Multi-service applications in the Cloud

1 A containerized chat room

We developed a simple chat room in Python that you can download here. The code has been adapted from this GitHub project.

Participants use a *client* program to connect to the chat room; the chat room is managed by a *server* application that receives the client connections and forwards the messages between the users. The archive contains the following files:

- *client.py*. Implementation of the chat room client.
- *server.py*. Implementation of the chat room server.
- *utils.py*. Library with utility functions used in both *client.py* and *server.py*.

Warning
Only one instance of the server is running. Several instances of the client can run at the same time. Client instances might not be running on the same host.
In order to execute the server, we need to specify the port number as a parameter.

Exercise
Exercise 1.1. Explain the methodology that you're going to follow to containerize the client and the server by using Docker. In particular explain:

1. How many images you're creating and why.
2. What you need to build the images.
3. How you're going to build the images.

Solution

1. We need to create two images, one for the client and one for the server. We cannot just create one image, because the client and the server are two separate entities.
2. We need to create two Dockerfiles, one for the client and one for the server.
3. We can run the docker build command on the two Dockerfiles. Alternatively, we can have two docker-compose files and build the images by using docker-compose. Careful: since client and server are two separate entities (they can be run on different machines, they might not be run at the same time), having only one docker-compose file would not be a wise decision.

Exercise

Exercise 1.2. Containerize the application by following the methodology that you outlined in Exercise 1.1.

> – **Warning**
> The size of the images should be kept as small as possible. Remember to include in the images only what you need to build and run the application.

Solution

For both the client and the server we need a Python environment. We use the Python environment *slim* as a base image to minimize the size of the image. Students who choose this base image are rewarded. If they choose another (bigger) Python environment, that is fine, but they don't get the totality of the points. If they use an OS as a base image (e.g., ubuntu), they get penalized.
The Dockerfile for the client (let's call it *Dockerfile-client*) is as follows.

```
FROM python:3.7-slim
RUN mkdir -p /app
WORKDIR /app
COPY ./client.py ./utils.py /app/
ENTRYPOINT ["python", "client.py"]
```

We build the image with the following command:
`docker build -t chat-client -f Dockerfile-client .`
The Dockerfile for the server (let's call it *Dockerfile-server*) is as follows.

```
FROM python:3.7-slim
RUN mkdir -p /app
WORKDIR /app
COPY ./server.py ./utils.py /app/
ENTRYPOINT ["python", "server.py"]
```

We build the image with the following command:
`docker build -t chat-server -f Dockerfile-server .`

Exercise

Exercise 1.3. If you built more than one image:

1. Do they have some common layers?
2. If so, is it something that has an impact on the build time and how?

Solution

The base image is the same for both images. Assuming that we build the client first, this build will take longer if the python-slim image has not been downloaded yet from the registry. The layers 2 and 3 are identical, so when we build the server they're going to be already there (this is indicated by the keywords CACHED when we build the server). It is important that here the students talk about the build cache mechanism.

1.1 Client and server on the same network

We want to execute:

- One instance of the server.
- Two instances of the client.

The server, as well as the clients, will **run in Docker containers attached to the same network**.

Exercise
Exercise 1.4. We want to make sure that:

1. The server and the clients can communicate with each other.
2. Other Docker containers **cannot** communicate neither with the server nor with the clients.

How can you satisfy both requirements in Docker? Explain your solution and justify it.

Solution

If we create the three containers without specifying any network configuration, then the three containers will be attached to the default `bridge` network. This way, we can satisfy the first requirement, but not the second (other containers attached to `bridge` will be able to communicate with the clients and the server).
We need to create a new network in Docker, let's call it `chat-room` that will be used exclusively for the server and the two clients.

Exercise
Exercise 1.5. Execute the server and the two clients by using the network configuration that you explained in the previous exercise.

- The server writes messages on the terminal. Remember to launch the container with the appropriate option in order to actually see those messages.
- Users need to interact with the client, that is: read and write messages. Remember to launch the container with the appropriate option in order to actually see those messages.

Write the **exact commands** that you typed for both configuring the network and launching the server and the clients.

Solution

First, we create the new network `chat-room` by typing the following command:

```
docker network create chat-room
```

Next, we launch the server with the following command:

```
docker container run --rm -t --name chat-server --network chat-room
chat-server 60876
```

The server takes in a port number as a parameter. If the students launch the server without specifying it, they'll get an error message that should lead them to understand what's missing.
In another terminal, we type the following command to run the first client:

```
docker container run --rm -it --name chat-client1 --network chat-room
chat-client chat-server 60876
```

Note that we attach the client to the same network as the server. When we launch the client, we need to pass as parameters the server host (here we can use the container's name `chat-server` of the server or its IP address) and the port number (same as we specified when we launched the server). When we launch the client, we'll need to specify a username and then we can start typing messages in the chat room.
Finally, we open another terminal to launch the second client with the following command:

```
docker container run --rm -it --name chat-client2 --network chat-room
chat-client chat-server 60876
```

The only thing that changes wrt the previous command is the name that we give the new container (`chat-client2`).
We can play with the chat to verify that the clients can exchange messages.

Shutting down the application
Shut down both the clients and the server before you move on.

- On the client-side, type `#quit` at any moment to exit the chat room.
- Type Ctrl-C to stop the server.

1.2 Client and server on different networks

We want to execute:

- One instance of the server.
- Two instances of the client.

However, neither client is connected to the same network as the server.

Exercise
Exercise 1.6. Why isn't the solution that you gave in Exercise 1.5 working?
What do you propose as a solution instead?
HINT. You might want to look at slide 77 of the second lecture.

Solution

The hostname `chat-server` is not reachable from the clients, because they're not in the same network as the server. So, the only way out is to use the mechanism of port mapping. That is: we launch the server on the container port 60876 but we also use the option `-p` to map that port to any available port (say, 7070) on the **host computer**. This way, the clients can connect to the server by specifying the IP address of the host machine and 7070 as the port number.

Exercise
Exercise 1.7. Execute the server and the two clients by making sure that neither client is connected to the same network as the server.
Write the **exact commands** that you typed for both configuring the network and launching the server and the clients.

Solution

As before, we can still connect the server to the network `chat-room`. However, since we want our server to be reachable from outside that network, we need to publish a port on the host. The option `-p` in the following command maps the container internal port 60876 to the port 7070 on the host computer.

```
docker container run --rm -it --name chat-server --network chat-room -p
7070:60876 chat-server 60876
```

Now, let's create a new network that we call `net-clients` to which we'll be attaching the two clients:

```
docker network create net-clients
```

And we finally launch the two clients. As the server of the host, we need to type the IP address assigned to the host. In my case, the IP address assigned to my host is 192.168.1.8

```
docker container run --rm -it --name chat-client1 --network net-clients
chat-client  192.168.1.8 7070
```

The second client is launched as follows:

```
docker container run --rm -it --name chat-client2 --network net-clients
chat-client  192.168.1.8 7070
```

2 Multi-service application: Docker Compose

We intend to build and deploy the *TripMeal* application by using **Docker Compose**. You can download the application here.

Credit
The source code has been readapted from this GitHub repository.

The downloaded file is an archive. Extracting the archive will create a folder named `tripmeal`.

Exercise
Exercise 2.1. Explain the structure of the content of the folder `tripmeal`.

Solution

The folder contains:
- A file `docker-compose.yml` that will contain the instructions to build and deploy the application.
- A subdirectory for each service composing the application. Each subdirectory contains the source code of the corresponding service, as well as a Dockerfile with the instructions to build an image for the service.

Exercise
Exercise 2.2. How many services has the application? Specify the technologies used to implement each service.

Solution

The application consists of two services:

- web: It is a web application, developed with a combination of HTML, Python and Flask
- db: It is a relational database. From the Dockerfile, which is already given, we understand that the DBMS used is MySQL.

The `Dockerfile` in the directory `db` is already implemented.

Exercise
Exercise 2.3. Write the `Dockerfile` in the directory `web`.

Solution

Different solutions exist, here is a proposition. It is important that we build a minimal image, so we choose the environment python:3-7-slim.

```
FROM python:3.7-slim
RUN mkdir -p /app &
    mkdir -p /app/templates &
    mkdir -p /app/static
RUN /usr/local/bin/python -m pip install --upgrade pip
COPY ./static /app/static/
COPY ./templates /app/templates/
COPY requirements.txt /app/
WORKDIR /app
RUN pip install -r requirements.txt
COPY app.py dbconnect.py /app/
ENTRYPOINT ["python","app.py"]
```

In the project folder, you'll find a file named `tripmeal.env`. It contains the definition of **environment variables** that are used in the application.

Exercise
Exercise 2.4. By reading the values of the environment variables, you can already know the values to assign to some of the keys in `docker-compose.yml`. Can you tell which ones?

Solution

- By reading the value of the variable SERVER_PORT we know the port where the container corresponding to the web service will be listening for incoming connections.
- By reading the value of the variable DATABASE_HOST we know the name that we must assign the container where the database will be running.

The other information are used internally by the database and/or the web service, they won't be useful to write the file `docker-compose.yml`

Exercise
Exercise 2.5. What do you need to create in file `docker-compose.yml` to enable the communication between the different services?

Solution

We need to define a network and attach both services to that network.

The database management system used by the application is MySQL. You can see it by reading the `Dockerfile` in the directory `tripmeal/db`.

Exercise
Exercise 2.6. Which base image is used to build a container for the database? Where is this base image stored? Can you find the documentation of this image in the Internet?

Solution

The base image is `mysql`. The tag of the image is `latest` since it is not specified. The image is stored in the DockerHub registry. In order to find it, we can simply look for an image called `mysql` on the DockerHub website. It turns out that the documentation is available at this page

Exercise
Exercise 2.7. By looking at the documentation of the database base image, can what do you need to do to store the data?

Solution

We need to create a volume. Moreover, we need to mount this volume to the directory `/var/lib/mysql` of the container where the database will run.

You're finally ready to complete the file `docker-compose.yml`.

Warning
Remember that you need to pass the **environment variables** to your application.
As a documentation of Docker Compose you can use:
- The examples that we've seen together.
- The overview presented on the official Docker website.
- You can also find the full specification of Compose here.

Exercise
Exercise 2.8. Write the file `docker-compose.yml`. Build, deploy and test your application.

Solution

```
version: "3"

services:
    web:
        build: web
        image: quercinigia/trip-meal-web
        env_file: tripmeal.env
        networks:
            - tripmeal-network
        ports:
            - "5000:5000"
    db:
        build: db
        image: quercinigia/trip-meal-db
        expose:
            - 3306
        networks:
            - tripmeal-network
        volumes:
            - db-data:/var/lib/mysql

volumes:
    db-data:

networks:
    tripmeal-network:
```

Activity
Push all the images that you build in this section to your DockerHub registry.

3 Multi-service application: Kubernetes

Deploy the application on Kubernetes.