Docker Compose

olsen software

Contents

1. Linking Containers Manually

- Overview
- Non-containerized application
- Containerized application
- Building the MySQL image
- Building the application image
- Running containers

Overview

- In a realistic application, you'll have lots of containers. For example:
  - Container #1 running MySQL (or MongoDB, etc.)
  - Container #2 running a Spring app, which talks to container #1
- In this chapter we'll show how to link containers together
  - The Spring Boot app will get data out of the MySQL database
- In the lab you'll do likewise with MongoDB
  - You'll tweak a Spring Boot app to get data out of MongoDB
  - Both the app and MongoDB will be running as containers, cool ☺

A Non-Containerized Application

- In the demos folder, take a look at the `Before` folder
  - It's a Spring Boot app that accesses data in a MySQL database
  - It's not containerized yet

- `application.properties` specifies connectivity info
  - It assumes MySQL is running on localhost
  - We'll need to change this property if MySQL is in a container!

```
spring.datasource.url=jdbc:mysql://localhost:3306/MYSCHEMA?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=c0nygre
```

application.properties

Containerized Application

- In the demos folder, now take a look at the `After` folder
  - This is a containerized version of the app, with 4 new files…
- `Dockerfile-mysql` and `myschema.sql`
  - Builds a Docker image for MySQL, using the SQL script to create and populate tables
- `Dockerfile-app`
  - Builds a Docker image for the Spring Boot app
- `docker-compose.yaml`
  - Uses docker-compose to automate container creation and linkage
  - See later for details…

Building the MySQL image

- `Dockerfile-mysql` builds an image for MySQL
  - Nothing new here, we saw all this in the previous chapter ☺

```
FROM mysql:5.7.19

EXPOSE 3306

ENV MYSQL_ROOT_PASSWORD=c0nygre

COPY myschema.sql /docker-entrypoint-initdb.d
```

Dockerfile-mysql

- To build this image, run the following command:

```
docker build -f Dockerfile-mysql -t emps/mysql .
```

Building the Application Image (1 of 2)

- `Dockerfile-app` builds an image for the Spring app
  - Similar to the image for a Spring app in an earlier lab

```
FROM openjdk:11.0

ADD target/employeeSpringBootApp-0.0.1-SNAPSHOT.jar app.jar

RUN sh -c
  'echo spring.datasource.url=jdbc:mysql://mysql:3306/MYSCHEMA?serverTimezone=UTC
                                                    > application.properties'

RUN sh -c 'echo spring.datasource.username=root    >> application.properties'

RUN sh -c 'echo spring.datasource.password=cOnygre >> application.properties'

ENTRYPOINT ["java","-jar","/app.jar"]
```

`Dockerfile-app`

- Note it creates a new `application.properties` file
  - Will be located in the same folder as the JAR, inside the container
  - Overwrites `application.properties` from the JAR
  - Specifies connectivity to a machine named mysql, not localhost

Building the Application Image (2 of 2)

- Before you can build the image, you must compile Java code into `.class` files, and package them into a `.jar` file
  - The following command achieves this (we've already done it)

```
mvn package
```

- This generates a `.jar` file as follows:
  - `target/employeeSpringBootApp-0.0.1-SNAPSHOT.jar`

- Now you can build the app Docker image as follows:

```
docker build -f Dockerfile-app -t emps/app .
```

Running Containers

- Create a docker network for the two containers to interact

```
docker network create mysql-net
```

- Run a MySQL container as follows:
  - Note the `--net` option

```
docker run --name mysql --net mysql-net -d -  3306:3306 emps/mysql
```

- Run an application container as follows:
  - Note the `--net` option

```
docker run --name app --net mysql-net emps/app
```

2. Automating the Process via Docker Compose

- Overview
- How Docker Compose works
- Defining a Docker Compose configuration file
- Building images and running containers

Overview

- In the previous section, you ran each container individually
  - This is quite a manual process
  - You have to remember to get the ports and names correct
  - This is very error-prone!

- A better approach would be to automate the creation of Docker images and containers via a configuration file
  - You can achieve this using a tool called Docker Compose

How Docker Compose Works

- You supply a configuration file
  - By default, you name the file `docker-compose.yaml`
- The configuration file specifies:
  - A list of services (how to build an image and run a container)
  - What volumes or mount points are needed by the containers
  - How the containers are linked together

Defining a Docker Compose Configuration File (1)

- Here's the structure of the Docker Compose configuration file for our example:

```
version: '3.0'

services:

   mysql:
      # Details for the MySQL service …

   app:
      # Details for the Spring Boot app service …
```

docker-compose.yaml

Defining a Docker Compose Configuration File (2)

- Here's how we configure the service for MySQL
  - Note it binds the MySQL data folder to a host-machine directory

```
mysql:
   container_name: mysql

   build:
      context: .
      dockerfile: Dockerfile-mysql

   image: emps/mysql:1.0.0

   ports:
      - "3306:3306"

   volumes:
      - /docker/emps/mysql:/var/lib/mysql

   restart: always

   environment:
      MYSQL_ROOT_PASSWORD: c0nygre

   command: --explicit_defaults_for_timestamp
```

docker-compose.yaml

Defining a Docker Compose Configuration File (3)

- Here's how we configure a service for our Spring app
  - Note the link to the mysql container

```
app:

   container_name: app

   build:
      context: .
      dockerfile: Dockerfile-app

   image: emps/app:1.0.0

   links:
      - mysql:mysql
```

docker-compose.yaml

Building Images and Running Containers

- You can run the Docker Compose file as follows
  - Builds images if they don't already exist
  - Runs container instances

```
docker-compose up
```

Any Questions?