Docker/Containerization of MySQL Database

(A follow-up)

* **Prelude:**

This is not a replacement of previous post. It is a follow-up with new content. All materials I have published and will publish, are available on Github. <https://github.com/seantshen/linkedin_posts>

In the previous post, there were a number of loose ends that need to be tied up and further explored. First of all, we are using MariaDB-client package to access MySQL. I like to make it consistent. In this case, I choose to stick with MySQL community edition, therefore, we need to modify the client software to using that.

Secondly, in launching the container, I have used a hardcoded password, and worse, I am passing it explicitly in the command line. I like to explore a better, more secure method. In this case, I will follow MySQL image distributor’s default mechanism (which is, letting the instantiation process generate a random initial password). We will show how to obtain that password for the root account, and then use it to configure the database further.

Next, this blog will explore the use of storage volumes for the container. Instead of letting the container use the default, in-container storage, we will map the storage to the host (the AWS EC2 VM). This is closer to the real life scenario, especially for a storage hungry platform such as a database. This is brand new topic that was not covered in previous blog.

Finally, we will use a Dockerfile to build our own image for use by ourselves to instantiate the containers. This is also new material that we did not cover previously.

We will also explore launching multiple containers with the same Docker image, thus demonstrating the true advantage of containerization/Docker, compared with VM. This is also new content, that was not covered in the earlier blog. And BTW, in order to do that, we will be forced to upgrade our AWS VM with more memory.

* **MariaDB vs. MySQL:**

This blog is not here to cover the glamourous history. But if you are interested, you can browse through Wikipedia: <https://en.wikipedia.org/wiki/MariaDB>.

Why does it matter to us? The only thing I know of, is that MySQL has implemented a new user password mechanism, that does not seem to be compatible with the original (MariaDb client). And therefore, Mariadb client (still called “mysql”) will have difficulty connecting MySQL database, unless the encryption mechanism used for the ID is changed back to the original/native, with the following command:

ALTER USER 'userID'@'host' \

IDENTIFIED WITH mysql\_native\_password BY 'Xyz123'

The concern is that, if there is this problem discovered, what else is there that may be also incompatible? Let’s skip this issue by making things consistent between client and database server.

Here are the steps to uninstall MariaDB-client package and reinstall MySQL-client package. Therefore may be ways to install both side by side. But I have not explore doing that on the EC2 instance I am using.

$ yum erase MariaDB-client -y

And then, one will need to put MySQL yum repo onto the Centos VM. And there is a quick way to do that:

$ sudo rpm -Uvh <https://repo.mysql.com/mysql80-community-release-el7-3.noarch.rpm>

After that, one will see the MySQL repo file under /etc/yum.repos.d:

mysql-community.repo

mysql-community-source.repo

Just make sure things are clean, one should remove mariadb.repo from the same location.

Once this is done, you can now launch the installation:

$ sudo yum install mysql-community-client -y

Then it is done. This is totally up to us whether we want to do what is covered in this section. It is not even the main topic we are covering. The main topic is containerization and Docker.

* **Avoiding Hard-Coding Passwords for MySQL in the Containers:**

In the previous post, we were creating the containers downloaded from Docker registry, using the following command, and as you can see, the password is hard coded in the command, which is ugly:

$ docker run --name=mysqltest1 -d --env="MYSQL\_ROOT\_PASSWORD=MyPassword" --publish 6033:3306 mysql/mysql-server:latest

As it turns out, we don’t have to specify “—env” at all. And if we do not provide a way on how to handle initial root user password creation, MySQL image will generate a random password for us when the container is instantiated. The password is output into the container’s log. It can be retrieved by using the command (we created a new container called “MyContainerName” in the example below:

$ docker run --name=MyContainerName -d --publish 6033:3306 mysql/mysql-server:latest

$ docker logs MyContainerName 2>&1 | grep PASSWORD

**[Entrypoint] GENERATED ROOT PASSWORD: 4KEdercaS=ajM3tokypOfORyd=ag**

$

This password, upon having been used to login, must be changed.

By the way, there are other ways to handle and secure passwords for MySQL containerization, and for handling “secrets” like passwords. More complete coverage is available at Docker site, and at site for MySQL docker image specifically.

* **Storage in Containers for MySQL**

If we do not do anything special, the MySQL instance spawned off in the container has the defaults, including its location to store database files. This can be examined by executing the following command to read the my.cnf file.

$ docker exec -it MyContainerName cat /etc/my.cnf

There, the database files are located in below directory, from inside the container

datadir=/var/lib/mysql

Upon running docker inspect MyContainerName, we know

"Mounts": [

{

"Type": "volume",

"Name": "6f51eedf0ae6f0ff6ae4242f570a717adc9a89cc198781a45035fad84fd1a299",

"Source": "/var/lib/docker/volumes/6f51eedf0ae6f0ff6ae4242f570a717adc9a89cc198781a45035fad84fd1a299/\_data",

"Destination": "/var/lib/mysql",

"Driver": "local",

"Mode": "",

"RW": true,

"Propagation": ""

}

]

We found where the MySQL data is located, physically, on the host machine. Yes!

However, life is too difficult that way. Plus, for an enterprise environment where MySQL database is of significant size, we don’t want the data located there, do we? So, we will need to allocate and map a host volume directly into a storage volume inside the container.

In order to do this, let’s assume we have a new volume, at host side outside the container. It is /data/MyContainerName (I did it by allocating a new EBS volume of 20G, attached to the EC2 volume all in AWS cloud).

==== commands to make such a file system on host machine =======

$ sudo fdisk -l #to see the device is visible to OS

$ sudo mkfs -t ext4 /dev/xvdf #format the file system

$ sudo mkdir -p /data/MyContainerName #make dir

$ sudo echo "/dev/xvdf /data/MyContainerName ext4 defaults 0 2" >> /etc/fstab

$ sudo mount -a

$ mount | grep "/data/MyContainerName" #this step should show that /data/MyContainerName has been mounted to /dev/xvdf

$ sudo mkdir /data/MyContainerName/data1 #creating a subdir to host the MySQL data from this container

============================================================

With the steps above (details, details), we have a mounted volume pointed by the directory of /data/MyContainerName/data1.

It’s time to use the new 20G volume we worked so hard to create. In order do that, we need to destroy the original container MyContainerName, and create a new one (with the same image, but this time, with a newly added -v switch to map the directory of the default MySQL data location to the host volume (we now call it MySQLContainerName just to distinguish from the old).

$ docker run --name=MySQLContainerName -d --publish 6033:3306 -v /data/MyContainerName/data1:/var/lib/mysql mysql/mysql-server:latest

If this is successful, we should see a docker container created:

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

01c757ac8df9 mysql/mysql-server:latest "/entrypoint.sh mysq…" 3 minutes ago Up 3 minutes (healthy) 33060/tcp, 0.0.0.0:6033->3306/tcp MySQLContainerName

By running "docker inspect" command, one can see the volume mapping we just implemented:

$ docker inspect MyContainerName

…

"Mounts": [

{

"Type": "bind",

"Source": "/data/MyContainerName/data1",

"Destination": "/var/lib/mysql",

"Mode": "",

"RW": true,

"Propagation": "rprivate"

}

]

Moreover, with the container (and MySQL running in it), one can go into the directory on the host side, and see the content pertaining to the MySQL database.

$ pwd

/data/MyContainerName/data1

$ ls

auto.cnf binlog.index client-cert.pem #ib\_16384\_1.dblwr ib\_logfile0 #innodb\_temp mysql.ibd performance\_schema server-cert.pem undo\_001

binlog.000001 ca-key.pem client-key.pem ib\_buffer\_pool ib\_logfile1 lost+found mysql.sock private\_key.pem server-key.pem undo\_002

binlog.000002 ca.pem #ib\_16384\_0.dblwr ibdata1 ibtmp1 mysql mysql.sock.lock public\_key.pem sys

$

* **Build Our Own Docker Image with Dockerfile**

What if you want to build your own Docker image with your own customization? A typical example would be, that I have customized configurations I need to incorporate into my own MySQL via a customized “my.cnf” file, that is different from the original that came with the vendor distribution?

As a simple example, we have this new version of the my.cnf file that configures the MySQL data in /data/mysql, instead of the default location (/var/lib/mysql). There are multiple ways of accomplishing customized initial configurations, but for our example, we took the default copy of my.cnf, modified it and saved into a directory on the host machine.

[mysqld]

#

# Remove leading # and set to the amount of RAM for the most important data

# cache in MySQL. Start at 70% of total RAM for dedicated server, else 10%.

# innodb\_buffer\_pool\_size = 128M

#

# Remove leading # to turn on a very important data integrity option: logging

# changes to the binary log between backups.

# log\_bin

#

# Remove leading # to set options mainly useful for reporting servers.

# The server defaults are faster for transactions and fast SELECTs.

# Adjust sizes as needed, experiment to find the optimal values.

# join\_buffer\_size = 128M

# sort\_buffer\_size = 2M

# read\_rnd\_buffer\_size = 2M

# Remove leading # to revert to previous value for default\_authentication\_plugin,

# this will increase compatibility with older clients. For background, see:

# https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar\_default\_authentication\_plugin

# default-authentication-plugin=mysql\_native\_password

skip-host-cache

skip-name-resolve

#datadir=/var/lib/mysql

datadir=/data/mysql

socket=/var/lib/mysql/mysql.sock

secure-file-priv=/var/lib/mysql-files

user=mysql

pid-file=/var/run/mysqld/mysqld.pid

We will use the “docker build” feature to build a slightly modified MySQL image, based on the vendor provided original.

Step1: Create an empty directory

Step2: Compose a file called “Dockerfile” and save it there. The file’s content is this:

FROM mysql/mysql-server:latest

LABEL Description="This image is used to build MySQL database for my Docker series on LinedIn" Version="1" Maintainer="seantshen@gmail.com"

COPY ./my.cnf /etc/my.cnf

RUN yum update -y

RUN mkdir -p /data/mysql

EXPOSE 3306

This “Dockerfile” tells the build process to pull FROM the vendor provided latest version of mysql/mysql-server. This is our container image’s starting point. The BUILD process will COPY our customized my.cnf file to override the default MySQL my.cnf file. Then it will RUN an YUM UPDATE to update all yum packages present, including the Linux operating system patches. Second RUN command is to create the data directory /data/mysql specified in our customized my.cnf file, preparing for the introduction of MySQL database to run.

Step3: Make sure the modified my.cnf file is in the same directory. This is what the directory looks like

$ ls

Dockerfile my.cnf

$

Step4: build our own Docker image!

Before we build, let’s check what images we have in our docker repository on the host machine:

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

mysql/mysql-server latest 8a3a24ad33be 11 days ago 366MB

$

As expected, we only have one image “mysql/mysql-server:latest” in the system. It is something we have been using up to now to instantiate our containers.

Now let’s build:

$ docker build --tag my\_first\_mysql:1.0 .

The output of the command above, will show you, step by step, what it does as it goes through the Dockerbuild file. It is actually interesting to look at it, so I am copying it here:

Sending build context to Docker daemon 5.632kB

Step 1/6 : FROM mysql/mysql-server:latest

---> 8a3a24ad33be

Step 2/6 : LABEL Description="This image is used to build MySQL database for my Docker tutorial series" Version="1" Maintainer="seantshen@gmail.com"

---> Running in 0e05456cbfb9

Removing intermediate container 0e05456cbfb9

---> 254951d5655c

Step 3/6 : COPY ./my.cnf /etc/my.cnf

---> e1e81f866c0d

Step 4/6 : RUN yum update -y

---> Running in 32597f2dd54f

Loaded plugins: ovl

Resolving Dependencies

--> Running transaction check

---> Package ca-certificates.noarch 0:2019.2.32-76.el7\_7 will be updated

---> Package ca-certificates.noarch 0:2020.2.41-70.0.el7\_8 will be an update

---> Package libgcc.x86\_64 0:4.8.5-39.0.3.el7 will be updated

---> Package libgcc.x86\_64 0:4.8.5-39.0.5.el7 will be an update

---> Package libstdc++.x86\_64 0:4.8.5-39.0.3.el7 will be updated

---> Package libstdc++.x86\_64 0:4.8.5-39.0.5.el7 will be an update

---> Package oraclelinux-release-el7.x86\_64 0:1.0-11.el7 will be updated

---> Package oraclelinux-release-el7.x86\_64 0:1.0-12.1.el7 will be an update

--> Finished Dependency Resolution

Dependencies Resolved

================================================================================

Package Arch Version Repository Size

================================================================================

Updating:

ca-certificates noarch 2020.2.41-70.0.el7\_8 ol7\_latest 382 k

libgcc x86\_64 4.8.5-39.0.5.el7 ol7\_latest 103 k

libstdc++ x86\_64 4.8.5-39.0.5.el7 ol7\_latest 306 k

oraclelinux-release-el7 x86\_64 1.0-12.1.el7 ol7\_latest 19 k

Transaction Summary

================================================================================

Upgrade 4 Packages

Total download size: 809 k

Downloading packages:

Delta RPMs disabled because /usr/bin/applydeltarpm not installed.

--------------------------------------------------------------------------------

Total 711 kB/s | 809 kB 00:01

Running transaction check

Running transaction test

Transaction test succeeded

Running transaction

Updating : libgcc-4.8.5-39.0.5.el7.x86\_64 1/8

Updating : libstdc++-4.8.5-39.0.5.el7.x86\_64 2/8

Updating : ca-certificates-2020.2.41-70.0.el7\_8.noarch 3/8

/usr/bin/ca-legacy: line 38: sln: command not found

Updating : oraclelinux-release-el7-1.0-12.1.el7.x86\_64 4/8

Cleanup : ca-certificates-2019.2.32-76.el7\_7.noarch 5/8

Cleanup : oraclelinux-release-el7-1.0-11.el7.x86\_64 6/8

Cleanup : libstdc++-4.8.5-39.0.3.el7.x86\_64 7/8

Cleanup : libgcc-4.8.5-39.0.3.el7.x86\_64 8/8

Verifying : libgcc-4.8.5-39.0.5.el7.x86\_64 1/8

Verifying : oraclelinux-release-el7-1.0-12.1.el7.x86\_64 2/8

Verifying : ca-certificates-2020.2.41-70.0.el7\_8.noarch 3/8

Verifying : libstdc++-4.8.5-39.0.5.el7.x86\_64 4/8

Verifying : libgcc-4.8.5-39.0.3.el7.x86\_64 5/8

Verifying : oraclelinux-release-el7-1.0-11.el7.x86\_64 6/8

Verifying : libstdc++-4.8.5-39.0.3.el7.x86\_64 7/8

Verifying : ca-certificates-2019.2.32-76.el7\_7.noarch 8/8

Updated:

ca-certificates.noarch 0:2020.2.41-70.0.el7\_8

libgcc.x86\_64 0:4.8.5-39.0.5.el7

libstdc++.x86\_64 0:4.8.5-39.0.5.el7

oraclelinux-release-el7.x86\_64 0:1.0-12.1.el7

Complete!

Removing intermediate container 32597f2dd54f

---> 77923ff36f05

Step 5/6 : RUN mkdir -p /data/mysql

---> Running in 3114b3c50bdf

Removing intermediate container 3114b3c50bdf

---> 89d9a3261a6b

Step 6/6 : EXPOSE 3306

---> Running in 63b38ac23708

Removing intermediate container 63b38ac23708

---> 80f1215e2190

Successfully built 80f1215e2190

Successfully tagged my\_first\_mysql:1.0

Nice! Looks like all steps have passed with no errors. Let’s see what images are now present in our local Docker repository:

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

my\_first\_mysql 1.0 80f1215e2190 13 minutes ago 582MB

mysql/mysql-server latest 8a3a24ad33be 11 days ago 366MB

$

Indeed, the above shows the new image has been created and now it is present in our system. Let’s instantiate an actual container from it. Before we do that, let’s check if we have any containers running:

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

01c757ac8df9 mysql/mysql-server:latest "/entrypoint.sh mysq…" 3 hours ago Up 3 hours (healthy) 33060/tcp, 0.0.0.0:6033->3306/tcp MySQLContainerName

$

At this point we have the container we have been using from the previous steps. We will see if we can kick off a different one, with our newly built image. In preparing for this, let’s again use the extra volume we created earlier, but with a different subdirectory:

$ sudo mkdir /data/MyContainerName/data2

Now we are ready:

$ docker run --name=MyOwnNewSQLContainer -d --publish 6044:3306 -v /data/MyContainerName/data2:/data/mysql my\_first\_mysql:1.0

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

911194ace555 my\_first\_mysql:1.0 "/entrypoint.sh mysq…" 9 seconds ago Up 7 seconds (health: starting) 33060/tcp, 0.0.0.0:6044->3306/tcp MyOwnNewSQLContainer

cc87f36fa9cc mysql/mysql-server:latest "/entrypoint.sh mysq…" 7 minutes ago Up 7 minutes (healthy) 33060/tcp, 0.0.0.0:6033->3306/tcp MySQLContainerName

If we run the “docker inspect” on each of them, we got their respective IP addresses our host Docker system automatically assigned to them:

$ docker inspect MyOwnNewSQLContainer| grep IPAddress

"SecondaryIPAddresses": null,

"IPAddress": "172.17.0.2",

"IPAddress": "172.17.0.2",

$ docker inspect MySQLContainerName| grep IPAddress

"SecondaryIPAddresses": null,

"IPAddress": "172.17.0.3",

"IPAddress": "172.17.0.3",

As you can see, we have successfully launched two containers running on the same host machine. Here are their vital parameters.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Container ID | ContainerName | Image | Local Data Dir in container | Mapped to host Directory | Local Port | IPAddress | Host Port |
| 911194ace555 | MyOwnNewSQLContainer | my\_first\_mysql:1.0 | /data/mysql | /data/MyContainerName/data2 | 3306 | 172.17.0.2 | 6044 |
| cc87f36fa9cc | MySQLContainerName | mysql/mysql-server:latest | /var/lib/mysql | /data/MyContainerName/data1 | 3306 | 172.17.0.3 | 6033 |

* **Conclusion:**

What is important in what was done here, are the following:

* We did things a little differently than our first published blog. MySQL initial root passwords are differently handled, and we start using, consistently, MySQL software both for client (host VM) and server (containers)
* We mapped the storage attached to the MySQL databases onto a defined volume on the host, so we easily know where it is, and we have total control where we want it to be, rather than relying on the default of MySQL container installation.
* We built a custom docker image based on the commercially provided image, with our own customizations, and successfully launched a container with it.
* We also saw, how to run two containers with total isolation on the same OS but with separate images