Docker Workshop
Lab 6
Using Docker Compose

1 References

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

https://docs.docker.com/compose/gettingstarted/

https://www.freecodecamp.org/news/what-is-docker-compose-how-to-use-it/

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

In this lab session, we will replicate the functionality of the previous lab, where we ran a bunch of Docker CLI commands with various options to get 3 different containers to communicate with each other on a shared network; but this time, through the use of a Docker Compose YAML file.

5 Basic Docker Compose for a MongoDB container

The main folder for this project is: `compose-demo`
This contains both the `frontend` and `backend` subfolders, which are the root project folders for these two apps respectively. These are the same projects that we used from the previous lab.

Create the Docker Compose file (`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:
```

Alternative names for the Docker Compose file that are supported include:

- `compose.yml`
- `docker-compose.yaml`
- `docker-compose.yml`

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

```
MONGO_INITDB_ROOT_USERNAME: max
```

```
- MONGO_INITDB_ROOT_USERNAME=max
```

Here, we are hardcoding the authentication credentials directly into the Docker Compose file, which is not good security practice just as is the case for a Dockerfile: however, we will leave it as it is just to demonstrate this approach of specifying the environment variables directly. Later, we will see how we can reference a local environment variable file which contains sensitive security info instead.

This Docker Compose specification provided above is equivalent to a standard CLI command that looks like this, which we have used before in a previous lab:

```
docker run --name mongodb -v data:/data/db --rm -d --network goals-net -e MONGO_INITDB_ROOT_USERNAME=mongoadmin -e MONGO_INITDB_ROOT_PASSWORD=secret mongo
```

In a Powershell terminal 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 all the services / containers specified in the Docker Compose file in attached mode. You will see a series of console messages which are essentially the log content from the MongoDB app.

As there is a lot of output from here, we will terminate from the attached container.
We can do this with Ctrl+C.

A better way is to start all the specified services / containers in detached mode with:

```
docker compose up -d
```

You will see a message that indicates 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-demo_default      Created                                    0.0s
 ✓ Container compose-demo-mongodb-1  Started
```

The names of the containers and network are both prepended with the name of the folder holding the Docker Compose file: `compose-demo`. 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 (`compose-demo_default`) . 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.

There is a distinction between the term **services and containers** in the context of a Docker Compose specification.

- A service is a definition or blueprint for creating one or more containers, with specifications on the image to use as well as configuration options like ports, volumes, environment variables, networks, etc.

- A container is a runtime instance of a service. Each time you scale a service or bring up the Compose app, Docker creates one or more containers per service.

We can check on the combined log output from all the services specified in the Docker Compose file (right now there is only the MongoDB container) with:

```
docker compose logs
```

You can remove all the services / containers that are associated with the Docker Compose file with:

```
docker compose down
```

And then start them up again in the same way 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 that command 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-demo`

In addition, you will probably see an anonymous volume that is created automatically by Docker Compose as a side result of the named volume mapping to `/data/db`

Stop and remove all services specified in the Docker Compose file again with:

```
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
```

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

```
docker compose down -v
```

You can try this for yourself by starting all the services again before ending and removing them with the command option above.

If you still have any anonymous volumes remaining in the volume listing, you can remove them with:

```
docker volume prune
```

This command deletes all the dangling volumes. Any kind of volume (whether anonymous or named) which still exists, but that is no longer attached to any running container is called a dangling volume.

6 Docker Compose for backend and frontend containers

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

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

Now we can add in the 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:
      context: ./backend
    ports:
      - '5000:80'
    volumes:
      - logs:/app/logs
      - ./backend:/app
      - /app/node_modules
```

```
    env_file:
      - ./env/backend.env
    depends_on:
      - mongodb

volumes:
  data:
  logs:
```

Notice how for the backend container, we have included all 3 different kinds of volumes (bind mount, named and anonymous) as well as a reference to the environment variable file we created earlier.

The equivalent of the above specification expressed as a Docker CLI command would be as follows:

```
docker run --name backend -v ${pwd}:/app -v logs:/app/logs -v /app/node_modules --rm -d -p 80:80 --env-file ./env/backend.env --network goals-net goals-node
```

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
```

To view the logs for a specific container (rather than all the containers specified in the Docker Compose file), we can directly reference that container name:

```
docker logs compose-demo-backend-1
```

Bring down all the services:

```
docker compose down
```

Finally, add in a definition for the frontend React development server 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:
      context: ./backend
    ports:
      - '5000:80'
    volumes:
      - logs:/app/logs
      - ./backend:/app
      - /app/node_modules
    env_file:
      - ./env/backend.env
    depends_on:
      - mongodb

  frontend:
    build:
      context: ./frontend
    ports:
      - '3000:3000'
    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 before.

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 Dockerfiles referenced within the Docker Compose file (for e.g. because you have modified source code) 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
```

6.1 Specifying custom container and network name

Finally, you can also alternatively specify your own custom container and also network names if you do not want the auto generated ones provided by Docker Compose, for e.g.

```
services:

  mongodb:
    image: 'mongo'
    container_name: mymongo
    volumes:
      - data:/data/db
    networks:
      - webapp-net
    environment:
       MONGO_INITDB_ROOT_USERNAME: max
       MONGO_INITDB_ROOT_PASSWORD: secret

  backend:
    build:
      context: ./backend
    container_name: mybackend
    ports:
      - '5000:80'
    volumes:
      - logs:/app/logs
      - ./backend:/app
      - /app/node_modules
    networks:
      - webapp-net
    env_file:
      - ./env/backend.env
    depends_on:
      - mongodb

  frontend:
```

```
    build:
      context: ./frontend
    container_name: myfrontend
    ports:
      - '3000:3000'
    depends_on:
      - backend

volumes:
  data:
  logs:

networks:
  webapp-net:
```

Here, we explicitly declare our own network `webapp-net` and place the containers `mybackend` and `mymongo` into it with the appropriate elements. Recall from an earlier lab that we did not need to place `myfrontend` into this network, since we only need to fetch the React app from the development server at port 3000 in our browser and the React app will then directly communicate with `mybackend` at localhost port 5000.

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
```

and also check for the networks

```
docker network ls
```

When you are satisfied, bring down all the containers with:

```
docker compose down
```