Docker/Containerization of MySQL Database

* **Prelude:**

I often times have trouble appreciating and visualizing things conceptually, if there are no concrete examples that proved to me beyond a shadow of doubt how things work. Somehow I tend to reject ideas if it is not laid bare in front of my eyes.

When it comes to containerization, particularly, Docker, I am the same way. And if I am like that, I suspect many are similar. So, in this blog, I am giving a complete tutorial, that can be repeated even if you don’t know anything about containerization, let alone, Docker.

I am not going to waste my time and yours to explain what a container is, what docker is, etc. Therefore plenty of articles on the web that do just that. Go read them if needed

Let’s get right to it.

* **Install Docker:**

I am using AWS EC2 instance, running Centos operating system. simply because this is the environment similar to RedHat Linux, which I am most familiar with. I assume the readers already are familiar with AWS, how to provision and access EC2. If not, get a tutorial or a class of some sort from Udemy.com or ACloud.Guru for the price of two cups of coffee.

Step 1: Login to your EC2 instance running CENTOS

This is what I have

$ hostnamectl

Static hostname: ip-172-31-39-20.us-east-2.compute.internal

Icon name: computer-vm

Chassis: vm

Machine ID: 3d5c05376530a2eb49e3e90576f83c5b

Boot ID: 1b93e94ae1f742a6ba3e564c1bec8631

Virtualization: xen

Operating System: CentOS Linux 7 (Core)

CPE OS Name: cpe:/o:centos:centos:7

Kernel: Linux 3.10.0-1062.12.1.el7.x86\_64

Architecture: x86-64

$

To install Docker the simplest thing I found is to do this:

$ curl -fsSL https://get.docker.com/ | sh

This will consolidate several steps into one, including, configuring configuration of yum repo, launch the installation.

Then, kick off and enable dockerd service

$ sudo systemctl start dockerd

$ sudo systemctl status dockerd

$ sudo systemctl enable dockerd

$ sudo usermod -aG docker $(whoami)

The last command, is to add the current user into docker group, so that it can run docker related commands without you have to login as root or sudo. Actually one more step after this, you do need to log out (exit the ssh session to your EC2) and connect back in.

After logging back in, run. It should work.

$ docker info

$ docker images

Since this is a brand new install of CENTOS and on it, docker, there is no image present.

* **Run a containerized MySQL database instance**

In rest of this step, we will run a containerized MySQL instance. This is to show, that you don’t have to allocate a VM, download the MySQL software, and have it deployed, in order to run MySQL database.

First, we need to pull the docker image down to the local machine where we want to run the container:

$ docker pull mysql/mysql-server:latest

$ docker images

Assuming it is successfully done, it now should show the image successfully downloaded.

Having the image in place, does not mean it is running. In order to run it, issue the following command:

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

Now, this command takes a little bit of explaination:

It says, to run a container, based on the image we just downloaded (mysql/mysql-server:latest). MySQL database server’s default port is 3306 as we all know. But I want this port to be published to the host OS (where I am running CENTOS) as 6033. And the last switch, is to preset root password of the newly created MySQL database, to be “MyPassword”

* **Preparing to test:**

In order to test how well we did, we need to install, on my CENTOS host VM, the necessary tools to conduct the test. In this case, I am going to cheat….by taking advantage of the compatibility of MariaDb and MySQL. Instead of installing MySQL utilities from Oracle, I am installing the MariaDb repos and clients.

In order to do this, I need to set up Yum repo for MariaDb

$ curl -sS https://downloads.mariadb.com/MariaDB/mariadb\_repo\_setup | sudo bash

Then, yum install MariaDB-client package

$ sudo yum install MariaDB-client

* **Test if the sucker works**

*Test 1: Connect within the container to see if it works*

$ docker exec -it mysqltest1 mysql -u root -p

-- here you have enter password

mysql> use mysql;

Database changed

mysql> select host, user from user;

+-----------+------------------+

| host | user |

+-----------+------------------+

| localhost | healthchecker |

| localhost | mysql.infoschema |

| localhost | mysql.session |

| localhost | mysql.sys |

| localhost | root |

+-----------+------------------+

5 rows in set (0.00 sec)

mysql> quit

Bye

Yay, that works….next,

$ docker exec mysqltest1 \

mysql \

--user=root \

--password=MyPassword \

-e "CREATE USER 'root'@'%' \

WITH mysql\_native\_password IDENTIFIED BY 'MyPassword';\

GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'%'; \

FLUSH PRIVILEGES;"

ALTER USER 'root'@'%' IDENTIFIED WITH mysql\_native\_password BY 'MyPassword';

This takes a little bit explanation:

* + Note that we are no longer using “-it” switch in second command, because we are directly executing the entire command of mysql inside container. No need for interaction with terminal.
  + “docker exec” identifies and executes a command inside the container. In this case, container of mysqltest1, and the command is the mysql (with its trailing arguments).
  + “mysql” directly runs this inside the container, with the actual SQL commands to create a user root regardless of hostnames ‘%’.
  + The result of this command, if successful, prepares us for subsequent tests.

Back to using “-it” in interactive mode:

$ docker exec -it mysqltest1 mysql -u root -p

This will prompt you to enter the password (MyPassword). To check the result of our previous test step, we issue this SQL

mysql> use mysql;

Database changed

mysql> select user, host from user;

+------------------+-----------+

| user | host |

+------------------+-----------+

| root | % |

| healthchecker | localhost |

| mysql.infoschema | localhost |

| mysql.session | localhost |

| mysql.sys | localhost |

| root | localhost |

+------------------+-----------+

6 rows in set (0.00 sec)

mysql>

As you can see, we have an extra entry with user as root and host as %. That is the direct result of the previous step. So previous step worked.

*Test 2: Interacting with the container-based MySQL from outside container*

If a containerized program (such as MySQL) can only be used inside its own container, then its usefulness is very much limited. Fortunately, it is not the case. A containerized application, can be used for general purposes, and can also to be consumed by other applications in other containers.

In this case, we will test and demonstration that this MySQL database that we instantiated in Docker container, can be used as a general purpose database, just like any other databases running on physical machines or VMs.

When we were preparing for the test, we had installed MariaDb-client package, which includes mysql utility.

* To find the TCP/IP connection strings for the MySQL database from the perspective of the host:

$ docker inspect mysqltest1 | grep IPAddress

"SecondaryIPAddresses": null,

"IPAddress": "172.17.0.2",

"IPAddress": "172.17.0.2",

$

So the IP address should be 172.17.0.2 and we know the default port (which we never changed) is 3306.

Therfore, this would work, from the VM where Docker is hosted.

$ mysql -h 172.17.0.2 -P 3306 -u root -p

Enter password:

Welcome to the MariaDB monitor. Commands end with ; or \g.

Your MySQL connection id is 772

Server version: 8.0.21 MySQL Community Server - GPL

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]>

So it did work.

* The second way database can be accessed, is via the IP address of the host machine and mapped port.

Remember, when we started the container, we used this command

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

Therefore, the internal port of 3306 is mapped to the host machine as 6033. All we need is the IP addresses of the host machine. For that, we will use AWS’ metadata retrival URL.

For private IP address

curl http://169.254.169.254/latest/meta-data/ local-ipv4

In my case, this returns

172.31.42.119

One can also use ifconfig -a to get the same answer.

For public IP address, use “dig” command

dig +short myip.opendns.com @resolver1.opendns.com

In my case, it returns

$ dig +short myip.opendns.com @resolver1.opendns.com

18.223.136.195

It also has loopback address

127.0.0.1

All three IP addresses can be used to reach the MySQL database, provided port 6033 is used. Lets give it a try

$ mysql -h 127.0.0.1 -P 6033 -u root -p

$ mysql -h 172.31.42.119 -P 6033 -u root -p

$ mysql -h 18.223.136.195 -P 6033 -u root -p

Plus the original path (with port 3306)

$ mysql -h 172.17.0.2 -P 3306 -u root -p

You have to take my word for it. They all work, after password is provided.

From within the AWS, private addresses can be used to connect to the database.

From outside AWS, the public address has to be used. Proper network security group has to be defined to open up the IP and port of course, in order for inbound connection to be made.

From my MacBook Pro (which obviously sits outside AWS), here is the result when I tried

Seans-MBP% mysql -h 18.223.136.195 -P 6033 -u root -p

Enter password:

Welcome to the MariaDB monitor. Commands end with ; or \g.

Your MySQL connection id is 903

Server version: 8.0.21 MySQL Community Server - GPL

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]>

As you can see, this succeeded, which means, this instance of MySQL running on a Centos OS host VM machine, via Docker container, is reachable just like a regularly installed MySQL on the same VM machine.