CONTAINER STORAGE

Overview

In this lab, we will illustrate the concept of volumes and configuration. We will learn how to use volume

- in a Dockerfile
- at runtime with the -v option
- using the volume API

Procedure

Data persistency without a volume

We will first illustrate how data is **not** persisted outside of a container by default.

- Let's run an interactive shell within an alpine container named alpha.

  docker container run --name alpha -it alpine sh

- We will create the /data folder and a dummy hello.txt file in it.

  mkdir /data && cd /data && touch hello.txt

We will then check how the read-write layer (container layer) is accessible from the host.

- Exit the container first.
- Inspect the container to get the location of the container's layer. We can use the inspect command and then scroll into the output until the GraphDriver key.

  docker container inspect alpha

  Alternatively, we can directly use the Go template notation and get the content of the GraphDriver keys right away.

  docker container inspect -f "{{ json .GraphDriver }}" alpha | jq

Note: you may have to install jq application on your docker host.

From our host, inspect the folder for which path is specified in UpperDir, we can see our /data and the hello.txt file we created are there.

- Use the below command to see the contents of the /data folder:

  ls /var/lib/docker/overlay2/[YOUR_ID]/diff/data

What happen if we remove our alpha container now?

docker container rm alpha

Examine the content of our /data folder. Explain your finding and justify your answer.

Defining a volume in a Dockerfile

Let's examine how volumes come into the picture to handle the data persistency.

- Create a Dockerfile based on alpine and define the /data as a volume. This means that anything written by a container in /data will be persisted outside of the Union filesystem.

  FROM alpine

  VOLUME ["/data"]

  ENTRYPOINT ["/bin/sh"]

  Note: we specify /bin/sh as the ENTRYPOINT so that if no command is provided in interactive mode we will end up in a shell inside our container.

- Build an image from this Dockerfile.

  docker image build -t img1 .

- Create a container in interactive mode (using -it flags) from this image and name it beta.

  docker container run --name beta -it img1

Now, we should end up in a shell within the container.

- Go into /data and create a hello.txt file.

  cd /data
  touch hello.txt
  ls

- Exit the container making sure it remains in running state: use the Control-P / Control-Q combination for this. Alternatively, you can work with second terminal.
- Verify that your container is running.

  docker container ls

We will now inspect this container in order to get the location of the volume (defined on /data) on the host.

- Use the inspect command and then scroll into the output until we find the Mounts key

  docker container inspect beta

  Alternatively, we can directly use the Go template notation and get the content of the Mounts keys right away.
  docker container inspect -f "{{ json .Mounts }}" beta | jq

The output will show that the volume defined in /data is stored in /var/lib/docker/volumes/……../_data on the host.

- Verify that your hello.txt file is present on the host by copying your own path (the one under the Source key).

Were you able to locate your hello.txt file?

- We now examine the file by removing the beta container.

  docker container stop beta && docker container rm beta

Were you able to locate your hello.txt file and explain your finding?

Defining a volume at runtime

We have seen a volume defined in a Dockerfile, we will see that volume can also be defined at runtime using the -v flag of the docker container run command.

- Create a container from the alpine image, we will use the -d option so it runs in background and also define a volume on /data as we've done previously. In order the PID 1 process remains active, we will use ping command to ping Google DNS and log the output in a file within the /data folder.

  docker container run --name zeta -d -v /data alpine sh -c 'ping 8.8.8.8 > /data/ping.txt'

- Inspect the container and get the Mounts key using the Go template notation.

  docker container inspect -f "{{ json .Mounts }}" zeta | jq

The output will be pretty much the same as we had when we defined the volume in the Dockerfile.

- Use the folder defined in the Source key, and check the content of the ping.txt within the /data folder.

What do you see in ping.txt file? Is that what you had expected?

- We now examine the file by removing the zeta container.

  docker container stop zeta && docker container rm zeta

Were you able to locate ping.txt file? What was its content and justify your finding?

Usage of the Volume API

The volume API allows us to create and perform operation on volume very easily.

- Create a volume named html.

  docker volume create --name html

- List the existing volume, our html volume should be the only one.

  docker volume ls

In the volume API, like for almost all the other Docker's API, there is an inspect command available to us.

- Use inspect command against the html volume.

  docker volume inspect html

The **Mountpoint** defined in the output is the path on the Docker host where the volume can be accessed.

What can you tell about this Mountpoint?

We can now use this volume and mount it on a specific path of a container.

- Use a Nginx image and mount the **html** volume **onto /usr/share/nginx/html** folder within the container.

  docker container run --name www -d -p 8080:80 -v html:/usr/share/nginx/html nginx

  Note: /usr/share/nginx/html is the default folder served by nginx. It contains 2 files: index.html and 50x.html

  Note: we use the -p option to map the nginx default port (80) to a port on the host (8080).

- From the host, examine the content of the volume.

  ls /var/lib/docker/volumes/html/_data

What do you find in the content of the volume?

From our host, we can now modify the index.html file and verify the changes are considered within the container.

- Modify the index.html file in your volume to include your name and student id.
- Verify that changes you have done in the index.html are reflected on your nginx page.

  Note: please reload the page if you cannot see the changes.

Mount host's folder into a container

Lastly, we will examine bind-mount and they allow us mounting a host's folder into a container's folder. This is done using the -v option of the docker container run command. Instead of specifying one single path (as we did when defining volumes) we will specify 2 paths separated by a column.

docker container run -v HOST_PATH:CONTAINER_PATH [OPTIONS] IMAGE [CMD]

Note: HOST_PATH and CONTAINER_PATH can be a folder or file. None of the Paths have to exist before starting the Container as they will be created automatically during the start.

First case:

- Let's run an alpine container bind mounting the local /tmp folder inside the container /data folder.

  docker container run -it -v /tmp:/data alpine sh

  We end up in a shell inside our container. By default, there is no /data folder in an alpine distribution.

What is the impact of the bind-mount?

- Examine the /data folder in your container.

ls /data

The /data folder has been created inside the container and it contains the content of the /tmp folder of the host. We can now, from the container, change files on the host and the other way round.

Second case:

- Let's run a nginx container bind mounting the local /tmp folder inside the /usr/share/nginx/html folder of the container.

  docker container run -it -v /tmp:/usr/share/nginx/html nginx bash

  Are the default index.html and 50x.html files still there in the container's /usr/share/nginx/html folder?

Bind-mounting is very useful in development as it enables, for instance, to share source code on the host with the container.