Lab 8 - Docker Compose

Introduction

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Compose works in all environments: production, staging, development, testing, as well as CI workflows.

1. Install Docker Compose

1.1 Login as **"root"** user on **aio110** host:

Copy

```
ssh root@aio110
```

1.2 Run this command to download the latest version of Docker Compose:

Copy

```
curl -L
https://github.com/docker/compose/releases/download/1.19.0/docker-
compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
```

Output:

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Curre
nt

                                 Dload  Upload   Total   Spent    Left
Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--
:--
0     0    0     0    0     0      0      0 --:--:--  0:00:01 --:--:--
0     0    0     0    0     0      0      0 --:--:--  0:00:02 --:--:--
```

```
....

....

 54 8288k   54 4537k    0     0   356k      0  0:00:23  0:00:12
0:00:11 1057
100 8288k  100 8288k    0     0   619k      0  0:00:13  0:00:13 --:--
:-- 2113
k
```

1.3 After the download has completed, make the binary executable with the command:

Copy

```
chmod +x /usr/local/bin/docker-compose
```

1.4 Verify the **version** has been installed.

Copy

```
docker-compose --version
```

Output:

```
docker-compose version 1.19.0, build 9e633ef
```

1.5 Make sure **bash** completion is installed.

Copy

```
curl -L https://raw.githubusercontent.com/docker/compose/$(docker-
compose version --short)/contrib/completion/bash/docker-compose >
/etc/bash_completion.d/docker-compose
```

Output:

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Curre
nt
```

```
                                 Dload  Upload   Total   Spent    Left
Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--
:--
0     0    0     0    0     0      0      0 --:--:--  0:00:01 --:--:--
0     0    0     0    0     0      0      0 --:--:--  0:00:02 --:--:--

....

....

 54 8288k   54 4537k    0     0   356k      0  0:00:23  0:00:12
0:00:11 1057
100 8288k  100 8288k    0     0   619k      0  0:00:13  0:00:13 --:--
:-- 2113
k
```

2. Get started with Docker Compose

Docker Compose is a four-step process:

- Create a directory for the install files.
- Create a YAML file named **docker-compose.yml** in that directory.
- Put the application group launch commands into **docker-compose.yml**.
- From inside that directory, use the command **docker-compose up** to launch the container(s).

2.1 Create a directory for a simple project called **test_compose**:

Copy

```
mkdir test_compose && cd test_compose
```

2.2 Create a YAML file named **docker-compose.yml** file,

In this example, you'll take the base application from Github and complete the **docker-compose.yml** file in it. This application uses **Node**, **NPM** and **MongoDB**.

Just like the Dockerfile, the **docker-compose.yml** file tells Docker how to build what you need for your containers.

a. Choose Your Docker Compose Version

The first line of any **docker-compose.yml** file is the version setting.

```
version: '3.3'
```

b. Define Node and Mongo Services

Services are how Docker refers to each container you want to build in the docker-compose file.

In this case, you'll create two services:

- **NodeJS** application
- **MongoDB** database.

```
services:

  app:

  db:
```

First, tell Docker what image you want to build the app service from by specifying that you'll be building from the **sample:1.0** image. So you'll specify that indented under the app tag.

```
app:

  image: sample:1.0
```

Of course that image doesn't exist, so you'll need to let Docker know where to find the Dockerfile to build it by setting a **build** context. If you don't, Docker will try to pull the image from Docker Hub and when that fails it will fail the docker-compose command altogether.

```
app:

  image: sample:1.0
```

```
  build: .
```

Here, you've specified that the build context is the current directory, so when Docker can't find the sample:1.0 image locally, it will build it using the Dockerfile in the current directory.

Next, you'll tell Docker what the container name should be once it's built the image to create the container from.

```
app:

  image: sample:1.0

  container_name: sample_app

  build: .
```

Now, when Docker builds the image, it will immediately create a container named sample_app from that image.

By default, NodeJS apps run on **port 3000**, so you'll need to map that port to 80, since this is the "production" docker-compose file. You do that using the ports tag in YAML.

```
app:

  image: sample:1.0

  container_name: sample_app

  build: .

  ports:

    - 80:3000
```

Here, you've mapped port 80 on the host operating system, to port 3000 from the container. That way, when you've moved this container to a production host, users of the application can go to the host machine's port 80 and have those requests answered from the container on port 3000.

Your application will be getting data from a MongoDB database and to do that the application will need a connection string that it will get from an environment variable called “MONGO_URI”.

```
app:

  image: sample:1.0

  container_name: sample_app

  build: .

  ports:

    - 80:3000

  environment:

    - MONGO_URI=mongodb://sampledb/sample
```

c. Create a Docker Network

For the application service to actually be able to reach the sample database, it will need to be on the same network. To put both of these services on the same network, create one in the docker-compose file by using the networks tag at the top level.

```
version: '3.3'



services:

  app:...

  db:...

networks:

  samplenet:

    driver: bridge
```

This creates a network called “**samplenet**” using a bridge type network. This will allow the two containers to communicate over a virtual network between them.

Back in the app section of the file, join the app service to the “samplenet” network:

```
app:

  image: sample:1.0

  container_name: sample_app

  build: .

  ports:

    - 80:3000

  environment:

    - MONGO_URI=mongodb://sampledb/sample

  networks:

    - samplenet
```

d. Create the MongoDB Service

Now the app service is ready, but it won’t be much good without the db service. So add the same kinds of things in the next section for the db service.

```
 db:

  image: mongo:3.0.15

  container_name: sample_db

  networks:

    samplenet:

      aliases:
```

```
        - "sampledb"
```

This service builds from the official MongoDB 3.0.15 image and creates a container named “sample_db”. It also joins the “samplenet” network with an alias of “sampledb”.

e. Use Docker Volumes

You’ll also want to create a volume mount in the database service.

```
db:

  image: mongo:3.0.15

  container_name: sample_db

  volumes:

    - ./db:/data/db

  networks:

    samplenet:

      aliases:

        - "sampledb"
```

Review Your Docker Compose File

With all that done, your final docker-compose.yml file should look like:

Copy

```
cat > docker-compose.yml <<EOF

version: '3.3'



services:
```

```
app:
  image: sample:1.0
  container_name: sample_app
  build: .
  ports:
    - 80:3000
  environment:
    - MONGO_URI=mongodb://sampledb/sample
  depends_on:
    - db
  networks:
    - samplenet
db:
  image: mongo:3.0.15
  container_name: sample_db
  volumes:
    - ./db:/data/db
  networks:
    samplenet:
      aliases:
        - "sampledb"
```

```
networks:

  samplenet:

    driver: bridge

EOF
```

If everything completes successfully, you can then go to **http://localhost/users** and see something like the image below.

f. Create a Dockerfile which references to the base images:

Copy

```
cat > Dockerfile <<EOF

FROM node:8.4

COPY . /app

WORKDIR /app

RUN ["npm", "install"]

EXPOSE 3000/tcp

CMD ["npm", "start"]

EOF
```

2.3 Use Docker Compose to launch this test container with the command:

Copy

```
docker-compose up -d
```

Output:

```
Building app
```

```
Step 1/6 : FROM node:8.4

8.4: Pulling from library/node

aa18ad1a0d33: Pulling fs layer

90f6d19ae388: Downloading [>
90f6d19ae388: Downloading [==>
90f6d19ae388: Downloading [=====>
90f6d19ae388: Downloading [=========>
90f6d19ae388: Downloading [============>
90f6d19ae388: Downloading [==============>
]  5.733MB/19.26MBng

90f6d19ae388: Downloading [=================>
90f6d19ae388: Downloading [====================>
aa18ad1a0d33: Downloading [>
]  538.4kB/52.6MBloading [=>
90f6d19ae388: Downloading [======================>
90f6d19ae388: Downloading [=========================>
90f6d19ae388: Downloading [============================>
]  10.91MB/19.26MB

90f6d19ae388: Downloading [==============================>
90f6d19ae388: Downloading [================================>
aa18ad1a0d33: Downloading [=>
]  1.064MB/52.6MBloading [==>
90f6d19ae388: Downloading [====================================>
90f6d19ae388: Downloading [=======================================>
]  14.88MB/19.26MB

90f6d19ae388: Downloading [=========================================>
aa18ad1a0d33: Downloading [=>
]  1.605MB/52.6MBB



90f6d19ae388: Downloading
[==============================================>    ]  17.82MB/19.26MB

aa18ad1a0d33: Downloading [==>
]  2.144MB/52.6MBload complete

aa18ad1a0d33: Pull complete
```

```
90f6d19ae388: Pull complete

94273a890192: Pull complete

c9110c904324: Pull complete

788d73c0fb6b: Pull complete

f221bb562f24: Pull complete

14414a6a768d: Pull complete

af6d2b2ad991: Pull complete

Digest:
sha256:080488acfe59bae32331ce28373b752f3f848be8b76c2c2d8fdce2820533650
4

Status: Downloaded newer image for node:8.4

 ---> 386940f92d24

Step 2/6 : COPY . /app

 ---> c8f9e3f487fc

Removing intermediate container 8c04a80cf68f

Step 3/6 : WORKDIR /app

 ---> 2e20b2e957d0

Removing intermediate container 3469e18d18fc

Step 4/6 : RUN npm install

 ---> Running in b81e97a080be

npm info it worked if it ends with ok

npm info using npm@5.3.0

....
```

```
....

up to date in 0.079s

npm info ok

 ---> 69aa699d74aa

Removing intermediate container b81e97a080be

Step 5/6 : EXPOSE 3000/tcp

 ---> Running in ba5e9d8a4842

 ---> 60ae58c57811

Removing intermediate container ba5e9d8a4842

Step 6/6 : CMD npm start

 ---> Running in 204ba4dca9e1

 ---> f1a88049ffd4

Removing intermediate container 204ba4dca9e1

Successfully built f1a88049ffd4

Successfully tagged sample:1.0

WARNING: Image for service app was built because it did not already
exist. To rebuild this image you must use `docker-compose build` or
`docker-compose upCreating sample_db ... done

Creating sample_app ... done

Creating sample_app ...
```

This command runs **docker-compose** in the background. If you want to see all of the output of docker-compose, run this command without the **-d** flag.

Remember, you will need to issue this command from inside the **test_compose** directory.

2.4 Verify that the container was created by using the command:

Copy

```
docker-compose ps
```

Output:

```
  Name                Command              State       Ports

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

sample_app   npm start                     Exit 254

sample_db    docker-entrypoint.sh mongod   Up         27017/tcp
```

2.5 Verify that the container was created in detailed information using **-a** flag:

Copy

```
docker ps -a
```

Output:

```
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS                      PORTS
NAMES

ae10c6ff799f        sample:1.0          "npm start"              3
minutes ago       Exited (254) 3 minutes ago
sample_app

3e585cda2a21        mongo:3.0.15        "docker-entrypoint..."   3
minutes ago       Up 3 minutes                27017/tcp
sample_db
```

3. Stopping and Deleting Containers

3.1 To stop all the containers in an application group, use the command:

Copy

```
docker-compose stop
```

Output:

```
Stopping sample_db ... done
```

3.2 After the containers have been stopped, you can remove all containers which were started with Docker Compose with the command:

Copy

```
docker-compose rm
```

Note: Press "**y**" to remove all containers.

Output:

```
Going to remove sample_app, sample_db

Are you sure? [yN] y

Removing sample_app ... done

Removing sample_db  ... done
```

3.3 To both stop all containers in an application group and remove them all with a single command, use:

Copy

```
docker-compose down
```

Output:

```
Removing network testcompose_samplenet
```

3.4 Verify that the docker-compose container is removed

Copy

```
docker-compose ps
```

Output:

```
Name   Command   State   Ports

------------------------------
```

3.5 Verify that the container was created in detailed information using **-a** flag:

Copy

```
docker ps -a
```

Output:

```
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

Copy

```
cd
```

4. Cleanup

4.1 To remove all the images run the below commands.

Copy

```
docker rmi `docker images -q` -f
```

4.2 Verify that containers are removed:

Copy

```
docker ps
```

Output:

```
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

4.3 Verify that docker images are removed:

Copy

```
docker images
```

Output:

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
```