Starting in docker

1.- Elements of Docker Image

In the case that we need to made a image of our project, we have to use a structure like this:

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip3 install -r requirements.txt

# This COPY command takes all the files located in the current directory
and copies them into the image

COPY . .

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

The detailed elements are the follow:

- **FROM**: Docker images can be inherited from other images. Therefore, instead of creating our own base image, we'll use the official Python image that already has all the tools and packages that we need to run a Python application.

- **WORKDIR**: This instructs Docker to use this path as the default location for all subsequent commands. By doing this, we do not have to type out full file paths but can use relative paths based on the working directory.

- **COPY**: The `COPY` command takes two parameters. The first parameter tells Docker what file(s) you would like to copy into the image. The second parameter tells Docker where you want that file(s) to be copied to. We'll copy the `requirements.txt` file into our working directory `/app`.

- **RUN**: Once we have our `requirements.txt` file inside the image, we can use the `RUN` command to execute the command `pip3 install`. This works exactly the same as if we were running `pip3 install` locally on our machine, but this time the modules are installed into the image.

- **CMD**: Tell Docker what command we want to run when our image is executed inside a container. *Note that we need to make the application externally visible (i.e. from outside the container) by specifying* `--host=0.0.0.0`

2.- Build an image

Now that we've created our Dockerfile, let's build our image. To do this, we use the `docker build` command. The `docker build` command builds Docker images from a Dockerfile and a "context". A build's context is the set of files located in the specified PATH or URL. The Docker build process can access any of the files located in this context.

The build command optionally takes a `--tag` flag. The tag is used to set the name of the image and an optional tag in the format `name:tag`. We'll leave off the optional `tag` for now to help simplify things. If you do not pass a tag, Docker uses "latest" as its default tag.

```
$ docker build --tag python-docker .
```

2.1 - View local images

To list images, simply run the `docker images` command.

```
$ docker images
```

2.2 - Tag images

To create a new tag for the image we've built above, run the following command.

```
$ docker tag python-docker:latest python-docker:v1.0.0
```

The `docker tag` command creates a new tag for an image. It does not create a new image. The tag points to the same image and is just another way to reference the image.

2.3 - Remove images

The `rmi` command stands for remove image.

```
$ docker rmi python-docker:v1.0.0
```

3.- Run the image

Now that we have an image, we can run that image and see if our application is running correctly.

A container is a normal operating system process except that this process is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host.

To run an image inside of a container, we use the `docker run` command. The `docker run` command requires one parameter which is the name of the image. Run the following command in your terminal:

```
$ docker run python-docker
```

3.1.- How to publish a port

To publish a port for our container, we'll use the `--publish` flag (`-p` for short) on the `docker run` command. The format of the `--publish` command is `[host port]:[container port]`. So, if we wanted to expose port 5000 inside the container to port 3000 outside the container, we would pass `3000:5000` to the `--publish` flag.

```
 docker run --publish 3000:5000 python-docker
```

3.2.- List containers

The `list containers` command is the following:

```
docker ps
```

3.3.- Stop, Start, Reestart and name containers

You can start, stop, and restart Docker containers. When we stop a container, it is not removed, but the status is changed to stopped and the process inside the container is stopped.

When we ran the `docker ps` command, the default output only shows running containers. When we pass the `--all` or `-a` for short, we see all containers on our machine.

```
$ docker ps -a
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS
ce02b3179f0f        python-docker         "python3 -m flask ru…"   16 minutes ago      Exited (0) 5 minutes ago
ec45285c456d        python-docker         "python3 -m flask ru…"   28 minutes ago      Exited (0) 20 minutes ago
fb7a41809e5d        python-docker         "python3 -m flask ru…"   37 minutes ago      Exited (0) 36 minutes ago
```

3.3.1.- Restarting containers

Let's restart the container that we just stopped. Locate the name of the container we just stopped and replace the name of the container below in the restart command.

```
$ docker restart ce02b3179f0f
```

Now list all the containers again using the `docker ps` command.

```
$ docker ps --all
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS
ce02b3179f0f        python-docker         "python3 -m flask ru…"   19 minutes ago      Up 8 seconds                  0.0.0.0
ec45285c456d        python-docker         "python3 -m flask ru…"   31 minutes ago      Exited (0) 23 minutes ago
fb7a41809e5d        python-docker         "python3 -m flask ru…"   40 minutes ago      Exited (0) 39 minutes ago
```

3.3.2.- Stopping and deleting containers

Now, let's stop and remove all of our containers:

```
docker stop ce02b3179f0f
```

Now that all of our containers are stopped, let's remove them. When you remove a container, it is no longer running, nor it is in the stopped status, but the process inside the container has been stopped and the metadata for the container has been removed.

```
$ docker ps -a
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS
ce02b3179f0f        python-docker         "python3 -m flask ru…"   16 minutes ago      Exited (0) 5 minutes ago
ec45285c456d        python-docker         "python3 -m flask ru…"   28 minutes ago      Exited (0) 20 minutes ago
fb7a41809e5d        python-docker         "python3 -m flask ru…"   37 minutes ago      Exited (0) 36 minutes ago
```

To remove a container, run the `docker rm` command with the container name or id. You can pass multiple container names/id's to the command using a single command:

```
docker rm ce02b3179f0f ec45285c456d fb7a41809e5d
```

4.- Run a database in a container (MySQL)

4.1.- Preparing the code

To use the database in our application it is necessary to update the code, adding two new endpoints:

```
import mysql.connector

```

```
@app.route('/widgets')
def get_widgets():
    mydb = mysql.connector.connect(
        host="mysqldb",
        user="root",
        password="p@ssw0rd1",
        database="inventory"
    )
    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()

    return json.dumps(json_data)

@app.route('/initdb')
def db_init():
    mydb = mysql.connector.connect(
        host="mysqldb",
        user="root",
        password="p@ssw0rd1"
    )
    cursor = mydb.cursor()

    cursor.execute("DROP DATABASE IF EXISTS inventory")
    cursor.execute("CREATE DATABASE inventory")
    cursor.close()

    mydb = mysql.connector.connect(
        host="mysqldb",
        user="root",
        password="p@ssw0rd1",
        database="inventory"
```

```
    )
    cursor = mydb.cursor()

    cursor.execute("DROP TABLE IF EXISTS widgets")
    cursor.execute("CREATE TABLE widgets (name VARCHAR(255), description
VARCHAR(255))")
    cursor.close()

    return 'init database'
```

- *In the folder 'Second example' we have all the code*

4.2.- Creating volume and network

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.

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
```

4.3.- Using Docker Compose

we 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.

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

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

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.

```
[]
```