# Docker

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

* Develop your application and its supporting components using containers.
* The container becomes the unit for distributing and testing your application.
* When you’re ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

## **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.

Diagram

Description automatically generated

### **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 registries**

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

### **Docker objects**

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

#### **Images**

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

#### **Containers**

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container’s network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

##### **Example docker run command**

The following command runs an ubuntu container, attaches interactively to your local command-line session, and runs /bin/bash.

$ docker run -i -t ubuntu /bin/bash

When you run this command, the following happens (assuming you are using the default registry configuration):

1. If you do not have the ubuntu image locally, Docker pulls it from your configured registry, as though you had run docker pull ubuntu manually.
2. Docker creates a new container, as though you had run a docker container create command manually.
3. Docker allocates a read-write filesystem to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.
4. Docker creates a network interface to connect the container to the default network, since you did not specify any networking options. This includes assigning an IP address to the container. By default, containers can connect to external networks using the host machine’s network connection.
5. Docker starts the container and executes /bin/bash. Because the container is running interactively and attached to your terminal (due to the -i and -t flags), you can provide input using your keyboard while the output is logged to your terminal.
6. When you type exit to terminate the /bin/bash command, the container stops but is not removed. You can start it again or remove it.

## Get started

Open a command prompt or bash window, and run the command:

docker run -d -p 80:80 docker/getting-started

You’ll notice a few flags being used. Here’s some more info on them:

* -d - run the container in detached mode (in the background)
* -p 80:80 - map port 80 of the host to port 80 in the container.  
  By default, when you create or run a container using docker create or docker run, it does not publish any of its ports to the outside world. To make a port available to services outside of Docker, or to Docker containers which are not connected to the container’s network, use the --publish or -p flag. This creates a firewall rule which maps a container port to a port on the Docker host to the outside world.
* docker/getting-started - the image to use

**Tip**

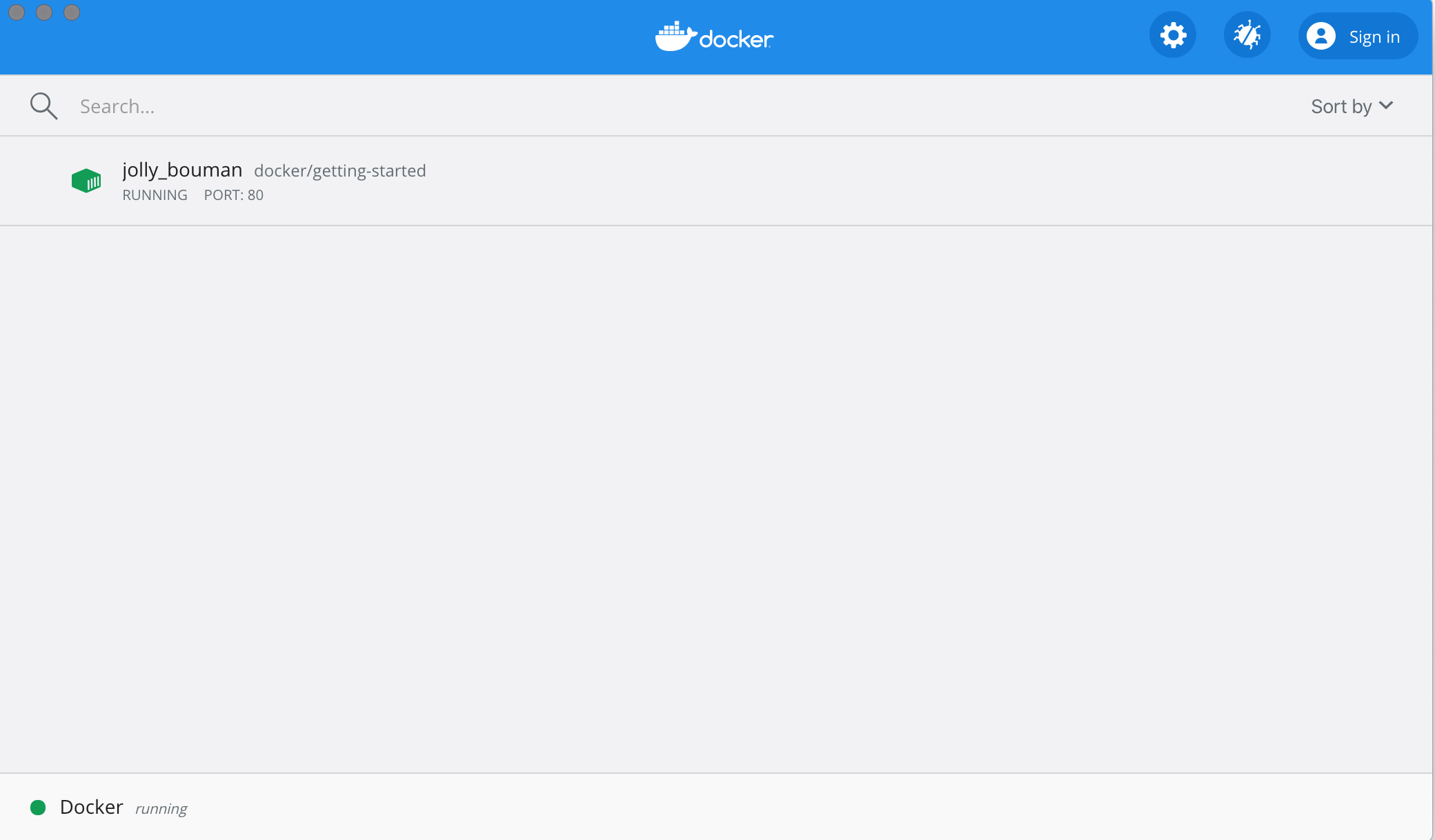
You can combine single character flags to shorten the full command. As an example, the command above could be written as:

docker run -dp 80:80 docker/getting-started

## **The Docker Dashboard**

Before going too far, we want to highlight the Docker Dashboard, which gives you a quick view of the containers running on your machine. The Docker Dashboard is available for Mac and Windows. It gives you quick access to container logs, lets you get a shell inside the container, and lets you easily manage container lifecycle (stop, remove, etc.).

To access the dashboard, follow the instructions for either [Mac](https://docs.docker.com/docker-for-mac/dashboard/) or [Windows](https://docs.docker.com/docker-for-windows/dashboard/). If you open the dashboard now, you will see this tutorial running! The container name (jolly\_bouman below) is a randomly created name. So, you’ll most likely have a different name.



## **What is a container?**

Now that you’ve run a container, what is a container? Simply put, a container is simply another process on your machine that has been isolated from all other processes on the host machine. That isolation leverages [kernel namespaces and cgroups](https://medium.com/@saschagrunert/demystifying-containers-part-i-kernel-space-2c53d6979504), features that have been in Linux for a long time. Docker has worked to make these capabilities approachable and easy to use.

## **What is a container image?**

When running a container, it uses an isolated filesystem. This custom filesystem is provided by a **container image**. Since the image contains the container’s filesystem, it must contain everything needed to run an application - all dependencies, configuration, scripts, binaries, etc. The image also contains other configuration for the container, such as environment variables, a default command to run, and other metadata.

We’ll dive deeper into images later on, covering topics such as layering, best practices, and more.

## Docker commands

## **Commmand Commonalities**

Here are a few things to know about Docker commands:

* Docker CLI management commands start with docker, then a space, then the management category, then a space, and then the command. For example, docker container stop stops a container.
* A command referring to a specific container or image requires the name or id of that container or image.

For example, docker container run my\_app is the command to build and run the container named my\_app.

## **Containers**

Use docker container my\_command

create — Create a container from an image.  
start — Start an existing container.  
run — Create a new container and start it.  
ls — List running containers.  
inspect — See lots of info about a container.  
logs — Print logs.  
stop — Gracefully stop running container.  
kill —Stop main process in container abruptly.  
rm— Delete a stopped container.

## **Images**

Use docker image my\_command

build — Build an image.  
push — Push an image to a remote registry.  
ls — List images.  
history — See intermediate image info.  
inspect — See lots of info about an image, including the layers.  
rm — Delete an image.

## **Misc**

docker version — List info about your Docker Client and Server versions.  
docker login — Log in to a Docker registry.  
docker system prune — Delete all unused containers, unused networks, and dangling images.

# Course: Docker for web developers

## Basic concepts

### Docker’s basic terms

* Docker runs an application such as MySQL in a single **container**. It’s a lightweight virtual machine-like package containing an OS, the application files, and all dependencies. Your web application will probably require several containers, e.g., your code (and language runtime), a database, a web server, etc.
* A container is launched from an **image**. In essence, it’s a container template that defines the OS, installation processes, settings, etc., in a **Dockerfile** configuration. Any number of containers can be started from the same image.

A picture containing shape

Description automatically generated

* Containers start in a clean (image) state and data are not permanently stored. You can mount Docker **volumes** or bind host folders to retain state between restarts.
* Containers are isolated from the host and other containers. You can define a **network** and open TCP/IP ports to permit communication.
* Each container is started with a single Docker command. **Docker Compose** is a utility that can launch multiple containers in one step using a docker-compose.yml configuration file.
* Optionally, **orchestration** tools such as Docker Swarm and Kubernetes can be used for container management and replication on production systems.

### Containers

#### VM vs Docker

(Note: VM software such as VMware and VirtualBox are known as **hypervisors**.)

* A virtual machine hypervisor emulates hardware so you can run a full Operating System.
* Docker emulates an Operating System so you can run isolated applications within their own file system.

Diagram

Description automatically generated

A picture containing text, electronics, calculator

Description automatically generated

It’s technically possible to run all your application’s dependencies in a single container, but there are no practical benefits for doing so, and management becomes more difficult. So always use separate containers for your app, database, etc.

#### Containers are isolated

Each container has its own specific IP, but it is available at localhost or 127.0.0.1. A TCP port must be exposed to communicate with the application it runs, e.g.,

* port 80 or 443 for a HTTP or HTTPS web servers
* 3306 for MySQL
* 27017 for MongoDB
* 80 for WordPress

For more than one container, the exposed ports should be different. For example, both the HTTP server and Wordpress have 80 as the default port. If you are using both together, only one can be exposed to 80, the other has to be exposed to some other port so bind exceptions of ports can be avoided.

Docker also allows you to access the container shell to enter terminal commands and expose further ports to attach debuggers and investigate problems.

#### Containers are stateless and disposable

Data written to the container’s file system is lost the moment it shuts down.

Any number of containers can be launched from the same base image. This makes scaling easy because every container instance is identical and disposable.

This may change the way you approach application development if you want to use Docker on production servers. Presume your application has a variable which counts the number of logged-in users. If it’s running in two containers, either could handle a login, so each would have a different user count.

Dockerized web applications should therefore avoid retaining state data in variables and local files. Your application can store data in a database such as redis, MySQL, or MongoDB so state persists between container instances.

📌 It may be impractical to deploy an existing application using Docker containers if it was developed in a non-stateless way from the start. However, you can still run the application in Docker containers during development.

Which begs the question: what if your database is running in a container?

It will also lose data when it restarts, so Docker offers **volumes and host folder bind mounts.**

# Docker commands

Format of a docker command: docker category subcommand

* The category can be **container, image, volume, network**, etc.
* The subcommand can be **create, rm, inspect, ls, start, stop,** etc
* A subcommand referring to a specific container, image, volume, network requires the name or id of that container, image, volume, network.

For example, docker container run my\_app is the command to build and run the container named my\_app.

**Containers**

Use docker container my\_command

create — Create a container from an image.  
start — Start an existing container.  
run — Create a new container and start it.  
ls — List running containers.  
inspect — See lots of info about a container.  
logs — Print logs.  
stop — Gracefully stop running container.  
kill —Stop main process in container abruptly.  
rm— Delete a stopped container.

**Images**

Use docker image my\_command

build — Build an image.  
push — Push an image to a remote registry.  
ls — List images.  
history — See intermediate image info.  
inspect — See lots of info about an image, including the layers.  
rm — Delete an image.

**Volume**

**Network**

**Misc**

docker version — List info about your Docker Client and Server versions.  
docker login — Log in to a Docker registry.  
docker system prune — Delete all unused containers, unused networks, and dangling images.

## Image

### docker pull image\_name

to pull an image from docker hub to your computer

## Container

### docker container create

create a container from an image

docker container create [options] image\_name

Argument image\_name: if the image\_name is not there then Docker tries to find and download it from Docker hub

Commonly used options:

-- name image\_name: the name for the new container.

If the image\_name is not in your local repos then Docker will try to download it from Docker hub.

The name should be of the format: “name:major.minor.patch”, for exp: mysql:8.0.25, or “name:latest”, which is equivalent to just “name”.

On Docker hub, if you don’t have the version, for exp “mysql”, it will be the latest version.

- it: this is combination of -- interactive, -i and -- tty, -t. This combination will keep a container running in the foreground (even after the application ends), and show an activity log.

--tty, -t: Docker official document reads “Allocate a pseudo-TTY” (TTY is a pseudo terminal).

In my opinion: a container is just CPU, RAM, HDD and it does not have a terminal (keyboard, monitor) yet. This option attaches a pseudo terminal to the container. When you run a container, the terminal you are using (i.e. cmd prompt) will become the terminal for the container if you add -- interactive, -i with this option.

-- interactive, -i: Docker official document reads “Keep STDIN open even if not attached”.

This will help you interact with a shell inside the docker.

-- publish [host\_ip:]<host\_port>:<container\_port>

- p [host\_ip:]<host\_port>:<container\_port>

([host\_ip] is needed when your host machine has multiple ip addresses)

(Note that if you use Host network of Docker, then you don’t need to use this option)

Example: -p 12:35 map port 12 of the host to port 35 in the container  
Your container can send packets to outside world, but when the outside world can send packets to only the host machine, not the Docker container (because Docker container does not have a real NIC). In order for any software (be it on the host machine or from the outside world) to send packets to the Docker container, there must be a firewall rule on the host machine to forward packets (incoming to the host machine) to the container. This firewall rule is created by -- publish host\_port:container\_port.

-- mount [type=bind|volume|tmpfs,][source=path/to/source,]target=path/to/target[,readonly]

-- volume [path/to/source]:path/to/target[:ro]

Both -- mount or -- volume (-v) can be used for both mounting a volume to containers and mounting a container directory to a host directory. Their syntaxes are different; the format of arguments in:

* -- mount: key1=value1, key2=value2, …
* -- volume: value1:value2: …

Mounting a volume to container: the path/to/source must be a volume name or omitted. When omitted, a docker volume will be created with a random name. If you specify a name but the volume does not exist, it will be created with the mentioned name.

Mounting a container directory to a host directory: obviously the path/to/source and path/to/target are paths. Note that Docker can easily distinguish when you use a path or a volume name since volume name, for example “myVolume”, does not have “/” separator like the path/to/a/directory.

The “type=bind|volume” of -- mount can be omitted since based on the path/to/source, Docker can understand when you want to mount a volume or mount a directory.

You can mount a directory to the host machine’s RAM by setting “type=tmpfs” for – mount. This way will make your data gone when you stop the container.

If you want to mount read-only, then add “,readonly” in -- mount or “:ro” in – volume.

Example:

-- network, -- net: connect to specific Docker network

-- env, -e: set environment variables

### docker container start

start an existing container

docker container start [option] container\_name

-- attach, -a: attach your terminal to the container

-- interactive, -i: Attach container's STDIN to the terminal that you’re using

Note that without the both option -ia, you cannot interact with your container even when the container is set to have a virtual terminal (by -it), so always use:

docker container start -ia container\_name

### docker container restart

Restarting a running container = stopping the container and then starting it

docker container restart container\_name

### docker container run

This command is the combination of docker container create and docker container start; it creates and runs a docker.

docker container run [options] image\_name

All options in docker container create like --name, -it, -p, --mount, --volume, --net, --env will be used in docker container run. The following is additional:

--detach, -d: run container in background and print container ID

By default, Docker runs the container in attached mode. Meaning it’s attached to the terminal session, where it displays output and messages. If you want to keep the container and current terminal session separate, you can run the container in the background using the -d attribute. Using detached mode also allows you to close the opened terminal session without stopping the container.

For example: docker container run -d e98b6ec72f51

How to run a Docker container in detached mode.

The output you receive will be similar to the one you see in the image above. The container will run the process and then stop. No other output will display inside the terminal session.

--rm: automatically remove the container when it exits

Once a container executes its tasks, it stops, but the file system it consists of remains on the system. If you only need a container to execute the outlined task and have no use of it or its file system afterward, you can set it up to delete once it is done. To run a container that will be automatically removed after exiting use the command:

docker container run --rm [docker\_image]

Review some options for docker container run that are in docker container create

-- name container\_name: specify a name for the container

When you use docker container run command, Docker automatically creates a container with randomly generated name. If you don’t want to use this random name, you can specify a name by --name.

-- it: run the container in the interactive mode

Docker allows you to run a container in interactive mode. This means you can execute commands inside the container while it is still running. By using the container interactively, you can access a command prompt inside the running container. To do so, run the following command:

docker container run -it [docker\_image] /bin/bash

The command prompt will change, moving you to the bash shell as in the example below.

Run a Docker container in interactive mode.

### docker container remove

to remove a container, you need to stop it first if it’s running and then

docker container rm container\_name

## Volume

### Create a volume

docker volume create my-vol

### List volumes

docker volume ls

Output

local my-vol

### Inspect a volume

docker volume inspect my-vol

Output:

[

{

"Driver": "local",

"Labels": {},

"Mountpoint": "/var/lib/docker/volumes/my-vol/\_data",

"Name": "my-vol",

"Options": {},

"Scope": "local"

}

]

### Remove a volume

docker volume rm my-vol

## Docker storage

### 3 level of storing Docker data: (1) image layer, (2) container layer, (3) external storage: volume, bind mount, tmpfs

Docker stores data pertaining to images, containers, volumes, etc under

* Windows: C:\ProgramData\DockerDesktop
* MacOS: ~/Library/Containers/com.docker.docker/Data/vms/0/
* Ubuntu, Fedora, Debian: /var/lib/docker/

Graphical user interface, application

Description automatically generated with medium confidence

Docker file location on Linux systems

When we run the docker build command, docker builds one layer for each instruction in the dockerfile. These image layers are **read-only image layers**.

When we run the docker run command, docker builds **writable container layer(s)**, which stores changes on image layers. While running a Docker container, if you modify a file then the modified file is copied to the writable container layer, and hence the original file from the image layer is not affected. This is called the **Copy-on-Write** mechanism. The writable container layer is kept on the host machine’s hard disk until the container is removed. Note that if you just stop the container but not yet remove it, then all the changes you have made on that container is still kept in the container layer.

You can also save data on your running container to an external storage, which can be a docker volume (a virtual storage) or a bind mounted directory on the host machine. If your host is Linux, you can choose to save data on your running container to the host machine RAM (tmpfs) and hence when you stop the container all data will be gone. Details for those two will be given in the next section.

A screenshot of a computer

Description automatically generated with medium confidence

Docker Layers

For example: you have an image that contains app.py file. While running this image as a container, you create a new file temp.txt and modify app.py, then both temp.txt and app.py are copied to the writable container layer. If you just shut down the container, then temp.txt and app.py are still stored in the container layer. However, if you delete the container, the container layer, and hence temp.txt and app.py will be gone.

Graphical user interface

Description automatically generated

Copy-on-Write Mechanism

To retain those 2 files after deleting the container, you can use save those two files on a docker volume, for example named data\_volume or a bind mounted directory, for example /data, on the host machine. Note that a docker volume is just a directory on the host machine, like a bind mounted directory, but the volume directory is managed exclusively by Docker while the bind mounted directory can be modified by both the Docker container and processes on the host machine.

Graphical user interface

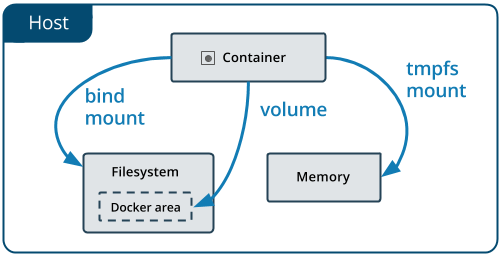
Description automatically generated

Persistent Storage with Volume Mapping

### Mount endpoint between container and host: volume, bind mount and tmpfs

By default all files created inside a container are stored on a writable container layer, which is deleted when the container is removed.

Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container is removed: volumes, and bind mounts. If you’re running Docker on Linux you can also use a tmpfs mount. If you’re running Docker on Windows you can also use a named pipe.



* **Bind mounts** may be stored anywhere on the host system. They may even be important system files or directories. The directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist.

Non-Docker processes on the Docker host or processes inside Docker container can modify the bind mounted directory on the host machine, which implies a security problem.

* **Volumes** are stored also in directories of the host machine; when you mount the volume into a container, this directory is what is mounted into the container. This is similar to the way that bind mounts work, *except that volumes are managed by Docker and are isolated from non-Docker processes of the host machine*.  
  You can create a volume explicitly using the docker volume create command, or Docker can create a volume during container or service creation.   
  A given volume can be mounted into multiple containers simultaneously. When no running container is using a volume, the volume is still available to Docker and is not removed automatically. You can remove unused volumes using docker volume prune.
* **tmpfs mounts** are stored in the host system’s memory only, and are never written to the host system’s filesystem.

## Docker network

A common use case for Docker is networked services, and Docker has its own networking model to let containers speak to both each other and the outside world.

### 4 networking types for containers: bridge, overlay, host, Macvlan.

**Bridge networks**

Bridge networks let containers running on the same Docker host communicate with each other. A new instance of Docker comes with a default bridge network named bridge, and by default all newly started containers connect to it.

For the best results, create your own bridge network. User-defined bridges have many features the bridge bridge does not:

* DNS resolution works automatically between containers on a custom bridge. This way, you don’t need to use raw IP addresses to communicate between them as you do on the bridge bridge. Containers can locate other containers via DNS using the container name.
* Containers can be added and removed from a custom bridge while they’re running.
* Environment variables can be shared between containers on a custom bridge.

**Overlay networks**

Bridge networks are for containers on the same host. Overlay networks are for containers running on different hosts, such as those in a Docker swarm. This lets containers across hosts find each other and communicate, without you having to worry about how to set that up for each individual participating container.

Docker’s swarm mode orchestrator automatically creates an overlay network, ingress. By default any services on the swarm attach themselves to ingress. But as with the default bridge, this isn’t the best choice for a production system, because the defaults may not be appropriate.

If you want to use an overlay network with containers not running in a swarm, that’s another use case for creating a custom overlay network. Note that each Docker host on an overlay network must have [the proper ports open to its peers](https://docs.docker.com/network/overlay/#operations-for-all-overlay-networks) to be seen, and without swarm mode [each node needs access to a key-value store](https://docs.docker.com/network/overlay-standalone.swarm/) of some kind.

**Host networking**

The host networking driver makes the programs inside the Docker container look like they are running on the host itself, from the perspective of the network. Normally you have to forward ports from the host machine into a container, but when the containers share the host's network, any network activity happens directly on the host machine - just as it would if the program was running locally on the host instead of inside a container.

The biggest boon of host networking is speed. If you need to give a container port access and you want to make it as close to the underlying OS as possible, this is the way to go. But it comes at the cost of flexibility: If you map port 80 to a container, no other container can use it on that host.

**Macvlan networking**

A Macvlan network is for applications that work directly with the underlying physical network, such as network-traffic monitoring applications. The macvlan driver doesn’t just assign an IP address to a container, but a physical MAC address as well. Macvlan should be reserved only for the applications that don’t work unless they rely on a physical network address.

### Managing network

All network management in Docker is done using: [docker network](https://docs.docker.com/engine/reference/commandline/network/) subcommand.

Many of its subcommands are similar to other Docker commands; for example, docker network ls displays all the configured networks on the current Docker instance.

#### Display all networks

To display all networks: docker network ls

After the docker installation you have 3 networks by default: bridge, host, null

$ docker network ls

NETWORK ID NAME DRIVER SCOPE

2e0adaa0ce4a bridge bridge local

0de3da43b973 host host local

724a28c6d86d none null local

#### Create a network

To create a network, use the create subcommand along with the --driver bridge|overlay|macvlan flag to indicate which driver to use (bridge, overlay, macvlan); if you don’t specify bridge type will be used:

$ docker network create --driver bridge my-bridge

Host-networked containers don’t require a network to be created for them. Instead, launch the container with the --network host flag. Any processes on the container listen on their preconfigured ports, so make sure those are set first.

The [options for creating a network](https://docs.docker.com/engine/reference/commandline/network_create/#options) also include specifying its subnet, IP address range, and network gateway, much as would be the case for creating a network using other means.

#### Connect/Disconnect a container to/from a network

Containers run by default on the bridge network. To use a particular network, just use the --network network\_name flag when launching the container.

You can also connect/disconnect a running container to/from a network:

$ docker network connect my\_network my\_container

$ docker network disconnect my\_network my\_container

This attaches/detaches my\_container to/from my\_network network, while preserving any existing network connections it already has.

#### Remove a network

When a container is spun down, any networks associated with it are left intact. If you want to remove networks manually, you can do so with the docker network rm <network\_name> command, or use docker network prune to remove all networks no longer in use on the host.

# Some notes about database docker

MySQL

Create a MySQL container (consisting of both server and client)

Create and run MySQL server

Quick command:

docker run --name mysql\_name --env MYSQL\_ROOT\_PASSWORD=mypassword -d mysql:tag

Longer command

To run MySQL client on the same MySQL server container:

Run bash on the MySQL server above:

docker container exec -it mysql\_container\_name bash

Enter MySQL client CLI by mysql and then you can show databases; to list databases.

root@bc5e4823cefa:/# mysql

mysql> show databases;

Connect sql client Adminer to MySQL container

- Root password:

- Network:

-

Note for the next time:

In the default network “bridge”, you have to use IP address, and cannot use containers’ names otherwise you have to use –link option, which is considered legacy.

For user-defined networks, you can use containers’ names, which is convenient.

The command to extract name of network, ipaddress, etc.

Containers on the default bridge network can only access each other by IP addresses, unless you use the [--link option](https://docs.docker.com/network/links/), which is considered legacy. On a user-defined bridge network, containers can resolve each other by name or alias.

Imagine an application with a web front-end and a database back-end. If you call your containers web and db, the web container can connect to the db container at db, no matter which Docker host the application stack is running on.

If you run the same application stack on the default bridge network, you need to manually create links between the containers (using the legacy --link flag). These links need to be created in both directions, so you can see this gets complex with more than two containers which need to communicate. Alternatively, you can manipulate the /etc/hosts files within the containers, but this creates problems that are difficult to debug.