PERSISTENT STORAGE

Docker volumes

Understanding union filesystems

- If you have a look at the output of docker images –a, you will see a number of images have no name (you'll see "none" instead of their names). They can be referred to only by the image ID.
- To understand the usage of those images, you have to know that when you download an "image" from a repository, you are not downloading just one file. Instead, you're pulling a number of images that have a parent-child relation. For example, an ubuntu:16.04 is built upon a number of parent images. Those images contain common libraries and dependencies for the child image (Ubuntu).
- The advantage of this model is that common layers needn't be reinstalled. This means that – for example – you can download several versions of the same applications without having to redeploy its common dependencies.
- However union filesystems – as any other filesystem type – is not the answer for every use-case. For example, it does not support the extended file attributes that SELinux requires to operate correctly. For those weaknesses, users work with a Docker feature called "volumes".

Why volumes?

- So far we have run containers out of Docker images on our system, but those were just examples.
- In a container like WordPress, if you are planning to test that in a "live" environment, where users will actually log in and start writing content. Data is stored in the MySQL database container linked to it. But where is this data physically stored? What if you want to access this data? What is you want to upgrade the database engine?
- How can you view the the logs that the application writes to troubleshoot any problems? This is the role of volumes.
- Volumes are simply "mount points" to the respective filesystems on the host OS.
- The container's internal filesystem (union filesystem) provides those mount points.

When to use volumes?

- By design, volumes were introduced to serve containers that need to write a lot of data to persistent storage. Accordingly, volumes should be used when working with:
  - *Database data rather than the engine that processes it.*
  - *The log data of a web application rather than the application itself.*
  - *Dynamically-changing data (like in a CMS) rather than static websites.*
- Storing data in volumes helps make your application more modular. It also helps set the "separation of concerns" principle correctly. You can easily change the application that deals with the data (the interface) while keeping the data intact. This saves time, effort, and builds a less-error-prone model.
- Volumes also act as a "controlled window" between the host and the container. Consider for an example a host that uses SSD disks specifically for database access. Without volumes, Docker does not (and should not) have any idea about the underlying disk type or speed where the data is stored. But the user that operates Docker can control this by using volumes.

LAB: using volumes to create persistent data

- In this lab we are going to demonstrate the use of volumes by creating a shared volume that will be used by two MySQL databases to share the same data.
- First, create a container that will hold that volume:
```
docker run -d --volume /var/lib/mysql --name mysql-shared busybox echo Hello, I am the volumes container
```
- Then create a MySQL container that uses the before created volume:
```
docker run -d --volumes-from mysql-shared --name mysql1 -e MYSQL_ROOT_PASSWORD=admin mysql
```
- We need some data to be added to the database. So we pull and run another MySQL container just for the sake of using the client:
```
docker run -it --rm --link mysql1:mysqldb mysql mysql -u root -padmin -h mysqldb
```
- Now from inside the MySQL console, seed some data:
```
mysql> create database myDB;
mysql> use myDB;
mysql> CREATE TABLE Persons (PersonID int,LastName varchar(255),FirstName varchar(255),Address varchar(255),City varchar(255));
mysql> INSERT INTO Persons VALUES (1,'Doe','John','some address', 'some city');
```
- Ensure that the data exists:
```
 mysql> select * from Persons;
```
- Ok, we need to test data persistence. To do that, remove the mysql1 container that you've created, and create a new one called mysql2 that will use the same volumes:
```
docker stop mysql1
docker rm mysql1
docker run -d --volumes-from mysql-shared --name mysql2 -e MYSQL_ROOT_PASSWORD=admin mysql
docker run -it --rm --link mysql2:mysqldb mysql mysql -u root -padmin -h mysqldb
mysql> use mydocker;
mysql> select * from Persons;
```
This should return the same table results grabbed from mysql1 container

The "bind mount" volume type

- It refers to volumes that have user-specified mount points on the host operating system.
- This type is used when you want to share data between the container and the host.
- Let's say for example that you want to populate the Persons table with some data. You have a CSV file containing user details and you want to automatically generate the MySQL INSERT statements for them.
- You write a simple Python script that will parse the CSV file and return the appropriate SQL statements.
- You build an Ubuntu image for this and it works fine. Let’s have a look at how this could be accomplished in the next LAB slide.

LAB: create a Python script image for parsing CSV file

- Assuming that the python script file to do the job is csv_to_insert.py (all lab files are contained in the labs directory)
- Start by creating a new directory and placing a Dockerfile inside:
```
mkdir csvDocker
vim Dockefile
```
- Inside the Dockerfile, insert the following:
```
FROM Ubuntu:latest
RUN apt-get update
RUN apt-get install python
RUN mkdir /csv
COPY * /csv/
RUN useradd –ms /bin/sh worker
WORKDIR /csv
RUN chown worker /csv/csv_to_insert.py
RUN chmod a+x csv_to_insert.py
USER worker
CMD ["/csv/csv_to_insert.py"]
```
- Build the image:
```
docker build –t afakharany/csv
```
- Try running a container from that image:
```
docker run –it --rm afakharany/csv
```
- You should see now a couple of INSERT statements for the records contained in the CSV file.

Problems with this approach

- The previous method is great for a one-time-run script. But what if you want update the CSV with new data? You will have to rebuild the image again. For that reason, it is better to create a bind-mount volume, where the CSV file can be shared.
- Create a directory on your host machine to the files (it can be with any name but for this example we'll make it /csv)
  `mkdir /csv`
- Copy the python script and the CSV file inside the directory:
  `cp csv_to_insert.py people.csv /csv/`
- Make some changes to the CSV file
- Now start the image again but this time mount the host filesystem inside the container as a bind volume:
  `docker run –it --rm –v /csv:/csv afakharany/csv`
- You should see that the changes made to the CSV file were reflected to the script output.

Mounting a volume in read-only mode

- Another useful feature of using volumes is that you can opt to make them read-only so that the container command is denied from making any changes to the data on the host filesystem.
- This can be applied on the previous example by adding :ro after the volume name. for example:
  ```
  docker run –it --rm –v /csv:/csv:ro afakharany/csv
  ```
- Now try to make any changes to the file:
  ```
  docker run –it --rm –v /csv:/csv:ro afakharany/csv rm people.csv
  ```
- You will see an error message indicating that you cannot make changes to a read-only filesystem.

Docker-managed volumes

- The other type of volumes in Docker is the managed one.
- When you use managed volumes, you do not specify a mount point for the volume on the host filesystem. Instead, Docker mages this volume on some location on the host filesystem.
- For example, we can run the our python example without knowing (or caring to know) where Docker will save the files:
  ```
  docker run –it --name csvContainer –v /csv afakharany/csv
  ```
- If you want to know where did Docker save that mount point, you can use docker inspect command as follows:
  ```
  docker inspect –f "{{json .Mounts}}“ csvContainer
  ```

Sharing volumes the host-dependent way

- Sometimes you may need to share volumes across multiple containers. Let's say you want the mysql1 container to have access to the INSERT commands that you the csv container created.
- You can do that using host-dependent volumes, in which one or more volumes on the host are shared among different containers:

```
docker run --name csv -d –v /share:/csv afakharany/csv
docker start mysql1
docker run -it –v /share:/mnt --rm --name mysql-client --link
mysql1:mysqldb mysql mysql –u root –padmin –h mysqldb
mysql>use Persons;
mysql> source /mnt/insert.sql;
```

- In the previous example we created three containers:
  - *The csv for running our Python script against the csv file and generating the SQL file*
  - *The mysql1 container (already created in a previous lab so we're just starting it)*
  - *The mysql-client container is just used for connecting to the mysql1 database. But notice here that we mounted the /csv on /mnt so that we can use the insert.sql file inside the container.*
- This way we shared the /share file system among two containers: the first one processed the csv file using Python to create an SQL file, and the second used that SQL file to actually feed the database.

Generalized volume sharing

- The host-dependent sharing method is fine if you are just using no more than a couple of volumes and no docker-managed ones. But as the number of volumes get larger, and if you are mixing between docker-managed and bind volumes, it becomes harder to use this method. Not only because of the more typing you'll have to make but also because you will have to use the inspect command to determine the physical location of the volumes' mount points on the host system.
- An easier approach is to use the ---volumes-from option, which we discussed earlier. For example, let's say we have another filesystem /csv2 containing data that will be needed while working with mysql1. But this filesystem is not shared with the host:
  `docker run --name csv -d -v /share:/csv -v /csv2 --rm afakharany/csv`
- Now to use both file systems we can either inspect the csv container to see where /csv2 points, or we can use the following command:
  `docker run -it -volumes-from csv --rm --name mysql-client --link mysql1:mysqldb mysql mysql -u root -padmin -h mysqldb`
  which will ensure that all volumes used by the csv container will be available to mysql-client.
- Notice that even if csv container was already inheriting volumes from another parent container, all the volumes (the current and the inherited) will be available to mysql-client.

When NOT to use --volumes-from?

- Inherited volumes will always maintain the same mount point. This means that if the container needs to find the data on a specific mount point name that is different than the one used in the parent container, this option won't work. For example, if mysql-client was expecting to find the required data at /share2 instead of /csv2, using `--volumes-from` won't be of a benefit.
- Another condition is when more than one mount point are having the same name. For example, what if we have another container against which we used –volumes-from, and it happened to have a file system under /csv? That means that we are going to to have two different file systems with the same name. Only one of them will be available to the consuming container.
- Finally, if you are planning to change the read/write access permissions of the inherited volumes, then --volumes-from won't work for you. For example, what if you needed to make the /csv2 read-only? because the volume was exposed in the parent container as with read-write permissions, you can't change that in the consuming container.

Referencing managed volumes

- As mentioned, managed volumes are mounted in a location on the host that is managed by Docker. Although the user can determine the physical location of the mount point, those volumes cannot be shared except by using --volumes-from, which uses the owning container to identify them. Similarly, they cannot be removed except when the owning container is removed.
- For that reason, it is a common practice to create a container for each managed volume used on the system; so that you can identify them for sharing and deleting easily.
- Removing the container in itself does not remove the managed volume except if –v option was used. For example `docker rm –v afakharany/csv`
- If there is more than one container using the volume, the volume will not be deleted (even if –v is used) unless the last container using the volume is removed.
- If you forget to use the –v flag when deleting a container that owns a managed volume, this volume becomes *orphaned*. It can, however, be deleted but this involves many manual steps.
- By definition, bind volumes cannot be deleted because they are not managed by Docker, they are managed by the host operating system.