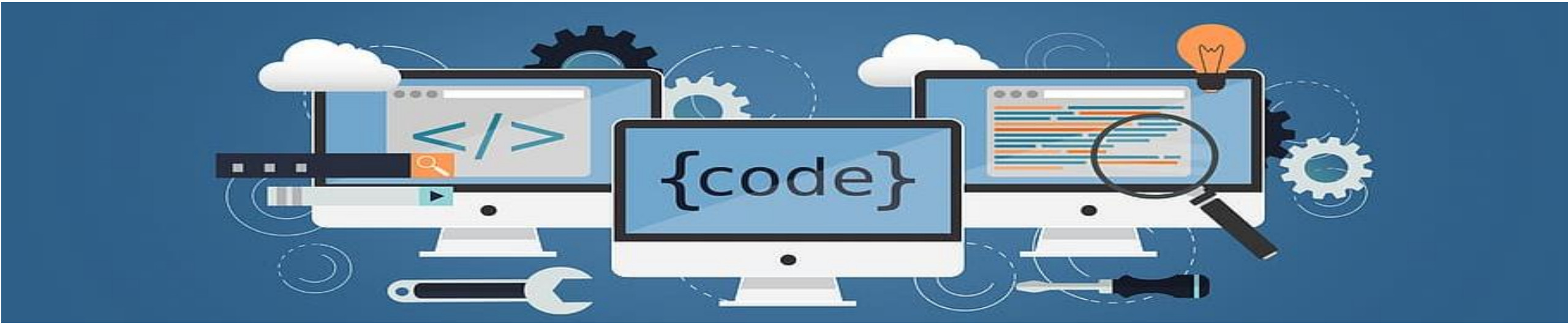
DES424
Cloud-based Application Development

Lecture 11: Docker Compose and Tutorial 2



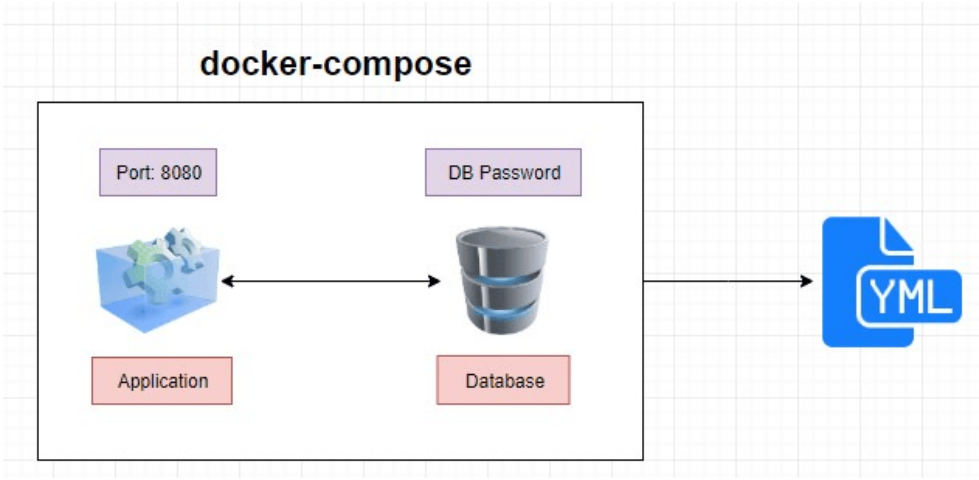
Dr. Apichon Witayangkurn
(apichon@siit.tu.ac.th)

Tutorial Roadmap

- Storage with Docker Volume
- Storage with Bind Host
- Create custom build web image with code from GitHub
- Dockerfile with build argument
- Dockerfile with environment variable
- Share your image on Docker Hub
- Deploy image on Amazon Elastic Beanstalk
- A Node.js application using Nginx, Docker and Redis
- Portainer – Docker Management
- Dashboard for monitoring dockers
- Voting App with docker-compose

Docker Compose

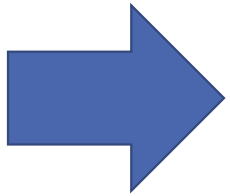
- It is a tool that assists in defining and sharing multi-container applications. By using Compose, we can define the services in a YAML file, as well as spin them up and tear them down with one single command.
- Using Compose is basically a three-step process:
  1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.
  2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
  3. Run docker compose up and the Docker compose command starts and runs your entire app.

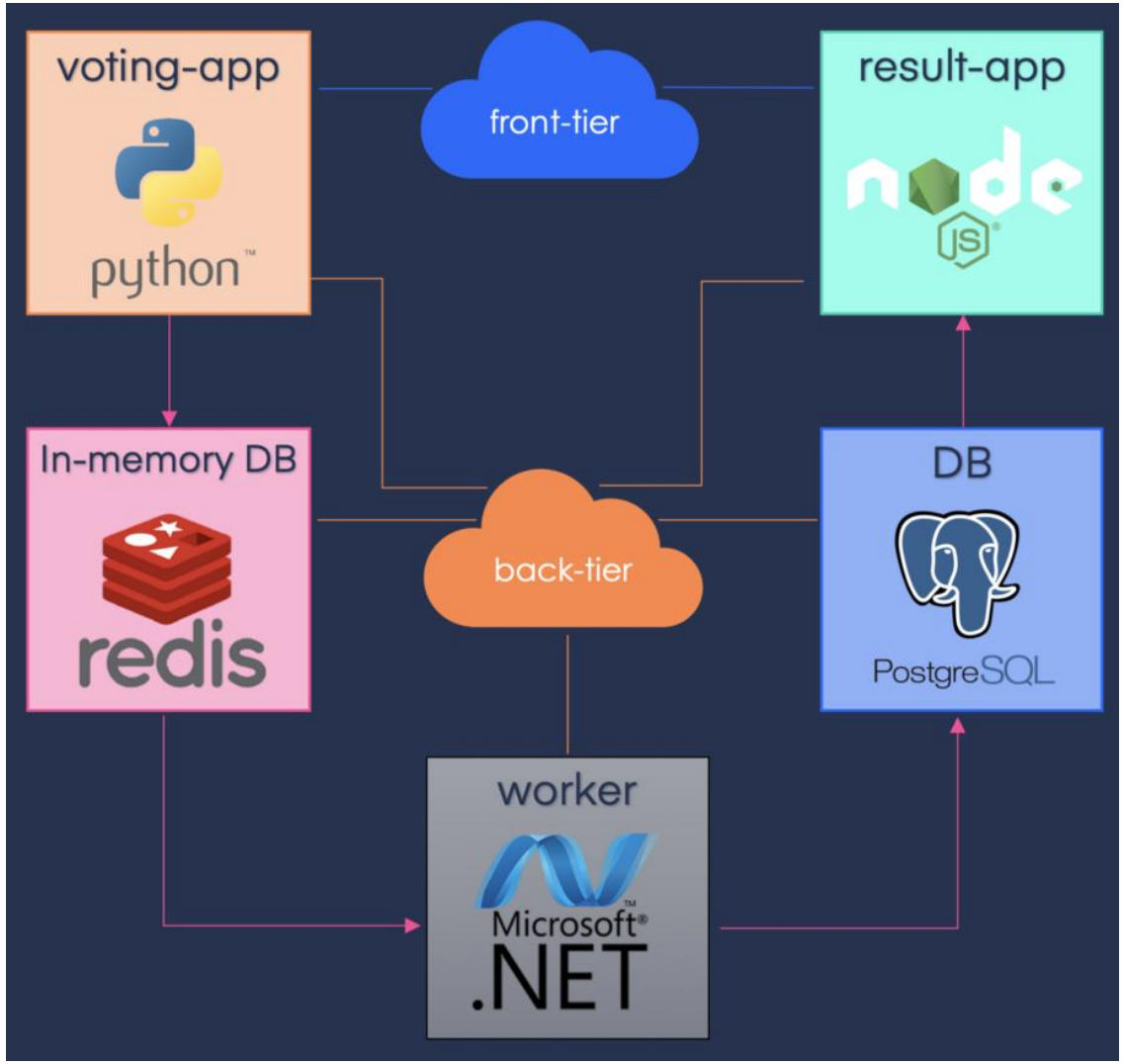


Docker Compose

docker-compose.yml

```
version: 3
services:
    redis:
        image: redis
        networks:
            - back-tier
    db:
        image: postgres:9.4
        networks:
            - back-tier
    vote:
        build: ./vote
        ports:
            - 5000:80
        networks:
            - front-tier
            - back-tier
    result:
    worker:
networks:
    front-tier:
    back-tier:
```





YAML for Docker Compose

- Docker Compose works by applying many rules declared within a single docker-compose.yml configuration file.
- In compose file (yml), it usually contain at least one service and optionally volumes and network.

Services refer to containers' configuration

Define physical areas of disk space shared between the host and a container

Define the network communication between containers, and between a container and the host

```
version: '3'
services:
  ...
volumes:
  ...
networks:
  ...
```

Example YAML for Docker Compose

```
version: '3'

services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
volumes:
  db_data:
```

Specify image

Set environment for db config

Database service (db)

Set service dependency

Map port with host

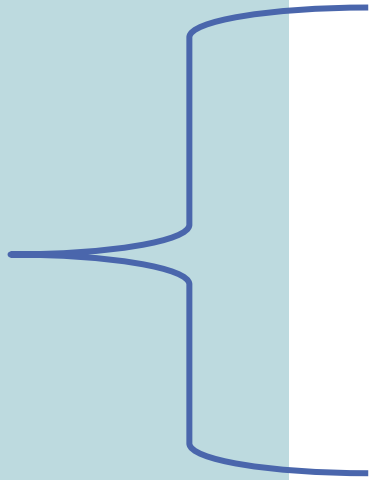
Web app (wordpress)

Volume visible to all services

Docker Compose : Services

- Services refer to containers' configuration.
- For example, let's take a dockerized web application consisting of a front end, a back end, and a database, then split those components into three images and define them as three different services in the configuration.
- Service can be image from Docker Hub or build via Dockerfile

```
services:
  frontend:
    image: my-vue-app
    ...
  backend:
    image: my-springboot-app
    ...
  db:
    image: postgres
    ...
```



```
services:
  my-service:
    image: ubuntu:latest
    ...
```

Pulling an Image

```
services:
  my-custom-app:
    build: /path/to/dockerfile/
    ...
```

Building an Image

```
services:
  my-custom-app:
    build: https://github.com/my-company/my-project.git
    image: my-project-image
    ...
```

Docker Compose : Networking

- A service can communicate with another service on the same network by simply referencing it by container name and port (for example network-example-service:80), provided that we've made the port accessible through the expose keyword:

```
services:
  network-example-service:
    image: karthequian/helloworld:latest
    expose:
      - "80"
```

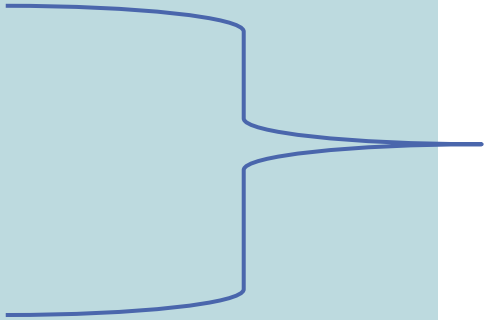
To reach a container from the host, the ports must be exposed declaratively through the ports keyword, which also allows us to choose if exposing the port differently in the host:

```
services:
  network-example-service:
    image: karthequian/helloworld:latest
    ports:
      - "80:80"
    ...
  my-custom-app:
    image: myapp:latest
    ports:
      - "8080:3000"
    ...
  my-custom-app-replica:
    image: myapp:latest
    ports:
      - "8081:3000"
    ...
```

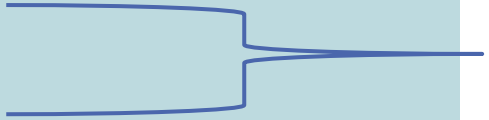
Docker Compose : Networking

- We can also create custom network aligning with system architecture.

```
services:
  webapp:
    image: my-vue-app
    ports:
        - "80:80"
    networks:
        - frontend_net
        - backend_net
  db:
    image: postgres
    networks:
        - backend_net
networks:
 frontend_net:
 backend_net:
```



frontend_net allow access from outside
backend_net use internally with database



Only backend_net for provide service to internal network

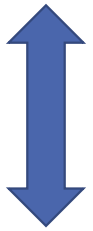
Docker Compose : Volume

- There are two types of volumes: named, and host ones.
- Docker manages named volumes, automatically mounting them in self-generated directories in the host. Host volumes also allow us to specify an existing folder in the host.
- We can configure host volumes at the service level and named volumes in the outer level of the configuration, in order to make the latter visible to other containers and not only to the one they belong:

```
services:
  volumes-example-service:
    image: alpine:latest
    volumes:
      - my-named-global-volume:/my-volumes/named-global-volume
      - /tmp:/my-volumes/host-volume
      - /home:/my-volumes/readonly-host-volume:ro
    ...
  another-volumes-example-service:
    image: alpine:latest
    volumes:
      - my-named-global-volume:/another-path/the-same-named-global-volume
    ...
volumes:
  my-named-global-volume:
```

Same volume as global

Visible only in this service
ro: read only



Same volume as global

Share volume, visible to all services

Docker Compose : Lifecycle Management

- To start for first time:

```
docker-compose up
```

- After the first time, however, we can simply use start to start the services:

```
docker-compose start
```

- In case our file has a different name than the default one (docker-compose.yml), we can exploit the -f and –-file flags to specify an alternate file name:

```
docker-compose -f custom-compose-file.yml start
```

- Compose can also run in the background as a daemon when launched with the -d option:

```
docker-compose up -d
```

- Shutdown

```
docker-compose stop
docker-compose down  <- delete everything
```

Tutorial Roadmap

- Storage with Docker Volume
- Storage with Bind Host
- Create custom build web image with code from GitHub
- Dockerfile with build argument
- Dockerfile with environment variable
- Share your image on Docker Hub
- Deploy image on Amazon Elastic Beanstalk
- A Node.js application using Nginx, Docker and Redis
- Portainer – Docker Management
- Dashboard for monitoring dockers
- Voting App with docker-compose

Persistent storage: with docker volume

- Run your docker with docker volume named "myvol_data".

```
docker run --name myweb_v -d -p 8080:80 -v myvol_data:/usr/share/nginx/html mytestweb
```

- If everything is ok, you should be able to access the website: localhost:8080.

Persistent storage: with docker volume

- Try change web content from bash terminal
- Open another terminal and run docker exec command to enter shell of container

```
docker exec -it myweb_v bash
```

- Go to data path of nginx: cd /usr/share/nginx/html

```
apichon@ubuntu:~/Desktop$ docker exec -it myweb_v bash
root@57885fd4aaf5:/# cd /usr/share/nginx/html
root@57885fd4aaf5:/usr/share/nginx/html# ls
50x.html  index.html
root@57885fd4aaf5:/usr/share/nginx/html#
```

- Edit index.html by adding school information but your don’t have text editor

```
<!DOCTYPE html>
<html>
    <body>
        <h1>My Name: Your name</h1>
        <p>Id: Your ID</p>
        <p>My School: SIIT </p>
    </body>
</html>
```

Persistent storage: with docker volume

- Since we did not include any text editor in the container, we need to install text editor
- In the container shell, run apt-get command to install nano editor

```
apt-get update
apt-get install nano

# then you can edit the file
nano  index.html
```

- Modify, save and refresh the browser

Persistent storage: with docker volume

- Let try another way by copying local file to volume in the container
- Create testcopy.html in the local directory (adding page title)

```
<!DOCTYPE html>
<html>
<head><title>DES424</title></head>
    <body>
        <h1>My Name: Your name</h1>
        <p>Id: Your ID</p>
        <p>My School: SIIT</p>
    </body>
</html>
```

- Copy file to container and test access via browser (myweb_v is container name)

docker cp /home/apichon/Desktop/testcopy.html **myweb_v:**/usr/share/nginx/html

```
apichon@ubuntu:~/Desktop$ docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED          STATUS          PORTS                                   NAMES
06aef91417bd   mytestweb   "/docker-entrypoint.…"   12 minutes ago   Up 12 minutes   0.0.0.0:8080->80/tcp, :::8080->80/tcp   myweb_v
apichon@ubuntu:~/Desktop$ pwd
/home/apichon/Desktop
apichon@ubuntu:~/Desktop$ docker cp /home/apichon/Desktop/testcopy.html myweb_v:/usr/share/nginx/html
```

Next Step: run new instance with existing volume

- Stop and remove the previous container
- Check if your volume still exist

```
docker stop myweb_v
docker rm myweb_v
docker volume ls
```

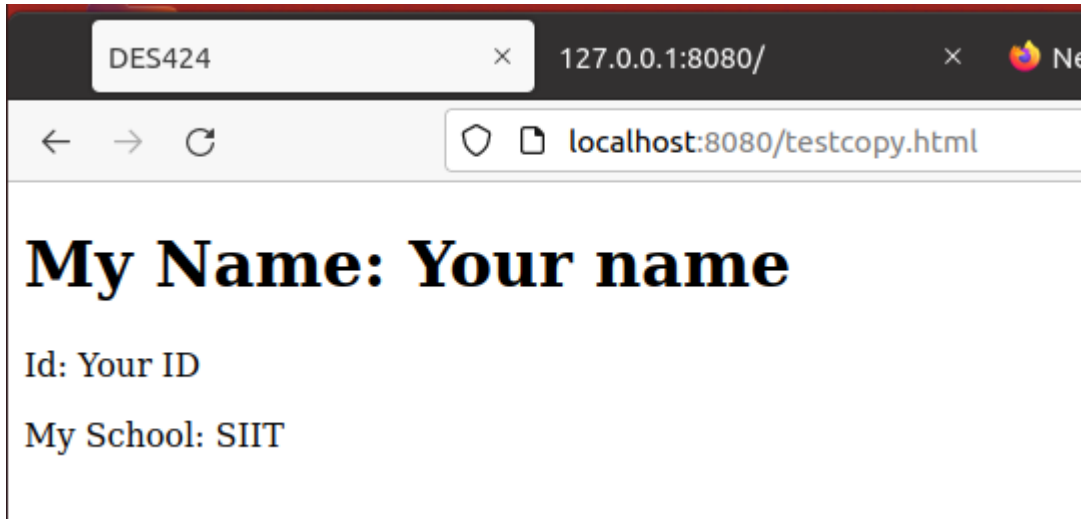
```
apichon@ubuntu:~/Desktop$ docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED          STATUS          PORTS                                       NAMES
57885fd4aaf5   mytestweb   "/docker-entrypoint.…"   33 minutes ago   Up 33 minutes   0.0.0.0:8080->80/tcp, :::8080->80/tcp   myweb_v
apichon@ubuntu:~/Desktop$ docker stop myweb_v
myweb_v
apichon@ubuntu:~/Desktop$ docker rm myweb_v
myweb_v
apichon@ubuntu:~/Desktop$ docker volume ls
DRIVER    VOLUME NAME
local     myvol_data
```

Next Step: run new instance with existing volume

- Run new container instance with the existing volume [myvol_data] and see if your modified content still exist

```
docker run --name myweb_v -d -p 8080:80 -v myvol_data:/usr/share/nginx/html mytestweb
```

- Test access to testcopy.html

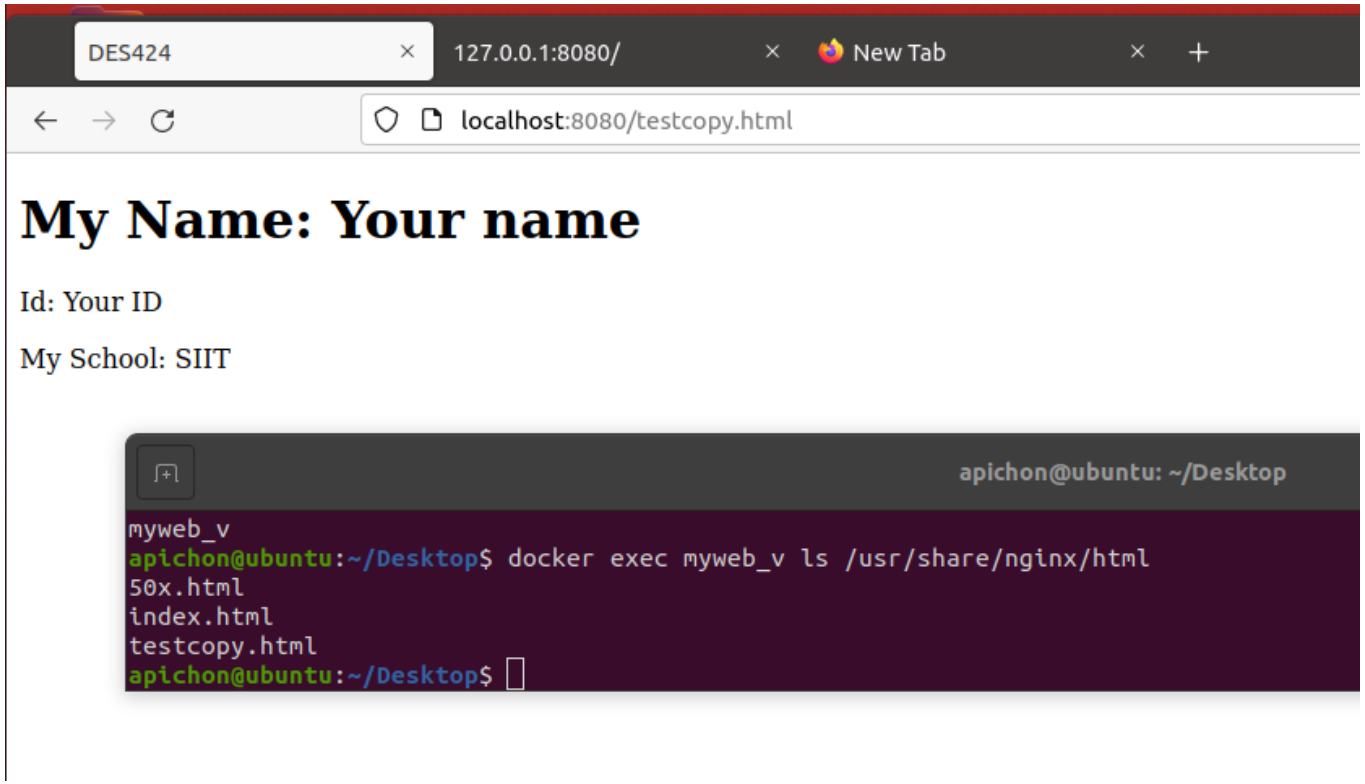


Tutorial check point : 1

- In your container, list files in /usr/share/nginx/html

```
docker exec myweb_v ls /usr/share/nginx/html
```

- Capture screen with terminal and browser and paste it on classroom assignment



Make sure to include both browser and terminal

Persistent storage: with bind mount

- You will create local directory and map that directory with a path in the container
- Create a directory named "my_local_vol" and create new file: index.html with the following content:

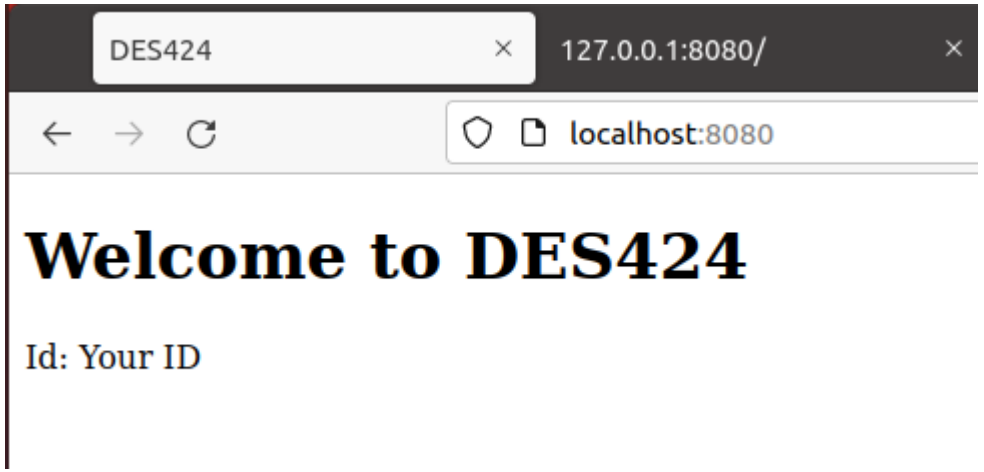
```
<!DOCTYPE html>
<html>
<head><title>DES424</title></head>
    <body>
        <h1>Welcome to DES424</h1>
        <p>Id: Your ID</p>
      </body>
</html>
```

- Run your docker image with mapping path to host: my_local_vol

```
docker run --name myweb_h -d -p 8080:80 -v /home/apichon/Desktop/my_local_vol:/usr/share/nginx/html mytestweb
```

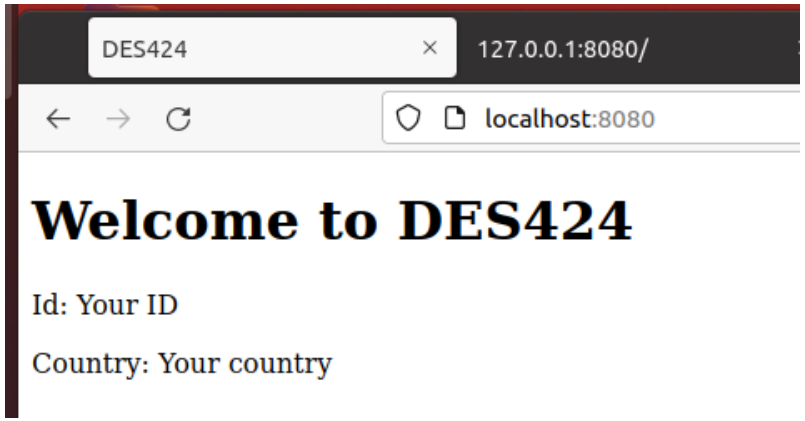
Persistent storage: with bind mount

- Open browser with localhost:8080, you should see the content from your host file



- Let edit index.html by adding your country in your host file (my_local_vol)

```
<!DOCTYPE html>
<html>
<head><title>DES424</title></head>
    <body>
        <h1>Welcome to DES424</h1>
        <p>Id: Your ID</p>
        <p>Country: Your country</p>
      </body>
</html>
```

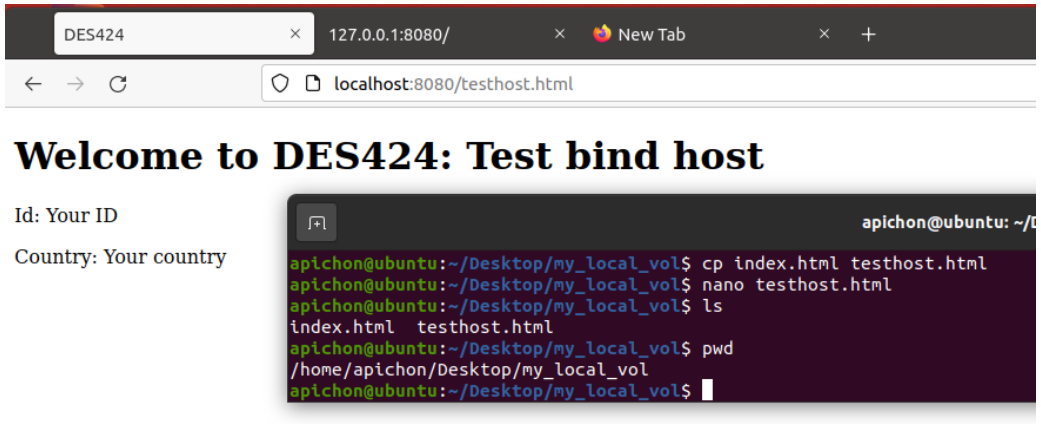


Tutorial check point : 2

- Create new file: testhost.html in your host mapping directory (my_local_vol)

```
<!DOCTYPE html>
<html>
<head><title>DES424</title></head>
    <body>
        <h1>Welcome to DES424: Test bind host</h1>
        <p>Id: Your ID</p>
        <p>Country: Your country</p>
      </body>
</html>
```

- List files in directory and show the page on browser
- Capture screen with terminal and browser and paste it on classroom assignment



Dockerfile: build with code from GitHub

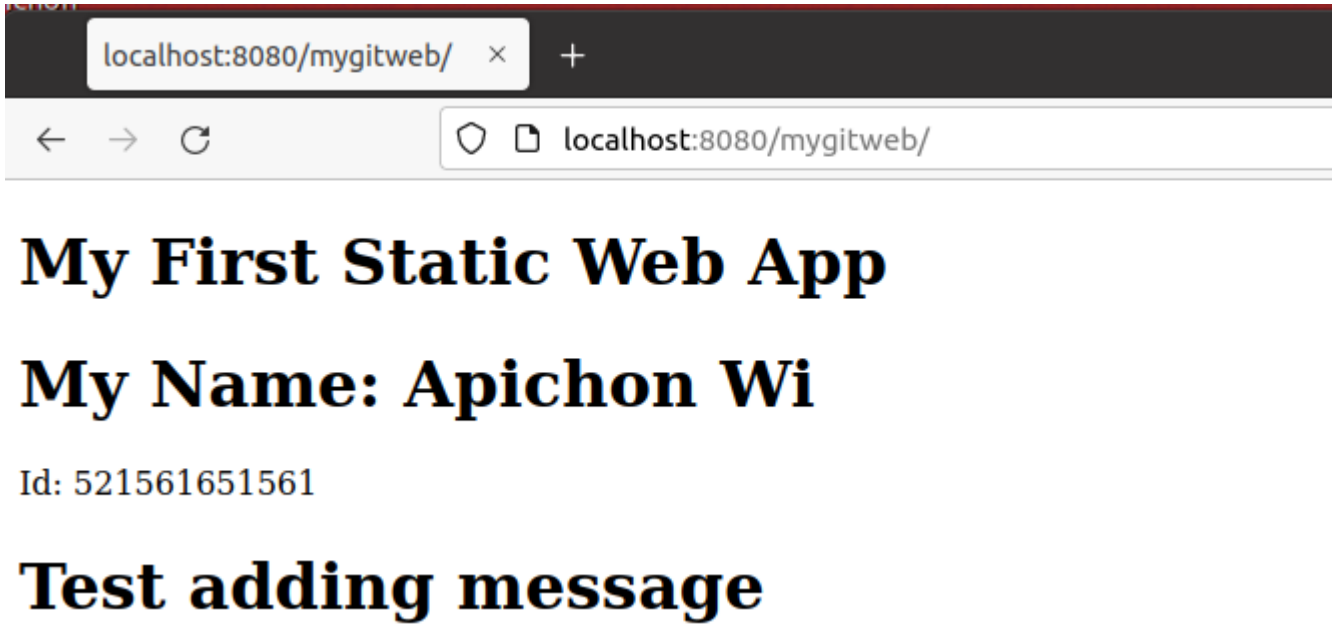
- We will build a custom image for web server (nginx) with the code from GitHub
- Create directory name my_git_docker and enter that directory.
- Create a file named "Dockerfile" (no extension) with the following content and save it.

```
FROM nginx
LABEL MAINTAINER "Your name"

RUN apt-get update && apt-get install -y git
RUN mkdir /mycode &&  cd /mycode &&  git clone https://github.com/apichonsiit/testgit.git
RUN cp -r /mycode/testgit/src /usr/share/nginx/html/mygitweb
```

- testgit is your project did in previous assignment
- Run command to build image: docker build -t mygitweb .

Dockerfile: build with code from GitHub

- Run your image: docker run --name myweb_g -d -p 8080:80 mygitweb
- Check your web on browser



Tutorial check point : 3

- In your git homework project, add logo image to index.html in /src and change your name to use argument instead.

```
<!DOCTYPE html>
<html><body>
        <h1>My Name: <MY_NAME></h1>
        <p>Id: <MY_ID></p>
        <img src="https://graduateadmissions.siit.tu.ac.th/plugin/img/siitlogo.png" alt="">
</body></html>
```

- <MY_NAME> and <MY_ID> will be replaced with argument passing when running container
- New Dockerfile

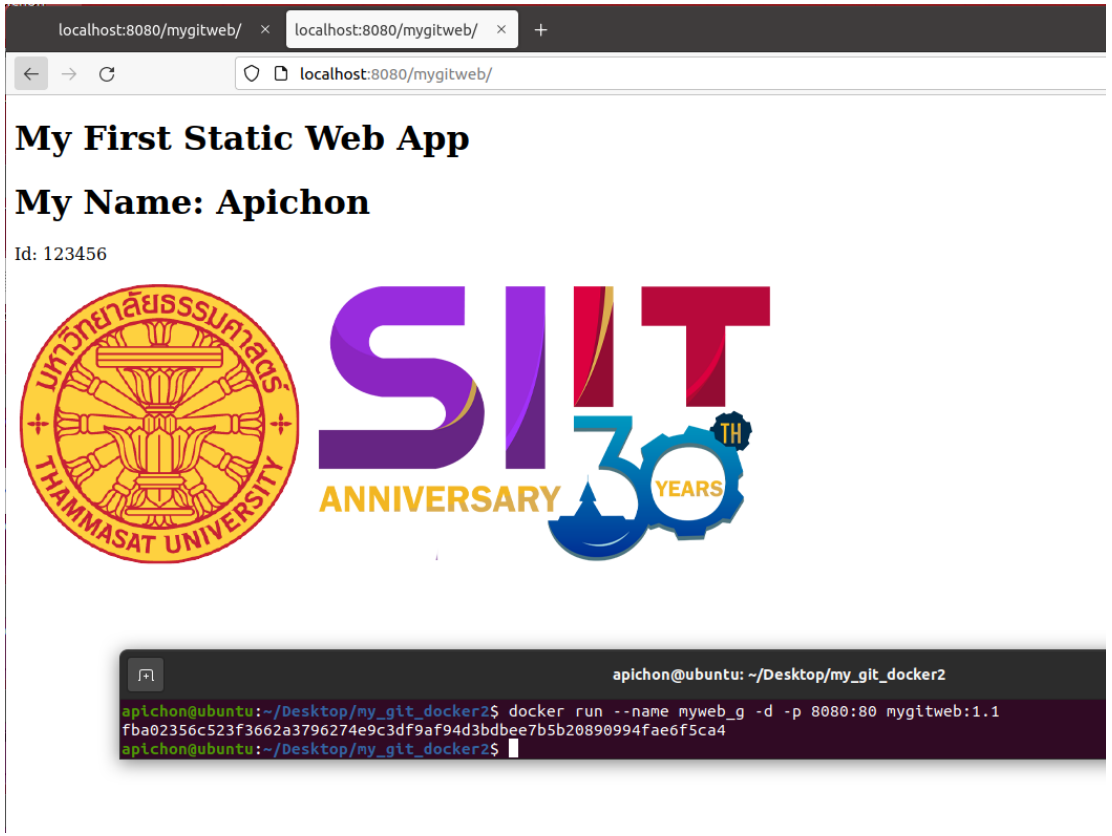
```
FROM nginx
LABEL MAINTAINER "Your name"
ARG MY_NAME=change my name
ARG MY_ID=change my id
RUN apt-get update && apt-get install -y git sed
RUN mkdir /mycode &&  cd /mycode &&  git clone https://github.com/apichonsiit/testgit.git
RUN cp -r /mycode/testgit/src /usr/share/nginx/html/mygitweb
RUN export MY_NAME=$MY_NAME && sed -i 's/<MY_NAME>/'"${MY_NAME}"'/gi' /usr/share/nginx/html/mygitweb/index.html
RUN export MY_ID=$MY_ID && sed -i 's/<MY_ID>/'"${MY_ID}"'/gi' /usr/share/nginx/html/mygitweb/index.html
```

Tutorial check point : 3

- Run command to build image:

```
docker build --no-cache --build-arg MY_NAME=Apichon --build-arg MY_ID=123456 -t mygitweb:1.1 .
```

- Capture screen with terminal and browser and paste it on classroom assignment



Dockerfile: passing environment value

- We will build a custom image and accept custom environment value at runtime.
- Create new file on your project: check4.html

```
<!DOCTYPE html>
<html>
<body>
        <h1>Welcome to DES424: Check Point 4</h1>
        <p>My Name: Your Name</p>
        <p>My ID: Your ID</p>
        <p>Host: <MY_HOST></p>
      </body>
</html>
```

- Create new Dockerfile

```
FROM nginx
LABEL MAINTAINER "Your name"
ENV MY_HOST localhost
RUN apt-get update && apt-get install -y git sed
RUN mkdir /mycode &&  cd /mycode &&  git clone https://github.com/apichonsiit/testgit.git
RUN cp -r /mycode/testgit/src /usr/share/nginx/html/mygitweb
CMD sed -i 's/< MY_HOST >/'"$MY_HOST"'/gi' /usr/share/nginx/html/mygitweb/check4.html
```

Dockerfile: passing environment value

- We will build a custom image and accept custom environment value at runtime.
- Create new file on your project: check4.html

```
<!DOCTYPE html>
<html>
<body>
        <h1>Welcome to DES424: Check Point 4</h1>
        <p>My Name: Your Name</p>
        <p>My ID: Your ID</p>
        <p>Host: <MY_HOST></p>
      </body>
</html>
```

- We need to run sed to replace text at runtime, hence, we need to use CMD but as inspected the image, it CMD was used for start nginx daemon. We will then need to create a shell script to run those two command.
- Create dotask.sh with the following scripts:

```
sed -i 's/<MY_HOST>/'"$MY_HOST"'/gi' /usr/share/nginx/html/mygitweb/check4.html
nginx -g 'daemon off;'
```

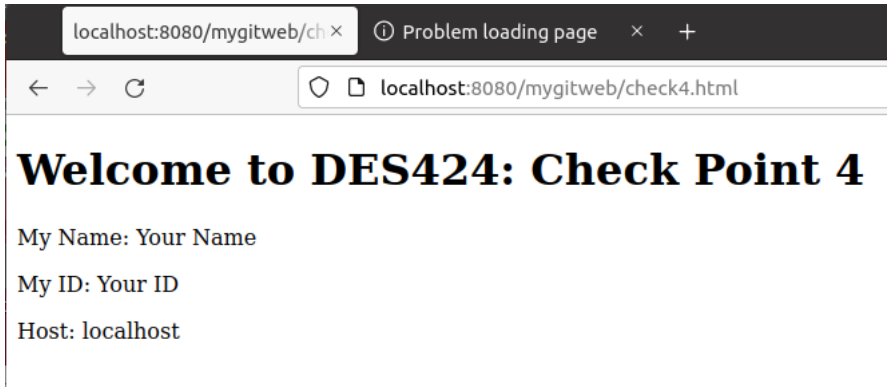
Dockerfile: passing environment value

- Create new Dockerfile

```
FROM nginx
LABEL MAINTAINER "Your name"
ENV MY_HOST localhost
COPY dotask.sh /
RUN apt-get update && apt-get install -y git sed
RUN mkdir /mycode &&  cd /mycode &&  git clone https://github.com/apichonsiit/testgit.git
RUN cp -r /mycode/testgit/src /usr/share/nginx/html/mygitweb
CMD ["sh","/dotask.sh"]
```

- Build Dockerfile with tag: 1.4 and run container

```
docker build --no-cache -t mygitweb:1.4
docker run -d --name myweb_g -p 8080:80 mygitweb:1.4
```



Tutorial check point : 4

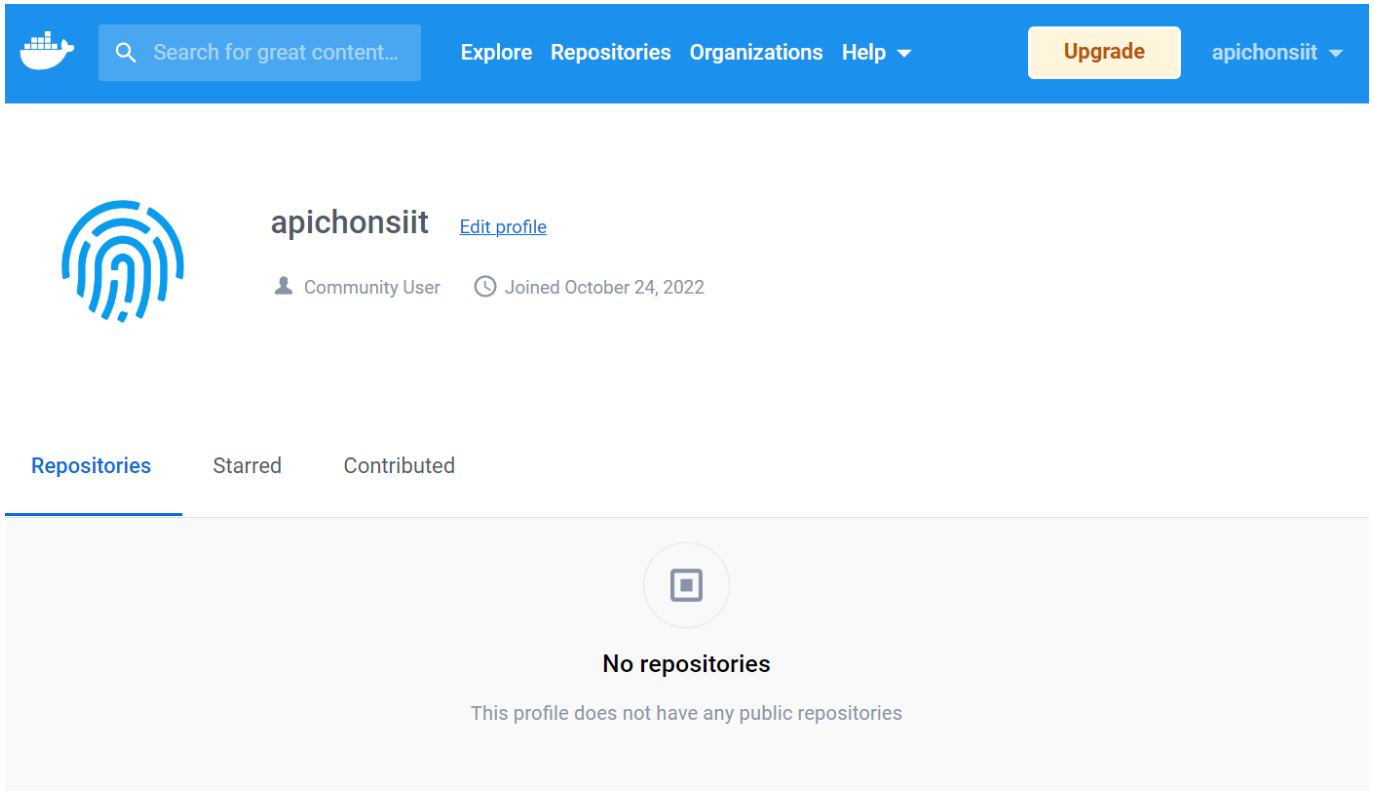
- Run container with passing MY_HOST:

```
docker run -d --name myweb_g -e MY_HOST=MyDockerOnVM -p 8080:80 mygitweb:1.4
```

- Capture screen with terminal and browser and paste it on classroom assignment

Push your custom image to Docker Hub

- Create an account on docker hub (hub.docker.com)
- During the registration, select personal plan



Push your custom image to Docker Hub

- In terminal, login to docker hub via command line (change username to yours)

```
docker login --username=apichonsiit
```

- It will prompt for password.

```
apichon@ubuntu:~/Desktop/check4$ docker login --username=apichonsiit
Password:
WARNING! Your password will be stored unencrypted in /home/apichon/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
apichon@ubuntu:~/Desktop/check4$
```

- Once login succeeded, try build and image again by specify docker hub registry (use same image with check point 4)

```
docker build --no-cache -t apichonsiit/mygitweb:1.4 .
```

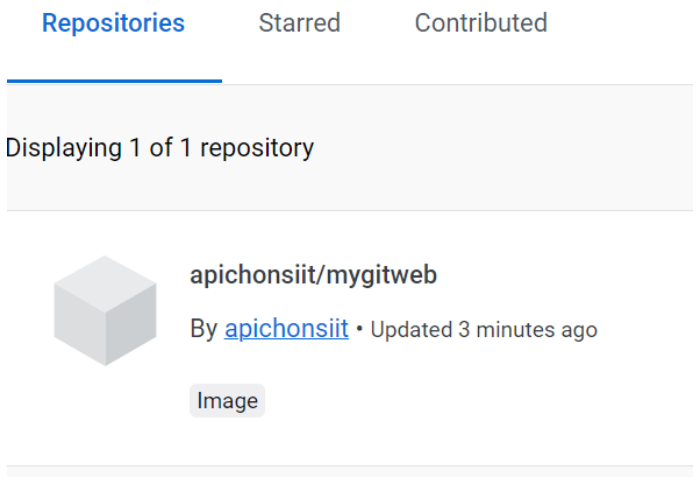
Push your custom image to Docker Hub

- Run push command to upload image to docker hub (change username to yours)

```
docker push apichonsiit/mygitweb:1.4
```

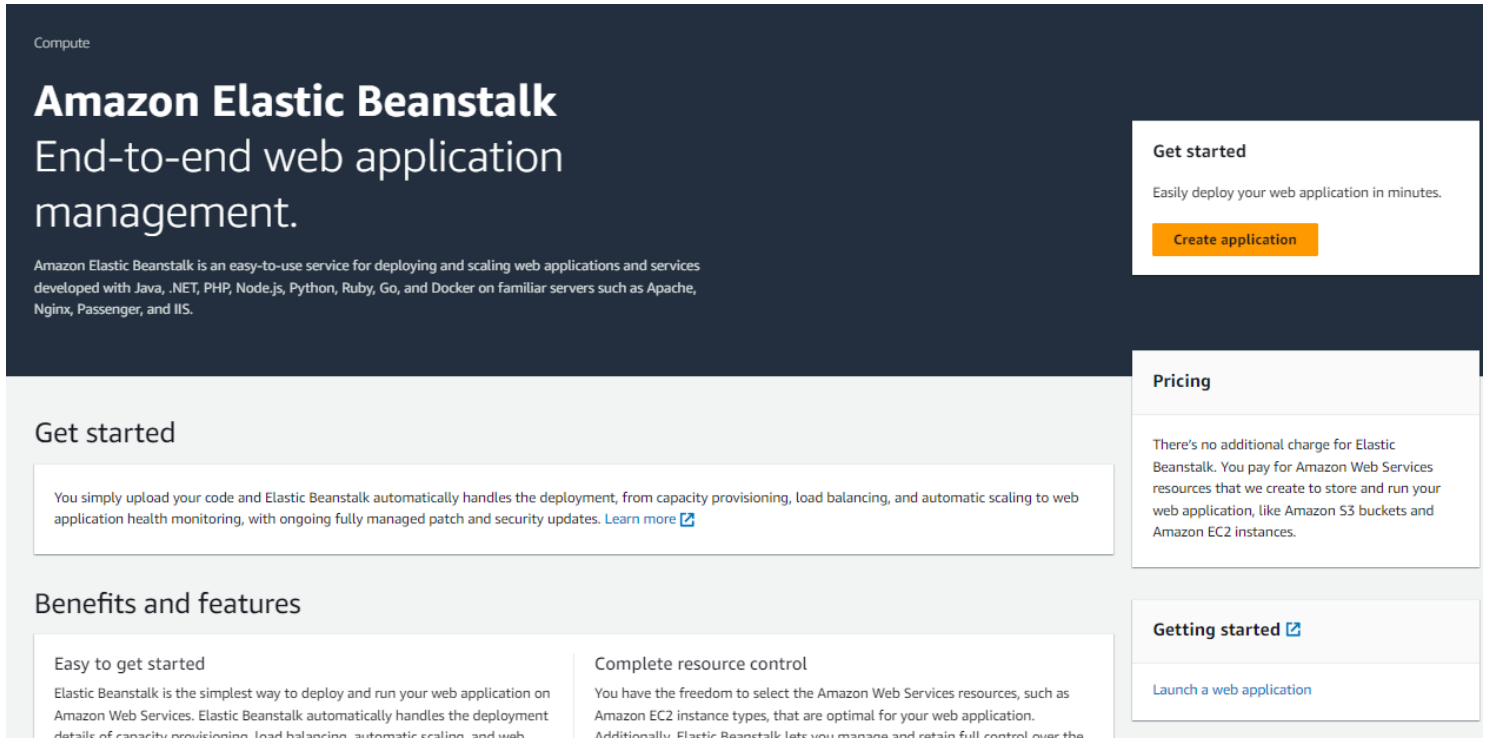
```
apichon@ubuntu:~/Desktop/check4$ docker push apichonsiit/mygitweb:1.4
The push refers to repository [docker.io/apichonsiit/mygitweb]
c3d6fd7bb675: Pushed
e07b3b16ca1e: Pushed
c9a73ac26ba7: Pushed
f689d8732adb: Pushed
d6a3537fc36a: Mounted from library/nginx
819eb3a45632: Mounted from library/nginx
5eda6fa69be4: Mounted from library/nginx
6f4f3ce1dca0: Mounted from library/nginx
58a06a0d345c: Mounted from library/nginx
fe7b1e9bf792: Mounted from library/nginx
1.4: digest: sha256:7237bc0c17c666a3d72512cfc1ad01843cffaf6f59ef30dec5ed05380b2cf720 size: 2405
```

- You can check your image (repository on Docker Hub)



Deploy image on Amazon Elastic Beanstalk

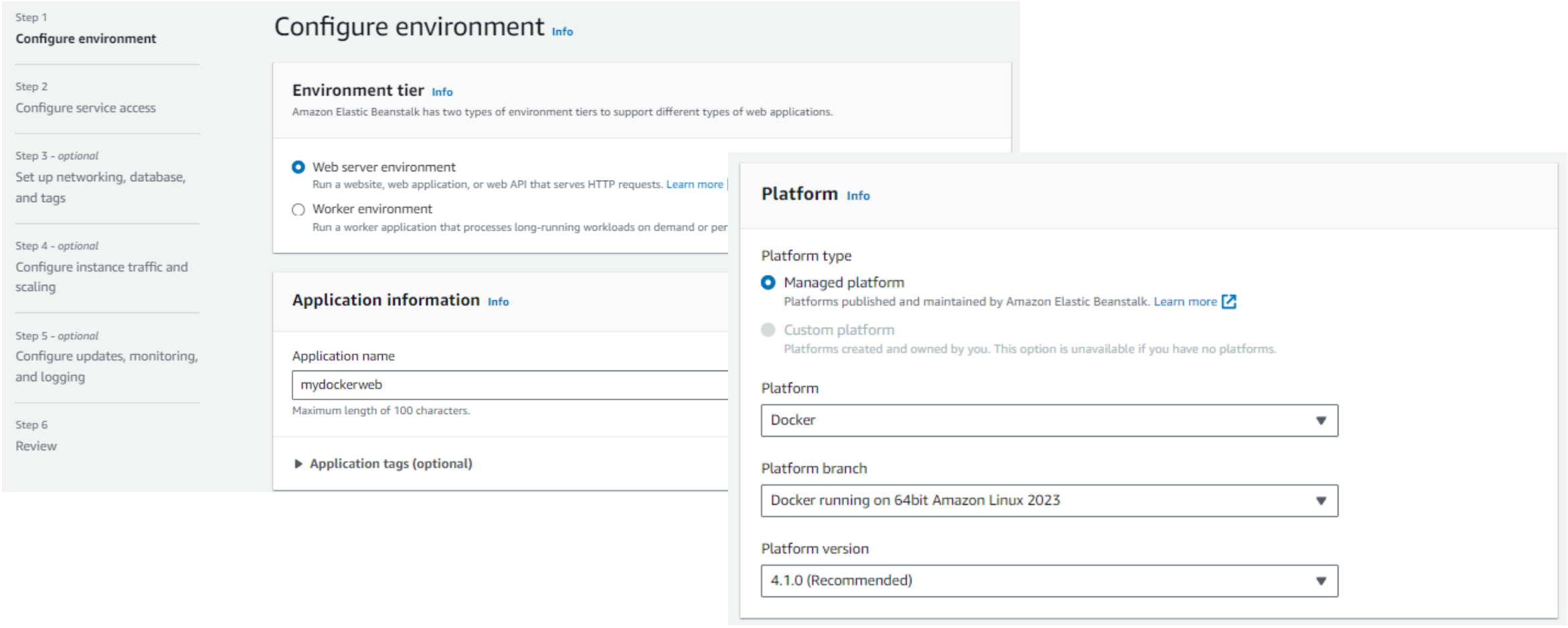
- Go to AWS Academy lab learner and service for Amazon Elastic Beanstalk



- Click on create application

Deploy image on Amazon Elastic Beanstalk

- Environment: Web server
- Define application name: mydockerweb
- Select Platform



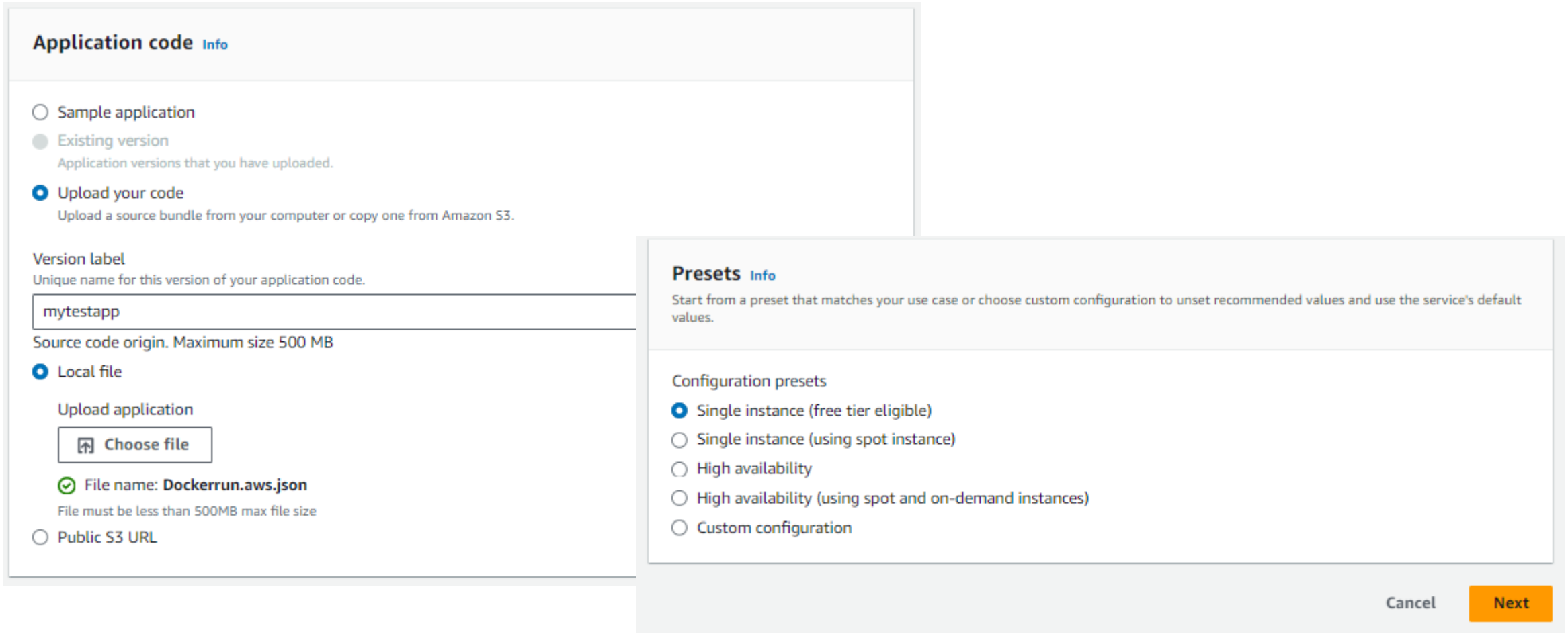
Deploy image on Amazon Elastic Beanstalk

- Create Dockerrun.aws.json
- Change to your docker image

```
{
  "AWSEBDockerrunVersion":"1",
  "Image":{
    "Name":"apichonsiit/mygitweb:1.4",
    "Update": "true"
  },

  "Ports":[
    {
      "ContainerPort":80,
      "HostPort":80
    }
  ]
}
```

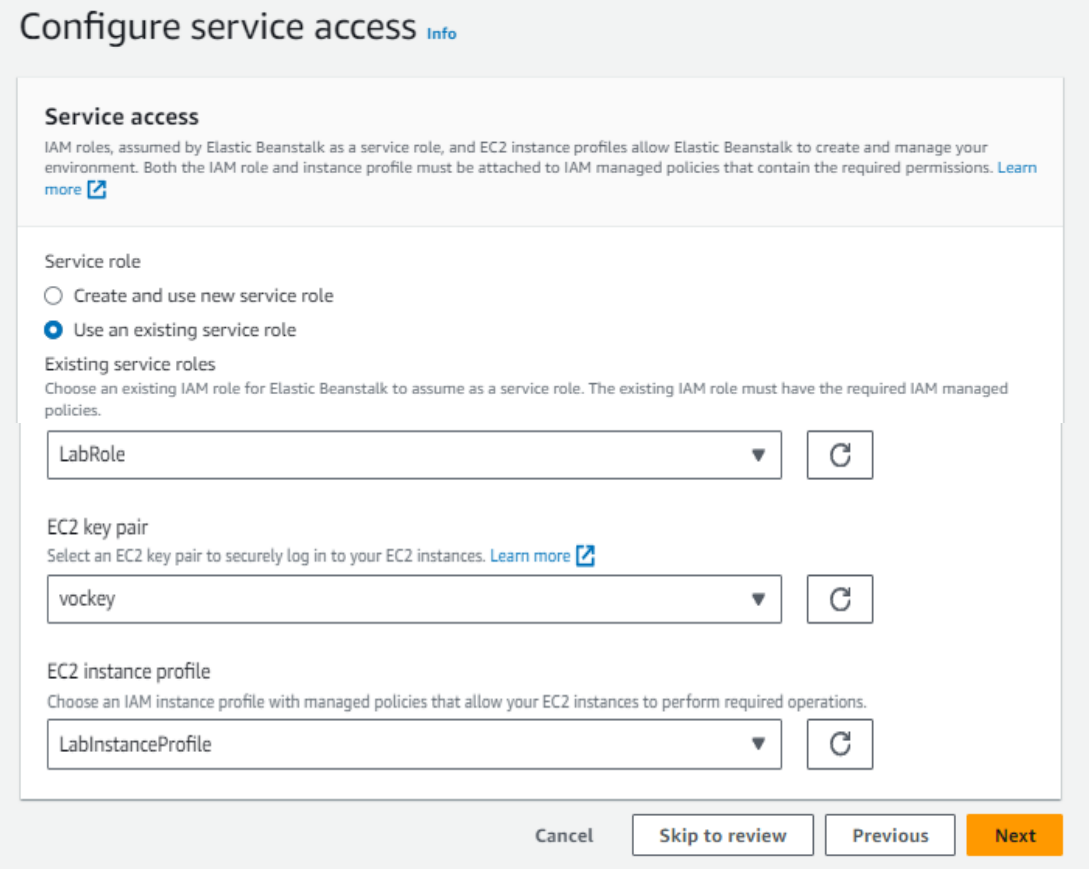
Deploy image on Amazon Elastic Beanstalk

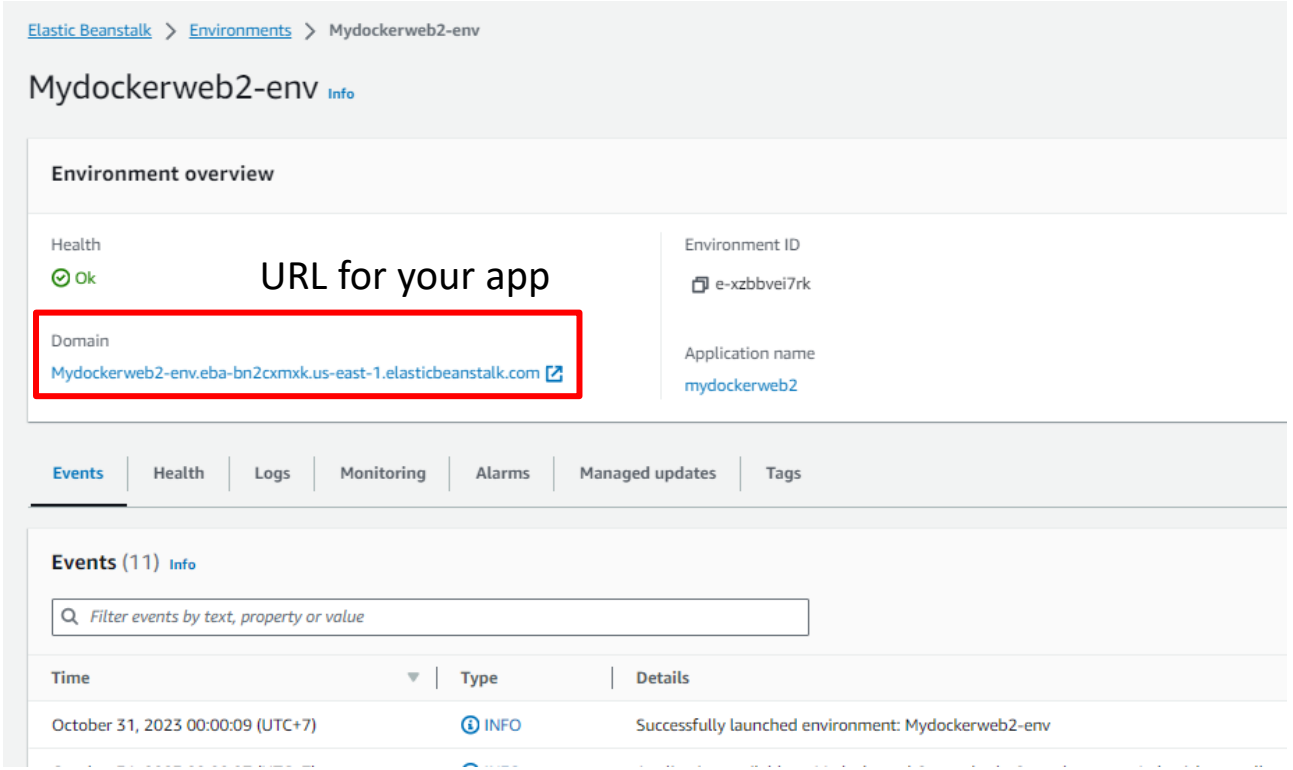
- Upload file created the previous slide
- Select single instance in Present and process with Next.



Deploy image on Amazon Elastic Beanstalk

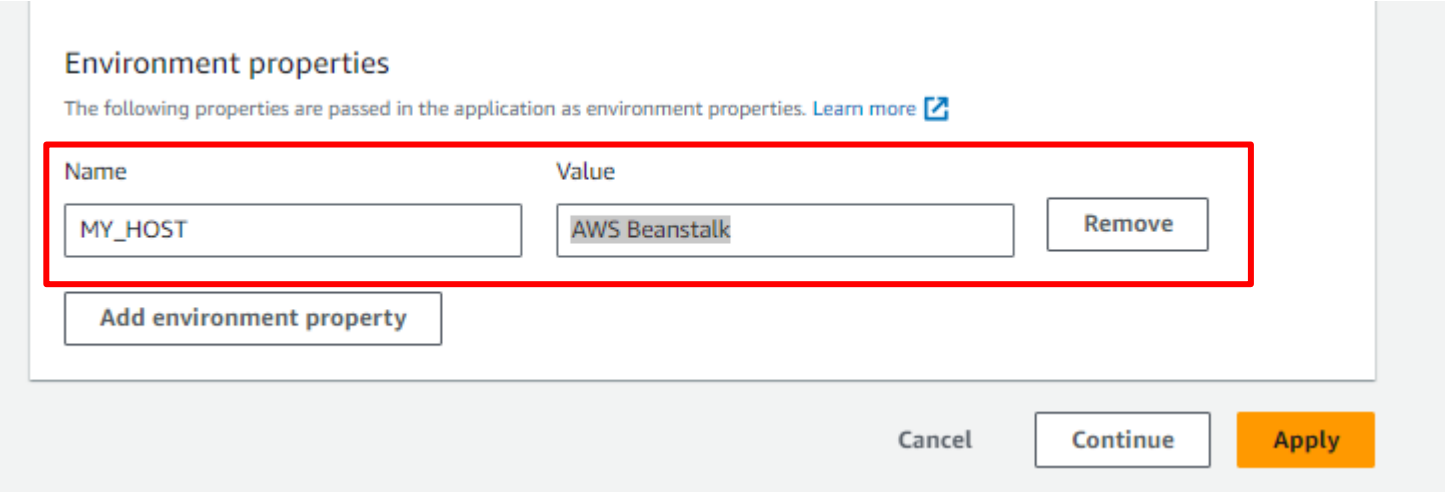
- Config Service Access, Skip to Review and submit
- Wait until deploy complete and health show ok
- Click link to access web





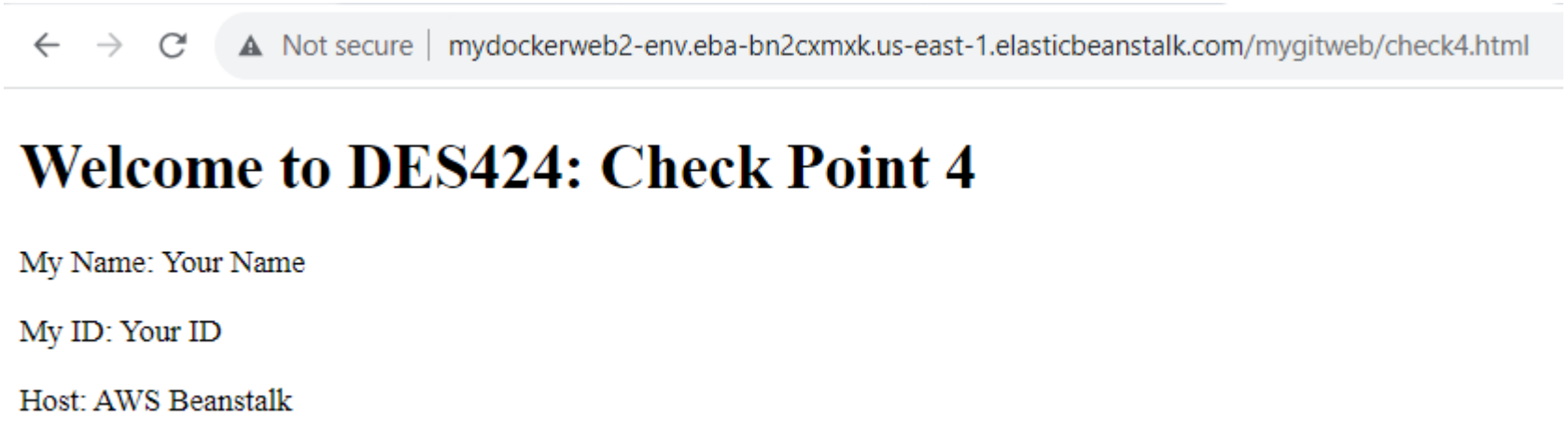
Deploy image on Amazon Elastic Beanstalk

- Access to app portal under configuration -> Environment Properties
- We will add environment variable: MY_HOST = AWS Beanstalk
- Then apply



Tutorial check point : 5

- Access to your container app
- Capture screen with terminal and browser and paste it on classroom assignment



Tutorial Roadmap

- Storage with Docker Volume
- Storage with Bind Host
- Create custom build web image with code from GitHub
- Dockerfile with build argument
- Dockerfile with environment variable
- Share your image on Docker Hub
- Deploy image on Azure App Service

- A Node.js application using Nginx, Docker and Redis
- Portainer – Docker Management
- Dashboard for monitoring dockers
- Voting App with docker-compose

Install docker-compose

- Install binary file of docker-compose
- change permission of the file
- Check if it is working

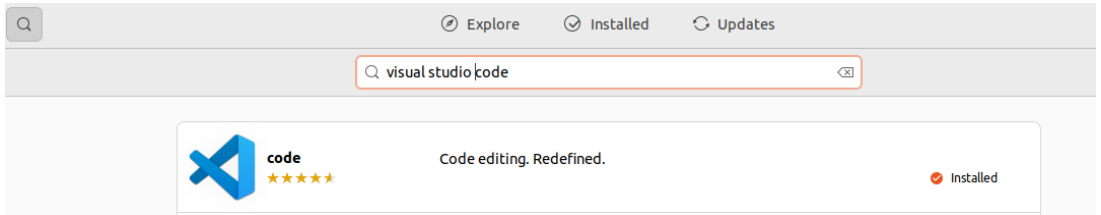
```
sudo curl -SL https://github.com/docker/compose/releases/download/v2.12.2/docker-compose-linux-x86_64 -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
docker-compose --version
```

```
apichon@ubuntu:~$ sudo curl -SL https://github.com/docker/compose/releases/download/v2.12.2/docker-compose-linux-x86_64 -o /usr/loc
al/bin/docker-compose
[sudo] password for apichon:
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 42.8M  100 42.8M    0     0  3240k      0  0:00:13  0:00:13 --:--:-- 2633k
```

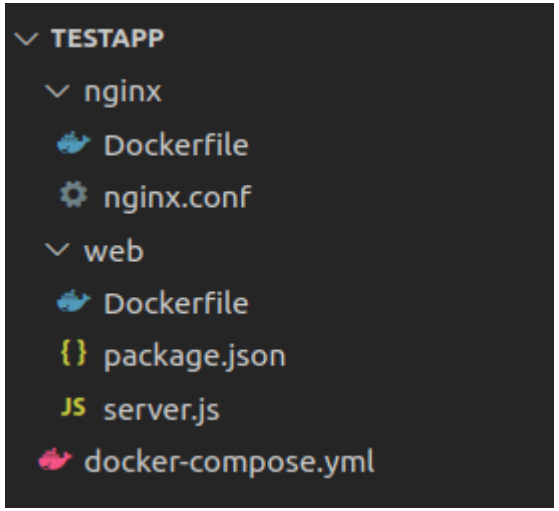
```
apichon@ubuntu:~$ sudo chmod +x /usr/local/bin/docker-compose
apichon@ubuntu:~$ docker-compose --version
Docker Compose version v2.12.2
```

A Node.js application using Nginx, Docker and Redis

- You will see how to containerize a NodeJS web application with Redis database and Nginx as a reverse proxy in front of NodeJS app using Docker. We are going to build the following project structure
- You may install Vs code on Ubuntu.



Project structure



- Create project structure similar to the figure.

https://dev.to/docker/dockerizing-a-nodejs-express-redis-app-with-nginx-proxy-using-docker-compose-17ge

Step1: Create a Docker compose file

- Create an empty file with the below content and save it by name - "docker-compose.yml"
- The compose file defines an application with four services redis, nginx, web1 and web2.
- When deploying the application, docker-compose maps port 80 of the web service container to port 80 of the host as specified in the file.

docker-compose.yml

```yaml
services:
  redis:
    image: 'redislabs/redismod'
    ports:
      - '6379:6379'
  web1:
    restart: on-failure
    build: ./web
    hostname: web1
    ports:
      - '81:5000'
  web2:
    restart: on-failure
    build: ./web
    hostname: web2
    ports:
      - '82:5000'
  nginx:
    build: ./nginx
    ports:
    - '80:80'
    depends_on:
    - web1
    - web2
```

https://dev.to/docker/dockerizing-a-nodejs-express-redis-app-with-nginx-proxy-using-docker-compose-17ge

Step 2. Create nginx directory and add the below files:

- Under nginx folder, create nginx.conf and Dockerfile

nginx > nginx.conf

```
upstream loadbalancer {
  server web1:5000;
  server web2:5000;
}

server {
  listen 80;
  server_name localhost;
  location / {
    proxy_pass http://loadbalancer;
  }
}
```

nginx > Dockerfile > ...

```
FROM nginx:1.21.6
RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

https://dev.to/docker/dockerizing-a-nodejs-express-redis-app-with-nginx-proxy-using-docker-compose-17ge

Step 3. Create a web directory and add the below files:

- Under web folder, create package.json, server.js and Dockerfile

web > server.js > ...

```
const os = require('os');
const express = require('express');
const app = express();
const redis = require('redis');
const redisClient = redis.createClient({
  host: 'redis',
  port: 6379
});

app.get('/', function(req, res) {
    redisClient.get('numVisits', function(err, numVisits) {
        numVisitsToDisplay = parseInt(numVisits) + 1;
        if (isNaN(numVisitsToDisplay)) {
            numVisitsToDisplay = 1;
        }
        res.send(os.hostname() +': Number of visits is: ' + numVisitsToDisplay);
        numVisits++;
        redisClient.set('numVisits', numVisits);
    });
});

app.listen(5000, function() {
    console.log('Web application is listening on port 5000');
});
```

web > package.json > ...

```
{
  "name": "web",
  "version": "1.0.0",
  "description": "Running Node.js and Express.js on Docker",
  "main": "server.js",
  ▷ Debug
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.17.2",
    "redis": "3.1.2"
  },
  "author": "",
  "license": "MIT"
}
```

web > Dockerfile > ...

```
FROM node:14.17.3-alpine3.14

WORKDIR /usr/src/app

COPY ./package.json ./
RUN npm install
COPY ./server.js ./

CMD ["npm","start"]

```

https://dev.to/docker/dockerizing-a-nodejs-express-redis-app-with-nginx-proxy-using-docker-compose-17ge

Step 4. Deploy the application

- Deploy the full-fledged app using docker-compose

```
$ docker-compose up -d
```

```
[+] Running 5/5
 ⠿ Network testapp_default     Created
 ⠿ Container testapp-web2-1    Started
 ⠿ Container testapp-redis-1   Started
 ⠿ Container testapp-web1-1    Started
 ⠿ Container testapp-nginx-1   Started
```

```
apichon@ubuntu:~/Desktop/test-compose/testapp$ docker ps
CONTAINER ID   IMAGE              COMMAND                  CREATED          STATUS          PORTS                                                  NAMES
137f47610f4d   testapp-nginx      "/docker-entrypoint.…"   20 seconds ago   Up 17 seconds   0.0.0.0:80->80/tcp, :::80->80/tcp                      testapp-nginx-1
b3afa333c5a6   testapp-web1       "docker-entrypoint.s…"   20 seconds ago   Up 18 seconds   0.0.0.0:81->5000/tcp, :::81->5000/tcp                  testapp-web1-1
bb48f3d554c6   testapp-web2       "docker-entrypoint.s…"   20 seconds ago   Up 18 seconds   0.0.0.0:82->5000/tcp, :::82->5000/tcp                  testapp-web2-1
6fad19e5313a   redislabs/redismod "redis-server --load…"   20 seconds ago   Up 18 seconds   0.0.0.0:6379->6379/tcp, :::6379->6379/tcp              testapp-redis-1
```

https://dev.to/docker/dockerizing-a-nodejs-express-redis-app-with-nginx-proxy-using-docker-compose-17ge

Step 5. Monitoring Redis keys

- Install redis-tools and run redis-cli

```
sudo apt-get install redis-tools
redis-cli
```

```
% redis-cli
127.0.0.1:6379> monitor
OK
1646485507.290868 [0 172.24.0.2:34330] "get" "numVisits"
1646485507.309070 [0 172.24.0.2:34330] "set" "numVisits" "5"
1646485509.228084 [0 172.24.0.2:34330] "get" "numVisits"
1646485509.241762 [0 172.24.0.2:34330] "set" "numVisits" "6"
1646485509.619369 [0 172.24.0.4:52082] "get" "numVisits"
1646485509.629739 [0 172.24.0.4:52082] "set" "numVisits" "7"
1646485509.990926 [0 172.24.0.2:34330] "get" "numVisits"
1646485509.999947 [0 172.24.0.2:34330] "set" "numVisits" "8"
1646485510.270934 [0 172.24.0.4:52082] "get" "numVisits"
1646485510.286785 [0 172.24.0.4:52082] "set" "numVisits" "9"
1646485510.469613 [0 172.24.0.2:34330] "get" "numVisits"
1646485510.480849 [0 172.24.0.2:34330] "set" "numVisits" "10"
1646485510.622615 [0 172.24.0.4:52082] "get" "numVisits"
1646485510.632720 [0 172.24.0.4:52082] "set" "numVisits" "11"
```


https://dev.to/docker/dockerizing-a-nodejs-express-redis-app-with-nginx-proxy-using-docker-compose-17ge

Step 6. Testing the app

- After the application starts, navigate to http://localhost:80 in your web browser or run

```
curl localhost:80
```

```
curl localhost:80
curl localhost:80
web1: Total number of visits is: 1
```

```
curl localhost:80
web1: Total number of visits is: 2
```

```
$ curl localhost:80
web2: Total number of visits is: 3
```

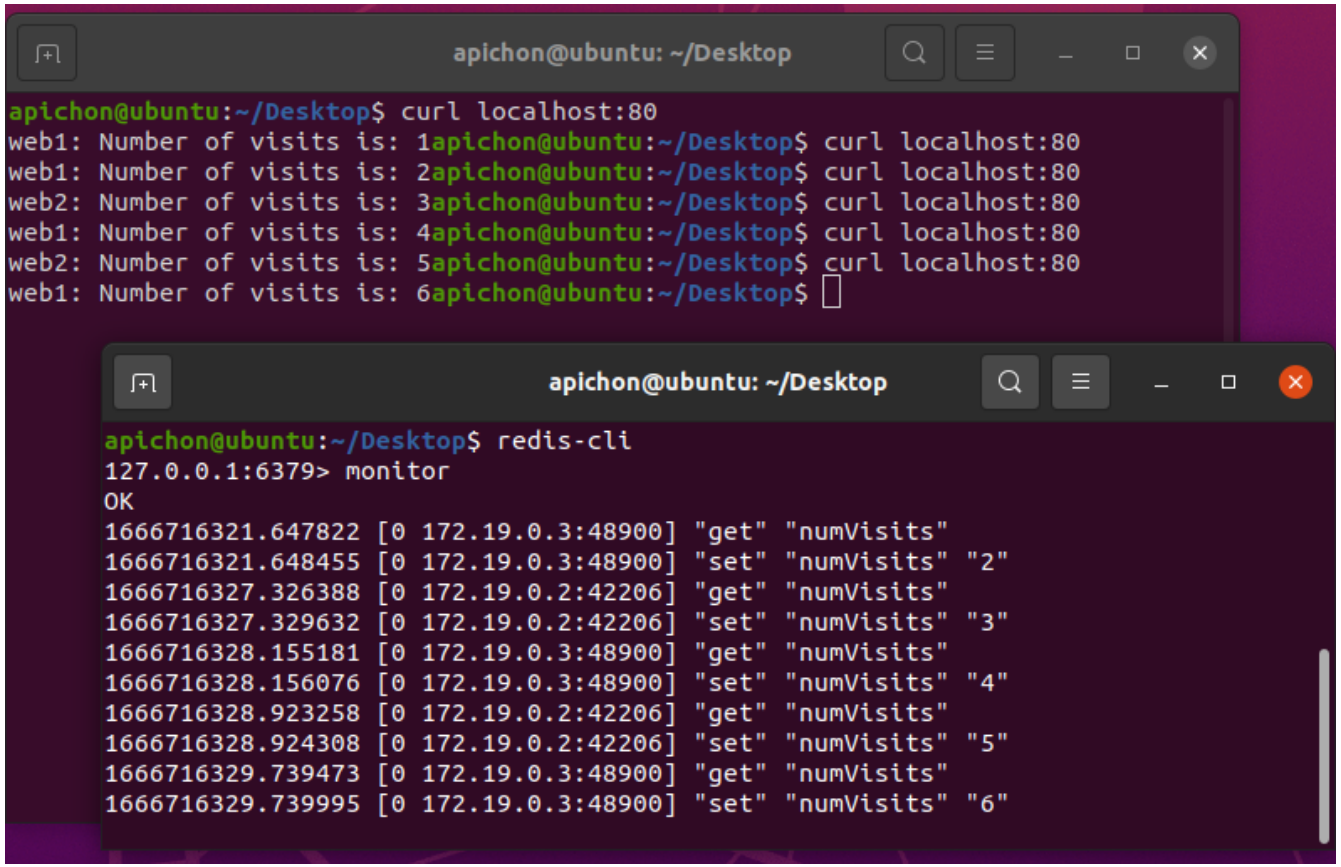
```
$ curl localhost:80
web2: Total number of visits is: 4
```

```
apichon@ubuntu:~/Desktop$ redis-cli
127.0.0.1:6379> monitor
OK
1666716321.647822 [0 172.19.0.3:48900] "get" "numVisits"
1666716321.648455 [0 172.19.0.3:48900] "set" "numVisits" "2"
1666716327.326388 [0 172.19.0.2:42206] "get" "numVisits"
1666716327.329632 [0 172.19.0.2:42206] "set" "numVisits" "3"
1666716328.155181 [0 172.19.0.3:48900] "get" "numVisits"
1666716328.156076 [0 172.19.0.3:48900] "set" "numVisits" "4"
1666716328.923258 [0 172.19.0.2:42206] "get" "numVisits"
1666716328.924308 [0 172.19.0.2:42206] "set" "numVisits" "5"
1666716329.739473 [0 172.19.0.3:48900] "get" "numVisits"
1666716329.739995 [0 172.19.0.3:48900] "set" "numVisits" "6"
```


https://dev.to/docker/dockerizing-a-nodejs-express-redis-app-with-nginx-proxy-using-docker-compose-17ge

Tutorial check point : 6

- Capture screen as following and paste it on classroom assignment



https://dev.to/docker/dockerizing-a-nodejs-express-redis-app-with-nginx-proxy-using-docker-compose-17ge

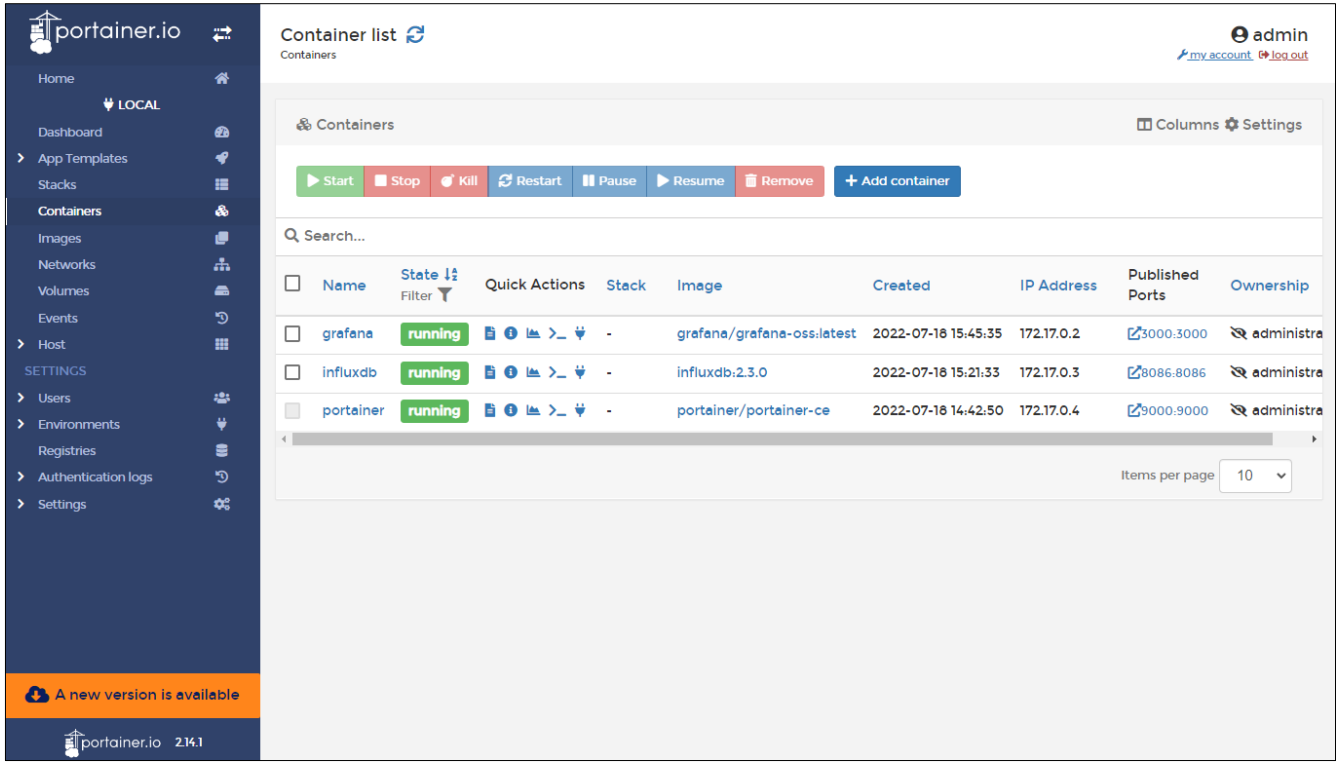
Tutorial Roadmap

- Storage with Docker Volume
- Storage with Bind Host
- Create custom build web image with code from GitHub
- Dockerfile with build argument
- Dockerfile with environment variable
- Share your image on Docker Hub
- Deploy image on Azure App Service
- A Node.js application using Nginx, Docker and Redis
- Portainer – Docker Management
- Dashboard for monitoring dockers
- Voting App with docker-compose

Portainer : GUI-based Container Management

https://www.portainer.io/

- **Portainer** is a popular Docker UI that helps you visualize your containers, images, volumes and networks. Portainer helps you take control of the Docker resources on your machine, avoiding lengthy terminal commands.



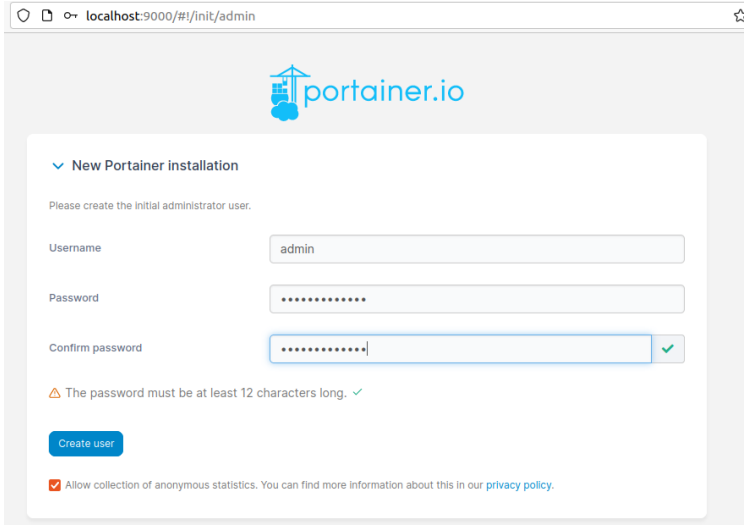
Install Portainer

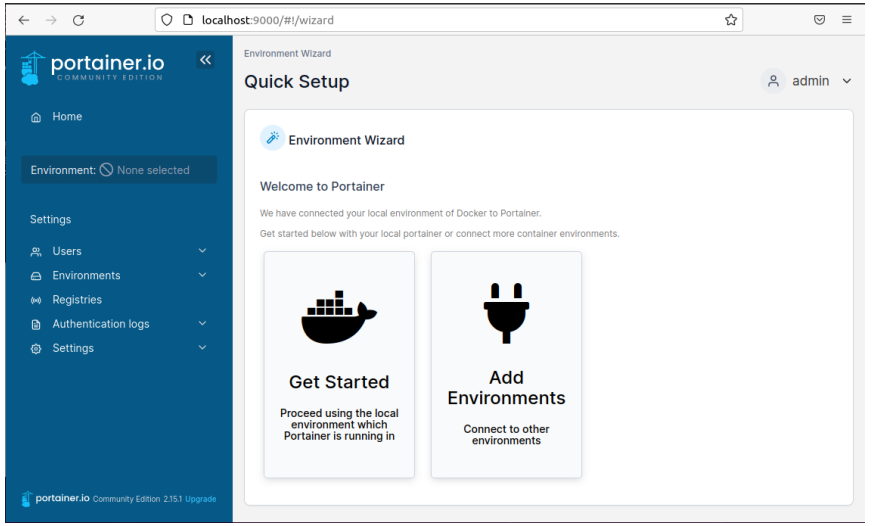
- Create docker volume for Portainer and run the container

```
docker volume create portainer_data
docker run -d -p 9000:9000 --name=portainer --restart=unless-stopped -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer-ce
```

```
apichon@ubuntu:~/Desktop$ docker volume create portainer_data
portainer_data
apichon@ubuntu:~/Desktop$ docker run -d -p 9000:9000 --name=portainer --restart=unless-stopped -v /var/run/docker.soc
k:/var/run/docker.sock -v portainer_data:/data portainer/portainer-ce
```

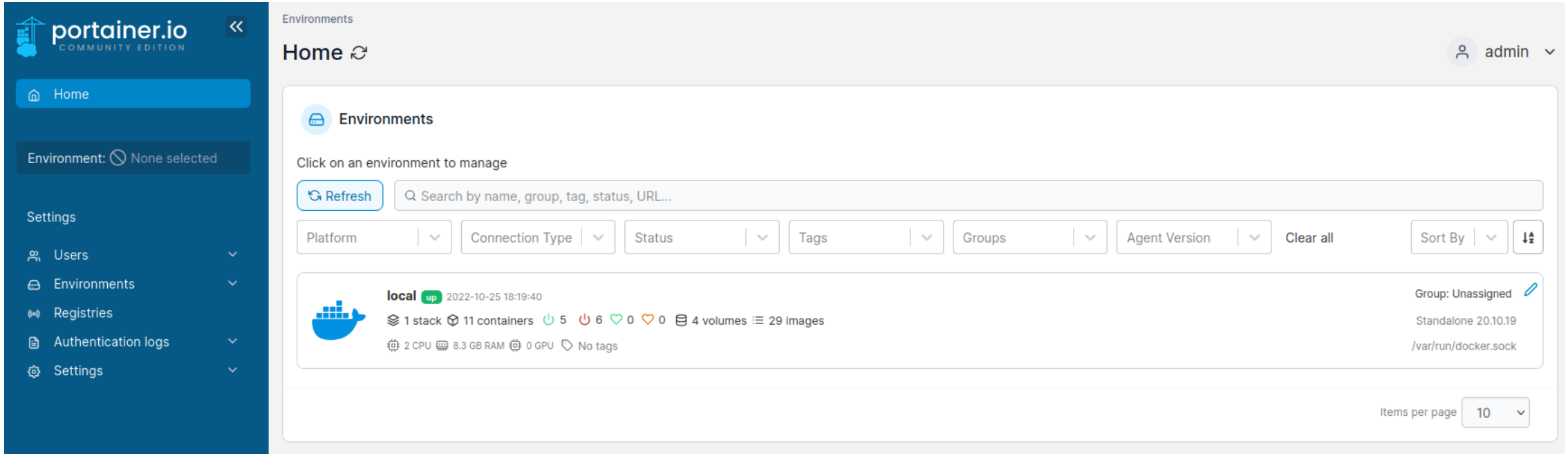
- Then, access to portainer web via localhost:9000
- For the firsts time, you will be asked to create an account





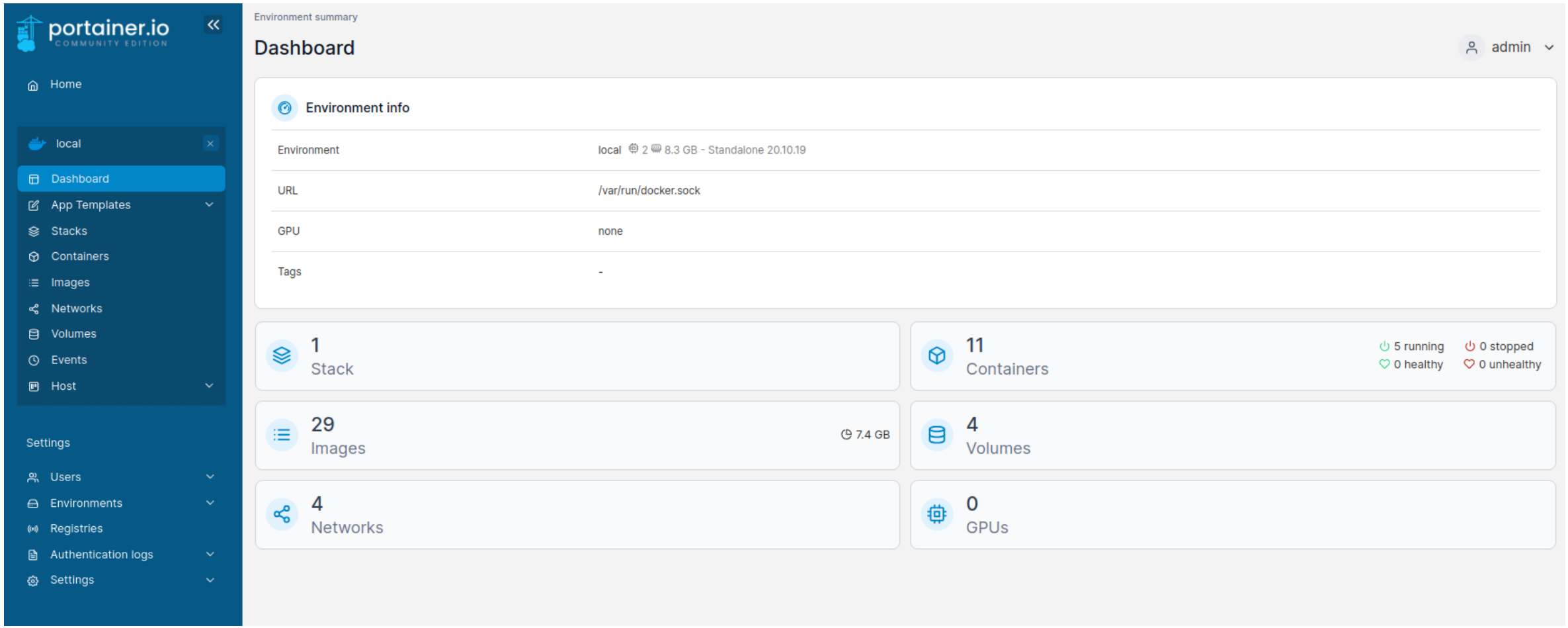
Portainer Web: select docker daemon

- Click on home and select local, that will link to your current local docker daemon.



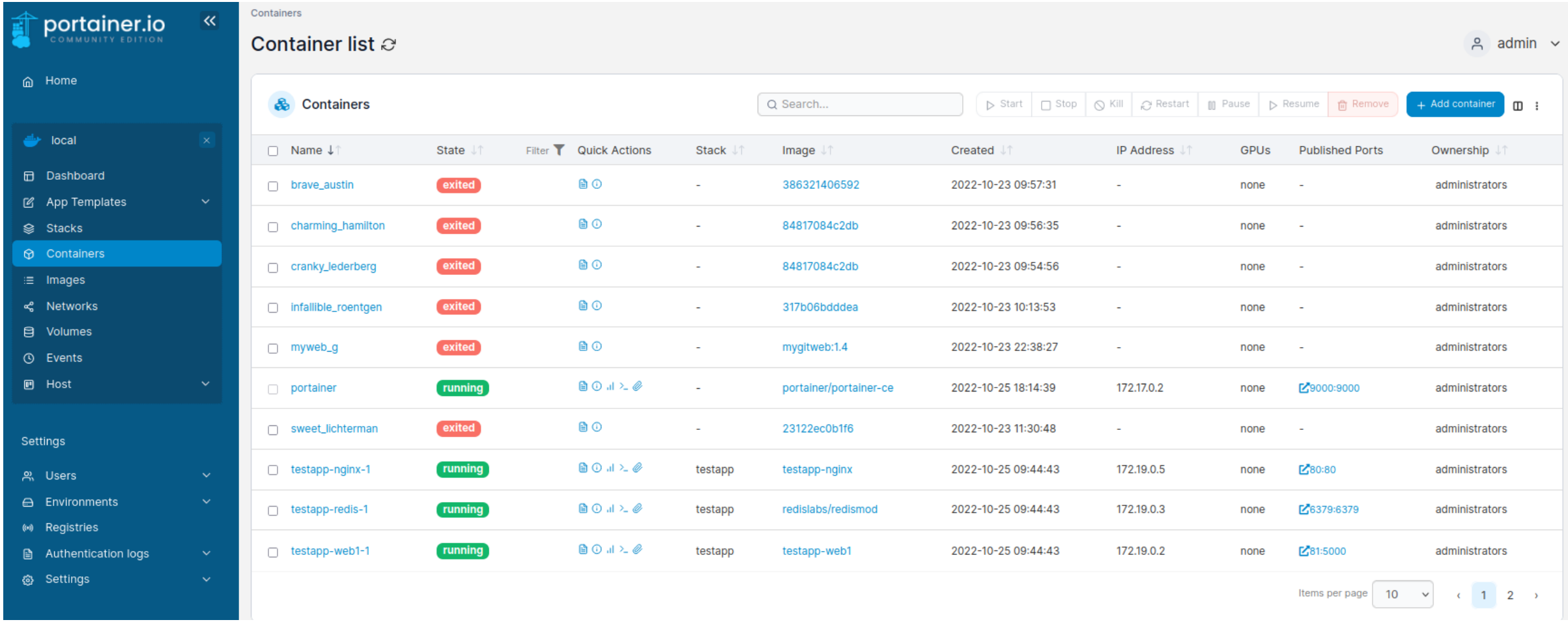
Portainer Web: Dashboard

- You can see information of your docker including images, containers, volume and networks.



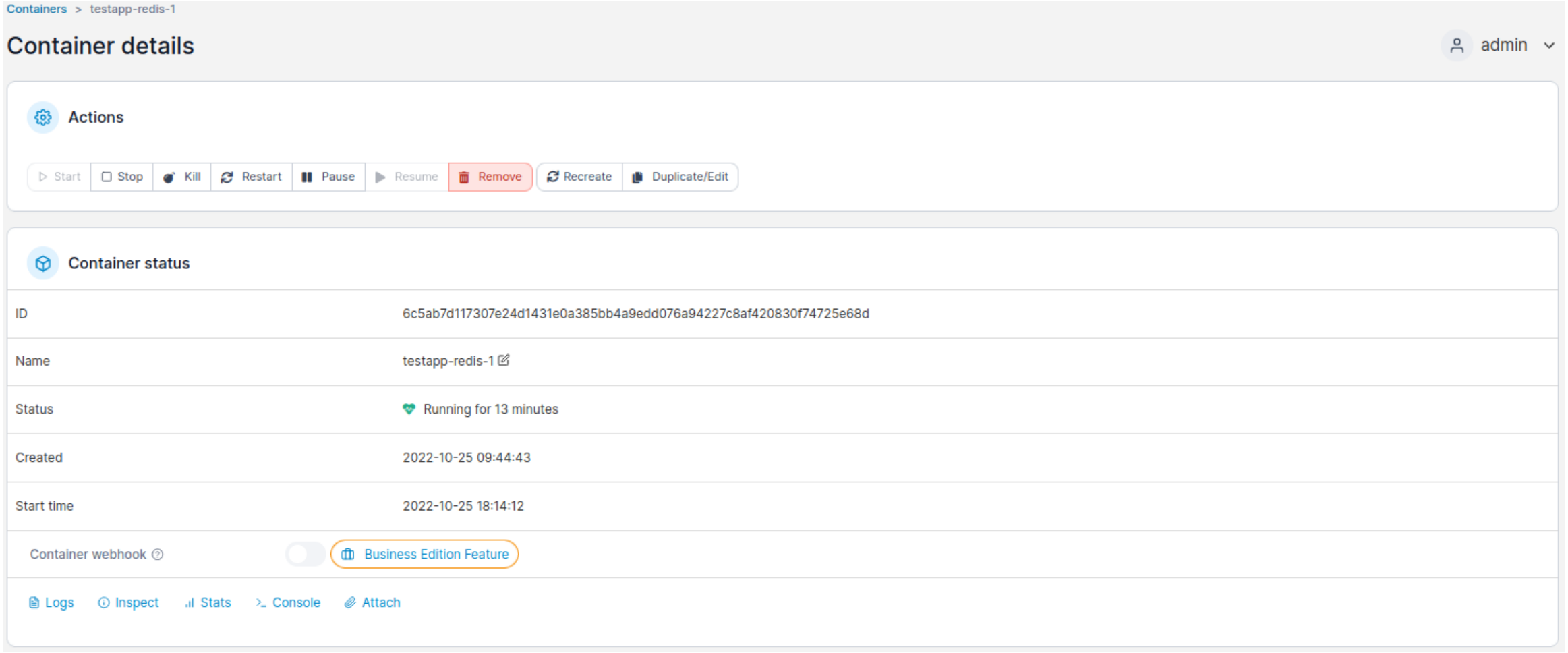
Portainer Web: containers

- Select containers, it will list all the containers in dockers.



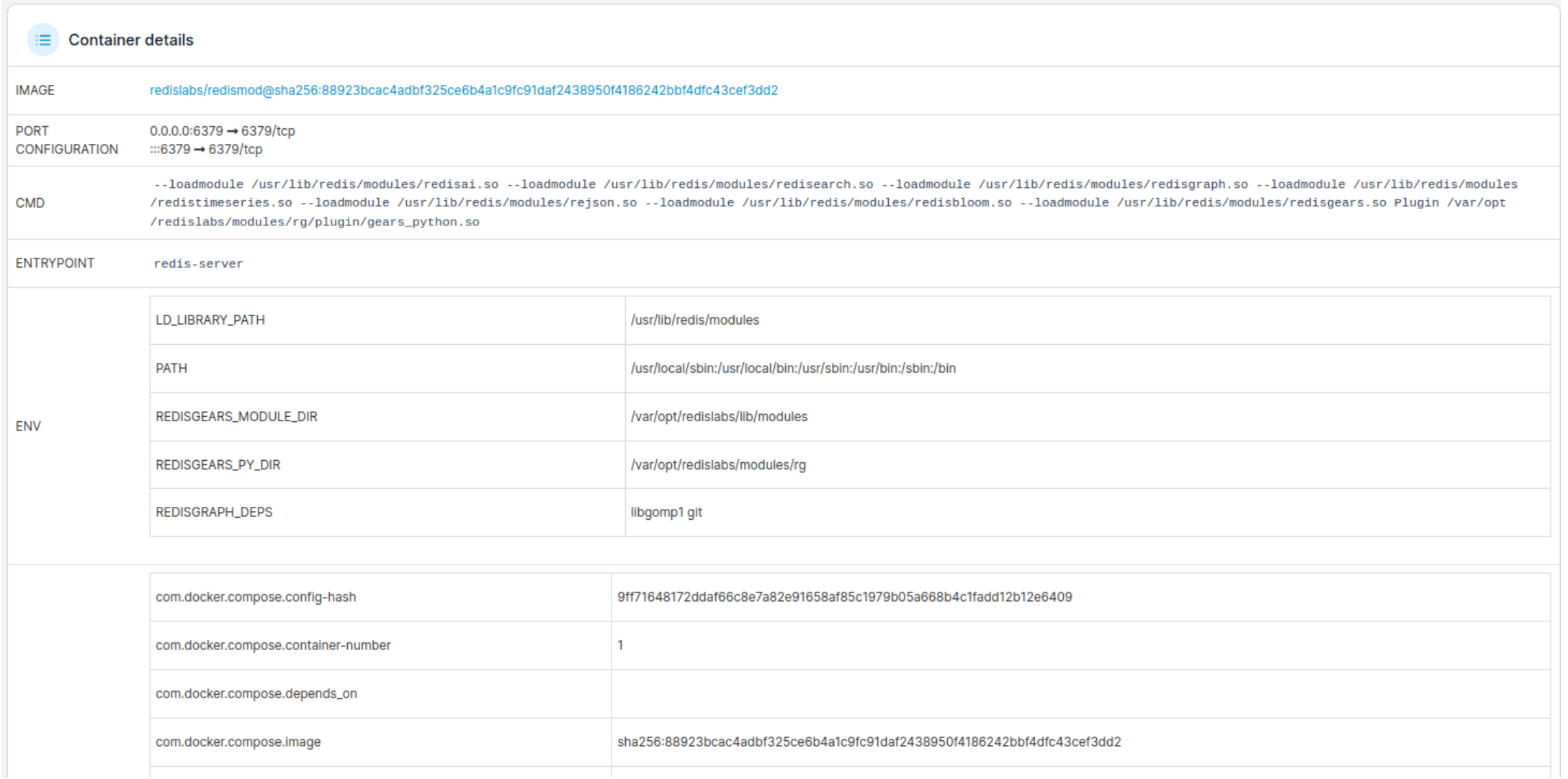
Portainer Web: container detail

- Select container, it will list the detail of that container.



Portainer Web: container detail

- Select container, it will list the detail of that container.



Container details

IMAGE	redislabs/redismod@sha256:88923bcac4adbf325ce6b4a1c9fc91daf2438950f4186242bbf4dfc43cef3dd2
PORT CONFIGURATION	0.0.0.0:6379 ➞ 6379/tcp :::6379 ➞ 6379/tcp
CMD	`--loadmodule /usr/lib/redis/modules/redisai.so --loadmodule /usr/lib/redis/modules/redisearch.so --loadmodule /usr/lib/redis/modules/redisgraph.so --loadmodule /usr/lib/redis/modules/redistimeseries.so --loadmodule /usr/lib/redis/modules/rejson.so --loadmodule /usr/lib/redis/modules/redisbloom.so --loadmodule /usr/lib/redis/modules/redisgears.so Plugin /var/opt/redislabs/modules/rg/plugin/gears_python.so`
ENTRYPOINT	`redis-server`

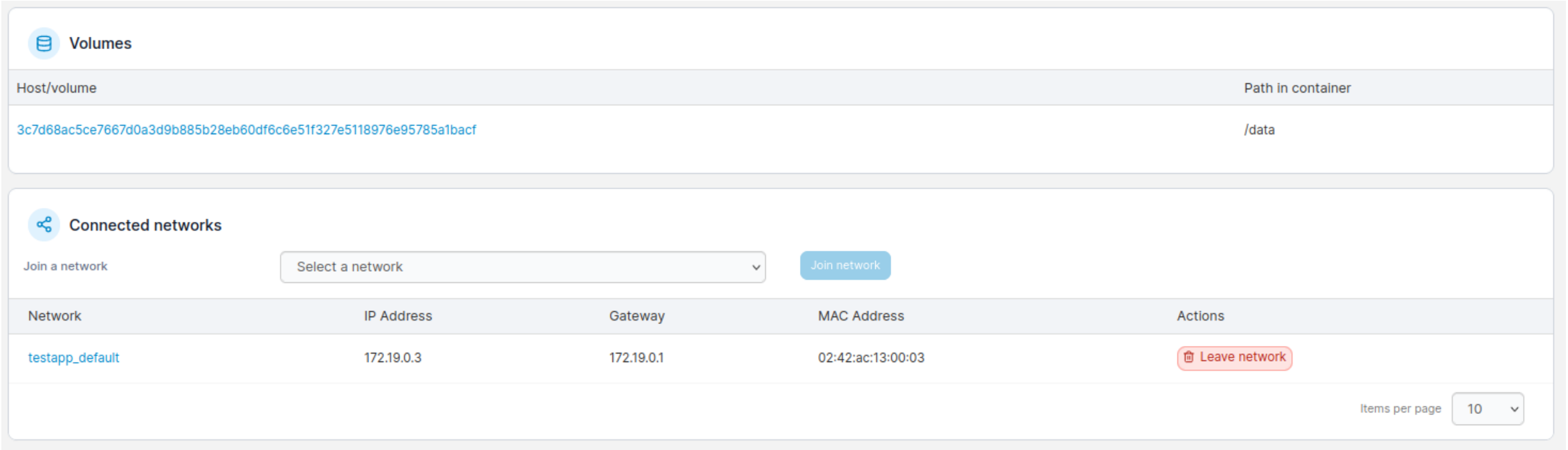
ENV

LD_LIBRARY_PATH	/usr/lib/redis/modules
PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
REDISGEARS_MODULE_DIR	/var/opt/redislabs/lib/modules
REDISGEARS_PY_DIR	/var/opt/redislabs/modules/rg
REDISGRAPH_DEPS	libgomp1 git

com.docker.compose.config-hash	9ff71648172ddaf66c8e7a82e91658af85c1979b05a668b4c1fadd12b12e6409
com.docker.compose.container-number	1
com.docker.compose.depends_on	
com.docker.compose.image	sha256:88923bcac4adbf325ce6b4a1c9fc91daf2438950f4186242bbf4dfc43cef3dd2

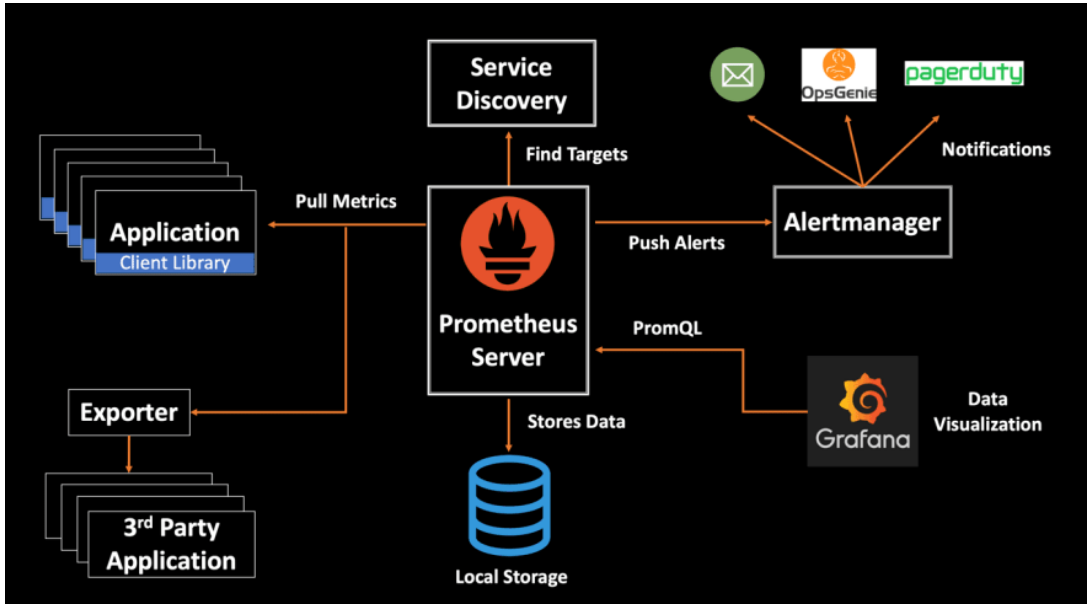
Portainer Web: container detail

- Select container, it will list the detail of that container.



Let try setup dashboard for monitoring dockers

- **Grafana** is a leading time-series, an open-source platform for visualization and monitoring. It allows you to query, visualize, set alerts, and understand metrics no matter where they are stored. You can create amazing dashboards in Grafana to visualize and monitor the metrics.
- **Prometheus** is an open-source time-series monitoring system for machine-centric and highly dynamic service-oriented architectures. It can literally monitor everything. It integrates with Grafana very smoothly.



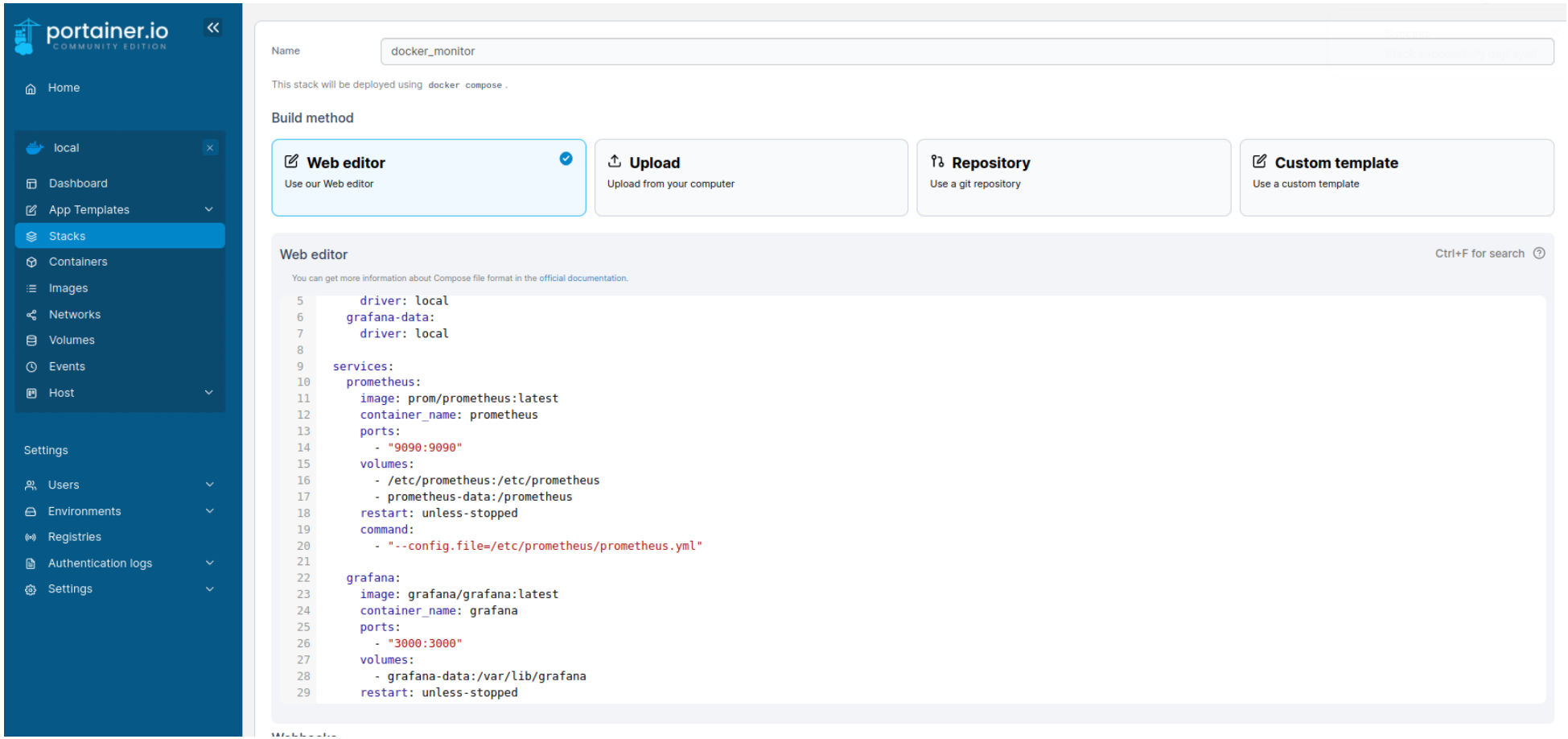
The **Prometheus server** consists of:

- Time Series Database that stores all the metric data like current CPU usage, memory usage etc.
- Data Retrieval Worker is responsible for all the data pulling activities from applications, services, servers etc. and pushing them into the database.
- HTTP Server API meant to accept queries for the stored data. The Server API is used to display the data in a dashboard or a Web UI.

https://dev.to/chinhh/server-monitoring-with-prometheus-and-grafana-266o

Step 1: Deploy Prometheus and Grafana

- Create a new stack and define or paste the content of your docker-compose file in a box web editor.



https://dev.to/chinhh/server-monitoring-with-prometheus-and-grafana-266o

Step 1: Deploy Prometheus and Grafana

- Copy all text from left to right to web editor of stack

```
version: '3'

volumes:
  prometheus-data:
    driver: local
  grafana-data:
    driver: local

services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
      - "9090:9090"
    volumes:
      - /etc/prometheus:/etc/prometheus
      - prometheus-data:/prometheus
    restart: unless-stopped
    command:
      - "--config.file=/etc/prometheus/prometheus.yml"

  node_exporter:
    image: quay.io/prometheus/node-exporter:latest
    container_name: node_exporter
    command:
      - '--path.rootfs=/host'
    pid: host
    restart: unless-stopped
    volumes:
      - '/:/host:ro,rslave'
```

```
cadvisor:
  image: google/cadvisor:latest
  container_name: cadvisor
  # ports:
  #   - "8080:8080"
  volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:ro
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
    - /dev/disk/:/dev/disk:ro
  devices:
    - /dev/kmsg
  restart: unless-stopped

grafana:
  image: grafana/grafana:latest
  container_name: grafana
  ports:
    - "3000:3000"
  volumes:
    - grafana-data:/var/lib/grafana
  restart: unless-stopped
```

Step 2: Configure Prometheus

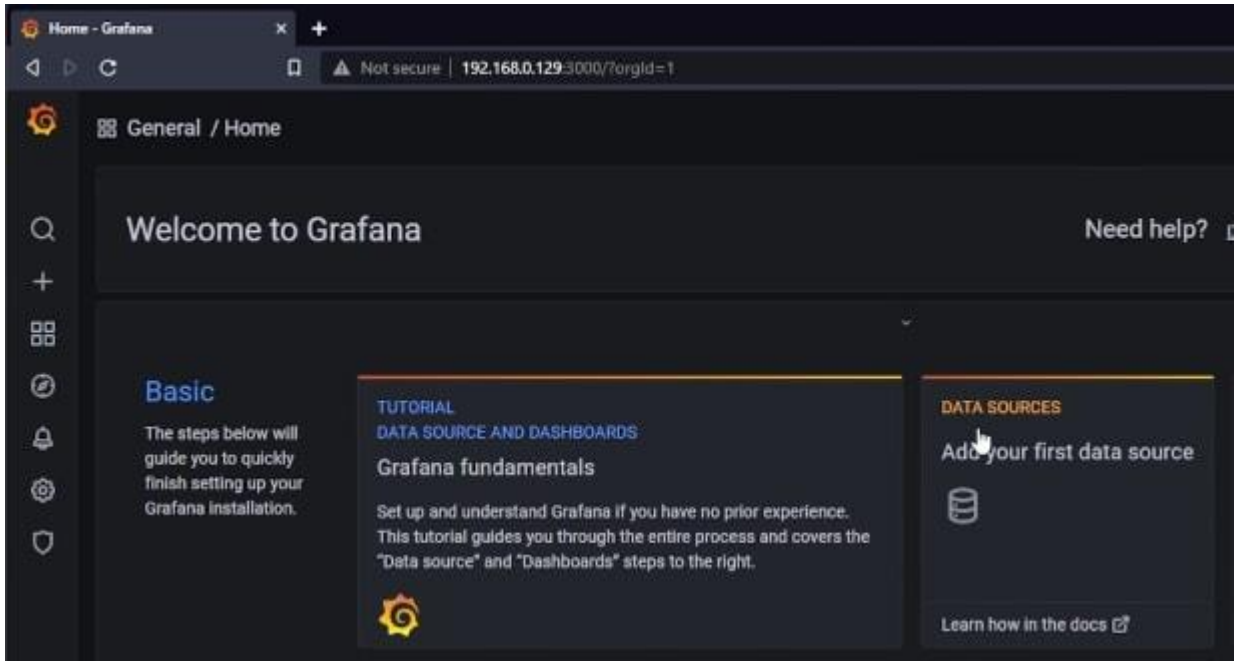
- Create directory (if not exist): /etc/prometheus
- Create file in that directory: nano /etc/prometheus/prometheus.yml

```
global:
  scrape_interval:     15s # By default, scrape targets every 15 seconds.

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: 'prometheus'
    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9090']
  # Example job for node_exporter
  - job_name: 'node_exporter'
    static_configs:
      - targets: ['node_exporter:9100']

  # Example job for cadvisor
  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']
```

- You may need to restart prometheus container

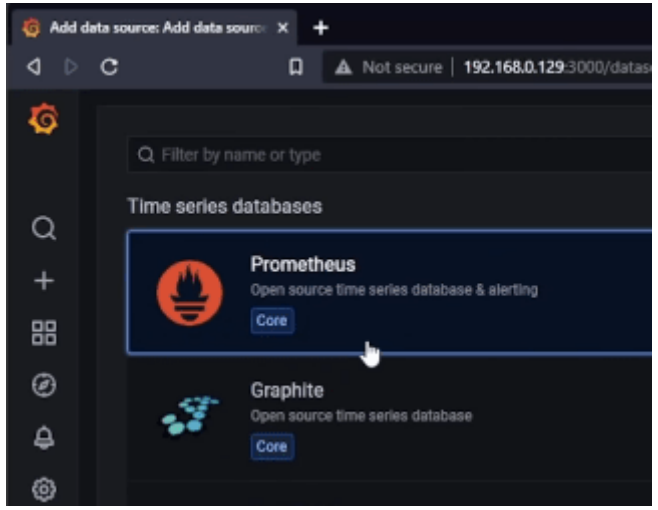
Step 3: Configure Grafana

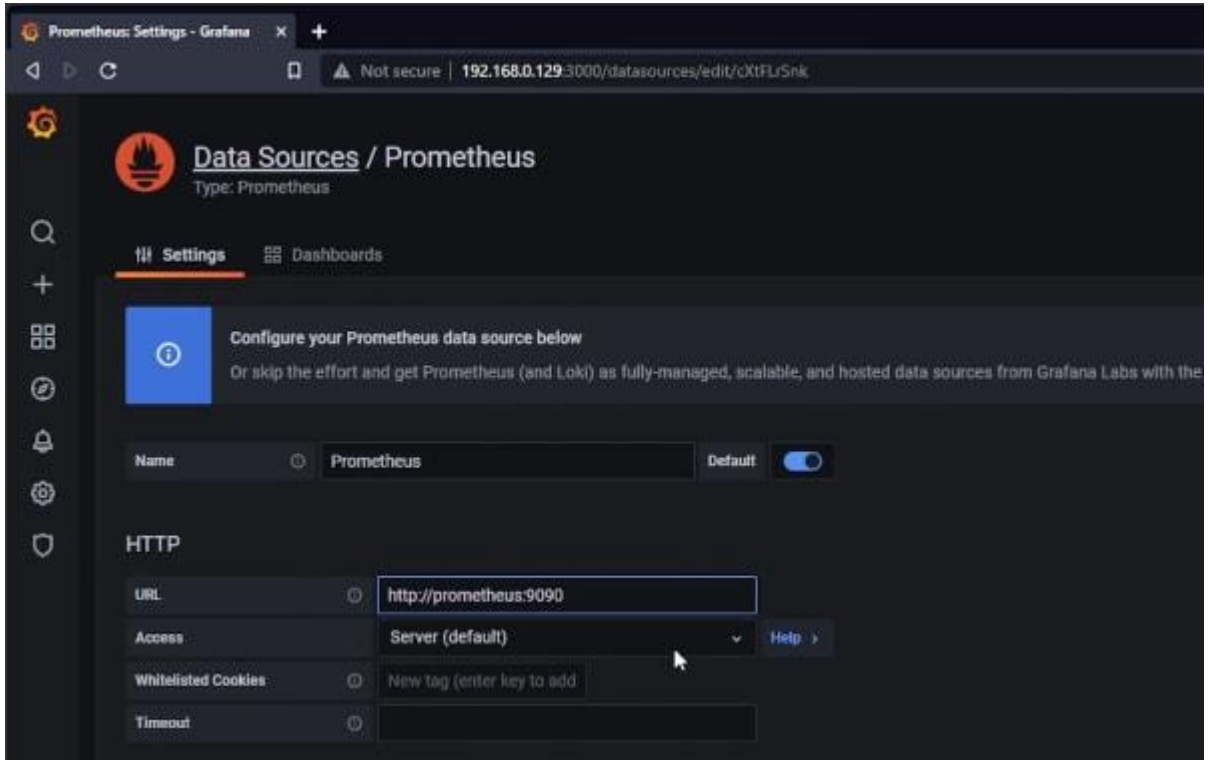
- In the Grafana page we built above at localhost:3000. Here we log in with the user and password as admin.
- Then select data sources to add prometheus



Step 3: Configure Grafana

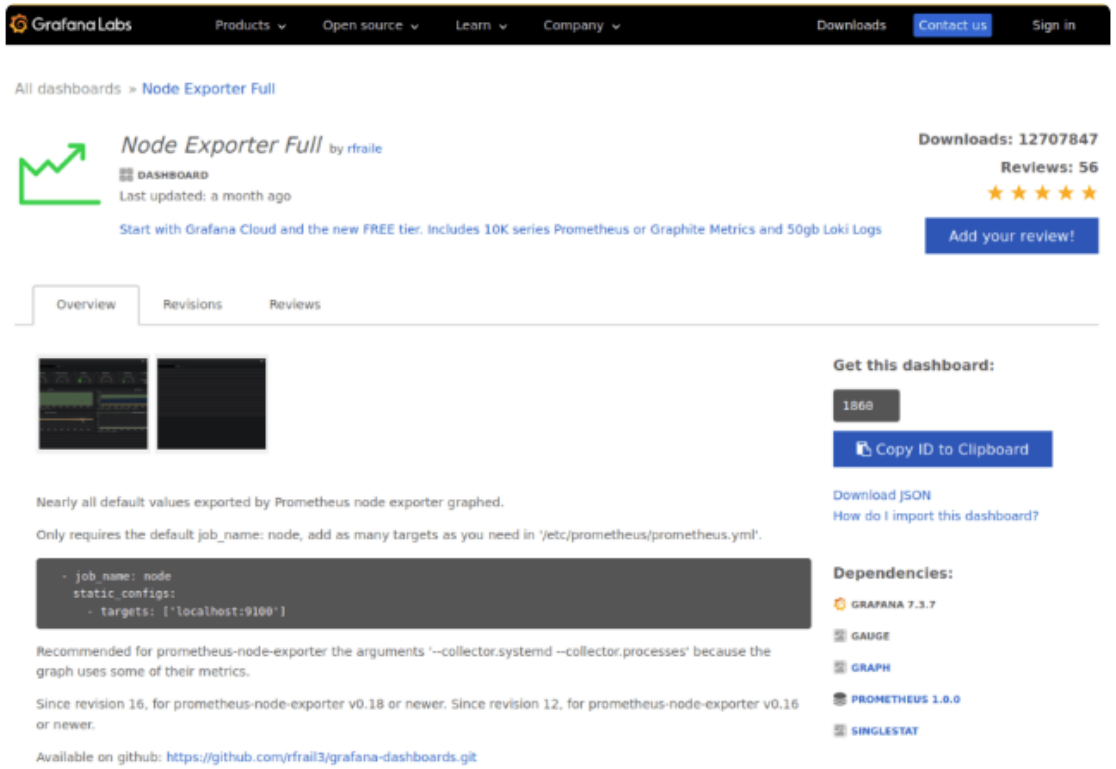
- select prometheus
- At the setting we need to reconfigure the HTTP tab at the URL is http://prometheus:9090



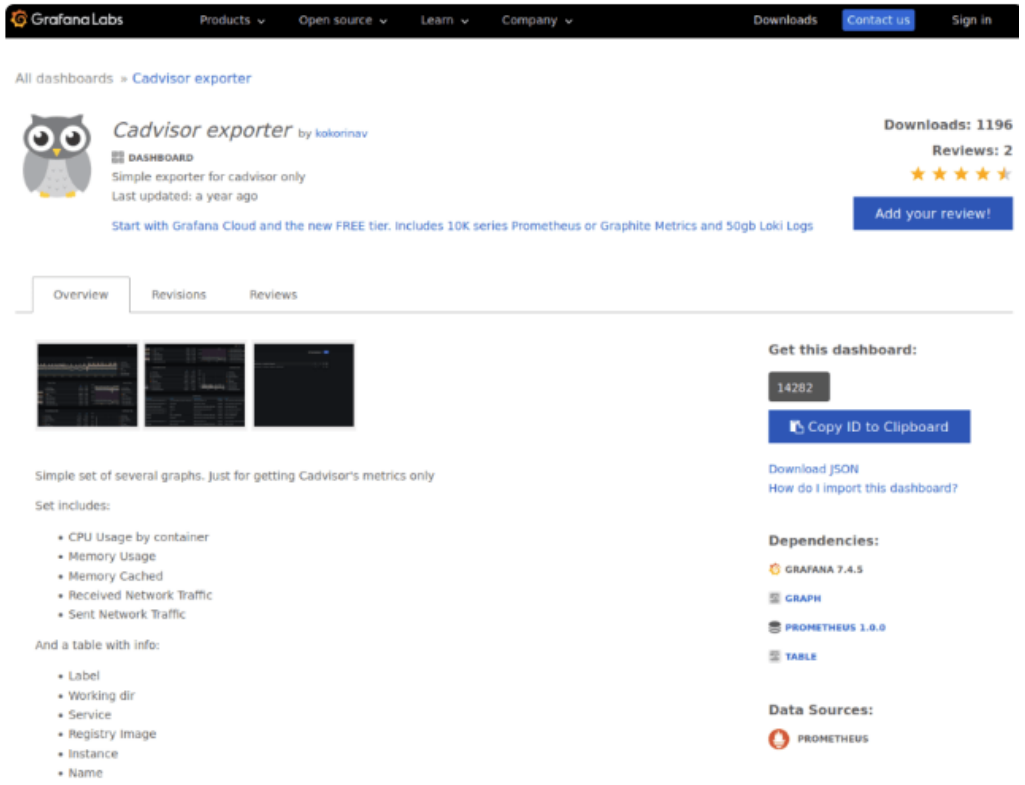


Step 4: Import Grafana Dashboards

- In the Grafana homepage https://grafana.com at the Dashboards tab we will see there are many dashboards. In this tutorial we will learn about node exporters and cadvisor.



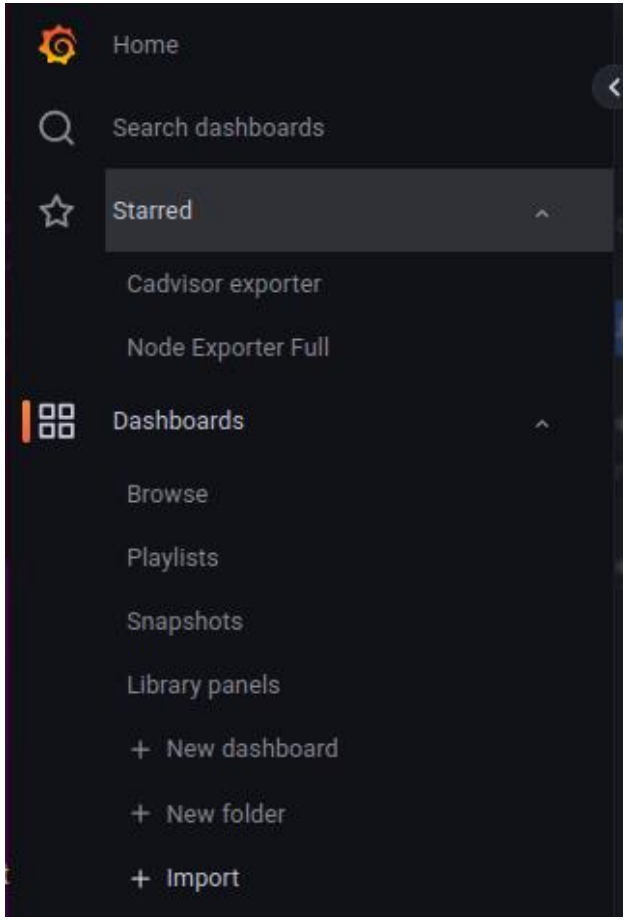
Server Monitoring (1860)

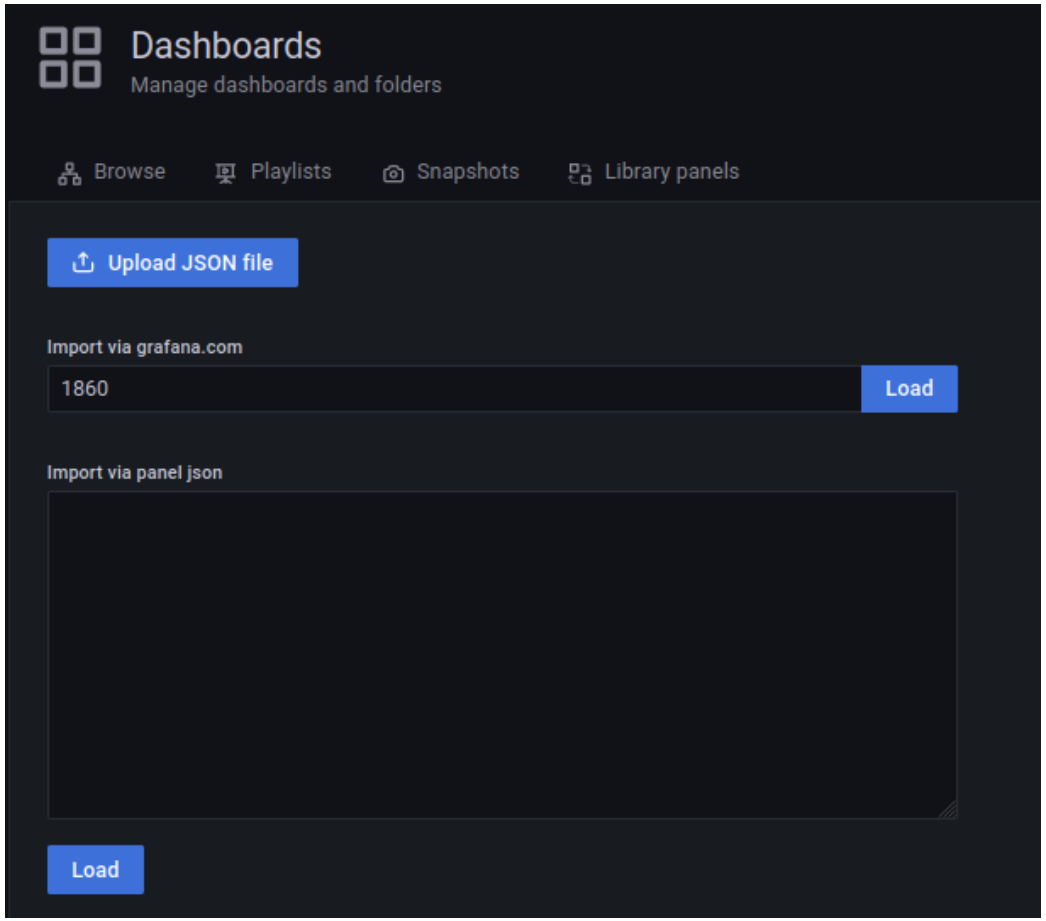


Docker Monitoring (14282)

Step 4: Import Grafana Dashboards

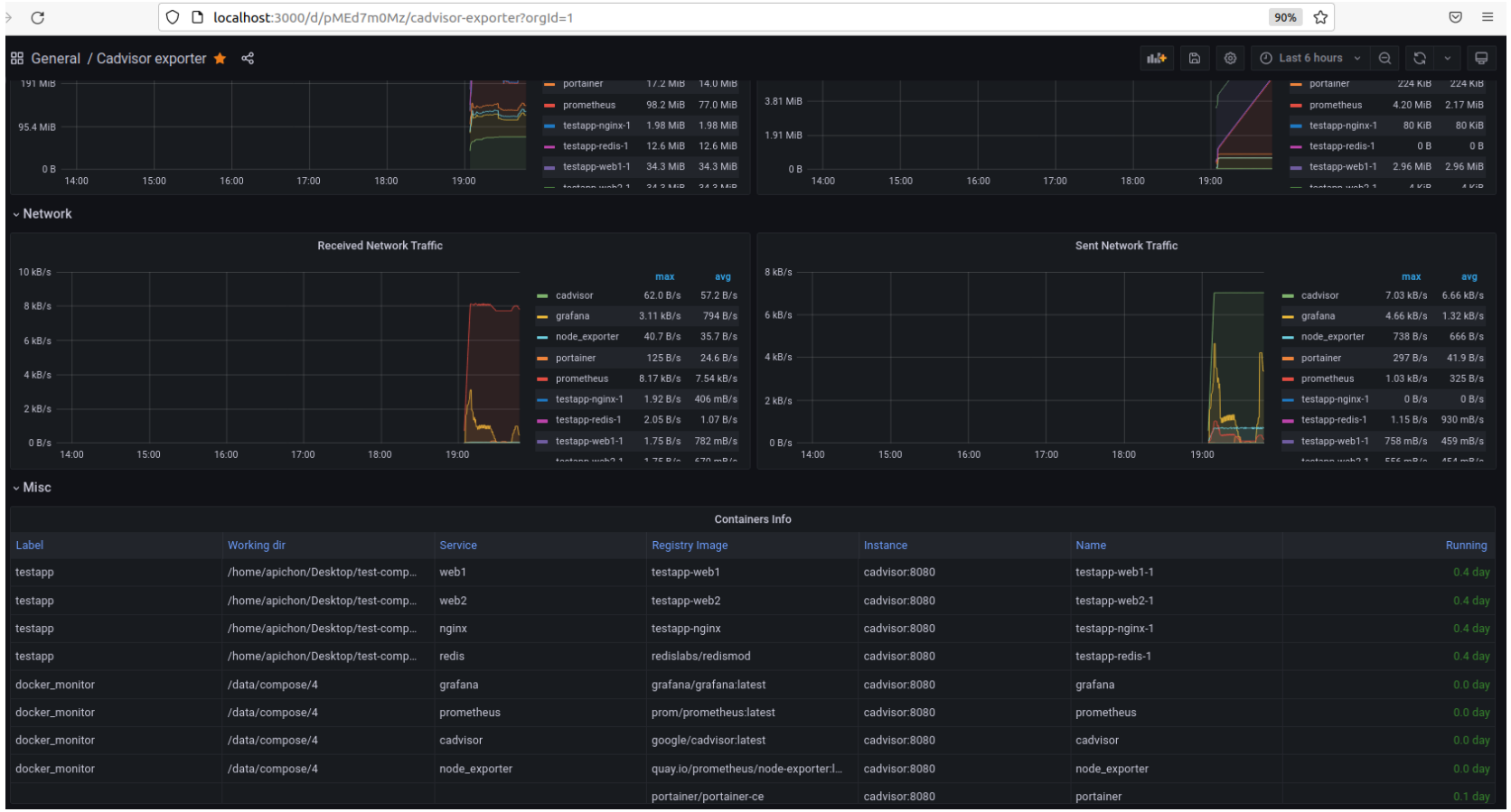
- On Dashboards Menu, select import
- Input dashboard id from website and click load





Tutorial check point : 7

- Access to cadvisor-export dashboard
- Capture screen of the dashboard



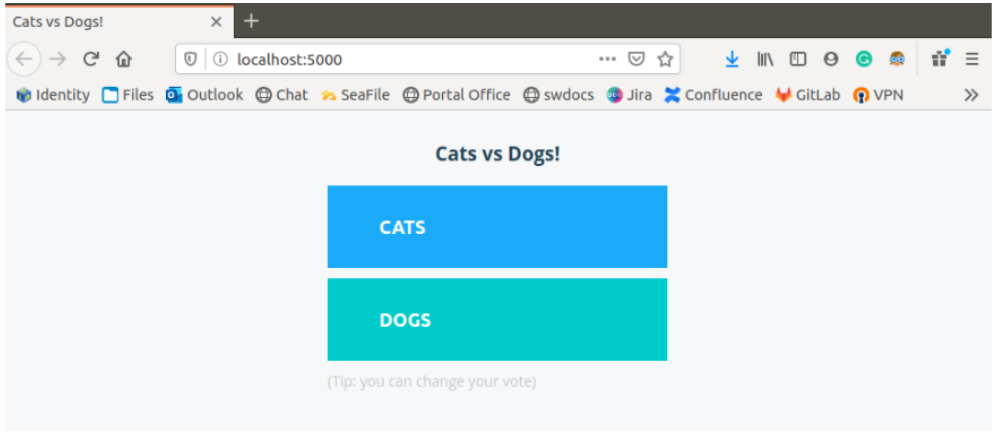
Tutorial Roadmap

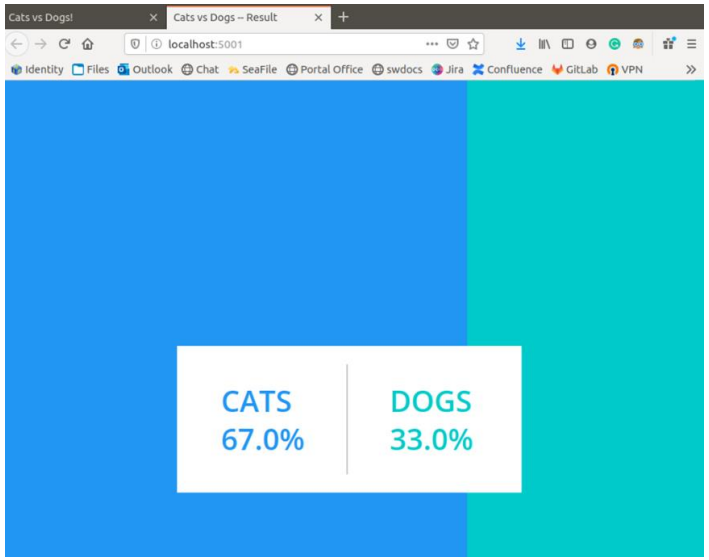
- Storage with Docker Volume
- Storage with Bind Host
- Create custom build web image with code from GitHub
- Dockerfile with build argument
- Dockerfile with environment variable
- Share your image on Docker Hub
- Deploy image on Azure App Service
- A Node.js application using Nginx, Docker and Redis
- Portainer – Docker Management
- Dashboard for monitoring dockers
- Voting App with docker-compose

Next: Voting App with docker-compose

We will deploy the existing voting app from GitHub

- **Voting-app** : Front-end of the application written in python flask framework, which allows the users to cast their votes.
- **Redis** : An in-memory data structure store, used as a temporary database to store the votes casted by the users.
- **Worker** : service written in .NET, that retrieves the votes data from redis and stores it into PostgreSQL DB service.
- **PostgreSQL** DB : PostgreSQL DB used as persistent storage database.
- **Result-app** : service written in node js, displays the voting results to the user

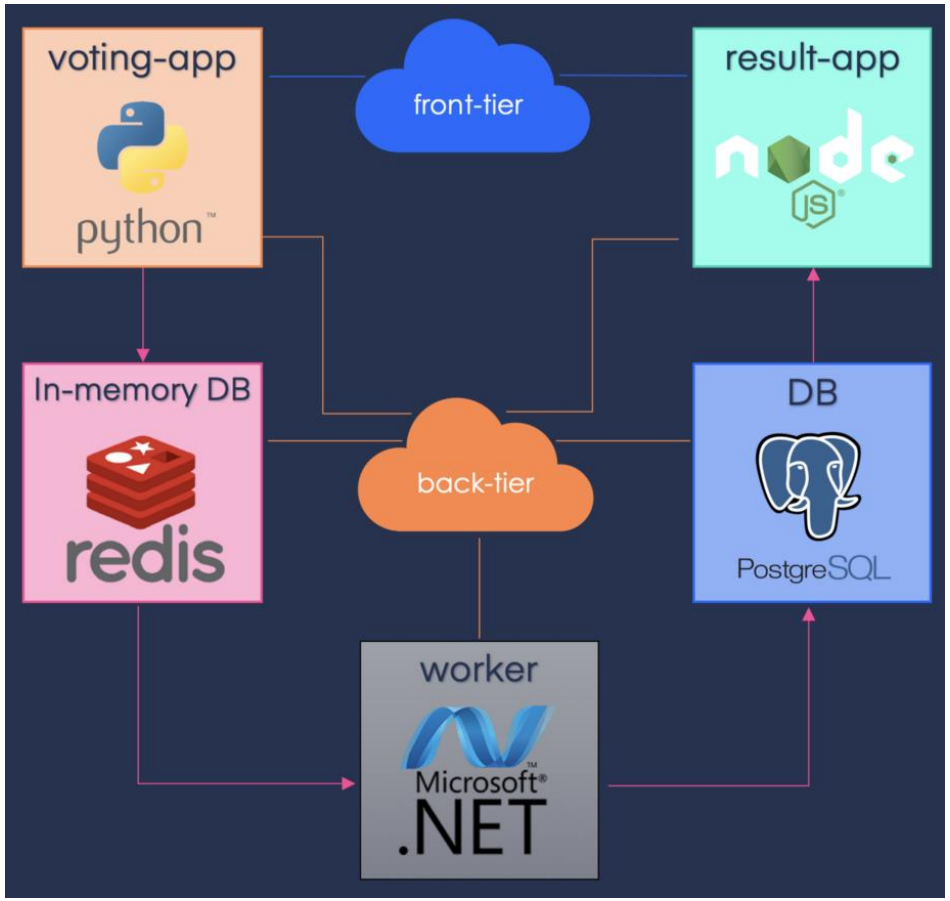




Next: Voting App with docker-compose

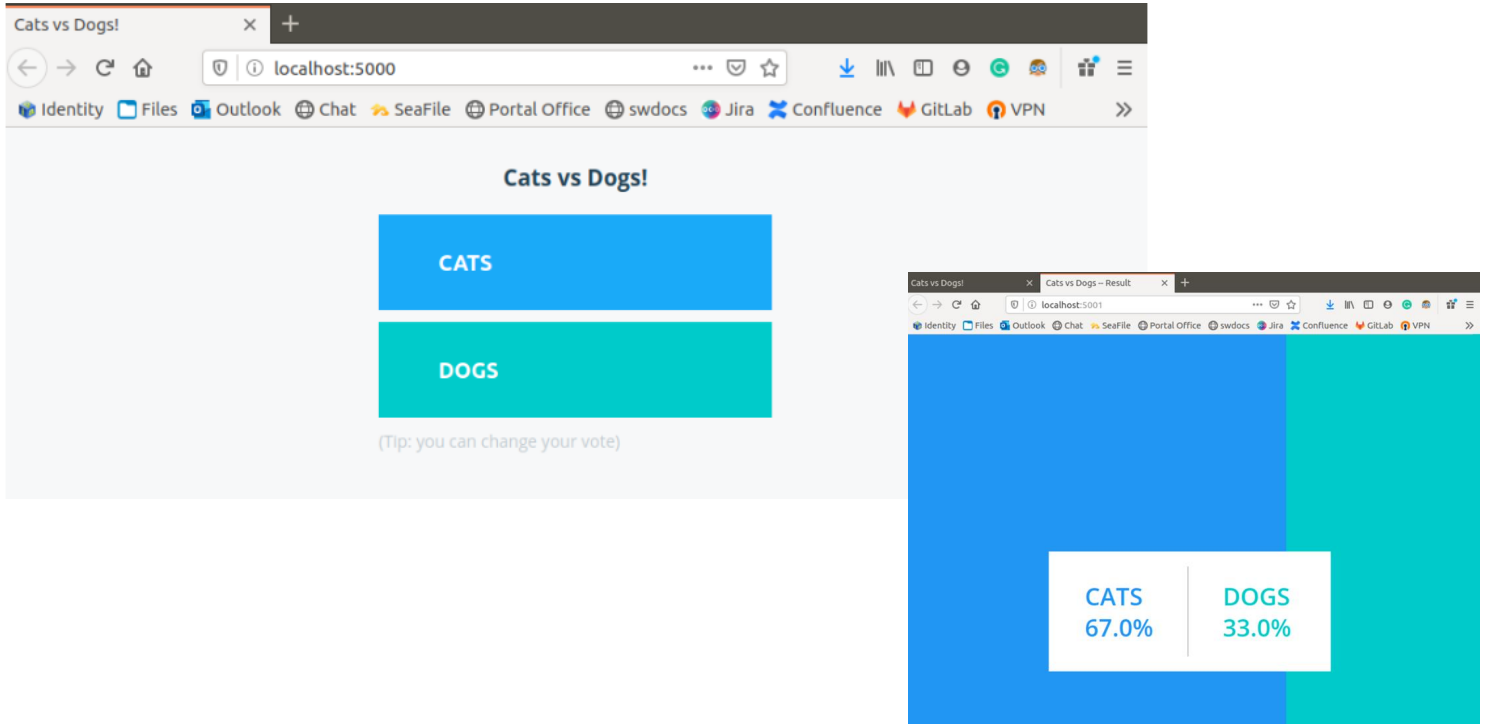
We will deploy the existing voting app from GitHub

- **Voting-app** : Front-end of the application written in python flask framework, which allows the users to cast their votes.
- **Redis** : An in-memory data structure store, used as a temporary database to store the votes casted by the users.
- **Worker** : service written in .NET, that retrieves the votes data from redis and stores it into PostgreSQL DB service.
- **PostgreSQL** DB : PostgreSQL DB used as persistent storage database.
- **Result-app** : service written in node js, displays the voting results to the user



Tutorial check point : 8

- Get code from GitHub and deploy
- https://github.com/dockersamples/example-voting-app
- Capture screen with current running container, vote web and result web



Done.