Volumes

Creating and mounting data volumes

Modifying the container layer

Before we dive into volumes, let's first discuss what happens if an application in a container changes something in the filesystem of the container. In this case, the changes are all happening in the writable container layer. Let's quickly demonstrate this by running a container, and execute a script in it that is creating a new file:

```
docker container run --name demo alpine /bin/sh -c 'echo "This is a test" > sample.txt'
```

Let's use the `diff` command to find out what has changed in the container's filesystem in relation to the filesystem of the original image:

```
docker container diff demo
```

The output should look like this:

```
A /sample.txt
```

If we now remove the container from memory, its container layer will also be removed, and with it, all the changes will be irreversibly deleted. If we need our changes to persist even beyond the lifetime of the container, this is not a solution, Luckily, we have better options, in the form of Docker volumes. Let's get to know them.

Creating volumes

Since at this time, when using Docker for Desktop on a macOS or Windows computer, containers are not running natively on macOS or Windows but rather in a (hidden) VM created by Docker for Desktop, for illustrative purposes it is best we use `docker-machine` to create and use an explicit VM running Docker.

- **Prepare**

> Install Docker Machine
>
> docker machine and docker engine
>
> err.....if install VirtualBox error in macOS

- Follow these steps:

```
# 1. list all VMs currently running in VirtualBox
docker-machine ls

# 2. create a VM with the following command
docker-machine create --driver virtualbox node-1

# 3. If you have a VM called `node-1` but it is not running, then please
start it
docker-machine start node-1

# 4. use SSH into this VM
docker-machine ssh node-1

# 5. the following are all executed inside the VM
# create a volume
docker volume create sample

# 6. the easiest way to find out where the data is stored on the host is by
using the `inspect` command
docker volume inspect sample

# The output of preceding command is like this:
[
    {
        "CreatedAt": "2020-07-01T03:04:54Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/mnt/sda1/var/lib/docker/volumes/sample/_data",
        "Name": "sample",
        "Options": {},
        "Scope": "local"
    }
]
```

The target folder is often a protected folder, and we thus might need to use *sudo* to navigate to this folder and execute any operations in it. On our LinuxKit–based VM in Docker Machine, access is also denied, yet we don't have *sudo* available either. Luckily, I have prepared a *fundamentalsofdocker/nsenter* utility container that allow us to access that backing folder of our *sample* volume we created earlier.

> `docker run -it --privileged --pid=host fundamentalsofdocker/nsenter`
>
> We are running the container with the `--privileged` flag. This means that any app running in the container gets access to the devices of the host. The `--pid=host` flag signifies that the container is allowed to access the process tree of the host (the hidden VM in which the Docker daemon is running). Now the preceding container runs the Linux `nsenter` tool to enter the Linux namespace of the host and then runs a shell within there, From this shell, we are thus granted access to all resources managed by the host.
>
> When running the container, we basically execute the following command inside the container:
> `nsenter -t 1 -m -u -n -i sh`
>
> If that sounds complicated to you, don't worry; you will understand more as we proceed through this book. If there is one takeaway for you out of this, then it is to realize how powerful the right use of containers can be.
>
> Further reading:
> pid—setting privileged

From within this container, we can now navigate to the folder representing the mount of volume, and then list its content:

> `cd /mnt/sda1/var/lib/docker/volumes/sample/_data && ls -l`

Exit the tool container by pressing Ctrl+D.

Mounting a volume

Once we have created a named volume, we can mount it into a container by following these steps:

```
# mounts the `sample` volume to the `/data` folder inside the container.
docker container run --name test -it -v sample:/data alpine /bin/sh

# inside the container, we can now create files in the `/data` folder and
then exit:
cd /data && echo "Some data" > data.txt

# If we nevigate to the host folder that containers the data of the volume
and list
# its content, we should see the file we created in last step.
```

```
# remember, we need to use the `fundamentalsofdocker/nsenter` tool container
to do so
docker run -it --rm --privileged --pid=host fundamentalsofdocker/nsenter
cd /mnt/sda1/var/lib/docker/volumes/sample/_data && ls -l

# Let's try to create a file in this folder from the host, and then use the
volume
# with another container.
echo "This file we create on the host" > host-data.txt

# exit the tool container by pressing `Ctrl+D`.
# now let's delete the `test` container and mount our volume to a different
container.
docker container rm test
docker container run --name test2 -it -v sample:/app/data centos:7 /bin/bash

# as expected, we should see two files. `data.txt` and `host-data.txt`
cd /app/data && ls -l
```

It is important to note that the folder inside the container to which we mount a Docker volume is excluded from the Union filesystem. That is, each change inside this folder and any of its subfolders will not be part of the container layer, but will persisted in the in the backing storage provided by the volume driver. This fact is really important since the container layer is deleted when the corresponding container is stopped and removed from the system.

Removing volumes

> *docker volume rm sample*

To remove all running containers in order to clean up the system, run the following command:

> *docker container rm -f $(docker container ls -aq)*

Accessing volumes created with Docker for Desktop

```
# create a `sample` volume and inspect it using Docker for Desktop on macOS
or Windows.
docker volume create sample
docker volume inspect smaple

# `inspect` output is like this:
[
    {
```

```
        "CreatedAt": "2020-07-01T07:59:06Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/sample/_data",
        "Name": "sample",
        "Options": {},
        "Scope": "local"
    }
]

# you will discover that there is no `Mountpoint` folder on your macOS or
Windows.
# this reason is that the path shown is in relation to the hidden VM that
the Docker
# for macOS uses to run ocntainers. At this time, Linux containers cannot
run natively
# on macOS, nor on Windows.

# generate files with data in the volume from within an `alpine` container.
docker container run --rm -it -v sample:/data alpine /bin/sh
echo "Hello World!" > /data/sample.txt

# Exit `alpine` container by pressing `Ctrl+D`

# To access the hidden VM from our macOS, we have two options. We can either
# use a special container and run it in privileged mode, or we can use the
# `screen` utility # to screen into the Docker driver.
# The first method is also applicable to Docker for Windows.

# first method:
docker run -it --rm --privileged --pid=host fundamentalsofdocker/nsenter
cd /var/lib/docker/volumes/sample/_data && ls -l

# second method [**Note: I failed in this step even if i followed the
book**]:
screen
~/Library/Containers/com.docker.docker/Data/tasks/com.docker.driver.amd64-
linux/tty
# by doing so, we will be greeted by an empty screen. Hit *Enter*, and a
# `docker-desktop:~#` command-line prompt will be displayed.
# we can now navigate to the volume folder.
docker-desktop:~'#' cd /var/lib/docker/volumes/sample/_data
# to exit this session with the Docker VM, press `Ctrl+A+K`
```

Sharing data between containers

To avoid concurrency problems such as race conditions, we ideally have only one application or process that is creating or modify data, while all other process concurrently accessing this data only read it. We can

enforce a process running in a container to only be able to read the data in a volume by mounting this volume as read-only. Have a look at the following command:

> docker container run -it --name writer -v shared-data:/data:ro alpine /bin/sh

ro here represent **read-only**.

Using host volumes

In certain scenarios, such as when developing new containerized applications or when a containerized application needs to consume data from a certain folder produced—say—by a legacy application, it is very useful to use volumes that mount a specific host folder.

> docker container run --rm -it -v $(pwd)/src:/app/src alpine:latest /bin/sh

Developers use these techniques all the time when they are working on their application that runs in a container, and want to make sure that the container always contains the latest changes they make to the code, without the need to rebuild the image and return the container after each change.

```
# create a simple web page:
mkdir ~/my-web && cd ~/my-web
echo "<h1>Personal Website</h1>" > index.html

# add a file called `Dockerfile` to the folder with this content:
FROM nginx:alpine
COPY . /usr/share/nginx/html

# build image:
docker image build -t my-website:1.0 .

# run a container from this image:
docker container run -d --name my-site -v $(pwd):/usr/share/nginx/html \
-p 8080:80 \
my-website:1.0

# Now append more content to the `index.html` and refresh the browser
# `localhost:8080/index.html`, You should see the changes.
```

We also call this an edit-and-continue experience. You can make as many changes in your web files and always immediately see the result in the browser, without having to rebuild the image and restart the container containing your website.

Defining volumes in images

Some applications, such as databases running in containers, need to persist their data beyond the lifetime of the container. In this case, they can use volumes.

There is a way of defining volumes in the Dockerfile. The keyword is VOLUME, and we can either add the absolute path to a single folder or a comma-separated list of paths. These paths represent folders of the container's filesystem. Let's look at a few sample of such volume definitions:

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

We can use the docker image inspect command to get information about the volumes defined in the Dockerfile. Let's see what MongoDB gives us by following these steps:

First, we pull the image with the following command:

```
docker image pull mongo:3.7
```

Then, we inspect this image, and use the --format parameter to only extract the essential part from the massive amount of data:

```
docker image inspect --format='{{json .ContainerConfig.Volumes}}' mongo:3.7 | jq .
```

Now, let's run an instance of MongoDB in the background as a daemon, as follows:

```
docker run --name my-mongo -d mongo:3.7
```

We can now use the docker container inspect command to get information about the volumes that have been created, among other things:

```
docker container inspect --format='{{json .Mounts}}' my-mongo | jq .
```

Configuring containers

Defining environment variables for containers

We can actually pass some configuration values into the container at start time. We can use the `--env` (or the short form, `-e`) parameter in the form `--env key=value` to do so.

> `docker container run --rm -it --env LOG_DIR=/var/my-log alpine /bin/sh`

We can, of course, define more than just one environment variable when we run a container. We just need to repeat the `--env` parameter.

> `docker container run --rm -it -e LOG_DIR=/var/my-log -e MAX_LOG=5 alpine /bin/sh`

Using configuration files

Complex applications can have many environment variables to configure, and thus our command to run the corresponding container can quickly become unwieldy. For this purpose, Docker allows us to pass a collection of environment variable definitions as a file with the parameter `--env-file`.

```
# create a file called `development.config` in your folder
mkdir -p ~/fod/config && cd ~/fod/config && vim development.config

# Add the following content to the file and save it
LOG_DIR=/var/my-log
MAX_LOG=5

# run a container to verify that the variables have indeed been
# created as environment variables
docker container run --rm -it --env-file ./development.config alpine sh -c
"export"
```

Defining environment variables in container images

Sometimes, we want to define some default value for an environment variable that must be present in each container instance of a given container image. We can do so in `Dockerfile` like this:

```
# create a Dockerfile with the following contents
FROM alpine:latest
ENV LOG_DIR=/var/my-log
ENV MAX_LOG=5

# create a container image using the preceding Dockerfile
docker image build -t my-alpine .
```

```
# run a container instance from this image
docker container run --rm -it my-alpine sh -c "export | grep LOG"
```

The good thing, though, is that we are not stuck with those variable values at all. We can override one or many of them, using the *--env* parameter. We can also override default values using environment files together with the *--env-file* parameter in the *docker container run* command.

Environment variables at build time

Sometimes, we would want to have the possibility to define some environment variables that are valid at the time when we build a container image. Imagine that you want to define a *BASE_IMAGE_VERSION* environment variable that shall then be used as a parameter in your *Dockerfile*. Imagining the following *Dockerfile*:

```
ARG BASE_IMAGE_VERSION=12.7-stretch
FROM node:${BASE_IMAGE_VERSION}
WORKDIR /app
COPY packages.json .
RUN npm install
COPY . .
CMD npm start
```

We are using the *ARG* keyword to define a default value that is used each time we build an image from the preceding *Dockerfile*. Now, if we want to create a special image for—say—testing purposes, we can override this variable at image build time using the *--build-arg* parameter:

> *docker image build --build-arg BASE_IMAGE_VERSION=12.7-alpine -t my-node-test .*

To summarize, environment variables defined via *--env* or *--env-file* are valid at container runtime. Variables defined with *ARG* in the *Dockerfile* or *--build-arg* in the *docker image build* command are valid at container image build time. The former are used to configure an application running inside a container, while the latter are used to parametrize the container image build process.

Further reading: *ARG*