Swarm Storage !

Πίνακας περιεχομένων

1. Manage data

By default all files created inside a container are stored on a writable container layer. This means that:

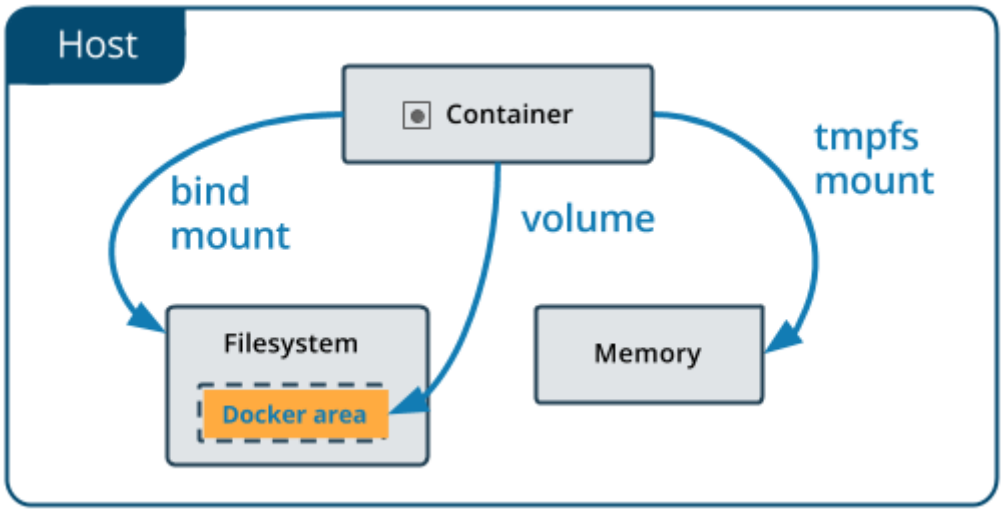
- The data doesn't persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.
- A container's writable layer is tightly coupled to the host machine where the container is running. You can't easily move the data somewhere else.
- 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.

Volumes and bind

Docker has two options for containers to store files in the host machine, so that the files are persisted even after the container stops:

- volumes,
- and bind mounts.

On Linux you can also use a tmpfs mount.



No matter which type of mount you choose to use, the data looks the same from within the container. It is exposed as either a directory or an individual file in the container's filesystem.

- 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.
- 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.

1.1. Volumes

Created and managed by Docker. You can create a volume explicitly using the **docker volume create** command, or Docker can create a volume during container or service creation.

When you create a volume, it is stored within a directory on the Docker host. 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 the core functionality of the host machine.

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.**

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

1.1.1. use cases for volumes

- **Sharing data among multiple running containers.** If you don't explicitly create it, a volume is created the first time it is mounted into a container. When that container stops or is removed, the volume still exists. Multiple containers can mount the same volume simultaneously, either read-write or read-only. Volumes are only removed when you explicitly remove them.
- When the Docker host is not guaranteed to have a given directory or file structure. Volumes help you decouple 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 you need to back up, restore, or migrate data from one Docker host to another, volumes are a better choice. You can stop containers using the volume, then back up the volume's directory (such as /var/lib/docker/volumes/<volume-name>).

1.2. Bind mounts

Bind mounts have limited functionality compared to 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 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 allow access to sensitive files

One side effect of using bind mounts, for better or for worse, is that **you can change the host filesystem via processes running in a container,** including creating, modifying, or deleting important system files or directories. This is a powerful ability which can have security implications, including impacting non-Docker processes on the host system

1.2.1. use cases for bind mounts

- **Sharing configuration files** from the host machine to containers. This is how Docker provides DNS resolution to containers by default, 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 Project target/ directory into a container, and each time you build the 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.

1.3. tmpfs mounts

A tmpfs mount is not persisted 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.

1.3.1. use cases for tmpfs mount

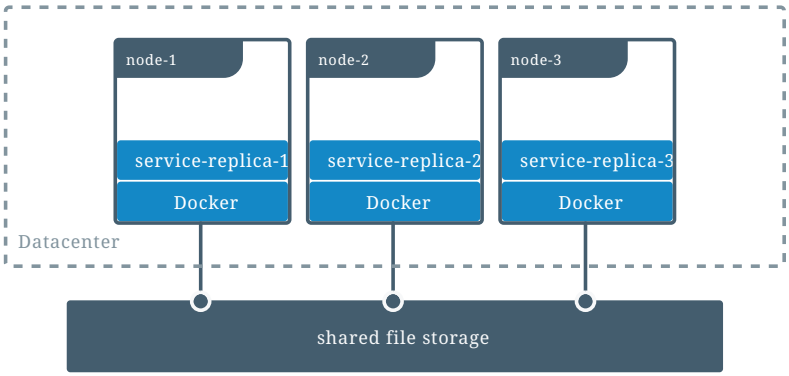
- tmpfs mounts are best used for cases when **you do not want the data to persist** either on the host machine or within the container. This may be **for security reasons or to protect the performance** of the container when your application needs to write a large volume of non-persistent state data.

keep the following in mind

- 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. Similarly, if you start a container and specify a volume which does not already exist, an empty volume is created for you. This is a good way to pre-populate data that another container needs.
- 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 contents of /mnt would be obscured by the contents of the USB drive until the USB drive were unmounted. The obscured files are not removed or altered, but are not accessible while the bind mount or volume is mounted.

2. Share data among machines

When building fault-tolerant applications, you need to configure multiple replicas of the same service to have access to the same files.



There are several ways to achieve this when developing your applications.

- One is to **add logic to your application** to store files on a cloud object storage system like **Amazon S3.**
- Another is to **create volumes** with a driver that supports writing files to an **external storage** system like **NFS** or **Amazon S3.**

Volume drivers

Volume drivers allow you to abstract the underlying storage system from the application logic.

2.1. External storage system

There are a lot of different systems to use.

We are going to look at the two most poppular systems (they both support any combination of the aforementioned ways).

A more detailed list of systems can be found at the bottom of the page.

2.1.1. MINIO

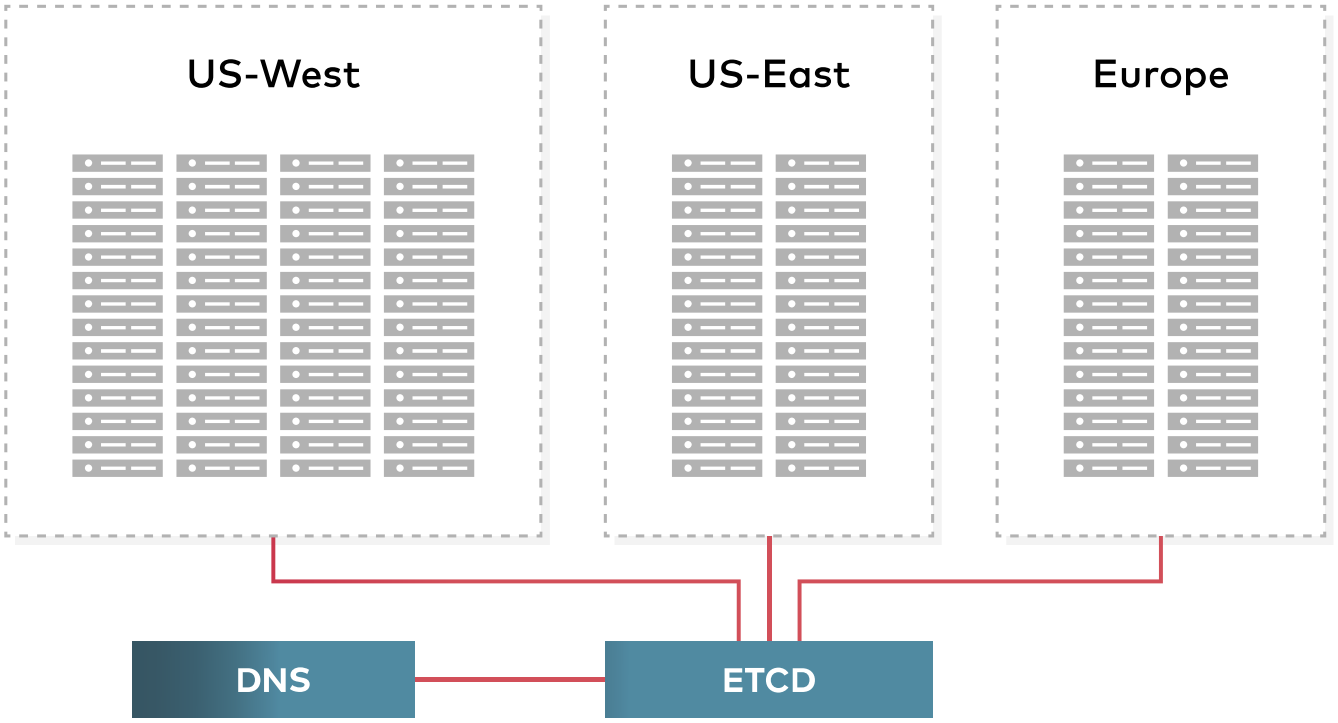
High Performance Object Storage,

open source

MinIO is 100% **open source under the Apache V2 license.** This means that MinIO's customers are free from lock in, free to inspect, free to innovate, free to modify and free to redistribute.

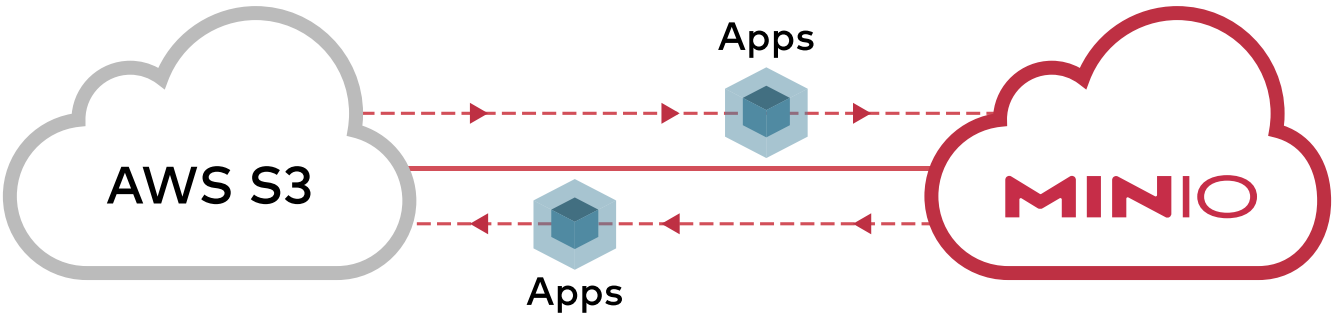
Scaling

At MinIO, scaling starts with a single cluster which can be federated with other MinIO clusters to create a global namespace, spanning multiple data centers if needed.



S3 compatibility

Amazon's S3 API is the defacto standard in the object storage world. MinIO is the defacto standard for S3 compatibility and was one of the first to adopt the API and the first to add support for S3 Select.



2.1.2. IPFS (Inter-Planetary File System)

A peer-to-peer hypermedia protocol, designed to make the web faster, safer, and more open.

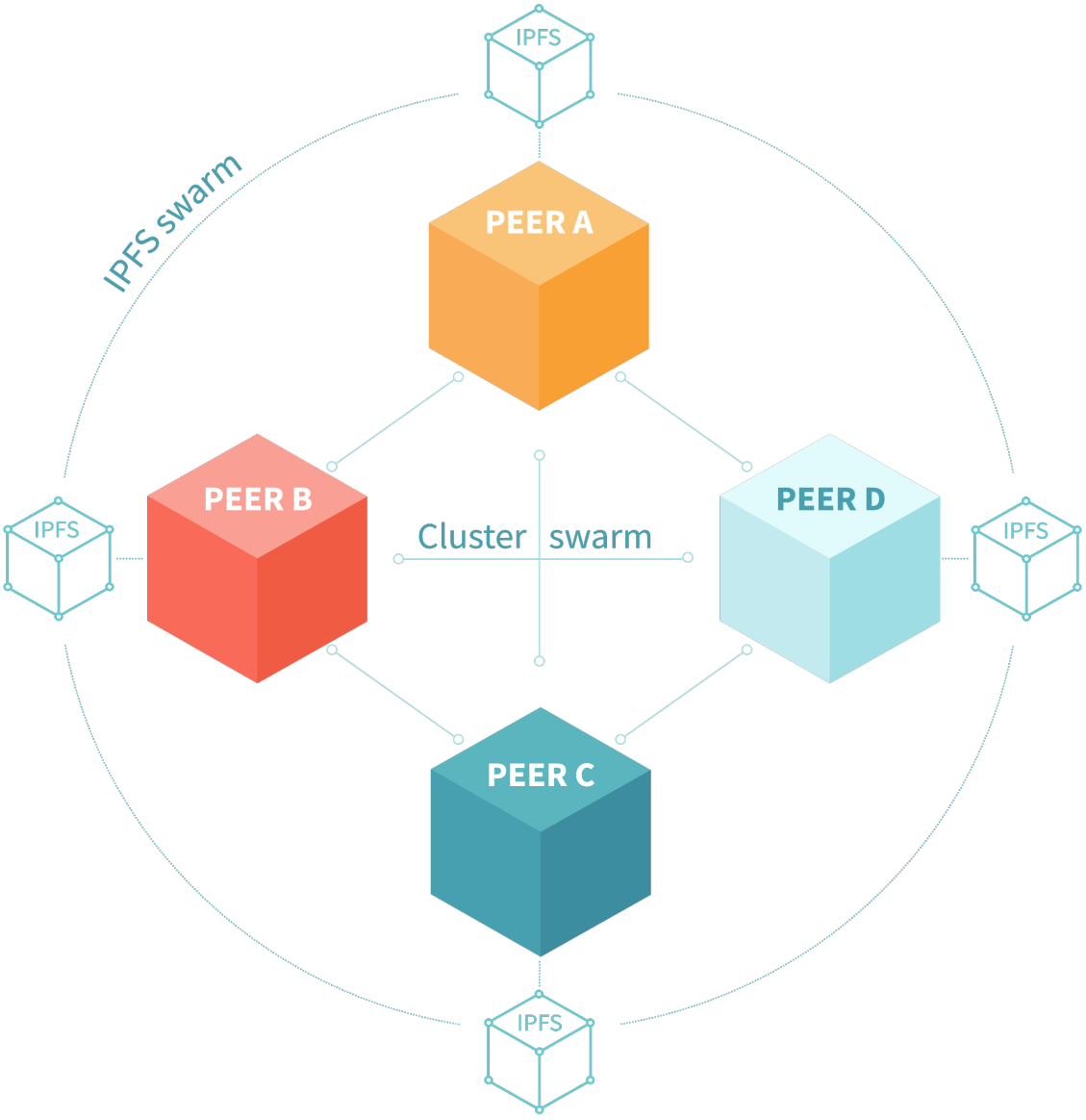
distributed system

IPFS is a distributed system for storing and accessing files, websites, applications, and data.

IPFS Cluster

Automated data availability and redundancy on IPFS

IPFS Cluster provides data orchestration across a swarm of IPFS daemons by allocating, replicating and tracking a global pinset distributed among multiple peers.



2.2. Extend docker

You can extend the capabilities of the Docker Engine by loading third-party plugins.

Docker supports **authorization, volume** and **network** driver plugins

2.2.1. available plugins

https://hub.docker.com/search?q=&type=plugin