### Storage Drivers

There are various storage drivers available in Docker, these includes:

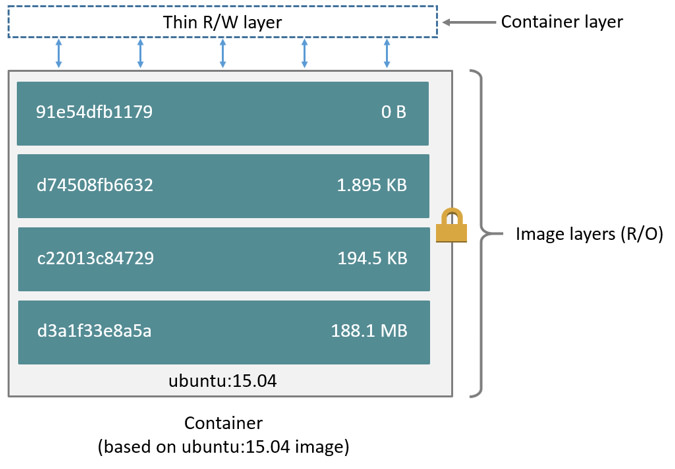
* AUFS
* Device Mapper
* **OverlayFS Default**
* ZFS
* VFS
* Brtfs

Storage drivers allow you to create data in the writable layer of your container. The files won’t be persisted after the container is deleted, and both read and write speeds are lower than native file system performance.

**Images & Layers**

A Docker image is built up from a series of layers. Each layer represents an instruction in the image’s Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

* # syntax=docker/dockerfile:1
* FROM ubuntu:18.04
* COPY . /app
* RUN make /app
* CMD python /app/app.py



This Dockerfile contains four commands, each of which creates a layer. The FROM statement starts out by creating a layer from the ubuntu:18.04 image. The COPY command adds some files from your Docker client’s current directory. The RUN command builds your application using the make command.

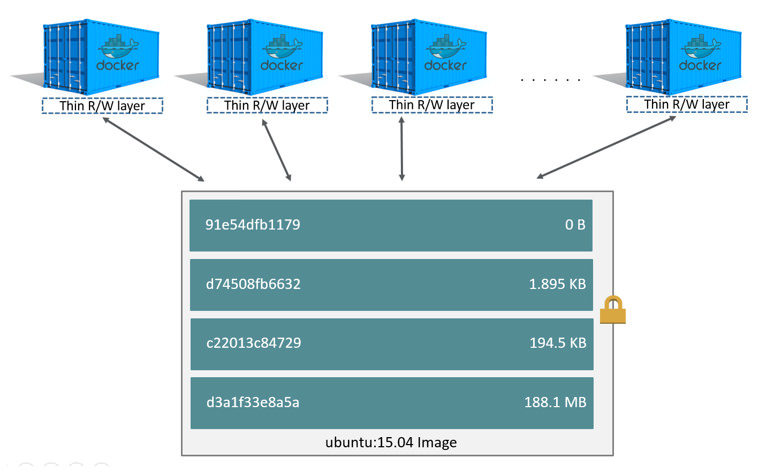
Finally, the last layer specifies what command to run within the container.

Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the “container layer”. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on the Ubuntu 15.04 image.

**A storage driver handles the details about the way these layers interact with each other. Different storage drivers are available, which have advantages and disadvantages in different situations.**

**Containers & Layers**

* The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.
* Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.



**Note**: If you need multiple images to have shared access to the exact same data, store this data in a Docker volume and mount it into your containers.

Docker uses storage drivers to manage the contents of the image layers and the writable container layer. Each storage driver handles the implementation differently, but all drivers use stackable image layers and the copy-on-write (CoW) strategy.

## **The copy-on-write (CoW) strategy**

Copy-on-write is a strategy of sharing and copying files for maximum efficiency. If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file. The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified. This minimizes I/O and the size of each of the subsequent layers. These advantages are explained in more depth below.

**Sharing promotes smaller images**

When you use docker pull to pull down an image from a repository, or when you create a container from an image that does not yet exist locally, each layer is pulled down separately, and stored in Docker’s local storage area, which is usually /var/lib/docker/ on Linux hosts. You can see these layers being pulled in this example:

$ docker pull ubuntu:18.04

* 18.04: Pulling from library/ubuntu
* f476d66f5408: Pull complete
* 8882c27f669e: Pull complete
* d9af21273955: Pull complete
* f5029279ec12: Pull complete
* Digest: sha256:ab6cb8de3ad7bb33e2534677f865008535427390b117d7939193f8d1a6613e34
* Status: Downloaded newer image for ubuntu:18.04

Each of these layers is stored in its own directory inside the Docker host’s local storage area. To examine the layers on the filesystem, list the contents of /var/lib/docker/<storage-driver>. This example uses the overlay2 storage driver:

$ ls /var/lib/docker/overlay2

* 16802227a96c24dcbeab5b37821e2b67a9f921749cd9a2e386d5a6d5bc6fc6d3
* 377d73dbb466e0bc7c9ee23166771b35ebdbe02ef17753d79fd3571d4ce659d7
* 3f02d96212b03e3383160d31d7c6aeca750d2d8a1879965b89fe8146594c453d
* ec1ec45792908e90484f7e629330666e7eee599f08729c93890a7205a6ba35f5l

Now imagine that you have two different Dockerfiles. You use the first one to create an image called acme/my-base-image:1.0.

* # syntax=docker/dockerfile:1
* FROM ubuntu:18.04
* COPY . /app

The second one is based on acme/my-base-image:1.0, but has some additional layers:

# syntax=docker/dockerfile:1

* FROM acme/my-base-image:1.0
* CMD /app/hello.sh

The second image contains all the layers from the first image, plus a new layer with the CMD instruction, and a read-write container layer. Docker already has all the layers from the first image, so it does not need to pull them again. The two images share any layers they have in common.

If you build images from the two Dockerfiles, you can use docker image ls and docker history commands to verify that the cryptographic IDs of the shared layers are the same.

1. Make a new directory cow-test/ and change into it.
2. Within cow-test/, create a new file called hello.sh with the following contents:
3. #!/bin/sh

echo "Hello world"

Save the file, and make it executable:

chmod +x hello.s

1. Copy the contents of the first Dockerfile above into a new file called Dockerfile.base.
2. Copy the contents of the second Dockerfile above into a new file called Dockerfile.
3. Within the cow-test/ directory, build the first image. Don’t forget to include the final . in the command. That sets the PATH, which tells Docker where to look for any files that need to be added to the image.

$ docker build -t acme/my-base-image:1.0 -f Dockerfile.base .

* Sending build context to Docker daemon 812.4MB
* Step 1/2 : FROM ubuntu:18.04
* ---> d131e0fa2585
* Step 2/2 : COPY . /app
* ---> Using cache
* ---> bd09118bcef6
* Successfully built bd09118bcef6
* Successfully tagged acme/my-base-image:1.0

1. Build the second image.

$ docker build -t acme/my-final-image:1.0 -f Dockerfile .

* Sending build context to Docker daemon 4.096kB
* Step 1/2 : FROM acme/my-base-image:1.0
* ---> bd09118bcef6
* Step 2/2 : CMD /app/hello.sh
* ---> Running in a07b694759ba
* ---> dbf995fc07ff
* Removing intermediate container a07b694759ba
* Successfully built dbf995fc07ff
* Successfully tagged acme/my-final-image:1.0

1. Check out the sizes of the images:

$ docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

acme/my-final-image 1.0 dbf995fc07ff 58 seconds ago 103MB

acme/my-base-image 1.0 bd09118bcef6 3 minutes ago 103MB

1. Check out the layers that comprise each image:

$ docker history bd09118bcef6

IMAGE CREATED CREATED BY SIZE COMMENT

bd09118bcef6 4 minutes ago /bin/sh -c #(nop) COPY dir:35a7eb158c1504e... 100B

d131e0fa2585 3 months ago /bin/sh -c #(nop) CMD ["/bin/bash"] 0B

<missing> 3 months ago /bin/sh -c mkdir -p /run/systemd && echo '... 7B

<missing> 3 months ago /bin/sh -c sed -i 's/^#\s\*\(deb.\*universe\... 2.78kB

<missing> 3 months ago /bin/sh -c rm -rf /var/lib/apt/lists/\* 0B

<missing> 3 months ago /bin/sh -c set -xe && echo '#!/bin/sh' >... 745B

<missing> 3 months ago /bin/sh -c #(nop) ADD file:eef57983bd66e3a... 103MB

$ docker history dbf995fc07ff

IMAGE CREATED CREATED BY SIZE COMMENT

dbf995fc07ff 3 minutes ago /bin/sh -c #(nop) CMD ["/bin/sh" "-c" "/a... 0B

bd09118bcef6 5 minutes ago /bin/sh -c #(nop) COPY dir:35a7eb158c1504e... 100B

d131e0fa2585 3 months ago /bin/sh -c #(nop) CMD ["/bin/bash"] 0B

<missing> 3 months ago /bin/sh -c mkdir -p /run/systemd && echo '... 7B

<missing> 3 months ago /bin/sh -c sed -i 's/^#\s\*\(deb.\*universe\... 2.78kB

<missing> 3 months ago /bin/sh -c rm -rf /var/lib/apt/lists/\* 0B

<missing> 3 months ago /bin/sh -c set -xe && echo '#!/bin/sh' >... 745B

<missing> 3 months ago /bin/sh -c #(nop) ADD file:eef57983bd66e3a... 103MB

Notice that all the layers are identical except the top layer of the second image. All the other layers are shared between the two images, and are only stored once in /var/lib/docker/. The new layer actually doesn’t take any room at all, because it is not changing any files, but only running a command.

**Note**: The <missing> lines in the docker history output indicate that those layers were built on another system and are not available locally. This can be ignored.

**Copying makes containers efficient**

When you start a container, a thin writable container layer is added on top of the other layers. Any changes the container makes to the filesystem are stored here. Any files the container does not change do not get copied to this writable layer. This means that the writable layer is as small as possible.

When an existing file in a container is modified, the storage driver performs a copy-on-write operation. The specifics steps involved depend on the specific storage driver. For the aufs, overlay, and overlay2 drivers, the copy-on-write operation follows this rough sequence:

* Search through the image layers for the file to update. The process starts at the newest layer and works down to the base layer one layer at a time. When results are found, they are added to a cache to speed future operations.
* Perform a copy\_up operation on the first copy of the file that is found, to copy the file to the container’s writable layer.
* Any modifications are made to this copy of the file, and the container cannot see the read-only copy of the file that exists in the lower layer.

Btrfs, ZFS, and other drivers handle the copy-on-write differently. You can read more about the methods of these drivers later in their detailed descriptions.

Containers that write a lot of data consume more space than containers that do not. This is because most write operations consume new space in the container’s thin writable top layer.

**Note: for write-heavy applications, you should not store the data in the container. Instead, use Docker volumes, which are independent of the running container and are designed to be efficient for I/O. In addition, volumes can be shared among containers and do not increase the size of your container’s writable layer.**

A copy\_up operation can incur a noticeable performance overhead. This overhead is different depending on which storage driver is in use. Large files, lots of layers, and deep directory trees can make the impact more noticeable. This is mitigated by the fact that each copy\_up operation only occurs the first time a given file is modified.

To verify the way that copy-on-write works, the following procedures spins up 5 containers based on the acme/my-final-image:1.0 image we built earlier and examines how much room they take up.

1] From a terminal on your Docker host, run the following docker run commands. The strings at the end are the IDs of each container.

* $ docker run -dit --name my\_container\_1 acme/my-final-image:1.0 bash \
* && docker run -dit --name my\_container\_2 acme/my-final-image:1.0 bash \
* && docker run -dit --name my\_container\_3 acme/my-final-image:1.0 bash \
* && docker run -dit --name my\_container\_4 acme/my-final-image:1.0 bash \
* && docker run -dit --name my\_container\_5 acme/my-final-image:1.0 bash
* c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c409
* dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea513
* 1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33d0c
* 38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc818544
* 1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e765

2] Run the docker ps command to verify the 5 containers are running.

3] List the contents of the local storage area.

* $ sudo ls /var/lib/docker/containers
* 1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e765
* 1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33d0c
* 38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc818544
* c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c409
* dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea51

**Now check out their sizes:**

* $ sudo du -sh /var/lib/docker/containers/\*
* 32K /var/lib/docker/containers/1a174fc216cccf18ec7d4fe14e008e30130b11ede0f0f94a87982e310cf2e765
* 32K /var/lib/docker/containers/1e7264576d78a3134fbaf7829bc24b1d96017cf2bc046b7cd8b08b5775c33d0c
* 32K /var/lib/docker/containers/38fa94212a419a082e6a6b87a8e2ec4a44dd327d7069b85892a707e3fc818544
* 32K /var/lib/docker/containers/c36785c423ec7e0422b2af7364a7ba4da6146cbba7981a0951fcc3fa0430c409
* 32K /var/lib/docker/containers/dcad7101795e4206e637d9358a818e5c32e13b349e62b00bf05cd5a4343ea513

Each of these containers only takes up 32k of space on the filesystem.

Not only does copy-on-write save space, but it also reduces start-up time. When you start a container (or multiple containers from the same image), Docker only needs to create the thin writable container layer.

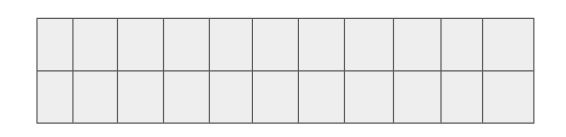
If Docker had to make an entire copy of the underlying image stack each time it started a new container, container start times and disk space used would be significantly increased. This would be similar to the way that virtual machines work, with one or more virtual disks per virtual machine.

## **Block vs Object Storage**

In block storage, the data is stored in terms of blocks

Data stored in blocks are normally read or written entirely a whole block at the same time

Most of the file systems are based on block devices.



Every block has an address and the application can be called via SCSI call via its address

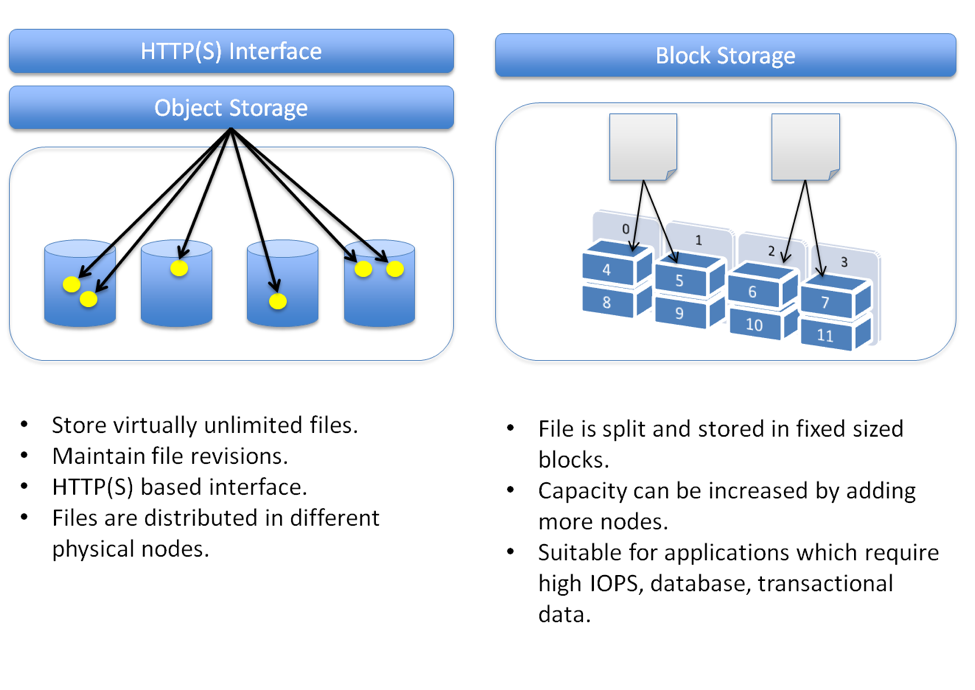
There is no storage side meta-data associated with the block except the address.

Thus block has no description, no owner

2.2 Object Storage: [S3]

Object storage is a data storage architecture that manages data as objects as opposed to blocks of storage.

An object is defined as a data (file) along with all its meta-data which is combined together as an object.

This object is given an ID which is calculated from the content of the object (from the data and metadata ). The application can then call the object with the unique object ID

## **Changing Storage Drivers**

To see what storage driver Docker is currently using, use docker info and look for the Storage Driver line:

$ docker info

* Containers: 0
* Images: 0
* Storage Driver: overlay2
* Backing Filesystem: xfs
* <...>

**Important**: When you change the storage driver, any existing images and containers become inaccessible. This is because their layers cannot be used by the new storage driver. If you revert your changes, you can access the old images and containers again, but any that you pulled or created using the new driver are then inaccessible.

$ systemctl stop docker

$ cd /etc/docker

$ vim daemon.json

{

“storage-driver”: “aufs”

}

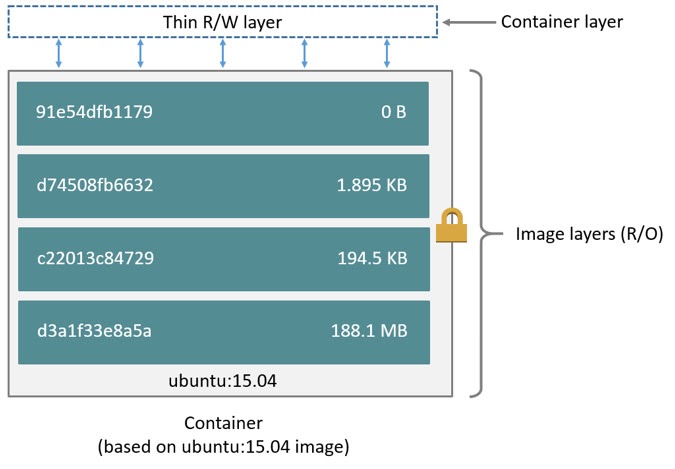
$ systemctl start docker

## **Docker Volume**

**Challenges with files in Container Writable Layer**

* By default, all files created inside a container are stored on a writable container layer. This means that:
* The data doesn’t persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.
* Writing into a container’s writable layer requires a storage driver to manage the filesystem. The storage driver provides a union filesystem, using the Linux kernel.
* This extra abstraction reduces performance as compared to using data volumes, which write directly to the host filesystem.

**/var/lib/docker/volume**



**Use Volumes**

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While [bind mounts](https://docs.docker.com/storage/bind-mounts/) are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

* Volumes are easier to back up or migrate than bind mounts.
* You can manage volumes using Docker CLI commands or the Docker API.
* Volumes work on both Linux and Windows containers.
* Volumes can be more safely shared among multiple containers.
* Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
* New volumes can have their content pre-populated by a container.
* Volumes on Docker Desktop have much higher performance than bind mounts from Mac and Windows hosts.

In addition, volumes are often a better choice than persisting data in a container’s writable layer, because a volume does not increase the size of the containers using it, and the volume’s contents exist outside the lifecycle of a given container.



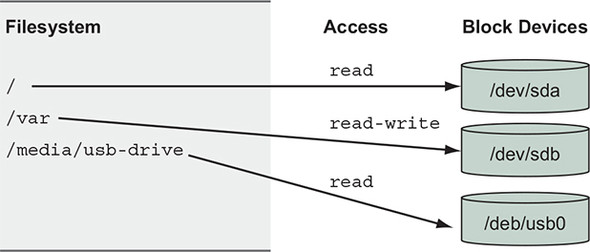
If your container generates non-persistent state data, consider using a [tmpfs mount](https://docs.docker.com/storage/tmpfs/) to avoid storing the data anywhere permanently, and to increase the container’s performance by avoiding writing into the container’s writable layer.

Volumes use rprivate bind propagation, and bind propagation is not configurable for volumes.

## **File Trees & Mount Points**

Unlike other operating systems, Linux unifies all storage into a single tree. Storage devices such as disk partitions or USB disk partitions are attached to specific locations in that tree. Those locations are called *mount points*. A mount point defines the location in the tree, the access properties to the data at that point (for example, writability), and the source of the data mounted at that point (for example, a specific hard disk, USB device, or memory-backed virtual disk). [Figure 4.1](https://livebook.manning.com/book/docker-in-action-second-edition/chapter-4/ch04fig01) depicts a filesystem constructed from multiple storage devices, with each device mounted to a specific location and level of access.

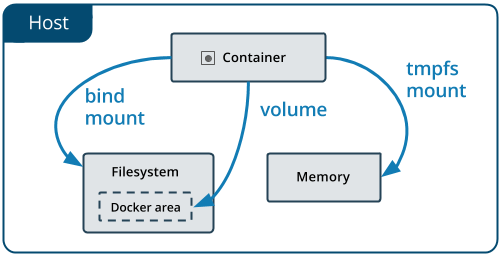
Mount points allow software and users to use the file tree in a Linux environment without knowing exactly how that tree is mapped into specific storage devices. This is particularly useful in container environments.



## **Mount types available in Docker Volume**

There are three mount types available in Docker

* **Volumes** **are stored in a part of the host filesystem**which is managed by Docker (**/var/lib/docker/volumes/ on Linux**). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
* **Bind mounts may be stored anywhere on the host system.** They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.
* **tmpfs mounts are stored in the host system’s memory only**, and are never written to the host system’s filesystem.  
    
  An easy way to visualize the difference among volumes, bind mounts, and tmpfs mounts is to think about where the data lives on the Docker host.



## **Bind Mounts**

* Bind mounts have limited functionality compared to [volumes](https://docs.docker.com/storage/volumes/).
* When you use a bind mount, a file or directory on the host machine is mounted into a container.
* The file or directory is referenced by its absolute path on the host machine. By contrast, when you use a volume, a new directory is created within Docker’s storage directory on the host machine, and Docker manages that directory’s contents.
* The file or directory does not need to exist on the Docker host already. It is created on demand if it does not yet exist. Bind mounts are very performant, but they rely on the host machine’s filesystem having a specific directory structure available. If you are developing new Docker applications, consider using [named volumes](https://docs.docker.com/storage/volumes/) instead. You can’t use Docker CLI commands to directly manage bind mounts

**Start a container with Bind mount**

**Using –mount option**

* $ docker run -d -it --name devtest --mount type=bind,source="$(pwd)"/target,target=/app nginx:latest

Use read-only bind mount

For some development applications, the container needs to write into the bind mount, so changes are propagated back to the Docker host. At other times, the container only needs read access.

* $ docker run -d -it --name devtest --mount type=bind,source="$(pwd)"/target,target=/app,readonly nginx:latest

**Using -v**

**bind mount:** note that the host path should start with ‘/’. Use $(pwd) for convenience.

* $ docker run -d -it --name devtest -v "$(pwd)"/target:/app nginx:latest
* $ docker container run -v /host-path:/container-path image-name

Readonly

* $ docker run -d -it --name devtest -v "$(pwd)"/target:/app:ro nginx:latest

## **TMPFS Mounts**

In-memory storage

Most service software and web applications use private key files, database passwords, API key files, or other sensitive configuration files, and need upload buffering space. In these cases, it is important that you never include those types of files in an image or write them to disk. Instead, you should use in-memory storage. You can add in-memory storage to containers with a special type of mount.

The data is written directly on to the host’s memory and deleted when the container is stopped. Very useful when it involves sensitive data that you simply don’t want to be permanen

Temporary filesystems are written to RAM (or to your swap file if RAM is filling up) and not to the host or the container’s own filesystem layer at Docker.com

$ docker run -d --name tmptest --mount type=tmpfs,destination=/app nginx:latest

## **Volume vs Bind Mounts**

* With Bind Mount, a file or directory on the *host machine* is mounted into a container. The file or directory is referenced by its full or relative path on the host machine.
* With Volume, a new directory is created **within Docker's storage directory** on the host machine, and **Docker manages** that directory's content.

**Volumes advantages over bind mounts:**

* Volumes are easier to back up or migrate than bind mounts.
* You can manage volumes using Docker CLI commands or the Docker API.
* Volumes work on both Linux and Windows containers.
* Volumes can be more safely shared among multiple containers.
* Volume drivers allow you to store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
* A new volume’s contents can be pre-populated by a container.

## **Docker Volume Create**

Creates a new volume that containers can consume and store data in. If a name is not specified, Docker generates a random name. **/var/lib/docker/volume**

$ docker volume create [OPTIONS] [VOLUME]

--driver , -d local Specify volume driver name

--label Set metadata for a volume

--name Specify volume name

--opt , -o Set driver specific options

**Example**

$ docker volume create hello

hello

$ docker run -d -v hello:/world busybox ls /world

The mount is created inside the container’s /world directory. Docker does not support relative paths for mount points inside the container.

Multiple containers can use the same volume in the same time period. This is useful if two containers need access to shared data

## **Docker Volume ls**

List all the volumes known to Docker. You can filter using the -f or --filter flag. Refer to the filtering section for more information about available filter options.

$ docker volume ls [OPTIONS]

Options

* --filter , -f Provide filter values (e.g. 'dangling=true')
* --format Pretty-print volumes using a Go template
* --quiet , -q Only display volume names

Example

$ docker volume ls

**Filtering**

The filtering flag (-f or --filter) format is of “key=value”. If there is more than one filter, then pass multiple flags (e.g., --filter "foo=bar" --filter "bif=baz")

The currently supported filters are:

* dangling (boolean - true or false, 0 or 1)
* driver (a volume driver’s name)
* label (label=<key> or label=<key>=<value>)
* name (a volume’s name)

$ docker volume ls -f dangling=true

$ docker volume create the-doctor --label is-timelord=yes

$ docker volume ls -f name=rose

**Format**

$ docker volume ls --format "{{.Name}}: {{.Driver}}"

## **Docker Volume rm**

Remove one or more volumes. You cannot remove a volume that is in use by a container.

$ docker volume rm [OPTIONS] VOLUME [VOLUME...]

--force , -f Force the removal of one or more volumes

Example   
$ docker volume rm hello

## **Docker Volume prune**

Remove all unused local volumes. Unused local volumes are those which are not referenced by any containers

$ docker volume prune [OPTIONS]

* --filter Provide filter values (e.g. 'label=<label>')
* --force , -f Do not prompt for confirmation

$ docker volume prune

## **Docker Volume Inspect**

Returns information about a volume. By default, this command renders all results in a JSON array. You can specify an alternate format to execute a given template for each result

$ docker volume inspect [OPTIONS] VOLUME [VOLUME...]

* --format , -f Format the output using the given Go template

Example

* $ docker volume inspect myvolume
* $ docker volume inspect --format '{{ .Mountpoint }}' myvolume

## **Remove Volume on container exit**

$ docker container run –rm -dt –name duck -v /test busybox ping -c10 google.com

## **Named Volume vs Anonymous Vol**

* $ docker rm -fv containerName [Volume gets deleted with container for anonymous]
* $ docker rm -fv containerName [Only container gets deleted]

## **Volume in Dockerfile**

It creates anonymous volume

FROM centos

VOLUME /opt

## **Persist Jenkins data using Vol**

What folder in container should I associate ? Get the information internet.

For Jenkins it is /var/Jenkins\_home

Create a Named volume in host and associate it in container