Managing Data in Docker

Prepared by:
Ms. Avita Katal
Assistant Professor (SG)
School of Computer Science
UPES, Dehradun

Objectives

- What do you understand by managing data in Docker
- Learn about different ways of managing data and their use cases
  - Volumes
  - Bind mounts
  - tmps mount

Manage data in Docker

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 file system**. The storage driver provides a **union file system**, using the Linux kernel. This extra abstraction reduces performance as compared to using ***data volumes*****,** which **write directly to the host file system.**

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

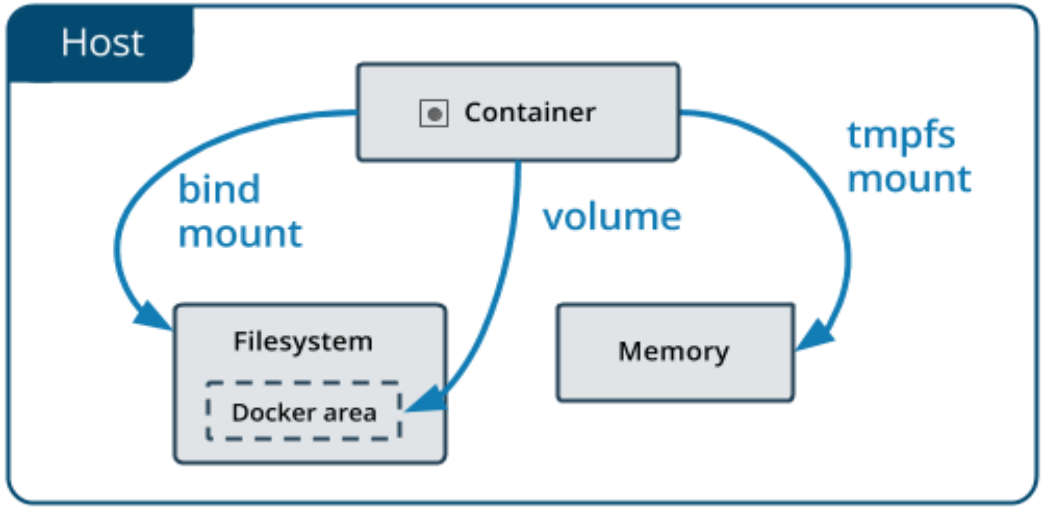
***volumes*, and *bind mounts*.**

Docker also supports containers **storing files in-memory on the host machine**. Such files are **not persisted.** If you're running **Docker on Linux**, ***tmpfs mount*** is used to store files in the host's system memory. If you're running **Docker on Windows**, ***named pipe*** is used to store files in the host's system memory.

Manage data in Docker

•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 file system.**

•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.**



- **Volumes** are stored in a part of the **host file system** which is *managed by Docker* **(/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the file** system. 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.**

Docker 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*.**

- When you mount a volume, it may **be named or anonymous**.
  - Anonymous volumes **are not given an explicit name** when they are first mounted into a container, so Docker gives **them a random name** that is guaranteed to be unique within a given Docker host. Besides the name, named and anonymous volumes behave in the same ways.

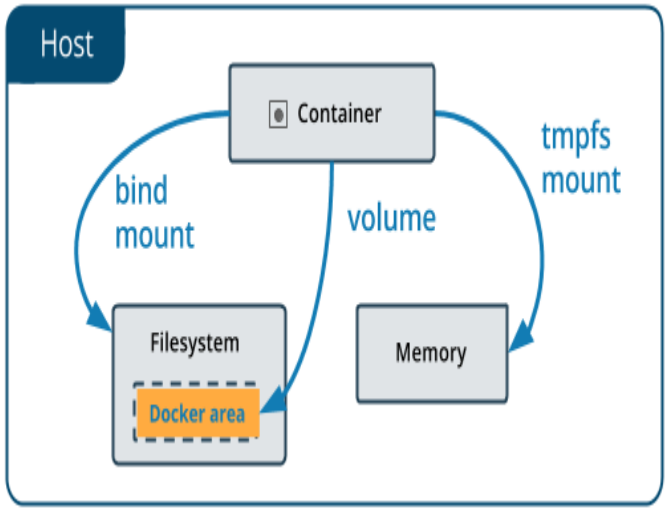
- Volumes also support the use of **volume drivers**, which allow you to store your data on **remote hosts or cloud providers**, among other possibilities.

Good use cases for volumes

- **Sharing data among multiple running containers.** 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>).
- When **your application requires high-performance I/O on Docker Desktop**. Volumes are stored in the **Linux VM rather than the host**, which means that the reads and writes have much lower latency and higher throughput.
- When **your application requires fully native file system** behavior on Docker Desktop.

Docker Volumes

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. **While 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 doesn't increase the size of the containers using it***, and the **volume's contents exist outside the lifecycle of a given container.**

Create and manage volumes

Unlike a bind mount, **you can create and manage volumes outside the scope of any container.**

Create a volume:

docker volume create my-vol

List volumes:

docker volume ls

Inspect a volume:

docker volume inspect my-vol

Remove a volume:

docker volume rm my-vol

Start a container with a volume

If you **start a container with a volume that doesn't yet exist, Docker creates the volume for you**. The following example mounts the volume myvol2 into /app/ in the container.

The **-v** and **--mount** examples below produce the same result. You can't run them both unless you remove the devtest container and the myvol2 volume after running the first one.

```
docker run -d \
  --name devtest \
  --mount source=myvol2,target=/app \
  nginx:latest
```

```
docker run -d \
  --name devtest \
  -v myvol2:/app \
  nginx:latest
```

Use docker inspect devtest to verify that Docker created the volume and it mounted correctly.

Stop the container and remove the volume. Note volume removal is a separate step.

docker container stop devtest
docker container rm devtest
docker volume rm myvol2

Bind Mounts

- Available since the **early days of Docker**. 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 file system** having a specific directory structure available.
- If you are developing new Docker applications, consider using **named volumes instead**. **You can't use Docker CLI commands to directly manage bind mounts.**

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** file system 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.**

Good use cases for bind mount

- **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 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. **If you use Docker for development this way, your production Dockerfile would copy the production-ready artifacts directly into the image, rather than relying on a bind mount.**

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

Start a container with bind mount

- Consider a case where you have a directory source and that when you build the source code, the artifacts are saved into another directory, **source/target/.**

- You want the artifacts to be available to the container at **/app/,** and you want the container to get access to a new build each time you build the source on your development host.

- Use the following command to bind-mount the **target/ directory** into your container at **/app/.** Run **the command from within the source directory**.

- The **$(pwd)** sub-command expands to the current working directory on Linux or macOS hosts. If you're on Windows, see also Path conversions on Windows.

The --mount and -v examples below produce the same result. You can't run them both unless you remove the devtest container after running the first one.

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

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

docker inspect devtest

to verify that the bind mount was created correctly. Look for the Mounts section.

Stop the container:

$ docker container stop devtest

$ docker container rm devtest

Tmps Mounts and Named Pipes

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.**

named pipes

- An npipe mount can be used for **communication between the Docker host and a container.**

- Common use case is to **run a third-party tool inside of a container and connect to the Docker Engine API using a named pipe.**

Good use cases for tmps mounts

- 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.**

Limitations for tmps mounts

- Unlike volumes and bind mounts, you **can't share tmpfs mounts between containers**.
- This functionality is only available **if you're running Docker on Linux.**
- Setting **permissions on tmpfs may cause them to reset after container restart.**

Use tmpfs mount in a container

- To use a tmpfs mount in a container, use the **--tmpfs flag**, or use the **--mount flag** with **type=tmpfs** and destination options.

- There is no source for **tmpfs mounts**. The following example creates a **tmpfs mount at /app** in a Nginx container.

- The first example uses the --mount flag and the second uses the --tmpfs flag.

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

```
docker run -d \
  -it \
  --name tmptest \
  --tmpfs /app \
  nginx:latest
```

Verify that the mount is a tmpfs mount by looking in the Mounts section of the docker inspect output:

Stop and remove the container:

$ docker stop tmptest

$ docker rm tmptest

Thank You