**Preface:**
Welcome! These notes capture my journey of learning Docker, one of the most transformative tools in modern software development and IT.

I'm **Shujah Ullah**, a passionate learner exploring emerging technologies and deeply immersed in the dynamic fields of IT, AIOps, and software development. For the past four years, I've worked as a **Digital Control Systems Technician** at GigaWatt Electric Power Plant. This role gave me firsthand experience with the power of computers and the rapid digitization of industries. Witnessing the dominance of technology in shaping the future, I decided to embrace the 4th Industrial Revolution and transition my career from control systems to the exciting domain of AIOps.

These notes are heavily inspired by the book *"Docker Deep Dive"* by Docker Captain Nigel Poulton. Some screenshots included here are taken from his book, as it serves as a foundational resource for my learning. I hope these notes serve as a helpful guide for others embarking on their own Docker learning journey!

Connect with me:
github : https://github.com/shujah-ullah
linkedin: https://www.linkedin.com/in/shujah-ullah-537222159/

# ==001 Docker Overview

What is Docker?
**Docker** is a platform that allows developers to build, package, and run applications in a standardized and portable environment called a **container**.
**Containers**
A container is lightweight, standalone, and portable software that packages an application and all its dependencies together. containerized app can run on any platform without setting up dependencies on the host OS as all code and dependencies are packaged together

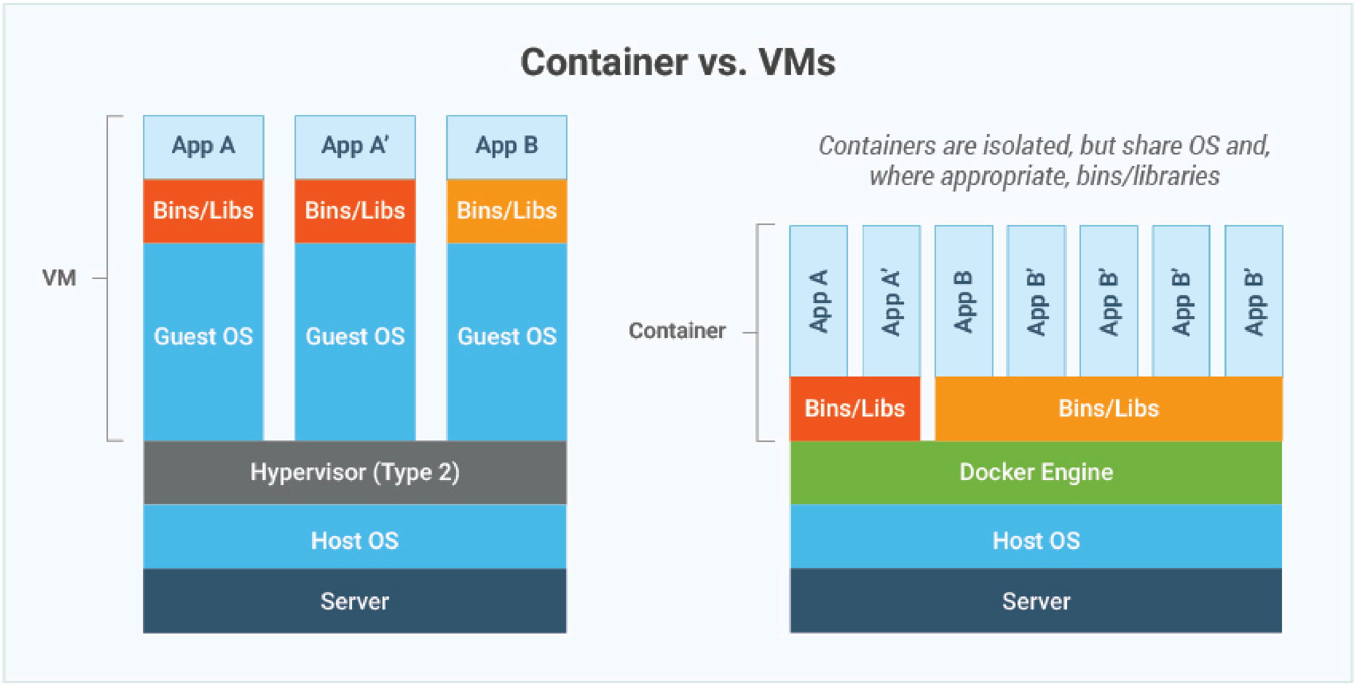
**Containers** vs **Virtual Machines**
Take example scenario, You're a developer tasked with deploying a **web application** (frontend + backend) for your team. It has the following components:

1. **Frontend**: React app.
2. **Backend**: Flask application (Python-based API).
3. **Database**: MySQL.

## Using Virtual Machines (VMs):

To deploy the app, you decide to use VMs:

1. **Setup**:
   - You create 3 separate VMs:
     - VM 1: For the React frontend.
     - VM 2: For the Flask backend.
     - VM 3: For MySQL.
   - Each VM has its own OS (e.g., Ubuntu), requiring resources like CPU, RAM, and storage.
2. **Configuration**:
   - Each VM needs manual setup: Install OS, configure the environment (e.g., Python, Node.js, MySQL).
   - Networking between VMs must be configured (e.g., opening ports, IP settings).
3. **Resource Usage**:
   - Each VM includes an entire OS, consuming significant system resources (RAM, CPU, and disk space).
4. **Startup Time**:
   - VMs take minutes to boot up and run the application.
5. **Portability**:
   - If you need to move the setup to another server, you'll need to recreate the entire VM environment or use complex VM migration tools.



Above diagram clearly illustrates differences between VMs and containers. in the VM hypervisor is exist on the top of Host operating system, each time we create a new VM we have to install a new operating system on top of our guest OS. each OS consumes CPU and memory resources separately which is not efficient usage of resources.

To solve this problems **containers** came into picture, a **container** is light weight, standalone, and portable software unit that packages an applications and all its dependencies together so you don't need to rely on what's installed on the host..
All the containerized applications share Host OS resources. Containers are an evolution of virtualization that provides process-level isolation rather than full system emulation like virtual machines.

## Using Containers:

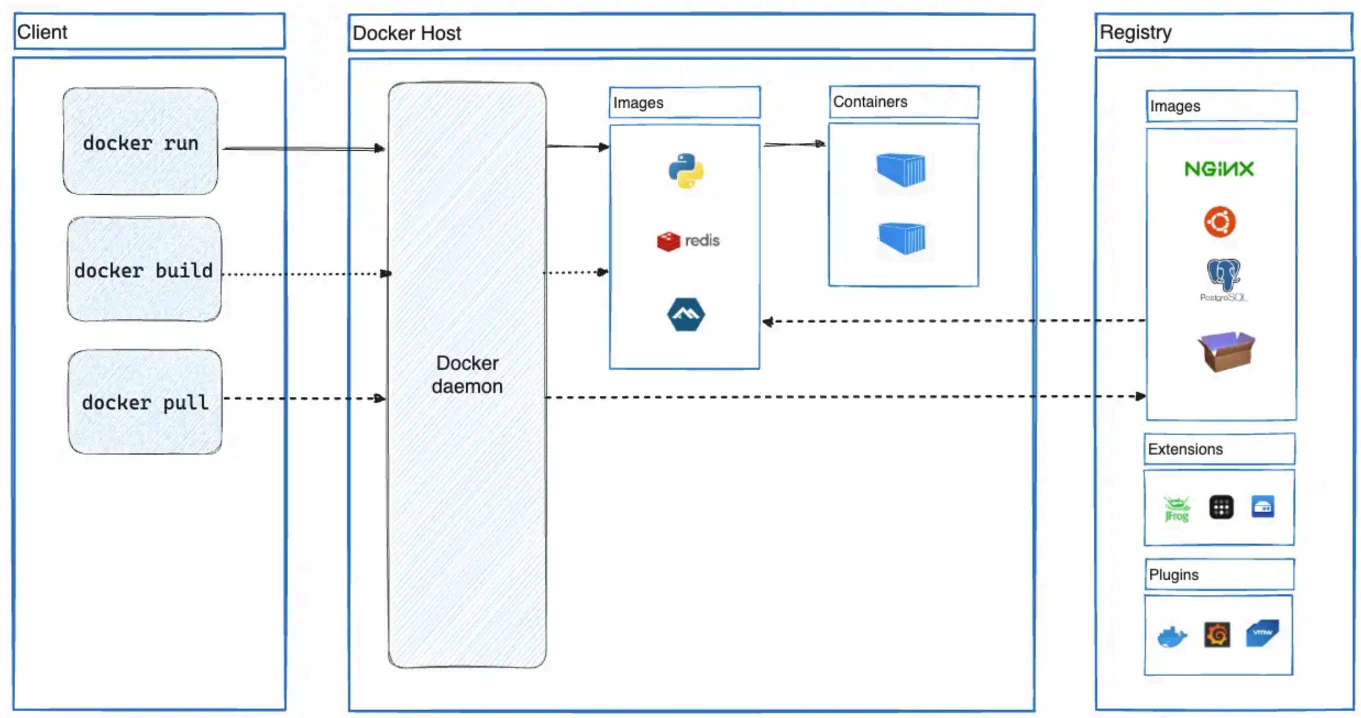
Now, let’s deploy the same app using containers with Docker:

1. **Setup**:
    - You create 3 containers:
        - Container 1: React frontend, based on a lightweight Node.js image.
        - Container 2: Flask backend, based on a Python image.
        - Container 3: MySQL, using the official MySQL Docker image.
    - All these containers share the host system’s OS kernel, reducing overhead.
    -

## Key Advantages of Containers over VMs in This Scenario:

| Aspect | VMs | Containers |
|---|---|---|
| **Resource Usage** | Requires multiple full OS installations | Shares host OS; lightweight |
| **Setup Time** | Manual and time-consuming | Automated with Dockerfile |
| **Portability** | Limited to hypervisor compatibility | Runs on any system with Docker |
| **Performance** | High overhead; slower startup | Low overhead; starts in seconds |
| **Consistency** | Risk of environment drift | Always consistent (defined in code) |

# 002 Docker Architecture



Docker is an Container Engine used to run and manage containerized applications.
Docker consist of 3 main components.

1. **Docker Client**
2. **Docker Host**
3. **Docker Registry / Docker Hub**
   Now lets have a look of each of these component
   Lets take example scenario that we have to run nginx server inside our docker engine.

Prerequisites are:

- Docker is installed and running on our machine.
- to setup docker https://docs.docker.com/get-started/get-docker/

we have to simply run following command to get nginx image and run it as container.

```
docker run nginx
```

Now lets understand how nginx image is pulled from docker registry to our machine after which you will understand Architecture.

## Docker Client:

first when you run "docker run" command on command line you are actually using **docker client** to execute commands. docker client communicates to docker host for further process.

Note when you install docker on your machine both docker client and docker host are installed together.

### Docker Host:

Docker Host contains *Docker Engine*.
Docker Daemon (**dockerd**), **containerd** and **runc** are part of Docker Engine but for now just focus on Docker Daemon.
**Docker Daemon** also known as **dockerd** exposes an REST API to communicate to external world and performs functions such as image management and image build.
Our command "docker run ngnix" is sent to **docker Daemon** (dockerd) via REST API. docker client and docker daemon can be on same machine.

### Docker Registry:

Now dockerd checks for image of nginx in local machine if it doesnt exist request is forwarded to **docker Registry**.
Docker Registry/Hub is repository where all Public or Private docker images are present.
finally nginx image is pulled from docker registry to local machine and run as container.

### Images vs Containers

image is a template packaged application, dependencies together. Docker images are small, fast and light weight as compared with VM images. For example alpine is image of linux OS and is only around 5MB of size.

Container is running instance of image or we can say that when image is running in production it is called container. container can start and stopped there are various docker commands to do so we will discuss in later articles.

## The Docker Engine

We have learned basic Components of Docker i.e Docker Client, Docker Daemon and Docker Registry but in fact this isn't whole picture.
When we pulled ngnix image from docker registry and run it as container then behind the scenes there are resources allocated from CPU for our container and container should run on isolated environment. so there are modules that are responsible for such task and to communicate with Kernel.

Looking back to evolution of Docker Engine, when first Docker Engine was released it consist of two major components;

- Docker Daemon
- LXC

The Docker daemon was a monolithic binary – it contained all the code for API, the
runtime, image builds, and more.
LXC provided the daemon with access to the fundamental building-blocks of
containers that existed in the Linux kernel. Things like namespaces and control
groups (cgroups).

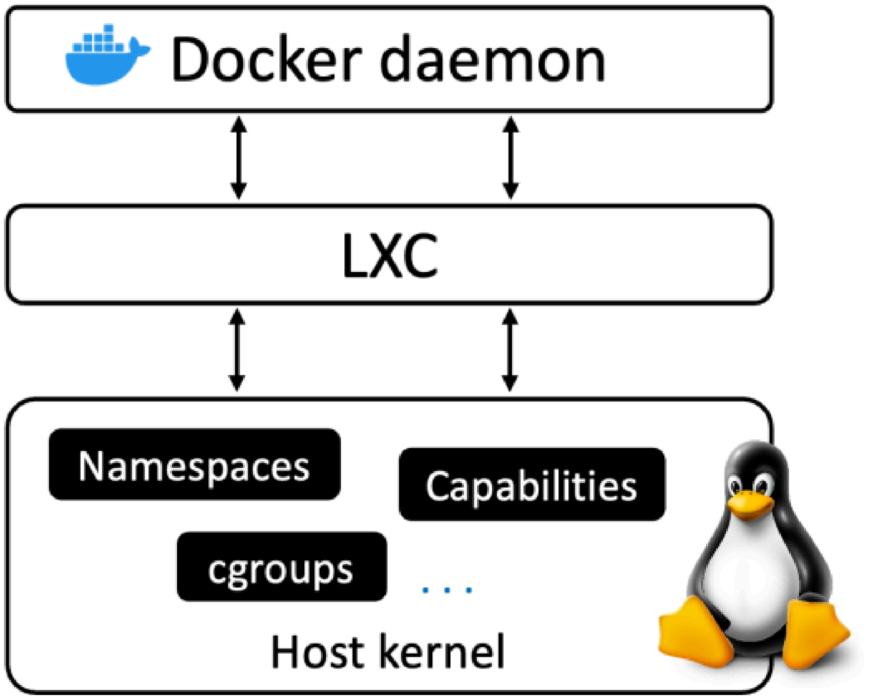


**Figure 5.2 Old Docker architecture.**

There was problems with this Design of Docker Engine.
First Docker Daemon was monolithic means it contained all the code for API, the
runtime, image builds, and more. every thing to run the containers and API calls etc are all
packaged inside monolithic binary of docker daemon.
due to monolithic nature of dockerd it got slower over time and harder to innovate it.
so Docker Inc decided to get rid of monolithic design of dockerd and convert it into modular
design.
The aim of this work was to break out as much of the functionality as possible from the daemon
and re-implement it in smaller specialized tools.

LXC was linux specific and is external tool used by docker engine, so Docker Inc build their own
tool called Libcontainer as replacement of LXC.

## Modern Docker Engine

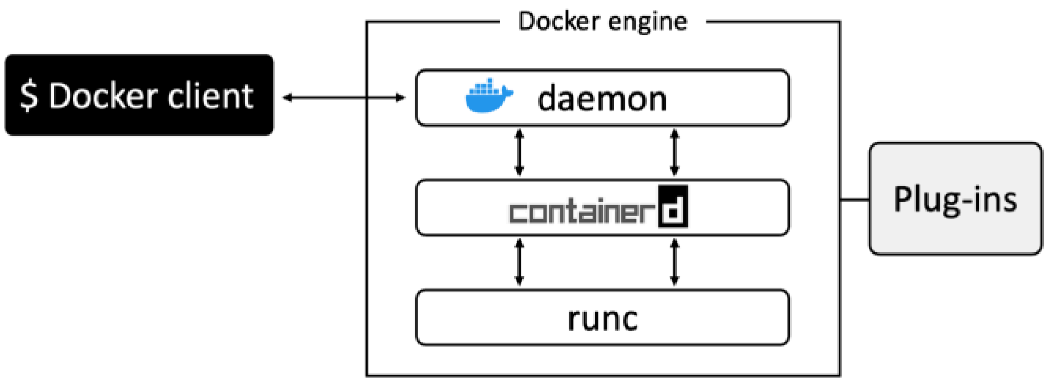


**Figure 5.1**

Docker Engine nowadays consist of:

- **Docker daemon (dockerd)**
- **High Level Runtime (Containerd)**
- **Low Level Runtime (runc)**
- **Shim**

  Now lets have brief over view of each component.

### Docker Daemon (dockerd):

Initially Docker Daemon contains all code for API, Runtime, build and manage images but now Docker daemon no longer contains any container runtime code — all container runtime code is implemented in a separate OCI-compliant layer.
Docker Daemon is Responsible of communicating to external world by exposing docker API, Orchestrates and manages high-level Docker tasks.
Dockerd act as orchestrator.

## Runtime:

A **container runtime** is software responsible for executing containers on a Host Operating System. It provides the necessary environment and tools to run containerized applications, ensuring they are isolated, portable, and lightweight.
Docker Engine uses two container runtimes in conjuction with docker daemon:
containerd; a high level runtime
runc; low level runtime

### High Level Runtime (Containerd):

High level Runtime provides **Container Life Cycle Management capabilities** such as Start, Stop, Pause, Remove Containers.

containerd a high level runtime was originally developed by Docker, Inc. and donated to the Cloud Native Computing Foundation (CNCF).

## Low Level Runtime (runc):

A low-level container runtime is responsible for **executing the container process** and directly interacting with the host operating system. It focuses on process isolation and compliance with runtime specifications.
Runc is low level runtime used by docker engine to **create containers** and execute it as standalone process in isolated environment.
runc interfaces with the OS kernel to pull together all of the constructs necessary to create a container (namespaces, cgroups etc.)

every container on docker is instance of runc.
The most important task of runc is to ensure that every running container is **OCI compliant**.

**OCI (Open Container Initiative)**: is a project by OCI community that defines standard specifications for images and container runtime.
OCI Image spec (https://github.com/opencontainers/image-spec)
OCI runtime specs (https://github.com/opencontainers/runtime-spec)

## Responsibilities of runc

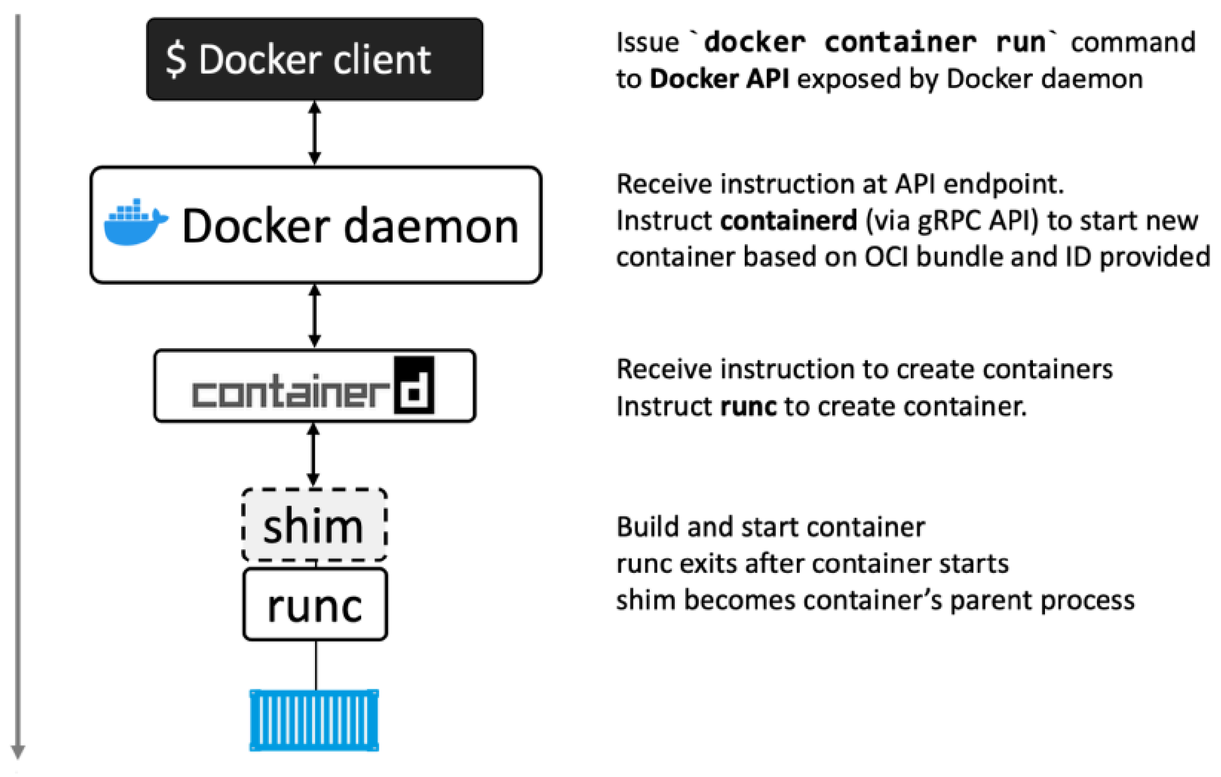
1. **Container Execution:**
    - Executes containers based on instructions from the high-level runtime.
    - Directly starts the container's main process.
2. **Isolation:**
    - Ensures containers are isolated using Linux kernel features:
        - **Namespaces:** For isolating processes, networking, and filesystems.
        - **Control Groups (cgroups):** For limiting CPU, memory, and other resources.
3. **Kernel Interactions:**
    - Works closely with the OS kernel to enforce container resource limits and isolation.
4. **OCI Compliance:**
    - Adheres to the **Open Container Initiative (OCI) Runtime Specification** to ensure standardization and compatibility.

## Benefits of Modular Docker Engine:

Now lets discuss why docker engine has evolved from older monolithic approach where docker daemon (dockerd) has a lots of responsibilities to modular approach where all logic and code to start and manage containers is decoupled from dockerd and shifted to runtime.
this approach of running containers is sometimes called daemonless containers.

1. **Modularity:** You can swap components (e.g., replace `runc` with `crun`) without affecting the entire stack.
2. **Standardization:** High-level runtimes can use any OCI-compliant low-level runtime.
3. **Flexibility:** Different use cases can leverage specialized runtimes for performance or security.
4. **Upgrading docker daemon seamlessly:** Having container runtime decoupled from docker daemon makes it possible to perform maintenance and upgrades on the Docker daemon without impacting running containers.

### How dockerd, containerd and runc work together.



Issue `docker container run` command to **Docker API** exposed by Docker daemon

Receive instruction at API endpoint.
Instruct **containerd** (via gRPC API) to start new container based on OCI bundle and ID provided

Receive instruction to create containers
Instruct **runc** to create container.

Build and start container
runc exits after container starts
shim becomes container's parent process

**Figure 5.4**

## 003 Docker Images and Containers.

## Images

Image / Docker Image /Container Image / OCI Image
Images are stopped containers.
Image >> Build time construct
Container >> Runtime Construct

## Images as layers

Images are madeup of multiple readonly layers.
An image starts with a base layer for example ubuntu OS as base layer, and then each time changes are made to base layer these changes are stacked on base layer and new layer is added.
Take example Python Application Image.
Layer 1: Ubuntu 22.04 (base)
Layer 2: Python 3.7
Layer 3: Source Code

### Storage Drivers

Docker has storage drivers whose job is to stack layers of image and present it as single unified file system. so all image layers are merged by storage driver and updated layer is placed at top of stack. the layer above stack contains combination of all layers under it.

All images in local repo are stored under storage drivers.
/var/lib/docker/storage-driver
storage drivers are overlay2, device mapper, btrfs, zfs

### Image Digest

An Image digest is also called as image ID.
Each layer of image is enrypted with SHA256 Hash, which is ID of that layer.
Each image has its unique ID, as its hashed so it ensures integrity of image. Any changes made to any of the layer of image results different Hash and different ID.

Docker inspect command shows all layers an image is made with along with their IDs

```
 docker inspect ubuntu:latest
 [     {
        "Id": "sha256:bd3d4369ae.......fa2645f5699037d7d8c6b415a10",
"RepoTags": [

            "ubuntu:latest"

        <Snip>

         "Layers": [
"sha256:c8a75145fc...894129005e461a43875a094b93412",
"sha256:c6f2b330b6...7214ed6aac305dd03f70b95cdc610",
"sha256:055757a193...3a9565d78962c7f368d5ac5984998",
"sha256:4837348061...12695f548406ea77feb5074e195e3",
"sha256:0cad5e07ba...4bae4cfc66b376265e16c32a0aae9"             ]         }
}   ]
```

## Sharing of image layers

Image Layers can be shared amongst each other.
This can be seen by following pull example:

```
$ docker pull -a nigelpoulton/tu-demo latest: Pulling from nigelpoulton/tu-
demo aad63a933944: Pull complete  f229563217f5: Pull complete  <Snip>>
Digest: sha256:c9f8e18822...6cbb9a74cf

v1: Pulling from nigelpoulton/tu-demo
aad63a933944: Already exists
f229563217f5: Already exists
```

Image layers that are already available in our local repo will not be pulled and they are shared between images.
Sharing of images enhances effeciency.

**Commands Related to images:**

- docker images
- docker inspect image
- docker history image
- docker rmi [image:tag]
- docker pull [repository:tag]
  repository = name/path of repo on docker hub
  tag = version of image
- docker images --filter dangling=true

# Containers

A container is Run time instance of an image.

An image is like a template and we call it build time construct. When image is running inside Container Engine we call it a container.

### Running a container.

Now let us Run an image as container.
command for running container is.
**docker run [-flags] [image-name] [command]**

```
docker run -it ubuntu /bin/bash
```

above command takes ubuntu image and run it as container.
-it flag opens interactive interface i.e. connects our terminal to the container shell.
/bin/bash is the command or process we want to run inside our container.
after executing above command our shell will look like this.

```
root@e37f24dc7e0a:/#
```
It means that we are inside container.

Now here question may arise that what is lifecycle of this container will it run forever or it may die?
answer is simple that container kept running until there is one or more process running inside it.
so inside our container terminal lets see running processes.
```
root@e37f24dc7e0a:/#ps -elf
F S UID PID PPID NI ADDR SZ WCHAN STIME TTY TIME CMD
4 S root 1 0 0 - 4558 wait 00:47 ? 00:00:00 /bin/bash
0 R root 11 1 0 - 8604 - 00:52 ? 00:00:00 ps -elf
```

You can see that we have /bin/bash Parent Process with PID = 1.
Killing main process (PID=1) in the container will also kill the container.
to confirm it type **exit** inside container shell. it will exit or kill /bin/shell process and you will be back to your host terminal.

now lets check running containers inside our docker engine, for which command is.
```
docker ps
```
you shouldn't see that container, it means conatiner is terminated as now process is running inside it.

Now lets run container again.
`docker run -it ubuntu /bin/bash
now instead of exit command hit Ctrl+PQ
which is signal to run shell terminal in background without terminating it.
This time if we check running containers under docker host we will see name of out container.

```
$ docker ps
CNTNR ID IMAGE COMMAND CREATED STATUS NAMES
e37..7e0a ubuntu:latest /bin/bash 6 mins Up 6mins sick_montalcini
```

Now you may ask how do i access my container shell again as it is running in background. You can do it by reattaching your container with a new shell terminal.

```
==`docker exec -it e37f24dc7e0a bash
root@e37f24dc7e0a:/#
```

above command will reattach running container with new bash terminal.

now if we check running processes inside our container we will see that there are two bash process.
one with PID = 1 this is our main process which when terminated container will be stopped.
second is of other bash terminal through which we have reattached our conatiner using docker exec command.
There is quick quiz for you,
what will happen when we type **exit** command now?
write your answer in comments.

**Important Container Commands:**

**To Start a container:**

docker run [-it] [image-name or ID] [/bin/bash]
Take image as input and start container out of it in interactive shell mode.

docker run [-d] [image-name or ID] [command]
Start container in Detached Mode (in Background), Container will exit after "command" is finished.
docker run [-d] [--name webserver] [image-name or ID]
Start Container in detached mode, also give it name as webserver.

docker exec [-it] [image-name or ID] [bash]
Reattach new shell terminal to container.

**To list Containers**

docker ps : Lists all Running Containers
dockers ps -a : Lists all Stopped + Running Conatiners.

**To Stop and Delete Containers.**

docker stop [Container-Name or ID]
Will stop a running container.

docker rm [Container-Name or ID]
will remove/delete container. it is recommended to stop it first.

docker rmi $(docker ps -aq) -f
will delete all containers without warning.

## Self-healing containers with restart policies

Containers can stop running due to following possible reasons.

1. We Stop it Manually.
2. Due to Failure (Returns Non-Zero exit code).
3. Docker daemon Stops.
   Docker provides us opportunity to write restart policies for containers. Containers can Automatically restart if stop due to any reason.

Following restart policies can be defined.
**no** : Don't Automatically Restart (Default).
**on-failure [:max-retries]** : Restart Container if it exits due to an error. Optionally we can set no of times container can restart using :max-retries.
**always** : Always restart container when it stops. If Its manually stopped,  it's restarted only when Docker daemon restarts or the container itself is manually restarted.
**unless-stopped** : Similar to `always`, except that when the container is stopped (manually or otherwise), it isn't restarted even after Docker daemon restarts.

# 004 Docker imp Commands.

systemctl status docker
systemctl start docker
docker version
docker images
ls /var/lib/docker
docker info
docker inspect [image name]
docker pull image:tag
docker images --digests [imagename]
docker --filter
docker --search
docker rmi alpine:latest
manifest list is list of architectures supported by an image
docker manifest inspect golang
docker manifest inspect golang | grep 'architecture|os'

# 005 Docker Build - Containerizing Applications.

## Docker File:****

**From**: Selects image from docker registry.

**Run** : used to execute a command in docker file.
docker run: used to run docker image as container

run apt-get update
run apt-get install -y curl
or
run apt-get update && run apt-get install -y curl.
only one intermediate layer will make by this cmd.
each run cmd creates a new layer in image except when multiple run cmds are joined using "&&".

**ENV**: used to define a variable.

ENV admin-user = "shujah"