Docker and its objects

Last amended: 23rd Nov, 2024
My folder: C:\Users\ashok\OneDrive\Documents\docker
References: Ref1; Ref2;

Table of Contents

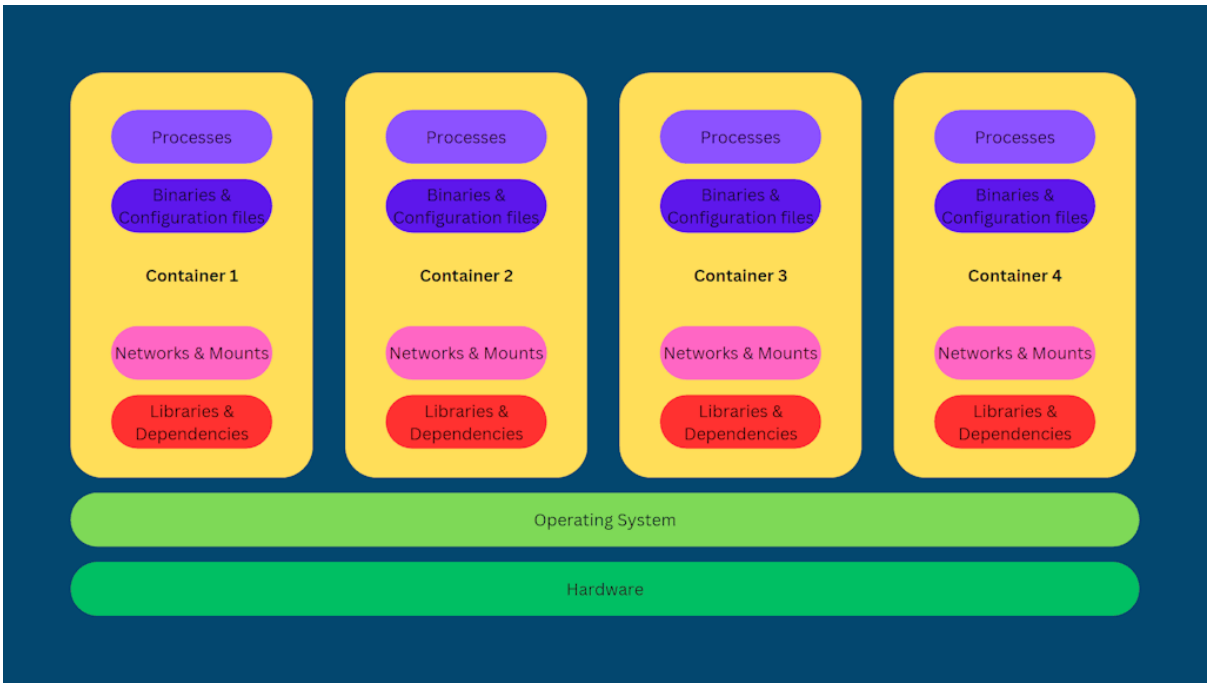
What is docker?

Docker is an open-source platform for building, packaging and running applications. Docker enables one to separate the applications from the infrastructure—hard and soft. It may be better to talk about Container first.

What is a Container:

A container is a lightweight and isolated environment that allows you to package and run an application and its dependencies. It encapsulates the software and all its dependencies into a single, self-contained unit that can run consistently across different computing environments.

Containers provide a consistent and reliable runtime environment, ensuring that an application will run the same way regardless of the host system. They achieve this by leveraging operating system-level virtualization, which allows multiple containers to run on the same host while remaining isolated from one another.



The history and development of containers can be traced back to the early 2000s, although the concept of process isolation and virtualization dates back even further.

Docker and chroot

1. Chroot: In the 1970s, the Unix operating system introduced the chroot system call, which allowed a process to have its root directory set to a different location. This provided a level of isolation by limiting a process's access to files and directories outside its designated root directory.

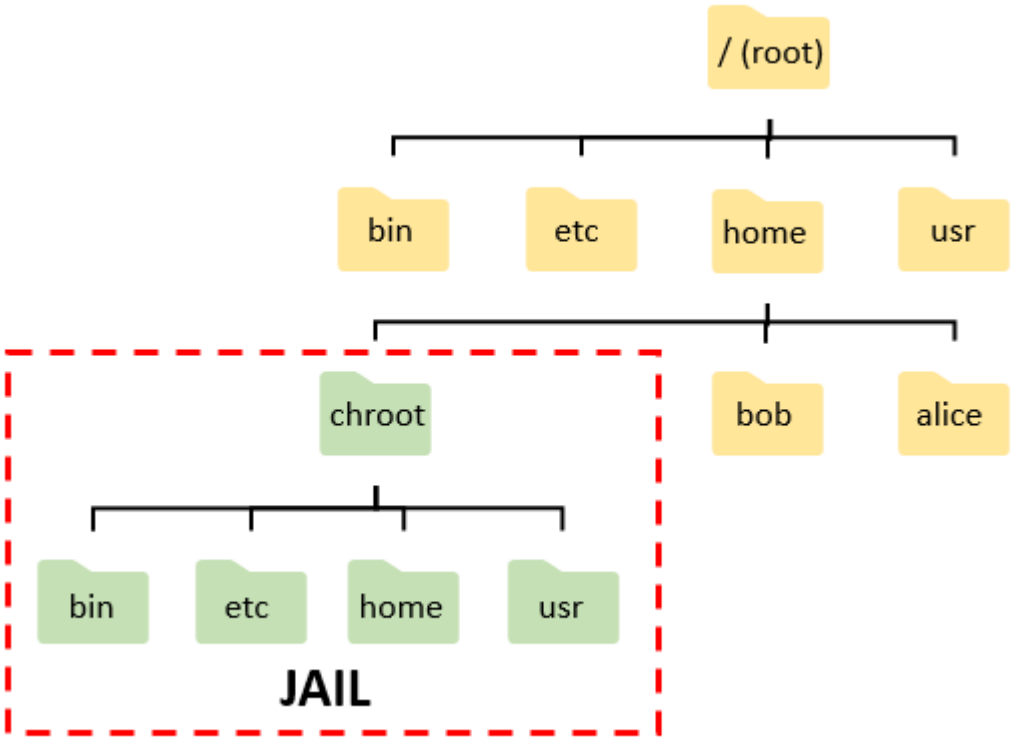


Figure 1: IF a service (such as web service) in the jail is compromised by an attacker, the vulnerability in this service cannot access critical resources outside chroot. Hence critical resources remain protected.

chroot creates a safe environment, separate from the rest of the system. Processes created in the chrooted environment cannot access files or resources outside of it. For that reason, compromising a service (such as web server or database server) running in a chrooted environment would not allow the attacker to compromise the entire system. (See this reference and this reference)

2. Jails: In the early 2000s, the FreeBSD operating system introduced a feature called "jails." Jails provided a lightweight form of virtualization by creating isolated environments within a host system. Each jail had its own file system, network stack, and processes, allowing multiple applications to run independently on the same physical machine.

3. Linux Containers (LXC): In 2008, the Linux Containers project was launched, aiming to provide lightweight operating system-level virtualization on Linux. LXC introduced the concept of control groups (cgroups) and namespaces, which allowed for resource allocation and isolation of processes, networks, and file systems. LXC was a significant step towards the modern container technology we use today.

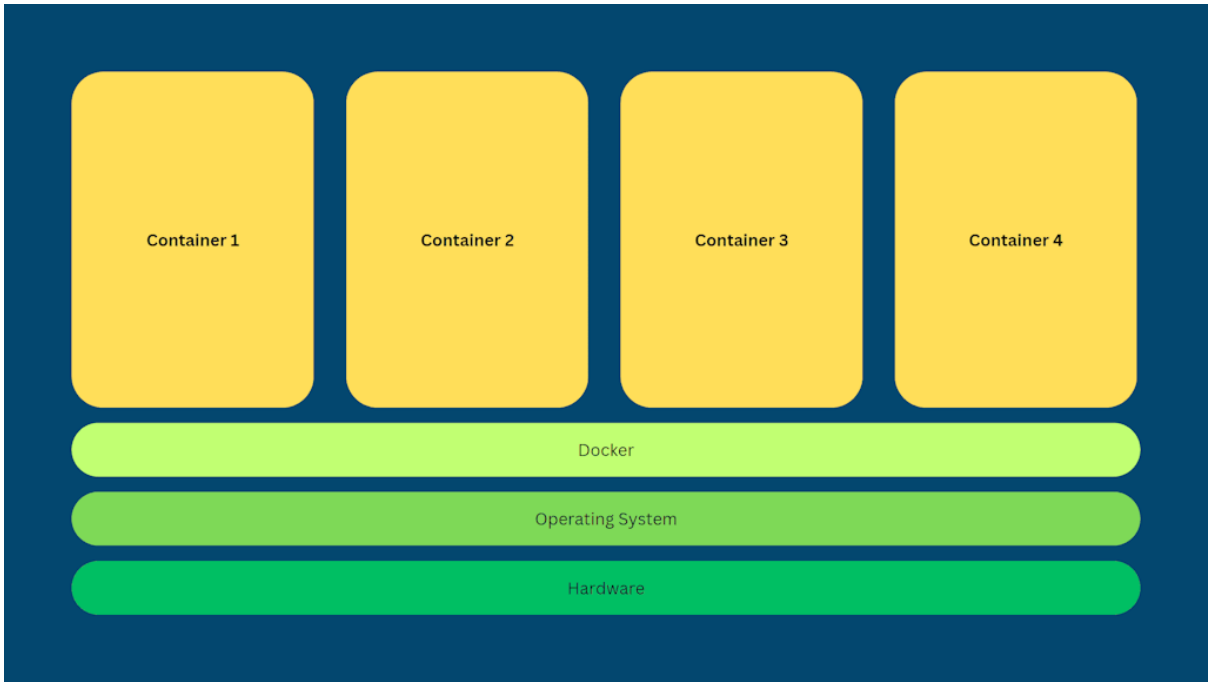
4. Docker: In 2013, Docker was released and had a transformative impact on the container ecosystem. Docker simplified the process of creating, managing, and distributing containers by introducing a user-friendly interface and a powerful

toolset. It popularized the use of container images, which are pre-built, portable packages containing the application and its dependencies. Docker also introduced a distributed registry called Docker Hub, making it easy to share and discover container images.

5. Container Orchestration: As container adoption grew, the need for managing and orchestrating large-scale deployments emerged. Several container orchestration platforms, such as Kubernetes, Docker Swarm, and Apache Mesos, were developed to address this demand. These platforms provided capabilities for automated scaling, load balancing, service discovery, and fault tolerance, enabling the management of containerized applications at scale.
6. Container Standards: To ensure interoperability and portability between different container platforms, industry standards like the Open Container Initiative (OCI) were established. The OCI defined specifications for container runtime and image formats, fostering compatibility across various container runtimes and tools.

What is Docker?

Docker is an open-source platform that enables developers to automate the **deployment** and **management** of applications within containers. It provides a standardized and efficient way to package applications and their dependencies into portable, self-contained units called container images. Docker provides user interfaces to create, configure, and manage Containers.



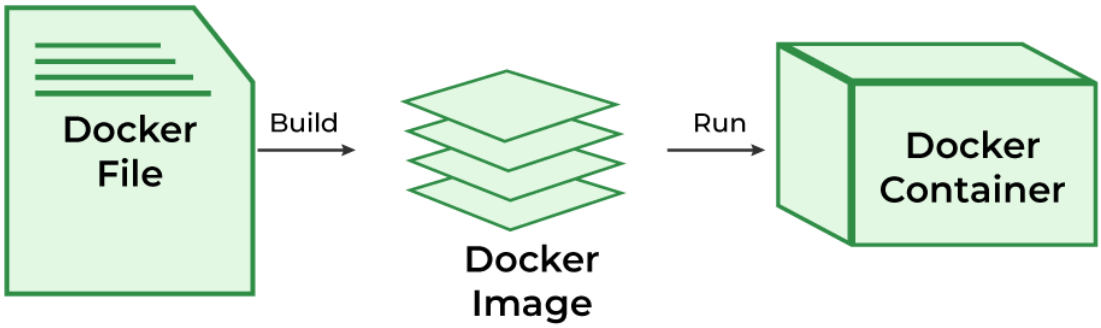
At its core, Docker utilizes containerization technology to create isolated environments where applications can run consistently across different computing environments. Containers created with Docker are lightweight, fast to start, and share the host system's operating system kernel, while still maintaining process isolation from one another.

Docker Images

A Docker image is a file that contains the instructions and files needed to create a Docker container:

- **Instructions**: A Docker image is a template that provides instructions for building a Docker container.
- **Files**: A Docker image contains the files needed for an application to run, including libraries, dependencies, application code, and tools.
- **Read-only**: A Docker image is a read-only file, meaning it can't be changed.

Docker images are portable and shareable, so you can use the same image in multiple locations.



You can create your own images or use images created by others that are published in a registry. For example, the public Docker Hub registry contains images for operating systems, databases, programming language frameworks, and code editors.

What is a Dockerfile?

A Dockerfile is a text file that contains instructions for building a container image. Docker uses the instructions in a Dockerfile to build an image. The instructions include commands to run, files to copy, and the startup command.

How it works

When you run the Docker run command, Docker uses the Dockerfile to build the image. A Dockerfile can include instructions like:

- **FROM**: Defines the base image for the image
- **RUN**: Executes commands in a new layer on top of the current image
- **WORKDIR**: Sets the working directory for subsequent instructions
- **COPY**: Copies files or directories from one location to another
- **CMD:** Defines the default program that runs when the container starts

Dockerfile can be simple at first and grow to support more complex scenarios. See this command reference written in Dockerfile.

Does old data persist in docker?

As each time docker is started, containers are reconstructed from images. Hence, old data does NOT persist in dockers.

Virtual Disk Image(.vdi files)

We have worked with VDI files in Oracle VirtualBox. These vdi files are also images. A VDI image, or Virtual Disk Image, is a file format used by Oracle VM VirtualBox to create guest hard disks for new virtual machines:

- **File extension**: VDI files have a .vdi extension
- **Compatibility**: VDI is generally compatible with other virtualization programs
- **Storage allocation**: VDI supports both fixed-size and dynamically allocated storage
- **Performance**: VDI files are more compact and outperform VHD or VHDX files on some key criteria
- **Speed**: VDI files are slower than VMDK files
- **Backups**: VDI does not support incremental backups, but it does have high-level redundancy

Docker Registry or Hub

Conceptually one can think of Docker Registry as *Sales Registry* for lands and buildings where registered documents are listed, kept and made available. A Docker registry is a system for storing, versioning, and distributing Docker images. Docker Hub is Docker's official cloud-based registry, and is the default registry used when installing the Docker engine:

Docker Hub

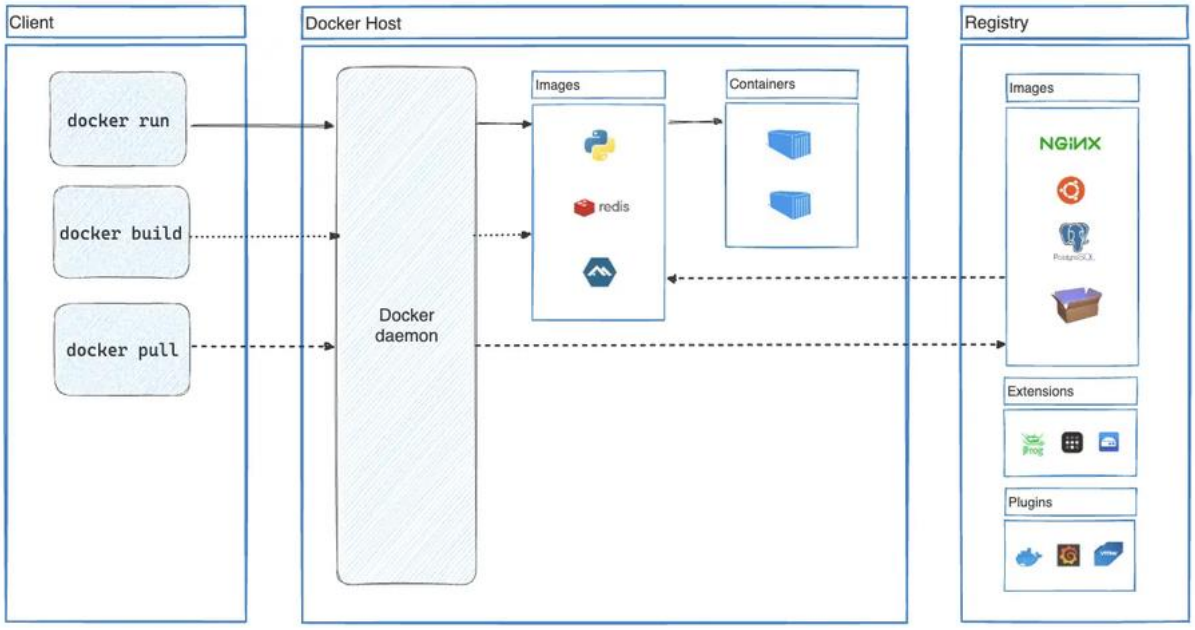
It is a public registry that anyone can use. It's a collaborative marketplace for developers and open source contributors. You can use Docker Hub to:

- Search, use, and share container images

- Host public repos for free
- Host private repos for teams and enterprises
- Access over 100,000 container images

Docker Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



Docker CLI: It is a client. Docker Command-Line Interface (CLI) is a command-line tool that allows users to interact with Docker and perform various container-related operations. It provides commands to build images, run containers, manage networks and volumes, and perform other essential tasks.

The Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command

uses the Docker API. The Docker client can communicate with more than one daemon.

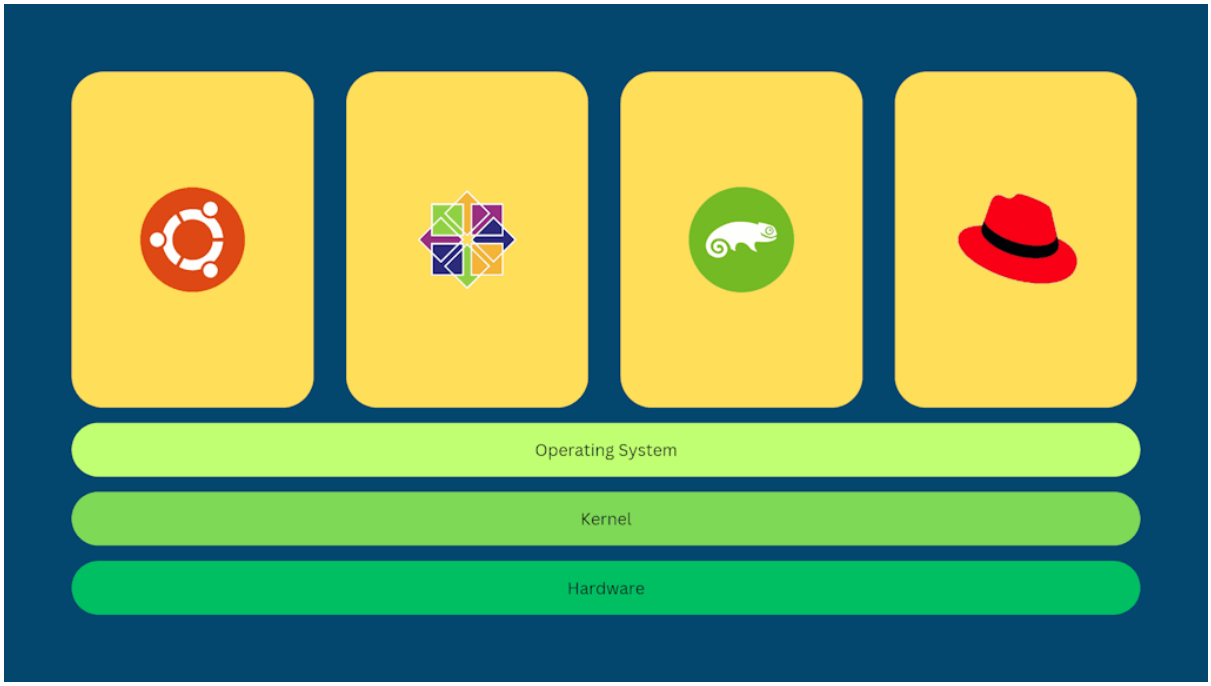
Docker Desktop

Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment. Installation on Ubuntu is little problematic. Its GUI enables one to build and share containerized applications. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

Docker desktop must be distinguished from Docker Engine. Docker Engine is everything the Docker desktop installs but without the GUI.

How Does Docker Work?

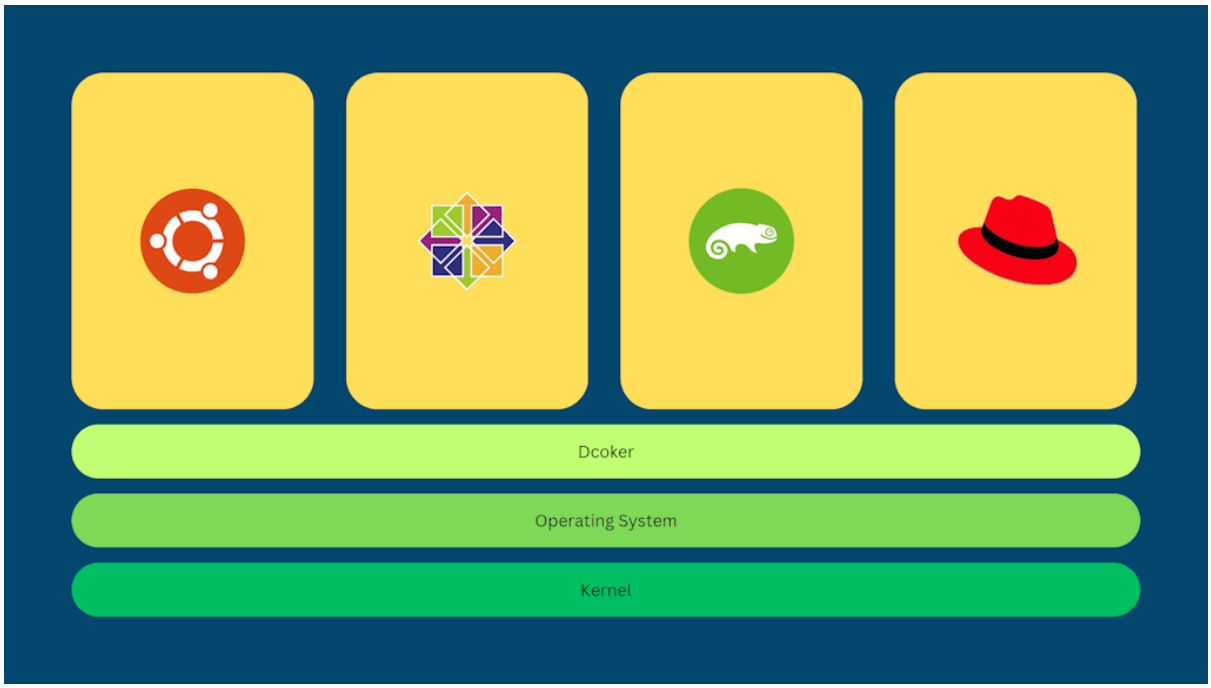
As you know, Containers, much like virtual machines, offer fully isolated environments where processes, services, network interfaces, and mounts can operate independently. However, the key distinction is that containers share the same operating system (OS) kernel. This sharing of the kernel ensures efficient resource utilization while maintaining isolation. Let's explore this in more detail.



To grasp the significance of containerization, it's essential to revisit fundamental concepts of operating systems. Operating systems like Ubuntu, Fedora, Suse, or Centos consist of two primary components: the OS kernel and a set of software.

The OS kernel acts as an intermediary between the hardware and software layers. In the case of Linux-based systems, the kernel remains the same across various distributions. It is the software layer above the kernel that distinguishes

different operating systems, encompassing aspects such as user interfaces, drivers, compilers, file managers, and developer tools.



Suppose we have a system with Ubuntu as the underlying OS and Docker installed. Docker enables the execution of containers based on different OS distributions as long as they share the same Linux kernel.

For example, if the host OS is Ubuntu, Docker can run containers based on distributions like Debian, Fedora, Suse, or Centos. Each Docker container contains only the additional software that distinguishes the respective operating systems. The underlying kernel of the Docker host seamlessly interacts with all the supported OS distributions.

Kernels of all Linux Distributions—Are they same?

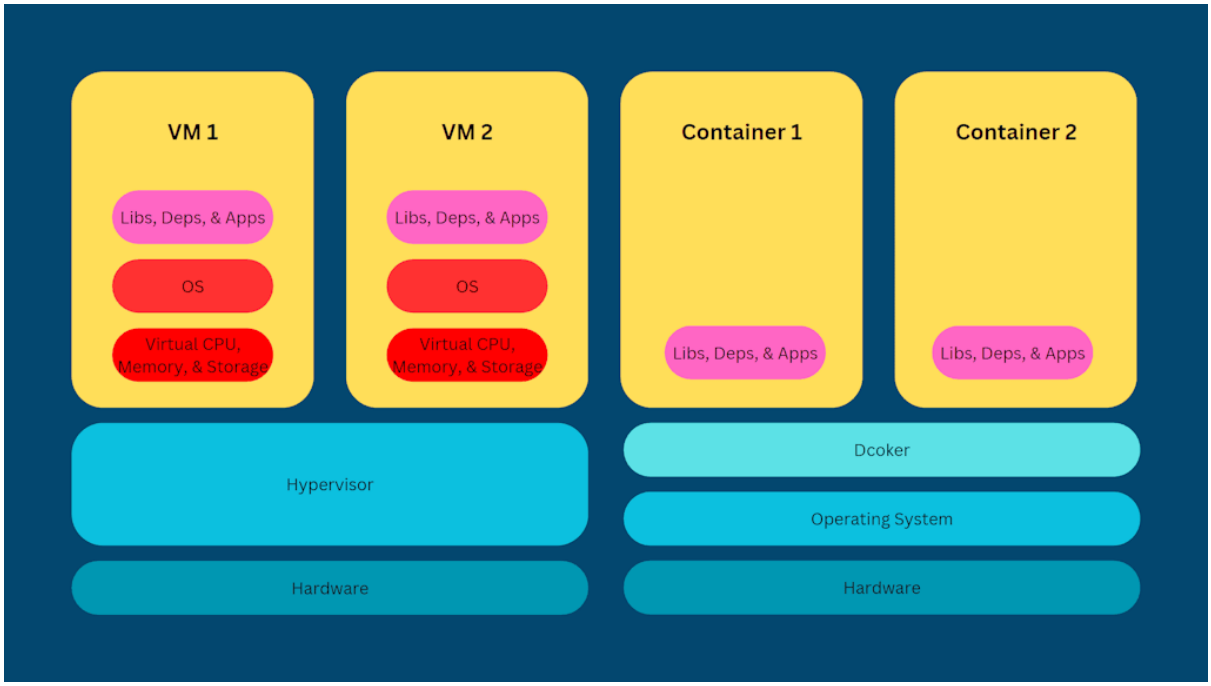
Here is a reply from Stack Overflow:

> All distros use the same "Linux" kernel, however all distros make slight changes to it in order make the kernel work best for them, however these changes will almost always get uploaded back to the top where Linus will merge them himself. So all use the Linux kernel, however they all have a few different lines of code in them to make them work best for that distro. It is also worth noting that distros will ship with the version of the kernel that they see fit for each version. Some distros choose a newer kernel then others.

Containers vs Virtual Machines

People from Virtualization backgrounds may often end up in confusion. The problem is not with them, the problem lies in some similarities between both technologies. Containers and virtual machines (VMs) are both technologies used for the isolation and deployment of applications, but they differ in their approach

and resource utilization. Let's do a comparison between containers and virtual machines:



Containers:

- Containers are lightweight and utilize operating system-level virtualization.
- They share the host system's operating system kernel, which makes them more efficient in terms of resource utilization.
- Containers have faster startup times and require less memory compared to virtual machines.
- They offer faster deployment and scaling capabilities.
- Containers provide process-level isolation, meaning each container runs in its own isolated environment while sharing the host's kernel.
- Container images contain only the necessary dependencies and libraries required for the application, making them smaller in size.
- Containers are well-suited for running microservices architectures and modern, cloud-native applications.
- They enable easier and more consistent application deployment across different environments.

Virtual Machines:

- Virtual machines, on the other hand, simulate the entire hardware stack and run a complete operating system.
- They require a hypervisor to emulate the virtual hardware and manage the guest operating systems.
- VMs are more resource-intensive as they allocate dedicated resources (CPU, memory, storage) to each instance.

- They have slower startup times compared to containers and require more memory due to the overhead of running a separate operating system.
- VMs provide stronger isolation between instances as they run independent operating systems.
- They are useful for running legacy applications, applications with specific OS requirements, or when complete isolation between instances is necessary.
- VMs offer more flexibility in terms of running different operating systems and configurations on the same physical hardware.

Amending *docker-compose.yml*

File, *docker-compose.yml*, can be amended to assign container names. Here is a sample *docker-compse.yml* file:

```
version: '2.1'
services:
  sql-client:
    image: jark/demo-sql-client:0.2
    container_name: sql-client
    depends_on:
      - kafka
      - jobmanager
      - elasticsearch
    environment:
      FLINK_JOBMANAGER_HOST: jobmanager
      ZOOKEEPER_CONNECT: zookeeper
      KAFKA_BOOTSTRAP: kafka
      MYSQL_HOST: mysql
      ES_HOST: elasticsearch
  jobmanager:
    image: flink:1.11.0-scala_2.11
    container_name: flinkjobmanager
    ports:
      - "8081:8081"
    command: jobmanager
    environment:
      - |
        FLINK_PROPERTIES=
        jobmanager.rpc.address: jobmanager
  taskmanager:
    image: flink:1.11.0-scala_2.11
    container_name: flinktaskmanager
    depends_on:
      - jobmanager
    command: taskmanager
    environment:
      - |
        FLINK_PROPERTIES=
        jobmanager.rpc.address: jobmanager
        taskmanager.numberOfTaskSlots: 10
  datagen:
    image: jark/datagen:0.2
    container_name: datagen
    command: "java -classpath /opt/datagen/flink-sql-demo.jar
myflink.SourceGenerator --input /opt/datagen/user_behavior.log
--output kafka kafka:9094 --speedup 2000"
    depends_on:
      - kafka
    environment:
      ZOOKEEPER_CONNECT: zookeeper
      KAFKA_BOOTSTRAP: kafka
  mysql:
    image: jark/mysql-example:0.2
```

```
    container_name: mysql
    ports:
      - "3306:3306"
    environment:
      - MYSQL_ROOT_PASSWORD=123456
  zookeeper:
    image: wurstmeister/zookeeper:3.4.6
    container_name: zookeeper
    ports:
      - "2181:2181"
  kafka:
    image: wurstmeister/kafka:2.12-2.2.1
    container_name: kafka
    ports:
      - "9092:9092"
      - "9094:9094"
    depends_on:
      - zookeeper
    environment:
      -
KAFKA_ADVERTISED_LISTENERS=INSIDE://:9094,OUTSIDE://localhost:9
092
      - KAFKA_LISTENERS=INSIDE://:9094,OUTSIDE://:9092
      -
KAFKA_LISTENER_SECURITY_PROTOCOL_MAP=INSIDE:PLAINTEXT,OUTSIDE:P
LAINTEXT
      - KAFKA_INTER_BROKER_LISTENER_NAME=INSIDE
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
      - KAFKA_CREATE_TOPICS="user_behavior:1:1"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.6.0
    container_name: elasticsearch
    environment:
      - cluster.name=docker-cluster
      - bootstrap.memory_lock=true
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
      - discovery.type=single-node
    ports:
      - "9200:9200"
      - "9300:9300"
    ulimits:
      memlock:
        soft: -1
        hard: -1
      nofile:
        soft: 65536
        hard: 65536
  kibana:
    image: docker.elastic.co/kibana/kibana:7.6.0
    container_name: kibana
    ports:
      - "5601:5601"
```

In this file the outermost-indented-names (in bold) are the names of docker *services*. Thus, the nine services are:

sql-client
jobmanager
taskmanager
datagen
mysql
zookeeper
kafka
elasticsearch

```
kibana
```

You can start each individual service by its name. If a service is dependent on another service, that service will also be started. List of all services on which a particular service depends is listed under the paragraph depends_on. See this example from the above file:

```
kafka:
  image: wurstmeister/kafka:2.12-2.2.1
  container_name: kafka
  ports:
    - "9092:9092"
    - "9094:9094"
  depends_on:     <=== kafka service depends upon zookeeper service
    - zookeeper
```

Code to start all services but '*datagen*' would be as below. Below, '*datagen*' service is commented out.:

```
sudo docker-compose up -d sql-client
sudo docker-compose up -d jobmanager
sudo docker-compose up -d taskmanager
#sudo docker-compose up -d datagen          ← Commented out
sudo docker-compose up -d mysql
sudo docker-compose up -d zookeeper
sudo docker-compose up -d kafka
sudo docker-compose up -d elasticsearch
sudo docker-compose up -d kibana
```

Also, each container can be assigned a name. We have assigned names to each container by writing a line immediately after *image* name, as for example:

```
image: docker.elastic.co/elasticsearch/elasticsearch:7.6.0
container_name: elasticsearch     ← Container name (lowercase)
```

Container names must be in lowercase.

Start all containers at once

To start all containers in docker issue command in the same folder where docker-compose.yml lives.

```
docker-compose up -d
```

This starts docker in detached mode (`flag -d`). That is, after the docker containers are started, terminal is made available.

How an image is created?

There are two ways to create docker images:

- The first way is to create them **using a Dockerfile**. A Dockerfile contains a set of instructions that runs on top of the parent image and creates an intermediate

container for every instruction. After completion of an instruction, the context is sent forward for the next instruction.

- The second way is by **creating the image from a base image**. All Dockerfiles start from a base image. A base is the image that your image extends. It refers to the contents of the FROM instruction in the Dockerfile.

```
FROM debian
```

For most cases, you don't need to create your own base image. Docker Hub contains a vast library of Docker images that are suitable for use as a base image in your build. Docker Official Images are specifically designed as a set of hardened, battle-tested images that support a wide variety of platforms, languages, and frameworks. There are also Docker Verified Publisher images, created by trusted publishing partners, verified by Docker.

Docker hubs

Here are some links to docker hubs. Plenty of docker images are available in these repositories.

Docker hub
Official images
Verified Publisher content

Docker files are also scattered throughout github.

Using Docker Hub

The following is the *Docker Hub* page of Postgres database (a highly reputed open-source database):

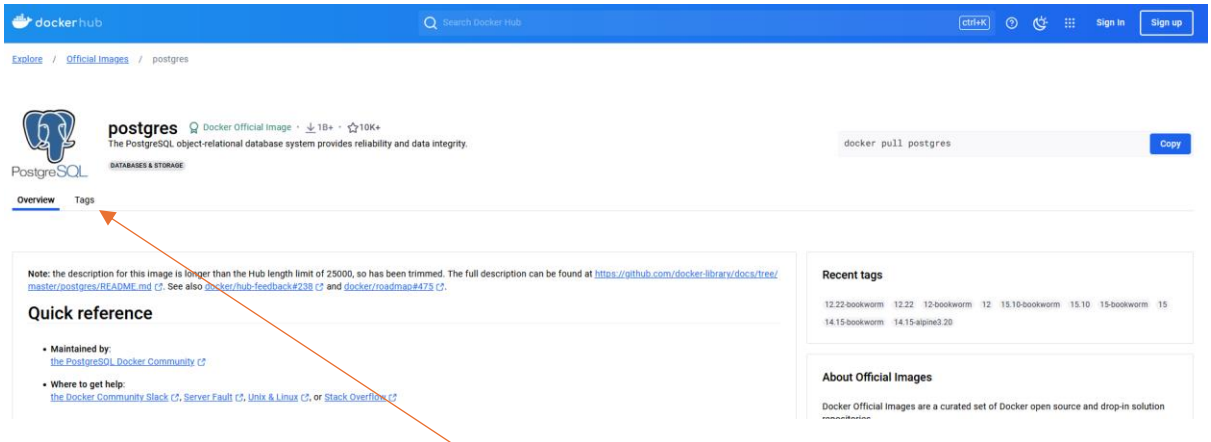


Figure 2: Notice the two tabs: Overview and Tags

Overview contains the general installation steps. *Tags* contain the list of current image(s) and earlier versions. For Postgres, the screenshot of *Tags* page is below:

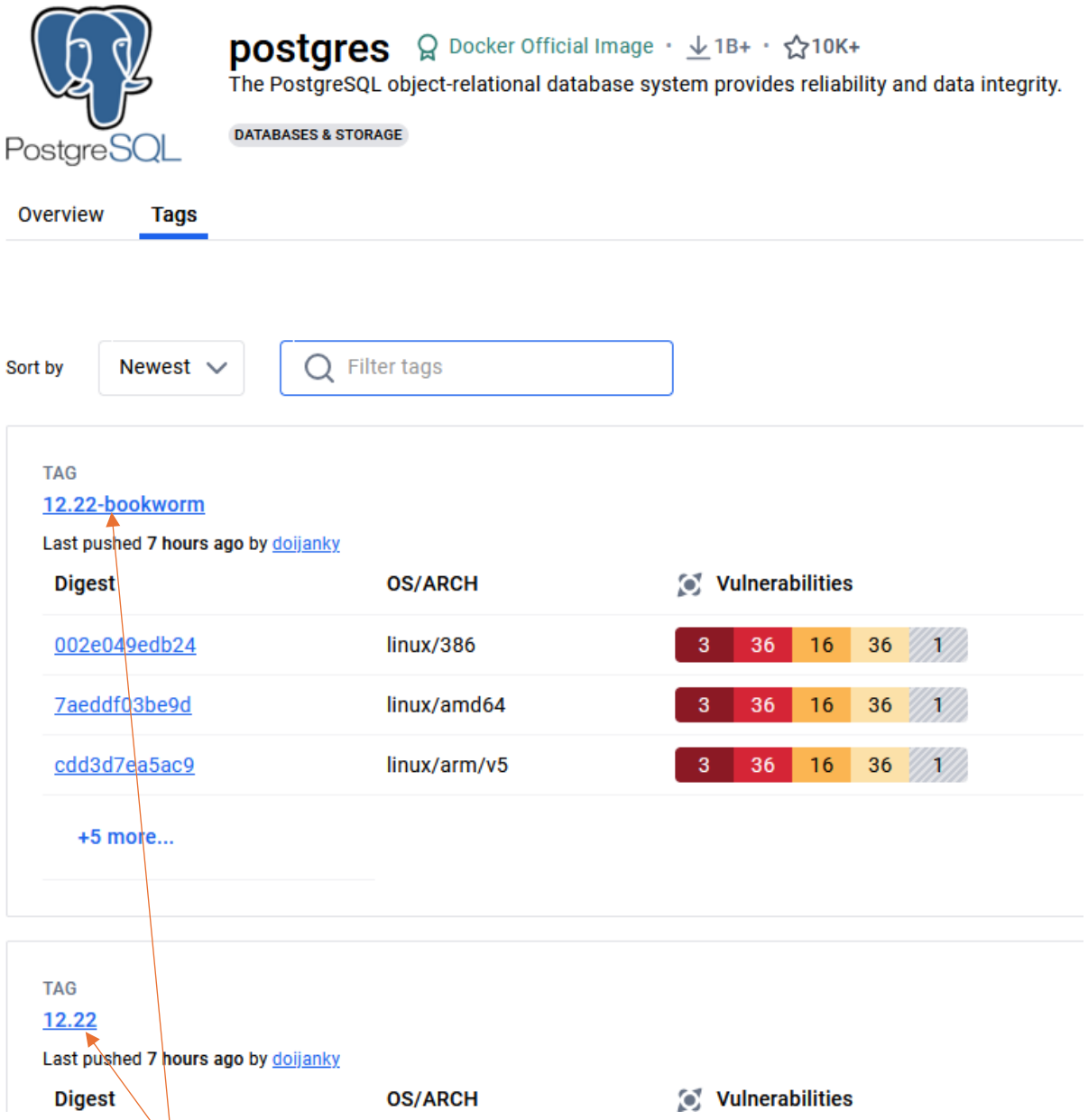


Figure 3: Two versions of images can be seen here.

For every image, there is a docker pull command:

Figure 4: Docker pull commands for the two versions of images.

Generally, we prefer to use the *pull* command for the latest image rather than for the earlier versions.

Installation steps for dockers hosted on GitHub

Many a docker compose (*docker-compose.yml*) files are on GitHub. But to use those files, the site advises you to first clone the GitHub repository on your ubuntu machine, then enter (i.e. `cd` to) the folder where the cloned files have been copied to and then issue the command `docker-compose up -d` to build images, install images and run containers. An example of this sequence of code is below:

```
# 1.0 Open Ubuntu terminal and clone the GitHub repo.
#     The GitHub URL will vary from repo to repo.

      git clone https://github.com/frappe/frappe_docker

# 1.0.1 Next, in the terminal, cd to the cloned folder:

      cd frappe_docker

# 1.0.2 Finally issue docker-compose command to build and
#       1install images and start containers:

      docker-compose -f pwd.yml up -d

      (Here pwd.yml is the docker-compose.yml file.
       Flag -f is for the .yml file)
```

Dockerfile vs *docker-compose.yml* files

On GitHub, you may find two files: one named as: *Dockerfile* and the other named as '*docker-compose.yml*' file. File, '*docker-compose.yml*' uses instructions from *Dockerfile:*

- to build images,
- to install the built images on your file system, and
- to start containers from images.

The reason why *Dockerfile* is needed on GitHub is that already built docker images cannot be uploaded on GitHub—image sizes being very large. This is unlike the arrangement in *Docker Hub* where instructions exist to download (using `docker pull` command) already built images and then install them on your machine (using `docker-compose up -d` command). Docker Hub does not require you to build images—they are already there.

So, *Dockerfile* contains instructions to first download all the specified different software to your Ubuntu machine so as to use them to first build images. Once this is done, these images are installed by `docker-compose` command in the Ubuntu file system.

##########