***Cheat-Sheet***

*Microservices 7th - 8th Week*

*Docker & CRUD Order´s Microservice*

Microservices Architecture (Introduction)

Teacher: Mr. Zheng Li

Student: Juan Albornoz

# 7th week´s practice:

# Installed tools and platform

# The docker container

# Managing volumes for data persistency

# Linking database with another docker container

# Building and running with Docker-compose

# Linking test.py and MySQL

# Executing

# 8th week´s practice:

# MySQL tables modeling

# CRUD operations and definitions

# 2.1. READ operation

# 2.2. CREATE operation

# 2.3. UPDATE operation

# 2.4. DELETE operation

# Installed tools and platform

DockerToolbox-19.03.1 I couldn’t install the main docker resources because of a windows version problem, so I used Docker Toolbox as an alternative. (Windows 10 Home Single)

Django 3.0.7 already installed and working.

Also tried with Flask 1.1.2, although I started with Django and I’m not into it.

# The docker container

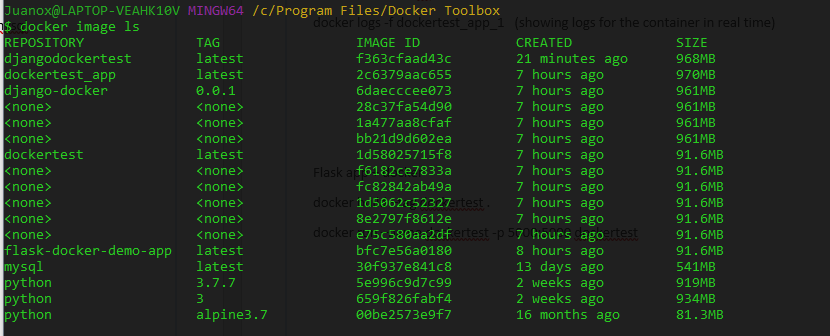
While I was trying to starting up the server for Django and Flask I had the opportunity to put my hands on docker commands a lot, some of those are the next:

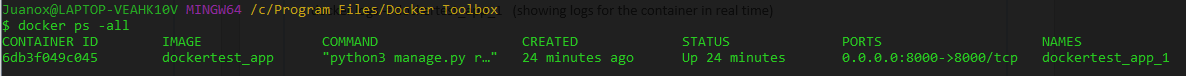
**$** **docker run hello-world**

Since the image was not able in my local, docker searched it for me in his sources.

**$** **docker run –rm hello-world**

Command for running an app and also remove it after.

**$** **docker image ls (list images)**

**$ docker ps -all** (list containers created and the run states of them)

# MySQL database in docker container

**$ docker run -d -p 33060:3306 --name mysql-db -e MYSQL\_ROOT\_PASSWORD=test mysql**

Command for running up a container with MySQL

**$ docker exec -it mysql-db mysql -p**

**$ create database testdb;**

**$ show databases;**

I can access the MySQL through the first command and create the database, show it, and all the database stuff.

TO KILL AND DELETE :

**$ docker kill mysql-db**

**$ docker rmi mysql-db**

**DELETE ALL IMAGES:**

**$docker image prune**

# Managing volumes for data persistency

Using volumes for data persistency in the DB:

**$ docker rm -f mysql-db**

I removed the container and killed it

**$ docker volume prune**

Delete all the existing volumes

**$ docker volume create mysql-db-data**

Create a volume for my database

**$ docker run -d -p 33060:3306 --name mysql-db -e MYSQL\_ROOT\_PASSWORD=secret –mount src=mysql-db-data,dst=/var/lib/mysql mysql**

Command for running up the MySQL database, but mounting it on my recent created volume for persistency.

# Building and running with Docker-compose

# Python flask web app + MySQL + docker [FILES AND DIRS]

**$ pip install docker-compose**

Compose is a tool for defining and running multi-container Docker applications. It uses an YML file to configure your application’s services.

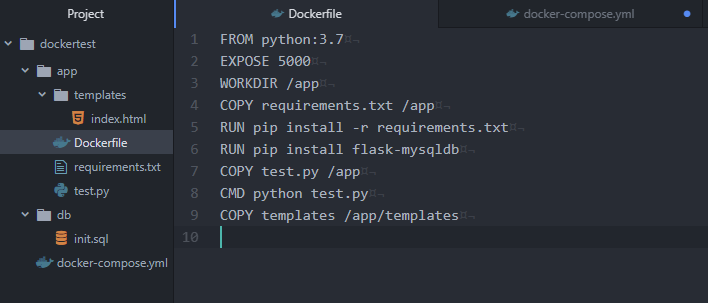
For building the image and running the container it requires three files:

docker-compose.yml for the app services configurations and running the containers

Dockerfile for the image building

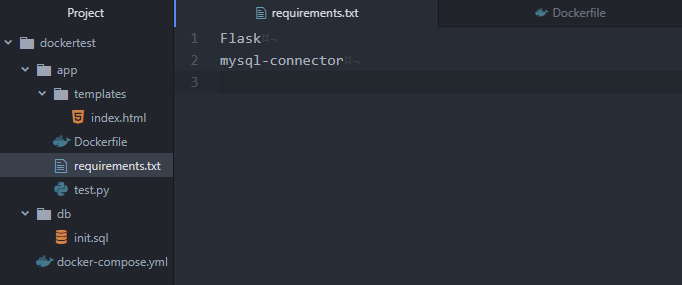
requirements.txt for the dependencies used by the image

The Dockerfile has the function of giving all the commands for building our image. This file also include the requirements.txt file for installing all the dependencies needed in the image. It’s saved inside the app.py directory.

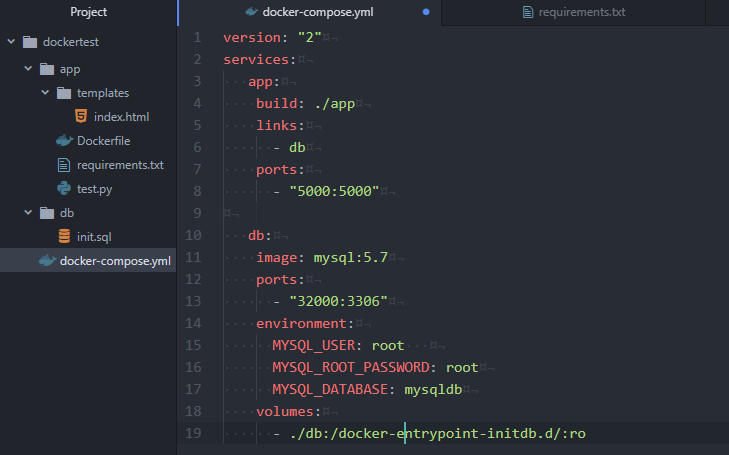


FROM, EXPOSE, WORKDIR, COPY, RUN and CMD are commands used by Docker.

The requirements.txt file is for dependencies and is also saved in the app.py directory:



The docker-compose.yml has the main services for the containers. Those are for the app itself and also for the MySQL database as it follows:

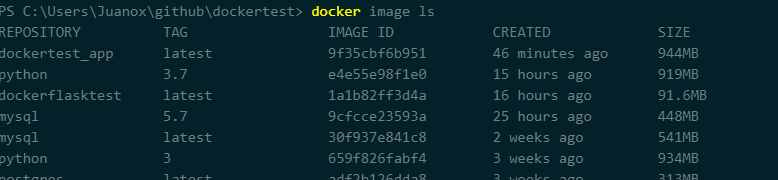


(\*) Notice that the docker-compose.yml file belongs to the Flask’s root directory

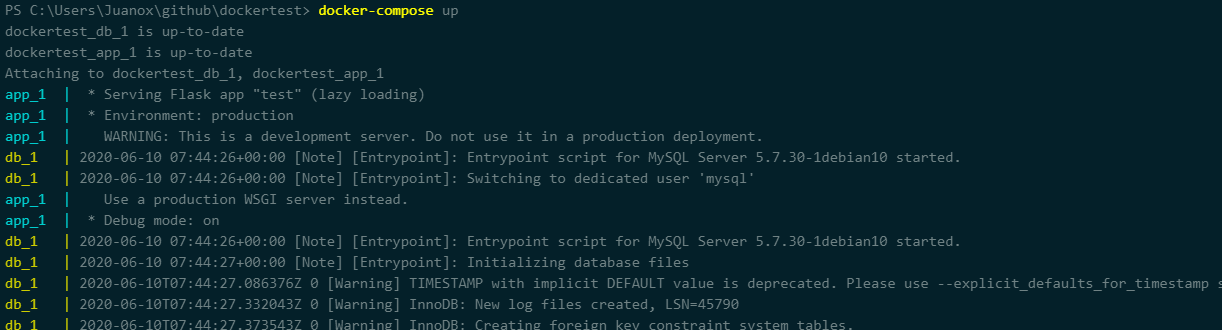
# Python flask web app + MySQL + docker [SHELL COMMANDS]

**$ docker-compose build**

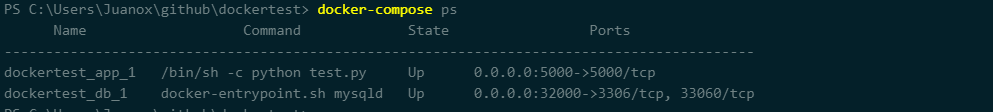
It takes the docker-compose.yml file and builds two images; the dockertest\_app image with and ID and a size and also the mysql 5.7 image as you can see:

$ **docker image ls**

**$ docker-compose up**

The command to create the containers and put them in running mode

**$docker-compose ps**



**TO KILL THE CONTAINERS:**

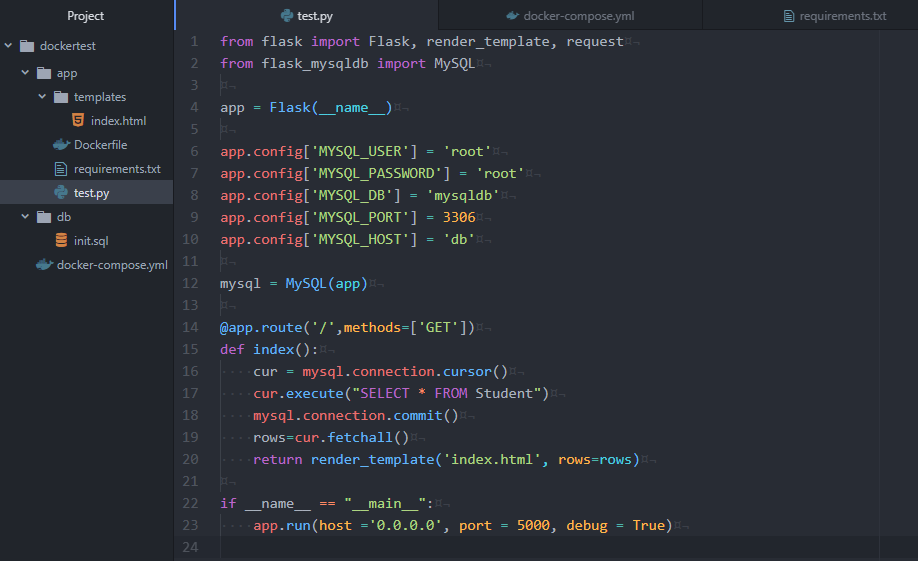
**$docker-compose down**

**TO REMOVE THE IMAGES:**

**$docker rmi dockertest\_app & $docker rmi mysql:version**

# Linking test.py and MySQL

* I used the MySQL module from flask\_mysqldb.
* mysql.connection.cursor() opens the connection with the database.
* execute() to giving instructions in SQL language.
* commit() to send it.
* fetchall() to fetch some few rows in my database.

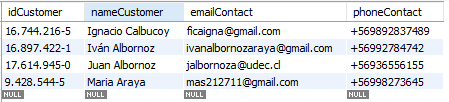


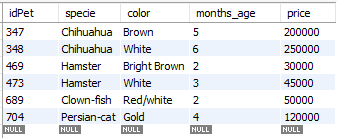
# Executing [192.68.99.100:5000]

# 8th week´s practice

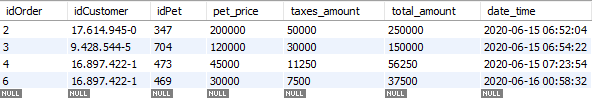
# MySQL tables modeling

Pet & costumer tables: These tables are just for a proper implementation of CRUD operations on the Order table.

Customer: Primary Key: *idCustomer*

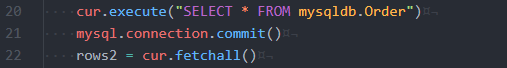
Pet: Primary Key: *idPet*

Order: Primary Key: *idOrder* and Foreign Key: *idCostumer*, *idPet*

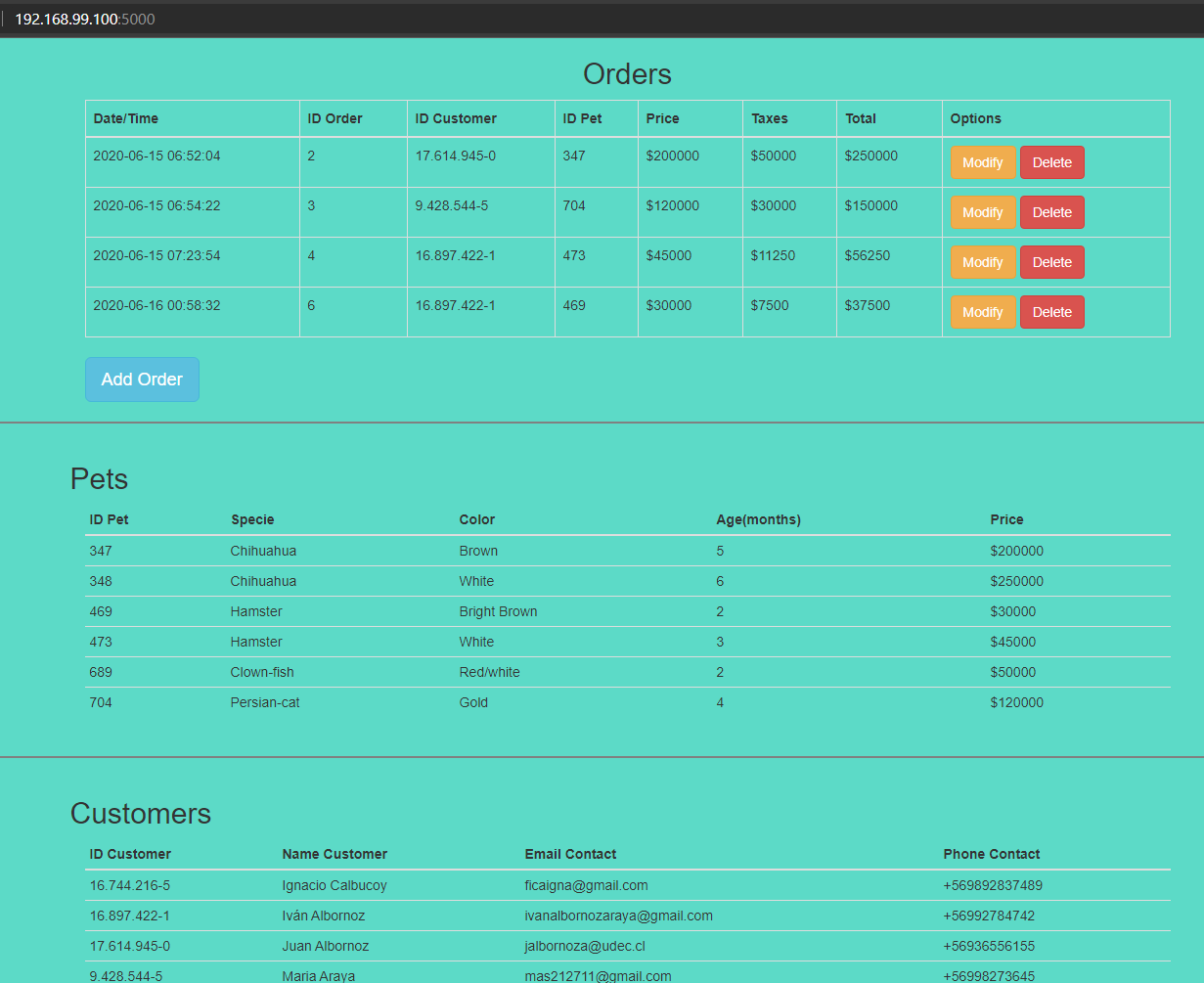
(\*) *taxes\_amount, total\_amount and date\_time* are autogenerated fields

# CRUD operations and definitions

# 2.1. READ operation

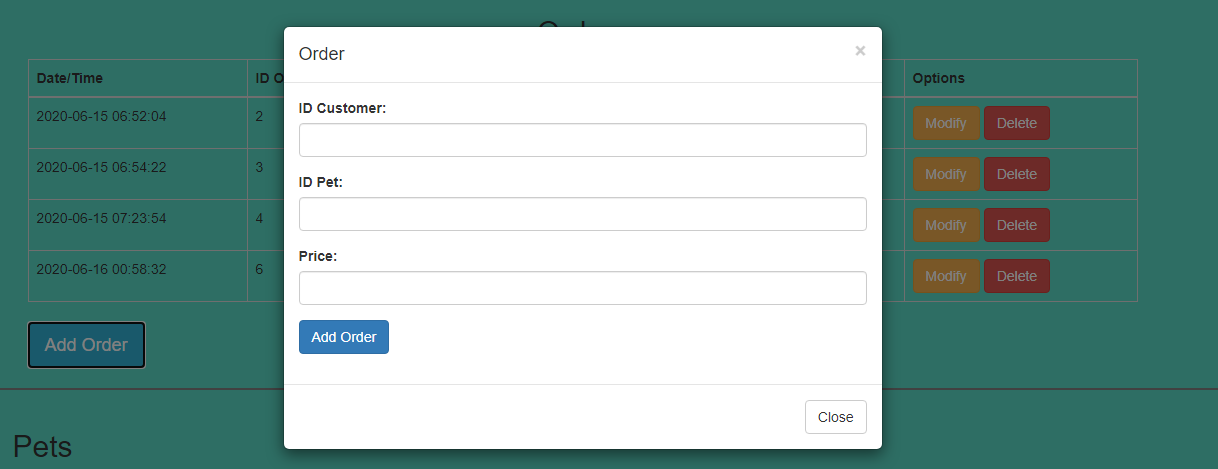
Read operations are implemented in the index()controllerfunction:

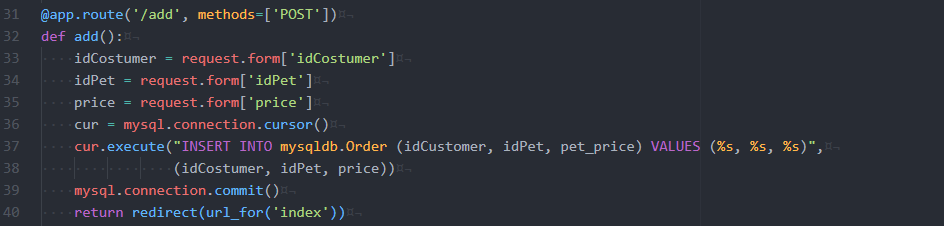
It executes the SQL instruction and fetches all the rows to render them in the template´s table as it follows:

****(\*) I used Bootstrap tables to render the data in the html file.

# 2.2. CREATE operation

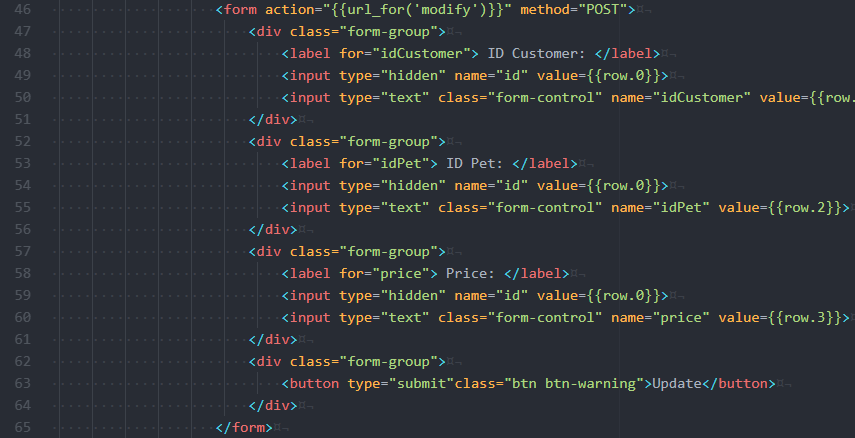
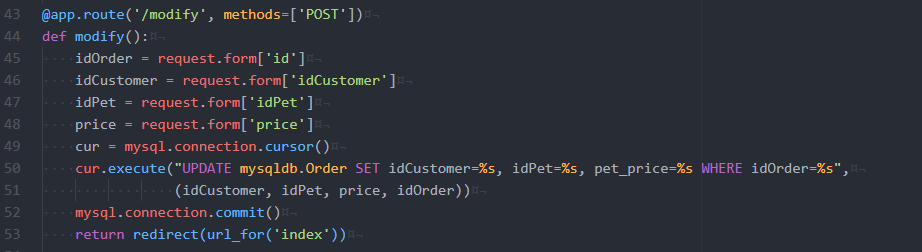
By using forms implemented in the flask environment I can catch the data entered in the template and send it back to the controller via “POST” method and a defined route to the def add() function. The form to add the order is the next:

(\*) I used *modal* resource from Bootstrap for showing the form in a kind of emergent window.

The data from the formulary is saves in variables and then passed to the SQL instruction. I’m using mysqldb from flask and it uses a format way to manage variables, in my case the variables are idCostumer, idPet and price required for creating a row in the Order table.

# 2.3. UPDATE operation

In a similar way to the CREATE operation, to update an Order def modify() uses “POST” method to receive the updated entries in the form (the same modal form used by CREATE). The main difference is that the form must be companied with the primary key as it follows (also catched by “POST”):

The controller with the def modify() function defined and the route statement: