CCGC 5001 - Virtualization

Module 5: Storage and Volumes

HUMBER

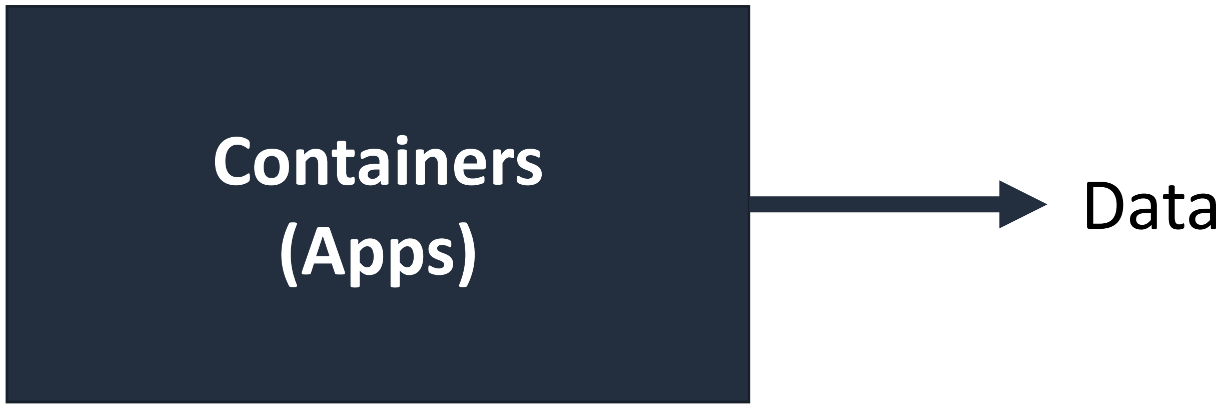
Module objectives

At the end of this module, you should be able to:

- Explain Docker storage
- Create and manage volumes
- Share data between multiple containers using data volumes
- Differentiate between volumes and bind mounts
- Define volumes in images

Storage overview

Where should application data be stored?



Issues with storing data inside containers

Containers are designed to be ephemeral (disposable)

When containers are stopped, data is not accessible

Containers are typically stored on each host

The container filesystem wasn’t designed for high performance I/O

Options for data storage with containers

Volumes

The recommended way to persist data, stored at /var/lib/docker/volumes/

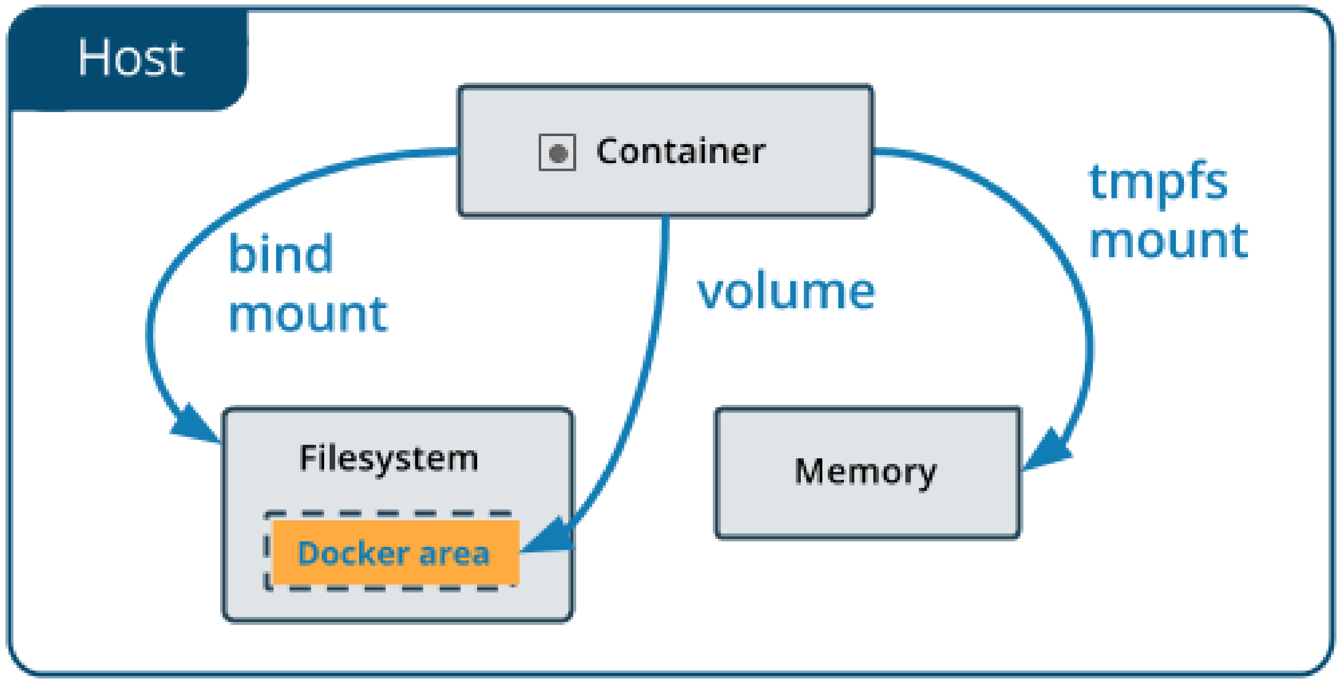
Bind Mounts

Have limited functionality and you must use the exact file path on the host; volumes recommended

Tmpfs mounts

Stored only in a host's memory in Linux (least recommended)

Options for data storage with containers



Block vs. object storage

Block

- Fixed chunks of data
- No metadata is stored
- Best for I/O intensive apps
- SAN storage uses block storage protocols like iSCSI

Container data is written as Block.

Object

- Data is stored with metadata and a unique identifier
- There is no organization to the objects
- Scalability is limitless
- Accessed with HTTP calls

Images in Docker registry are written as Objects.

Modifying the container layer

Here an application that changes the something in the filesystem of the container.

```
$ docker container run --name demo alpine /bin/sh -c 'echo "Today is Monday" > sample.txt'
```

What will happen if we remove the container?

Volumes

Volumes are the preferred mechanism for persisting data generated by and used by containers.

Advantages:

- Easier to backup or migrate than bind mounts
- Manage volumes using Docker CLI or API
- Work on both Linux or Windows containers
- Volumes can be shared among multiple containers
- Store volumes on remote hosts or cloud
- Volumes can have their content pre-populated by a container

Create and manage volumes

You can create and manage volumes outside the scope of container.

Create a volume:

```
$ docker volume create my-vol
```

List volumes:

```
$ docker volume ls
```

Remove a volume:

```
$ docker volume rm my-vol
```

Create and manage volumes

Inspect a volume:

```
$ docker volume inspect my-vol

[
    {
        "CreatedAt": "2021-03-14T19:26:40Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-vol/_data",
        "Name": "my-vol",
        "Options": {},
        "Scope": "local"
    }
]
```

Target folder is protected and requires escalated privileges.

Mounting a volume

We can mount named volumes into a container using –v option.

```
$ docker container run --name test -it -v my-vol:/data alpine /bin/sh
```

Inside the container, we can create files in the /data folder.

```
/ # cd /data
/data # echo "some data" > data.txt
/data # echo "some more data" > data2.txt
```

We can verify the data files from Docker host.

We can also create files in this /data folder from the host.

Sharing data between containers

Application running in container that produces some data can be consumed by another application running in different container.

How can we achieve this?

```
$ docker container run -it --name writer -v shared-data:/data alpine /bin/sh
/ # echo "I can create a file" > /data/sample.txt
/# exit

$ docker container run -it --name reader -v shared-data:/app/data:ro ubuntu:19.04 /bin/bash
/# ls -l /app/data
-rw-r--r-- 1 root root 20 Mar 14 20:42 sample.txt

/# echo "Try to break read only" > /app/data/data.txt
bash: /app/data/data.txt: Read-only file system
```

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 *absolute path* on the host machine.
- Bind mounts are very performant, but they rely on the host machine’s filesystem having a specific directory structure available.

An example

```
$ mkdir my-web && cd my-web
$ echo "<h1>Personal Website</h1>" > index.html
```

Let’s create a Dockerfile.

```
FROM nginx:alpine
COPY . /usr/share/nginx/html
```

Build the image.

```
$ docker image build -t my-website:1.0 .
```

Run the container.

```
$ docker container run -d --name my-website -v $(pwd):/usr/share/nginx/html -p 8080:80 my-website:1.0
```

Choosing the –v or --mount flag

-v syntax combines all the options together in one field, while the **--mount** syntax separate them.

-v or --volume: Consists of three fields, separated by colon characters (:)

Choosing the –v or --mount flag

--mount: Consists of multiple key-value pairs, separated by commas and each consisting of a <key>=<value> tuple.

- The **type** of the mount, which can be **bind**, **volume**, or **tmpfs**.
- The **source** of the mount. For bind mounts, this is the path to the file or directory on the Docker host. May be specified as **source** or **src**.
- The **destination** takes as its value the path where the file or directory is mounted in the container. May be specified as destination, **dst**, or **target**.
- The **readonly** option, if present, causes the bind mount to be mounted into the container as read-only.
- The **--mount** flag does not support z or Z options for modifying selinux labels.

Start a container with a bind mount

Using --mount:

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

Using -v or volume:

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

Have you noticed any differences between –v and --mount behavior?

```
 "Mounts": [
        {
            "Type": "bind",
            "Source": "/home/ec2-user/target",
            "Destination": "/app",
            "Mode": "",
            "RW": true,
            "Propagation": "rprivate"
        }
    ]
```

Defining volumes in images

Volumes can be defined in Dockerfile. The KEYWORD to do so is VOLUME.

```
VOLUME /app/data
VOLUME /app/data, /app/profiles, /app/config
VOLUME ["/app/data", "/app/profiles", "/app/config"]
```

Module summary

In summary, in this module, you learned:

- Docker volumes and how to manage them
- Sharing data between containers
- Techniques of defining volumes such as by names, by mounting a host directory, or by defining volumes in a container image

Thank you