Lab 6: Composing a Flask-MySQL app with docker and docker-compose

Lab submission requirements are in RED.

Dockerize and Compose a web app based on Flask and MySQL using Docker and Docker Compose.

It's considered a best practice for a container to have only one responsibility and one process, so for our app we will need at least two containers. One for running the app itself, and one for running the database. How do we coordinate these containers? This is where *docker-compose* comes in.

Docker 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.

Reference: https://docs.docker.com/compose/overview/

You should have Docker Compose already installed from installing Docker Desktop.

```
% docker compose
Usage:  docker compose [OPTIONS] COMMAND
Define and run multi-container applications with Docker
Options:
      --all-resources              Include all resources, even those not used by services
      --ansi string                Control when to print ANSI control characters
("never"|"always"|"auto") (default "auto")
      --compatibility              Run compose in backward compatibility mode
      --dry-run                    Execute command in dry run mode
      --env-file stringArray       Specify an alternate environment file
  -f, --file stringArray           Compose configuration files
      --parallel int               Control max parallelism, -1 for unlimited (default -1)
      --profile stringArray        Specify a profile to enable
      --progress string            Set type of progress output (auto, tty, plain, json, quiet)
(default "auto")
      --project-directory string   Specify an alternate working directory
                                   (default: the path of the, first specified, Compose file)
  -p, --project-name string        Project name
```

If not, here are the installation instructions. You should not need to do this.
https://docs.docker.com/compose/install/

Part 1 - Compose an App

Create the following directory structure for your app.

```
% mkdir docker-compose-lab
% cd docker-compose-lab
% mkdir app
% mkdir db
% cd app
```

Create a Flask app.

First, create app.py.
Contains the Flask app which connects to the database and exposes one REST API endpoint.

```
from flask import Flask
import mysql.connector
import json

app = Flask(__name__)

def favorite_colors() -> list[dict]:
    config = {
        'user': 'root',
        'password': 'root',
        'host': 'db',
        'port': '3306',
        'database': 'knights'
    }
    connection = mysql.connector.connect(**config)
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM favorite_colors')
    results = [{name: color} for (name, color) in cursor]
    cursor.close()
    connection.close()

    return results

@app.route('/')
def index() -> str:
    # return json.dumps({'favorite_colors': favorite_colors()})
    return 'boo!'

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

Second, create requirements.txt

```
Flask
mysql-connector
```

Third, create a Dockerfile for creating our Docker image.

```
# Use an official Python runtime as an image
FROM python:3.11


# EXPOSE instruction indicates the ports on which a container
# will listen for connections
# Flask apps listen to port 5000 by default, so we expose it
EXPOSE 5000

# WORKDIR sets the working directory for following COPY and CMD
# instructions
# Notice we haven’t created a directory by this name - this
# instruction creates a directory with this name if it doesn’t exist
WORKDIR /app

# Install any needed packages specified in requirements.txt
COPY requirements.txt /app
RUN pip install -r requirements.txt

# Run app.py when the container launches
COPY app.py /app
CMD python app.py
```

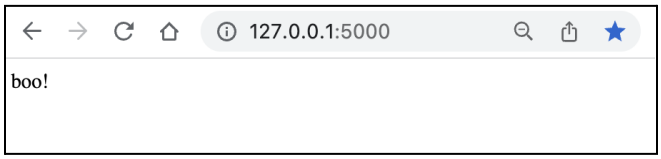
```
Your app subdirectory should have the following.
% ls app
Dockerfile      app.py              requirements.txt
```

Let's test out your web app.

```
% python -m venv .venv
% source .venv/bin/activate
% pip install -r requirements.txt
% python app.py
```

```
(.venv) (my_quant_lab) jayurbain@Jays-MBP docker-compose-lab % python app.py
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.1.151:5000
Press CTRL+C to quit
127.0.0.1 - - [26/Mar/2023 08:49:24] "GET / HTTP/1.1" 200 -
```

Hey, we're getting good at this!



Ctrl+C to stop.

Before we forget, enable the database request in our app.

Comment out:
`return 'boo!'`
And uncomment:
`return json.dumps({'favorite_colors': favorite_colors()})`

If we wanted to first test the database portion of the app, we could install mysql. But we're busy and have enough to do!

Create the db app

Move to the *db* subdirectory and create a MySql *init.sql* SQL script to initialize the database before the first time the app runs.

```
% cd ../db
% nano init.sql

CREATE DATABASE knights;
use knights;

CREATE TABLE favorite_colors (
  name VARCHAR(20),
  color VARCHAR(10)
);

INSERT INTO favorite_colors
  (name, color)
VALUES
  ('Lancelot', 'blue'),
  ('Galahad', 'yellow');
```

Composing our app

Move to the application root directory

```
% cd ..
% ls
app  db
```

Create a *docker-compose.yml* in your application root directory *docker-compose-lab.* This is what worked for me. Depending on your version of docker compose ($ docker-compose –version) you may have to make some minor adjustments.

```
version: "2.4"
services:
  app:
    build: ./app
    links:
      - db
    ports:
      - "5000:5000"
  db:
    image: mysql:5.7
    platform: linux/x86_64
    ports:
      - "32000:3306"
    environment:
      MYSQL_ROOT_PASSWORD: root
    volumes:
      - ./db:/docker-entrypoint-initdb.d/:ro
```

IMPORTANT: If you're running a MAC M1 or M2 Arm processor, change `mysql:5.7` to `mariadb` as follows: `image: mariadb`

Explanation:
We are using two **services**, one is a container which exposes the REST API (app), and one contains the database (db).

build: specifies the directory which contains the Dockerfile containing the instructions for building this service.

links: links this service (app) to another container (db). This will also allow us to use the name of the service instead of having to find the IP address of the database container, and express a dependency which will determine the order of start up for the containers.

ports: mapping of <Host>:<Container> ports.

image: Like the FROM instruction from the Dockerfile. Instead of writing a new Dockerfile, we are using an existing image from a repository. This exhibits some of the power of containerized apps. We write the core part of our application and *compose* the rest with existing containers. It's important to specify the version. If your installed mysql client is not the same version, problems may occur.

platform: linux/x86_64:

environment: add environment variables. The specified variable is required for this image, and as its name suggests, configures the password for the root user of MySQL in this container. More variables can be specified here. I know, I know, I know. I shouldn't use root for the password. But I'm an aging professor who likes to experiment rather than being a good DBA.

ports: Since we might have a running mysql instance on our host using the default 3306 port, we're mapping it to a different one. You may not need to do this. Notice that the mapping is only from host to container, so our app service container will still use port 3306 to connect to the database.

volumes: Since we want the container to be initialized with your database schema, wire the directory containing our init.sql script to the entry point for this container, which by the image's specification runs all .sql scripts in the given directory.

Double check your directory structure.

```
% ls
app             db              docker-compose.yml
```

Before we start the dockerized app, review the code connecting to the database in app.py:

```
config = {
        'user': 'root',
        'password': 'root',
        'host': 'db',
        'port': '3306',
        'database': 'knights'
    }
connection = mysql.connector.connect(**config)
```

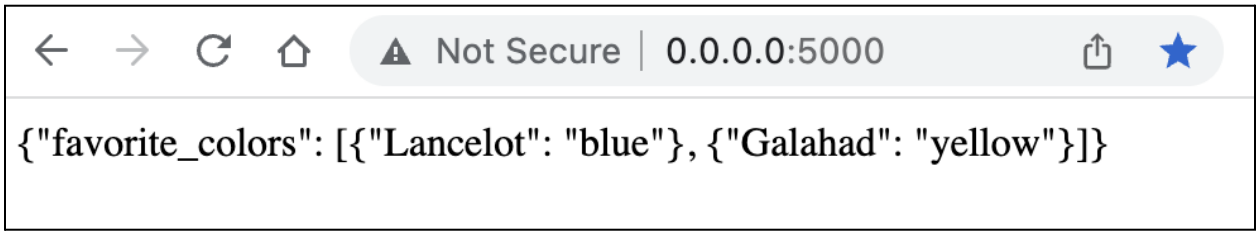
We are connecting as *root* with the password configured in the docker-compose file. Notice that we explicitly define the host (which is localhost by default) since the SQL service is actually in a different container than the one running this code. We can (and should) use the name 'db' since this is the name of the service we defined and linked to earlier, and the port is 3306 and not 32000 since this code is not running on the host.

Running the app

In order to run your dockerized app, execute the *docker-compose up* command.

```
 % docker compose up

[+] Running 12/12
 ⠿ db Pulled
33.4s
   ⠿ 2ec521688c56 Pull complete
…
```




```
Ctrl+C to stop
```

Stop in utter amazement for what you have just done. Your first microservices app!

Use the detach option to run docker-compose in the background.

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

List running compose projects

```
% docker compose ls
% docker-compose ls
NAME                STATUS              CONFIG FILES
docker-compose-lab   running(2)
/Users/jayurbain/Dropbox/MSOE/Microservices/docker-compose-lab/docker-compose.yml
```

Stop services

```
% docker compose stop

% docker compose ls
NAME                STATUS              CONFIG FILES
```

Wow! This is really powerful. Think of the apps you could compose with your docker app leveraging the power of all of the existing docker images out there.

Part 1: Submit your docker compose yml file and a snapshot of your running app.

Part 2: Compose your own app.

Using the app from above or starting from scratch, compose your own app. Your app must consist of at least 2 components. If you use your app from step 1, replace at least one component.

Here are several examples:

https://docs.docker.com/compose/support-and-feedback/samples-for-compose/

https://github.com/docker/awesome-compose/tree/master

Part 2: Submit your code, Dockerfile, docker compose yaml file and a snapshot of your running app.

Please submit part 1 and part 2 submissions, along with feedback on the lab as a single pdf document to canvas.

Notes:
The docker-entrypoint-initdb.d folder will only be run once while the container is created (instantiated) so you actually have to do a

```
% docker-compose down -v
```

to re-activate this for the next run.