Docker Container Exercise

1 Introduction

This exercise aims to become familiar with Docker containers and Docker compose. You are going to create a website (server) in Python that will contain a sentence. This sentence must be retrieved by a program (client) in Python that will display the sentence. You should have two containers , the first one will run the server and the second one will run the client. Then you will run Docker-Compose to launch the two containers at once. Before starting, it is import to understand:

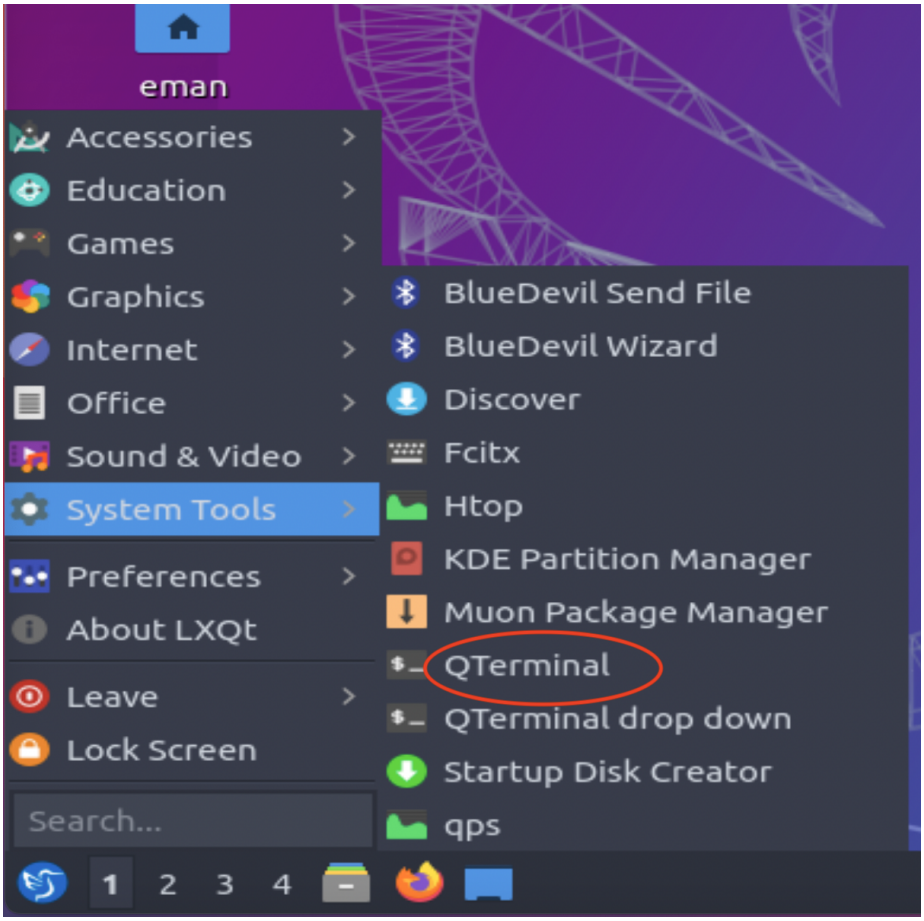
- What is Docker? Docker is a tool designed to make it easier to create, deploy, and run applications using containers.
- Why do developers need Containers? Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and deploy it as one package.
- What is the difference between Docker and Docker compose? Image that you want to develop and run application and this application has one container/service then **Docker** can be used to manage this container/service. While, if your application has more than one container/service then you need **Docker-Compose** to manage these containers/services.

2 Setup environment

To start working, you have to set up your environment as follows:

1. Create a new VM using Lubuntu 20.04.
2. Install Docker On your VM.

- Open Terminal:



- Manage Docker as a non-root user:

  The Docker daemon binds to a Unix socket instead of a TCP port. By default that Unix socket is owned by the user root and other users can only access it using sudo. The Docker daemon always runs as the root user. Instead of preface the docker command with sudo, create a Unix group called docker and add users to it.

  To create the docker group and add your user:

  (a) Create the docker group.

```
eman@eman-virtualbox:~$ sudo groupadd docker
[sudo] password for eman:
```

(b) Add your user to the docker group:

```
eman@eman-virtualbox:~$ sudo usermod -aG docker eman
eman@eman-virtualbox:~$
```

(c) Restart the virtual machine for changes to take effect, you can also run the following command to activate the changes to groups.

```
eman@eman-virtualbox:~$ newgrp docker
eman@eman-virtualbox:~$
```

- After creating docker group update your packages:

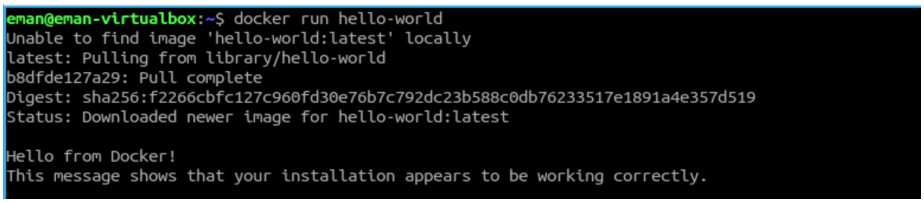
```
eman@eman-virtualbox:~$ sudo apt update
[sudo] password for eman:
Hit:1 http://no.archive.ubuntu.com/ubuntu focal InRelease
Hit:2 http://no.archive.ubuntu.com/ubuntu focal-updates InRelease
Get:3 http://security.ubuntu.com/ubuntu focal-security InRelease [109 kB]
Hit:4 http://no.archive.ubuntu.com/ubuntu focal-backports InRelease
Fetched 109 kB in 1s (176 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
All packages are up to date.
```

- Install Docker:

```
eman@eman-virtualbox:~$ sudo apt install docker.io
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  bridge-utils containerd pigz runc ubuntu-fan
Suggested packages:
  ifupdown aufs-tools cgroupfs-mount | cgroup-lite debootstrap docker-doc
  rinse zfs-fuse | zfsutils
The following NEW packages will be installed:
  bridge-utils containerd docker.io pigz runc ubuntu-fan
0 upgraded, 6 newly installed, 0 to remove and 0 not upgraded.
Need to get 68,9 MB of archives.
After this operation, 339 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://no.archive.ubuntu.com/ubuntu focal/universe amd64 pigz amd64 2.4-1 [57,4 kB]
Get:2 http://no.archive.ubuntu.com/ubuntu focal/main amd64 bridge-utils amd64 1.6-2ubuntu1 [30,5
 kB]
Get:3 http://no.archive.ubuntu.com/ubuntu focal-updates/main amd64 runc amd64 1.0.0~rc93-0ubuntu
1~20.04.1 [4.012 kB]
```

- To verify that you can run docker commands without "sudo" and

verify Docker installation:



After install Docker successfully, you can run set of Docker commands to know and work with the images and containers. In the verifying step, We installed docker image from "Docker Hub", "hello-world", and run this image. Once you run the image, Docker automatically generate a container from this image and run this container. Based on that you can do the following commands:

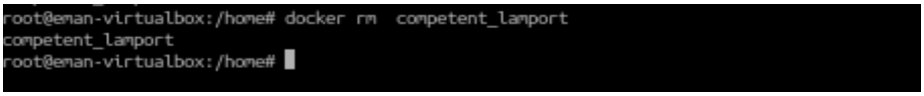
(a) To list all running Docker containers use the command: docker ps

(b) To list all containers, both running and stopped, add –a : docker ps -a

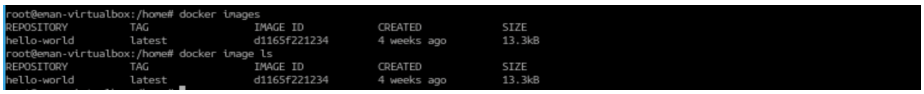
(c) To list containers by their ID use –aq (quiet): docker ps -aq

(d) To Stop a specific container use docker stop [container name] : docker stop type-container-name

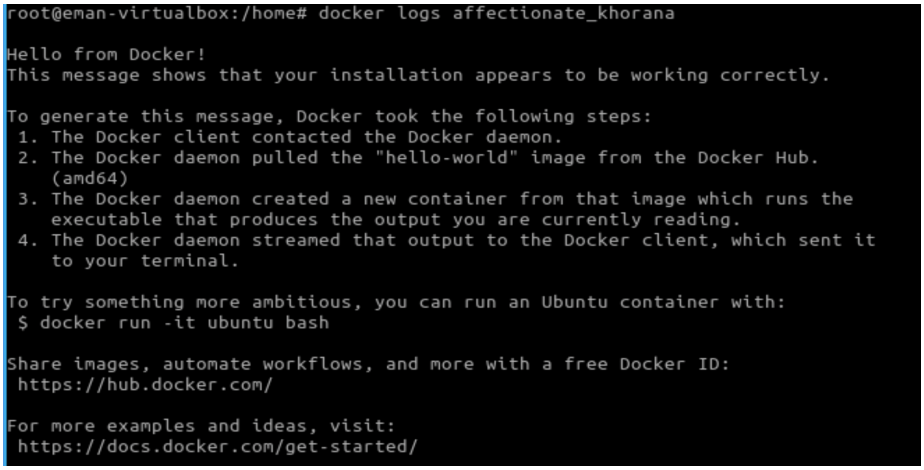
(e) To Remove a specific container use docker rm [container name] : docker rm type-container-name



(f) To list all Docker images docker: docker image OR docker image ls



(g) To show information logged by a running container: docker logs type-container-name



3 Create Project Structure

In this step, you will create two folder (client and server) and one YAMl file (docker-compose.yml).
Note: YAML is commonly used for configuration files and in applications where data is being stored or transmitted. YAML is also used for swagger / OpenAPI.
The current directory now is /home then start to run the following commands to create the project structure:

1. Create main directory of the project and change current directory to "dockerProject": $ mkdir dockerProject
2. Change directory to "dockerProject" directory: $ cd dockerProject
3. Create Docker-compose yaml file using the following command: $ touch docker-compose.yml
4. Create a sub-directory for server: $ mkdir server-side
5. Create a sub-directory for client: $ mkdir client-side
6. Your project structure should be as following:

```
eman@eman-virtualbox:~/dockerProject$ tree
.
├── client-side
├── docker-compose.yml
└── server-side

2 directories, 1 file
```

7. To able to write and run different files, you need to change all permissions type of the files in the current directory:

```
eman@eman-virtualbox:~/dockerProject$ chmod -R 777 ./
eman@eman-virtualbox:~/dockerProject$ ls -l
total 8
drwxrwxrwx 2 eman docker 4096 Apr 22 00:46 client-side
-rwxrwxrwx 1 eman docker    0 Apr 22 00:42 docker-compose.yml
drwxrwxrwx 2 eman docker 4096 Apr 22 00:46 server-side
eman@eman-virtualbox:~/dockerProject$
```

4 Implementing a basic server

Change the current directory to server-side directory: $ cd server-side
Create a python file to be our server-app: $ nano server.py
Let's make a very basic python server you can wrap inside a docker container to run anywhere:

```
1 import socket
2 print("Python server starting...")
3 local_port = 4242
4 sock = socket.socket()
5 sock.bind(("", local_port))
6 sock.listen()
7 print("Python server running on port {}".format(local_port))
```

```
while True:
    (conn, addr) = sock.accept()
    print("New connection from {}".format(addr))
    conn.send("Hello from python inside docker!\n".encode())
    conn.close()
```

Listing 1: Server Python code

Tasks:

1. Implement your own server - or copy this one.
2. Start your server and see that you can connect to it either from your browser or using netcat (e.g. nc localhost 4242 or http://localhost:4242 in browser).

Starting the server as a container:

Create docker file: it will contain the necessary instructions to create the environment of the web-server. Run the following command: $ nano Dockerfile

```
FROM python:latest

# In order to launch our python code, we must import the 'server.py'
# We use the keyword 'ADD' to do that.
# Just a remember, the first parameter 'server.py' is the name of the file on the host.
# The second parameter '/server-side/' is the path where to put the file on the image.
# Here we put files at the image '/server-side/' folder.

ADD server.py /server-side/

# 'WORKDIR' command changes the base directory of your image.
# Here we define '/server-side/' as base directory (where all commands will be executed).

WORKDIR /server-side/
```

Save and Close Dockerfile suing CTRL + X.

From inside /dockerProject/server-side you can now create an image containing your server:

```
\$ docker build -t my_server .
```

This will build a container image named my_server. You can now start a container running this image (you can start many containers using the same image, like instantiating objects from classes):

```
\$ docker run -p 8080:4242 --name server1 --rm -t my_server python3
./server.py
```

Let's go through the options we just passed in:

1. -p exposes port 4242 inside the container, as port 8080 in the host (which in this case is the virtualbox vm running lubuntu).

2. –name specifies a name of your container - if you don't specify one you'll get one, such as backstabbing_mayer or nostalgic_keller.

3. –rm will remove the container after it stops - but it does not remove the image! This is a lifesaver that can help you avoid a lot of garbage containers that are already stopped.

4. -t for "tty" tells docker to pass the command output to the standard out of the host and pass in signals. This is helpful for debugging as it makes the output of your docker command behave as if you started the program locally.

You should now see something like this:

```
\$ docker run -p 8080:4242 --name server1 --rm -t my_server python3
./server.py
python3 ./server.py
Python server starting...
Python server running on port 4242
```

In another terminal you can now connect to the mapped port, 8080in this case, which will pass the incoming traffic onto the container and then in to the python server running on port 4242 inside. You can use a web browser, or a terminal program like HTTPie or curl. Try it a couple of times and you should see something like this on the client OR open your browser and type: "localhost:8080":

```
\$ curl localhost:8080
Hello from python inside docker!
\$ curl localhost:8080
Hello from python inside docker!
```

Tasks:

Reproduce the above with your own server

5 Making a client inside another container

Let your docker server run and make a fresh terminal window and go to your client directory, dockerProject/client-side.

When we started the server we added -p 8080:80 to specify that the service should be available in port 8080 on our host. But how do we reach it from inside another container?

- In -p 8080:80, the first port number is the host port and the second one is the port inside the container, so we should be able to reach the service from port 80.
- But which IP address do we connect to? Each container behaves as its own machine, so localhost would mean "this container", which is not what we want. We want to connect to the container named server1.

Finding the local IP of a docker container:

- To find the IP of your running container by inspecting your docker network, run the following: $ docker network inspect bridge.
- You should get the following output:

```
[
    {
        "Name": "bridge",
        "Id":
"e2160c5f0670a0436d899e148935c0300d7b9521cb26b265b2078fe70cc823be",
        "Created": "2021-04-21T13:08:05.478701721+02:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {

"f2ecadda7dff90e53a14570026d0a1fdc7f2413146d66e5daa7292249c2e35c4": {
                "Name": "server1",
                "EndpointID":
"97d3966992a76226b46e086746c963f481cf7f48336eb5f7b594599ba0562c29",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```


- We don't need most of this information, but notice this part:

```
"Config": [
                {
                    "Subnet": "172.17.0.0/16"
                }
            ]
```

Which tells us that the default docker network bridge (named bridge from the perspective of $ docker network ls) is using the 172.17.0.0/16 subnet. This means that all your containers will have IP address from this range.

The more important thing to notice is that inside the "Containers" object you'll find your server: "Name": "server1"- which in this case has "IPv4Address": "172.17.0.2/16". So my server should be reachable from other containers on 172.17.0.2 . Tasks:

- Find the IP of your running server using $ docker network inspect bridge

Containerizing netcat

Before we make our own client let's try to use one we already know - netcat.

Installing netcat inside a container

Now that you've found your container's internal IP, you can reach it from another container using e.g. netcat. By default, the ubuntu image doesn't come with netcat, but we can install that and run it in a throw-away container like so:

```
\$ docker run -ti --rm ubuntu bash -c "apt update; apt install -y
    netcat; nc 172.17.0.2 4242"
```

Here we're specifying that we want to make a container based on ubuntu, run bash and then these 3 commands inside of bash:

- apt update - update the list of packages inside your ubuntu container
- apt install -y netcat - install netcat (will be availalbe as nc)
- nc 172.17.0.2 4242 - run netcat, connecting to your server container's IP on port 4242 (the local container port - not the host port)

The output should be something like this:

```
\$ docker run -ti --rm ubuntu bash -c "apt update; apt install -y
    netcat;nc 172.17.0.2 4242"
Get:1 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
Get:2 http://security.ubuntu.com/ubuntu focal-security InRelease
    [109 kB]
Get:3 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114
     kB]
```

```
Get:4 http://archive.ubuntu.com/ubuntu focal-backports InRelease
    [101 kB]
Get:5 http://archive.ubuntu.com/ubuntu focal/main amd64 Packages
    [1275 kB]
...
Setting up netcat (1.206-1ubuntu1) ...
Processing triggers for libc-bin (2.31-0ubuntu9.2) ...
Hello from python inside docker!
```

A netcat dockerfile

The above is fine for a one-off, but if we want to do this over and over it would be nice to have a docker file instead. Something like this could work:

```
FROM ubuntu:latest
RUN apt update
RUN apt install -y netcat
```

Adding that to dockerProject/netcat/Dockerfile we can now build the docker image:

```
\$ docker build -t netcat1 .
```

...and verify that it got created:

```
$ docker images
REPOSITORY          TAG       IMAGE ID       CREATED             SIZE
netcat1             latest    99503a369f9a   4 minutes ago       102MB
...
```

We can now run netcat from inside a throw-away container:

```
$ docker run -ti --rm netcat1 nc 172.17.0.2 4242
Hello from python inside docker!
```

(Note that this command will hang - that's just netcat waiting for input from you! Typing anything and then enter should make it stop)

A nicer version - docker as a command wrapper

The advantage of the dockerfile is that you can now run the docker run command above over and over and since the image is already built you don't have to download or install anything every time. The container is created every time, but that's impressively fast. So fast that we could use this as a wrapper to give us a version of netcat that behaves exactly the same across linux, Windows and Mac.

If all we use this image for is to run netcat, why not have that as the default command?
Adding an ENTRYPOINT or CMD at the end lets us specify which command to run by default whenever the container starts.

```
FROM ubuntu:latest
RUN apt update
RUN apt install -y netcat
ENTRYPOINT ["nc"]
```

To give our image a short and meaningful name we can rebuild this one with:

```
\$ docker build -t nc .
```

We now have a docker image simply named nc, like its default command. Now make sure your python server is started, and we can can reach it from our netcat container like this:

```
\$ docker run -ti --rm netcat1 nc 172.17.0.2 4242
Hello from python inside docker!
```

Getting DNS for free with custom networks

The above is nice, but it's not fun having to look up the IP address of our docker containers manually all the time. If we make a custom network however, Docker will provide dns names for our containers for us.
List your existing docker networks using docker network ls:

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
e2160c5f0670   bridge    bridge    local
f1b25a897eed   host      host      local
bddb71c2f04e   none      null      local
```

Create a new docker network called skynet:

```
$ docker network create skynet
8a07ba497d0cef5408bcb449a72511a56de8056dcb3c227884296d6d259ad88f
```

Verify that it's there:

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
e2160c5f0670   bridge    bridge    local
f1b25a897eed   host      host      local
bddb71c2f04e   none      null      local
8a07ba497d0c   skynet    bridge    local
```

You can now add any docker container to skynet by adding –network skynet to the docker run command. Stop your python server and restart it using this command:

```
\$ docker run -p 8080:4242 --name server1 --network skynet --rm --
    hostname python_server -t my_server python3 ./server.py
```

You should see output like:

```
Python server starting...
Python server running on port 4242
```

Run the command "$ docker network inspect skynet", you should now see that your server came up in the new network:

```
[
    {
        "Name": "skynet",
        "Id":
"8a07ba497d0cef5408bcb449a72511a56de8056dcb3c227884296d6d259ad88f",
        "Created": "2021-04-21T17:11:07.865016896+02:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {

"a8fd12f3b46e2a1331ce3bc99c2e025774f89e180aeaa7f857c9ef9a801ee89a": {
                "Name": "server1",
                "EndpointID":
"b46df3648882d5cdd8d0456ccbf9c16d1285d5eb4bf570173112d927a13b0758",
                "MacAddress": "02:42:ac:13:00:02",
                "IPv4Address": "172.19.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```


Notice how this is a completely different IP range from before. You can connect to this service using that new IP address and netcat, if you add your netcat to the skynet network as well:

```
\$ docker run -ti --rm --network skynet nc 172.19.0.2 4242
Hello from python inside docker!
```

But a nice advantage is that you can now also connect by hostname, which we

specified to be python_server when we started the server:

```
\$ docker run -ti --rm --network skynet nc python_server 4242
Hello from python inside docker!
```

Making a python client It should now be straightforward to make our own python client. In dockerProject/client-side let's create some python client code, using command:

```
\$ nano client.py
```

```
import socket

for i in range(0,10):
    sock = socket.socket()
    sock.connect(("python_server", 4242))
    res = sock.recv(256)
    print("Response: {}".format(res.decode()))
```

This will simply connect to our server 10 times and print the response. Now create a minimal Dockerfile:

```
\$ nano Dockerfile
```

```
FROM python:latest
ADD client.py . ./
```

Build an image:

```
$ docker build -t python_client .
```

Start a container based on that image, calling the client from inside:

```
$ docker run -ti --rm --network skynet python_client python3 ./client.py
Response: Hello from python inside docker!

Response: Hello from python inside docker!

Response: Hello from python inside docker!

Response: Hello from python inside docker!

Response: Hello from python inside docker!

Response: Hello from python inside docker!

Response: Hello from python inside docker!

Response: Hello from python inside docker!

Response: Hello from python inside docker!

Response: Hello from python inside docker!
```

NOTE: It would be natural to add the python3 ./client.py part to the dockerfile as CMD ["python3", "./client.py"] - this way you wouldn't have to specify the command on the command line whenever you start your container. Feel free to try that! For this exercise however, we're going to add that in the dockercompose.yaml, which will start both the client and server for us at once.

6 Start Docker Compose

You have created two different applications, the server, and the client, both with a Dockerfile. Now you are going to edit the 'docker-compose.yml' at the root of the project.

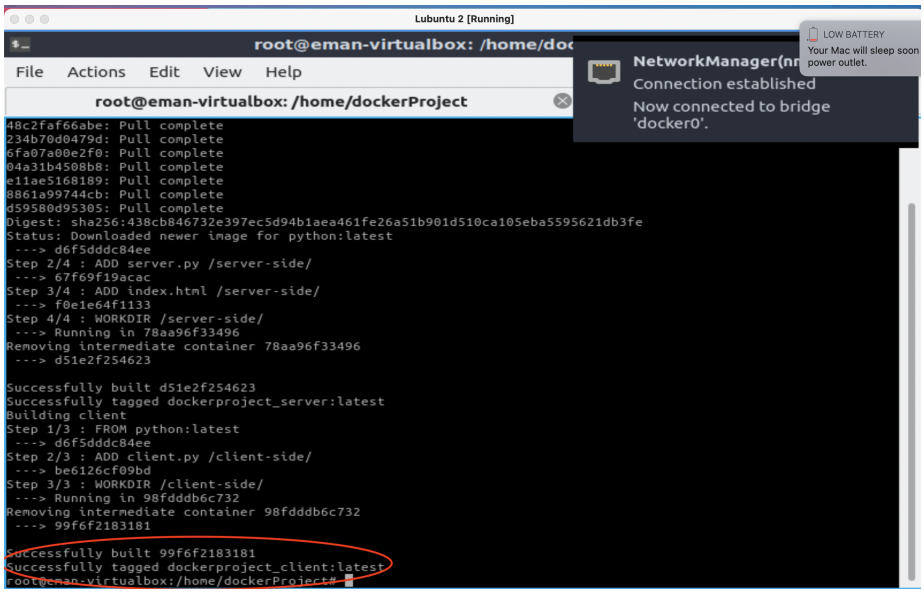
- Check current working directory by using the command "pwd". If the current working directory not the root of the project then use the command "cd .." until you reach the root "/dockerProject".
- Edit docker-compose.yml file: run the command "nano docker-compose.yml" then write following commands:

```
1    # A docker-compose must always start by the version tag.
2    version: "3.3"
3    services:
4     server:
```

```
5         build: server-side/
6         command: python ./server.py
7         ports:
8          - 8080:4242
9       client:
10        build: client-side/
11        command: python ./client.py
12        depends_on:
13         - server
14      networks:
15        default:
16         external:
17          name: skynet
18
```

Here we used the skynet network that we creates before.

- Run the following command to make sure that no error within YMAL file: $ docker-compose config
- Install Docker-compose: run the command "apt install docker-compose"
- Build Docker-Compose: now it is time to build up the whole project "dockerProject" by using command "docker-compose build".



- Run Docker-Compose: The docker-compose is built! Now it's time to start by running the command "docker-compose up".
- Finally open the browser in your VM and type the 'http://localhost:8080/'.