**Dockers and Containers**

**Installation**

<https://docs.docker.com/engine/install/ubuntu/>

#### Set up the repository.

1. Update the apt package index and install packages to allow apt to use a repository over HTTPS:

$ sudo apt-get update

$ apt-get install \

> apt-transport-https \

> ca-certificates \

> curl \

> gnupg \

> lsb-release

1. Add Docker’s official GPG key:

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

1. Use the following command to set up the **stable** repository.

echo \

> "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \

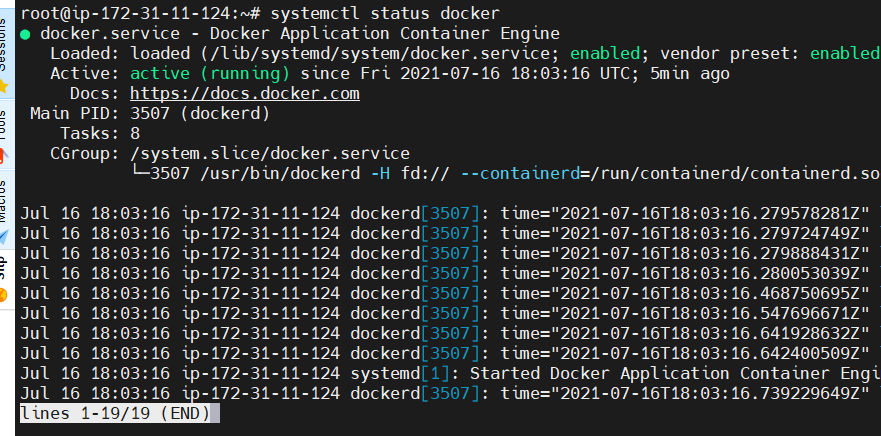
#### **Install Docker Engine**

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io

#### **Check the status of the docker**

systemctl status docker



If it is not up and running, then run the command

systemctl start docker

Dockerfile: A text file with instructions to build image Automation of Docker Image Creation

FROM

RUN

CMD

Step 1 : Create a file named Dockerfile

Step 2 : Add instructions in Dockerfile

Step 3 :

Build dockerfile to create image

Step 4 : Run image to create container

COMMANDS : docker build : docker build -t ImageName:Tag directoryOfDocekrfile : docker run image

Create a folder

mkdir DockerFiles

touch Dockerfile

nano Dockerfile

FROM ubuntu

RUN apt-get udpate

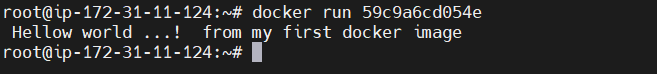
CMD [“echo”, “hellow world my first docker file”]

docker build .

docker build -t myimage1:1.0 .

docker images

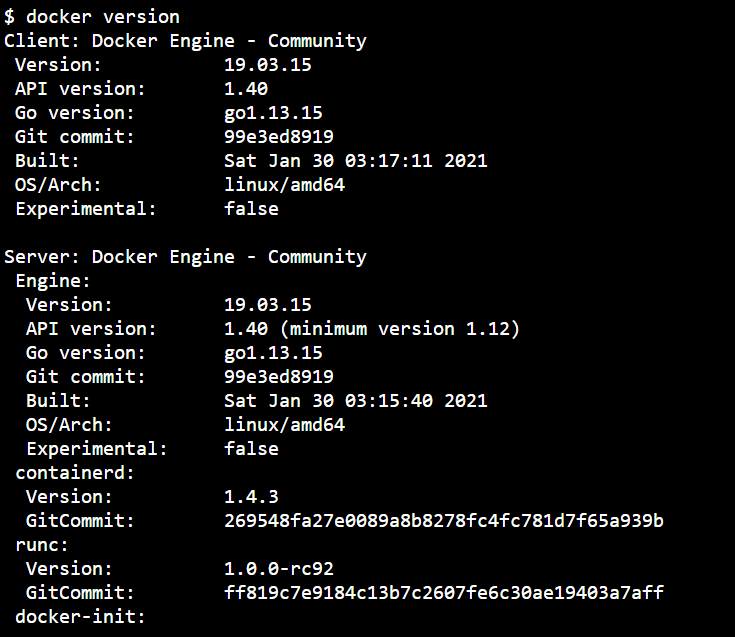
docker run <image name or container name>



**Docker Practice LAB**

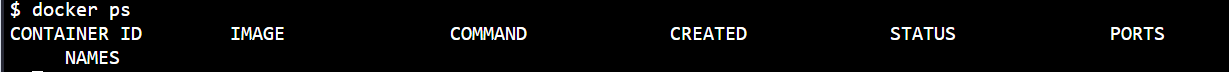
* What is the docker version running?

$ docker version



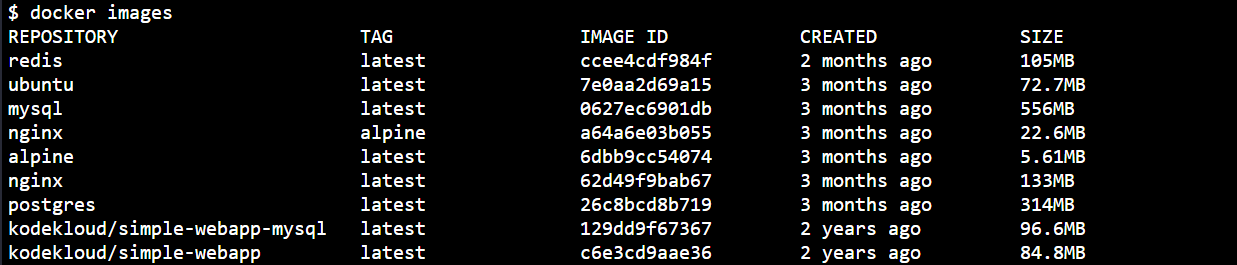
* How many containers are running on this host?

$ docker ps



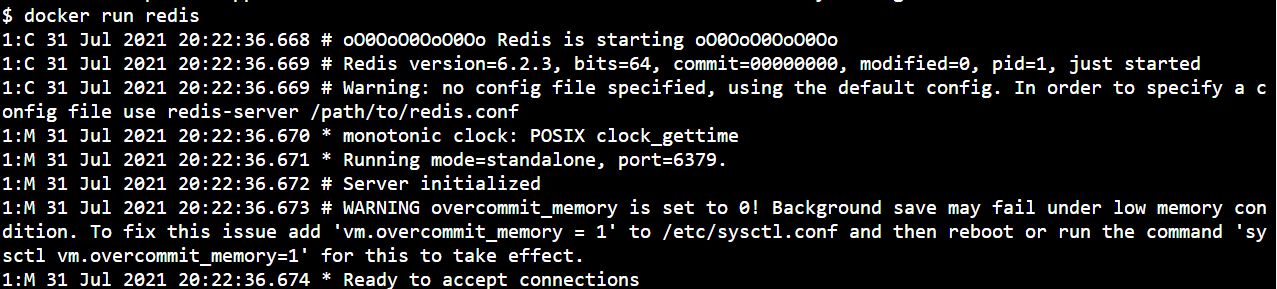
* How many images are available on this host?

$ docker images



* Run a container using the redis image

$ docker run redis



* Stop the container you just created

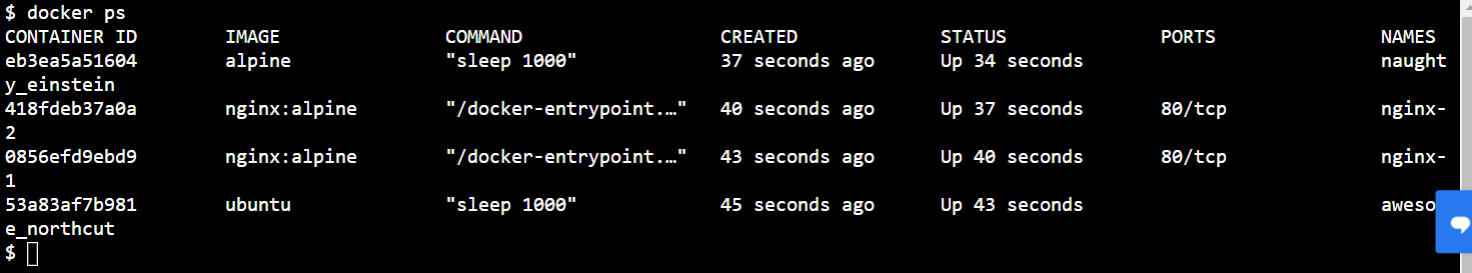
Hit CTRL+C if you are on the container's terminal. Or else run

$docker stop <container-id>.



* How many containers are RUNNING on this host now?

$ docker ps



* How many containers are PRESENT on the host now?

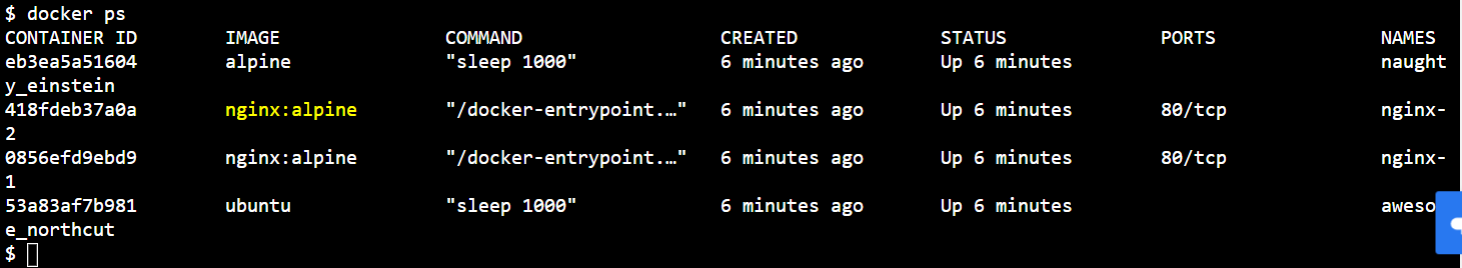
Including both Running and Not Running ones

$ docker ps -a



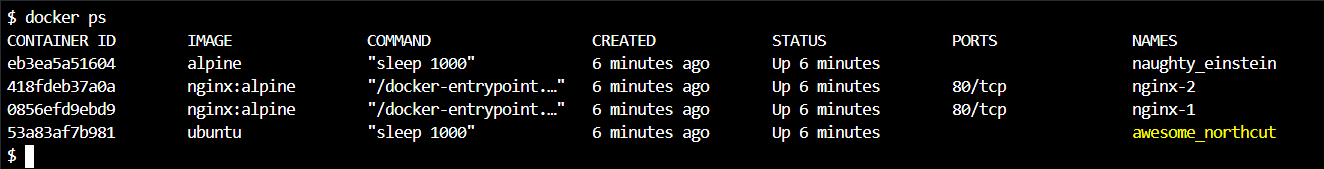
* What is the image used to run the nginx-1 container?

$ docker ps



* What is the name of the container created using the ubuntu image?

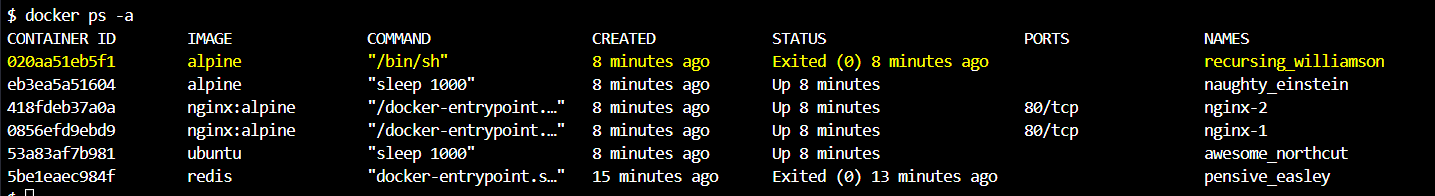
$ docker ps



* What is the ID of the container that uses the alpine image and is not running?

$ docker ps -a

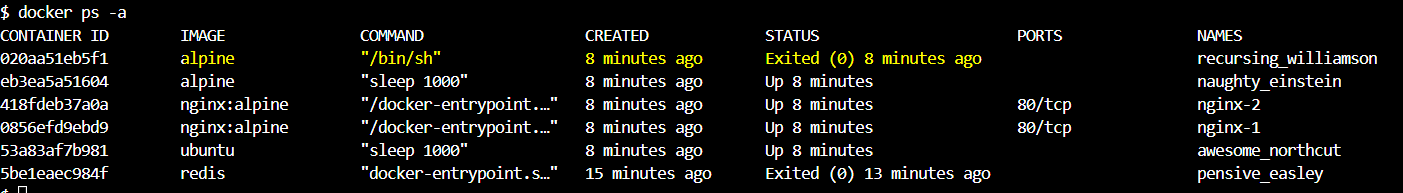
Check for the command and container id



* What is the state of the stopped alpine container?

$ docker ps -a

Check for the status



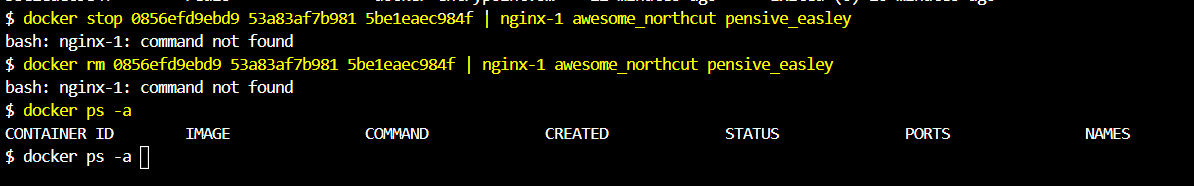
* Delete all containers from the Docker Host.

Both Running and Not Running ones. Remember you may have to stop containers before deleting them.

To stop containers run the command

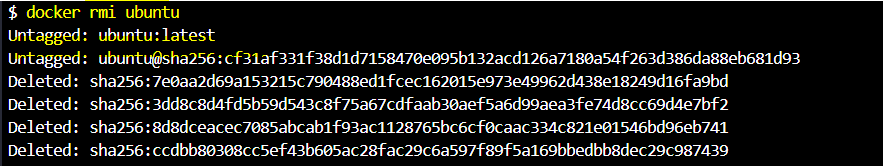
$docker stop <container id | container name>  
And then to delete them run

$docker rm <container id | container name>



* Delete the ubuntu Image.

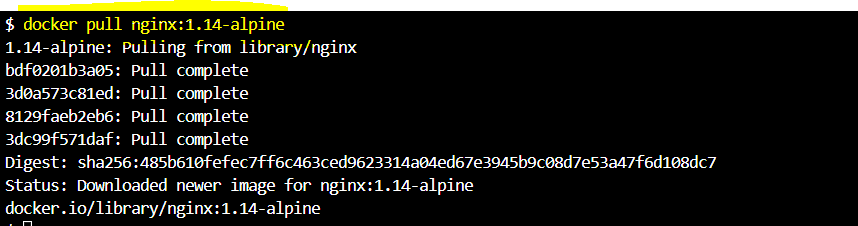
$ docker rmi ubuntu



* You are required to pull a docker image which will be used to run a container later. Pull the image nginx:1.14-alpine

Only pull the image, do not create a container.

$ docker pull nginx:1.14-alpine



* Run a container with the nginx:1.14-alpine image and name it webapp

Run the command

$docker run -d --name webapp nginx:1.14-alpine

and check the status of created container by

$docker ps command.



* Cleanup: Delete all images on the host

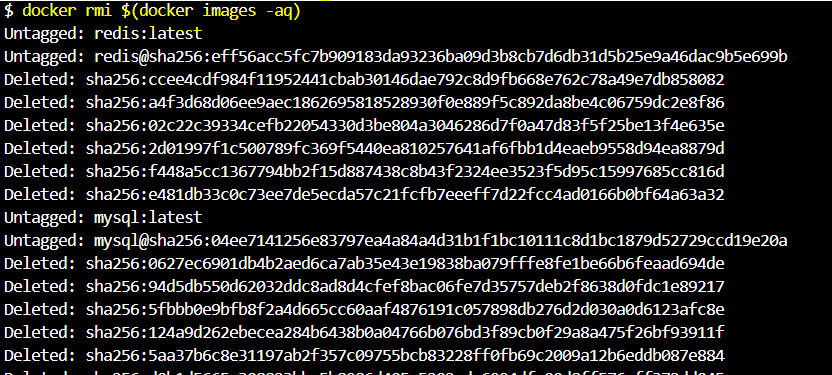
Remove containers as necessary

Run the command

docker rmi <IMAGE:TAG>

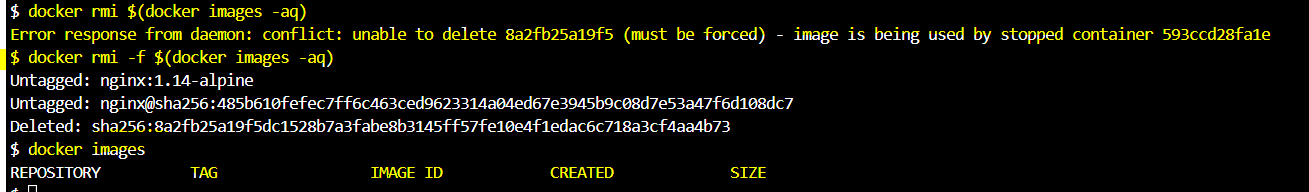
Stop and delete all the containers being used by images.  
Then run the command to delete all the available images:

$docker rmi $(docker images -aq)



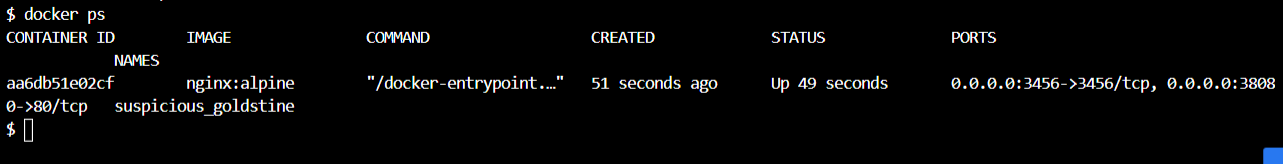
Didn’t delete all the image need to use force command to delete

docker rmi -f $(docker images -aq)



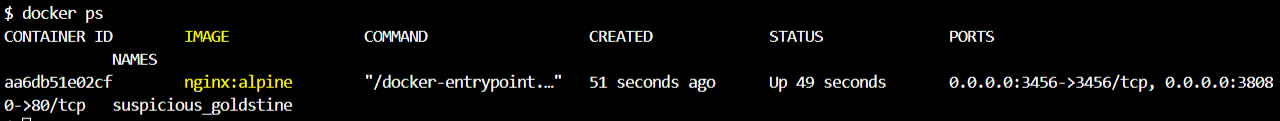
* How many containers are running on this host?

$docker ps



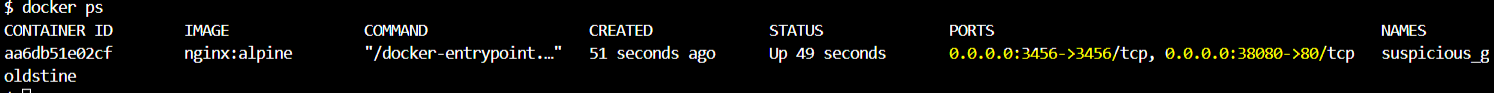
* What is the image used by the container?

Run the command docker ps and look under the IMAGE column.



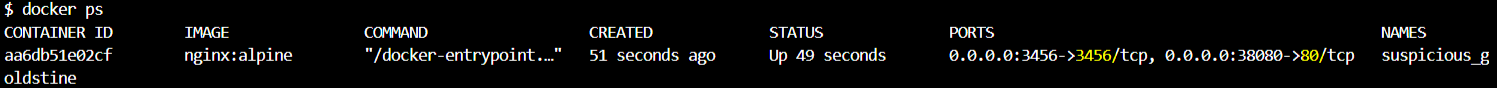
* How many ports are published on this container?

Run the command docker ps and look under the PORTS column.



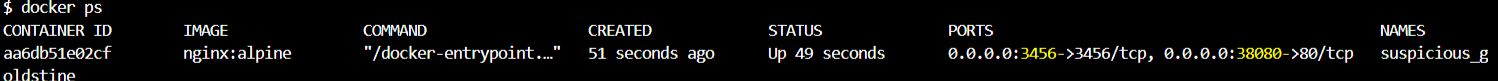
* Which of the below ports are the exposed on the CONTAINER?

Run the command docker ps and look under the PORTS column. Ports on the right(after ->) are exposed on the container.



* Which of the below ports are published on Host?

Run the command docker ps and look under the PORTS column. Ports on the left(before ->) are published on the host.



* Run an instance of kodekloud/simple-webapp with a tag blue and map port 8080 on the container to 38282 on the host.

Run the command:

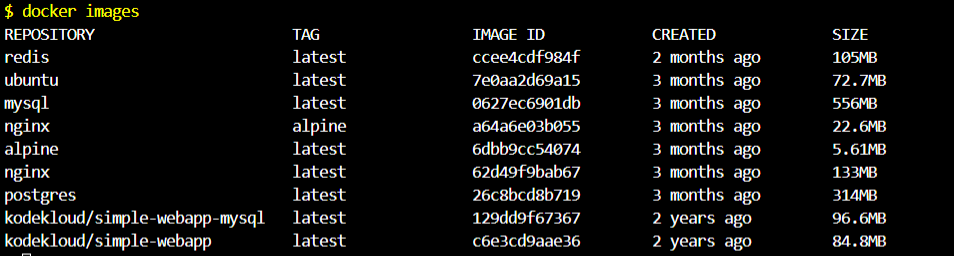
$docker run -p 38282:8080 kodekloud/simple-webapp:blue  
You can run this container in the background after adding the -d flag.



**Docker IMAGES**

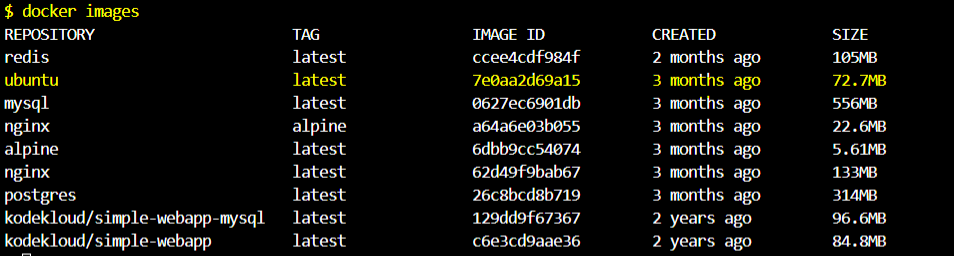
* How many images are available on this host?

Run the command docker images and count the number of available images.



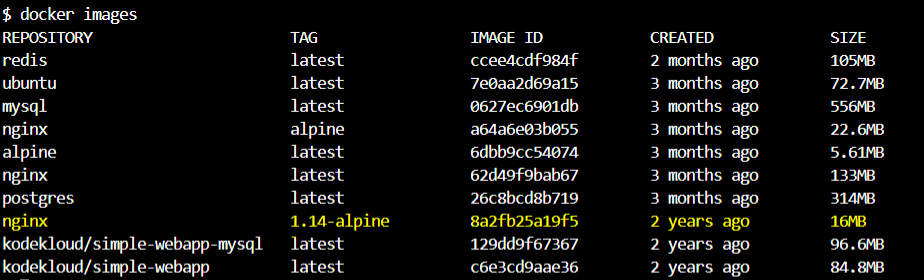
* What is the size of the Ubuntu image?

Run the command docker images and look under the SIZE column.



* We just pulled a new image. What is the tag on the newly pulled NGINX image?

Run the docker images command and look for the TAG column.



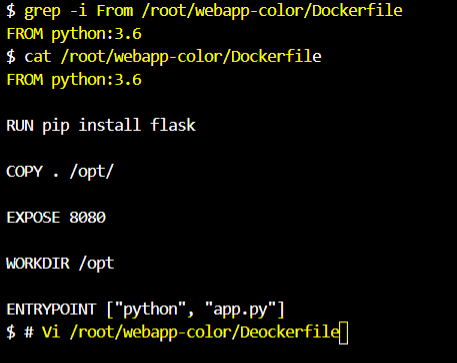
* We just downloaded the code of an application. What is the base image used in the Dockerfile?

Inspect the Dockerfile in the webapp-color directory.

You can either open the file using

vi /root/webapp-color/Dockerfile (or using commands such as cat/more/less/vim e.t.c) and look for the FROM instruction or search for it directly using

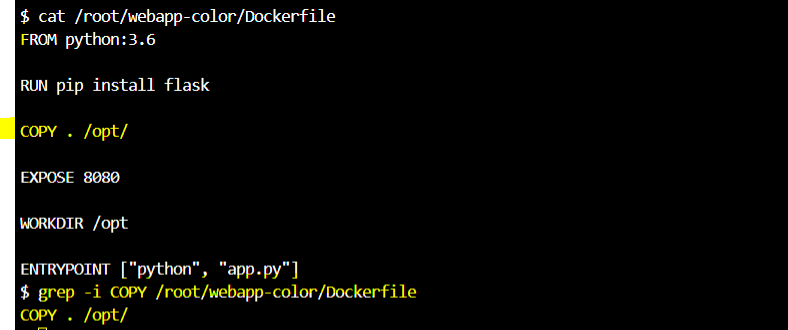
grep -i FROM /root/webapp-color/Dockerfile.



* To what location within the container is the application code copied to during a Docker build?

Inspect the Dockerfile in the webapp-color directory.

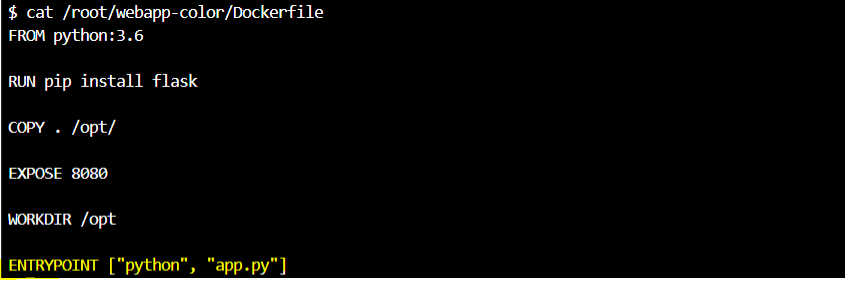
Open the Dockerfile and look for COPY command.



* When a container is created using the image built with this Dockerfile, what is the command used to RUN the application inside it.

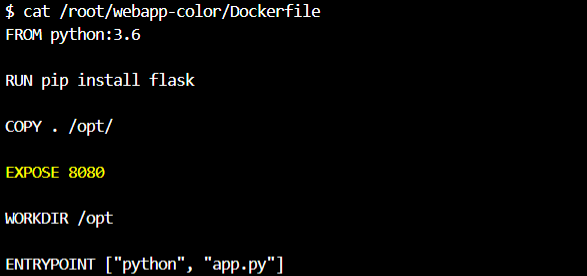
Inspect the Dockerfile in the webapp-color directory.

Open the Dockerfile and look for ENTRYPOINT command.



* What port is the web application run within the container? Inspect the Dockerfile in the webapp-color directory.

Open the Dockerfile and look for the port in EXPOSE command.



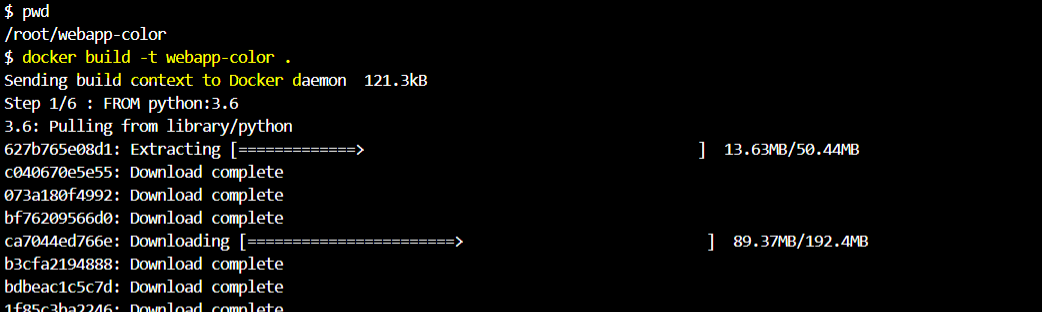
* Build a docker image using the Dockerfile and name it webapp-color. No tag to be specified.

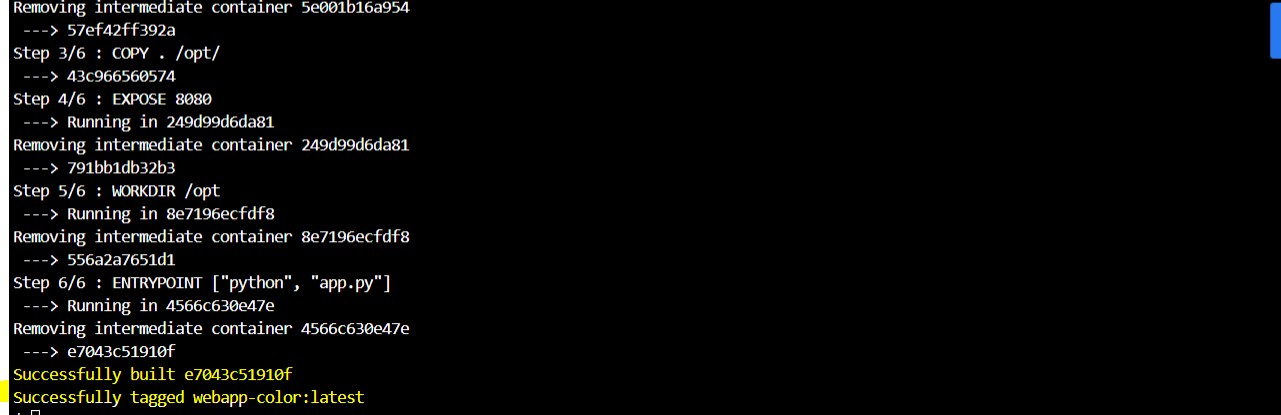
Move to the directory first by using the cd command and verify the path of the working directory from pwd command :

Now, run the docker build command within that directory :

Run the command

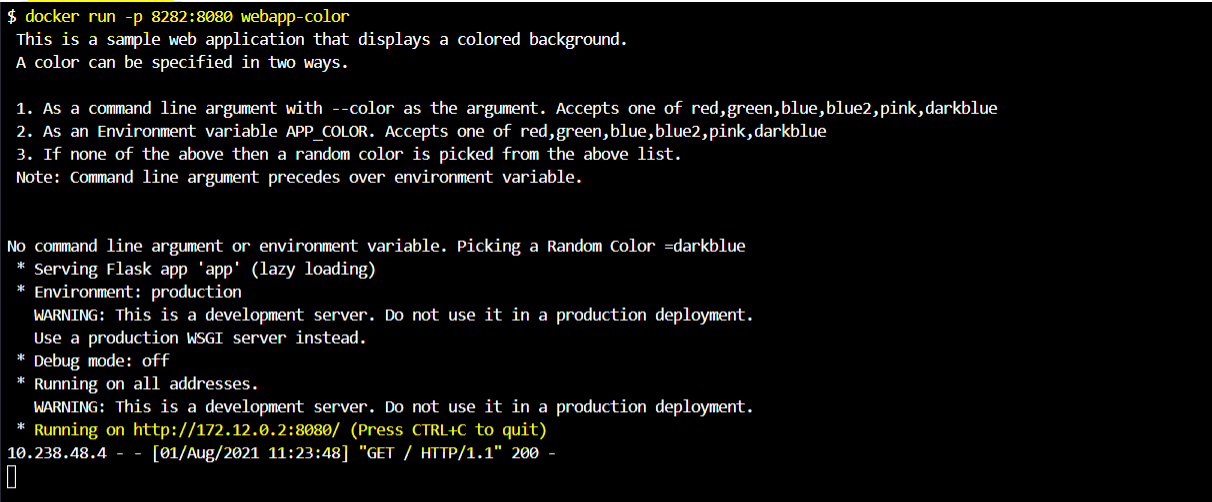
$docker build -t webapp-color .





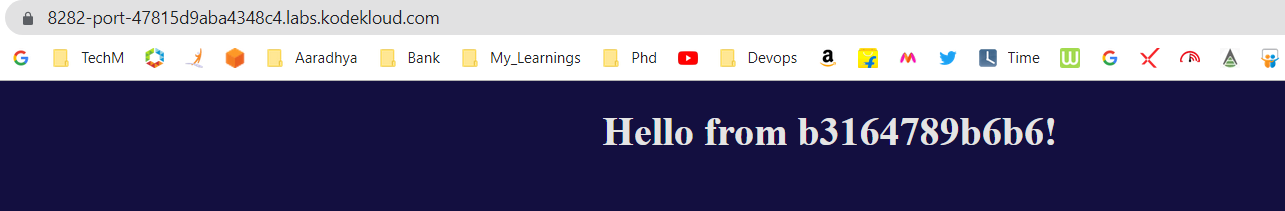
* Run an instance of the image webapp-color and publish port 8080 on the container to 8282 on the host.

Command: docker run -p 8282:8080 webapp-color



* Access the application by clicking on the tab named HOST:8282 above your terminal.

After you are done, you may stop the running container by hitting CTRL + C if you wish to.

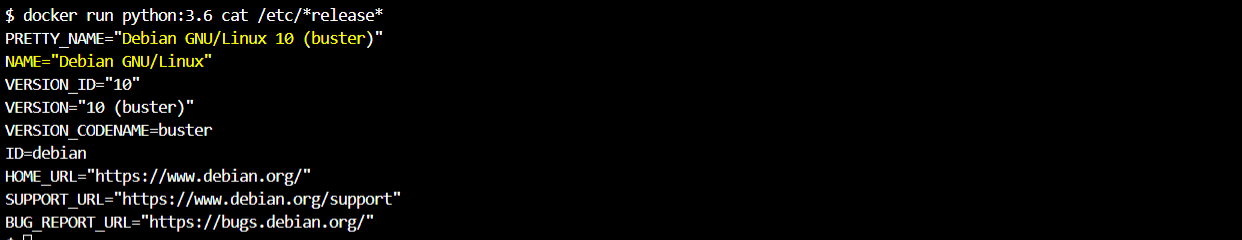


* What is the base Operating System used by the python:3.6 image?

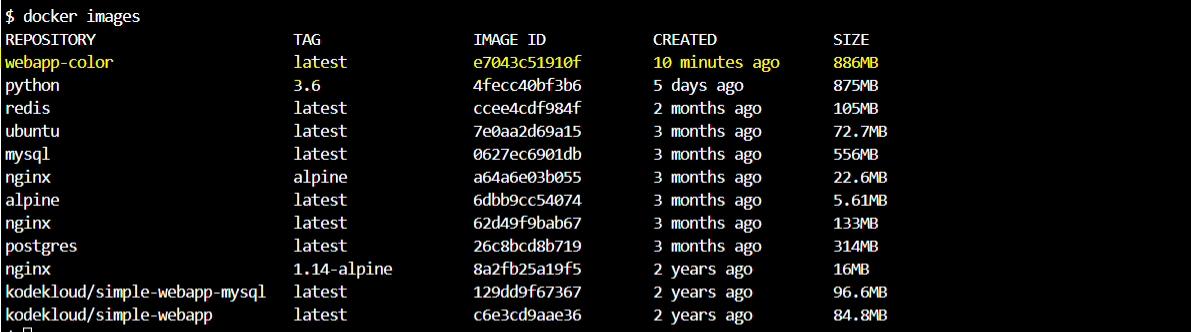
If required, run an instance of the image to figure it out.

Answer: Debian

Run $docker run python:3.6 cat /etc/\*release\* command.



* What is the approximate size of the webapp-color image?



The size is very big we need to make it small and modify the image

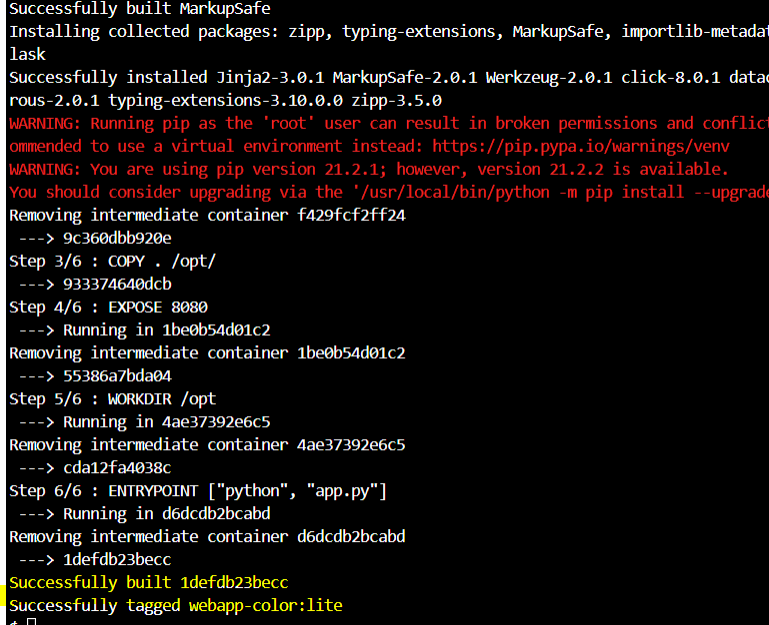
* Build a new smaller docker image by modifying the same Dockerfile and name it webapp-color and tag it lite.

Hint: Find a smaller base image for python:3.6. Make sure the final image is less than 150MB.

Modify Dockerfile to use python:3.6-alpine image and then build

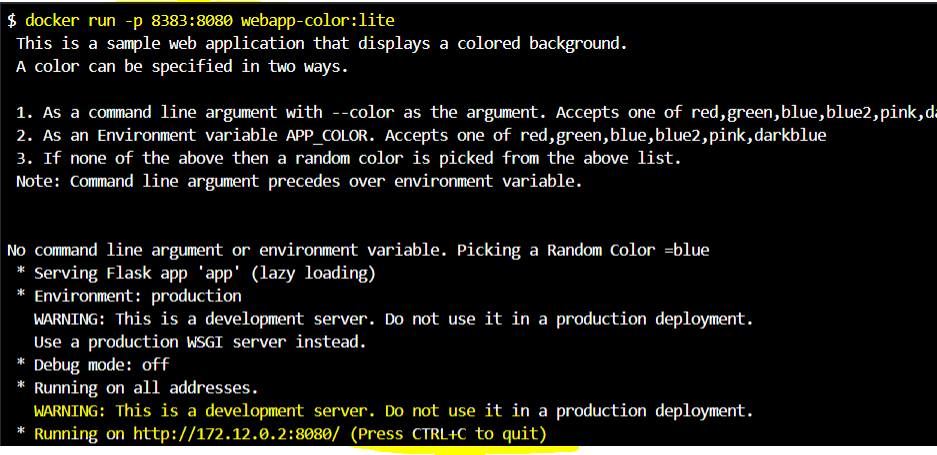


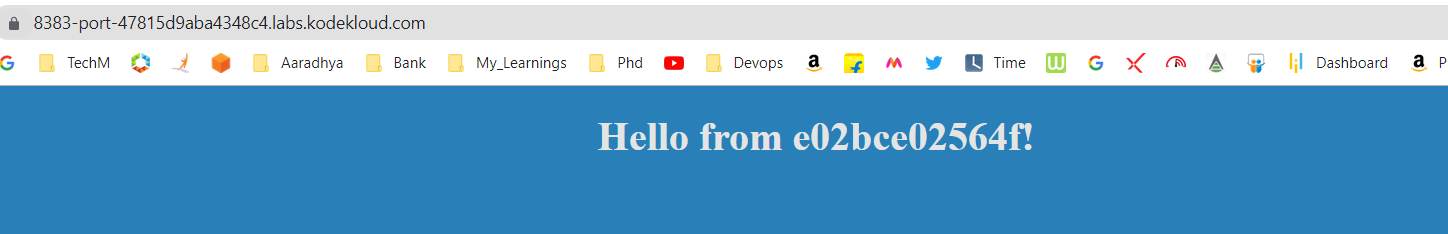
using docker build -t webapp-color:lite .



* Run an instance of the new image webapp-color:lite and publish port 8080 on the container to 8383 on the host.

Command: docker run -p 8383:8080 webapp-color:lite  
To run the container in the background, add the -d flag.

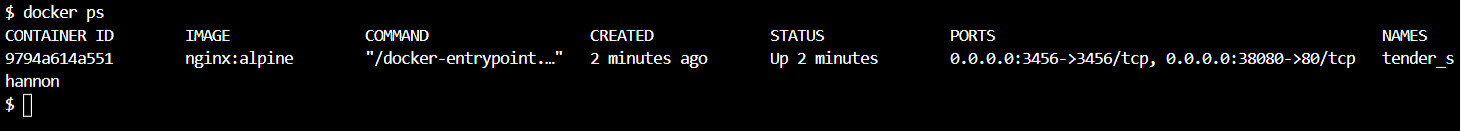




**Docke**r **RUN Commands**

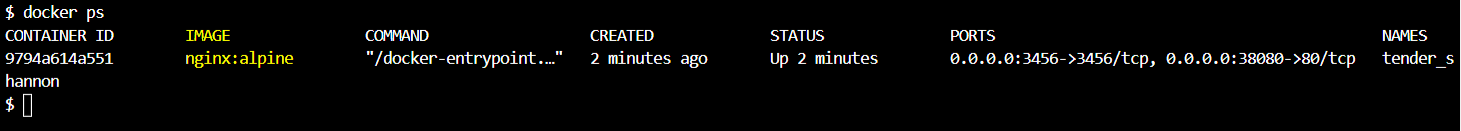
* Let us first inspect the environment. How many containers are running on this host?

Run the command docker ps and count the number of running containers.



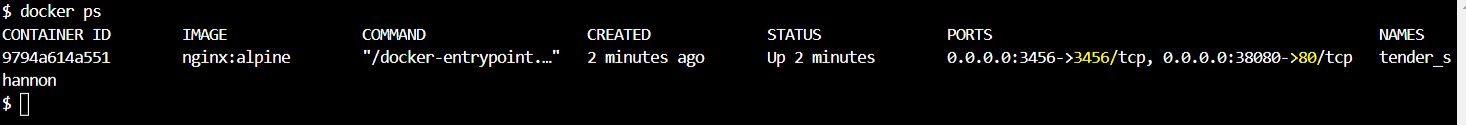
* What is the image used by the container?

Run the command docker ps and look under the IMAGE column.



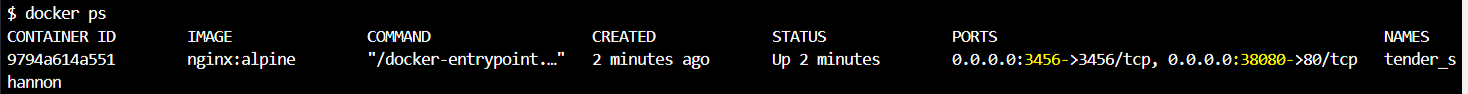
* How many ports are published on this container?

Run the command docker ps and look under the PORTS column.



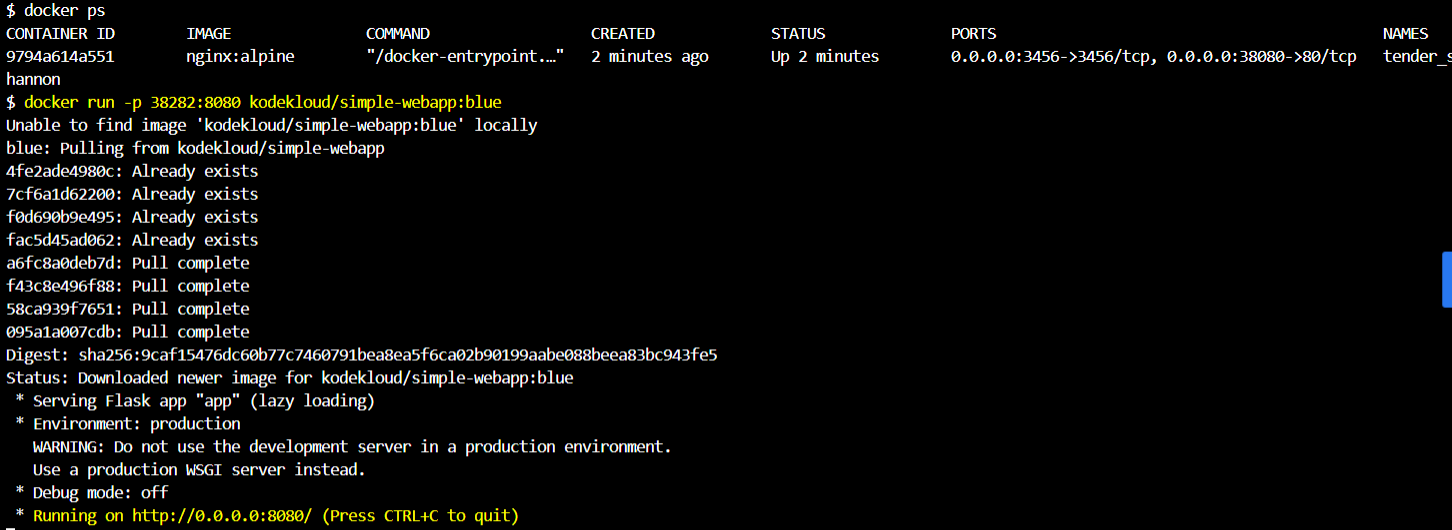
* Which of the below ports are published on Host?

Run the command docker ps and look under the PORTS column. Ports on the left(before ->) are published on the host.



* Run an instance of kodekloud/simple-webapp with a tag blue and map port 8080 on the container to 38282 on the host.

Run the command: docker run -p 38282:8080 kodekloud/simple-webapp:blue  
You can run this container in the background after adding the -d flag.

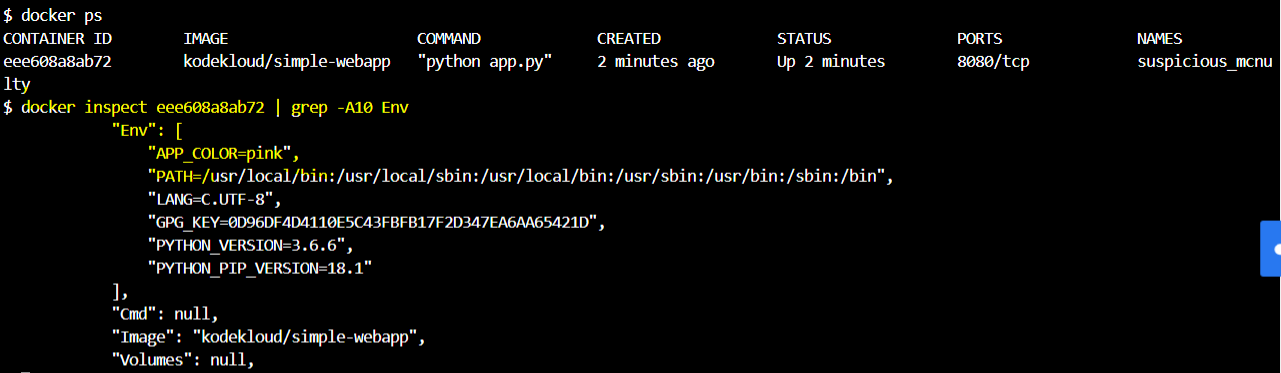


**ENVIRNOMENT VARIABLES**

* Inspect the environment variables set on the running container and identify the value set to the APP\_COLOR variable.

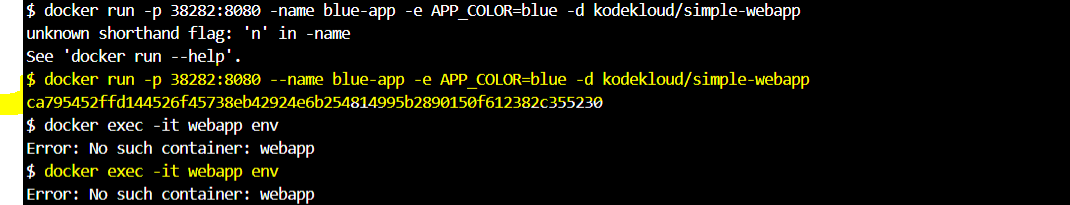
To know the running containers, run the command docker ps and identify the running container name or container id.  
After knowing the container name or id, run the command docker inspect container-name/container-id and look for the values under Env.

Run this command to get the env fields from the inspect command: docker inspect <container-name> | grep -A 10 Env  
Replace container-name with the correct one.

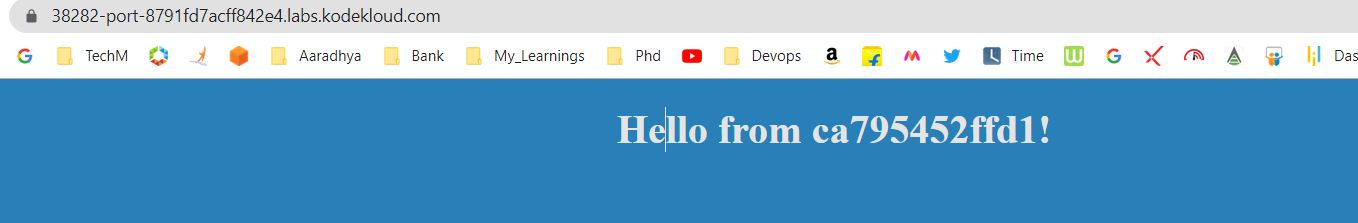


* Run a container named blue-app using image kodekloud/simple-webapp and set the environment variable APP\_COLOR to blue. Make the application available on port 38282 on the host. The application listens on port 8080.

Run the command : docker run -p 38282:8080 --name blue-app -e APP\_COLOR=blue -d kodekloud/simple-webapp  
To know the env field from within a webapp container, run docker exec -it webapp env



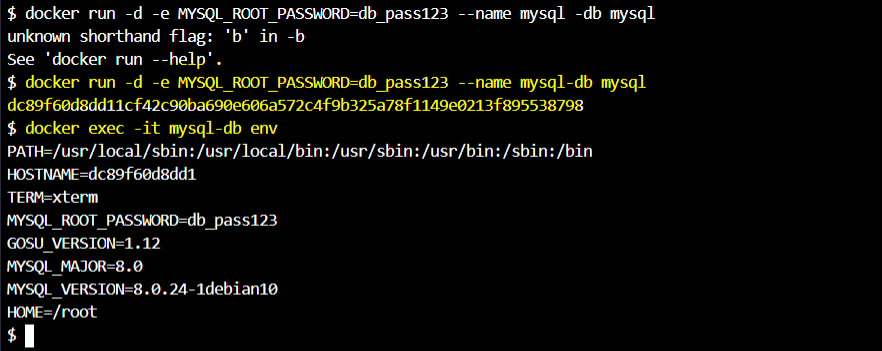
* View the application by clicking the link HOST:38282 above your terminal and ensure it has the right color.



* Deploy a mysql database using the mysql image and name it mysql-db.

Set the database password to use db\_pass123. Lookup the mysql image on Docker Hub and identify the correct environment variable to use for setting the root password.

Run the command: docker run -d -e MYSQL\_ROOT\_PASSWORD=db\_pass123 --name mysql-db mysql  
To know the env field from within a mysql-db container, run docker exec -it mysql-db env



**COMMAND & ENTRYPOINT**

* Dockerfiles for a few commonly used Docker Images are given in the /root (current) directory. Inspect them and try to answer the following questions.



* What is the ENTRYPOINT configured on the mysql image?

Open the file /root/Dockerfile-mysql and inspect the ENTRYPOINT instruction.

Run: cat Dockerfile-mysql | grep ENTRYPOINT



* What is the CMD configured on the wordpress image?

Open the file /root/Dockerfile-wordpress and inspect the CMD instruction.

Run: cat Dockerfile-wordpress | grep CMD



* What is the final command run at startup when the wordpress image is run. Consider both ENTRYPOINT and CMD instructions

What is the final command run at startup when the wordpress image is run. Consider both ENTRYPOINT and CMD instructions



* What is the command run at startup when the ubuntu image is run?

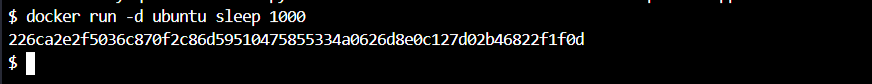
Open the file /root/ Dockerfile-ubuntu and inspect the CMD instruction

Run: cat Dockerfile-ubuntu | grep CMD



* Run an instance of the ubuntu image to run the sleep 1000 command at startup. Run it in detached mode.

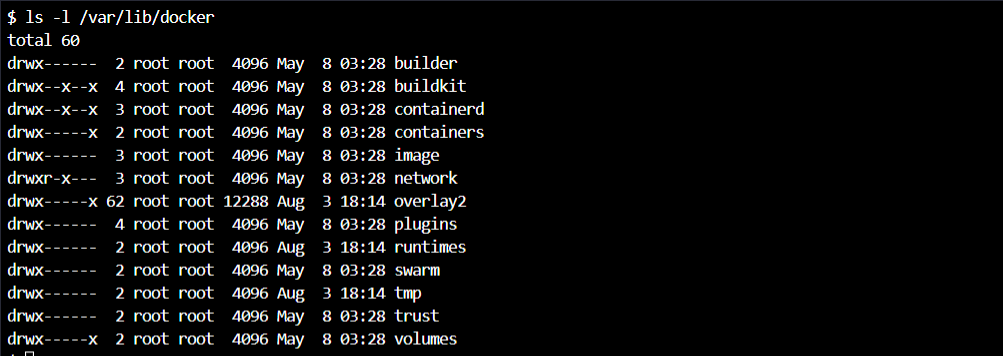
Command: docker run -d ubuntu sleep 1000



**DOCKER STORAGE**

* What location are the files related to the docker containers and images stored?

Location is /var/lib/docker.



* What directory under /var/lib/docker are the files related to the container alpine-3 image stored?

82514e8e27d272d4ce744b11e86a68d6d1d3be7f5022a0a8424a0f190bc8c580

It's available in the location cd /var/lib/docker/containers/.

Run docker ps -a command and match the container id of alpine-3 with the directory name.



* Run a mysql container named mysql-db using the mysql image. Set database password to db\_pass123

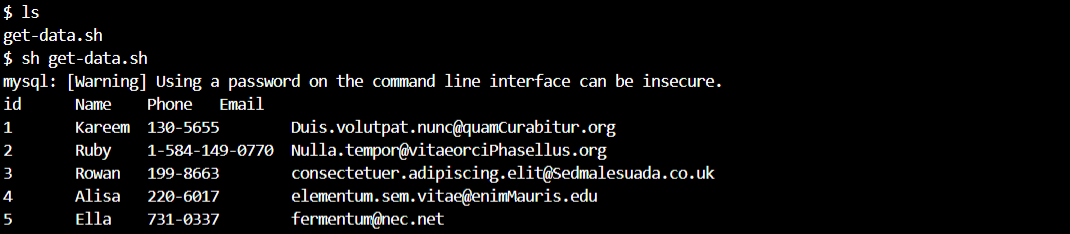
Note: Remember to run it in the detached mode.

Run the command: docker run -d --name mysql-db -e MYSQL\_ROOT\_PASSWORD=db\_pass123 mysql



* We have just written some data into the database. To view the information we wrote, run the get-data.sh script available in the /root directory. How many customers data have been written to the database?

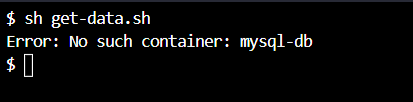
Command: sh get-data.sh



* The database crashed. Are you able to view the data now?

Use the same command to try and view data. Try to find the container.

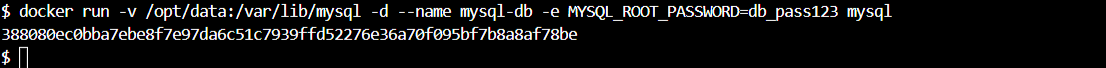
No



* Run a mysql container again, but this time map a volume to the container so that the data stored by the container is stored at /opt/data on the host.

Use the same name : mysql-db and same password: db\_pass123 as before. Mysql stores data at /var/lib/mysql inside the container.

Run the command: docker run -v /opt/data:/var/lib/mysql -d --name mysql-db -e MYSQL\_ROOT\_PASSWORD=db\_pass123 mysql



* We have now re-written data again. Run the get-data.sh script to ensure data is present.

Command: sh get-data.sh

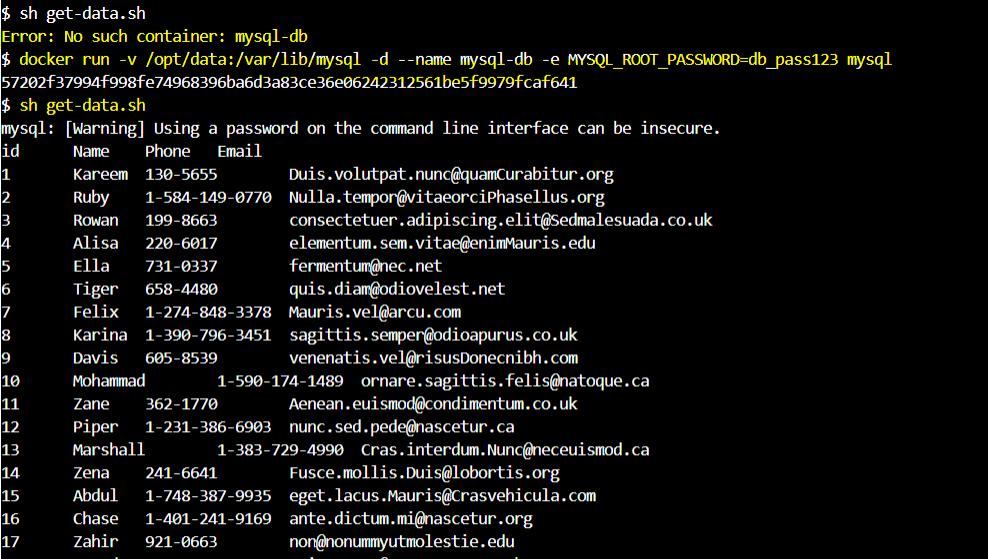


* Disaster strikes.. again! And the database crashed again. But this time we have the data stored at /opt/data directory. Re-deploy a new mysql instance using the same options as before.

Just run the same command as before. Here it is for your convenience: docker run -v /opt/data:/var/lib/mysql -d --name mysql-db -e MYSQL\_ROOT\_PASSWORD=db\_pass123 mysql

* Fetch data and make sure it is present.

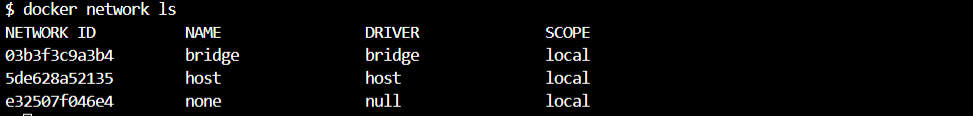
command: sh get-data.sh



**DOCKER NETWORK**

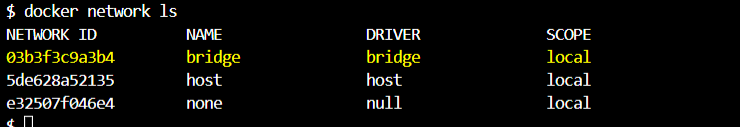
* Explore the current setup and identify the number of networks that exist on this system.

docker network ls



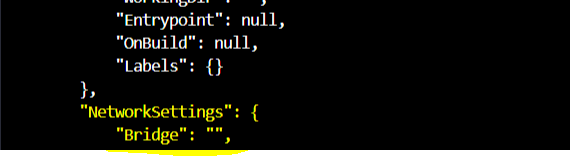
* What is the ID associated with the bridge network?

fb9c1a74a36514f6f2d767eee7be024a0ffd40b6cf208886de42ee8003f5a6a603b3f3c9a3b4823b43cf36ee4e6597020ac100ae1de6216b67023ec1541f4407



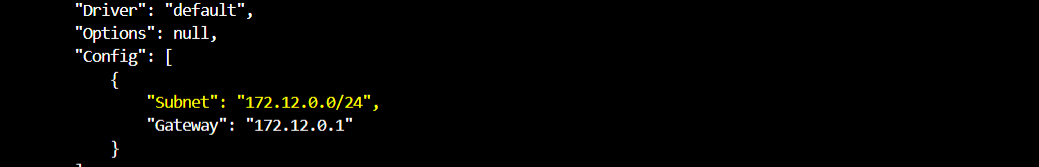
* We just ran a container named alpine-1. Identify the network it is attached to.

Run the command docker inspect alpine-1 and look under the Networks section.



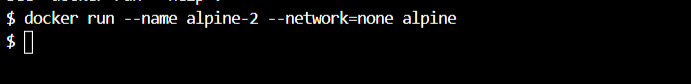
* What is the subnet configured on bridge network?

Run the command docker network inspect bridge



* Run a container named alpine-2 using the alpine image and attach it to the none network.

Run the command: docker run --name alpine-2 --network=none alpine



* Create a new network named wp-mysql-network using the bridge driver. Allocate subnet 182.18.0.1/24. Configure Gateway 182.18.0.1

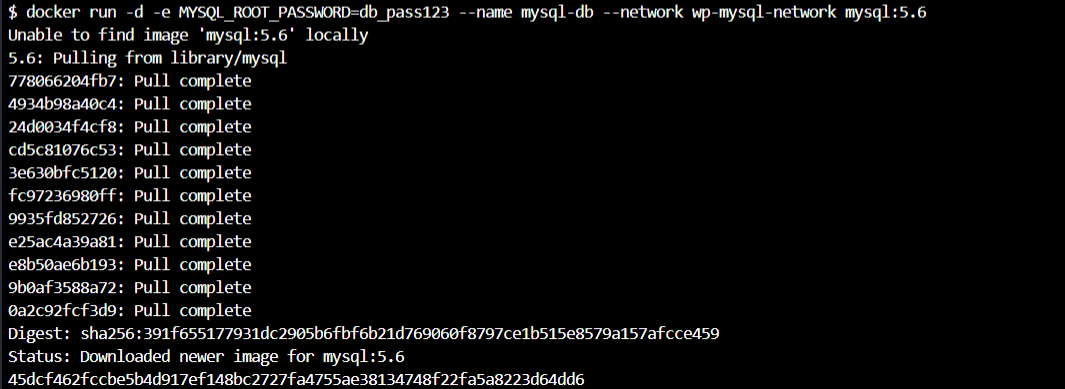
Run the command: docker network create --driver bridge --subnet 182.18.0.1/24 --gateway 182.18.0.1 wp-mysql-network  
Inspect the created network by docker network inspect wp-mysql-network



* Deploy a mysql database using the mysql:5.6 image and name it mysql-db. Attach it to the newly created network wp-mysql-network

Set the database password to use db\_pass123. The environment variable to set is MYSQL\_ROOT\_PASSWORD.

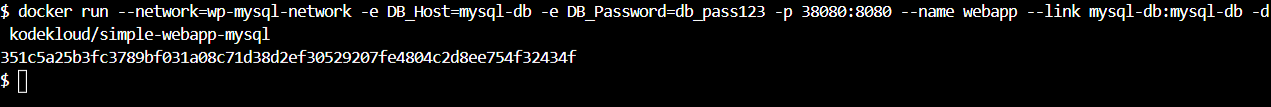
Run the command: docker run -d -e MYSQL\_ROOT\_PASSWORD=db\_pass123 --name mysql-db --network wp-mysql-network mysql:5.6



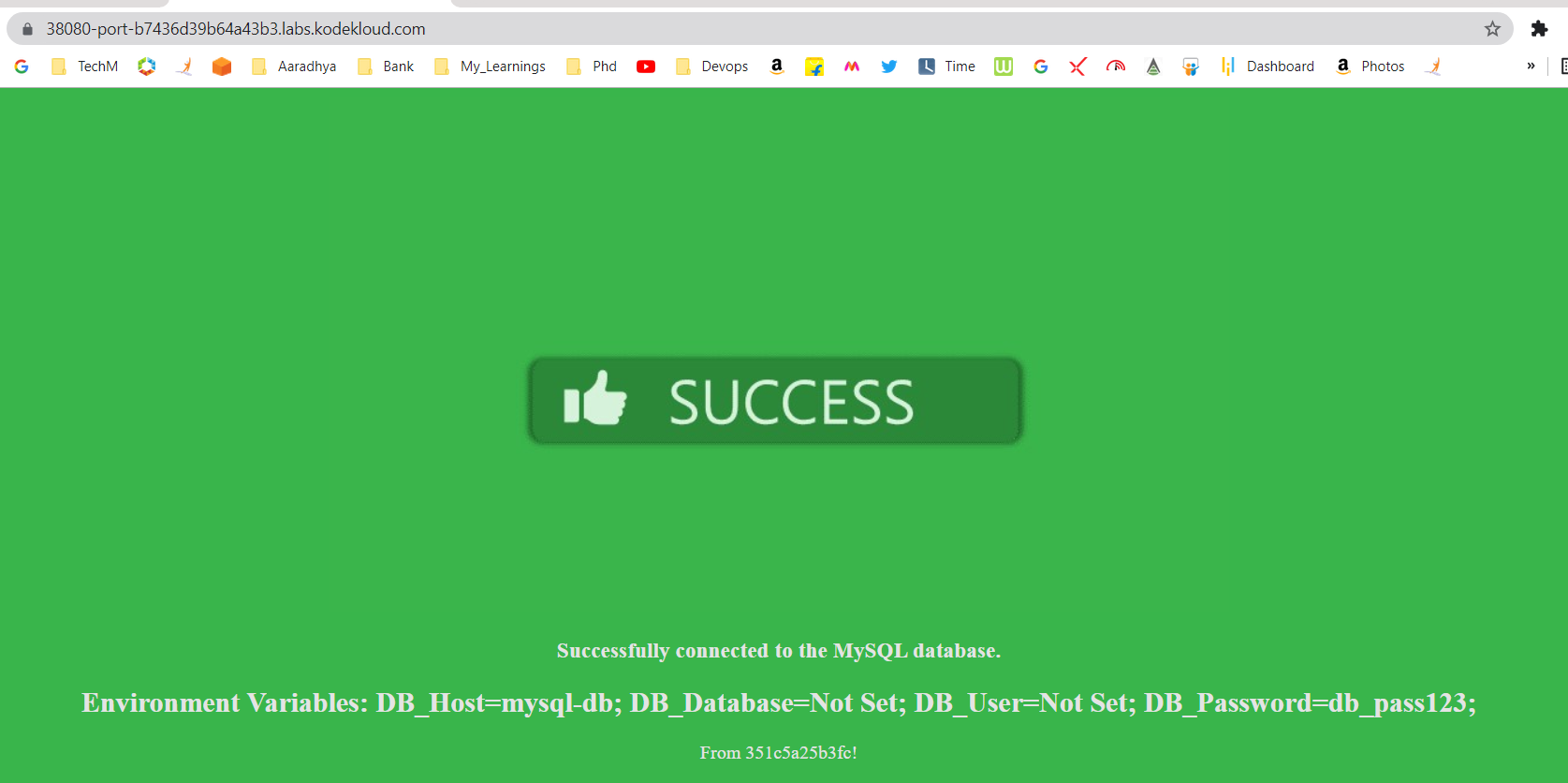
* Deploy a web application named webapp using the kodekloud/simple-webapp-mysql image. Expose the port to 38080 on the host.  
    
  The application makes use of two environment variable:  
  1: DB\_Host with the value mysql-db.  
  2: DB\_Password with the value db\_pass123.  
  Make sure to attach it to the newly created network called wp-mysql-network.

Also make sure to link the MySQL and the webapp container.

Run the command: docker run --network=wp-mysql-network -e DB\_Host=mysql-db -e DB\_Password=db\_pass123 -p 38080:8080 --name webapp --link mysql-db:mysql-db -d kodekloud/simple-webapp-mysql



* If you are successfull, you should be able to view the application by clicking on the HOST:38080 at the top of your terminal. You should see a green success message.



**DOCKER COMPOSE**

* First create a redis database container called redis, image redis:alpine.

if you are unsure, check the hints section for the exact commands.

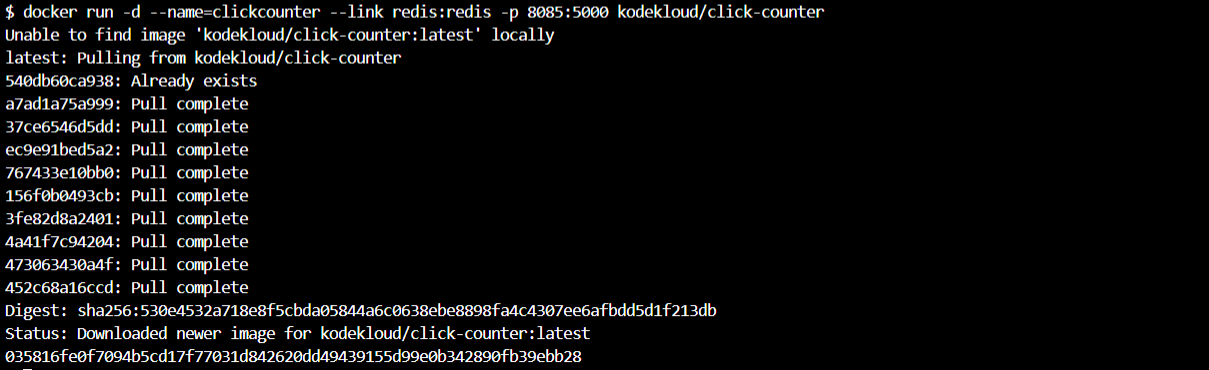
Run the command: docker run --name redis -d redis:alpine



* Next, create a simple container called clickcounter with the image kodekloud/click-counter, link it to the redis container that we created in the previous task and then expose it on the host port 8085

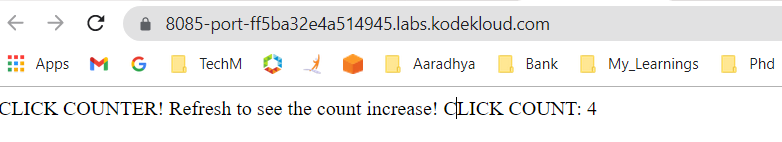
The clickcounter app run on port 5000.  
if you are unsure, check the hints section for the exact commands.

Run the command: docker run -d --name=clickcounter --link redis:redis -p 8085:5000 kodekloud/click-counter



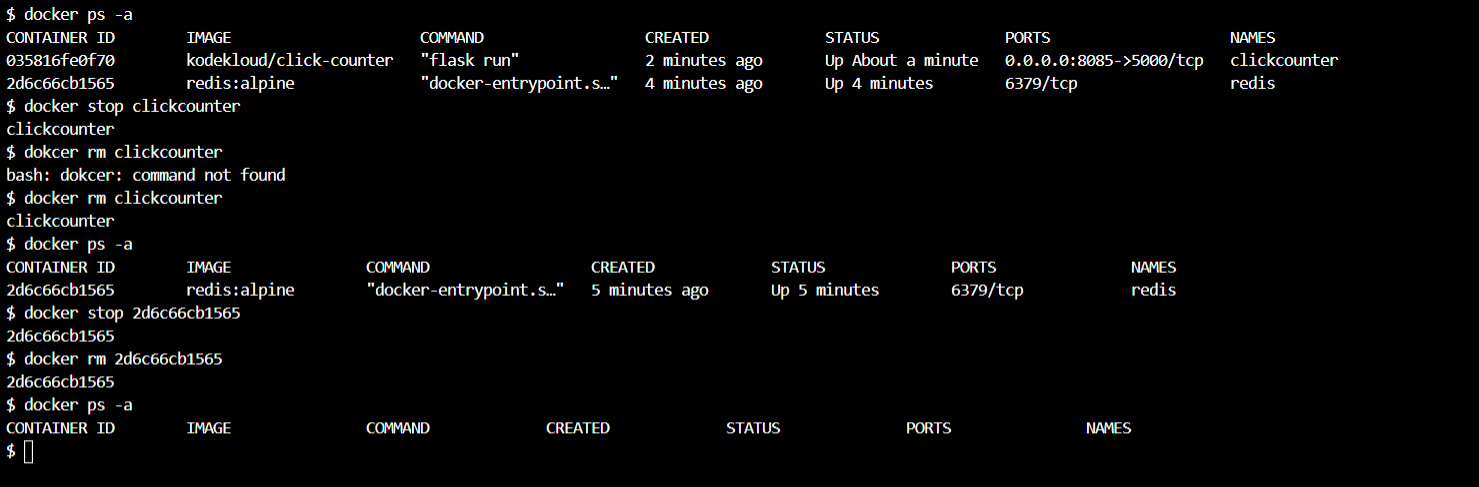
* You can now access this application ysing the Click-Counter tab above the terminal.

Refresh the page and see the count increase.



* Let's clean up the actions carried out in previous steps. Delete the redis and the clickcounter containers.

To stop the containers: docker stop <CONTAINER-NAME>  
To delete the containers: docker rm <CONTAINER-NAME>



* Create a docker-compose.yml file under the directory /root/clickcounter. Once done, run docker-compose up.

The compose file should have exact specification of the clickcounter and redis container that we created earlier.

Use the below compose file:

services:

redis:

image: redis:alpine

clickcounter:

image: kodekloud/click-counter

ports:

- 8085:5000

version: '3.0'

Then run a docker-compose up -d command. To run containers in a background, added -d flag.

