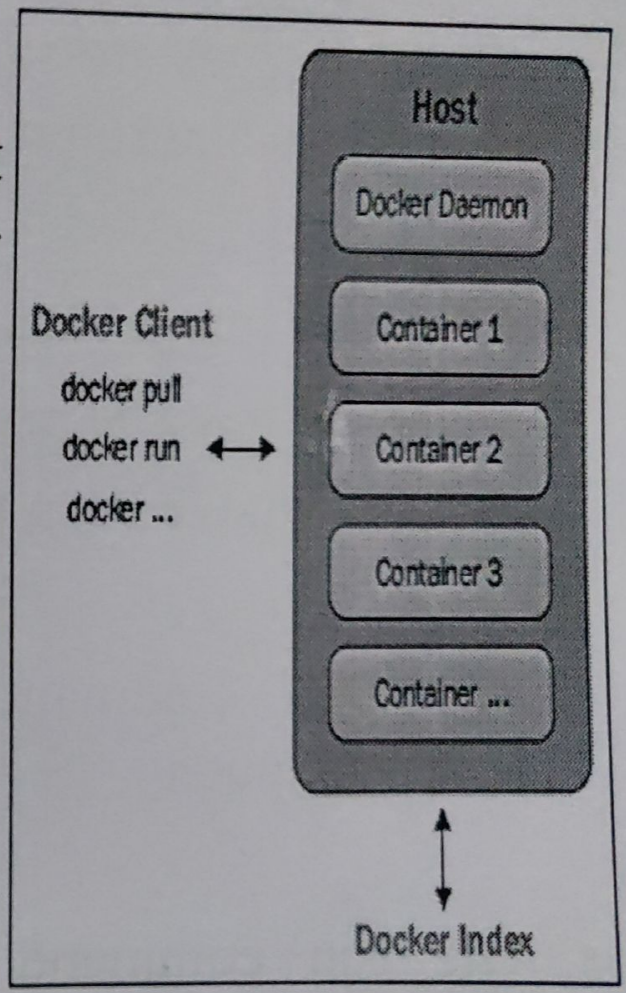
  $ docker run hello-world

- The storage location of all the Docker images and containers is /var/lib/docker

  $ cd /var/lib/docker

How the Docker works?

- Docker has client-server architecture.
- Its binary consists of the Docker client and server daemon, and it can reside ir the same host.



- The client can communicate via sockets or the RESTful API to either a local or remote Docker daemon.

- The Docker daemon builds, runs, and distributes containers.

Pulling and Running Containers

- Pull the 'ubuntu' image using tag

  $ docker pull ubuntu:xenial (16.0 version of ubuntu)
- Then run the docker image as a container with interactively

  $ docker run -i -t ubuntu:xenial /bin/bash
- Then run the cmd inside the container ‘ps aux’and then 'exit‘ it.

- Note: If you exit the container, then container also stopped.
- *$ docker ps* (No conatiers)

- We can start the stopped container by using 'restart' command.
- *$ docker restart 'container_id/container_name'*

Verifying & Accessing the Container

- The images can be inspected to see the layers
- The inspect command returns a JSON array by default
- It can be used on images and containers
- The –f or –format can be used to extract the JSON fields

$ docker inspect 'ubuntu:xenial'
$ docker inspect -f {{.Architecture}} 'ubuntu:xenial'
$ docker inspect 'container_id' | grep IP
$ docker ps (to list all the Running containers)

- If we would like to access the running container terminal...
  > docker attach 'container_name/container_id'

Naming Our Containers

- Running the containers with name

  $ docker run --it --name container_name ubuntu

- Renaming the existing containers
- $ docker rename old_container_name new_container_name

  – docker run –it —name client ubuntu:xenial /bin/bash
  – docker stop client
  – docker rm client

Listing and Removing Containers

- $ docker ps -a
- $ docker ps -a -q (which displays container Ids only)
- $ docker ps -a -q | wc –l

- $ docker rm container id /name (which removes the container)
- $ docker rm container_name container_id (Multiple containers)

- $ docker rm -f container_name or id (It removes the container forcibly the running containers)
- $ docker rm `docker ps -a -q` (It removers all the containers)

Listing and Removing Base Images

-- Images can be listed using the images command

$ docker images

-- Removing the Image

$ docker rmi image_name or id

-- Removing the Image forcibly

$ docker rmi -f image_name or id (Forcibly to remove)

Running container as Daemon Process

- --> Run the container as a Daemon (Back ground) process

  $ docker run -itd ubuntu:xenial /bin/bash

- To view the list of the all Containers (Running & Stopped)

  $ docker ps –a

- Run one more container as a Daemon (Back ground) process

  $ docker run -itd ubuntu:xenial /bin/bash

  $ docker ps

  $ docker inspect 'container_name/id'

  $ docker inspect 'container_id' | grep IP

  $ docker attach 'container_id'

Inspect Container Processes

- Instead of running the ps directly within the container, then use the following command to get the top command output of the container

  $ docker top container_name

- Use the following command to track the docker performance

  $ docker stats container_name

  (in another window log in docker conatiner and run apt-get install apache2)

Docker Events

- Exploring how to get an overall idea of how docker containers are performing on our system by running the docker events command

  $ docker events

- If you would like to see the list of all the events which are happened last 1 hour

  $ docker events --since '1h'

- **Docker events filter:**

  $ docker events --filter event=attach (if anyone is performed attach event, it triggers that event)

- we can pass multiple events

  $ docker events --filter event=attach --filter event=stop

Scenario: Docker Events

- To track all the events, first run the docker events command and in parallel window try to perform some activities in docker container.

  $ docker events (in one window)

  $ docker exec --it container_name /bin/bash (in another window)

  $ exit

  $ docker attach container_name

  $ docker start container_name

  $ docker kill container_name

Note: whenever we exit the container which started using exec cmd, then that bash shell is exited. But if we use 'attach' cmd and then if we exit the bash then the container is stopped)

Port Exposure on Containers

→ By default the ports are not exposed to Host port whenever we run the nginx server (It shows ports but not mapped to the Host Machine)

```
$ docker run -itd nginx:latest
$ docker ps
```

→ Mapping container port to the Host OS with Random port

```
$  docker run -itd -p 80 nginx:latest
$ elinks http://localhost:32768
```

→ Mapping Container port to the Host OS with custom port

```
$  docker run -itd -p 8080:80 nginx:latest
$ elinks http://localhost:8080
```

Port Exposure on Containers

→map all the container ports to the Host

$ docker run -P nginx:latest

$ verify with elinks

→ map Container port to the Host with N/w interface

$ docker run -p 127.0.0.1:8080:80 nginx:latest

$ elinks http://127.0.0.1:8080 (only on this n/w interface nginx url is available)

Network: List and Inspect

- If you run 'ifconfig' on the Host OS, we can find the 'docker' n/w interface which is binded with HOST n/w interface (eth0) using bridge n/w.

- Run the following command to list all the n/ws which are available in Host OS.

  $ docker network ls

  $ docker network ls --no-trunc (This command displays complete ID of the n/w in the list)

- To view the complete information of the n/w Driver (bridge)

  $ docker network inspect bridge (driver_name)

Network: Create and Remove

- How to access the manual page of the docker commands

  $ man docker-network-create

- Create a network

  $ docker network create --subnet 10.1.0.0/24 --gateway 10.1.0.1 mybridge01

- Verify n/w driver has been created or not

  $ docker network ls

  $ ifconfig (new interface is available in ifconfig)

- How to remove a network driver

  $ docker network rm n/w_name_or_id

- Note: While removing the network drivers, we shouldn't remove the default drivers(bridge, host, null). If we remove the defaults one by mistake, we need to uninstall the docker and install it.

Network: Assign to Containers

- Create a network

  $ docker network create --subnet 10.1.0.0/16 --gateway 10.1.0.1 --ip-range=10.1.4.0/24 --driver=bridge --label=host4network bridge04

  $ docker network ls

  $ ifconfig

- $ docker network inspect bridge04
- Launch container with the newly create driver
- $ docker run -it --name nettest1 --net bridge04 centos:latest /bin/bash
- within the container
- $ yum update -y
- $ yum install net-tools -y

Network: Assign to Containers

- Check that IP addresses of the docker containers are assigned as per the bridge driver

  $ ifconfig

- To check the default gateway

  $ netstat -rn

- Check the resolve conf file

  $ cat /etc/resolv.conf

  $ exit

- Assign the static IP address to the Docker Container
- $ docker run --it --name netest2 --net bridge04 --ip 10.1.0.100 centos:latest /bin/bash

Scenario – Working with Multiple Images

- If you do any changes within container, then those changes are available within that container. If you create a new container instance, then these are not available.

$ docker run -it ubuntu:xenial /bin/bash

- Create a some file in the user home dir

$ cd /root

$ vi test.txt

- Then exit the container.

Scenario – Working with Multiple Images

- Then run the new container, but we don't find the file in container.

  $ docker run -it ubuntu:xenial /bin/bash

  $ ls /root

- Then restart the previous container and verify the file. That exists.

  $ docker restart container_id

  $ docker attach container_id

  $ ls -l /root

Container Volume Management

- Run the container by using Volume option

  $ docker run -it --name volTest1 -v /mydata centos:latest /bin/bash

- Run the following commands within the container

  $ df -h (verify that mount info)

  $ cd /mydata

  $ echo "This is test container file" > mytest.txt

- On the Host OS, go to /var/lib/docker/volumes directory

- Run the volumes by mapping Host Dir with the Container

  $ docker run -it --name volTest2 -v /Host_OS_Dir:/mydata centos:latest /bin/bash

Packaging Customized Container- commit

- First start the ubuntu container and do some changes in the container

  $ docker run -it ubuntu:xenial /bin/bash

  $ cd /root

  $ echo "This is version 1 file" > file_ver.txt

  $ apt-get update

  $ apt-get install -y telnet openssh-server

- Then verify the installation of these services

  $ which sshd

  $ which telnet

  $ exit the container

Packaging Customized Container- commit

- Now create an custom Image

  $ docker commit -m "msg" -a "author" container_id 'new image name‘

  $ docker images

  $ docker run -it new_image_name /bin/bash

Taking Control of Our Tags

- As a result of having a number of images, we may need to tag them with non-default names for testing/updating/distribution.

- Additionally, tags are an important precursor to publishing our images to public or private repositories.

```
$ docker tag image_name_or_id mine/centos:v1.0
```

Pushing to Docker Hub

- Go to docker hub and create a repository name with 'myubuntu'
- Login to docker hub

  $ docker login (enter the username and password)

- How to logout the user session

  $ docker logout

- The user information is stored in the ~/.docker/config.json file

  $ docker images

  $ docker tag imagename usernmae/reponame:versiontag

Pushing to Docker Hub

- Login to the the session

  $ docker login --username=username (then enter the password)

  $ docker images

  $ docker push username/reponame:versiontag

- Once pushed successfully, go to docker hub and verify that image is updated to repo

Automating Builds

- Docker image creation can be automated
- Create a directory containing all the files required for the build
- Add a file called Dockerfile which defines the build process. The directory becomes the build context
- Each command in the build file creates a layer of the image. A new container is created at each stage.
- Can be versioned in a version control system like Git or SVN, along with all dependencies

Dockerfile

- --> Create a Directory and then Dockerfile with the following content in that dir

FROM centos:latest

MAINTAINER rns <rns@gmail.com>

RUN yum update

RUN yum install -y telnet net-tools

$ docker build -t "image_name" .

- Run the container with new image and then verify the custom changes in the container.

Dockerfile Directives: USER and RUN

- --> Create a Directory and then Dockerfile with the following content in that dir

FROM centos:latest

MAINTAINER rns <rns@gmail.com>

USER test

$ docker build –t centos:nonroot:v1 .

$ docker run –it centos:nonroot:v1 /bin/bash

--> Once the image is ready, If you try to run the container, it throws exception because no specific user in the container with the name 'test'

→ Add the following directive with in the Dockerfile after MAINTAINER

RUN useradd –ms /bin/bash test

Dockerfile Directives: USER and RUN

- Now the container is started with user as 'user'. But this user doesn't have any permissions. Then how to login with root user?

- Start the container

- Then use the 'exec' command with -u arg as 0 for root user

  $ docker exec –it –u 0 container_name /bin/bash

RUN Order of Execution

→ *Add the following entry after USER directive in Dockerfile and build the image*

USER test

RUN echo "export 192.168.0.0/24" > /etc/exports.list

$ docker build –t centos/config:v1

→ Build will get failed because 'permission denied'

→ Change the order of the RUN command

RUN echo "export 192.168.0.0/24" > /etc/exports.list

USER test

Dockerfile Directives: ENV

- Now set the environment variables at system level by using 'ENV' directive. Modify the 'Dockerfile' with the following content

ENV TEST_ENV "Testing Environment Variable"

```
FROM centos:latest
MAINTAINER rns rns@gmail.com
RUN useradd -ms /bin/bash test
RUN yum update -y
RUN yum install -y net-tools wget
USER test
ENV TEST_ENV "Testing Environment Variable"
```

- Build the image and then check the environment variable within the container with env

Dockerfile Directives: CMD vs. RUN

- Implement the Docker file with CMD and RUN cmd as follows

RUN useradd –ms /bin/bash test

CMD "echo" "This is custom container message"

USER test

- Then build it

  $ docker build -t centos7/echo:v1 .

- Run the container and view the message

  $ docker run centos7/echo:v1

- Verify the stopped container from the list (docker ps -a) and start the container

  $ docker start container_id

- Note: It doesn't display the message. If you run it once again using 'docker run' command only the echo message will be displayed.

Dockerfile Directives: ENTRYPOINT

- Implement the Docker file

RUN useradd –ms /bin/bash test

ENTRYPOINT “echo” “This is custom message with Entrypoint”

USER test

--> Build it

$ docker build -t centos7/entry:v1 .

$ docker images

--> Run it

$ docker run -it centos7/entry:v1

Dockerfile Directives: ENTRYPOINT

- Note: The diff between CMD and ENTRYPOINT is, CMD commands can be overridden during container initialization time but where as ENTRYPOINT always takes the same command which is mentioned in the ENTRYPOINT.

```
$ docker run centos7/entry:v1
$ docker run centos7/echo:v1
$ docker run centos7/echo:v1 /bin/echo "Hello from lab"
$ docker run centos7/echo:v1 /bin/echo
$ docker run centos7/entry:v1
```

Dockerfile Directives: EXPOSE

- --> Docker file for Apache Installation and starting the Service

RUN yum update –y

RUN yum install –y httpd net-tools

RUN "echo" "This is custom index page during image creation " >>
/var/www/html/index.html

ENTRYPOINT apachectl "-DFOREGROUND"

→ Build it and Run it and check the Container

→ Get the IP address of the container and verify the Apache URL.

→ View the index.html file content

$ docker exec apacheWeb1 /bin/cat /var/www/html/index.html

--> Let's work on the Port Expose using –P option and –p 8080:80

Dockerfile Directives: EXPOSE

- --> Expose the port in the Dockerfile
- --> Add the EXPOSE Directive (before ENTRYPOINT) within the Docker file

EXPOSE 80

ENTRYPOINT apachectl "-DFOREGROUND"

- Build it

  $ docker build -t centos7/apache:v1 .

- Run it

  $ docker run -d --name apacheWeb4 -P centos7/apache:v1

  $ docker ps

Docker File Commands - Copy

- The Dockerfile COPY command copies files into the container
  - The source files or directories must be in the build context
  - The source files can contain UNIX shell wildcards ? * []
  - Destination directories must end in a / and will get created if they don't exist

  - COPY jdk*.rpm /tmp/

Docker File Commands - Add

- The Dockerfile ADD command copies files and remote file URLs into the container
  - The source files or directories must be in the build context or remote URLs
  - The source files can contain UNIX shell wildcards ? * []
  - Destination directories must end in a / and will get created if they don't exist
  - Local source files in tar or compressed tar format get unpacked

  - ADD apache-maven*.tar.gz /opt/