Docker Lab 6
Using Docker Compose

1 REFERENCES 1
2 COMMANDS COVERED 1
3 LAB SETUP 1
4 BASIC DOCKER COMPOSE FOR A SINGLE CONTAINER 2
5 ADDING MULTIPLE CONTAINERS 4

1 References

The official reference for all Docker Compose file elements:
https://docs.docker.com/compose/compose-file/

2 Commands covered

3 Lab setup

This is identical to the previous lab

The root folders for all the various JavaScript / Node projects here can be found in the `Lab 6` subfolder in the main `labcode` folder of your downloaded zip for this workshop.

4 Docker Compose

Docker Compose is a tool for defining and running multi-container applications. Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single, comprehensible YAML configuration file. Then, with a single command, you create and start all the services from your configuration file.

Compose works in all environments; production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:

Start, stop, and rebuild services

View the status of running services
Stream the log output of running services
Run a one-off command on a service

5 Basic Docker Compose for a single container

The main folder for this project is: `compose-01-starting-setup`
This contains both the `frontend` and `backend` subfolders, which are the root project folders for these two apps respectively.

Create the Docker Compose file (`docker-compose.yaml`) in the main folder which contains both the `frontend` and `backend` subfolders

```yaml
services:
  mongodb:
    image: 'mongo'
    volumes:
      - data:/data/db
    environment:
       MONGO_INITDB_ROOT_USERNAME: max
       MONGO_INITDB_ROOT_PASSWORD: secret

volumes:
  data:
```

Note that these 2 different formats for specifying environment variables are equivalent:

```
MONGO_INITDB_ROOT_USERNAME: max

- MONGO_INITDB_ROOT_USERNAME=max
```

In the project folder holding the docker compose file, start up all the services specified in the Docker Compose file with:

```
docker compose up
```

This starts the service / container specified in the Docker Compose file in attached mode.

Notice that the messages clearly indicate that a default Docker network will be created together with the creation of the container specified in the Docker Compose file.

```
[+] Running 2/2
 ✓ Network compose-01-starting-setup_default     Created                    0.1s
 ✓ Container compose-01-starting-setup-mongodb-1  Created                   0.1s
```

The names of the containers and network are both prepended with the name of the folder holding the Docker Compose file: `compose-01-starting-setup`. This is the default behavior of Docker Compose.
All the containers created in a Docker Compose file will be automatically placed in the default generated Docker network. There is no need to explicitly specify this, for e.g. in the way that you would do using the `--network` option when starting a container via the `docker run` command.

Once the container is up and running, you can terminate from the attached container anytime with Ctrl+C.

You can alternatively start all the specified services / containers in detached mode with:

```
docker compose up -d
```

To see all the containers associated with the various services specified in the Docker Compose file, type:

```
docker compose ps
```

The usual `docker ps` is also possible, however docker ps will show ALL running containers regardless of whether they were started up via the Docker Compose file, while docker compose ps only shows containers that were started from that given Docker Compose file.

You can verify this for yourselves by starting a few random containers such as:

```
docker run -d --name spiderman -it alpine
```

```
docker run -d --name superman -it alpine
```

and then trying both `docker compose ps` and `docker ps`

Check for the data volume specified in the container with:

docker volume ls

Notice its name has been appended with the name of the folder containing the Docker Compose file: `compose-01-starting-setup`

To stop and remove all services specified in the Docker Compose file, we type:

```
docker compose down
```

This does not remove data volumes specified in the Docker Compose file. Check that the volumes still exist with:

```
docker volume ls
```

To remove these data volumes after bringing down the services, we need to additionally type:

```
docker compose down -v
```

6 Adding multiple containers

Create a subfolder `env` in the top level folder to place an environment file `backend.env`

```
MONGODB_USERNAME=max
MONGODB_PASSWORD=secret
```

Add in definition for the backend Node container in the Docker compose file:

```
services:
  mongodb:
    image: 'mongo'
    volumes:
      - data:/data/db
    environment:
       MONGO_INITDB_ROOT_USERNAME: max
       MONGO_INITDB_ROOT_PASSWORD: secret

  backend:
    build: ./backend
    ports:
      - '80:80'
    volumes:
      - logs:/app/logs
      - ./backend:/app
      - /app/node_modules
    env_file:
      - ./env/backend.env
    depends_on:
      - mongodb

  #frontend:

volumes:
  data:
  logs:
```

Since we have configured the values for our authentication credentials to the MongoDB directly in the env file and are referencing this file in the Docker Compose, we no longer need to hard code these credentials as environment variables in the Dockerfile that is being use to build the backend container

```
FROM node:14

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 80

#Remove these two environment variables below
#ENV MONGODB_USERNAME=root
#ENV MONGODB_PASSWORD=secret

CMD ["npm", "start"]
```

A best practice from a security viewpoint is to keep all sensitive info such as authentication credentials specified in an environment file that is only stored locally on a secure machine. The Dockerfile and Docker Compose file are likely to be uploaded to a public Git repo and are more sensitive to exploitation, and should therefore not contain these credentials.

Based on this concept, we should ideally also remove the credentials for the MongoDB container that we have placed in the Docker Compose file as below, and move them to an environment file, the same way that we have done for the backend container.

```
services:
  mongodb:
    image: 'mongo'
    volumes:
      - data:/data/db
    environment:
       MONGO_INITDB_ROOT_USERNAME: max
       MONGO_INITDB_ROOT_PASSWORD: secret
```

However, we will keep them here in this example just to demonstrate how environment variable values can be specified directly in the Docker Compose file and also in a separate environment variable file.

As usual, in the project holding the docker compose file, type:

```
docker compose up -d
```

To see all the containers associated with the various services specified in the Docker Compose file, type:

```
docker compose ps
```

Check the logs for the backend container

```
docker logs compose-01-starting-setup-backend-1
```

Bring down all the services:

```
docker compose down
```

Finally, add in definition for the frontend Node container in the Docker compose file:

```
services:
  mongodb:
    image: 'mongo'
    volumes:
      - data:/data/db
    environment:
       MONGO_INITDB_ROOT_USERNAME: max
       MONGO_INITDB_ROOT_PASSWORD: secret

  backend:
    build: ./backend
    ports:
      - '80:80'
    volumes:
      - logs:/app/logs
      - ./backend:/app
      - /app/node_modules
    env_file:
      - ./env/backend.env
    depends_on:
      - mongodb

  frontend:
    build: ./frontend
    ports:
      - '3000:3000'
    volumes:
      - ./frontend/src:/app/src
    stdin_open: true
    tty: true
    depends_on:
```

```
      - backend

volumes:
  data:
  logs:
```

As usual, in the project holding the docker compose file, type:

```
docker compose up -d
```

To see all the containers associated with the various services specified in the Docker Compose file, type:

```
docker compose ps
```

You should now be able to see the app running at localhost:3000 and interact with it in the same way as in the past.

If you take down all the services and bring them up again:

```
docker compose down
```

```
docker compose up -d
```

You should be able to see the goal items persisted through the use of volumes that were specified in the Docker Compose file.

Sometimes you may want to force a rebuild of all the images generated from Dockerfile within the Docker Compose file (for reasons seen earlier) during the process of generating and bringing up all the services, you will then run:

```
docker compose up --build
```

Alternatively, if you just want to force a rebuild of the images generated from Dockerfile (not base images pulled from the DockerHub registry) without at the same time creating services from them, then perform:

```
docker compose build
```

You can also alternatively specify your own container name if you do not want the auto generated ones provided by Docker Compose (which can be very long), for e.g.

```
services:
  mongodb:
    image: 'mongo'
    container_name: mymongo
    volumes:
```

```
      - data:/data/db
    environment:
       MONGO_INITDB_ROOT_USERNAME: max
       MONGO_INITDB_ROOT_PASSWORD: secret

  backend:
    build: ./backend
    container_name: mybackend
    ports:
      - '80:80'
    volumes:
      - logs:/app/logs
      - ./backend:/app
      - /app/node_modules
    env_file:
      - ./env/backend.env
    depends_on:
      - mongodb

  frontend:
    build: ./frontend
    container_name: myfrontend
    ports:
      - '3000:3000'
    volumes:
      - ./frontend/src:/app/src
    stdin_open: true
    tty: true
    depends_on:
      - backend

volumes:
  data:
  logs:
```

You can start it up again in the usual way and check for the explicit assigned name of these containers:

```
docker compose up -d

docker compose ps
```