Data management in Docker
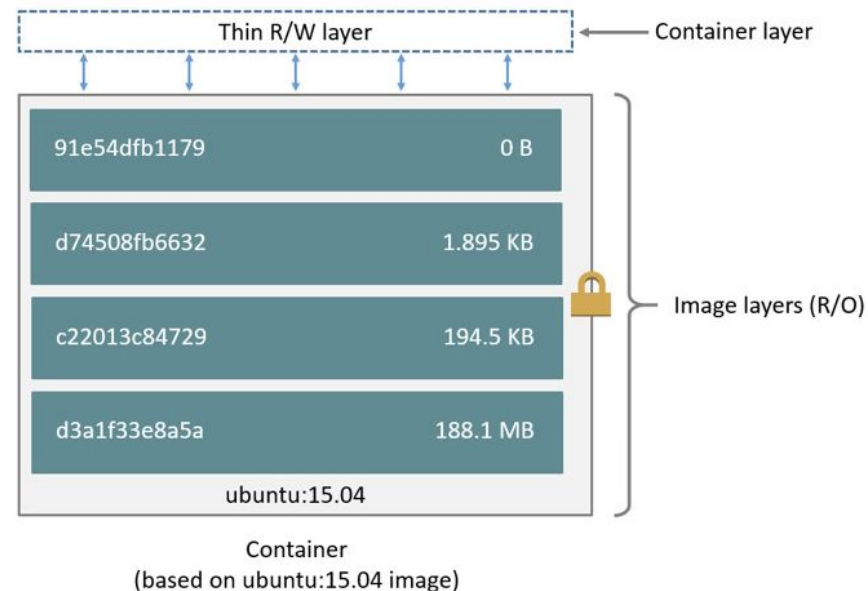
Stefano Nicotri (INFN Bari)
nicotri@infn.it

Outline

- Recap of data layers in Docker containers
- Storage drivers
- Persistence of data
- Bind mounts
- Volumes
- tmpfs mounts
- Real life examples

Docker image layers

- A Docker Image consists of read-only layers built on top of each other.
- Docker uses the **Union File System** (UFS) to build an image.
- The image is shared across containers.
- 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.

Thin R/W layer — Container layer

91e54dfb1179	0 B
d74508fb6632	1.895 KB
c22013c84729	194.5 KB
d3a1f33e8a5a	188.1 MB

ubuntu:15.04

Image layers (R/O)

Container
(based on ubuntu:15.04 image)

Docker image vs container

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.

When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.
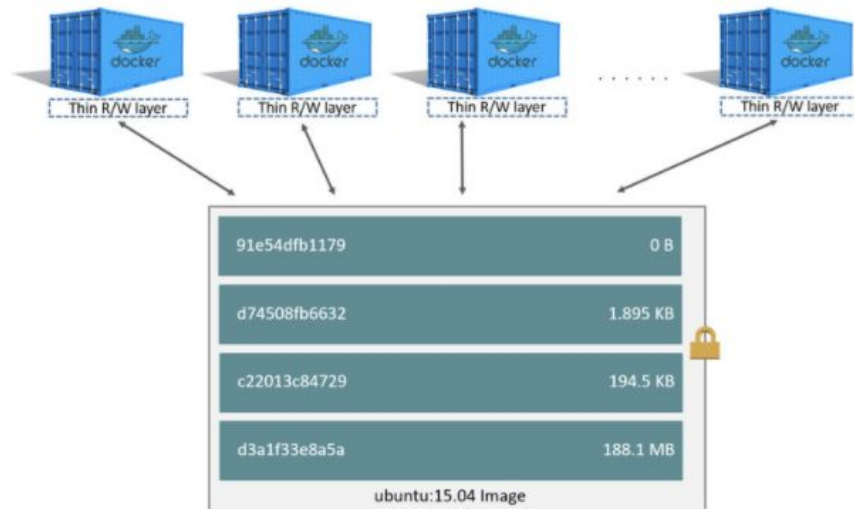
Docker storage drivers

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.

Docker supports the following storage drivers:

- *overlay2* is the preferred storage driver, for all currently supported Linux distributions, and requires no extra configuration.
- *aufs* was the preferred storage driver for Docker 18.06 and older, when running on Ubuntu 14.04 on kernel 3.13 which had no support for overlay2.
- *fuse-overlayfs* is preferred only for running Rootless Docker on a host that does not provide support for rootless overlay2. On Ubuntu and Debian 10, the fuse-overlayfs driver does not need to be used as overlay2 works even in rootless mode.
- *devicemapper* is supported, but requires direct-lvm for production environments, because loopback-lvm, while zero-configuration, has very poor performance. devicemapper was the recommended storage driver for CentOS and RHEL, as their kernel version did not support overlay2. However, current versions of CentOS and RHEL now have support for overlay2, which is now the recommended driver.
- The *btrfs* and *zfs* storage drivers are used if they are the backing filesystem (the filesystem of the host on which Docker is installed). These file systems allow for advanced options, such as creating "snapshots", but require more maintenance and setup. Each of these relies on the backing filesystem being configured correctly.
- The *vfs* storage driver is intended for testing purposes, and for situations where no copy-on-write filesystem can be used. Performance of this storage driver is poor, and is not generally recommended for production use.

More info at https://docs.docker.com/storage/storagedriver/select-storage-driver/

Copy-On-Write mechanism

COW is a standard UNIX pattern that provides a single shared copy of some data until the data is modified.

Docker makes use of copy-on-write technology with both images and containers. This **CoW strategy optimizes both image disk space usage and the performance of container start times**. At start time, Docker only has to create the thin writable layer for each container.

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. (Source: Docker docs)

Data in containers

When files are **created inside a container** they are stored on a writable container layer and **don't persist** when that container no longer exists

Data created in this way are "tightly linked" to the container: you can't easily move them somewhere else.

Writing data into a container requires a **storage driver** to manage the filesystem → reduced performances as compared to using volumes, which write directly to the host filesystem.

Docker storage drivers

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**.

Available storage drivers:

- *overlay2* is the preferred storage driver, for all currently supported Linux distributions, and requires no extra configuration.
- *aufs* was the preferred storage driver for Docker 18.06 and older, when running on Ubuntu 14.04 on kernel 3.13 which had no support for overlay2.
- *fuse-overlayfs* is preferred only for running Rootless Docker on a host that does not provide support for rootless overlay2. On Ubuntu and Debian 10, the fuse-overlayfs driver does not need to be used as overlay2 works even in rootless mode.
- *devicemapper* is supported, but requires direct-lvm for production environments, because loopback-lvm, while zero-configuration, has very poor performance. devicemapper was the recommended storage driver for CentOS and RHEL, as their kernel version did not support overlay2. However, current versions of CentOS and RHEL now have support for overlay2, which is now the recommended driver.
- The *btrfs* and *zfs* storage drivers are used if they are the backing filesystem (the filesystem of the host on which Docker is installed). These file systems allow for advanced options, such as creating "snapshots", but require more maintenance and setup. Each of these relies on the backing filesystem being configured correctly.
- The *vfs* storage driver is intended for testing purposes, and for situations where no copy-on-write filesystem can be used. Performance of this storage driver is poor, and is not generally recommended for production use.

More info at https://docs.docker.com/storage/storagedriver/select-storage-driver/

Data in containers

Instead, you can **store files on the host machine**, making them available to containers, so that the files are persisted even after the container stops → **volumes, and bind mounts**.

Another possibility: **containers storing files in-memory on the host machine** (files don't persist)

Persistent data with volumes

- **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 → **potentially high risk**

- **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**

- **tmpfs** mounts are stored in the host system's memory only, and are never written to the host's files system
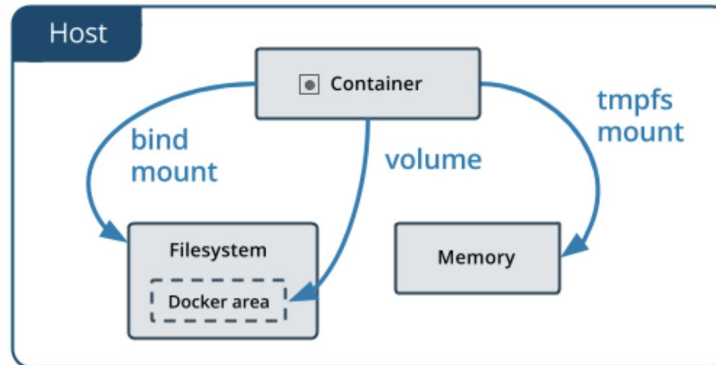
Bind mounts

Bind mounts are the *legacy* solution to manage data in Docker containers.

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 full path on the host machine.

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.

Bind mounts cannot be directly managed through Docker CLI commands

Use cases for bind mounts

Sharing configuration files from the host machine to containers. For example, DNS resolution is shared to containers by mounting /etc/resolv.conf from the host machine into each container

Sharing source code or build artifacts between a development environment on the Docker host and a container. For instance, you may mount a Maven target/ directory into a container, and each time you build the Maven project on the Docker host, the container gets access to the rebuilt artifacts

When the file or directory structure of the Docker host is guaranteed to be consistent with the bind mounts the containers require

Volumes

create a volume → `docker volume create my-vol`

a "volume" named `my-vol` is created within an isolated path managed by Docker (/var/lib/docker/volumes/ on linux)

when attached to a container, the corresponding directory is mounted within the container (similarly to the bind mount, but not involving "dangerous" directories)

If you don't explicitly create it, a volume is created the first time it is mounted into a container

volumes can be named, to ease their management

volumes can be mounted to multiple containers simultaneously (either rw or ro)

when a volume is not used by any container, it is still available to be attached to new containers

unused volumes can be destroyed with the `docker volume prune` command (otherwise they will stay there)

volumes allow more possibilities…

Docker volume plugins

Volumes also support the use of volume drivers, which allow you to store your data on remote hosts or cloud providers, among other possibilities.

- Extend the functionality of the Docker Engine
- Use the extensible Docker plugin API
- Allows an end-user to consume existing storage and its functionality
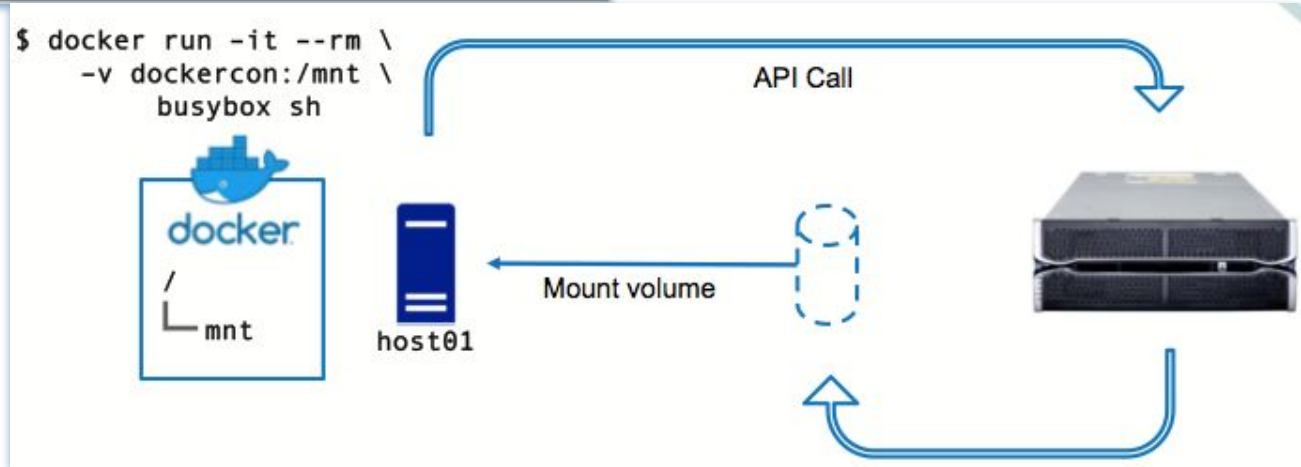- Create Docker storage volumes that are linked to containers lifecycle (can be persisted afterwards if needed)

More details: https://docs.docker.com/engine/extend/legacy_plugins/#volume-plugins

Volume plugin workflow

```
[dan@dockercon ~]$ docker plugin install store/storagedriver/array


[dan@dockercon ~]$ docker volume -d array -o ssd -o 32Gb fast_volume
fast_volume

[dan@dockercon ~]$ docker volume ls
DRIVER              VOLUME NAME
array               fast_volume
```

```
$ docker run -it --rm \
     -v dockercon:/mnt \
        busybox sh
```

API Call

Mount volume

host01

Corso base su docker 5-7 Set, 2022

Plugin benefits

Data-intensive applications: Volume plugins expose specialized functionality in storage providers that can be utilised for data−intensive workloads.

Data migration: Volume plugins make it easy to move data across hosts in the form of snapshots (e.g. enable migration of production databases from one host to another with minimum downtime)

Stateful application failover: Ability to have volumes that can be easily moved and re-attached, allowing easy failover to new machines/instances and re-attaching of data volumes.

Advantages of using volumes

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

Volumes don't increase the size of the containers using them

Each volume's contents exist outside the lifecycle of a given container.

When you need to back up, restore, or migrate data from one Docker host to another, you just stop containers using the volume and back up the volume's directory (such as /var/lib/docker/volumes/<volume-name>)

Use cases for volumes

mostly all use cases :) (you are strongly encouraged to always use volumes):

Sharing data among multiple running containers

When the Docker host is not guaranteed to have a given directory or file structure.

Decoupling the configuration of the Docker host from the container runtime.

When you want to store your container's data on a remote host or a cloud provider, rather than locally.

When your application requires high-performance I/O or fully native file system behavior on Docker Desktop, since volumes are stored in the Linux VM rather than the host, whereas bind mounts are remoted to macOS or Windows, where the file systems behave slightly differently

tmpfs mounts

A tmpfs mount is **not persistent** on disk, either on the Docker host or within a container. It can be used by a container during the lifetime of the container, to store non-persistent state or sensitive information. For instance, internally, swarm services use tmpfs mounts to mount secrets into a service's containers.

Use cases for tmpfs mounts

tmpfs mounts are typically used when you do not want the data to persist either on the host machine or within the container

This may be for security reasons or to protect the performance of the container when your application needs to write a large volume of non-persistent state data

What happens if we deal with non-empty directories?

If you mount an empty volume into a directory in the container in which files or directories exist, these files or directories are propagated (copied) into the volume

If you mount a bind mount or non-empty volume into a directory in the container in which some files or directories exist, these files or directories are obscured by the mount, just as if you saved files into /mnt on a Linux host and then mounted a USB drive into /mnt. The obscured files are not removed or altered, but are not accessible while the bind mount or volume is mounted

Examples

Commands to manage Docker volumes

```
$ docker help volume

Usage:  docker volume COMMAND

Manage volumes

Commands:
  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

Run 'docker volume COMMAND --help' for more information on a command.
```

Some "advanced" features

Task:

Create a volume called `volume-nfs` using the `local` driver, using it to mount an NFS share from `192.168.1.1` in `rw` to `/opt/my-nfs-share`

Create and delete volumes

Solution:

```
$ docker volume create --driver local \
    --opt type=nfs \
    --opt o=addr=192.168.1.1,rw \
    --opt device=:/path/to/dir \
    volume-nfs

$ docker inspect volume-nfs
[
    {
        "CreatedAt": "2022-08-30T12:33:05Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/volume-nfs/_data",
        "Name": "volume-nfs",
        "Options": {
            "device": ":/path/to/dir",
            "o": "addr=192.168.1.1,rw",
            "type": "nfs"
        },
        "Scope": "local"
    }
]
```

Create and delete volumes

The options depend on the driver. This commands create a tmpfs type volume using the local driver, specifying its size and uid

```
$ docker volume create --driver local \
    --opt type=tmpfs \
    --opt device=tmpfs \
    --opt o=size=100m,uid=1000 \
  volume-tmpfs

$ docker volume inspect volume-tmpfs
[
    {
        "CreatedAt": "2022-08-30T12:36:26Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/volume-tmpfs/_data",
        "Name": "volume-tmpfs",
        "Options": {
            "device": "tmpfs",
            "o": "size=100m,uid=1000",
            "type": "tmpfs"
        },
        "Scope": "local"
    }
]
```

Attach a volume to an existing container

This is tricky: it's not possible.

Containers must have their volumes configured on startup, which means to add a new volume, you must recreate the container (or hack it, which is not a very good solution....)

Thank you for your attention