Running Apache Cassandra® Single and Multi-Node Clusters on Docker with Docker Compose

Sometimes you might need to spin up a local test database quickly–a database that doesn't need to last beyond a set time or number of uses. Or maybe you want to integrate Apache Cassandra® into an existing **Docker** setup.

Either way, you're going to want to run Cassandra on Docker, which means running it in a container with Docker as the container manager. This tutorial is here to guide you through running a single and multi-node setup of Apache Cassandra on Docker.

Prerequisites

Before getting started, you'll need to have a few things already installed, and a few basic skills. These will make deploying and running your Cassandra database in Docker a seamless experience:

- **Docker installed**
- Basic knowledge of containers and Docker (see the **Docker documentation** for more insight)
- Basic command line knowledge
- A code editor (I use **VSCode**)
- CQL shell, aka **cqlsh**, installed (instructions for installing a standalone cqlsh without installing Cassandra can be found **here**)

Method 1: Running a single Cassandra node using Docker CLI

This method uses the **Docker CLI** to create a container based on the **latest official Cassandra image**. In this example we will:

- Set up the Docker container
- Test that it's set up by connecting to it and running cqlsh
- Clean up the container once you're done with using it.

Setting up the container

You can run Cassandra on your machine by opening up a terminal and using the following command in the Docker CLI:

docker run –name my-cassandra-db -d cassandra:latest

Let's look at what this command does:

- Docker uses the '***run***' subcommand to run new containers.
- The '–name' field allows us to name the container, which helps for later use and cleanup; we'll use the name '***my-cassandra-db'***.
- The '-d' flag tells Docker to run the container in the background, so we can run other commands or close the terminal without turning off the container.
- The final argument ***'cassandra:latest'*** is the image to build the container from; we're using the **latest official Cassandra image**.

When you run this, you should see an ID, like the screenshot below:

To check and make sure everything is running smoothly, run the following command:

docker ps -a

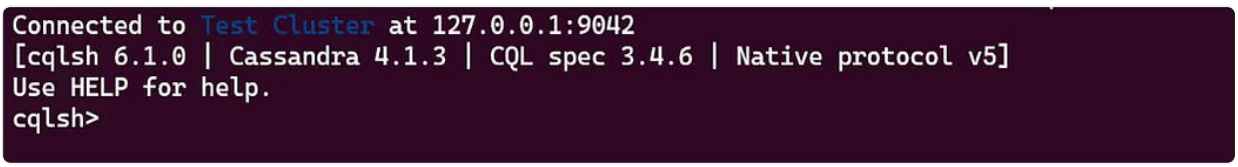
You should see something like this:

Connecting to the container

Now that the data container has been created, you can now connect to it using the following command:

docker exec -it my-cassandra-db cqlsh

This will run **cqlsh, or CQL Shell**, inside your container, allowing you to make queries to your new Cassandra database. You should see a prompt like the following:



Cleaning up the container

Once you're done, you can clean up the container with the 'docker rm' command. First, you'll need to stop the container though, so you must to run the following 2 commands:

docker stop my-cassandra-db

docker rm my-cassandra-db

This will delete the database container, including all data that was written to the database. You'll see a prompt like the following, which, if it worked correctly, will show the ID of the container being stopped/removed:

```
nodebotanist@kassian01-PC:~/code$ sudo docker stop my-cassandra-db
my-cassandra-db
nodebotanist@kassian01-PC:~/code$ sudo docker rm my-cassandra-db
my-cassandra-db
```

Method 2: Deploying a three-node Apache Cassandra cluster using Docker compose

This method allows you to have multiple nodes running on a single machine. But in which situations would you want to use this method? Some examples include testing the consistency level of your queries, your replication setup, and more.

Writing a docker-compose.yml

The first step is creating a docker-compose.yml file that describes our Cassandra cluster. In your code editor, create a ***docker-compose.yml*** file and enter the following into it:

```
version: '3.8'

networks:

  cassandra:

services:

  cassandra1:

    image: cassandra:latest

    container_name: cassandra1

    hostname: cassandra1

    networks:

      - cassandra

    ports:
```

```
22
23          - "9042:9042"
24
25      environment: &environment
26
27          CASSANDRA_SEEDS: "cassandra1,cassandra2"
28
29          CASSANDRA_CLUSTER_NAME: MyTestCluster
30
31          CASSANDRA_DC: DC1
32
33          CASSANDRA_RACK: RACK1
34
35          CASSANDRA_ENDPOINT_SNITCH: GossipingPropertyFileSnitch
36
37          CASSANDRA_NUM_TOKENS: 128
38
39    cassandra2:
40
41      image: cassandra:latest
42
43      container_name: cassandra2
44
45      hostname: cassandra2
46
47      networks:
48
49        - cassandra
50
51      ports:
52
53        - "9043:9042"
54
55      environment: *environment
56
57      depends_on:
58
59        cassandra1:
60
61          condition: service_started
```

```
62
63    cassandra3:
64
65      image: cassandra:latest
66
67      container_name: cassandra3
68
69      hostname: cassandra3
70
71      networks:
72
73        - cassandra
74
75      ports:
76
77        - "9044:9042"
78
79      environment: *environment
80
81      depends_on:
82
83        cassandra2:
84
85          condition: service_started
```

So what does this all mean? Let's examine it part-by-part:
First, we declare our docker compose version.

```
1   version: '3.8'
```

Then, we declared a network called cassandra to host our cluster.

```
1   networks:
2    cassandra:
```

Under services, cassandra1 is started. (NOTE: the depends on service start conditions in cassandra2 and cassandra3's `depends_on~ attributes prevent them from starting until the service on cassandra1 and cassandra2 have started, respectively.) We also set the port

forwarding here so that our local 9042 port will map to the container's 9042. We also add it to the cassandra network we established:

```
services:

  cassandra1:

    image: cassandra:latest

    container_name: cassandra1

    hostname: cassandra1

    networks:

      - cassandra

    ports:

      - "9042:9042"

    environment: &environment

        CASSANDRA_SEEDS: "cassandra1,cassandra2"

        CASSANDRA_CLUSTER_NAME: MyTestCluster

        CASSANDRA_DC: DC1

        CASSANDRA_RACK: RACK1

        CASSANDRA_ENDPOINT_SNITCH: GossipingPropertyFileSnitch

        CASSANDRA_NUM_TOKENS: 128
```

Finally, we set some environment variables needed for startup, such as declaring CASSANDRA_SEEDS to be cassandra1 and cassandra2.

The configurations for containers 'cassandra2 'and 'cassandra3' are very similar; the only real difference are the names.

- Both use the same cassandra:latest image, set container names, add themselves to the Cassandra network, and expose their 9042 port.
- They also point to the same environment variables as cassandra1 with the *environment syntax.

Their only difference? cassandra2 waits on cassandra1, and cassandra3 waits on cassandra2. Here is the code section that this maps to:

```
cassandra2:

    image: cassandra:latest

    container_name: cassandra2

    hostname: cassandra2

    networks:

      - cassandra

    ports:

      - "9043:9042"

    environment: *environment

    depends_on:

      cassandra1:

        condition: service_started

  cassandra3:

    image: cassandra:latest

    container_name: cassandra3

    hostname: cassandra3

    networks:
```

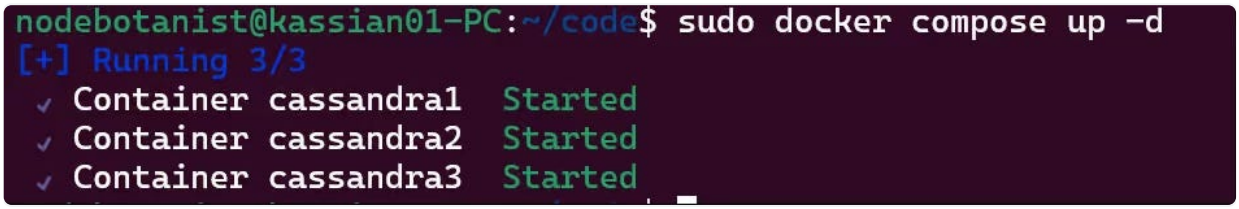
```
      - cassandra

    ports:

      - "9044:9042"

    environment: *environment

    depends_on:

      cassandra2:

        condition: service_started
```

Deploying your Cassandra cluster and running commands

To deploy your Cassandra cluster, use the Docker CLI in the same folder as your docker-compose.yml to run the following command (the -d causes the containers to run in the background):

```
docker compose up -d
```

Quite a few things should happen in your terminal when you run the command, but when the dust has settled you should see something like this:



If you run the 'docker ps -a,' command, you should see three running containers:

To access your Cassandra cluster, you can use csqlsh to connect to the container database using the following commands:

```
sudo docker exec -it cassandra1 cqlsh
```

You can also check the cluster configuration using:

```
docker exec -it cassandra1 nodetool status
```

Which will get you something like this:

```
nodebotanist@kassian01-PC:~/code$ sudo docker exec -it cassandra1 nodetool status
Datacenter: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load        Tokens  Owns (effective)  Host ID                               Rack
UN  172.19.0.2  107.97 KiB  128     67.3%             41ca10ef-41d7-45a6-b75e-94934308afac  RACK1
UN  172.19.0.4  190.03 KiB  128     61.3%             8bffc366-898c-464e-809d-86ccb907495a  RACK1
UN  172.19.0.3  113.04 KiB  128     71.4%             27f73589-ce40-441a-a08d-44a3aadb44b0  RACK1
```

And the node info with:

```
docker exec -it cassandra1 nodetool info
```

From which you'll see something similar to the following:

```
nodebotanist@kassian01-PC:~/code$ sudo docker exec -it cassandra1 nodetool info
ID                     : 41ca10ef-41d7-45a6-b75e-94934308afac
Gossip active          : true
Native Transport active: true
Load                   : 107.97 KiB
Generation No          : 1694031947
Uptime (seconds)       : 207
Heap Memory (MB)       : 695.87 / 3859.13
Off Heap Memory (MB)   : 0.00
Data Center            : DC1
Rack                   : RACK1
Exceptions             : 0
Key Cache              : entries 10, size 896 bytes, capacity 100 MiB, 254 hits, 282 requests, 0.901 recent hit rate, 14400 save period in s
econds
Row Cache              : entries 0, size 0 bytes, capacity 0 bytes, 0 hits, 0 requests, NaN recent hit rate, 0 save period in seconds
Counter Cache          : entries 0, size 0 bytes, capacity 50 MiB, 0 hits, 0 requests, NaN recent hit rate, 7200 save period in seconds
Network Cache          : size 8 MiB, overflow size: 0 bytes, capacity 128 MiB
Percent Repaired       : 100.0%
Token                  : (invoke with -T/--tokens to see all 128 tokens)
```

You can also run these commands on the cassandra2 and cassandra3 containers.

Cleaning up

Once you're done with the database cluster, you can take it down and remove it with the following command:

```
docker compose down
```

This will stop and destroy all three containers, outputting something like this:

```
nodebotanist@kassian01-PC:~/code$ sudo docker compose down
[+] Running 4/4
 ✓ Container cassandra3    Removed
 ✓ Container cassandra2    Removed
 ✓ Container cassandra1    Removed
 ✓ Network code_cassandra  Removed
```

Now that we've covered two ways to run Cassandra in Docker, let's look at a few things to keep in mind when you're using it.

Important things to know about running Cassandra in Docker

Data Permanence

Unless you declare volumes on the machine that maps to container volumes, the data you write to your Cassandra database will be erased when the container is destroyed. (You can read more about using Docker volumes **here)**.

Performance and Resources

Apache Cassandra can take a lot of resources, especially when a cluster is deployed on a single machine. This can affect the performance of queries, and you'll need a decent amount of CPU and RAM to run a cluster locally.

Conclusion

There are several ways to run Apache Cassandra on Docker, and we hope this post has illuminated a few ways to do so. If you're interested in learning more about Cassandra, you can find out more about **how data modelling works with Cassandra**, or **how PostgreSQL and Cassandra differ.**