Docker

Docker

Docker is a platform that allows you to develop, deploy and launch environments in containers. These environments are independent and operate on their own.

Container

The container is at the base of Docker, by launching an image, a container starts also storing this image.

Image

An image is a kind of executable that includes everything it needs to launch an application (code, library, configuration files, ...).

Container And virtual machine

In a sense, we can see container and virtual machines in much the same way, both can run applications, however size differences can be noted.

A container is launched natively on Linux and shares the kernel of the host machine with other containers taking no more space than another executable, making it light.

Unlike a virtual machine that launches a whole guest OS with virtual access, in general a VM allocates more necessary resources than necessary by the application, making them heavy.

Generality

In a simplified way, Docker makes it possible to make operational images available regardless of the configuration of the host machine.

These images run independently, for a Python image, it can launch a Python program, of course you must configure this image so that it runs the given program.

For example, for a machine that does not have Python installed by creating a Python image, it is possible to do so, because the image embeds everything necessary to run Python.

If you still don't understand, one of the strengths is a unified environment, in fact by creating an image, we decide on the configuration of it and therefore everyone works on the same version. On a Python 3 image, everyone will work on this version, even if a machine does not have it installed.

Installation

Docker engine: https://docs.docker.com/install/linux/docker-ce/ubuntu/

Docker compose: https://docs.docker.com/compose/install/

Docker Hub

Docker Hub is a service offered by Docker for finding and sharing images. Here is a small list of the services offered by Docker Hub:

- Directories: The push or pull of image container
- Organizations: Creation of private directories for access to containers
- Official images: Jumper of container images provided by Docker
- and many others ...

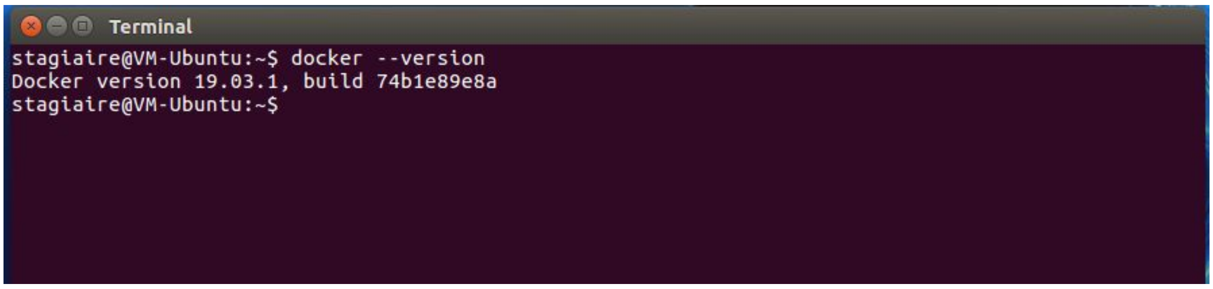
Here is the site link:https://hub.docker.com/

If you type in the hello-world search bar on the Docker Hub, you will find the image used during the **docker run hello-world** command located above.
There are a multitude of images such as mysql or ubuntu.

Test version Docker

Open the terminal and type this command to verify that docker is installed:

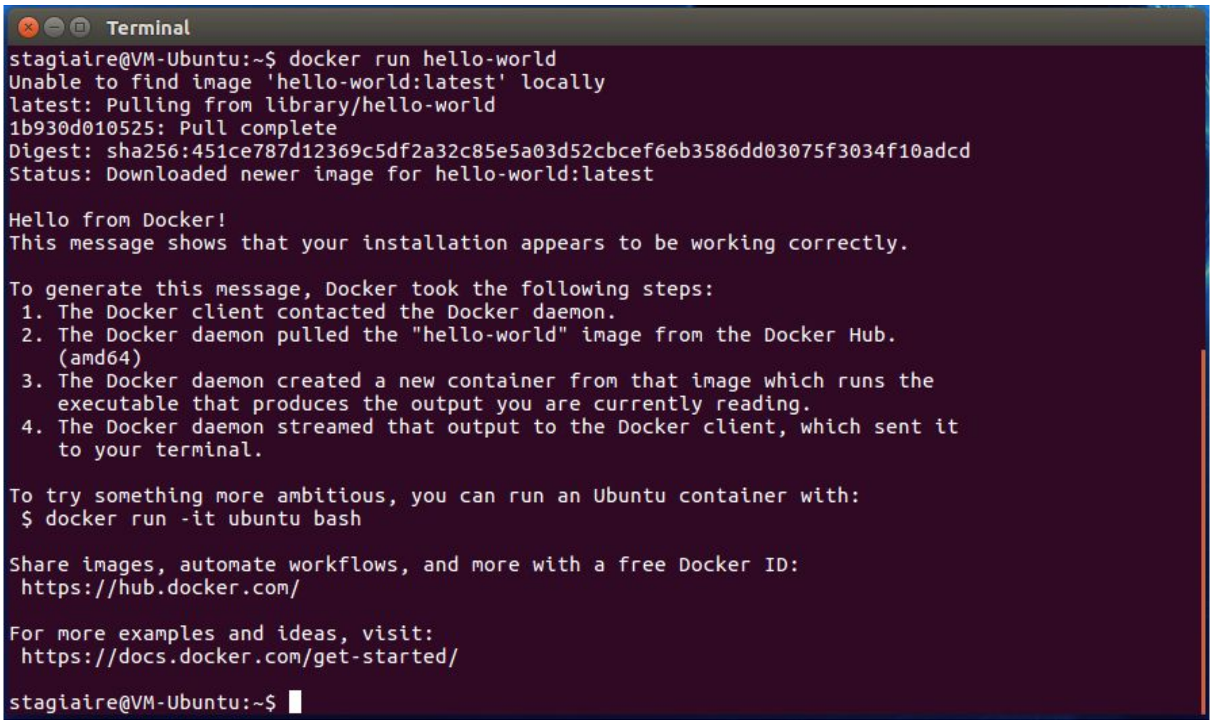
```
docker --version
```



```
stagiaire@VM-Ubuntu:~$ docker --version
Docker version 19.03.1, build 74b1e89e8a
stagiaire@VM-Ubuntu:~$
```

Then still in the terminal, type the command:

```
docker run hello-world
```

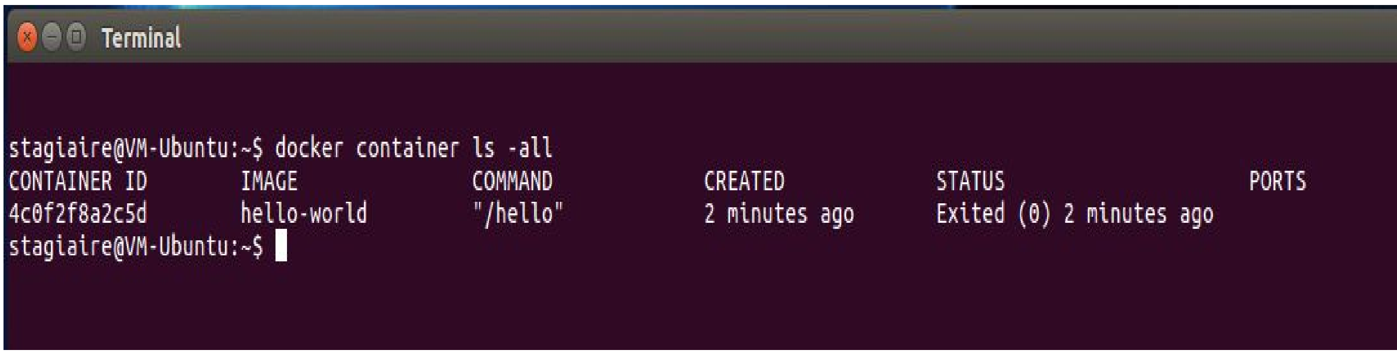


```
stagiaire@VM-Ubuntu:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:451ce787d12369c5df2a32c85e5a03d52cbcef6eb3586dd03075f3034f10adcd
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

stagiaire@VM-Ubuntu:~$
```

This command launches an image called hello-world, as we can see on the second line, docker does not find the image "hello-world" and will therefore pull this image on the docker hub. The steps used are described in points 1 to 4 on the terminal.

On the terminal, type the command:

```
docker container ls -all
```

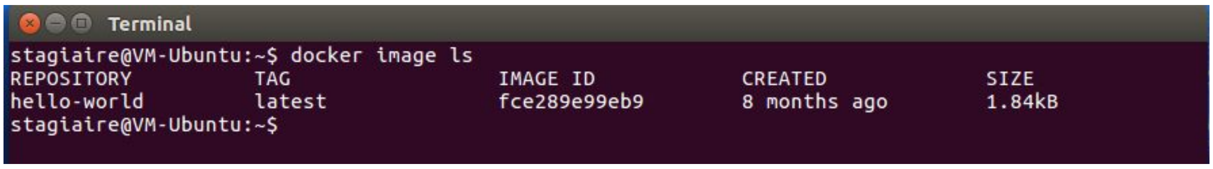


```
stagiaire@VM-Ubuntu:~$ docker container ls -all
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                   PORTS
4c0f2f8a2c5d        hello-world         "/hello"            2 minutes ago       Exited (0) 2 minutes ago
stagiaire@VM-Ubuntu:~$
```

This command makes it possible to list all the containers, we can see the ID of the container storing the image "hello-world".

This time type the command:

```
docker image ls
```



```
stagiaire@VM-Ubuntu:~$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              fce289e99eb9        8 months ago        1.84kB
stagiaire@VM-Ubuntu:~$
```

The command displays the rotating images, here we find our image hello-world 'image.

Delete containers and images

To delete a container:

```
sudo docker rm id_container
```

Replace id_container with ID

To delete all containers:

```
sudo docker rm $(docker ps -a -q)
```

The -a option allows you to have all the containers and -q their ID

To delete an image:

```
sudo docker rmi id_image
```

To delete all images:

```
sudo docker rmi $(docker images -q)
```

The -q option allows you to have the IDs of the images only.

Configure your Docker image

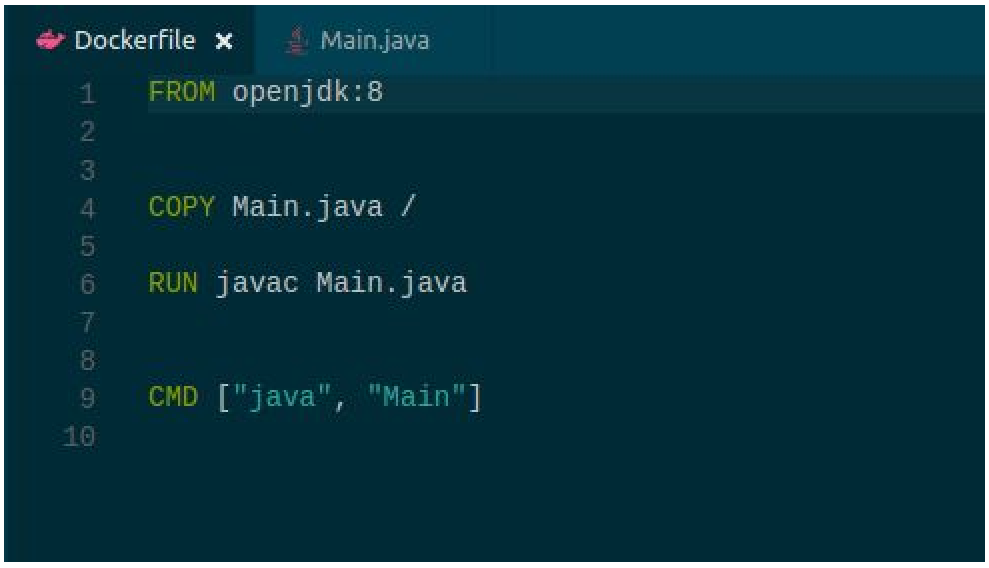
The configuration is an important step, we add for example the configuration files, an example is the file where the code is for a Java file to copy it, then compile it then execute it. All these configurations go through the Dockerfile.

Dockerfile

The dockerfile defines everything that will be in the container.

You will make a small example with a simple Java program.

To do this, create a folder and then inside it create an empty file called **Dockerfile**:



```
1   FROM openjdk:8
2
3
4   COPY Main.java /
5
6   RUN javac Main.java
7
8
9   CMD ["java", "Main"]
10
```

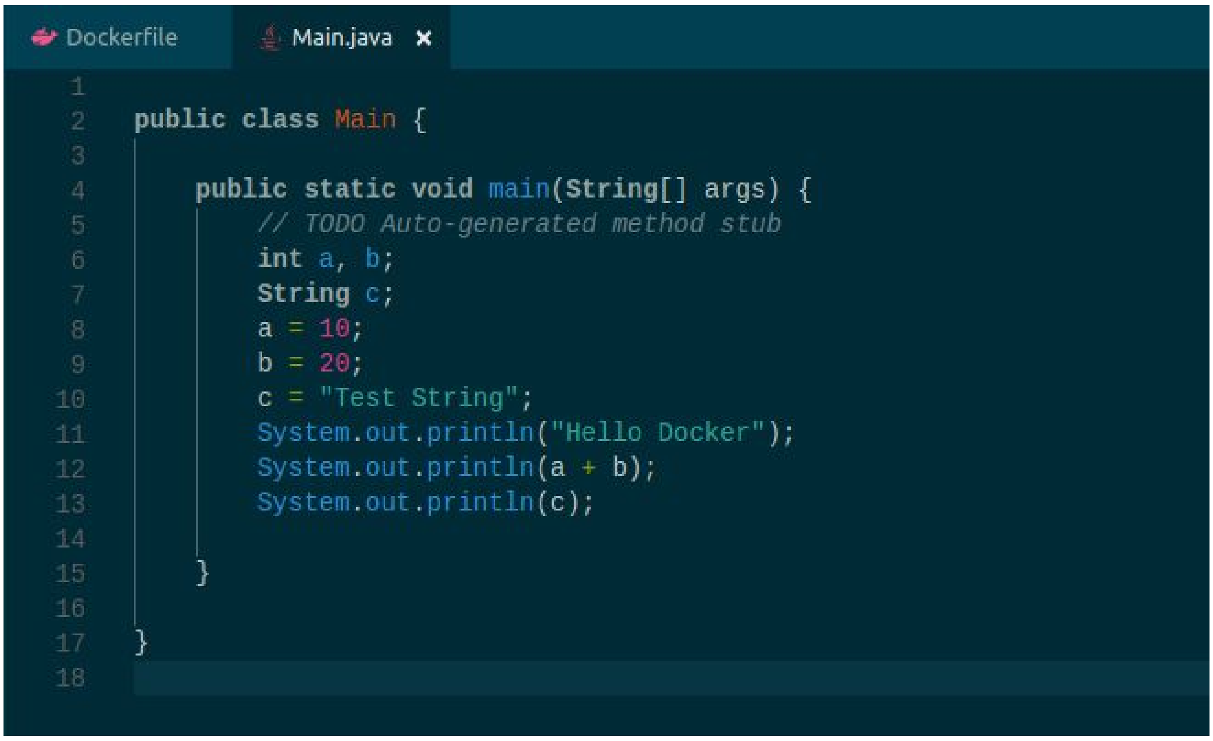
In line 1, ‘**FROM openjdk: 8**’, this indicates the desired image, this will allow us to rotate a Java image, for more information I invite you to go to the Docker Hub.

In line 4, ‘**COPY Main.java**’, this command allows you to copy a file called Main.java to the current directory of the image.

In line 6, ‘**RUN javac Main.java**’, the command allows to launch a command called ‘javac Main.java’ which is the command to compile a Java file (here Main.java).

In line 9, ‘**CMD [“ java ”,“ Main ”]**’, we execute the Main file when launching the container.

Now we need a **Main.java** file, in the same directory create this file in the same folder as the **Dockerfile**.


Dockerfile Main.java

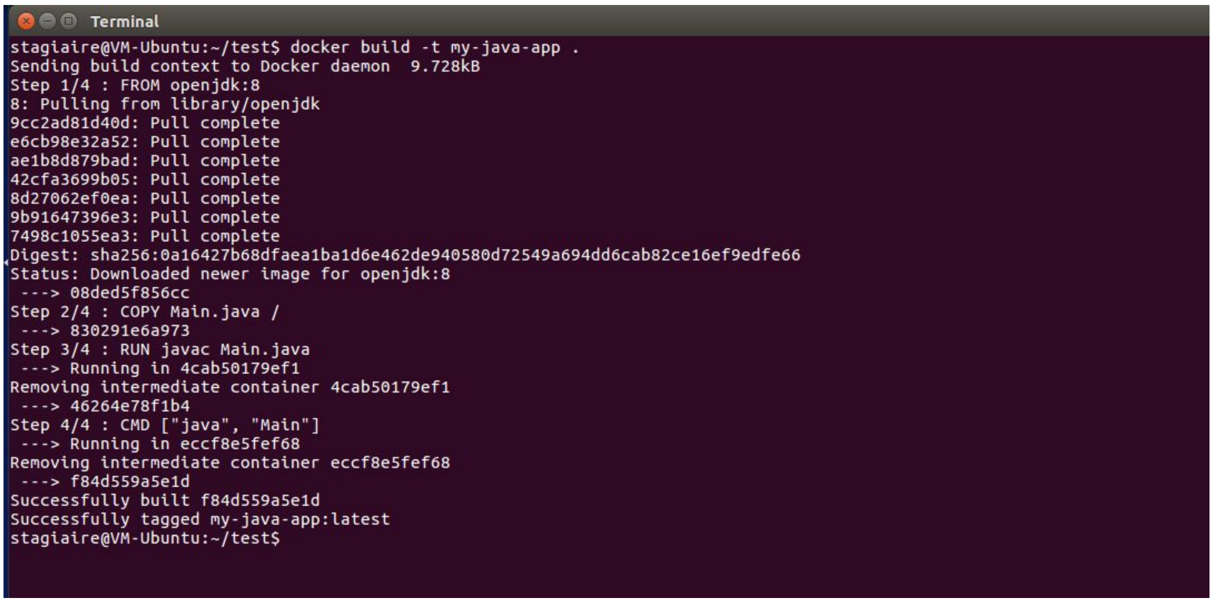
```java

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int a, b;
        String c;
        a = 10;
        b = 20;
        c = "Test String";
        System.out.println("Hello Docker");
        System.out.println(a + b);
        System.out.println(c);

    }

}

```

Once your Main.java file is complete, then on the terminal, go to the folder where your Dockerfile is located as well as the Main.java file.

Enter the command:

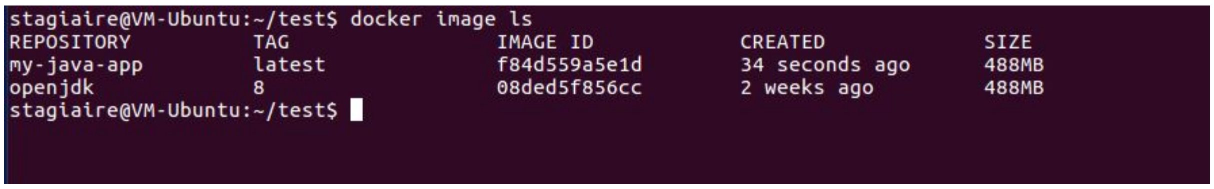
```
    docker build -t my-java-app .
```

The -t option allows to give a name to the image (here my-java-app).


Terminal

```
stagiaire@VM-Ubuntu:~/test$ docker build -t my-java-app .
Sending build context to Docker daemon  9.728kB
Step 1/4 : FROM openjdk:8
8: Pulling from library/openjdk
9cc2ad81d40d: Pull complete
e6cb98e32a52: Pull complete
ae1b8d879bad: Pull complete
42cfa3699b05: Pull complete
8d27062ef0ea: Pull complete
9b91647396e3: Pull complete
7498c1055ea3: Pull complete
Digest: sha256:0a16427b68dfaea1ba1d6e462de940580d72549a694dd6cab82ce16ef9edfe66
Status: Downloaded newer image for openjdk:8
 ---> 08ded5f856cc
Step 2/4 : COPY Main.java /
 ---> 830291e6a973
Step 3/4 : RUN javac Main.java
 ---> Running in 4cab50179ef1
Removing intermediate container 4cab50179ef1
 ---> 46264e78f1b4
Step 4/4 : CMD ["java", "Main"]
 ---> Running in eccf8e5fef68
Removing intermediate container eccf8e5fef68
 ---> f84d559a5e1d
Successfully built f84d559a5e1d
Successfully tagged my-java-app:latest
stagiaire@VM-Ubuntu:~/test$
```

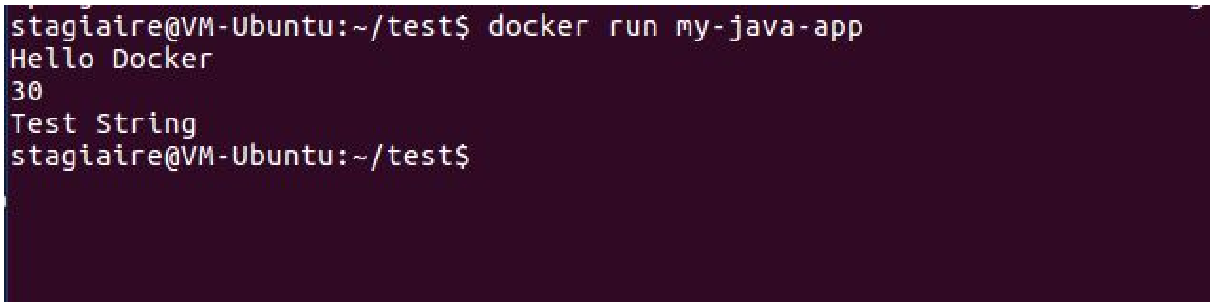
By looking in the terminal with the docker image ls command, we can find our ‘my-java-app’.



```
stagiaire@VM-Ubuntu:~/test$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my-java-app         latest              f84d559a5e1d        34 seconds ago      488MB
openjdk             8                   08ded5f856cc        2 weeks ago         488MB
stagiaire@VM-Ubuntu:~/test$ 
```

Launch the container with the command:

```
docker run my-java-app
```



```
stagiaire@VM-Ubuntu:~/test$ docker run my-java-app
Hello Docker
30
Test String
stagiaire@VM-Ubuntu:~/test$
```

As you can see, on the terminal, the display instructions found in the Main.java file are displayed.

Docker Compose

Docker Compose is a tool that allows you to define and launch several Docker containers. For this we use a YAML file to configure the services of the application. Then with a simple command, you can create and launch all the services from your configuration.

Using Docker Compose is accomplished in three steps:
- Definition of the application environment with a Dockerfile
- Definition of services with docker-compose.yml, this will allow them to run together in an isolated environment
- Launching docker-compose up in the terminal, Docker Compose starts and launches the whole application

For this example, we are going to launch a docker-compose for the fullstack project that you carried out in training:

Create a directory to test this, in this directory you will create an empty file called **docker-compose.yml** then add this content:

```
version: '3'
services:
  nginx:
   container_name: nginx
   image: nginx:1.13
   restart: always
   ports:
   - 80:80
   - 443:443
   volumes:
   - ./nginx/conf.d:/etc/nginx/conf.d

  angular:
    container_name: angular
    build: ./angular
    restart: always
    ports:
      - 4200:4200
    depends_on:
      - app

  mysql:
   container_name: mysql
   image: mysql/mysql-server:5.7
   environment:
    MYSQL_DATABASE: test
    MYSQL_ROOT_PASSWORD: Formation123
    MYSQL_ROOT_HOST: '%'
   ports:
   - "3306:3306"
   restart: always

  springboot:
    restart: always
    build:
      context: ./springboot
      dockerfile: Dockerfile-app
    working_dir: /springboot
    volumes:
      - ./springboot:/springboot
      - ~/.m2:/root/.m2
    expose:
      - "8080"
    depends_on:
      - nginx
      - mysql
    command: mvn clean spring-boot:run -Dspring-boot.run.profiles=docker
```

Line 1: the version, some images or arguments only work from a certain version, it is recommended to always indicate this.

In line 2: services, this is where we will indicate all the services that the docker-compose will run.

Then we indicate the name of the service, here we have 4 services for:

- nginx
- mysql
- springboot
- angular

Here are some details on the content of the docker-compose:
Container-name: this is where we will name our container
Image: image to use for the container
Ports: the port used by the container
Volumes: allows you to create folders, to preserve persistent data, such as configuration files
Build: allows you to create an image from a docker file
Depends_on: allows to link the containers together
Environment: allows to enter values, here used for mysql with the database and the password

the docker-file for springboot only contains: **FROM maven: 3.5-jdk-8**, we indicate the version of maven that we want to use.

The docker-file of angular is a little more complex:

```
# base image
FROM node:10.15.0

# set working directory
RUN mkdir /usr/src/app
WORKDIR /usr/src/app

# add `/usr/src/app/node_modules/.bin` to $PATH
ENV PATH /usr/src/app/node_modules/.bin:$PATH

# install and cache app dependencies
COPY package.json /usr/src/app/package.json
RUN npm install
```

```
# add app
COPY . /usr/src/app

EXPOSE 4200

# start app
CMD ng serve --host 0.0.0.0 --port 4200
```

With the comments you should understand. It is however important to pay attention to the angular version you are using, otherwise you will have to modify the node version and the json files such as package.json. You can delete the node-module folder from your project because the docker-file will create it in your docker.

If everything is good in your project folder you have 3 folders, angular, springboot and nginx, plus the docker-compose file.

To launch your docker go to your project folder with the terminal. Start the command:
docker-compose up
This command builds the images if they do not exist and launches the containers

If everything is good. then go to localhost: 4200 to see the result with angular. You should be able to add and view users.

Some useful docker-compose commands:
docker-compose build: to create the images, do not launch the containers
docker-compose down: to stop the containers, also delete the containers, networks, volumes and images made from the up.