Docker Volumes

Jan 25, 2017 • @lucjuggery

Understanding what is this volume thing

In this lab, we will illustrate the concept of volume. We will see how to use volume

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

We will also see what is bind-mounting on a simple example.

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 c1.

docker container run --name c1 -ti 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.

Let exit the container first.

exit

Let’s inspect our container in order 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, like the following.

docker container inspect c1

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

docker container inspect -f "{{ json .GraphDriver }}" c1 | python -m json.tool

You should then get an output like the following (the ID will not be the same though)

{

"Data": {

"LowerDir": "/var/lib/docker/overlay2/55922a6b646ba6681c5eca253a19e90270e3872329a239a82877b2f8c505c9a2-init/diff:/var/lib/docker/overlay2/30474f5fc34277d1d9e5ed5b48e2fb979eee9805a61a0b2c4bf33b766ba65a16/diff",

"MergedDir": "/var/lib/docker/overlay2/55922a6b646ba6681c5eca253a19e90270e3872329a239a82877b2f8c505c9a2/merged",

"UpperDir": "/var/lib/docker/overlay2/55922a6b646ba6681c5eca253a19e90270e3872329a239a82877b2f8c505c9a2/diff",

"WorkDir": "/var/lib/docker/overlay2/55922a6b646ba6681c5eca253a19e90270e3872329a239a82877b2f8c505c9a2/work"

},

"Name": "overlay2"

}

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

Try 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 c1 container now ? Let’s try.

docker container rm c1

It seems the folder defined in the **UpperDir** above does not exist anymore. Do you confirm that ? Try running the ls command again and see the results.

This shows that data created in a container is not persisted. It’s removed with the container’s layer when the container is deleted.

Defining a volume in a Dockerfile

We will now see how volumes come into the picture to handle the data persistency.

We will start by creating 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.

Create a Dockerfile with the following content

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.

Let’s build an image from this Dockerfile.

docker image build -t img1 .

We will then create a container in interactive mode (using -ti flags) from this image and name it c2.

docker container run --name c2 -ti img1

We should then end up in a shell within the container. From there, we will go into /data and create a hello.txt file.

cd /data

touch hello.txt

ls

Let’s exit the container making sure it remains running: use the Control-P / Control-Q combination for this. Use the following command to make sure it’s still running.

docker container ls

Note: the container, named c2, should be listed there.

We will now inspect this container in order to get the location of the volume (defined on /data) on the host. We can use the inspect command and then scroll into the output until we find the **Mounts** key…

docker container inspect c2

Or we can directly use the Go template notation and get the content of the **Mounts** keys right away.

docker container inspect -f "{{ json .Mounts }}" c2 | python -m json.tool

You should then get an output like the following (the ID will not be the same though)

[

{

"Destination": "/data",

"Driver": "local",

"Mode": "",

"Name": "2f5b7c6b77494934293fc7a09198dd3c20406f05272121728632a4aab545401c",

"Propagation": "",

"RW": true,

"Source": "/var/lib/docker/volumes/2f5b7c6b77494934293fc7a09198dd3c20406f05272121728632a4aab545401c/\_data",

"Type": "volume"

}

]

This output shows that the volume defined in /data is stored in **/var/lib/docker/volumes/2f5…01c/\_data** on the host (removing part of the ID for a better readability).

Copy your own path (the one under the **Source** key) and make sure the **hello.txt** file we created (from within the container) is there.

We now remove the c2 container.

docker container stop c2 && docker container rm c2

Check that the folder defined under the **Source** key is still there and contains **hello.txt** file.

From the above, we can see that a volume bypasses the union filesystem and is not dependent on a container’s lifecycle.

Defining a volume at runtime

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

Let’s create a container from the alpine image, we’ll 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 use the following command that pings Google DNS and log the output in a file within the /data folder.

ping 8.8.8.8 > /data/ping.txt

The container is ran that way:

docker container run --name c3 -d -v /data alpine sh -c 'ping 8.8.8.8 > /data/ping.txt'

Let’s inspect the container and get the **Mounts** key using the Go template notation.

docker container inspect -f "{{ json .Mounts }}" c3 | python -m json.tool

We have pretty much the same output as we had when we defined the volume in the Dockerfile.

[

{

"Type": "volume",

"Name": "af621cde2717307e5bf91be850c5a00474d58b8cdc8d6e37f2e373631c2f1331",

"Source": "/var/lib/docker/volumes/af621cde2717307e5bf91be850c5a00474d58b8cdc8d6e37f2e373631c2f1331/\_data",

"Destination": "/data",

"Driver": "local",

"Mode": "",

"RW": true,

"Propagation": ""

}

]

If we use the folder defined in the **Source** key, and check the content of the ping.txt within the /data folder, we get something similar to the following.

tail -f /var/lib/docker/volumes/OUR\_ID/\_data/ping.txt

64 bytes from 8.8.8.8: seq=34 ttl=37 time=0.462 ms

64 bytes from 8.8.8.8: seq=35 ttl=37 time=0.436 ms

64 bytes from 8.8.8.8: seq=36 ttl=37 time=0.512 ms

64 bytes from 8.8.8.8: seq=37 ttl=37 time=0.487 ms

64 bytes from 8.8.8.8: seq=38 ttl=37 time=0.409 ms

64 bytes from 8.8.8.8: seq=39 ttl=37 time=0.438 ms

64 bytes from 8.8.8.8: seq=40 ttl=37 time=0.477 ms

...

The ping.txt file is updated regularly by the command running in the **c3** container.

Stopping and removing the container will obviously stop the ping command but the /data/ping.txt file will still be there. Give it a try :)

Usage of the Volume API

The volume API introduced in Docker 1.9 enables to perform operations on volume very easily.

First have a look at the commands available in the volume API.

docker volume --help

We will start with the create command, and create a volume named **html**.

docker volume create --name html

If we list the existing volume, our **html** volume should be the only one.

docker volume ls

The output should be something like

DRIVER VOLUME NAME

[other previously created volumes]

local html

In the volume API, like for almost all the other Docker’s API, there is an **inspect** command. Let’s use it against the **html** volume.

docker volume inspect html

The output should be the following one.

[

{

"Driver": "local",

"Labels": {},

"Mountpoint": "/var/lib/docker/volumes/html/\_data",

"Name": "html",

"Options": {},

"Scope": "local"

}

]

The **Mountpoint** defined here is the path on the Docker host where the volume can be accessed. We can note that this path uses the name of the volume instead of the auto-generated ID we saw in the example above.

We can now use this volume and mount it on a specific path of a container. We will use a Nginx image and mount the **html** volume onto **/usr/share/nginx/html** folder within the container.

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

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

Note: we use the -p option to map the nginx default port (80) to a port on the host (8080). We will come back to this in the lesson dedicated to the networking.

From the host, let’s have a look at the content of the volume.

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

The content of the **/usr/share/nginx/html** folder of the **www** container has been copied into the **/var/lib/docker/volumes/html/\_data** folder on the host.

Let’s have a look at the nginx’s [welcome page](https://training.play-with-docker.com/)

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

cat<<END >/var/lib/docker/volumes/html/\_data/index.html

SOMEONE HERE ?

END

Let’s have a look at the nginx’s [welcome page](https://training.play-with-docker.com/). We can see the changes we have done in the index.html.

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

Mount host’s folder into a container

The last item we will talk about is named bind-mount and consist of 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 specified 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. HOST\_PATH must exist before running this command.

Several cases to consider:

* the CONTAINER\_PATH does not exist within the container
* the CONTAINER\_PATH exists within the container

1st case

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

docker container run -ti -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 ?

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.

2nd 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 -ti -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 ?

ls /usr/share/nginx/html

No ! The content of the container’s folder has been overridden with the content of the host folder.

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