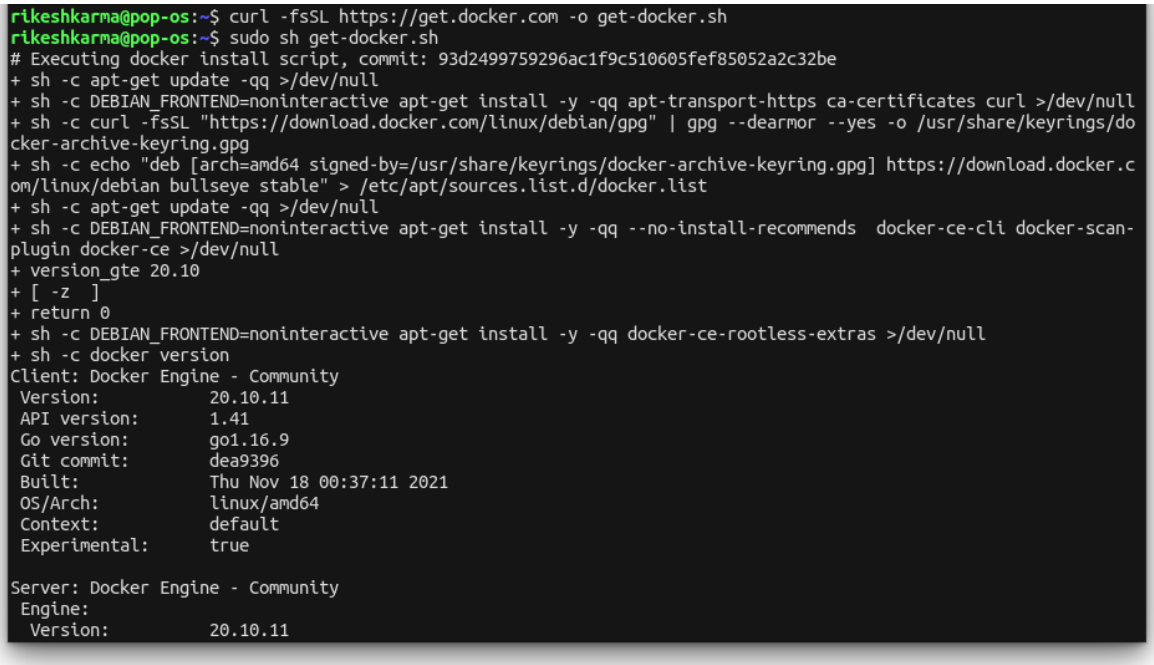
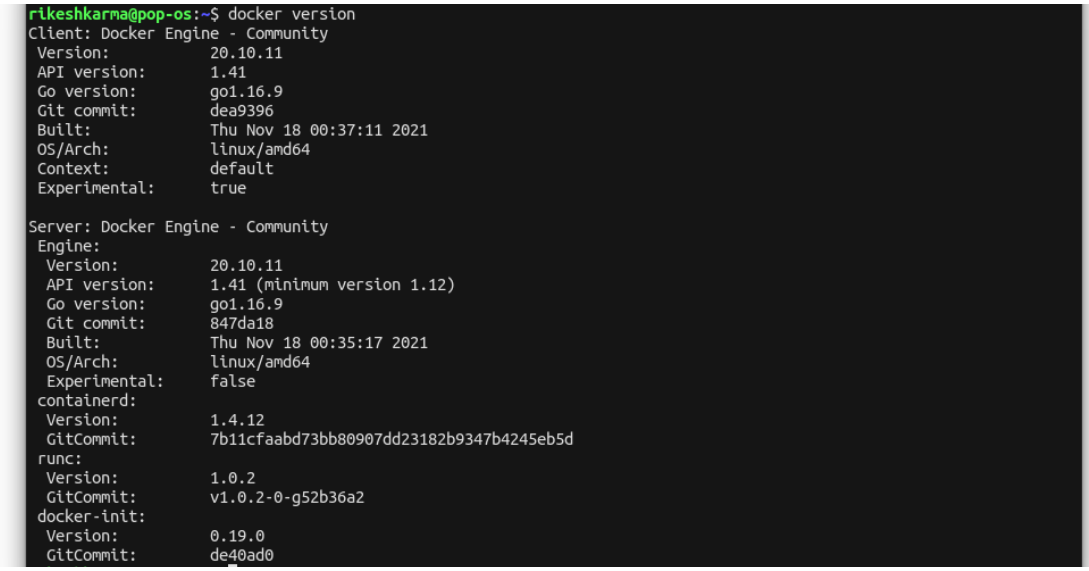
1. Install docker and then create a container (Nginx) with docker CLI (volume, network, port).

- ➔ Docker is an open-source containerization platform. It enables developers to package applications into containers—standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment. Containers simplify the delivery of distributed applications and have become increasingly popular as organizations shift to cloud-native development and hybrid multi-cloudenvironments.
  Developers can create containers without Docker, but the platform makes it easier, simpler, and safer to build, deploy and manage containers. Docker is essentially a toolkit that enables developers to build, deploy, run, update, and stop containers using simple commands and work-saving automation through a single API.

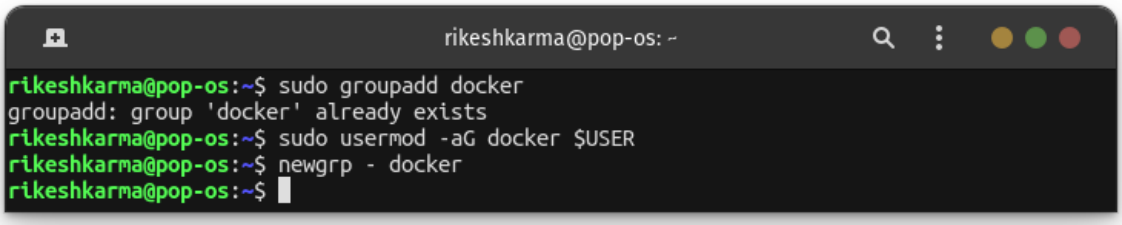
- ➔ To install docker we will be using the convenience script.
  - ◆ Let's invoke the following commands one after another:
    - *curl -fsSL https://get.docker.com -o get-docker.sh*
    - *DRY_RUN=1 sh ./get-docker.sh*
      - ○ 'DRY_RUN=1' is optional, we can use it if we want to learn what steps the script will execute during installation.



- Now we can very our installation by using the following command:
  - *docker version*



  - After we verify docker installation, let's add a docker group, add our current user to the docker group and switch the session to the docker group running the following commands:
    - *sudo groupadd docker*
    - *sudo usermod -aG docker $USER*
    - *newgrp - docker*



➔ Now to create an Nginx container we need to run the command below in the terminal:

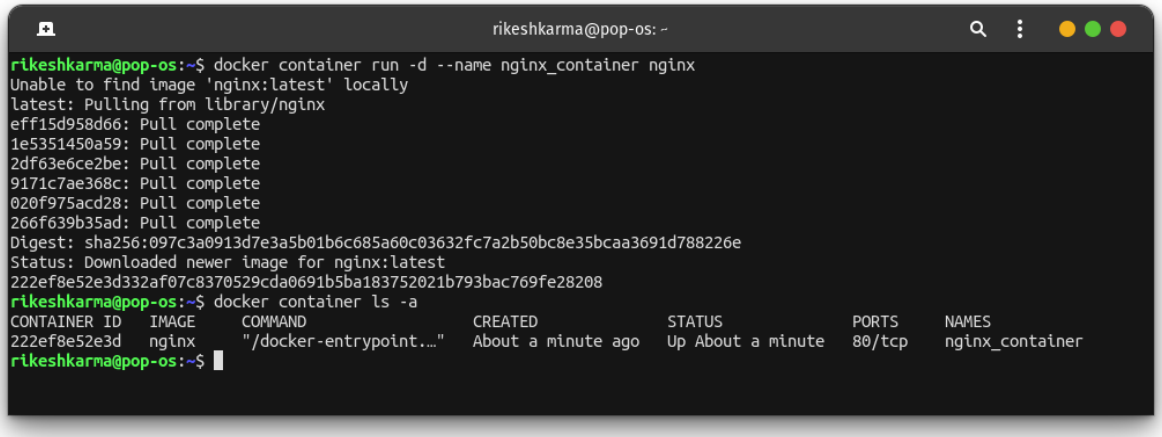
- *docker container run -d --name nginx_container -v /home/rikeshkarma/docker/index:usr/share/nginx/html --network bridge -p 123:80 nginx*
  - ***-d:-*** *This option is used to run the container in the detached mode.*
  - ***--name:-*** *This option is used to provide the name to the container, in our case, nginx_container* is the name we have provided to our

container. If we don't use this option, the docker will assign the random alphanumeric string to the container.

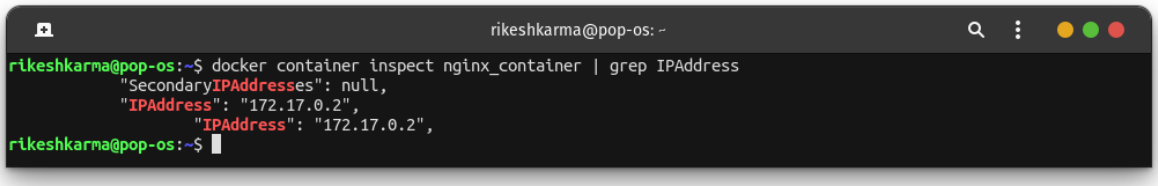
- **-v:-** This option is used to mount the volumn from the host to the container.
- **--network:-** This option will help to choose a network or we can create a new network too.
- **-p:-** This option will help to assign port from host to container (host:container)
- ***Nginx:-*** This is the name of the Nginx image.

◆ We can list the containers and see the Nginx should be up and running. We can run the following command to see the list of containers:

- *docker container ls -a*



- We can see that the Nginx server is running with default port **80** which is actually configured in the Nginx image Dockerfile. If we want to access the Nginx server, we must know the IP address of the Nginx server too. And in docker, each container has its host and IP.
- We can run the following command to display the properties of the container and file out the IP address.
  - *docker container inspects nginx_container | grep IPAddress*
    - **inspect:-** This command is used to display information on one or more containers.
    - **grep IPAddress:-** This is not the docker command, however, this is Linux specific command to find out any word from the command's output.



- From the above command and the snapshot, we can see the result. And we can observe that the container's IP is **172.17.0.2**.

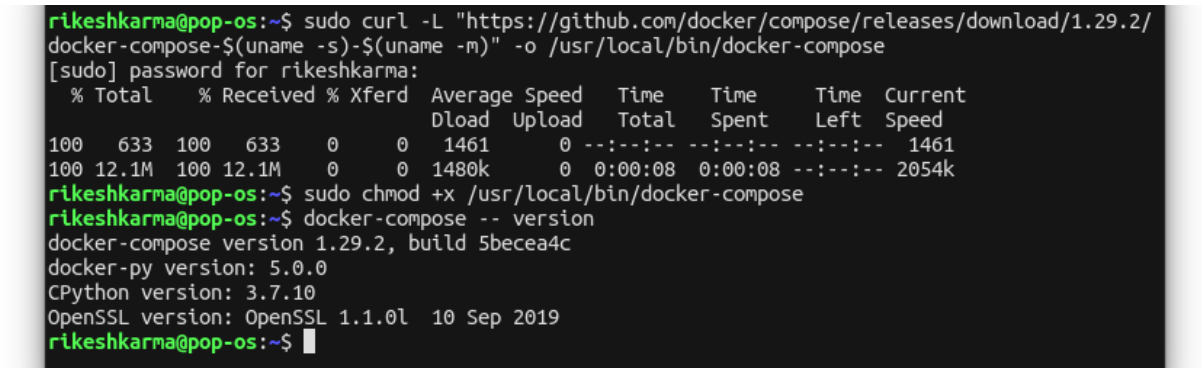
2. Create a docker container with a docker-compose file for mysql8 and also volume mount and port.

➔ Firstly let's get Docker Compose on our system by running following commands sequentially:

- *sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose*
- *sudo chmod +x /usr/local/bin/docker-compose*

➔ We can test if our installation was setup correctly by running:

- *docker-compose --version*



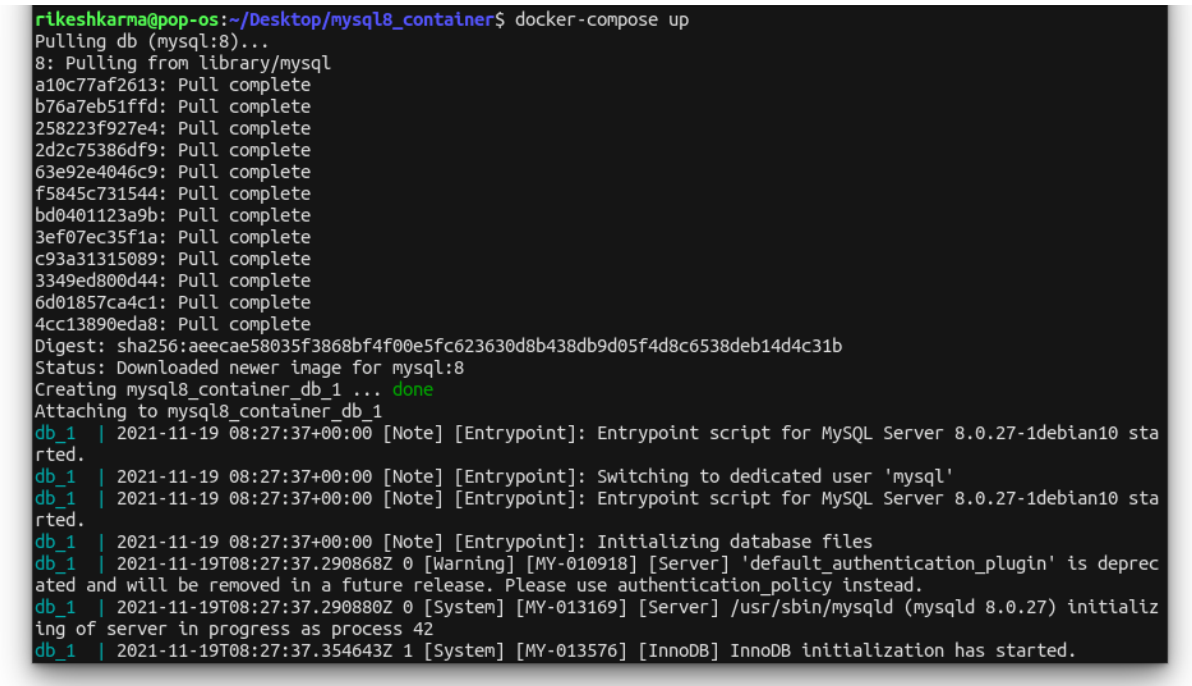
➔ Now we can create a file called *docker-compose.yml*. This file is essentially an instructions sheet for Docker where we tell docker what to do while creating the container. We write the following lines of code on the docker-compose.yml file as on the snapshot below:

docker-compose.yml

```yaml
version: '3.3'

services:
  db:
    image: mysql:8
    command: --default-authentication-plugin=mysql_native_password
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: sql_testpassword
    volumes:
      - my-test-db:/var/lib/mysql
    ports:
      - '3306:3306'
# Names our volume
volumes:
  my-test-db:
```

➔ Now after this we can start our container by running the following command in the terminal inside the directory where we created our *docker-compose.yml* file.

- *docker-compose up*



```
db_1  |
db_1  | 2021-11-19 08:30:34+00:00 [Note] [Entrypoint]: Stopping temporary server
db_1  | 2021-11-19T08:30:34.958625Z 10 [System] [MY-013172] [Server] Received SHUTDOWN from user root. Shuttin
g down mysqld (Version: 8.0.27).
db_1  | 2021-11-19T08:31:11.411210Z 0 [System] [MY-010910] [Server] /usr/sbin/mysqld: Shutdown complete (mysql
d 8.0.27)  MySQL Community Server - GPL.
db_1  | 2021-11-19 08:31:11+00:00 [Note] [Entrypoint]: Temporary server stopped
db_1  |
db_1  | 2021-11-19 08:31:11+00:00 [Note] [Entrypoint]: MySQL init process done. Ready for start up.
db_1  |
db_1  | 2021-11-19T08:31:12.271303Z 0 [Warning] [MY-010918] [Server] 'default_authentication_plugin' is deprec
ated and will be removed in a future release. Please use authentication_policy instead.
db_1  | 2021-11-19T08:31:12.271325Z 0 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 8.0.27) starting
as process 1
db_1  | 2021-11-19T08:31:12.310193Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
db_1  | 2021-11-19T08:31:14.075917Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
db_1  | 2021-11-19T08:31:15.375272Z 0 [Warning] [MY-013746] [Server] A deprecated TLS version TLSv1 is enabled
 for channel mysql_main
db_1  | 2021-11-19T08:31:15.375320Z 0 [Warning] [MY-013746] [Server] A deprecated TLS version TLSv1.1 is enabl
ed for channel mysql_main
db_1  | 2021-11-19T08:31:15.376472Z 0 [Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.
db_1  | 2021-11-19T08:31:15.376527Z 0 [System] [MY-013602] [Server] Channel mysql_main configured to support T
LS. Encrypted connections are now supported for this channel.
db_1  | 2021-11-19T08:31:15.575885Z 0 [Warning] [MY-011810] [Server] Insecure configuration for --pid-file: Lo
cation '/var/run/mysqld' in the path is accessible to all OS users. Consider choosing a different directory.
db_1  | 2021-11-19T08:31:15.601174Z 0 [System] [MY-011323] [Server] X Plugin ready for connections. Bind-addre
ss: '::' port: 33060, socket: /var/run/mysqld/mysqlx.sock
db_1  | 2021-11-19T08:31:15.601268Z 0 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. V
ersion: '8.0.27'  socket: '/var/run/mysqld/mysqld.sock'  port: 3306  MySQL Community Server - GPL.
```

➔ We have created a container with a docker-compose file for mysql8. We can observe that the MySQL instance is running on *localhost:3306*.

3. Create a Dockerfile for nodejs, react and Nginx and also create a container with docker-compose file and map the port and volume in the same network.

➔ To create a Dockerfile for node js:

◆ Firstly I created a simpler Node.js project using the following commands:

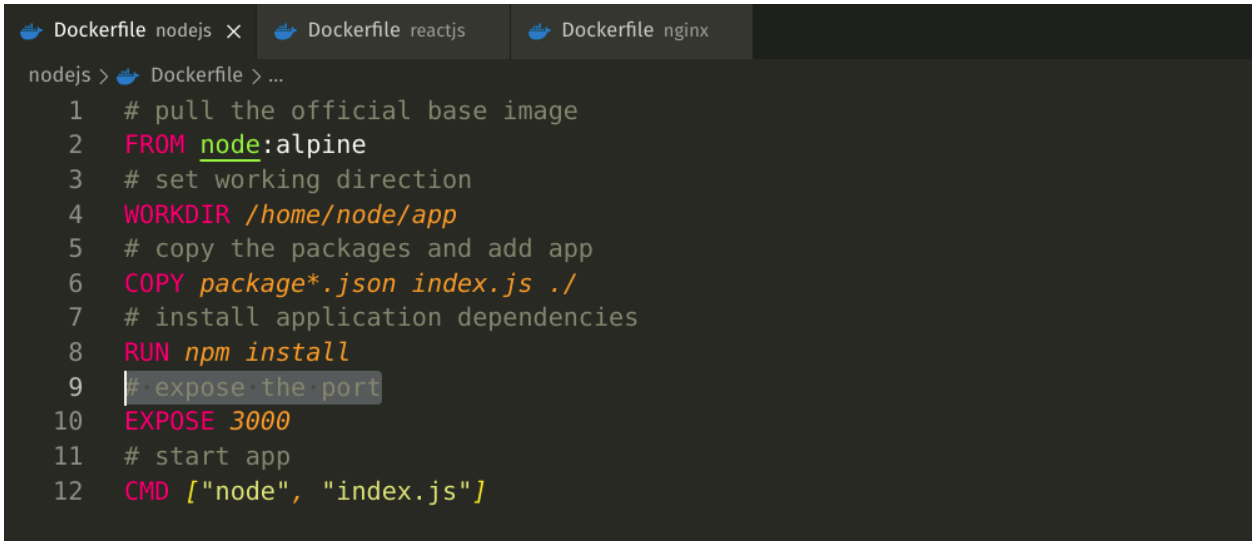
- To initialize the project:
  - *npm init -y*
- To install express.js
  - *npm install express*
- Then I create a file named “index.js” and write this code:

index.js > ...

```
1  // load express module with `require` directive
2  var express = require('express')
3  var app = express()
4
5  // define request response in root URL (/)
6  app.get('/', function (req, res) {
7    res.send('Hello World!')
8  })
9
10 // launch listening server on port 3000
11 app.listen(3000, function () {
12   console.log('app listening on port 3000!')
13 })
```

- Then we can run the following command to test if it works:
  - *node index.js*

◆ Now after knowing our project is running the file. Let's create a Dockerfile,
- Let's create a file named "*Dockerfile*" in the project directory.
- Then write this code inside the Dockerfile:



- We can also create .*dockerignore* if we want to which will ignore the files/ directory we don't want to push to docker hub. This file works like .*gitignore*.

➔ To create a Dockerfile for reactjs:
◆ As I have already installed *create-react-app* for a previous assignment, I can skip the installation part here.
◆ Now let's create a new React application using the following command:
- *create-react-app reactjs*
- Then cd into the reactjs directory and run to the following command to check if its working fine:
  - *cd reactjs*
  - *npm start*

◆ Now as our reactjs app is working fine. Let's create a *Dockerfile* for this app same as we did for nodejs:
- Let's create a file named "*Dockerfile*" in the project directory.
- Then write this code inside the Dockerfile:

Dockerfile nodejs | Dockerfile reactjs | Dockerfile nginx

reactjs > Dockerfile > ...

```
# pull the official base image
FROM node:alpine
# set working direction
WORKDIR /app
# add `/app/node_modules/.bin` to $PATH
ENV PATH /app/node_modules/.bin:$PATH
# copy the packages and add app
COPY package*.json . ./
# install application dependencies
RUN npm install
# expose the port
EXPOSE 4000
# start app
CMD ["node", "start"]
```

- We can also create *.dockerignore* if we want to which will ignore the files/ directory we don't want to push to docker hub. This file works like *.gitignore*.

Dockerfile | .dockerignore

reactjs > .dockerignore

```
node_modules
npm-debug.log
```

➔ To create a Dockerfile for Nginx:

- ◆ For this let's create a directory with the name *"Nginx"* and inside the directory, let's create a simple HTML file named *"index.html".*This is our home page when we access the Nginx webserver. Let's write this simple code inside the index.html file.

index.html

nginx > index.html > html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Nginx landing page!!</title>
  </head>
  <body>
    <h1>Welcome to Nginx Web Server</h1>
    <p>This server is running through Docker container</p>
  </body>
</html>
```

- Now let's create a Dockerfile for Nginx like we have done before:

Dockerfile nodejs | Dockerfile reactjs | Dockerfile nginx

nginx > Dockerfile > ...

```
# pull the official base image
FROM nginx:latest
# copy the packages and add app
COPY ./index.html /usr/share/nginx/html/
# expose the port
EXPOSE 5000
```

➔ After we are done creating Dockerfiles for all the services. Let's now create a *docker-compose.yml* file to combine all three created Dockerfiles.

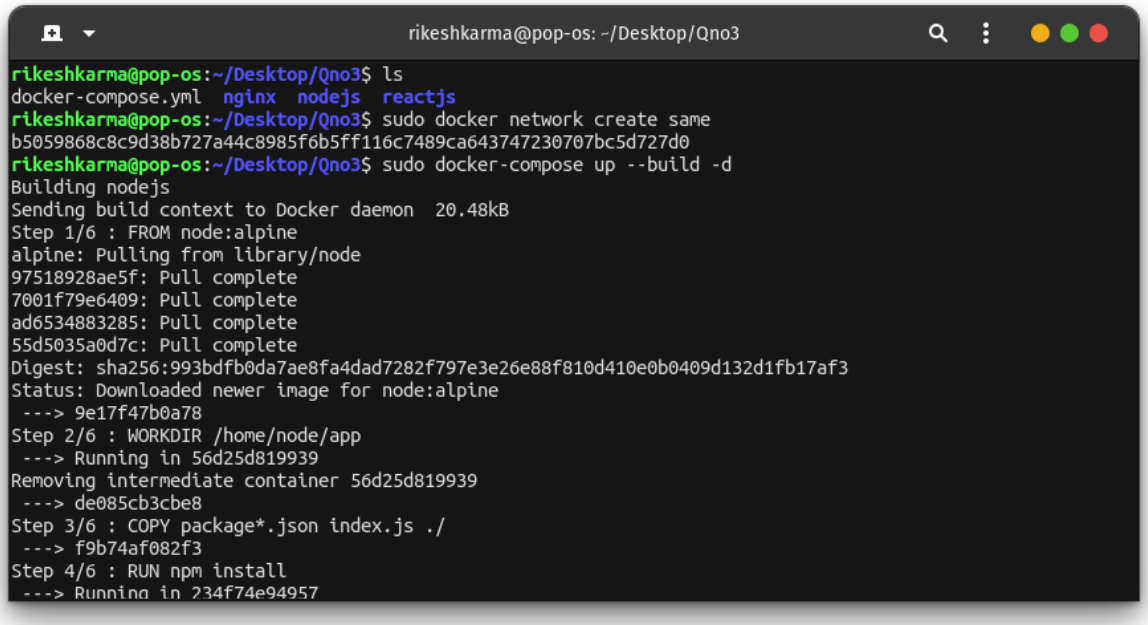
- At the root of this task, let's create a *docker-compose.yml* file.

docker-compose.yml

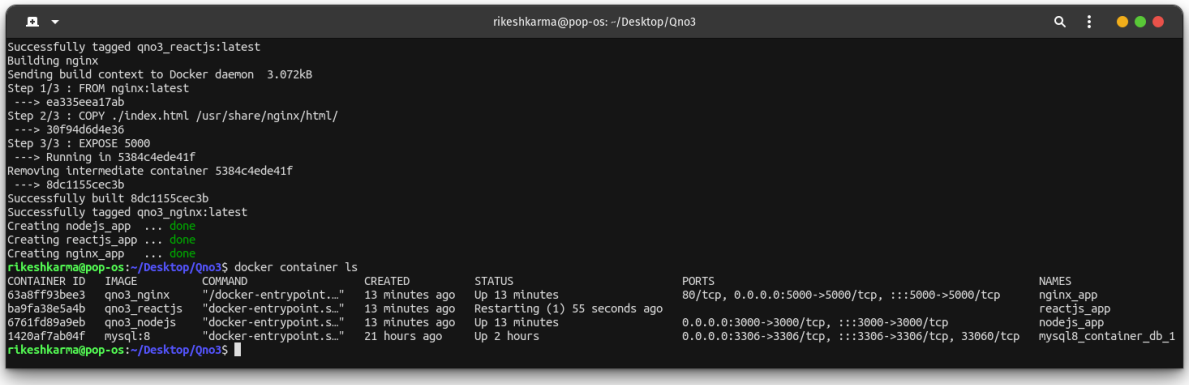
```
version: '3.8'

services:
  nodejs:
    # Gives location to the Dockerfile
    build:
      dockerfile: Dockerfile
      context: ./nodejs
    # Restarts the container if any errors
    restart: always
    # Names our container for this service
    container_name: nodejs_app
    # Mapping our local files/ directory from host path to container path
    volumes:
      - ./nodejs:/usr/src/app
    ports:
      - "3000:3000"

  reactjs:
    # Gives location to the Dockerfile
    build:
      dockerfile: Dockerfile
      context: ./reactjs
    # Restarts the container if any errors
    restart: always
    # Names our container for this service
    container_name: reactjs_app
    # Mapping our local files/ directory from host path to container path
    volumes:
      - ./reactjs:/usr/src/app
    ports:
      - "4000:4000"

  nginx:
    # Gives location to the Dockerfile
    build:
      dockerfile: Dockerfile
      context: ./nginx
    # Restarts the container if any errors
    restart: always
    # Names our container for this service
    container_name: nginx_app
    # Mapping our local files/ directory from host path to container path
    volumes:
      - ./nginx/index.html:/usr/share/nginx/html/index.html
    ports:
      - "5000:5000"

# To map the ports of all servies to use the same network
networks:
  default:
    external:
      name: same
```

➔ After creating the docker compose file, now let's create a network so that all the services will be mapped to the same port using the following command:
  - ◆ *sudo docker network create same*

➔ Now let's build the docker image by running this command from the root folder:
  - ◆ *sudo docker-compose up -d*



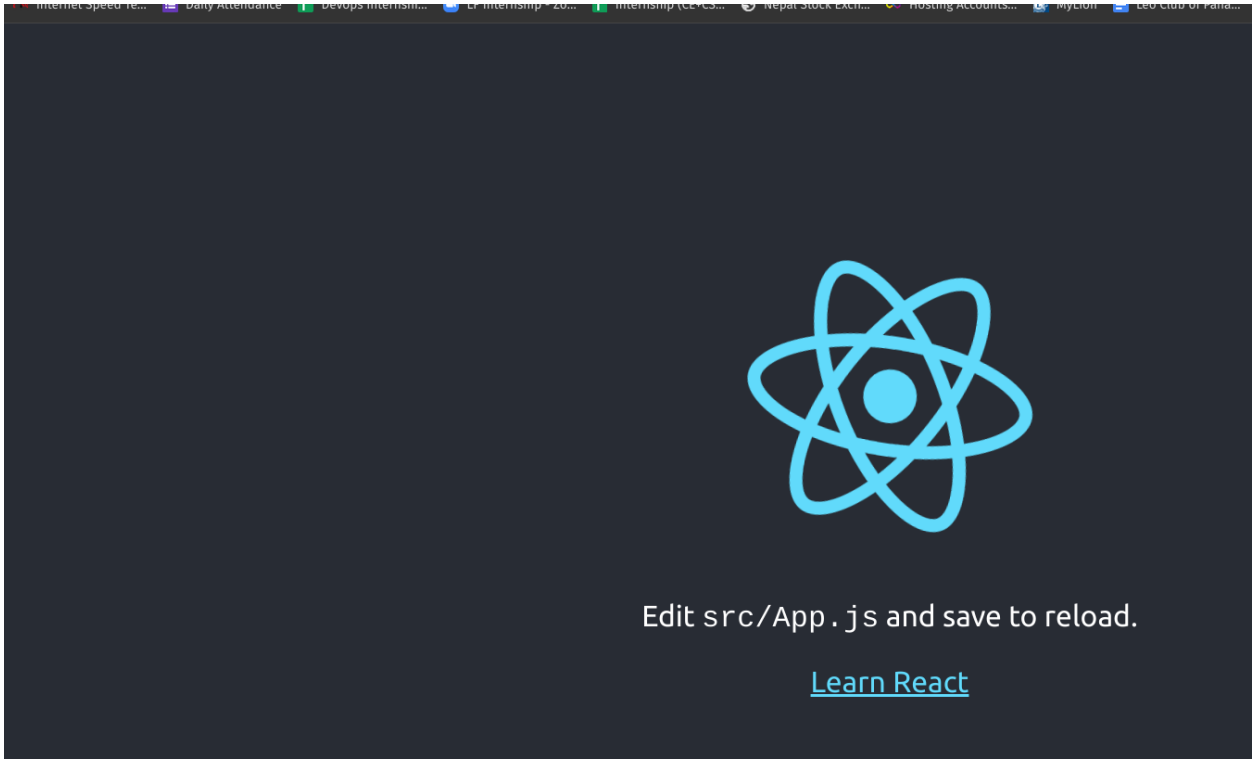
➔ Now after the completion of the process. We can check to see if the containers are running:



➔ As we can observe all three services are running, we can browse them on the ports to see if all three services are running.
  - ◆ NodeJS

Hello World!

- ReactJS

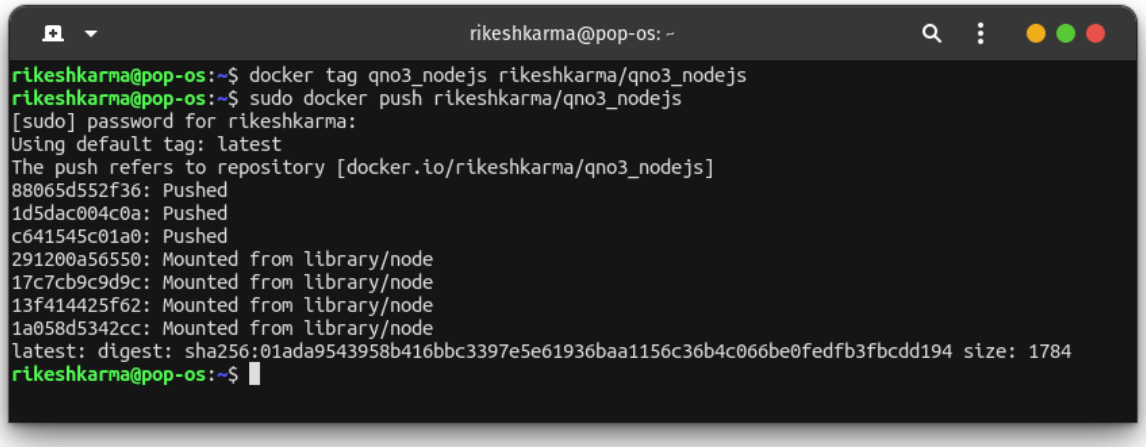


- Nginx

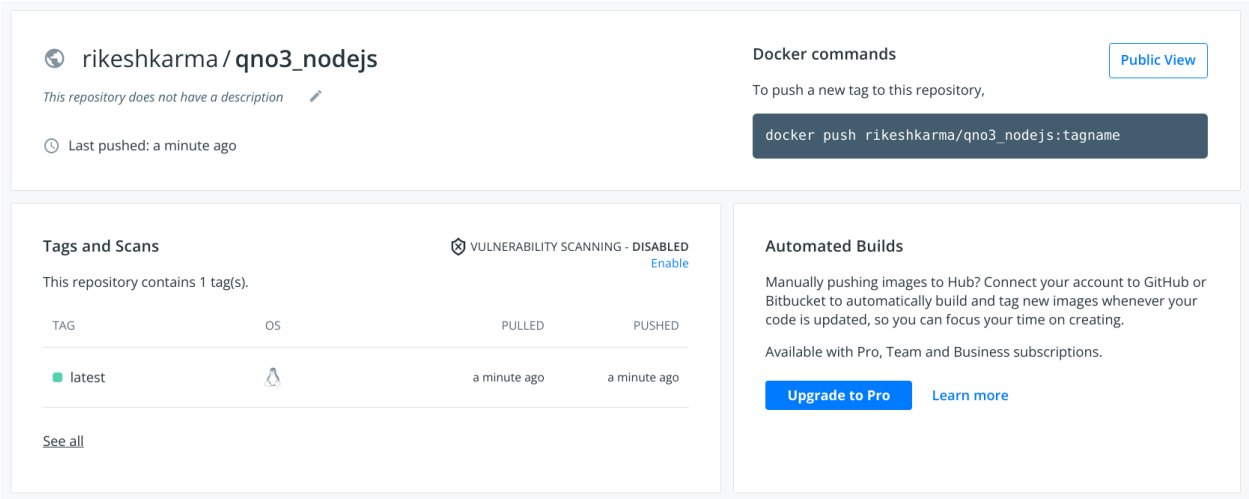
4. Now push the created docker images to the docker hub repository.
   - ➔ To push the created docker images to the docker hub repository:
     - First login to the docker hub in CLI using the following command:
       - *sudo docker login*
       - Fill in your credentials and log in.

```
rikeshkarma@pop-os:~$ sudo docker login
[sudo] password for rikeshkarma:
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker
ID, head over to https://hub.docker.com to create one.
Username: rikeshkarma
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
rikeshkarma@pop-os:~$
```

- Now first let's push our node js image to the docker hub repository by using the following command:
  - Before we push we need to tag the locally created image to the docker hub, which means we need to tag the image with the docker hub username using the following command:
    - *docker tag qno3_nodejs rikeshkarma/qno3_nodejs*

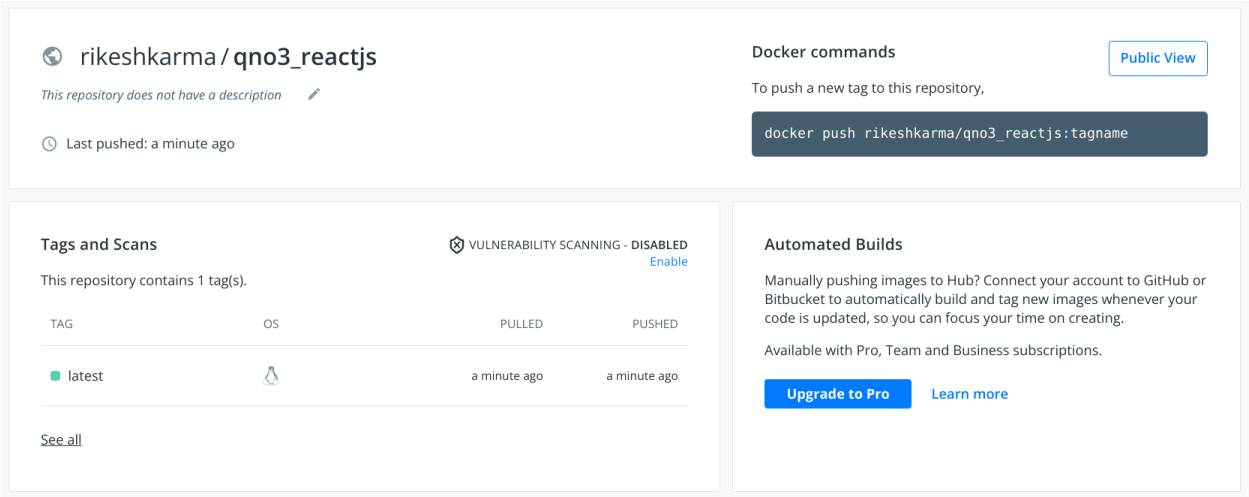


    - *sudo docker push rikeshkarma/qno3_nodejs*



- Now first let's push our react js image to the dockerdocker hub repository by using the following command:
  - Before we push we need to tag the locally created image to the docker hub, which means we need to tag the image with the docker hub username using the following command:
    - *docker tag qno3_reactjs rikeshkarma/qno3_reactjs*
    - *sudo docker push rikeshkarma/qno3_reactjs*

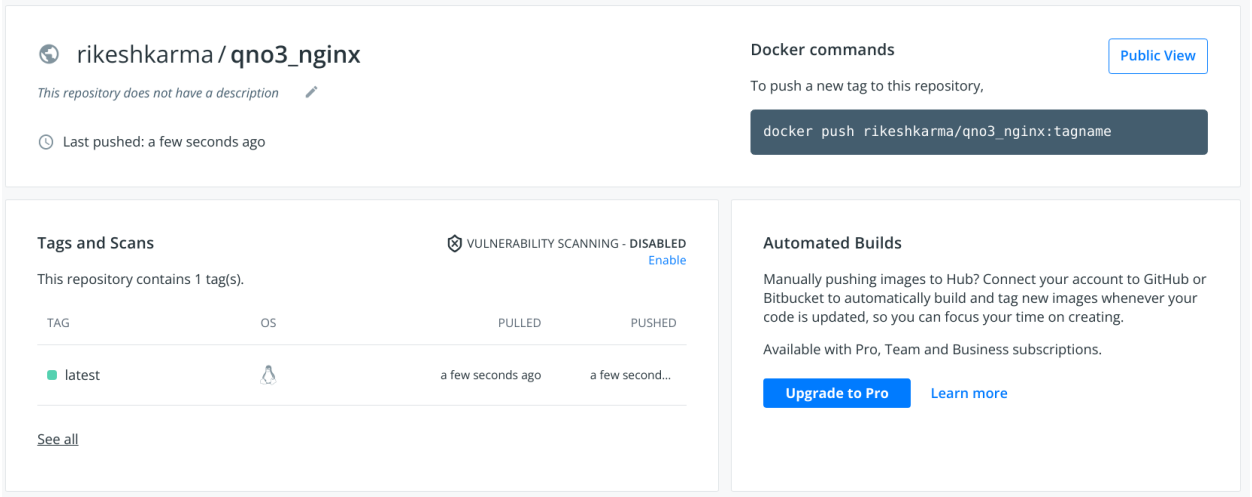
```
rikeshkarma@pop-os: ~
rikeshkarma@pop-os:~$ docker tag qno3_reactjs rikeshkarma/qno3_reactjs
rikeshkarma@pop-os:~$ sudo docker push rikeshkarma/qno3_reactjs
Using default tag: latest
The push refers to repository [docker.io/rikeshkarma/qno3_reactjs]
ecf674a8fa07: Pushed
a00c736320ef: Pushed
d5da76cbff0c: Pushed
291200a56550: Mounted from rikeshkarma/qno3_nodejs
17c7cb9c9d9c: Mounted from rikeshkarma/qno3_nodejs
13f414425f62: Mounted from rikeshkarma/qno3_nodejs
1a058d5342cc: Mounted from rikeshkarma/qno3_nodejs
latest: digest: sha256:7eb814ccaf3b67955975c21d2b06e7d83ab2995e31417ef0c71ccf85cd974ef4 size: 1787
rikeshkarma@pop-os:~$
```



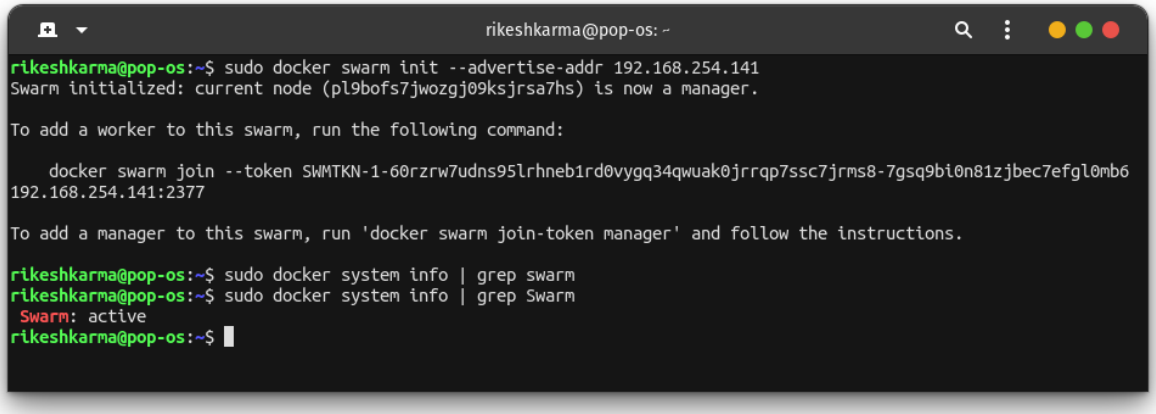
- Now first let's push our node js image to docker hub repository by using the following command:
  - Before we push we need to tag the locally created image to the docket hub, which means we need to tag the image with the docker hub username using the following command:
    - *docker tag qno3_nginx rikeshkarma/qno3_nginx*

```
rikeshkarma@pop-os:~$ docker tag qno3_nginx rikeshkarma/qno3_nginx
rikeshkarma@pop-os:~$ sudo docker push rikeshkarma/qno3_nginx
Using default tag: latest
The push refers to repository [docker.io/rikeshkarma/qno3_nginx]
a64486a117ab: Pushed
8525cde30b22: Mounted from library/nginx
1e8ad06c81b6: Mounted from library/nginx
49eeddd2150f: Mounted from library/nginx
ff4c72779430: Mounted from library/nginx
37380c5830fe: Mounted from library/nginx
e1bbcf243d0e: Mounted from library/nginx
latest: digest: sha256:7dd41d9f0a6143dbd32fa391c7a256cd2ee636b27ce422d5979b40c785e4a867 size: 1777
rikeshkarma@pop-os:~$
```

- *sudo docker push rikeshkarma/qno3_nginx*

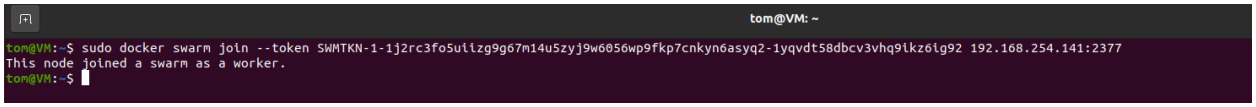


5. Install docker swarm and manage the cluster of all created containers.
   - ➔ Docker Swarm is a clustering tool that turns a group of Docker hosts into a single virtual server. Docker Swarm ensures availability and high performance for your application by distributing it over the number of Docker hosts inside a cluster. Docker Swarm also allows you to increase the number of container instances for the same application.
   - ➔ The Docker swarm has already been installed automatically while we install the docker engine, so we don't need to do it manually. Now we just need to initialize it, for initializing we can invoke the following command:
     - ◆ *sudo docker swarm init --advertise-addr <host-ip>*
     - ◆ After using this command the docker swarm should be initialized.
     - ◆ Let's verify that in case using the following command:
       - *sudo docker system info | grep Swarm*

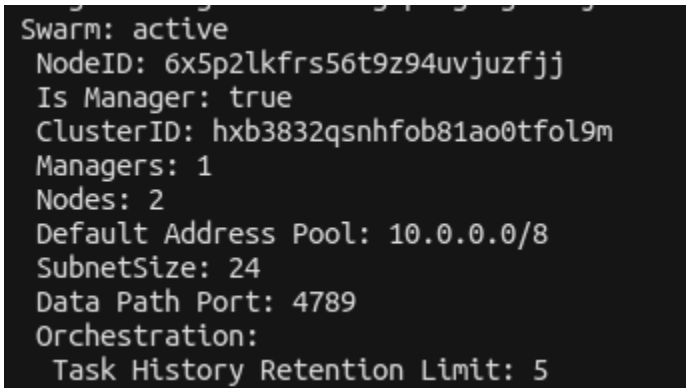


➔ Firstly we need to join the worker's node to manage the clusters of all created containers, so let's do it::

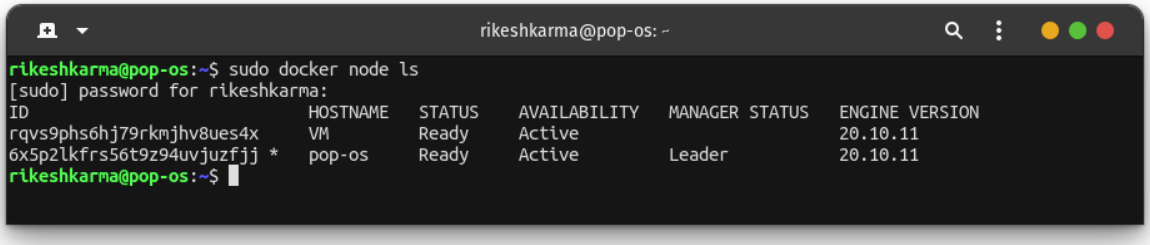
- ◆ Initially, we need to install Docker Engine in the worker node. toun the command that we got when we used swarm init in the VM so as to add a worker in the swarm. The command generated was:
  - *docker swarm join --token SWMTKN-1-1j2rc3fo5uiizg9g67m14u5zyj9w6056wp9fkp7cnkyn6asyq2-1yqvdt58dbcv3vhq9ikz6ig92 192.168.254.141:2377*



- ◆ Now we can see that the node has been added to swarn as a worker if we view the Docker info in manager mode using the following command:
  - *sudo docker info*



  - We can see that now there are **2 nodes**.
- ◆ We can check the information about the nodes using the following command:
  - *sudo docker node ls*



  - We can see that there is a VM as a node as well. We can also see that our host machine is leader and the VM is the worker (empty MANAGER STATUS) also both are active.

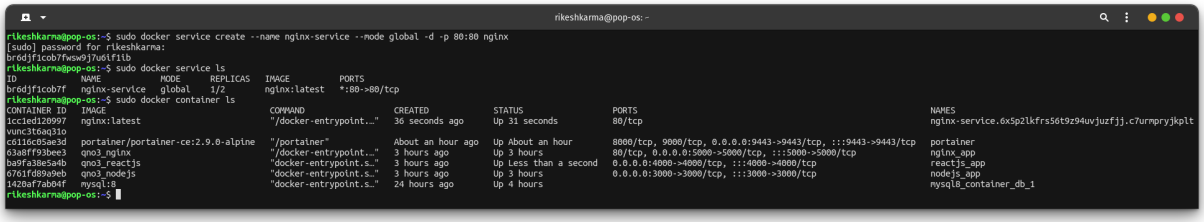
➔ Swarm services use a declarative model, which means that you define the desired state of the service, and rely upon Docker to maintain this state.

➔ Now let's create a swarm service using the following command:

- *sudo docker service create --name Nginx-service --mode global -d -p 80:80 Nginx*
  - **--mode:-** This option will help to create the container in our desired mode. As our model for this command is global, which means two containers will be created on the 2 nodes we have i.e manager node and the worker node.

➔ After creating the service we can check the services and list the containers running by invoking the following commands:

- *sudo docker service ls*
- *sudo docker container ls*



➔ Now we can deploy the docker-compose.yml to the worker node with custom build images we need to invoke the following commands:

- But before that, we need to make some changes to the docker-compose.yml file. The snapshot of yml file after making the modifications is below and the file is also available in the task repo:

- Added modifications in the docker-compose.yml files are:
  - **image:-** links to the previously pushed docker image. (dockerhubID/image-name)
  - **networks:-** changes have been name on the restart policy, stacking the network and replicas have been added to all three services. Also, mapping ports of all services drivers has been added as an overlay.

docker-compose.yml

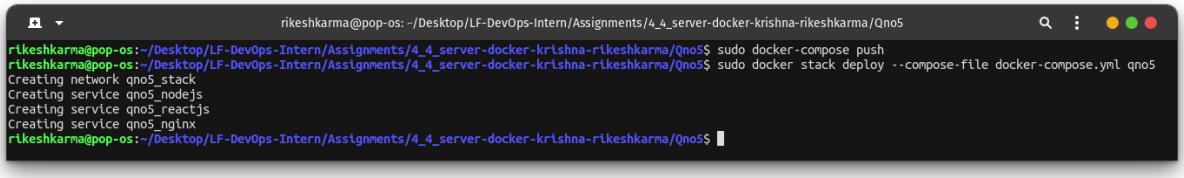
```yaml
version: '3.8'

services:
  nodejs:
    # Gives link to the Docker image
    image: rikeshkarma/qno3_nodejs
    ports:
      - "3000:3000"
    networks:
      - stack
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure

  reactjs:
    # Gives link to the Docker image
    image: rikeshkarma/qno3_reactjs
    ports:
      - "4000:4000"
    networks:
      - stack
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure

  nginx:
    # Gives link to the Docker image
    image: rikeshkarma/qno3_nginx
    networks:
      - stack
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
    ports:
      - "5000:5000"

# To map the ports of all servies to use the same network
networks:
  stack:
    driver : overlay
```

➔ We assign the service to run containers in the choice of our node by changing this on the docker-compose.yml file: (following to run on the manager mode)

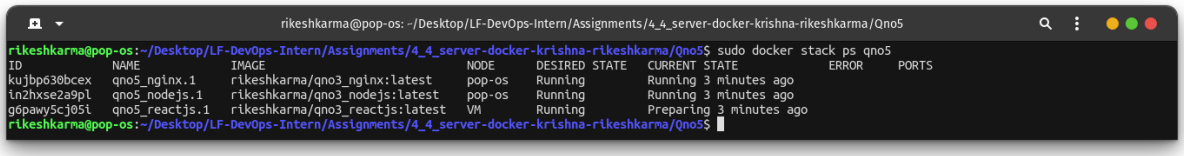
- *deploy:*
  placement:
  constraints:
  - *node.role == manager*

➔ Now let'snaming deploy to the worker node i.e in the VM.

- First, we need to push the compose file to the Docker hub.
  - For this let's navigate to the directory where the docker-compose.yml file is located.
  - Then run the following command in the terminal:
    - *sudo docker-compose push*
- Now to deploy stack with composer file use the following command:
  - *sudo docker stack deploy --compose-file docker-compose.yml qno5*



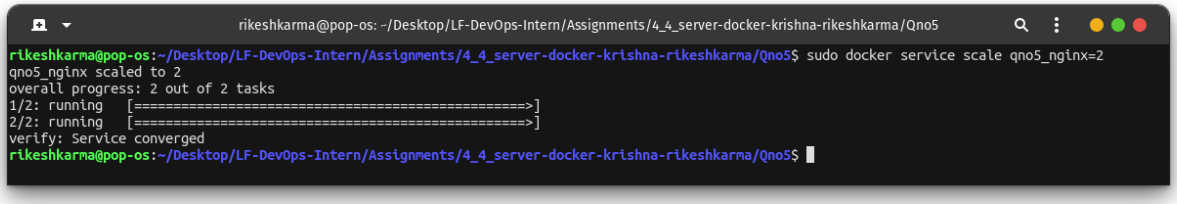
- We can now check the details of the stack we just deployed using the following command:
  - *sudo docker stack ps qno5*



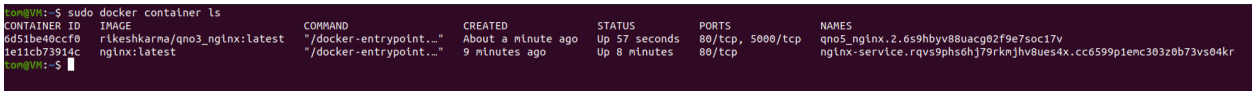
  - We can observe that the qno5_reactjs container has been deployed to the worker node i.e in the VM.
- We can now check on the VM as well for the containers, it must show the reactjs container created.
  - We can use the following command to verify:
    - *sudo docker container ls*

- If we need we can also scale containers using the service from the manager node to the worker node. For this we need to invoke this command:
  - *sudo docker service scale <network-name>=num.*
  - In our case we will use this command:

- *sudo docker service scale qno5_nginx=2*



- ◆ Let's check on the VM i.e worker-node for the running containers again using this command:
  - *sudo docker container ls*



  - Here the container with my react js is not showing because just before this my VM crashed while taking the screenshot for the previous ls command to show reactjs running on the worker node and I tried to review the previous processes but due to some issue it's not showing, my apologies.

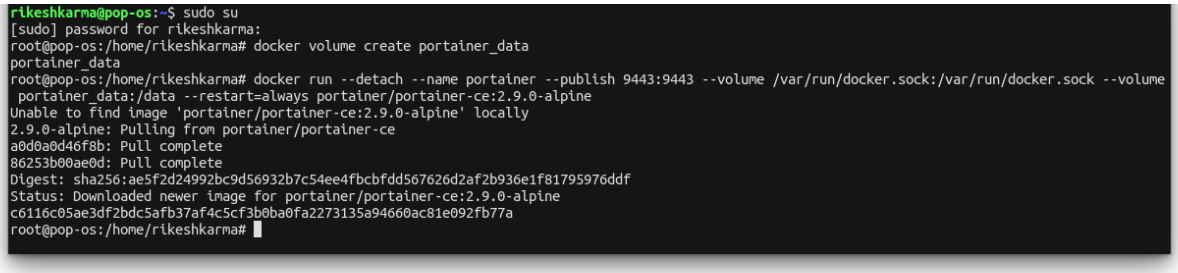
6. Install the portainer and manage the docker from GUI. Practice the features and attach the snapshots.

➔ Portainer is a Web User Interface (WUI/GUI) dashboard tool, which can be used to monitor and centrally manage a docker platform. With Portainer, We can add/login to a registry, Manage multiple hosts, Manage hosts on different networks, Build a Container (Docker) Image, pull and push an image from/to container registry, Manage running/deployed containers, deploy a stack/container, Manage accounts, and many more. Other than the standalone docker platform, Portainer also supports Docker Swarm and Kubernetes cluster. Above that, All of them can be managed from a single centrally managed dashboard UI.

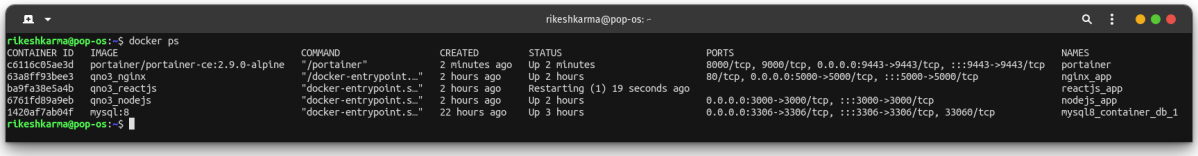
➔ Now let's start by installing portainer first:

- ◆ The installation requires root access for the proper functioning of the dashboard so let's invoke this command first:
  - *sudo us*
- ◆ Let's create a docker volume for portainer data by using this command:
  - *docker volume create portainer_data*
- ◆ Now let's deploy portainer server on our local machine using this command:

- *docker run --detach --name portainer --publish 9443:9443 --volume /var/run/docker.sock:/var/run/docker.sock --volume portainer_data:/data --restart=always portainer/portainer-ce:2.9.0-alpine*

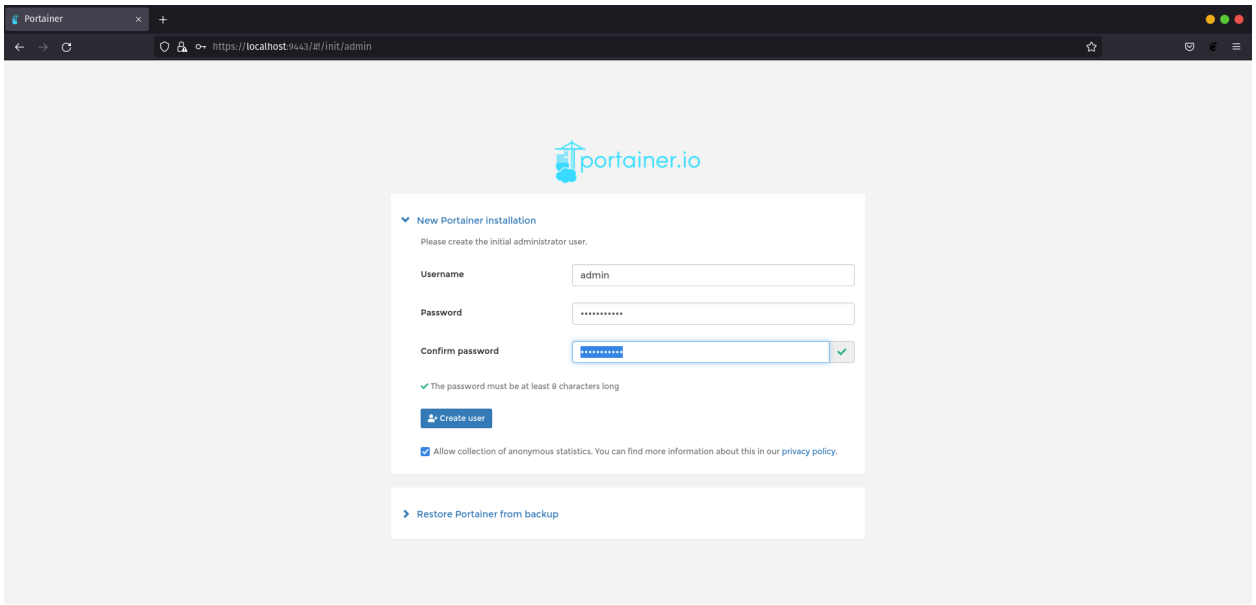


- Now Port 9443 can be used for web UI access. Docker socket is required to be mounted on Portainer container to allow it to access docker daemon.
- We can check if portainer is up and running or not by invoking the following command:
    - *docker ps*

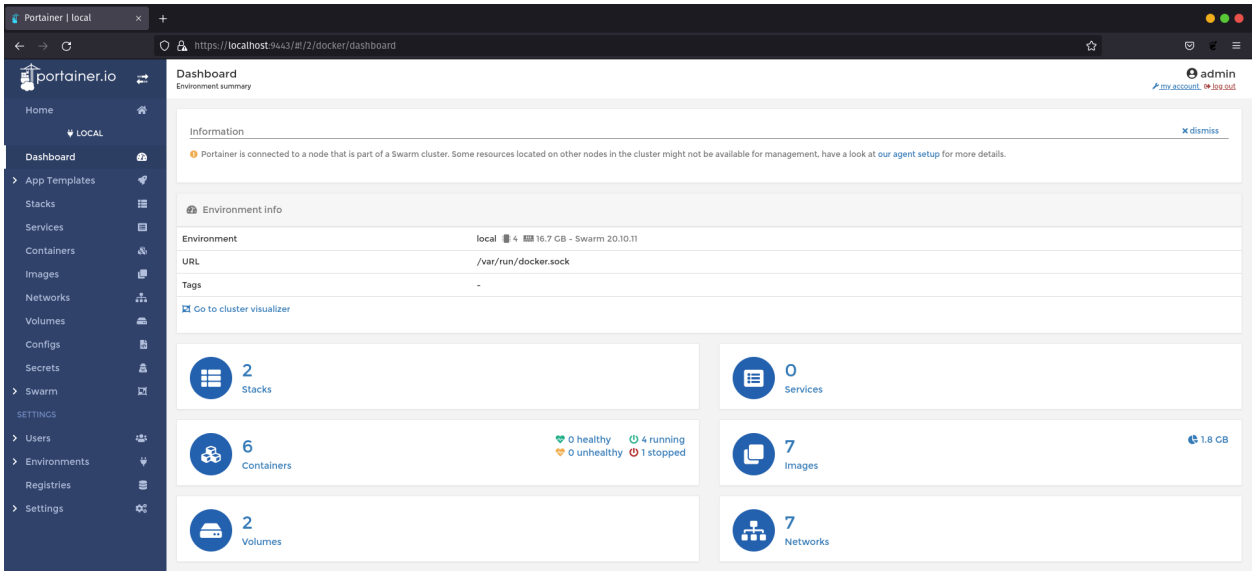


- We can observea lot that portainer server is up and running with CONTAINER ID c6116c05ae3d.

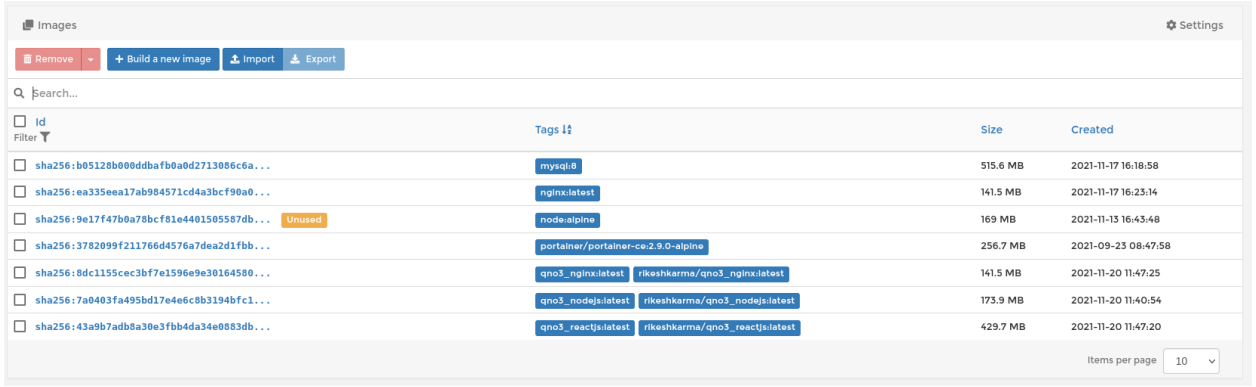
- Now let's check on our browser at *https://localhost:9443.*



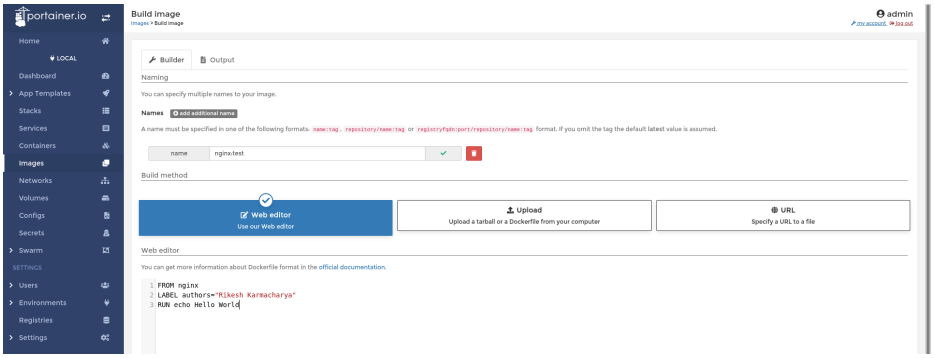
- Let's setup the admin credentials first. Then we can see our dashboard with all our local services, containers, images, volumes, networks and others.



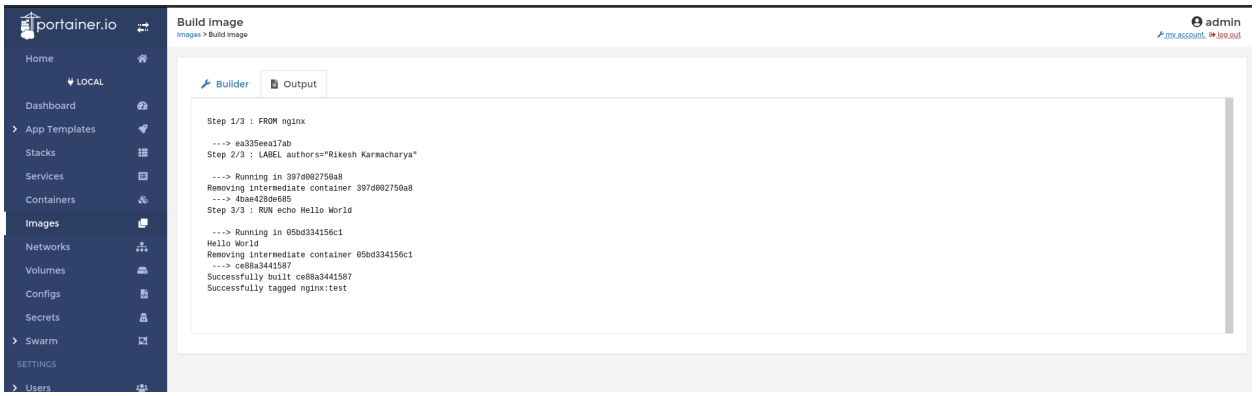
- We can find alot of features here. Let's practice some of the features:
  - Build a container/ docker image
    - Image -> Build a new image



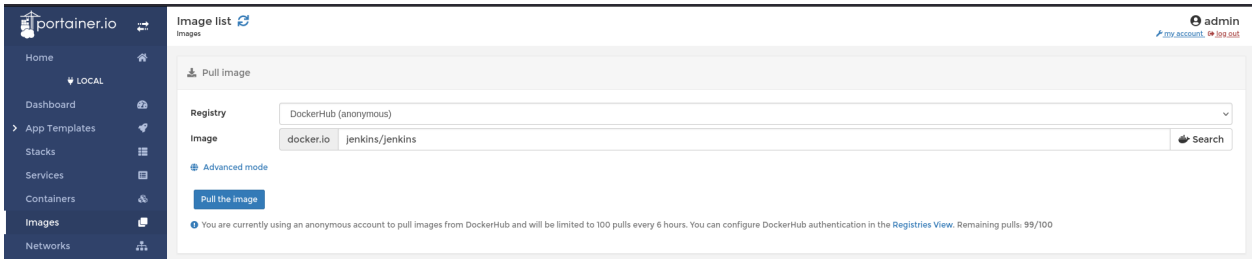
    - In image section, we can see all the list of the pulled images in the environment. We can also see the **Build a new image** button. We can click on it and we can see that we can write a Dockerfile in the web editor or upload a tarball or use an URL to build an image. Let's build a sample Nginx:test image,



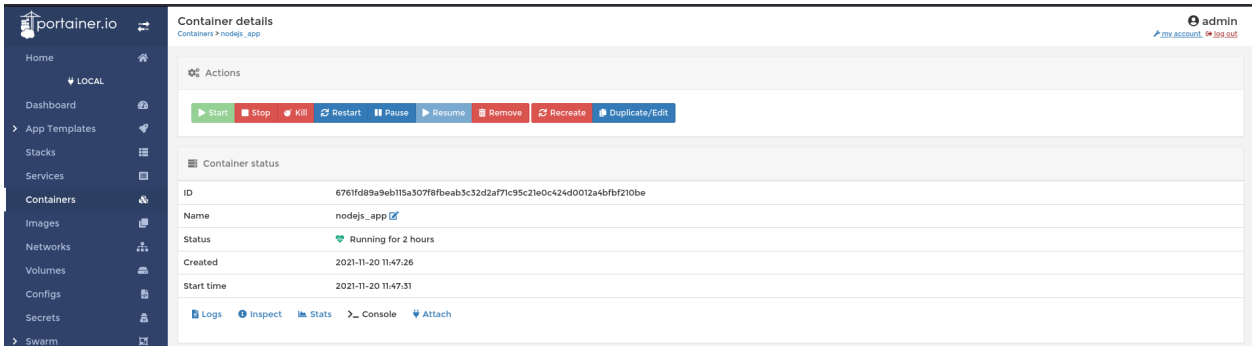
- We can see this working in the snapshot below:



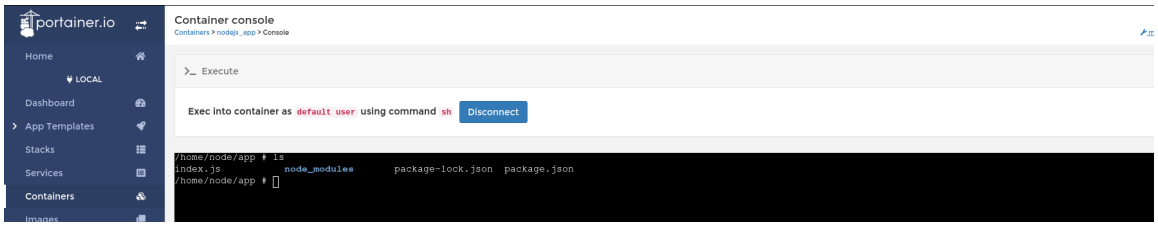
- Pull an image from a registry
  - Let's click on images from the left panel and choose registry from the drop-down list. We need to add image name with tag and then pull the image:



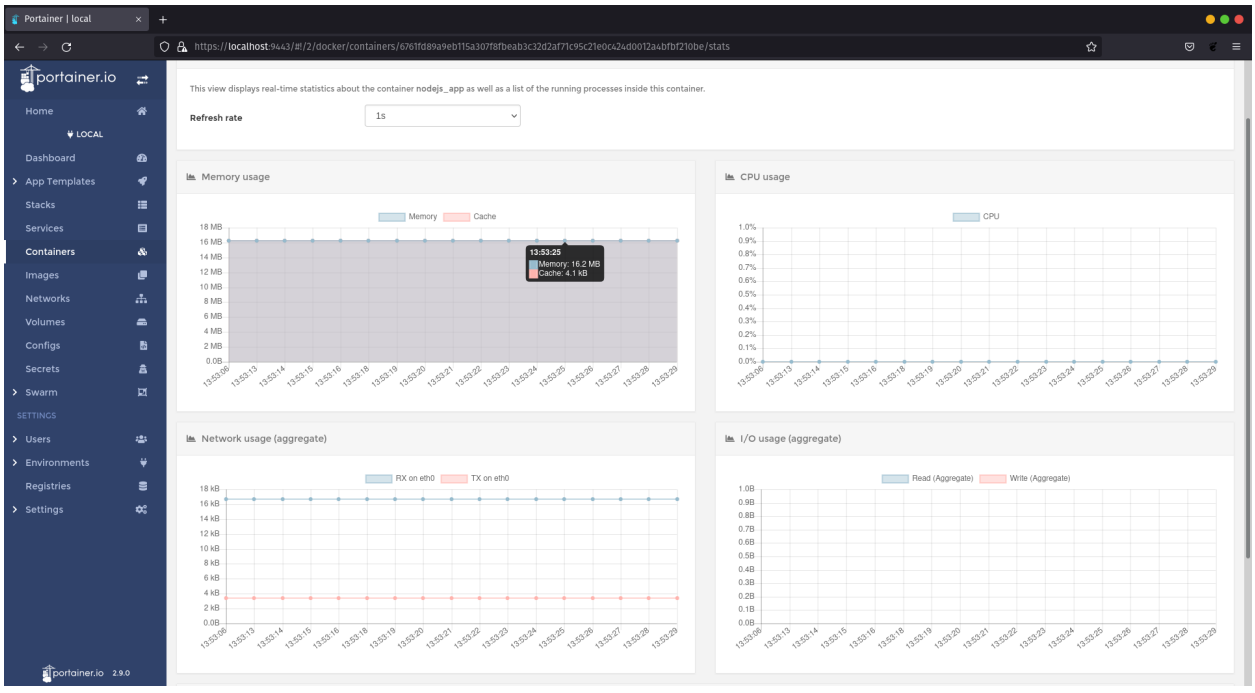
- Run commands inside the container (Exec inside the container)
  - Now let's select containers from the left panel and choose any running container. Then we can click on the console as in the snapshot below:



- After that, we can give it with a custom command or use shell binary present in the container.

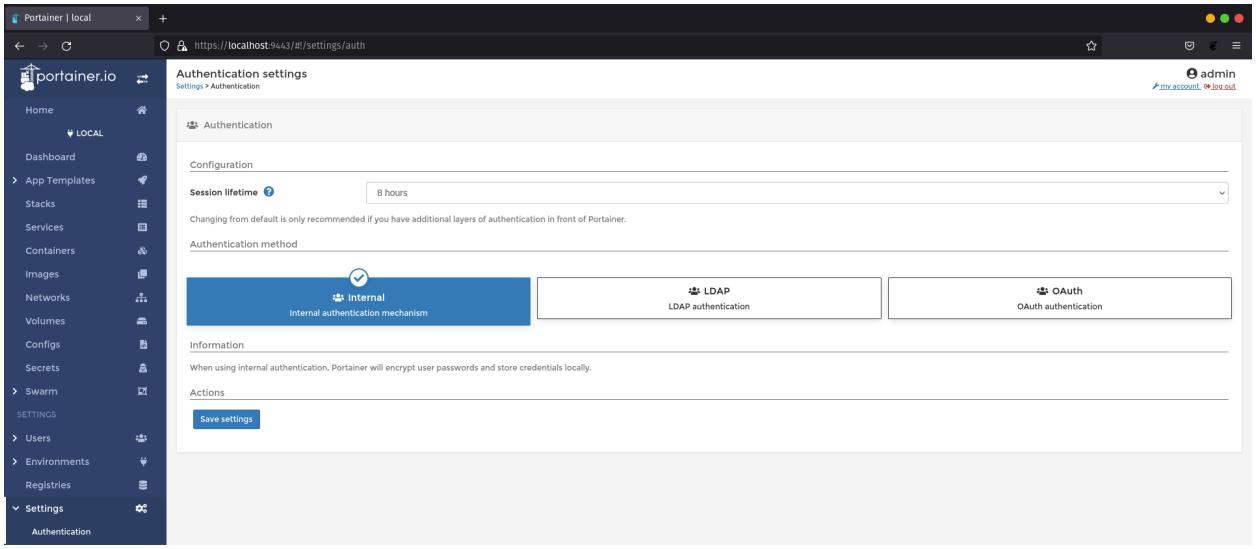


- Monitor resource usage of a container
  - Like before opening a container from a running container's list.
  - Then click on stats to monitor resource usages of the particular container.



- Manage accounts

- We can also set up custom authentication methods under settings and then click on authentication as on the snapshot below:
- We can create and manage Roles, Teams, and users from Servings and then the Users Section. This can be used to grant access to other developers and cut off their duties.



Findings:

- ➢ While creating a container from the docker-compose.yml file, I learned that Docker Compose provides us with a lot of flexibility and configurations that if we want to switch our configuration, in our case let's say database then we can do it in about a minute or even in seconds if we want, just by changing some lines in the *docker-compose.yml* file. And we just need to do is restart the Docket Compose.
- ➢ Found out that there is .dockerignore similar to .gitignore which allows us to mention a list of files and/or directories that we want to ignore while building the image. This also then helps us to reduce the size of the image and also helps to speed up the docker build process.
- ➢ While using the version of the docker-compose file, we should always update to the latest version whenever possible because it will likely contain bug fixes.
- ➢ Portainer creates its self-signed SSL certificate for secure communication between Portainer Server and User's browser and between Portainer Server and other Portainer agents. If you want to add your certificates, that can be done from

web UI or It can also be added during installation using sslcert and sslkey arguments passed as below:

- *--sslcert /path/to/certs/cername.crt --sslkey /path/to/certs/keyname.key*