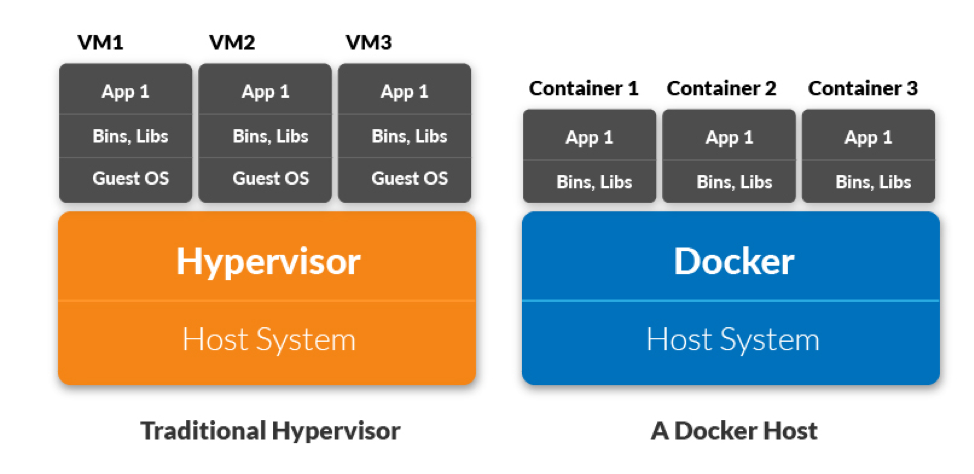
# Introduction

## What Is Docker?

When developers are writing code, they are working on their own local development environment. When they are ready to move that code to production, this is where problems arise. The code that worked perfectly on their machine might not work in production. The reasons for this are varied: different operating systems, different dependencies, different libraries, etc.

To solve this problem, back to the old days, we had to set up multiple physical PCs or install multiple virtual machines to run and test software on multiple OSs. This manner is stable, but it has a very big problem – heavy. Docker is born to change that.

Docker is a tool that allows developers, sys-admins, etc. to **deploy applications in a sandbox** (called *containers*). The key benefit is that users can package an application with all of its dependencies into a standardized unit which can be **run everywhere**.



## What Is Docker Image?

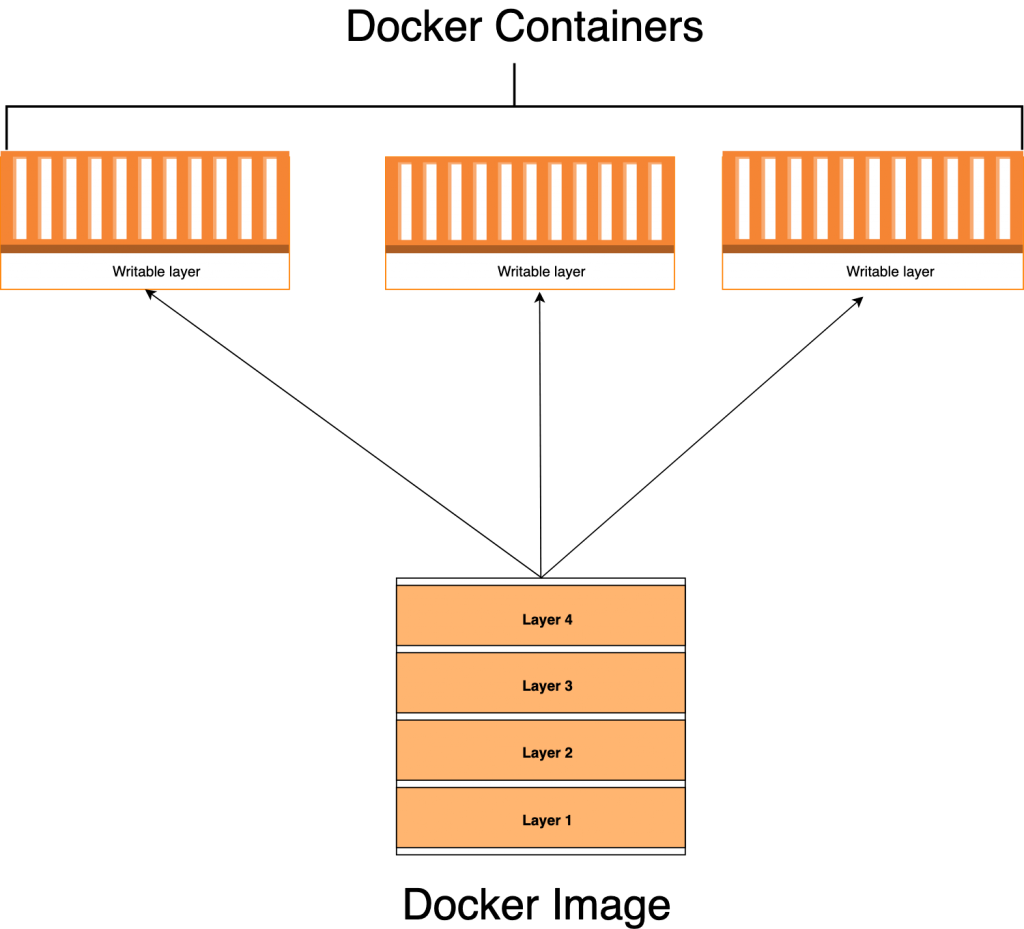
A Docker image is run on a [container](https://www.docker.com/resources/what-container) by using the [Docker Engine](https://www.docker.com/products/container-runtime). Docker images have many benefits such as portability (applicable to multiple environments and platforms), customizable, and highly scalable.

Unlike traditional virtual machines, the Docker engine runs on a layer between the host OS kernel and the isolated application services that are being containerized.

## What Is Docker Container?

Docker containers are instances of Docker images. That means multiple containers can run on the same image independently.

When you create a Docker container, you’re adding a writable layer on top of the Docker image:



* **Containers vs Virtual Machines:**

The industry standard is to use Virtual Machines to run software applications inside a guest OS, which runs on virtual hardware powered by the server’s host OS.

VMs are great at providing full process isolation for applications: there are very few ways a problem in the host operating system can affect the software running in the guest operating system, and vice-versa. But this isolation comes at great cost — the computational overhead spent virtualizing hardware for a guest OS to use is significant.

Containers take a different approach: by leveraging the low-level mechanics of the host OS, containers provide most of the isolation of virtual machines at a fraction of the computing power.

But note that once a Docker container is closed, the data inside it is no longer available and the new running Docker image will create a new container.

# Installation

## On Windows

<https://docs.docker.com/docker-for-windows/install/>

**Notes**:

* You might be required to reboot your computer once or twice during the installation process.
* The installer will automatically enable Hyper-V and support for Windows Containers if they are not already turned on.

**Folder tree**:

C:\PROGRAMDATA\DOCKERDESKTOP

│ service.0.txt

│ service.txt

│

├───tmp-d4w

│ host.docker.internal

│

├───version-bin

└───vm-data

DockerDesktop.vhdx

C:\PROGRAMDATA\DOCKER

└───cli-plugins

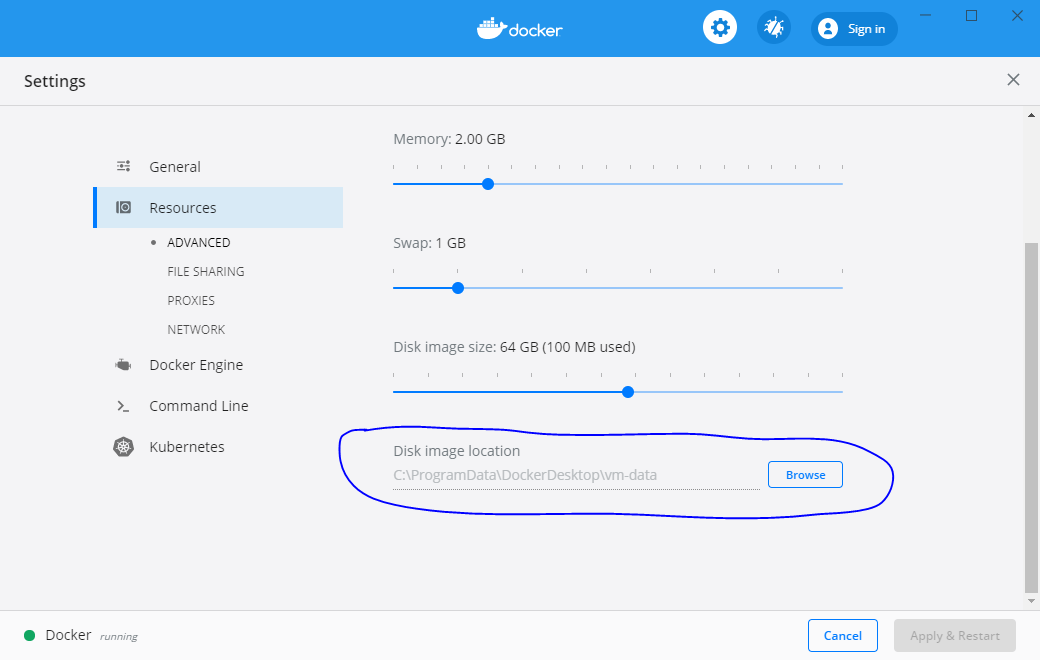
docker-app.exe

docker-buildx.exe

docker-mutagen.exe

### Configuration Before Start

**Change Docker image location in Windows**



<https://www.pbworks.net/change-docker-images-location-in-windows/#:~:text=Click%20%E2%80%9CApply%E2%80%9D%2C%20docker%20will,bottom%20click%20on%20%E2%80%9CRestart%E2%80%9D.>

## On Ubuntu

<https://phoenixnap.com/kb/how-to-install-docker-on-ubuntu-18-04>

## Q&A

**Docker Toolbox or Docker for Window**

<https://nickjanetakis.com/blog/should-you-use-the-docker-toolbox-or-docker-for-mac-windows>

**Docker Quickstart Terminal**

<https://medium.com/@peorth/using-docker-with-virtualbox-and-windows-10-b351e7a34adc>

# Command Lists

## Images

### docker pull

**Syntax**: docker pull [OPTIONS] NAME[:TAG|@DIGEST]

**Usage**: Pull an image or a repository from a registry

### docker images

**Syntax**: docker images [OPTIONS] [REPOSITORY[:TAG]]

**Usage**: List images

**Common Options**:

|  |  |
| --- | --- |
| **Option** | **Usage** |
| --a, --all | Show all images (default hides intermediate images) |

### docker search

**Syntax**: docker search [OPTIONS] TERM

**Usage**: Search the Docker Hub for images

### docker build

**Syntax**: docker build [OPTIONS] PATH | URL | -

**Usage**: Build an image from a Dockerfile

**Common Options**:

|  |  |
| --- | --- |
| **Option** | **Usage** |
| -t, --tag list | Add a tag to the built image (in the 'name:tag' format) |
| --rm | Remove intermediate containers after a successful build (default --rm true) |
| --force-rm | Always remove intermediate containers, even if unsuccessful building |
| --no-cache | Do not use cache when building the image |

**Tip:**

* The --no-cache option is used to **rebuild** an existing image.

**Note**:

* If you build an image without tagging it, the image will appear on the list of dangling images because it has no association with a tagged image. You can avoid this situation by **always providing a tag when you build**. If you forget, you can retroactively tag the built image with the docker tag command.

### docker commit

**Syntax**: docker commit <image-id> <image-name>

**Usage**: Create a new image from a container's changes

### docker push

**Syntax**: docker push

**Usage**: Push an image or a repository to a registry

### docker inspect

**Syntax**: docker inspect [OPTIONS] NAME|ID [NAME|ID...]

**Usage**: Display detailed information on a Docker image

### docker rmi

**Syntax**: docker rmi [OPTIONS] IMAGE [IMAGE...]

**Usage**: Remove one or more images

**Common Options**:

|  |  |
| --- | --- |
| **Option** | **Usage** |
| -f, --force | Force removal of the image |

## Containers

### docker run

**Syntax**:

docker run [OPTIONS] IMAGE-ID [COMMAND] [ARG...]

# or

docker run [OPTIONS] REPOSITORY:TAG [COMMAND] [ARG...]

**Usage**: Run a command in a new container.

**Common Options**:

|  |  |
| --- | --- |
| **Option** | **Usage** |
| -d, --detach | Run container in background and print container ID |
| -a, --attach list | Attach to STDIN, STDOUT or STDERR |
| -i, --interactive | Keep STDIN open even if not attached |
| -t, --tty | Allocate a pseudo-TTY |
| --rm | Automatically remove the container when it exits |
| --name <string> | Assign a name to the container  Now the container can be called by either name or ID |

**Commands**:

If the user specifies the command to docker run, they will override the default specified in CMD in Dockerfile.

For example, the official Ubuntu Dockerfile has:

CMD ["/bin/bash"]

So, for the Ubuntu case, docker run ... ubuntu /bin/bash is same as docker run ... ubuntu. And what you will see is:

host@host-name: $ run -it ubuntu /bin/bash

root@container-id: # # Now you're directed to console prompt to type other commands

For other images, such as the official elasticsearch image, the CMD executes a script to start elasticsearch server. So for this case, after executing the run command, you won't see the console prompt, but the server will start:

host@host-name: $ run ... elasticsearch

elasticsearch server logs ... # The server starts

### docker create

**Syntax**: docker create

**Usage**: Create a new container

### docker ps

**Syntax**: docker ps [OPTIONS]

**Usage**: List containers

**Common Options**:

|  |  |
| --- | --- |
| **Option** | **Usage** |
| --a, --all | Show all containers (default shows just running) |

### docker start

**Syntax**: docker start [OPTIONS] CONTAINER-ID-OR-NAME [CONTAINER...]

**Usage**: Start one or more stopped containers

**Common Options**:

|  |  |
| --- | --- |
| **Option** | **Usage** |
| -d, --detach | Attach STDOUT/STDERR and forward signals |
| -i, --interactive | Attach container's STDIN |

### docker attach

**Syntax**: docker attach [OPTIONS] CONTAINER-ID-OR-NAME

**Usage**: Attach local standard input, output, and error streams to a running container

### docker stop

**Syntax**: docker stop [OPTIONS] CONTAINER-ID-OR-NAME [CONTAINER...]

**Usage**: Stop one or more running containers

### docker restart

**Syntax**: docker restart [OPTIONS] CONTAINER [CONTAINER...]

**Usage**: Restart one or more containers

### docker exec

**Syntax**: docker exec [OPTIONS] CONTAINER-ID-OR-NAME COMMAND [ARG...]

**Usage**: Run a command in a running container

**Common Options**:

|  |  |
| --- | --- |
| **Option** | **Usage** |
| -d, --detach | Detached mode: run command in the background |
| -i, --interactive | Keep STDIN open even if not attached |
| -t, --tty | Allocate a pseudo-TTY |

**Use Cases**:

* To run a running container in another terminal (in other words, launch several sessions connected to the same container), open a new terminal and run:

docker exec -it CONTAINER-ID-OR-NAME bash

### docker rm

**Syntax**: docker rm [OPTIONS] CONTAINER [CONTAINER...]

**Usage**: Remove one or more containers

**Common Options**:

|  |  |
| --- | --- |
| **Option** | **Usage** |
| -f, --force | Force the removal of a running container (uses SIGKILL) |

## Others

### docker volume

**Syntax**: docker volume COMMAND

**Usage**: Manage volumes

**Commands**:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| create | Create a volume |
| inspect | Display detailed information on one or more volumes |
| ls | List volumes |
| prune | Remove all unused local volumes |
| rm | Remove one or more volumes |

### Others

cp Copy files/folders between a container and the local filesystem

diff Inspect changes to files or directories on a container's filesystem

events Get real time events from the server

export Export a container's filesystem as a tar archive

history Show the history of an image

import Import the contents from a tarball to create a filesystem image

info Display system-wide information

kill Kill one or more running containers

load Load an image from a tar archive or STDIN

login Log in to a Docker registry

logout Log out from a Docker registry

logs Fetch the logs of a container

pause Pause all processes within one or more containers

port List port mappings or a specific mapping for the container

rename Rename a container

rmi Remove one or more images

save Save one or more images to a tar archive (streamed to STDOUT by default)

stats Display a live stream of container(s) resource usage statistics

tag Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE

top Display the running processes of a container

unpause Unpause all processes within one or more containers

update Update configuration of one or more containers

version Show the Docker version information

wait Block until one or more containers stop, then print their exit codes

# Example: Create Linux Container on Windows 10

Follow steps below:

## Start Docker

On Windows: Start Docker Desktop

On Linux:

Start: sudo systemctl start docker

Stop: sudo systemctl stop docker

## Pull a Base Image

Before you can create a Linux container, you need to pull a base image from Docker’s repository using the docker pull <image-name> command. Open PowerShell or CMD and run:

# Pull the latest Ubuntu base image

docker pull ubuntu

# OR pull a specific version of Ubuntu, e.g. 18.04:

# docker pull ubuntu:18.04 # repo name is "ubuntu", tag is "18.04"

Once done, check the result by listing all available images on the local computer (with information about image size, image ID, and tags):

docker images

**Tip:**

* To search the repository for Ubuntu images before pulling:

docker search ubuntu

## Create a New Container from the Base Image

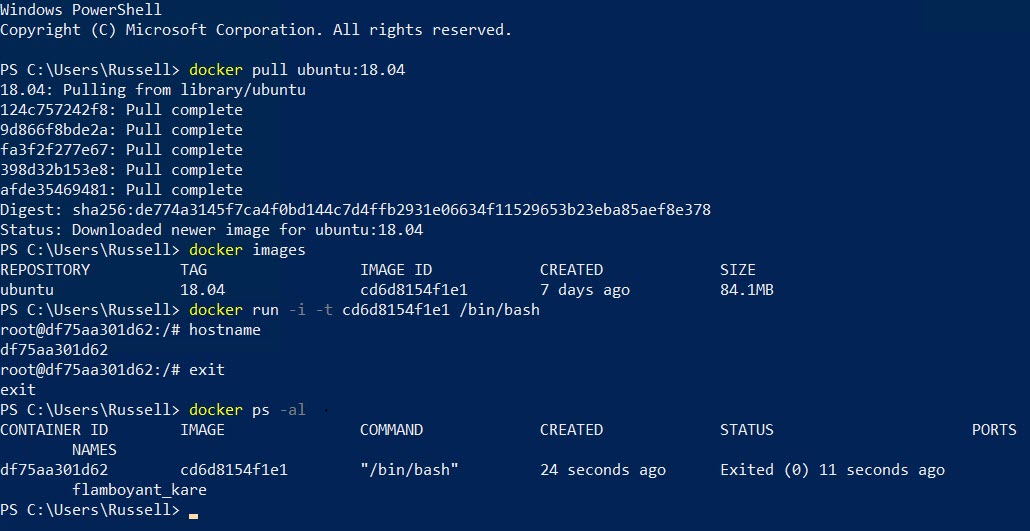
To create a new Linux container, run:

docker run -i -t <image-id> /bin/bash

The -i (interactive) and -t (terminal) parameters allow the bash process to start in the container, attaches the console to the process’s standard input, output, and standard error, and allocates a pseudo-tty text-only console.

Once the container has been created, you’ll be presented with a bash prompt. If you want, you can:

* Type hostname and press enter to see the container’s Linux hostname.
* Type exit and press enter stop the container.



*You might not know!*

docker run is similar to docker create  but the latter creates a writeable container layer over the image and prepares it for running the command you specify. The container is not started. It is useful in scenarios where you want to set up a container in advance and have it ready to go using docker start.

The command docker run is used to only START a container for the very first time. To run an existing container what you need is docker start $container-name.

## Start the Created Container

First, list all the containers on the local device (with container ID, image ID or image name, command, status, port, etc.) with:

docker ps -al

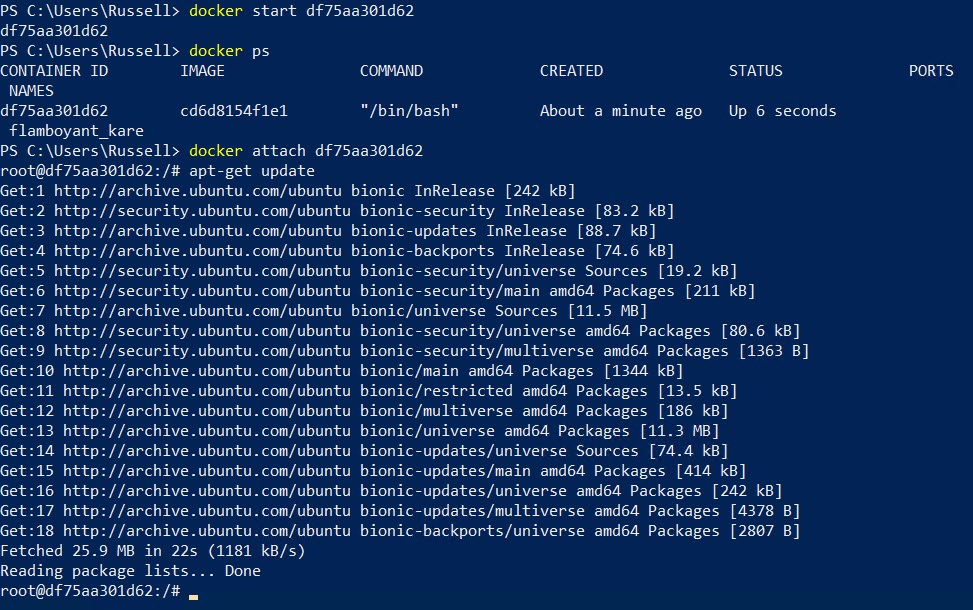
The -a (or --all) and -l (--latest) options shows all latest containers. Without -a, only running containers are shown. And without -l, later containers are shown.

Start the container with:

docker start <container-id>

Now connect to the container’s terminal to actually work on the container with:

docker attach <container-id>



**Tip:**

* To run a running container in another terminal (in other words, launch several sessions connected to the same container), open a new terminal and run:

docker exec -it <container-id> bash

## Create a New Image from a Container

Let’s make a minor change to our Ubuntu container, and then create a new local image from it.

In this example, let's install the Vim editor in the container before stop it:

apt-get update

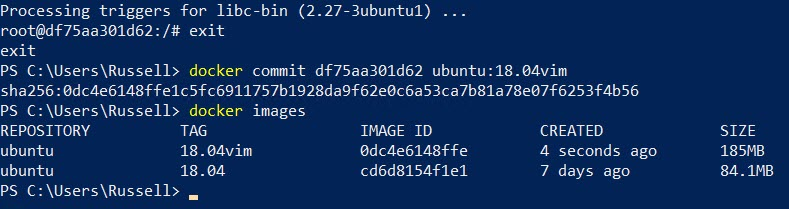
apt-get install vim

exit

Finally, use the docker commit <image-id> <image-name> command to create a new image from the container. You can overwrite an existing image or create a new image. But in this case, we'll create a new image with a different name. In this example, let's add a tag 18.04vim to help identify it.

docker commit <container-id> ubuntu:18.04vim

Run docker images again and you'll see the new image in the list. You can now use this image to create a Linux container that includes the vim text editor:



# Creating Docker Image from Dockerfile

## What Is Dockerfile?

Dockerfiles are simply text files that **contain build instructions (commands) used by Docker to create a new container image based on an existing image**. It is written in a special syntax which we have to learn to use.

## What Is Dockerignore?

The thumb rule is to always **keep your Docker image as lean as possible**. This means packaging only what your applications need to run.

In reality, source code usually contains other files and directories, such as .git, .vscode, README.md, etc. Those are essential for our development workflow, but won’t stop our app from running. It’s a best practice not to have them in your image - that’s what .dockerignore is for. We use it to prevent such files and directories from making their way into our build.

Example of a dockerignore:

.git

.gitignore

node\_modules

npm-debug.log

Dockerfile\*

docker-compose\*

README.md

LICENSE

.vscode

## Example

In this example, we'' create Docker image for Elasticsearch from Ubuntu base image. Follow steps below:

1. Create a Dockerfile

Create a new file named "Dockerfile" with your favorite text editor. "Dockerfile" is the default name but you can use any filename you want (and even have multiple dockerfiles in the same folder).

2. Import the base image

A base image (or parent image) is where your image is based. It’s your starting point. It could be a Linux distribution, Widows, MySQL, Redis, etc.

Base images don’t just fall from the sky. They’re created - and you too can create one from scratch. But there are many base images out there that you can use, so you don’t need to create one in most cases.

To add the base image to Dockerfile, use the FROM command followed by the base image name and tag:

FROM ubuntu:8.04

3.

## Dockerfile Instructions

### CMD

Note: If the Dockerfile defines ENTRYPOINT, then regarding what CMD command is. The ENTRYPOINT always runs first.

### ENTRYPOINT

**CMD vs ENTRYPOINT:**

Both CMD and ENTRYPOINT define what command gets executed when running a container. There are few rules that describe their co-operation.

* Dockerfile should specify at least one of CMD or ENTRYPOINT commands.
* ENTRYPOINT should be defined when using the container as an executable.
* CMD should be used as a way of defining default arguments for an ENTRYPOINT command or for executing an ad-hoc command in a container.
* CMD will be overridden when running the container with alternative arguments.

The tables below show what command is executed for different ENTRYPOINT / CMD combinations:

-- No ENTRYPOINT

╔════════════════════════════╦═════════════════════════════╗

║ No CMD ║ error, not allowed ║

╟────────────────────────────╫─────────────────────────────╢

║ CMD ["exec\_cmd", "p1\_cmd"] ║ exec\_cmd p1\_cmd ║

╟────────────────────────────╫─────────────────────────────╢

║ CMD ["p1\_cmd", "p2\_cmd"] ║ p1\_cmd p2\_cmd ║

╟────────────────────────────╫─────────────────────────────╢

║ CMD exec\_cmd p1\_cmd ║ /bin/sh -c exec\_cmd p1\_cmd ║

╚════════════════════════════╩═════════════════════════════╝

-- ENTRYPOINT exec\_entry p1\_entry

╔════════════════════════════╦══════════════════════════════════╗

║ No CMD ║ /bin/sh -c exec\_entry p1\_entry ║

╟────────────────────────────╫──────────────────────────────────╢

║ CMD ["exec\_cmd", "p1\_cmd"] ║ /bin/sh -c exec\_entry p1\_entry ║

╟────────────────────────────╫──────────────────────────────────╢

║ CMD ["p1\_cmd", "p2\_cmd"] ║ /bin/sh -c exec\_entry p1\_entry ║

╟────────────────────────────╫──────────────────────────────────╢

║ CMD exec\_cmd p1\_cmd ║ /bin/sh -c exec\_entry p1\_entry ║

╚════════════════════════════╩══════════════════════════════════╝

-- ENTRYPOINT ["exec\_entry", "p1\_entry"]

╔════════════════════════════╦═════════════════════════════════════════════════╗

║ No CMD ║ exec\_entry p1\_entry ║

╟────────────────────────────╫─────────────────────────────────────────────────╢

║ CMD ["exec\_cmd", "p1\_cmd"] ║ exec\_entry p1\_entry exec\_cmd p1\_cmd ║

╟────────────────────────────╫─────────────────────────────────────────────────╢

║ CMD ["p1\_cmd", "p2\_cmd"] ║ exec\_entry p1\_entry p1\_cmd p2\_cmd ║

╟────────────────────────────╫─────────────────────────────────────────────────╢

║ CMD exec\_cmd p1\_cmd ║ exec\_entry p1\_entry /bin/sh -c exec\_cmd p1\_cmd ║

╚════════════════════════════╩═════════════════════════════════════════════════╝

# <none> Images

## Immediate Images

<https://www.projectatomic.io/blog/2015/07/what-are-docker-none-none-images/>

## Dangling Images

Docker images consist of multiple layers. Dangling images are layers that have no relationship to any tagged images. They **no longer serve a purpose and consume disk space**.

Locate all dangling images:

docker images -f dangling=true

Remove all dangling images:

docker image prune

# Dangling Volumes

Volumes exist independent from containers, **so when a container is removed, a volume is not automatically removed** at the same time. When a volume exists and is no longer connected to any containers, it’s called a dangling volume. We should remove it.

Locate all dangling volumes:

docker volume ls -f dangling=true

Remove all dangling volumes:

docker volume prune

# Docker and VirtualBox

# Docker and Kubernetes