**V**irtualization is a process of creating a software-based version of something whether that can be compute, storage, networking, servers or applications.

Virtual Machine is basically a software-based computer. Virtualization uses software to create an abstraction layer over computer hardware that allows the hardware elements of a single computer—processors, memory, storage and more—to be divided into multiple virtual computers, commonly called virtual machines (VMs). Each VM runs its own operating system (OS) and behaves like an independent computer, even though it is running on just a portion of the actual underlying computer hardware.

Each VM requires its own underlying OS, and the hardware is virtualized.

A hypervisor is software, firmware, or hardware that creates and runs VMs.

There are two types of Hypervisors

1. Type1 or Bare Metal Hypervisor
2. Type2 or Hosted Hypervisor

|  |  |
| --- | --- |
| 1. **Bare Metal Hypervisor:** 2. When a Hypervisor is installed on the hardware of a physical machine it’s called bare metal Hypervisor. Some bare metal hypervisors are directly embedded in to the firmware at the motherboard level. 3. The term bare metal refers to the fact that virtualization software resides on the hard disk of the hardware, where the OS is usually installed. Bare metal hypervisor is the most commonly deployed type of hypervisor. | **Hosted Hypervisor**  These run on the OS of the host machine.  These hypervisors have higher latency than bare metal because request between hypervisor and the hardware pass through the extra layer of OS. |

|  |  |
| --- | --- |
|  | **Pros**   1. Multiple VM’s on Single Machine 2. Consolidate apps into single physical machine 3. Faster Server Providing   **Cons**   1. Requires separate OS for each VM. Each OS will need licenses (cost), even RedHat Linux charges for the Enterprise Linux 2. Atleast 30 % of the resources will be utilized for 3. CapEx Cost for setup virtualization technologies in its DC. 4. OpEx (Operation Costs) for operating right from manpower, Maintaining DC etc. |

|  |  |
| --- | --- |
| **Containers** | 1. Containers virtualizes at the OS level but VM’s virtualizes at Hardware level. 2. Containers are very light weight as it contains only the application to be run and the required libraries for that application. 3. It boots up in matter of seconds 4. Take less disk and memory compared to VM’s. |

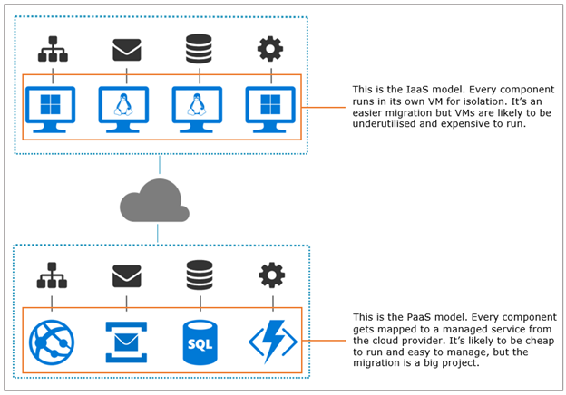
**VM vs Containers**

1. Containers does not replace VM
2. VM Virtualizes Hardware
3. Containers Virtualizes at OS level.
4. Containers can run inside VM’s.

**Migrating Apps to Cloud**

Apps can be migrated to two different types of cloud IAAS or PAAS.

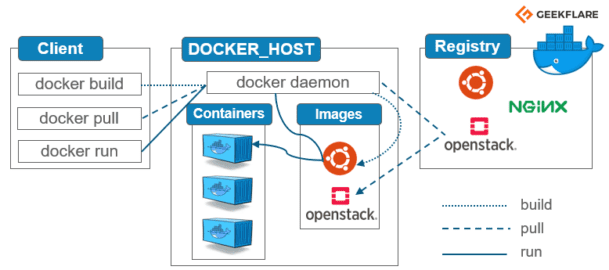
|  |  |
| --- | --- |
| IAAS  You spin up a virtual machine for each component. With this you get portability across clouds.  Disadvantage is higher running costs. | PAAS  If you use PAAS like Azure or AWS. It won’t be a as-it-is migration to PAAS, every component gets mapped to cloud provider service like AWS EC2, RDS etc. which means you have to adapt to cloud specific architecture and it’s a lock-in to the vendor cloud.  Advantage of this is it will get you lower running costs.  Disadvantage is vendor cloud lock-in and big migration effort. |



Third option is to You migrate your application to a container, and then you can run the whole application in containers using Azure Kubernetes Service or Amazon's Elastic Container Service, or on your own Docker cluster in the datacenter.

First step would be to move the entire applications as it is to the container. This will be a monolith application inside a container. Plan for further segregation of application to smaller microservices.

**Docker Architecture**



Docker is a platform that uses OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels.

Docker is a client-server application. You need both client and server for running a docker application on your computer. The client-server combination is called Docker Engine.

Docker Engine is an application which follows client-server architecture. It is installed on the host machine. There are three components in the Docker Engine:

* **Server**: It is the docker daemon called dockerd. It can create images, containers, networks etc and manage them.
* **Rest API**: It is used to instruct docker daemon what to do.
* **Command Line Interface (CLI)**: It is a client which is used to enter docker commands.

**Docker Client**

Docker users can interact with Docker through a client. When any docker commands runs, the client sends them to dockerd daemon, which carries them out.

Docker Daemon exposes API to the client through REST. Client can call operations on docker using REST API.

Docker client can communicate with more than one daemon.

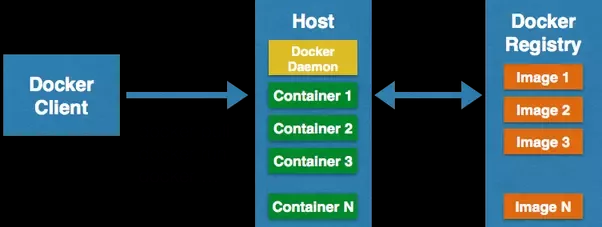
**Docker Registries**

It is the location where the Docker images are stored. It can be a public docker registry or a private docker registry. Docker Hub is the default place of docker images, its stores’ public registry. You can also create and run your own private registry.

When you execute docker pull or docker run commands, the required docker image is pulled from the configured registry. When you execute docker push command, the docker image is stored on the configured registry.

**DockerHost**

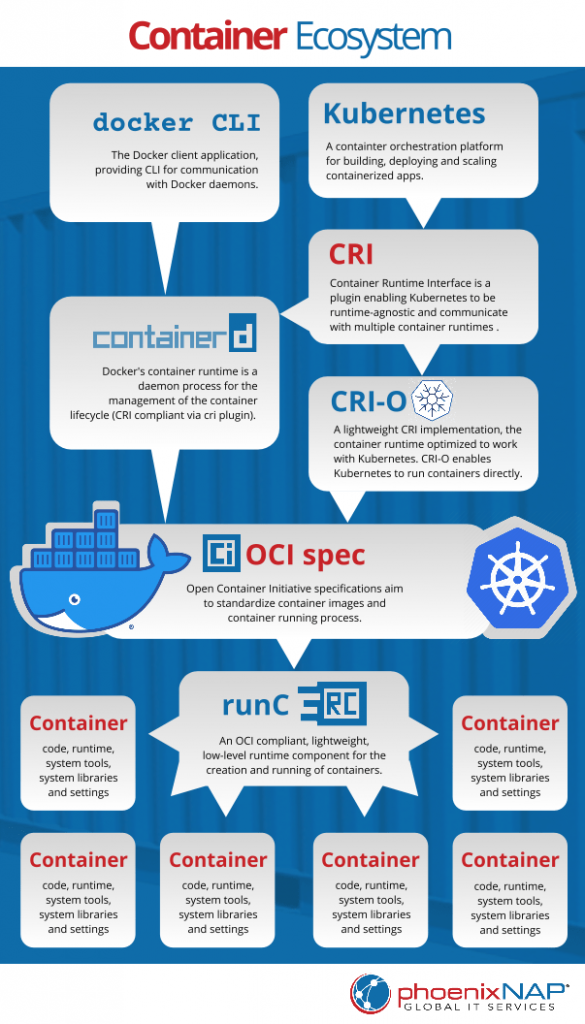
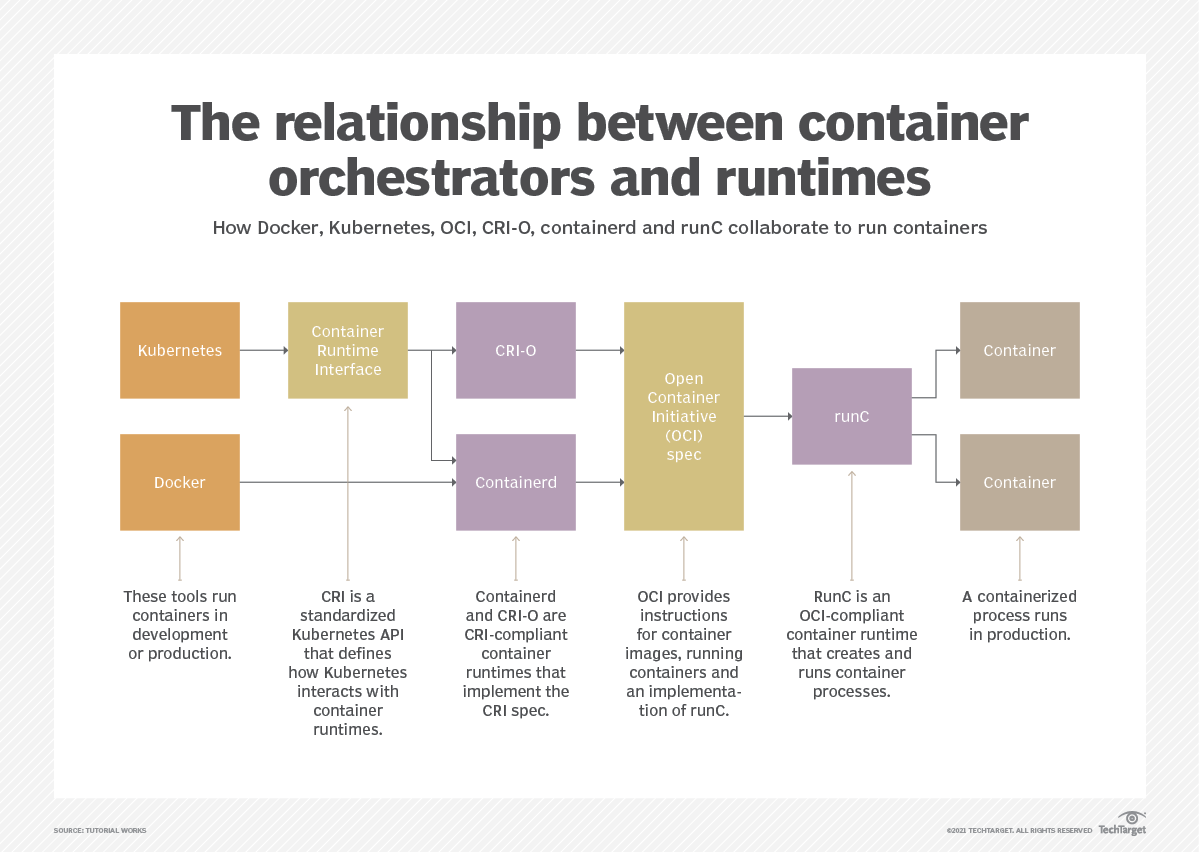
Docker host is the server (machine) on which Docker daemon runs. Docker containers are running instances of Docker images.



**Understanding Continerd, runc, CRI-O and dockerd**

There are two big standards around containers:

1. **Open Container Initiative (OCI):** It’s a specification for the container image format, runtime and distribution.
2. **Container Runtime Interface (CRI) in Kubernetes**: It’s a standardized Kubernetes API that defines how Kubernetes interacts with runtime.



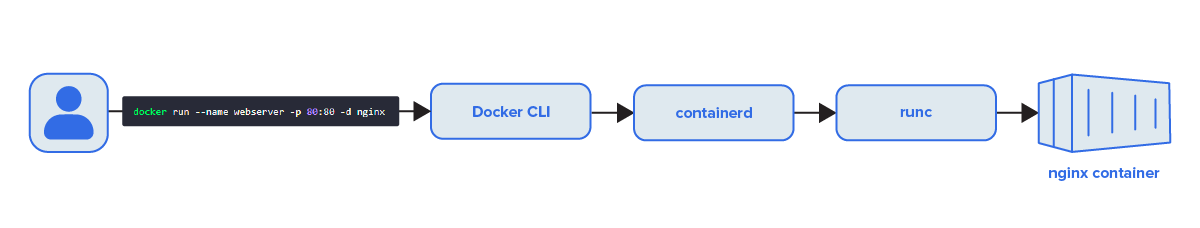
**All these Docker Cli, Daemon, Containerd, runc are entirely separate programs/services.**

**Action flow when below command is run**

docker run --name webserver -p 80:80 -d myapp

1. Docker CLI understands what we want to do and then sends instructions to containerd.
2. containerd does its magic and downloads the Nginx image if it's not available.
3. Next, containerd tells runc to start this container
4. And we finally get our result: nginx running in a little isolated container box.

It's easy to see that the Docker CLI is not necessarily required for this action. This means we don't really need Docker with all of its parts, such as the Docker CLI, Docker Daemon, and some of its other bits and pieces. However, we still need containerd and runc to start a container.



So why not tell Containerd directly about our intention? If we skip running Docker CLI, and the Docker Daemon, at least we would use less memory on our system, right? It would be more efficient, that is true. It's one of the reasons why Kubernetes removed Docker and opted to use C**ontainerd** directly. But that's a tradeoff that is only useful on servers running hundreds of containers.

1. **Docker Cli (Client)**

**/usr/bin/docker**

Docker users can interact with Docker through a client. When any docker commands runs, the client sends them to dockerd daemon, which carries them out.

Docker Daemon exposes API to the client through REST. Client can call operations on docker using REST API.

Docker client can communicate with more than one daemon.

1. **Docker Daemon**

**/usr/bin/dockerd**

Docker daemon is a separate service (program) that always runs in the background, waiting for the instructions.

docker run --name webserver -p 80:80 -d myapp

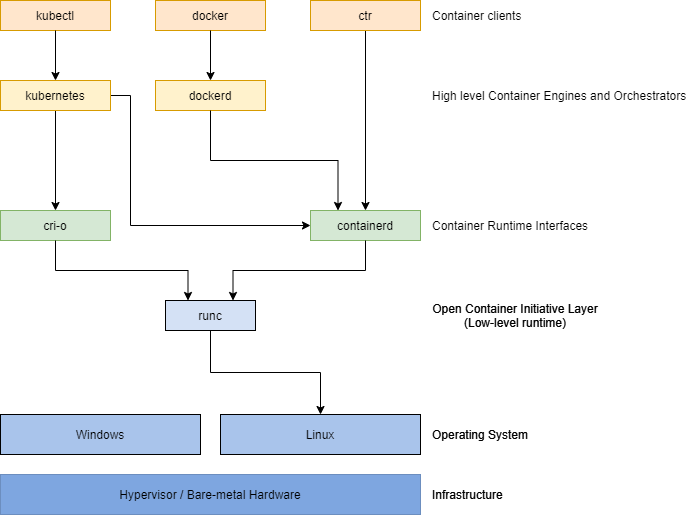
When the user enters above command on docker CLI, docker CLI accepts the command and it will call the api on docker daemon. The daemon in turn calls another app called container runtime i.e., Containerd.

1. **Containerd**

Containerd is a high-level container runtime. Containerd is another separate service that is kind of container manager. It takes care of things such as

1. Pull and push of container images.
2. Starting, Stopping and restarting of Containers. Controls the container life cycle.
3. Setting up networking between these containers to the outside world.
4. Managing data and files stored inside these containers.

Containerd came from docker and it’s made CRI compliant through its cri plugin. You can call ctr command-line tool to call containerd directly. For example, you can use ctr to pull and push container images from oci-compliant repository like DockerHub.



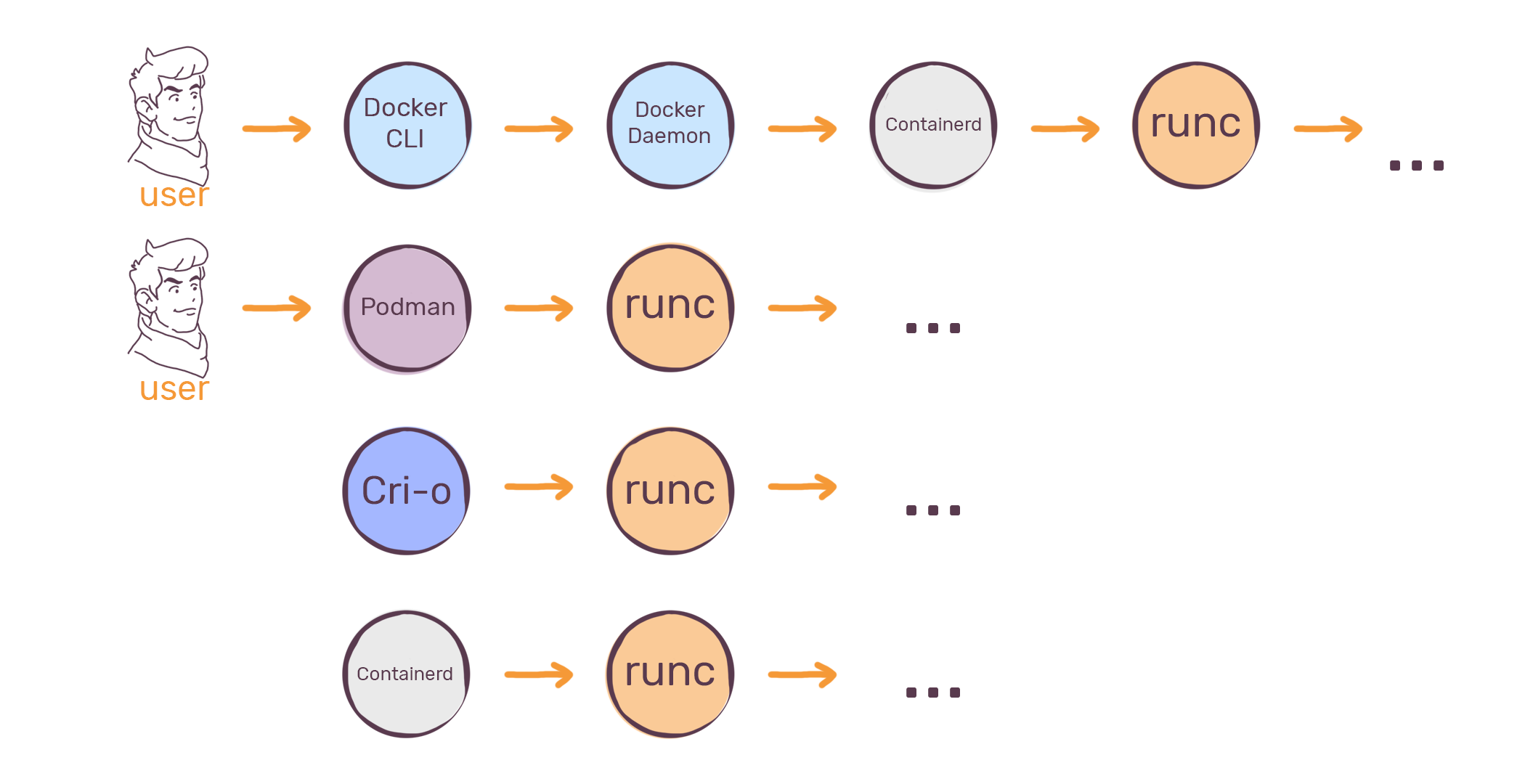
1. **CRI-O:**

CRI-O is a high-level container runtime similar to containerd. It is a light weight implementation of CRI. It was specifically built for Kubernetes by RedHat. CRI-O is a lean and efficient software that allows users run containers directly from Kubernetes.

1. **runc:**

runc is a low-level container runtime that provides an interface for creating and running containers. In other words, runc provides basic functionality for creating and running containers. Usually, runc will be called by high-level container runtimes like containerd / cri-o.

One of the key features of Runc is its ability to isolate containers using Linux namespaces. Namespaces allow different containers to have their own isolated view of system resources, such as process IDs, network interfaces, and file systems. This isolation ensures that containers remain independent and do not interfere with each other, providing a secure and reliable runtime environment.



**Docker Objects**

1. Images
2. Containers
3. Volumes
4. Networks

**Images:**

Docker images are read-only binary templates with instructions to create a docker container. Docker images are built and uploaded in to docker registries (public/private). Docker images are pulled from registries and run to create containers.

You can create a Docker image in one of two ways:

**Dockerfile Method:**

A Docker-File is a file which contains instructions/commands to package up an application. Create an image from the docker-file and push it to docker registry. Docker-Client will execute create command on docker-daemon to create docker-image from the docker-file.

**Interactive Method:** By running a container from an existing Docker image, manually changing that container environment through a series of live steps and saving the resulting state as a new image.

A Docker image is a logical collection of image layers. Layers are the files which are physically stored in the Docker engine’s cache.  Each instruction type in the Dockerfile will create an image layer.

Here is why that’s important: image layers can be shared between different images and different containers.

If you have lots of containers all running NodeJS apps, they will all share the same set of image layers that contain the NodeJS runtime.

Suppose you have 3 images using NodeJS images as base image. Assume NodeJS image occupies 75MB of space. If you run "docker image ls" command it will logically show each is occupying 75 x 3 = 225MB of space. But in reality it's not going to occupy that much of space. "docker image ls" will show how much each image will occupy if they exist independently.

"docker system df" command will show you how much disk space actually each image is occupying. For above example it shows it is occupying 75MB on disk. The reason all images are sharing the same base image. Since images can be shared they can't be edited, otherwise a change in one image would cascade to all the other images that share the changed layer. Docker enforces that by making image layers read-only. Once you create a layer by building an image, that layer can be shared by other images, but it can’t be changed. You can take advantage of that to make your Docker images smaller and your builds faster, by optimizing your Dockerfile.

Every Dockerfile instruction results in an image layer, but if the instruction does not change between builds, and the content going into the instruction is the same, then Docker knows it can use the previous layer in the cache. That saves executing the Dockerfile instruction again and generating a duplicate layer. The input is the same, so the output will be the same and Docker can just use what is already there in the cache.

Docker calculates if the input has a match in the cache by generating a hash, which is like a digital fingerprint representing the input. The hash is made from the Dockerfile instruction and the contents of any files being copied. If there is no match for the hash in the existing image layers, then Docker executes the instruction – and that breaks the cache. As soon as the cache is broken, Docker executes all the instructions that follow, even if they have not changed.

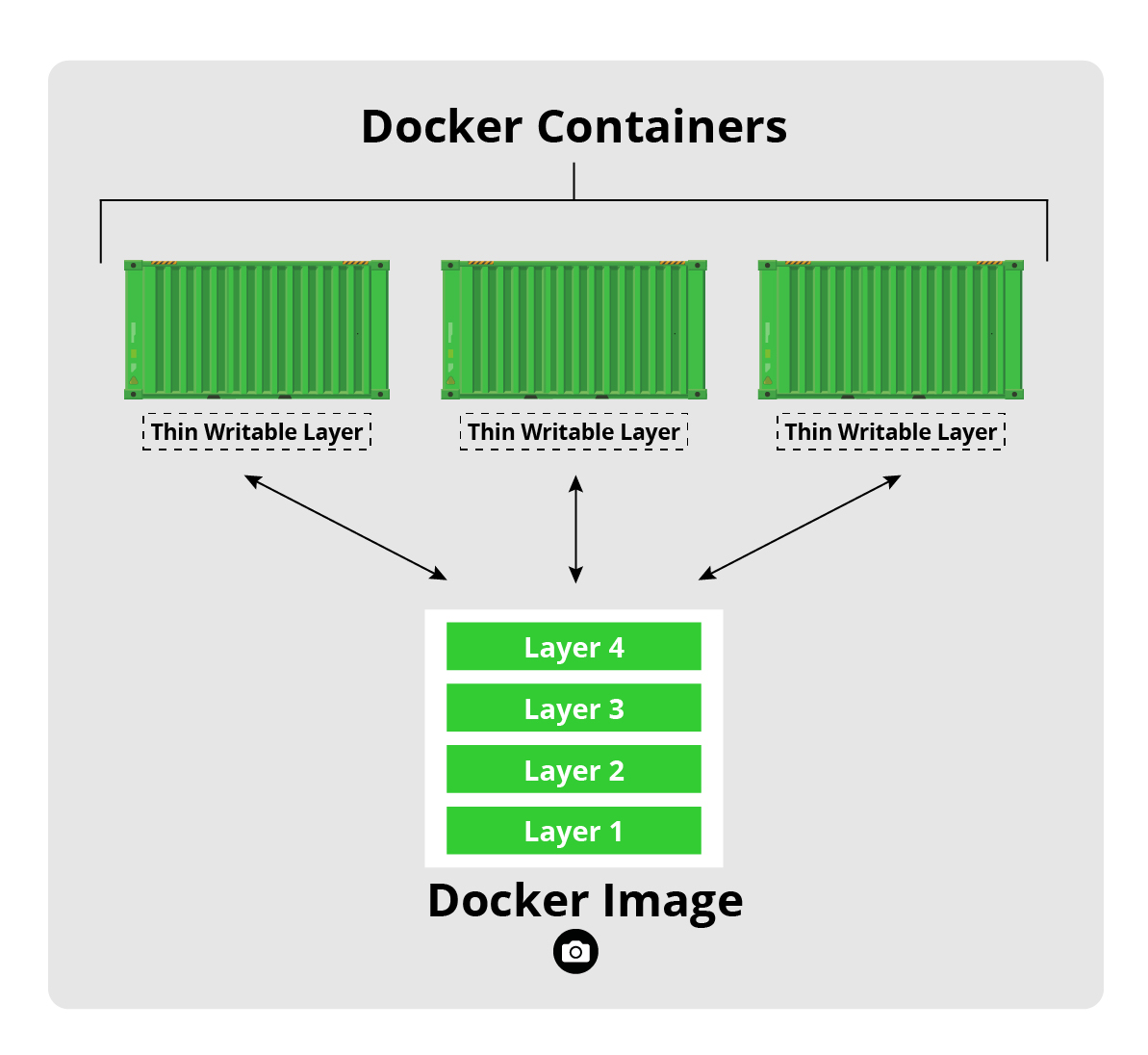
Any Dockerfile you write should be optimized so that the instructions are ordered by how frequently they change – with instructions that are unlikely to change at the start of the Dockerfile, and instructions most likely to change at the end. The goal is that most builds will only need to execute the last instruction, using the cache for everything else. That saves time, disk space and network bandwidth when you start sharing your images.

|  |  |
| --- | --- |
| FROM diamol/node    WORKDIR /web-ping  COPY app.js .  ENV TARGET="blog.sixeyed.com"  ENV METHOD="HEAD"  ENV INTERVAL="3000"    CMD ["node", "/web-ping/app.js"] | Sample Dockerfile script  App.js is the file that will be frequently between the builds. Developer will change the code of this file frequently.  All image layers after app.js will be built every time even though there is no change for the instructions after app.js. |
| FROM diamol/node    CMD ["node", "/web-ping/app.js"]    ENV TARGET="blog.sixeyed.com" \  METHOD="HEAD" \  INTERVAL="3000"    WORKDIR /web-ping  COPY app.js . | Above script can be optimized by ordering the instructions based on the order in which they will change frequently.  COPY app.js command should be placed as a last instruction as it is the only file that gets changed. |

**Container Layer**

Each time Docker launches a container from an image, it adds a thin writable layer, known as the container layer, which stores all changes to the container throughout its runtime.

As this layer is the only difference between a live operational container and the source Docker image itself, any number of like-for-like containers can potentially share access to the same underlying image while maintaining their own individual state.



**Base images** are images that have no parent image, usually images with an OS like ubuntu or debian.

**Child images** are images that build on base images and add additional functionality.

Docker images are read-only templates from which Docker containers are launched. Each image consists of a series of layers. Docker makes use of a union filesystem to combine these layers into a single image. Union filesystems allow files and directories of separate filesystems, known as branches, to be transparently overlaid, forming a single coherent filesystem.

Docker images are built on Docker Engine, distributed using the registry, and run as containers. Multiple versions of an image may be stored in the registry using the format image-name:tag. image-name is the name of the image and tag is a version assigned to the image by the user. By default, the tag value is latest and typically refers to the latest release of the image. For example, jboss/wildfly:latest is the image name for the WildFly’s latest release of the application server. A previous version of the WildFly Docker container can be started with the image jboss/wildfly:9.0.0.Final. Once the image is downloaded from the registry, multiple instances of the container can be started easily using the run command.