Week 6. Session 3.
Cloud Bootcamp practical work.
Based on Docker documentation resources.

> WARNING: the `curl` command used below to test your app will not work on Windows. Use a browser address bar, or the VSCode Thunder Client extension instead.
>
> TIP: on Windows, run any `docker` commands from your system command prompt **after** confirming that Docker and Python are installed.
>
> TIP: on all OSs, ensure you activate the correct Python `venv` environment before running the Flask app, or installing new app dependencies. - On Mac/Linux: `source .venv/bin/activate` - On Windows: `.venv/Scripts/activate`

Developing Cloud Apps With Containers

Prerequisites

Work through the steps to build an image and run it as a containerized application the Cloud Bootcamp Week 6 Session 2 practical work.

Introduction

We'll walk through setting up a local development environment for the application we built previously. We'll use Docker to build our images and Docker Compose to make coordinating everything a whole lot easier.

Motivation

It is rare that an application developer creates all of the services needed to run an application from scratch. For example, databases and HTTP servers are both needed in many applications, but it would be impractical to develop these as well as your app code.

Instead, developers reuse other services inside their own applications, leading to many cloud applications being **collections of multiple services**, in some cases "microservices" when the responsibility of each service is very limited.

This worksheet takes you through building a **minimalistic multi-service cloud-deployable application** (deployment will be covered later).

We will create a web application service that runs Flask, and a database management service that runs MySQL. We will then use Docker to build images and run containers for these services, and configure how they interact with one another both on the network level, and on the data storage level using host volumes.

Run a database in a container

First, we'll take a look at running a database in a container and how we use volumes and networking to persist our data and allow our application to talk with the database. Then we'll pull everything together into a Compose file which allows us to setup and run a local development environment with one command.

Instead of downloading MySQL, installing, configuring, and then running the MySQL database as a service, we can use the Docker Official Image for MySQL and run it in a container.

Before we run MySQL in a container, we'll create a couple of volumes that Docker can manage to store our persistent data and configuration. Let's use the managed volumes feature that Docker provides instead of using bind mounts. You can read all about Using volumes in the official Docker documentation.

Let's create our volumes now. We'll create one for the data and one for configuration of MySQL.

```
$ docker volume create mysql
$ docker volume create mysql_config
```

Now we'll create a network that our application and database will use to talk to each other. The network is called a user-defined bridge network and gives us a nice DNS lookup service which we can use when creating our connection string.

```
$ docker network create mysqlnet
```

Now we can run MySQL in a container and attach to the volumes and network we created above. Docker pulls the image from Hub and runs it for you locally. In the following command, option `-v` is for starting the container with volumes. For more information, see Docker volumes.

```
 docker run --rm -d -v mysql:/var/lib/mysql\
  -v mysql_config:/etc/mysql -p 3306:3306\
  --network mysqlnet\
  --name mysqldb\
  -e MYSQL_ROOT_PASSWORD=p@ssw0rd1\
  mysql
```

Now, let's make sure that our MySQL database is running and that we can connect to it. Connect to the running MySQL database inside the container using the following command and enter "p@ssw0rd1" when prompted for the password:

```
$ docker exec -ti mysqldb mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.23 MySQL Community Server - GPL
```

```
Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Connect the application to the database

In the above command, we logged in to the MySQL database by passing the 'mysql' command to the `mysqldb` container. Press CTRL-D to exit the MySQL interactive terminal.

Next, we'll update the sample application we created during Cloud Bootcamp Week 6 Session 2. To see the directory structure of the Python app, review the documentation/videos from that session.

Now that we have a running MySQL, let's update the `app.py` to use MySQL as a datastore. Let's also add some routes to our server. One for fetching records and one for creating our database and table.

> IMPORTANT: if you are new to coding, then the additional code below may seem overwhelming. Don't worry if you don't understand it, simply read the in-code comments to get an idea of what is happening, and copy and paste the code into your existing `app.py` to replace what you had before.
>
> Most of the code involved is "standard boilerplate" meaning that it would appear in pretty much any application doing similar work: connecting to a database, sending queries to the database, and receiving the resulting data.

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

app = Flask(__name__)

# keep our old route
@app.route('/')
def hello_world():
    return 'Hello, Docker!'

# a new route to query the `widgets` table in our application DB
@app.route('/widgets')
```

```
def get_widgets():
    # connect to the database
    mydb = mysql.connector.connect(
        host="mysqldb",
        user="root",
        password="p@ssw0rd1",
        database="inventory"
    )

    # now query the database
    # you don't need to worry about how this code works!
    cursor = mydb.cursor()
    cursor.execute("SELECT * FROM widgets")
    row_headers=[x[0] for x in cursor.description] #this will extract row headers
    results = cursor.fetchall()
    json_data=[]
    for result in results:
        json_data.append(dict(zip(row_headers,result)))
    cursor.close()

    # once we have the data, we can return it to the client
    return json.dumps(json_data)

# a new route to create the database and table to be queried
@app.route('/initdb')
def db_init():
    # connect to the database manager
    mydb = mysql.connector.connect(
        host="mysqldb",
        user="root",
        password="p@ssw0rd1"
    )

    # create the database
    cursor = mydb.cursor()
    cursor.execute("DROP DATABASE IF EXISTS inventory")
    cursor.execute("CREATE DATABASE inventory")
    cursor.close()

    # connect to the database
    mydb = mysql.connector.connect(
        host="mysqldb",
        user="root",
        password="p@ssw0rd1",
        database="inventory"
    )
    # create the table
    cursor = mydb.cursor()
    cursor.execute("DROP TABLE IF EXISTS widgets")
    cursor.execute("CREATE TABLE widgets (name VARCHAR(255), description VARCHAR(255))")
    cursor.close()

    # return a string to confirm it worked
    return 'init database'

# this section allows us to run the app with `python app.py`
# on the command line
if __name__ == "__main__":
    app.run(host ='0.0.0.0')
```

We've added the MySQL module and updated the code to connect to the database server, created a database and table. We also created a route to fetch widgets. **We now need to rebuild our image so it contains our changes.**

First, let's add the `mysql-connector-python` module to our application using pip.

> IMPORTANT: run the following commands with the correct `venv` active, that you created previously.
>
> Otherwise, the installation will be to your system Python and not to the environment in which Flask is installed.

```
$ pip3 install mysql-connector-python
$ pip3 freeze | grep mysql-connector-python >> requirements.txt  # save the dependency
```

Now we can build our image.

```
$ docker build --tag python-docker-dev .  # IMPORTANT: don't forget the `.`
```

If you have any containers running from the previous sections using the name `rest-server` or port 8000, **stop them now**.

Now, let's add the container to the database network and then run our container. This allows us to access the database by its container name.

```
 docker run\
  --rm -d\
  --network mysqlnet\
  --name rest-server\
  -p 8000:5000\
  python-docker-dev
```

Let's test that our application is connected to the database and is able to add a note.

```
$ curl http://localhost:8000/initdb
$ curl http://localhost:8000/widgets
```

You should receive the following JSON back from our service.

```
[]
```

Use Docker Compose to Combine Container Services

In this section, we'll create a Compose file to start our python-docker and the MySQL database using a single command.

Open the `python-docker` directory in your IDE or a text editor and create a new file named `docker-compose.dev.yml`. Copy and paste the following commands into the file.

```
version: '3.8'

services:
 web:
  build:
   context: .
  ports:
  - 8000:5000
  volumes:
  - ./:/app

 mysqldb:
  image: mysql
  ports:
  - 3306:3306
  environment:
  - MYSQL_ROOT_PASSWORD=p@ssw0rd1
  volumes:
  - mysql:/var/lib/mysql
  - mysql_config:/etc/mysql

volumes:
  mysql:
  mysql_config:
```

This Compose file is super convenient as we do not have to type all the parameters to pass to the `docker run` command. We can declaratively do that using a Compose file.

We expose port 8000 so that we can reach the dev web server inside the container. We also map our local source code into the running container to make changes in our text editor and have those changes picked up in the container.

Another really cool feature of using a Compose file is that we have service resolution set up to use the service names (here `web` and `mysqldb`). Therefore, we are now able to use "mysqldb" in our connection string in the Flask application. The reason we use "mysqldb" is because that is what we've named our MySQL service in the Compose file.

> Note that we did not specify a network for the 2 services. When we use docker-compose (rather than `docker run` as we did before) it automatically creates a network and connects the services to it. For more information see Networking in Compose

If you have any containers running from the previous sections, stop them now.

Now, to start our application and to confirm that it is running properly, run the following command:

```
$ docker compose -f docker-compose.dev.yml up --build
```

We pass the `--build` flag so Docker will compile our image and then start the containers.

Now let's test our API endpoint. Open a new terminal then make a GET request to the server using the curl commands:

```
$ curl http://localhost:8000/initdb
$ curl http://localhost:8000/widgets
```

You should receive the following response:

```
[]
```

This is because our database is empty.

Monitoring Docker Compose Applications

A very helpful extension is available in VSCode to help manage Docker containers.

Install the Docker extension to use it.

Some handy functionality includes: - Overview of running containers and their status - Start/stop/restart any container - Prune/delete containers - Connect directly to the container logfiles in a new terminal - Connect directly to the container shell in a new terminal

Bootcamp Tasks

Basic Tasks

- Install the VSCode Docker extension mentioned above. Then after launching your Docker compose app, use the extension to load two terminals, one for each container. Check that the `web` service container has your code in the `/app` directory and that the `mysqldb` service container has the `mysql` command available (useful if you need to add or manage data in the container directly).

- Modify your Docker compose configuration `yaml` file so that the two services it defines are available on ports different than those currently published to the host (internally to the containers, the ports should stay the same).
- See if you can find information in the Docker compose docs or elsewhere to set up a *dependency* between the two services defined in the `yaml` file. Make the `web` service dependent on the `mysqldb` service.
- Review the docker compose documentation to find out how to stop all of the running containers with a single command. Can you work out how to remove the volumes the services are using in this command too?

Challenge Tasks

- Add a third service to your docker compose configuration. For example try adding the Nginx reverse HTTP proxy. Configure the proxy to pass requests to the `web` service. Remove port mappings from the `web` service so that accessing the service must be done through the proxy (this is how it is usually done in production environments).
- Investigate the use of multiple docker compose `yaml` files and in particular see if you can find out how to provide an "override" docker compose file which changes the default functionality of the main docker compose file. Override docker compose files are used by developers to change the default service configurations (which are used in production deployments) and run additional services that are only needed for development such as testing services. The overrides are not used in production, for efficiency and security. Write an override file that adds a `selenium` test service to the orchestrated services that are loaded (you'll find Selenium images on the Docker Hub).