Jenkins Continue –

10 – 11:30 Recap

How to create Node and add into pipeline

Exercise
1st Assignment -

- Freestyle Job –
- SCM – Git with credentials
- Build Maven / Ant project.
- Post Build Action: Execute Test Case
- Post Build Action: Based on Parameters call Next Job – Pipeline Job

2nd Assignment –

- Create Pipeline with Script
- SCM – Git config – Add credentials into system configuration.
- Git Clone
- Create file and add + commit to Git.

3rd Assignment –

- Create Jenkins file for above assignment and run job with Jenkins files.

Creating Nodes-

Docker Commands

Docker ---version

Docker search

Docker pull Hello-World

Docker images

Docker run Hello-World

Docker ps

Docker ps – a

Docker logs -t container nama

TABLE OF CONTENTS

- Docker Container Commands
  - List all the Running Containers
  - List all the Containers (irrespective of the state)
  - List all the Running Containers with the File Size
  - List the IDs of the Running Containers
  - List the IDs of all the Containers (irrespective of the state)
  - Filter container list
  - Creating a new Container using Docker Image
  - Creating a new Container using Docker Image with some fixed name
  - Start a Docker Container
  - Stop a running Docker Container
  - Restart a Docker container
  - Pause a running Container
  - Unpause a paused Container
  - Docker Run command
    - Docker Run command in Foreground and Detached Modes

- Delete the container on the exited state
- Run the container in daemon mode
- Run Docker Container with a name using the run command
- Listing Processes running in a Docker Container
- Map ports of a Docker Container
- Rename a Docker Container
- Run the Docker Container in an Interactive Mode
- Get Inside the Running Container (Literally!)
- Start a Docker Container and keep it running
- Copy a File from a Container to a Host
- Copy a File from the Host to the Docker Container
- Remove a specific Docker Container
- Remove a Docker Container after it exits
- Delete all the Stopped Containers
- Delete all the Docker Containers
- Create a Docker image from a Docker Container
- Run command inside the Docker Container
- Set Environment Variable in a Docker Container
- Set Environment Variable in a Docker Container using a File

• Docker Image Commands

- List all the Docker Images
- List the Docker Image Ids
- List all the Docker Images (including dangling images)
- Build a Docker Image
- Build Docker Images with a different tag
- Build a Docker Image using a custom named Dockerfile
- Build a Docker Image from a Dockerfile that is not in the Current Directory
- Show History of a Docker Image
- Rename an existing Docker Image

- Remove Docker images
- Force delete a Docker Image
- Unused Docker Images
- Dangling Docker Images
- List Dangling Docker Images
- Remove all the Dangling Docker Images
- Remove all the Dangling and Unused Docker Images
- Login to Docker
- Push a Docker Image to the Docker Registry
- Download a Docker Image from the registry

• Docker logs

- Get Logs of the Docker container
- Monitor the Docker Container Logs
- Get the last 2 lines of the Container Logs

• Docker Network Commands

- List all the Networks
- Create a Network
- Connect a Docker Container to a Network:
- Connect a Docker Container to a Network on Start
- Disconnect a Docker container from a Network:
- Remove a Network
- Show Information about one or more Networks
- Get the IP Address of the running Docker Container

• Docker Volumes

- Create Docker Volume
- List Docker Volumes
- Mounting Docker Volume using the -v Flag
- Mounting Docker Volume using the --mount Flag
- Get Details about a Docker Volume

- Remove a Docker Volume
- Volume Mount using bind-mount
- Creating Bind Mount Volume using the --mount flag
- Docker System-wide Commands
- Docker Info
- Docker Stats of the running Container
- Docker Stats of all the Containers
- Show the Docker Version
- Get Detailed Info about an Object (Container, Image, Volume, etc)
- Get the Summary of Docker Usage
- Clean your Docker system
- Conclusion

Docker is a containerization technology to build, ship and run applications inside containers. We can create different Docker containers for packaging different software as it uses virtualization at the level of an operating system.

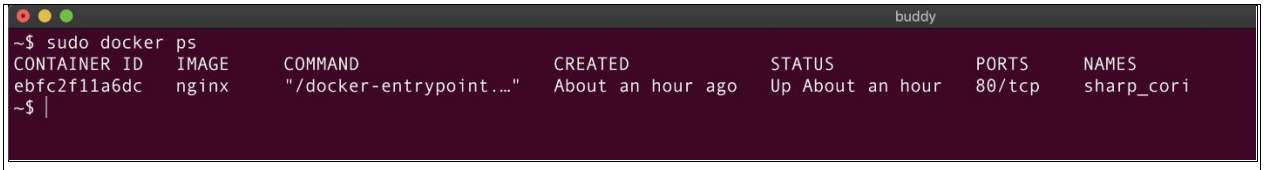
This article is a comprehensive list of Docker commands that you can refer to while building applications and packaging them in Docker containers.

Let's first look at some of the fundamental container commands!

Docker Container Commands

List all the Running Containers

```
docker ps
```



We can also use `docker container ps` or `docker container ls` to list all the running containers.

There is no difference between docker ps and docker container ps commands with respect to the result of their execution.

docker ps is an old syntax and is supported for backward compatibility.

The docker container ls command is an appropriate version of the command compared to ps as ls is a shortcut for list.

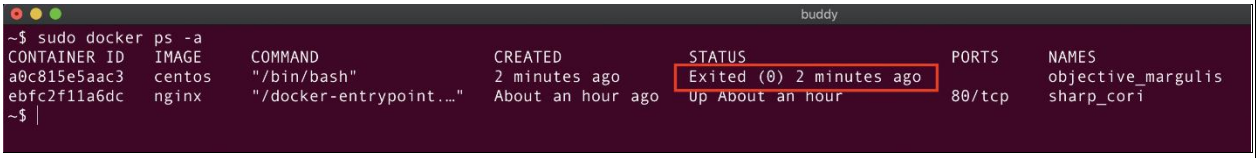
List all the Containers (irrespective of the state)

```
docker ps -a
```

Please note: -a is the short form for --all and they both can be used interchangeably.

```
docker ps -all
```

This command is used for listing all the containers (active and inactive).

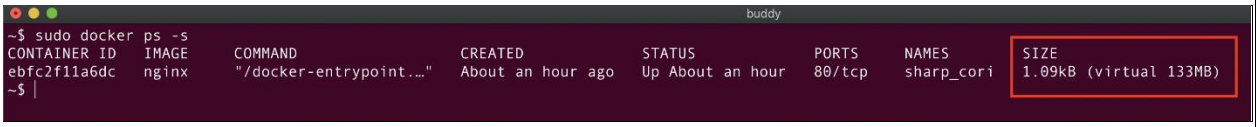


As can be seen from the screenshot above, the container objective_margulis is not running while the container sharp_cori is up since the last hour.

List all the Running Containers with the File Size

```
docker ps -s
docker container ls -s
```

-s is the short form --size



This command adds SIZE column to the output.

As it can be seen from the screenshot above, 1.09kB is the disk space used by the container (writable layer). In simple words, the value in the SIZE column represents the size of the data that is written by the container in its writable layer.

This text virtual 133MB represents the amount of disk space used by the image of this container.

List the IDs of the Running Containers

```
docker ps -q
docker container ls -q
```

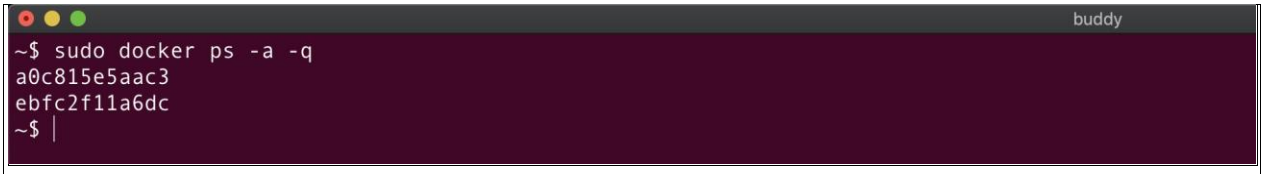
-q is the short form for --quiet. This command modifies the docker ps output and displays only the Ids of the running containers.

List the IDs of all the Containers (irrespective of the state)

```
docker ps -a -q
```

We can also write the above command by combining a and q as:

```
docker ps -aq
```



Filter container list

We can filter the output of docker ps or docker ps -a commands using the --filter option as:

```
docker ps -f name=un
```

-f is the short form for --filter

```
buddy
~$ sudo docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED             STATUS             PORTS     NAMES
eb9f50c01730   centos    "/bin/bash"              2 minutes ago       Up 2 minutes                 unruffled_leavitt
ebfc2f11a6dc   nginx     "/docker-entrypoint.…"   About an hour ago   Up About an hour   80/tcp    sharp_cori
~$
~$ sudo docker ps -f name=un
CONTAINER ID   IMAGE     COMMAND       CREATED         STATUS         PORTS     NAMES
eb9f50c01730   centos    "/bin/bash"   2 minutes ago   Up 2 minutes             unruffled_leavitt
~$
```

In the above screenshot, the command filters the containers and only displays the ones whose name starts with un

Similarly, we can add -f option with the docker ps -a command:

```
docker ps -a -f name=ar
```

```
buddy
~$ sudo docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED             STATUS                     PORTS     NAMES
eb9f50c01730   centos    "/bin/bash"              4 minutes ago       Up 4 minutes                         unruffled_leavitt
a0c815e5aac3   centos    "/bin/bash"              21 minutes ago      Exited (0) 21 minutes ago            objective_margulis
ebfc2f11a6dc   nginx     "/docker-entrypoint.…"   About an hour ago   Up About an hour           80/tcp    sharp_cori
~$
~$ sudo docker ps -a -f name=ar
CONTAINER ID   IMAGE     COMMAND                  CREATED             STATUS                     PORTS     NAMES
a0c815e5aac3   centos    "/bin/bash"              21 minutes ago      Exited (0) 21 minutes ago            objective_margulis
ebfc2f11a6dc   nginx     "/docker-entrypoint.…"   About an hour ago   Up About an hour           80/tcp    sharp_cori
~$
```

We can also filter the containers on the basis of the status as:

```
docker ps -a -f status=running
```

```
buddy
~$ sudo docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED             STATUS                     PORTS     NAMES
eb9f50c01730   centos    "/bin/bash"              11 minutes ago      Up 11 minutes                        unruffled_leavitt
a0c815e5aac3   centos    "/bin/bash"              28 minutes ago      Exited (0) 28 minutes ago            objective_margulis
ebfc2f11a6dc   nginx     "/docker-entrypoint.…"   2 hours ago         Up 2 hours                 80/tcp    sharp_cori
~$
~$ sudo docker ps -a -f status=running
CONTAINER ID   IMAGE     COMMAND                  CREATED             STATUS          PORTS     NAMES
eb9f50c01730   centos    "/bin/bash"              11 minutes ago      Up 11 minutes             unruffled_leavitt
ebfc2f11a6dc   nginx     "/docker-entrypoint.…"   2 hours ago         Up 2 hours      80/tcp    sharp_cori
~$
```

As seen in the above screenshot, the filter command filters the containers and displays only the running ones in the list!

You can check the other filter options available from the official Docker documentation

Creating a new Container using Dockdocer Image

```
docker create <image_name>
```

The docker create command is used to create a new container using a Docker image. It does not run the container but just adds a writable layer on top of the Docker image. We'll have to run the docker start command to run the created container.

As docker create command interacts with the containerobject, we can also use the below command:

```
docker container create <image_name>
```

Let's create a container using an nginx Docker image:

```
docker create nginx
```

```
buddy
~$ sudo docker create nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
b4d181a07f80: Pull complete
edb81c9bc1f5: Pull complete
b21fed559b9f: Pull complete
03e6a2452751: Pull complete
b82f7f888feb: Pull complete
5430e98eba64: Pull complete
Digest: sha256:47ae43cdfc7064d28800bc42e79a429540c7c80168e8c8952778c0d5af1c09db
Status: Downloaded newer image for nginx:latest
45f7e9d947e0e99f5d57e22b8d3f3d567502c089ab31ab3dd06639bce6a8a709
~$
~$
```

Perfect! The container is created. Let's verify that using the docker ps command:

```
buddy
~$ sudo docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
~$
~$ sudo docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS    PORTS     NAMES
45f7e9d947e0   nginx     "/docker-entrypoint.…"   52 seconds ago   Created             trusting_bouman
~$
```

The status of the container is Created as expected!

Please note the name of the nginx container trusting_bouman is some random string and would be different on your system.

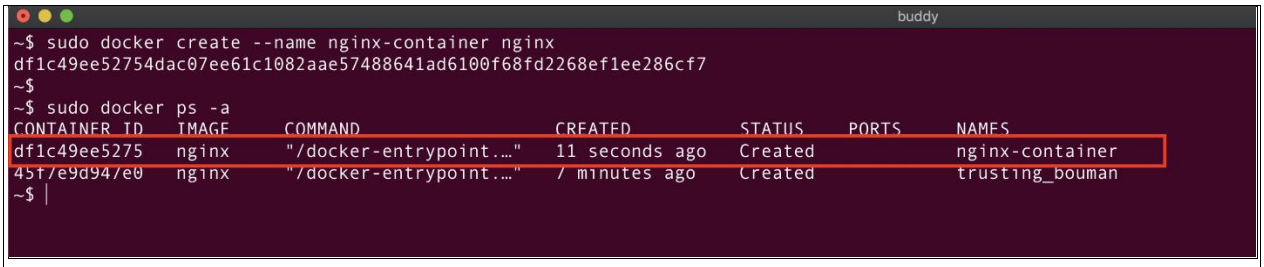
We can also create a Docker container with some fixed names. Let's do that right away!

Creating a new Container using Docker Image with some fixed name

```
docker create --name <container_name> <image_name>
docker container create --name <container_name> <image_name>
```

Let's create a container named nginx-container using nginx image:

```
docker create --name nginx-container nginx
```



Neat! The container nginx-container is created!

Start a Docker Container

We can run an already created command using the below command:

```
docker start <container_id or container_name>
```

OR

```
docker container start <container_id or container_name>
```

We can use the docker start command either using the container ID or name.

Let's start the nginx-container:

```
docker start nginx-container
```

We can also start it as:

```
docker start df1c49ee5275
```

As seen from the above screenshot, nginx-container is created and the docker ps command is used to verify the status of the container.

Stop a running Docker Container

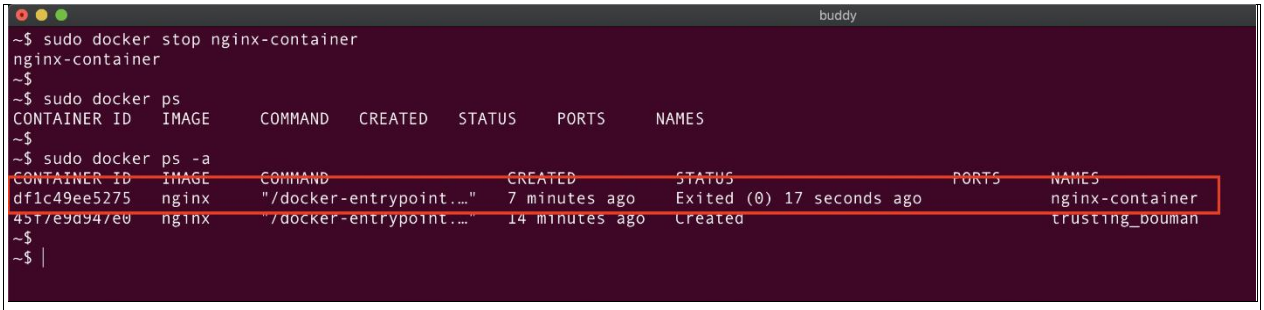
```
docker stop <container_id or container_name>
```

OR

```
docker container <container_id or container_name>
```

Here's the command for stopping the nginx-container:

```
docker stop nginx-container
```



As seen in the above screenshot, the container nginx-container is exited 17 seconds ago. This container won't be listed in the docker ps command.

Restart a Docker container

```
docker restart <container_id or container_name>
```

OR

```
docker container restart <container_id or container_name>
```

The nginx-container is now restarted and is up for the last 8 seconds.

Pause a running Container

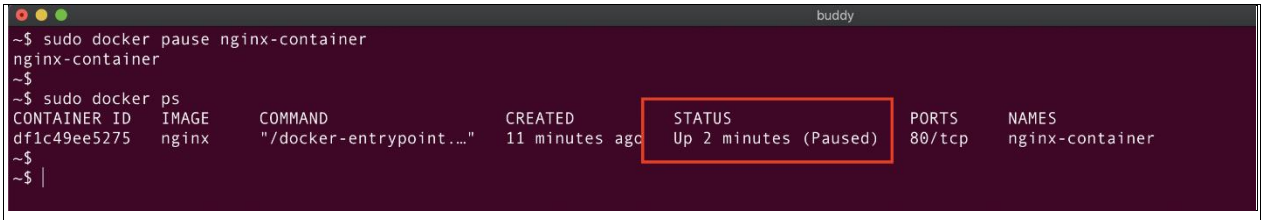
```
docker pause <container_id or container_name>
```

OR

```
docker container pause <container_id or container_name>
```

Let's try to pause the nginx-container:

```
docker pause nginx-container
```



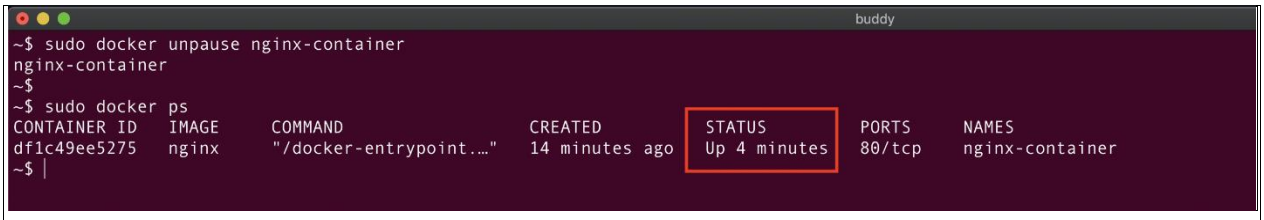
Unpause a paused Container

To again run the paused container, we can use the below command:

```
docker unpause <container_id or container_name>
docker container unpause <container_id or container_name>
```

To unpause the nginx-container, we can use the below command

```
docker unpause nginx-container
```



Docker Run command

As implied from the name, this command is used to run the container!

It is a combination of the create and the start commands. This command creates the container and starts it in one go!

The docker create or docker container create command creates the container and to run the container, we have to use the docker start command.

It rarely happens that we will create the container and run it later. Generally, in real-world cases, we would create and run the container in one go using the docker run command.

Here's how we can run a Docker container:

```
docker run <image_name>
docker container run <image_name>
```

If the Docker image is locally available, Docker will run the container using that image otherwise it would download the image from the remote repository.

Docker Run command in Foreground and Detached Modes

The Docker container can run in two modes:

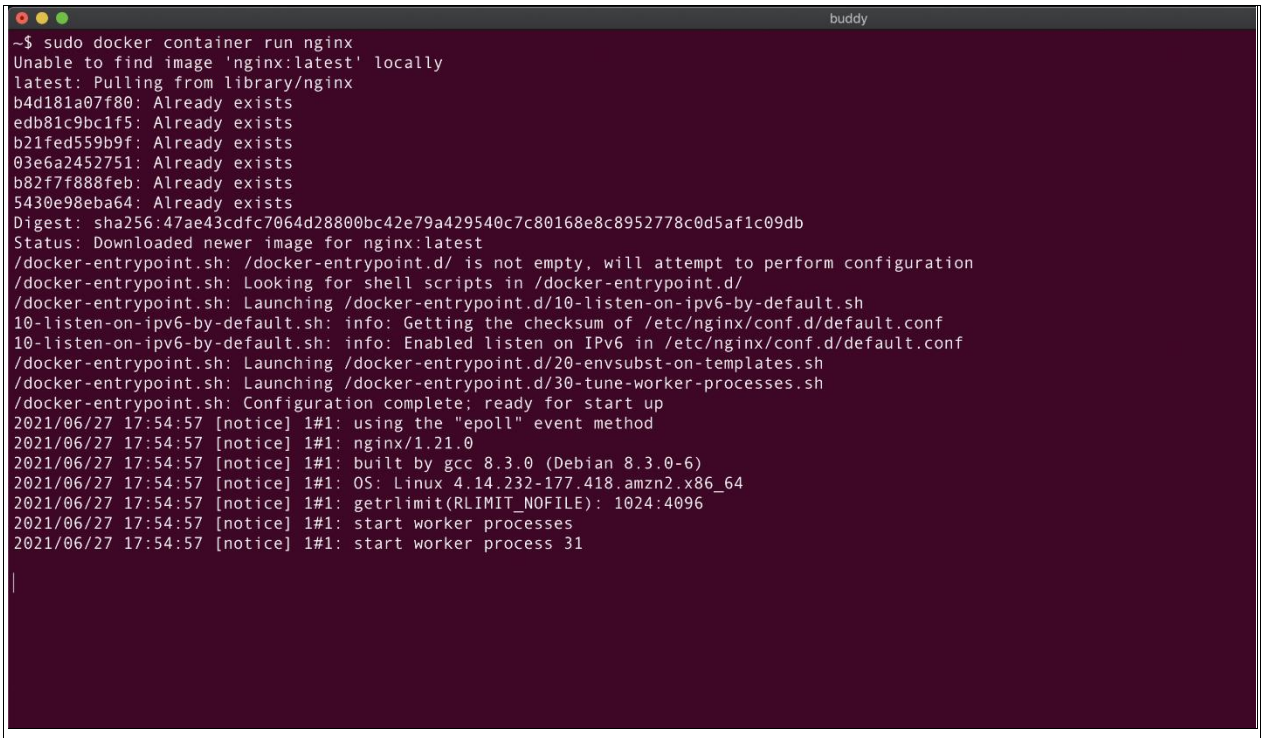
1. Foreground mode
2. Background or detached mode

Docker runs the container in the foreground mode by default. In this mode, Docker starts the root process in the container in the foreground and attaches the standard *input(stdin)*, *output(stdout)*, and *error(stderr)* of the process to the terminal session.

In the foreground mode, you cannot execute any other command on the terminal session until the container is running. This is the same as running a Linux process in the foreground mode.

Let's create and run a container in the foreground mode using the nginx Docker image:

```
docker container run nginx
```

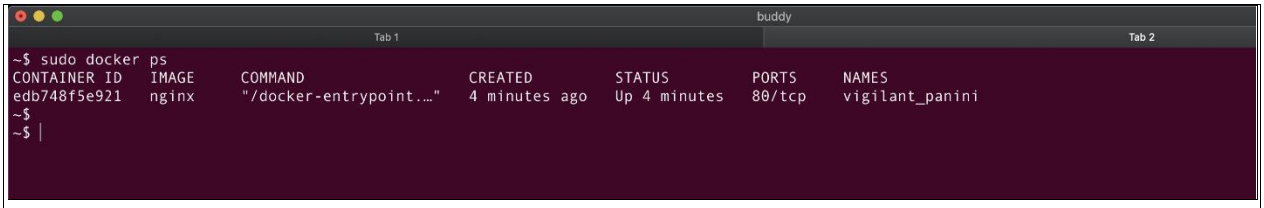


The above screenshot shows the output of the Nginx container on the terminal. The Nginx process is running in the foreground and hence this terminal session cannot be used for executing other commands or performing any other operation.

If you end the terminal session by closing the terminal tab or by using the exit command, the container will die automatically and the Nginx process would stop running.

Please open a new terminal tab to verify if the container is running:

```
docker ps
```



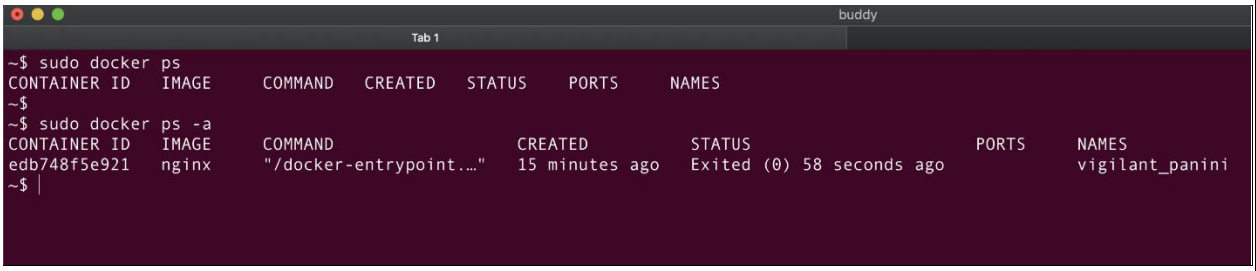
The run command worked as expected - it created the container and also started the container.

The long-running processes such as Nginx are run in the background mode and so are not dependent on the terminal session.

If we end the terminal session by closing the terminal tab or by pressing CTRL + C, the container will die automatically.

Let's stop the nginx process using CTRL+C!

Now if you execute the docker ps command, you can see that the container is in *exited* stated but it is not deleted. This is because when the container exits or when we stop the container, the filesystem of the container continues to persist on the host machine.



Delete the container on the exited state

If you want to also delete the container after it is exited, you can start the container in the foreground using this command:

```
docker run --rm nginx
```

This command would start the nginx container in the foreground mode. If you kill the terminal session, it would stop the container and delete it!

--rm option removes the filesystem of the container once it is stopped or when the container exits automatically.

Run the container in daemon mode

To run the container in the background or in daemon mode, we can use the -d(--detach) option.

```
docker run -d <image_name>
```

-d option will run the container in the background mode and print the container ID

Let's run a container with nginx image in the background mode:

```
buddy
~$ sudo docker run -d nginx
cd213b8859362f0dec03c0081b47256363caf1b1348d8da6f1e7bdecc4302401
~$
```

From the above image, we can see that the container process is not attached to the terminal session and the container is running in the background mode.

Let's verify if the container is in the running state using the docker ps command:

```
buddy
~$ sudo docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS     NAMES
cd213b885936   nginx     "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   80/tcp    busy_heisenberg
~$
~$
```

Run Docker Container with a name using the run command

We can use --name option to assign a name to the container as shown below:

```
docker container run -d --name <container_name> <image_name>
```

Let's create a container named nginx-container from nginx image:

```
docker container run -d --name nginx-container nginx
```

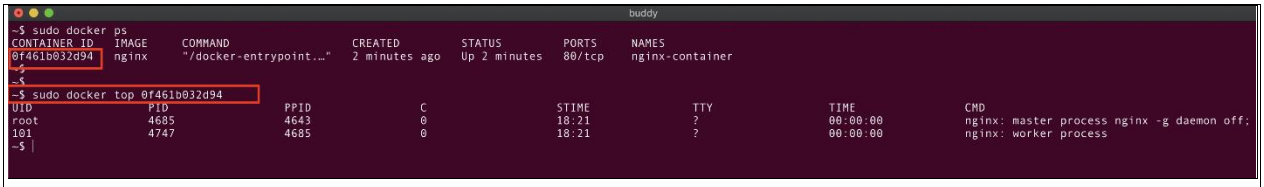
```
buddy
~$ sudo docker container run -d --name nginx-container nginx
0f461b032d944c31eae5a06599be6ce09e299e7dac05b9158c0f1424769b3cbf
~$
~$ sudo docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS     NAMES
0f461b032d94   nginx     "/docker-entrypoint.…"   5 seconds ago    Up 3 seconds    80/tcp    nginx-container
~$
```

Listing Processes running in a Docker Container

```
docker top <container_name or container_id>
```

OR

```
docker container top <container_name or container_id>
```



Here, we can see that inside nginx-container there are two processes running with the Ids 4685 and 4747.

Map ports of a Docker Container

Docker exposes the same port on the host machine that is exposed by the container.

We know that nginx process listens on port 80. Hence, a container running an nginx process will expose port 80 on the container.

Docker will map port 80 of the nginx container with port 80 of the host machine. The nginx container will be accessible to the outside world from port 80 of the host machine.

We know that one port cannot be used by multiple processes.

But what if we want to run multiple nginx containers on the same host machine?

We can use port mappings provided by Docker to achieve this!

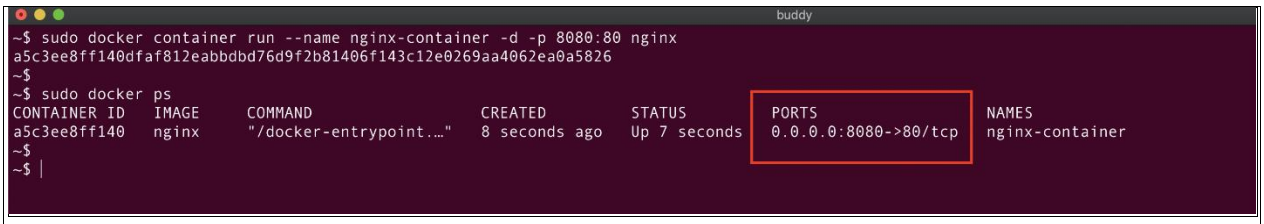
The port mappings allow us to map a port on the container with a different port on the host machine.

We can map the ports of the container while creating the container as shown below:

```
docker container run --name <container_name? -d -p <host_post>:<container_port>
<image_name>
```

To expose nginx container (port 80) on port 8080 of the host machine, we can use the below command:

```
docker container run --name nginx-container -d -p 8080:80 nginx
```



The port mappings are also indicated in the `docker ps` command as shown in the above screenshot!

As the container is mapped to the port `8080` of the host machine, we can access the Nginx container on `8080` using the curl command. Let's try that out!

```
buddy
~$ curl http://127.0.0.1:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
~$
```

This works as expected!

Rename a Docker Container

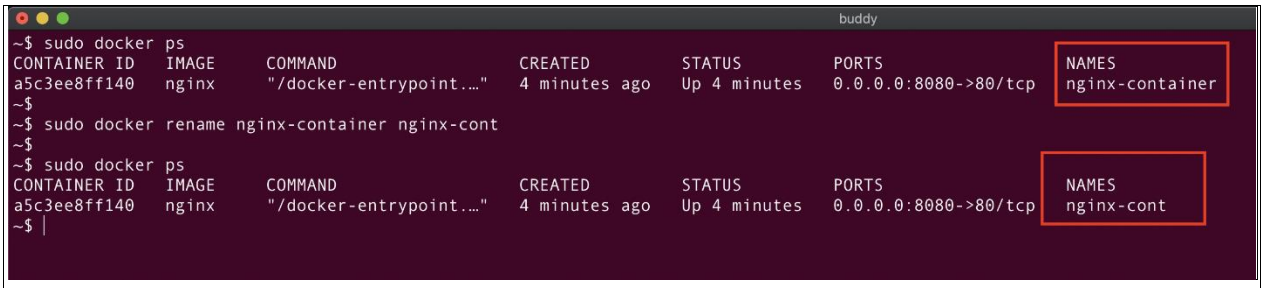
```
docker rename <old_name> <new_name>
```

OR

```
docker container rename <old_name> <new_name>
```

Let's rename `nginx-container` to `nginx-cont`:

```
docker rename nginx-container nginx-cont
```



Run the Docker Container in an Interactive Mode

When we run the container in an interactive mode, Docker will attach the stdin(standard input) of the container to the terminal.

This will give us entry inside the container and now we can run any command inside the container.

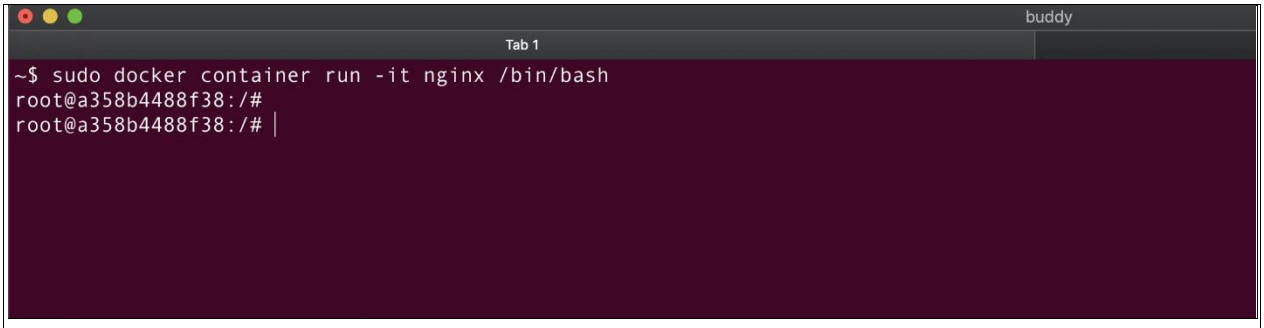
Let's understand this with an example:

```
docker container run -it <image_name> /bin/bash
```

This command simply means that we want to start the bash shell inside the container.

Let's create a container from an Nginx image and run the bash command inside the container:

```
docker container run -it nginx /bin/bash
```

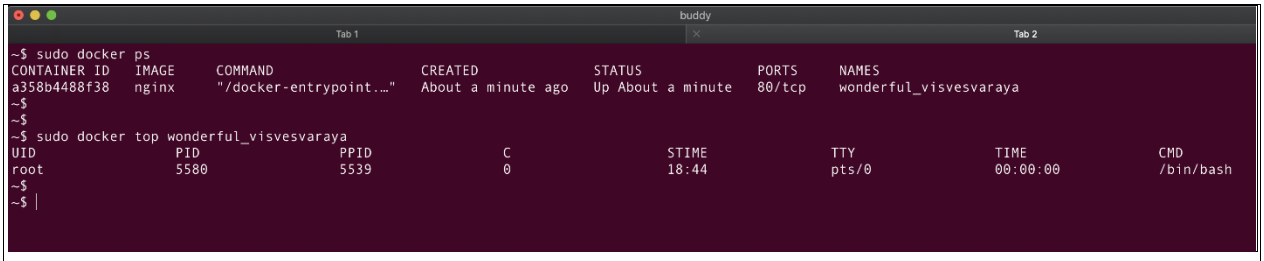


Super! We're inside the container!

We can now execute any command inside the container.

Please open another tab and let's list down the processes running inside the container:

```
docker top wonderful_visvesvaraya
```



As seen in the above screenshot, only one process is running inside the nginx container. This is the same process that we have opened in Tab 1.

When we had checked the processes running inside the nginx-container earlier, we saw two processes (nginx master and worker) inside the nginx container but now there is just one process - *bash shell*.

This is because we passed /bin/bash command to the docker run command. This command overrides the dockerfile CMD or Entrypoint commands and so the Nginx processes were not started inside the Nginx container.

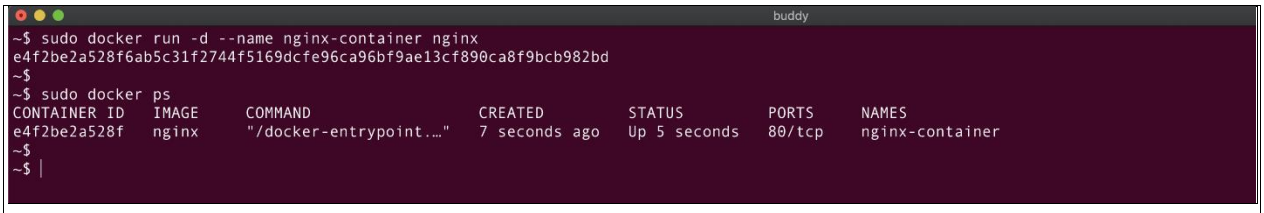
Get Inside the Running Container (Literally!)

If you have a container that is already running and you want to go inside that container, here's a command that will take you to the container world:

```
docker exec -it <contaner_id or container_name> /bin/bash
```

Let's start the nginx container in the daemon mode:

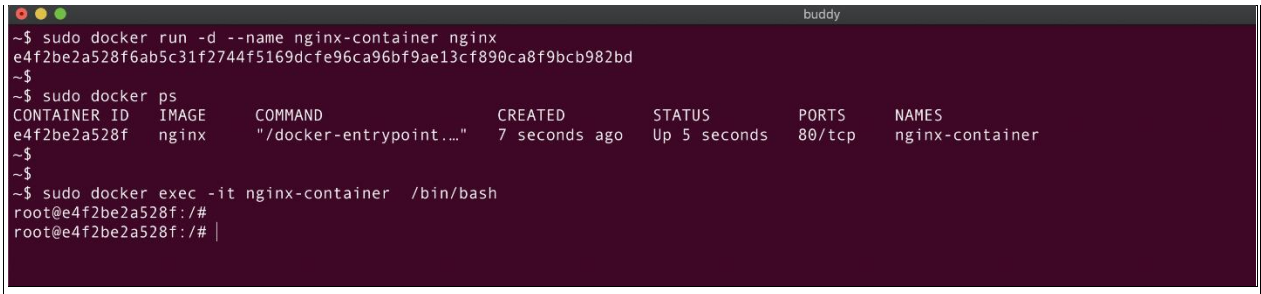
```
docker run -d --name nginx-container nginx
```



We can go inside this container as:

```
docker exec -it nginx-container  /bin/bash
```

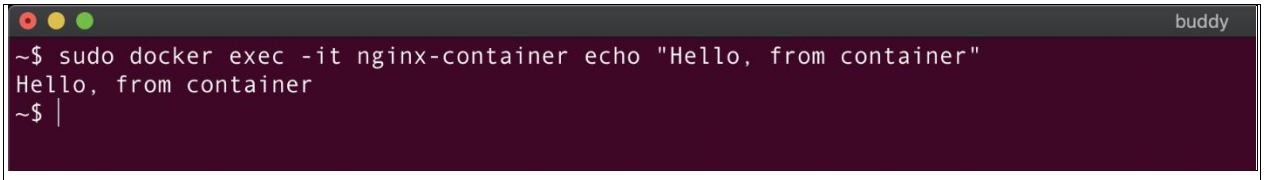
The docker exec command executes the /bin/bash command and starts a bash shell session inside the container as shown in the below screenshot:



Sweet! This is quite a handy command and is used more often to get into the running container and perform some operations.

Note: Apart from `/bin/bash`, we can pass any command that we want to execute inside the container. Here's a quick example:

```
docker exec -it nginx-container echo "Hello, from container"
```



Start a Docker Container and keep it running

The life of a container depends on the root process inside the container. So far we have run `nginx` process inside the containers.

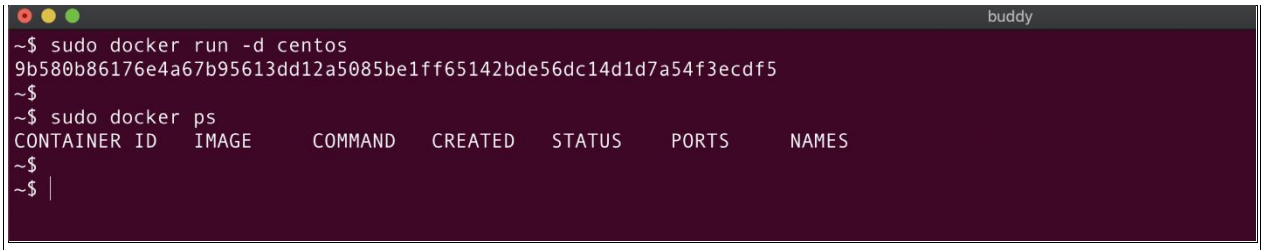
`nginx` is a long and continuous running process and hence the Nginx container continues to run and does not die!

If we run a process that is short-lived inside the container, it will die once that process dies!

For example, if we run a container using the `centos` Docker image, it will die as soon as the process inside the `centos` dies.

`centos` has a default command `bash`. The `bash` command runs and dies immediately.

Hence, the `centos` container also dies immediately as shown below:



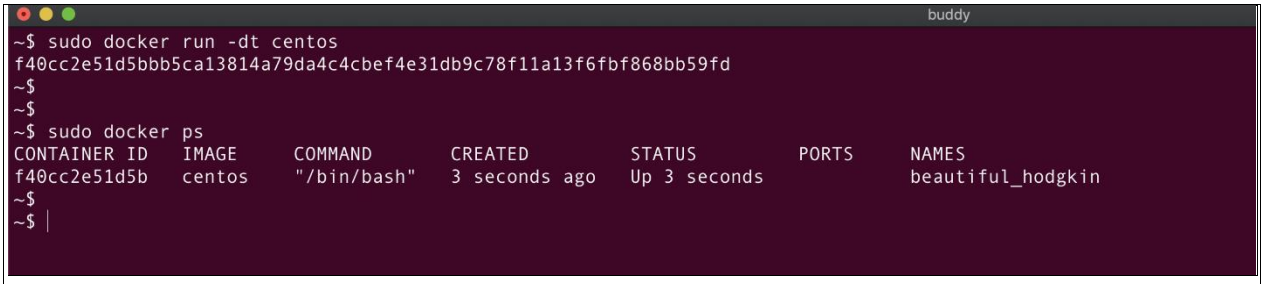
To continue running the container that has a short-lived process, we can use the below command:

```
docker run -dt <image_name>
```

-d will run the ubuntu container in the background and -t option allocates a "pseudo-tty"

To continuously run the container created from centos image, we can use the below command:

```
docker run -dt centos
```



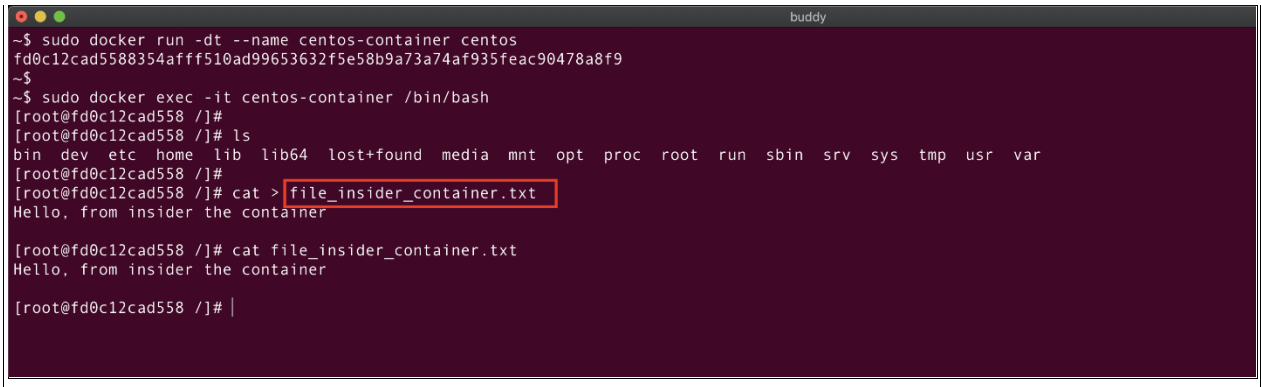
Copy a File from a Container to a Host

The Docker container and the host file systems are isolated from each other. We cannot use the normal cp or copy command to copy content from the container to the host and vice-versa.

To copy content from the container to the host machine, we can use the below command:

```
docker cp <container_id or container_name>:<source_file_path> <destination_path>
```

Let's understand this with the help of an example:

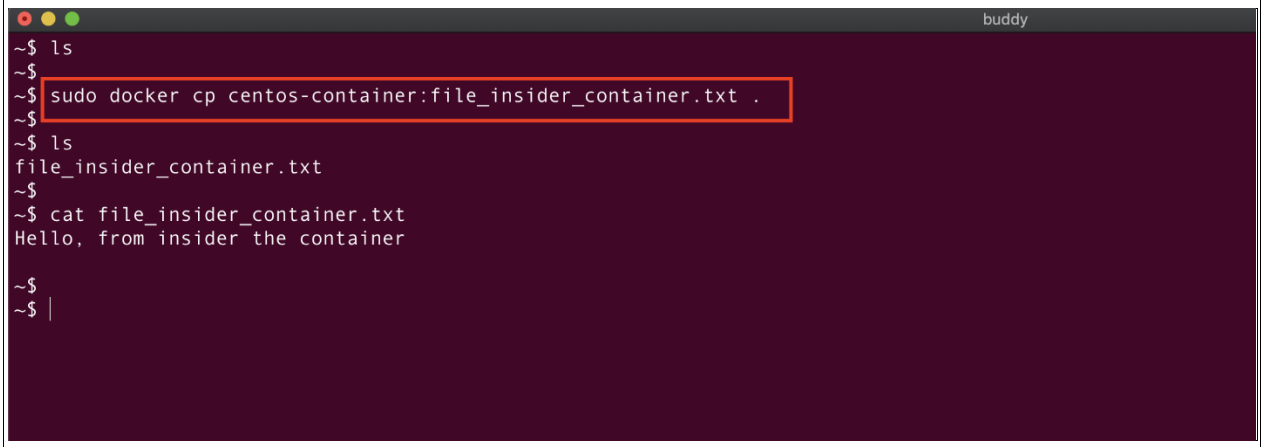


As seen in the above screenshot, we have created a file file_insider_container.txt inside the centos-container.

Let's copy this file from the container to the host machine:

```
docker cp centos-container:file_insider_container.txt .
```

The . in the above command signifies that the destination path is the current location on the host machine.



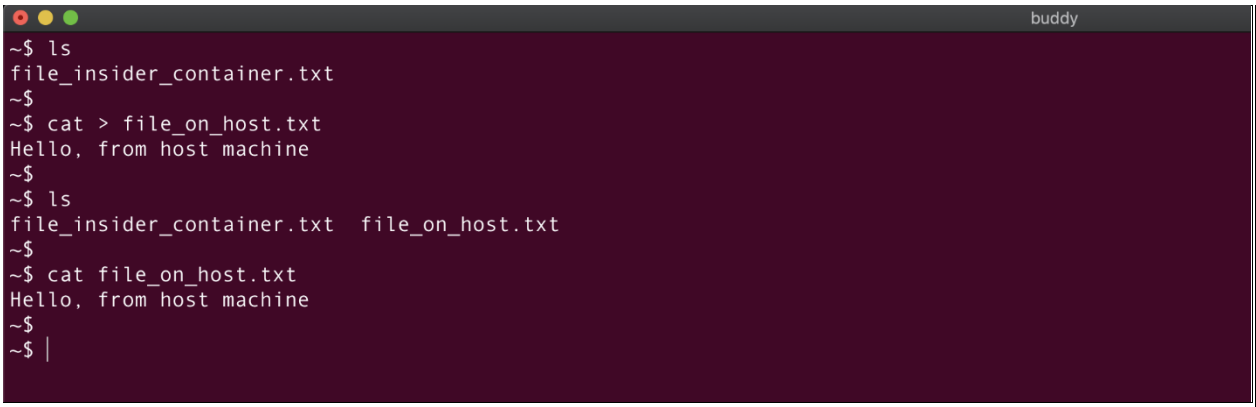
We can see that file_inside_container.txt has been copied on the current directory of the host machine.

Copy a File from the Host to the Docker Container

We can copy a file from the host system to the Docker container as:

```
docker cp <location_of_file_on_host> <container_id or
container_name>:<file_desinaion>
```

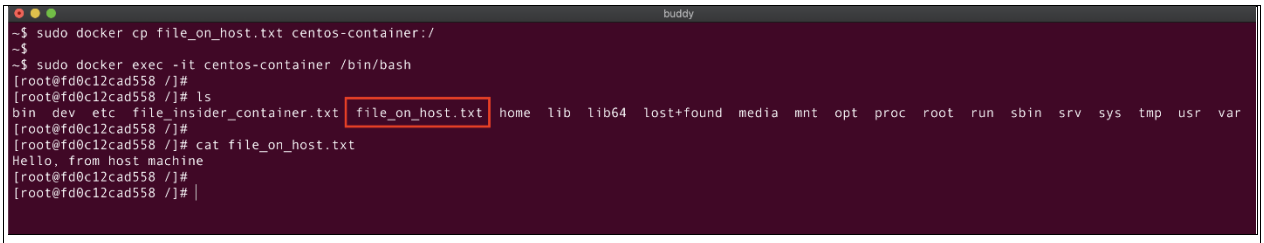
Let's create a file file_on_host.txt on the machine and copy it inside the container.



As seen in the above screenshot, we have created a file file_on_host.txt on the host machine.

Let's copy this file inside the centos-container using the below command:

```
docker cp file_on_host.txt centos-container:/
```



The file file_on_host.txt has been copied inside the centos-container container.

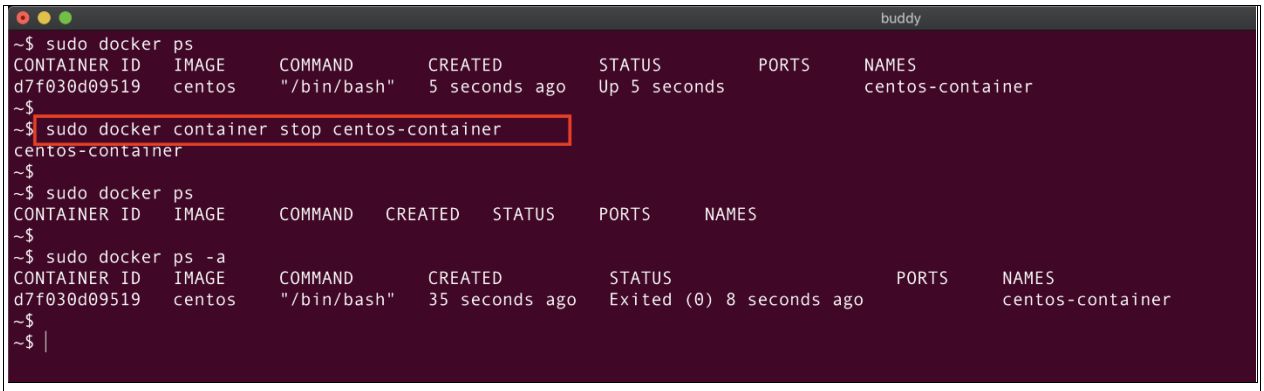
Remove a specific Docker Container

To remove a Docker container, you first have to stop the running container and then delete it:

```
docker stop <container_name or container_id>
```

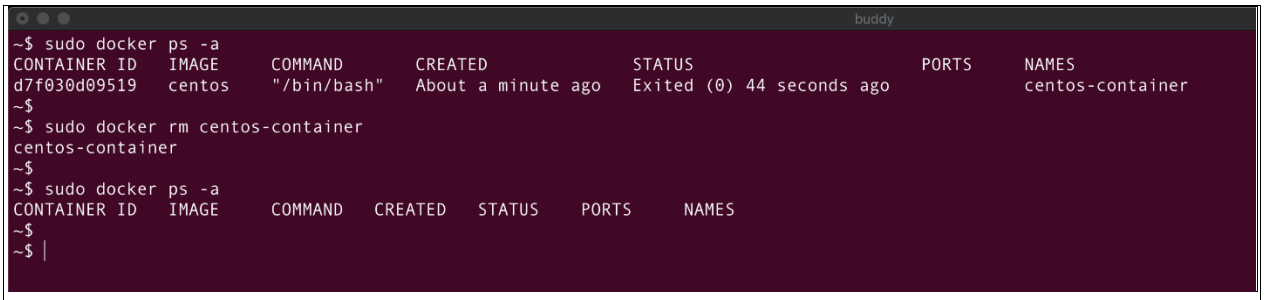
OR

```
docker container stop <container_name or container_id>
```



Once the container stops, you can remove it using the below command:

```
docker rm <container_name or container_id>
```



If you want to directly remove the container without stopping it, you can use the below command:

```
docker rm -f <container_name or container_id>
```

-f is for forcefully removing the running container

Remove a Docker Container after it exits

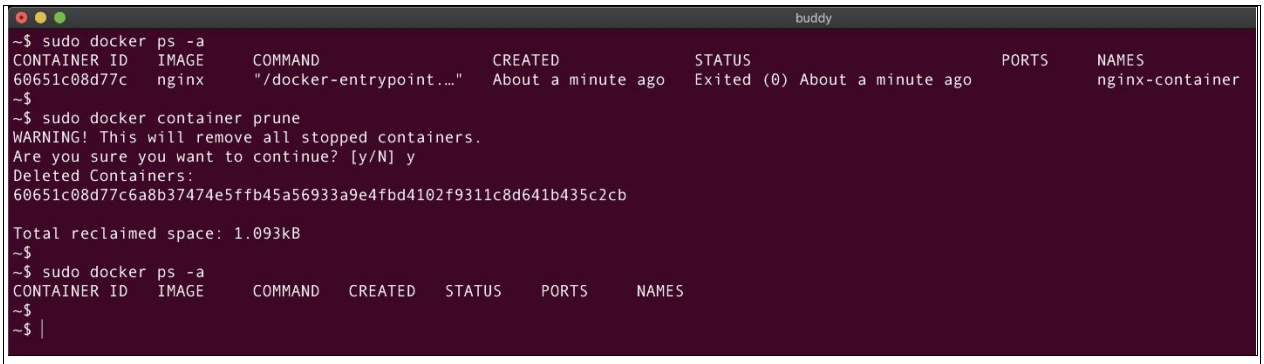
```
docker run --rm <image_name>
```

This will create and run a container that will be deleted automatically once the container stops.

Delete all the Stopped Containers

To delete all the stopped containers, we can use the below command:

```
docker container prune
```



Delete all the Docker Containers

This command is used to delete all the running as well as the stopped containers:

```
docker rm -f $(docker ps -a -q)
docker container rm -f $(docker ps -a -q)
```

We know that docker ps -a -q will list all (running as well as not running) container Ids. Once we get the Ids using the rm option, we can then remove all the containers.

Create a Docker image from a Docker Container

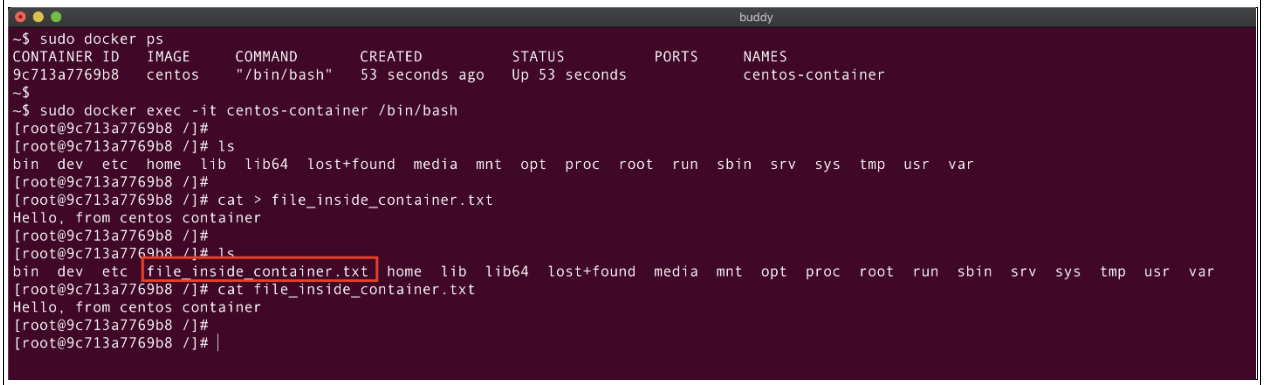
```
docker container commit <container_id or container_name> <new_image_name>
```

OR

```
docker commit <container_id or container_name> <new_image_name>
```

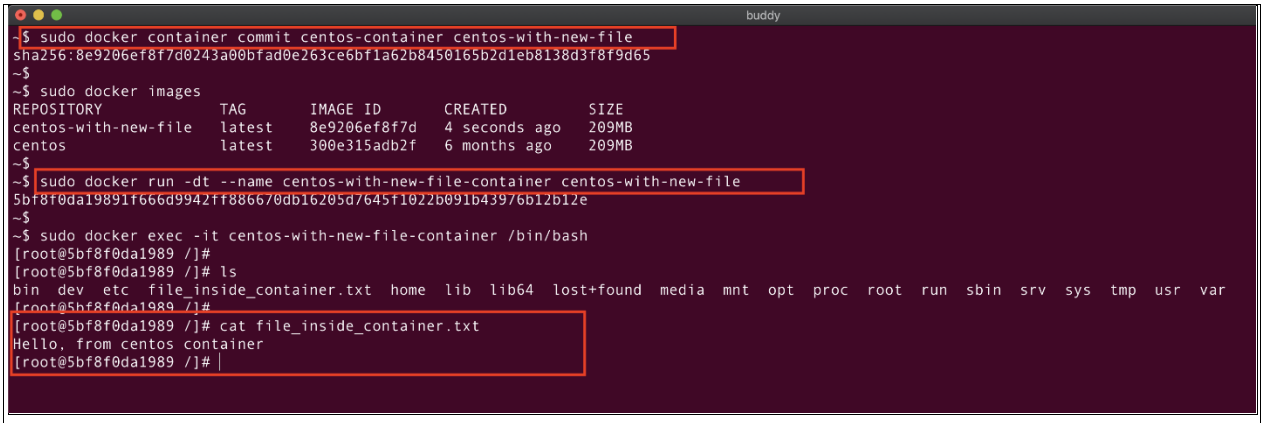
Consider the example below:

We'll create a file file_inside_container.txt inside the running container centos-container:



Let's now create an image (centos-with-new-file) from centos-container using the below command:

```
docker container commit centos-container centos-with-new-file
```



As shown in the above image, we created a new image centos-with-new-file and using this image, we created a new container centos-with-new-file-container.

As centos-with-new-file image was created from centos-container that had a file file_inside_container.txt, the containers created from centos-with-new-file image would also have file_inside_container.txt.

Run command inside the Docker Container

We have already seen that we can directly go inside the container using the docker exec command.

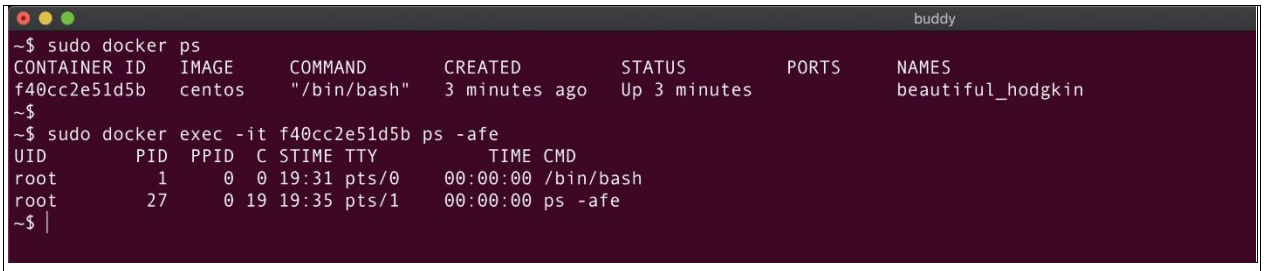
Let's see how we can execute commands inside the container without actually going inside the container.

To run any command inside the container, we can use the docker exec command as shown below:

```
docker exec -it <container_id or container_name> <command>
```

For example, to get the list of processes running inside the centos image, we can use the below command:

```
docker exec -it f40cc2e51d5b ps -afe
```



Set Environment Variable in a Docker Container

We can set environment variables inside the container environment using the below command:

```
docker run --env ENV_VAR1=value1 --env ENV_VAR1=value2 --name <container_name>
<image_name>
```

To create an environment variable with the name NAME and value Buddy inside the centos image, we can use the below command:

```
docker run -dt --env NAME=Buddy --name centos-container centos
```

-dt is added to run the centos-container in daemon mode and to prevent it from dying immediately.

To check if the environment variable has been set inside the container, we can use the below commands:

```
docker exec -it centos-container printenv
```

The docker exec command executes printenv (to print environment variable) inside the centos-container container and prints the output on the terminal.

Set Environment Variable in a Docker Container using a File

We can also set environment variables inside the container using a file.

The file file1.txt consists of key-value pairs as shown below:

```
~$ cat file1.txt
NAME=Buddy
TUTORIAL=Docker
~$
```

To create environment variables from a file we can use the below command:

```
docker run --env-file <path_to_the_file> --name <container_name> <image_name>
```

The --env-file flag takes a filename as an argument. Each line in the file should be in the format VAR=VAL.

Let's create environment variables inside the centos-container-1 container using the file file1.txt:

```
docker run -dt --env-file file1.txt --name centos-container-1 centos
```

-dt is added to run the centos-container in daemon mode and to prevent it from dying.

```
buddy
~$ sudo docker run -dt --env-file file1.txt --name centos-container-1 centos
a6474a63d8aebc3596300b4a105bf2a39d0154993332d68f299bd68969cb37d3
~$
~$ sudo docker exec -it centos-container-1 printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=a6474a63d8ae
TERM=xterm
NAME=Buddy
TUTORIAL=Docker
HOME=/root
~$
```

Docker Image Commands

In this section, we will look at the Docker image commands.

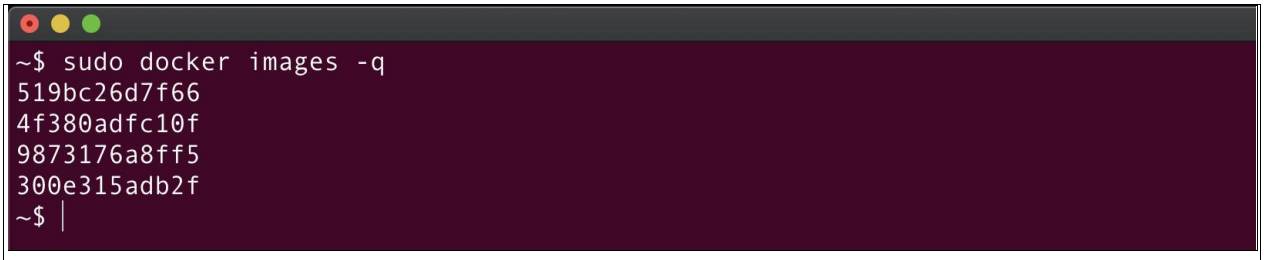
List all the Docker Images

Here's a command to list all the images that are locally stored:

```
docker images
```

List the Docker Image Ids

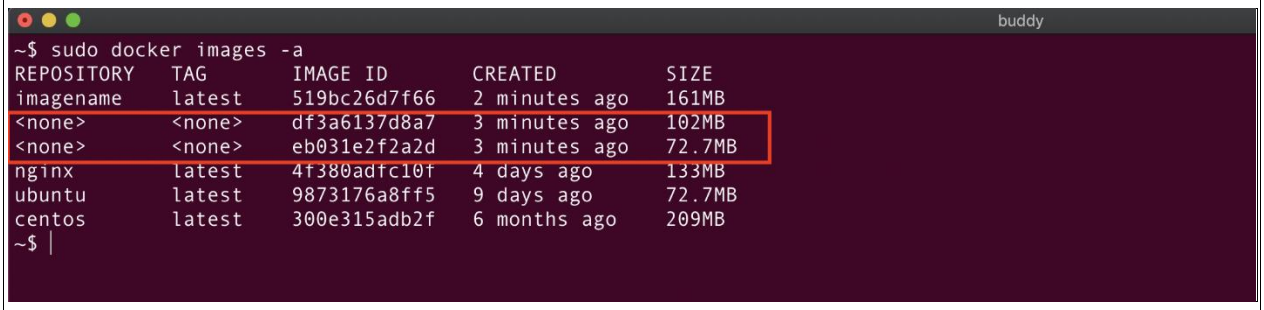
```
docker images -q
```



List all the Docker Images (including dangling images)

```
docker images -a
```

-a stands for all



Build a Docker Image

We can build a Docker image from a Dockerfile using the below command:

```
docker build -t <image_name> <context_dir>
```

Docker will try to find a file named Dockerfile inside the context_dir and it would then create a Docker image using the Dockerfile file.

Let's create a Dockerfile as shown below in the current directory:

```
buddy
~$ cat Dockerfile
FROM centos:latest
MAINTAINER buddy@gmail.com
~$
```

We can now build a Docker image centos from this Dockerfile using the below command:

```
docker build . -t centos_buddy
```

Here, dot (.) indicates that the context directory is the current directory.

```
buddy
~$ sudo docker build . -t centos_buddy
Sending build context to Docker daemon  352.8kB
Step 1/2 : FROM centos:latest
latest: Pulling from library/centos
7a0437f04f83: Already exists
Digest: sha256:5528e8b1b1719d34604c87e11dcd1c0a20bedf46e83b5632cdeac91b8c04efc1
Status: Downloaded newer image for centos:latest
 ---> 300e315adb2f
Step 2/2 : MAINTAINER buddy@gmail.com
 ---> Running in 3d87747b6dc0
Removing intermediate container 3d87747b6dc0
 ---> a60a851db0fd
Successfully built a60a851db0fd
Successfully tagged centos_buddy:latest
~$
~$
```

Let's verify if the image has been created:

```
buddy
~$ sudo docker images
REPOSITORY     TAG       IMAGE ID       CREATED          SIZE
centos_buddy   latest    a60a851db0fd   44 seconds ago   209MB
nginx          latest    4f380adfc10f   4 days ago       133MB
ubuntu         latest    9873176a8ff5   9 days ago       72.7MB
centos         latest    300e315adb2f   6 months ago     209MB
~$
```

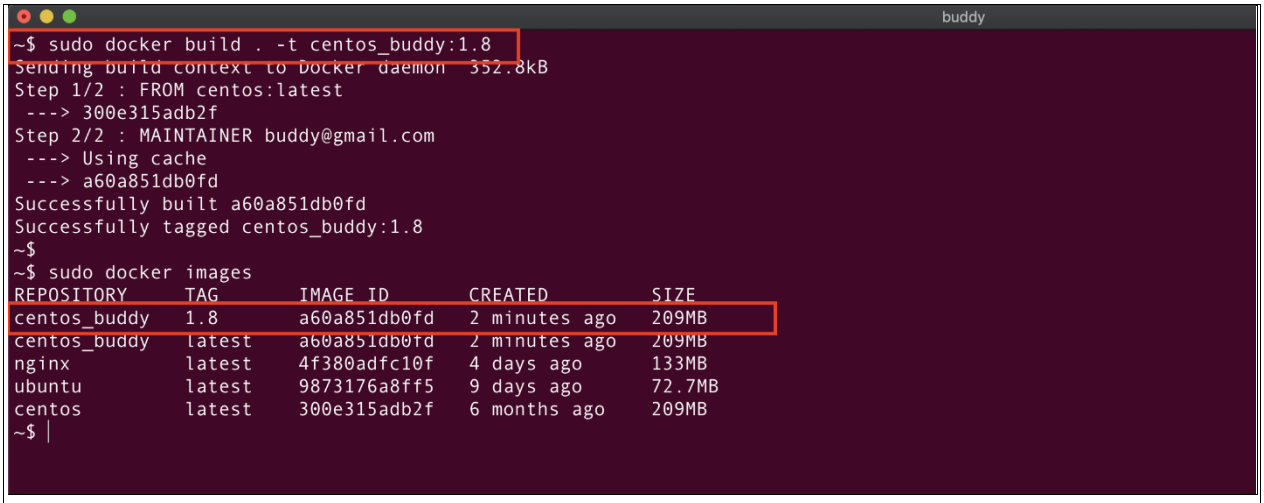
Build Docker Images with a different tag

```
docker build . -t <image_name>:<tag or version>
```

The above command will build an image with imagename and tag 1.8.

Here's an example:

```
docker build . -t centos_buddy:1.8
```



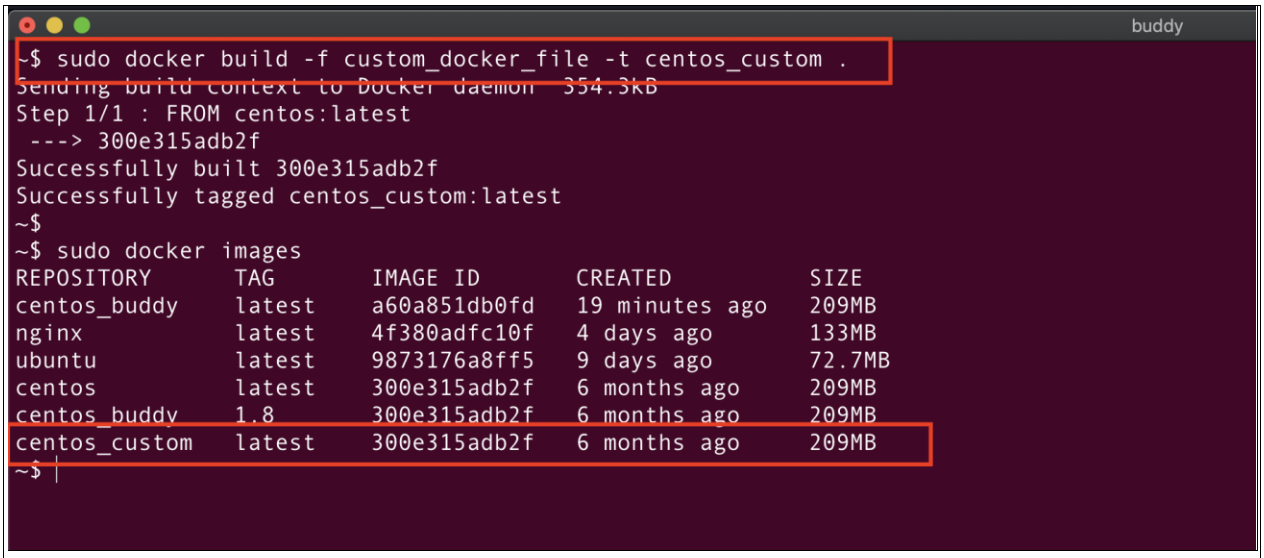
Build a Docker Image using a custom named Dockerfile

We can build a Docker image from a Dockerfile which is not named Dockerfile using the below command:

```
docker build -f <custom_docker_file_name> -t <image_name> .
```

Let's create a Dockerfile with name custom_docker_file:

```
docker build -f custom_docker_file -t centos_custom .
```

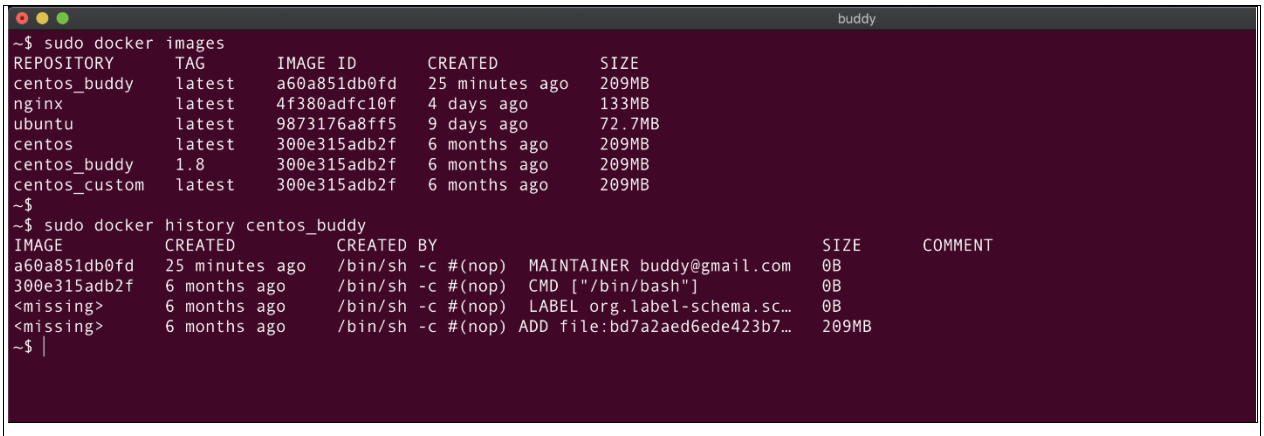


Build a Docker Image from a Dockerfile that is not in the Current Directory

```
docker build -f </path/to/dockerfilename>  -t <image_name> .
```

Show History of a Docker Image

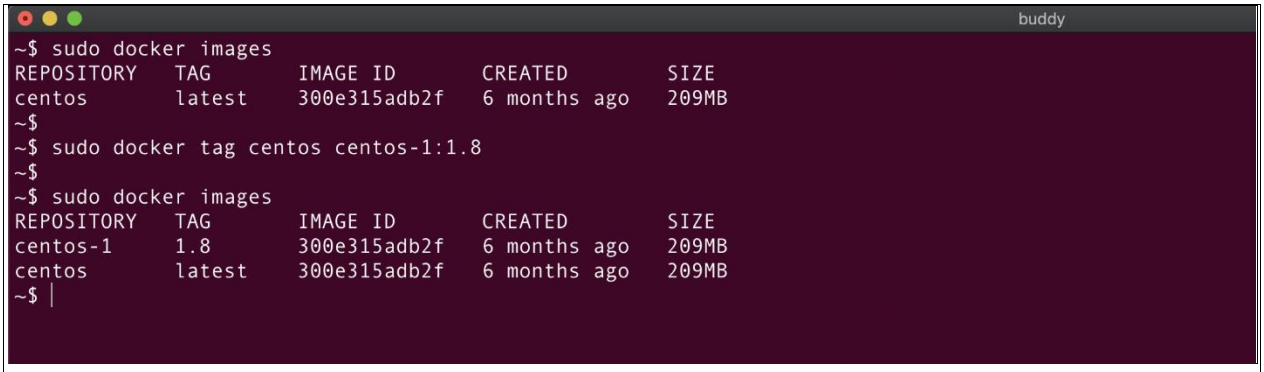
```
docker history <imagename or imageid>
```



The above screenshot shows the history for the image centos_buddy.

Rename an existing Docker Image

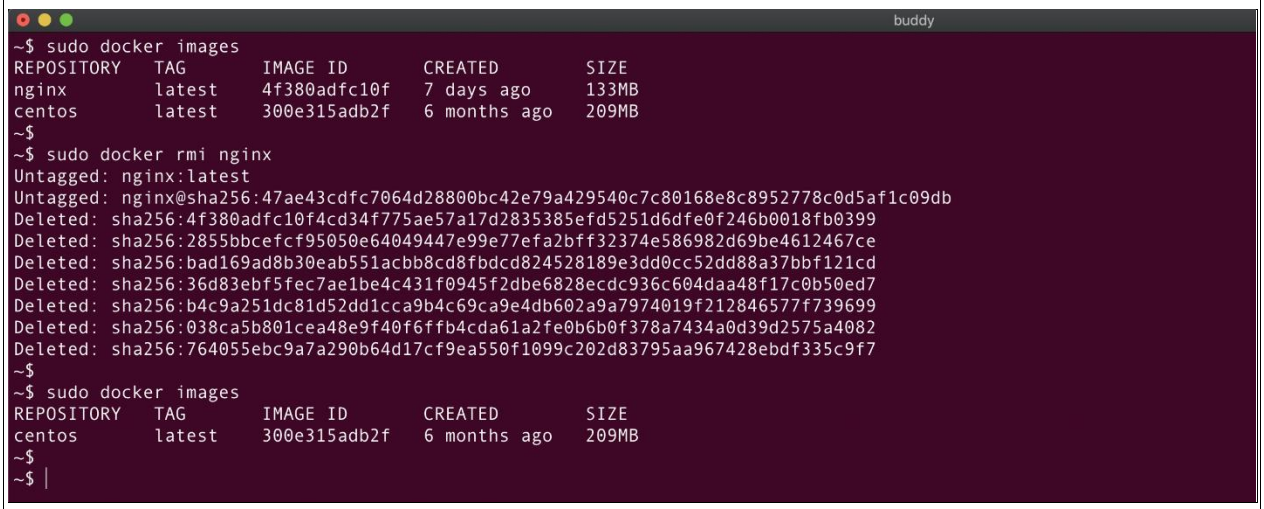
```
docker tag <imagename> <newname>:<version>
```



Remove Docker images

```
docker rmi <image_name or image_id>
```

This command will delete the Docker image if the image is not used by any container.



Force delete a Docker Image

```
docker rmi -f <image_name or image_id>
```

Unused Docker Images

Unused Docker images are not used by any containers.

The images that are displayed when we do `docker ps -a` are used by some of the existing containers.

So, the unused images are:

(All images from docker images -a) - (all images from docker ps -a)

Dangling Docker Images

When we build a Docker image using Dockerfile, Docker creates an image with the given name.

Here's a simple example:

```
docker build . -t imagename
```

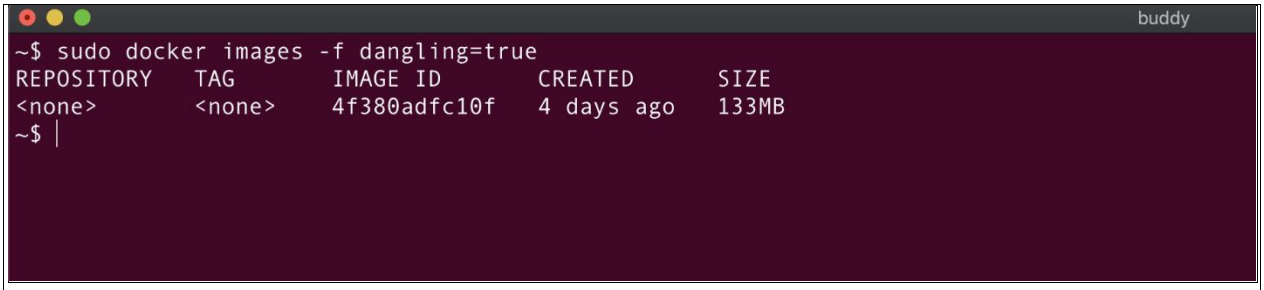
Docker will create an image from the Dockerfile in the current directory with the name imagename.

If we do some changes in the Dockerfile and rebuild the image again with the same name, Docker will update the name of the previous image to <none> and tag it <none>.

These images with the name <none> and tag <none> are called dangling images.

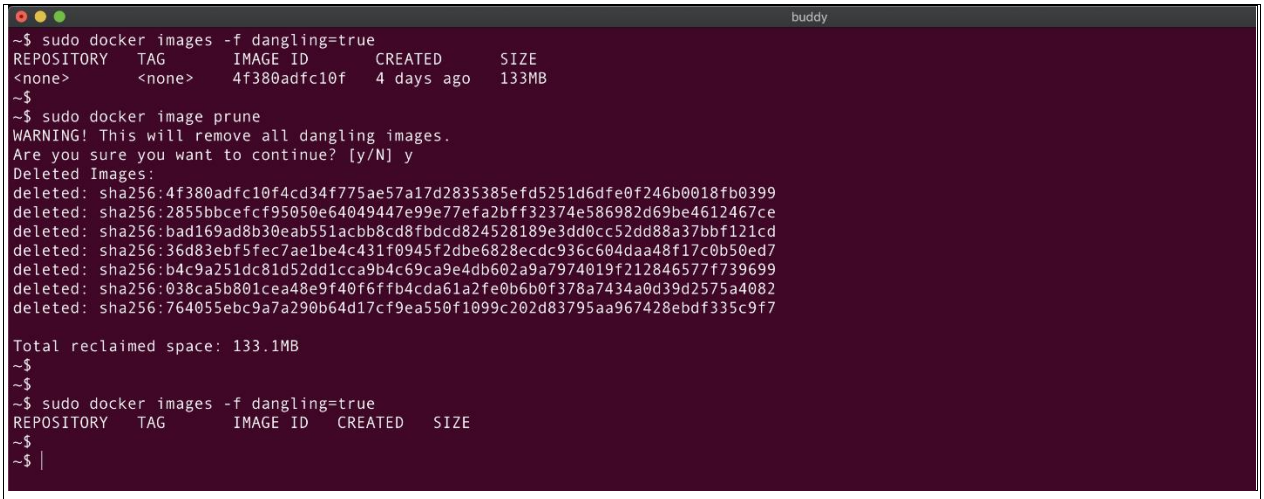
List Dangling Docker Images

```
docker images -f dangling=true
```



Remove all the Dangling Docker Images

```
docker image prune
```



If the dangling images are referenced by containers (either running or not running), Docker will not prune these dangling images.

To remove dangling images, we've to make sure that they are not referenced by any container.

We can first run `docker container prune` to remove all the stopped containers and the `docker images` command will now remove the dangling images that were referenced by these stopped containers.

We can also use the below command to remove the dangling images:

```
docker rmi $(docker images -f dangling=true -q)
```

`docker images -f dangling=true -q` would return the Ids of all the dangling images.

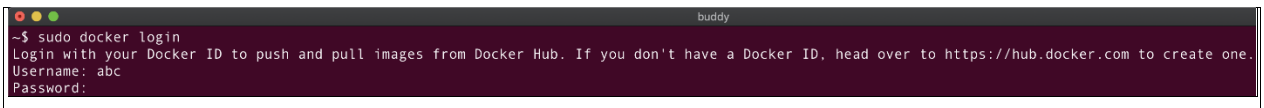
Remove all the Dangling and Unused Docker Images

```
docker image prune -a
```

Login to Docker

To login to Docker hub, we can use the below command:

```
docker login
```



You can enter your username and password to log in to Docker hub.

Push a Docker Image to the Docker Registry

Once we are logged in to Docker hub, we can push the Docker images to the registry using the below command:

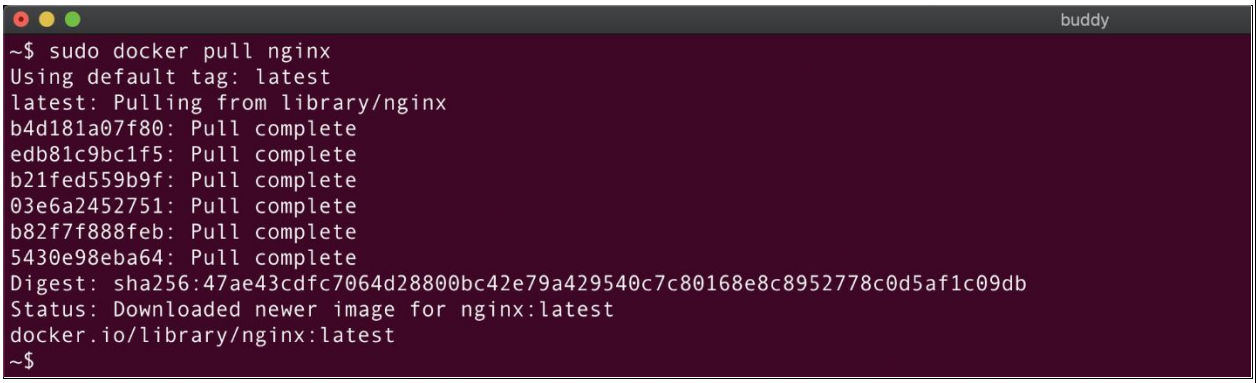
```
docker push repository_name/imagename:tag
```

Download a Docker Image from the registry

```
docker pull imagename:tag
```

If the image is not present on the host machine, Docker will pull the image from the Docker registry.

If no tag is specified, Docker will pull the latest image.



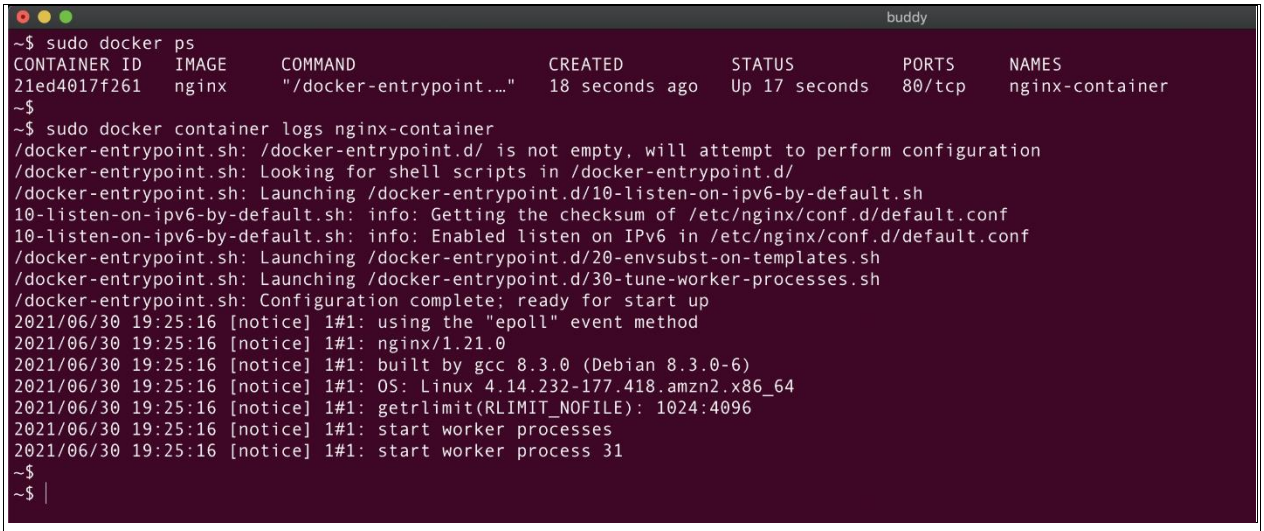
Docker logs

Let's now look at some commands useful for checking logs in a Docker container:

Get Logs of the Docker container

We can get the logs of the Docker container as:

```
docker container logs <container_id or container_name>dock
```



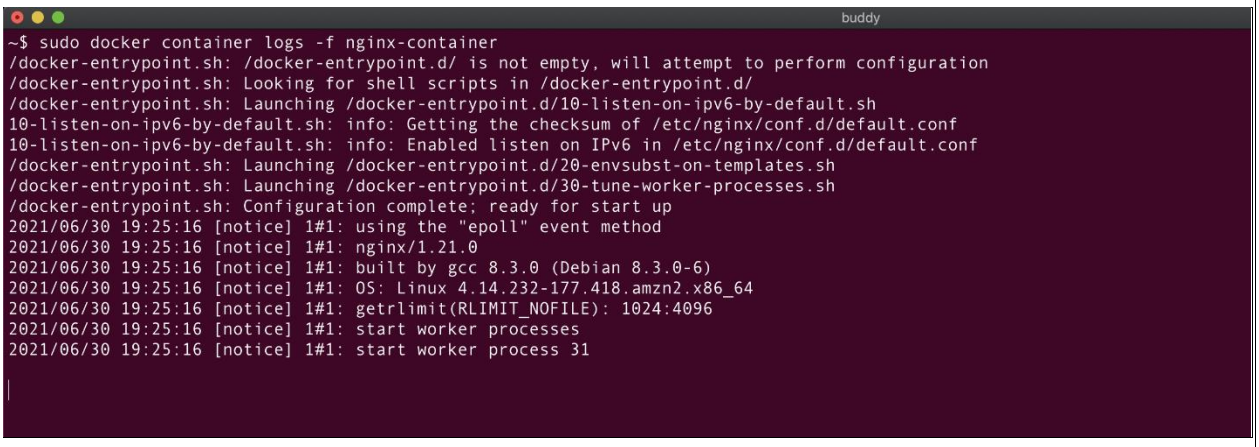
The above screenshot shows the logs of the container nginx-container.

Monitor the Docker Container Logs

To display the last few lines of the container logs and monitor them, we can use the below command:

```
docker container logs -f <container_id or container_name>
```

The new messages in the container would be displayed here! This is similar to the tail -f command.



Get the last 2 lines of the Container Logs

```
docker container logs --tail 2 <container_id or container_name>
```

```
buddy
~$ sudo docker container logs --tail 2 nginx-container
2021/06/30 19:25:16 [notice] 1#1: start worker processes
2021/06/30 19:25:16 [notice] 1#1: start worker process 31
~$
```

Docker Network Commands

Here are some of the useful Docker Network commands:

List all the Networks

```
docker network ls
```

```
buddy
~$ sudo docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
687204004d78   bridge    bridge    local
55a2e8c8af6b   host      host      local
096926ffad37   none      null      local
~$
~$
```

Create a Network

```
docker network create --driver <driver-name> <bridge-name>
```

driver-name can be either bridge or overlay

bridge would be used by default if --driver option is not provided.

```
docker network create --driver bridge new-network
```

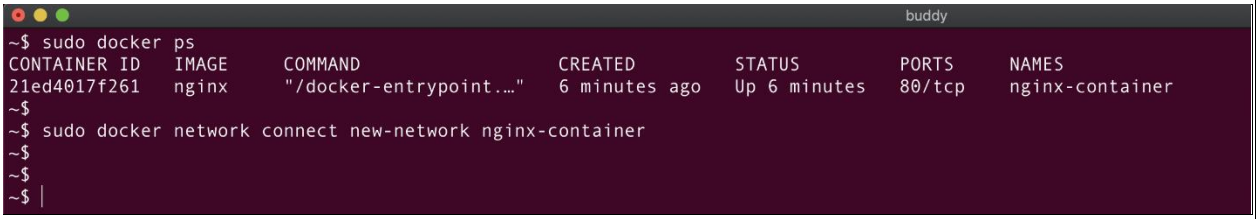
```
buddy
~$ sudo docker network create --driver bridge new-network
98fb57e1e88f34114a164189ea53e45160d1610f1a63112ead7ba5c2c7de40e3
~$
~$ sudo docker network ls
NETWORK ID     NAME          DRIVER    SCOPE
687204004d78   bridge        bridge    local
55a2e8c8af6b   host          host      local
98fb57e1e88f   new-network   bridge    local
096926ffad37   none          null      local
~$
~$
```

The network called new-network is created successfully!

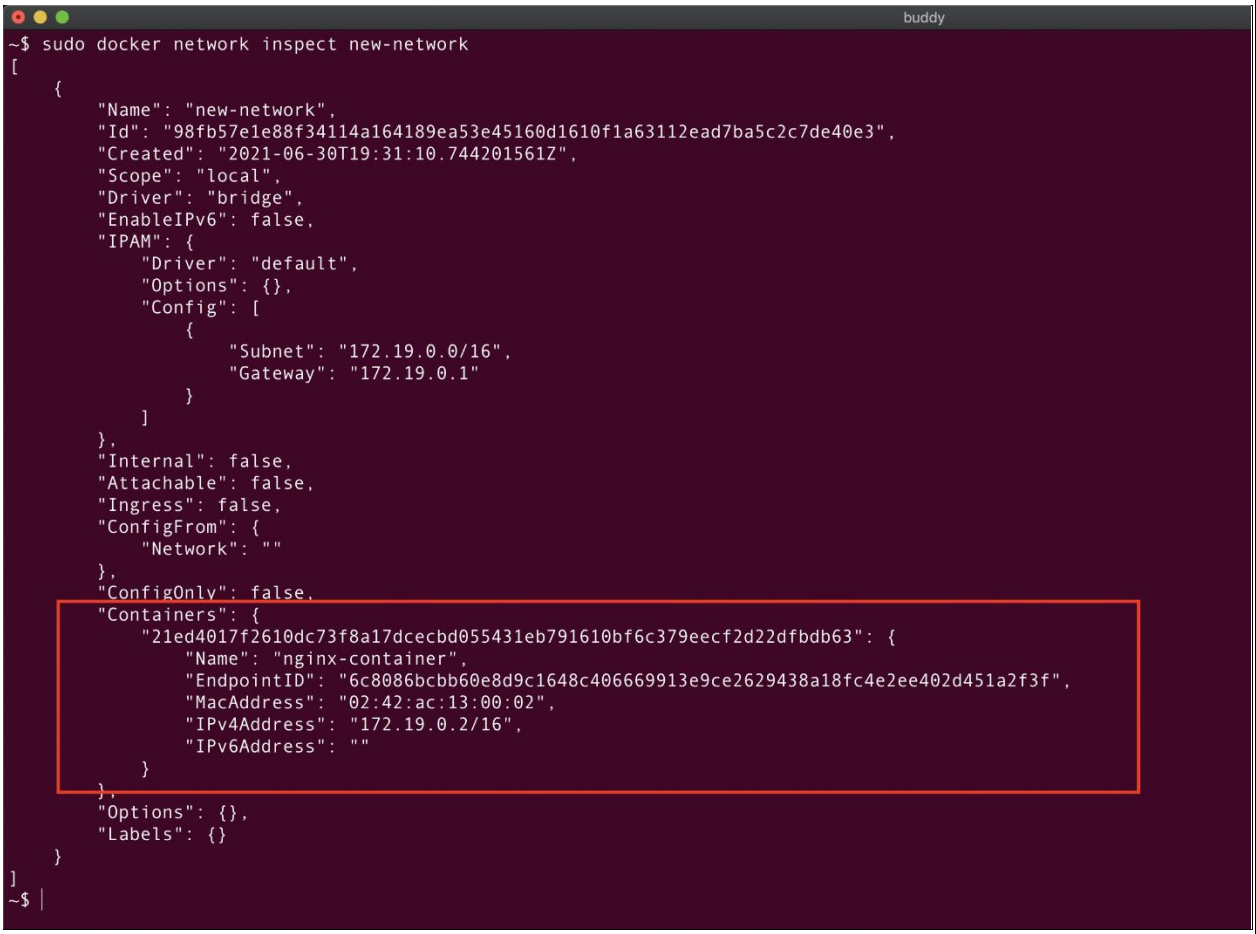
Connect a Docker Container to a Network:

```
docker network connect <network_id or network_name> <container_id or container_name>
```

The above command will connect the container with the specified network. Let's look at an example:



The container nginx-container is connected to the network new-network and we can verify this using docker network inspect:



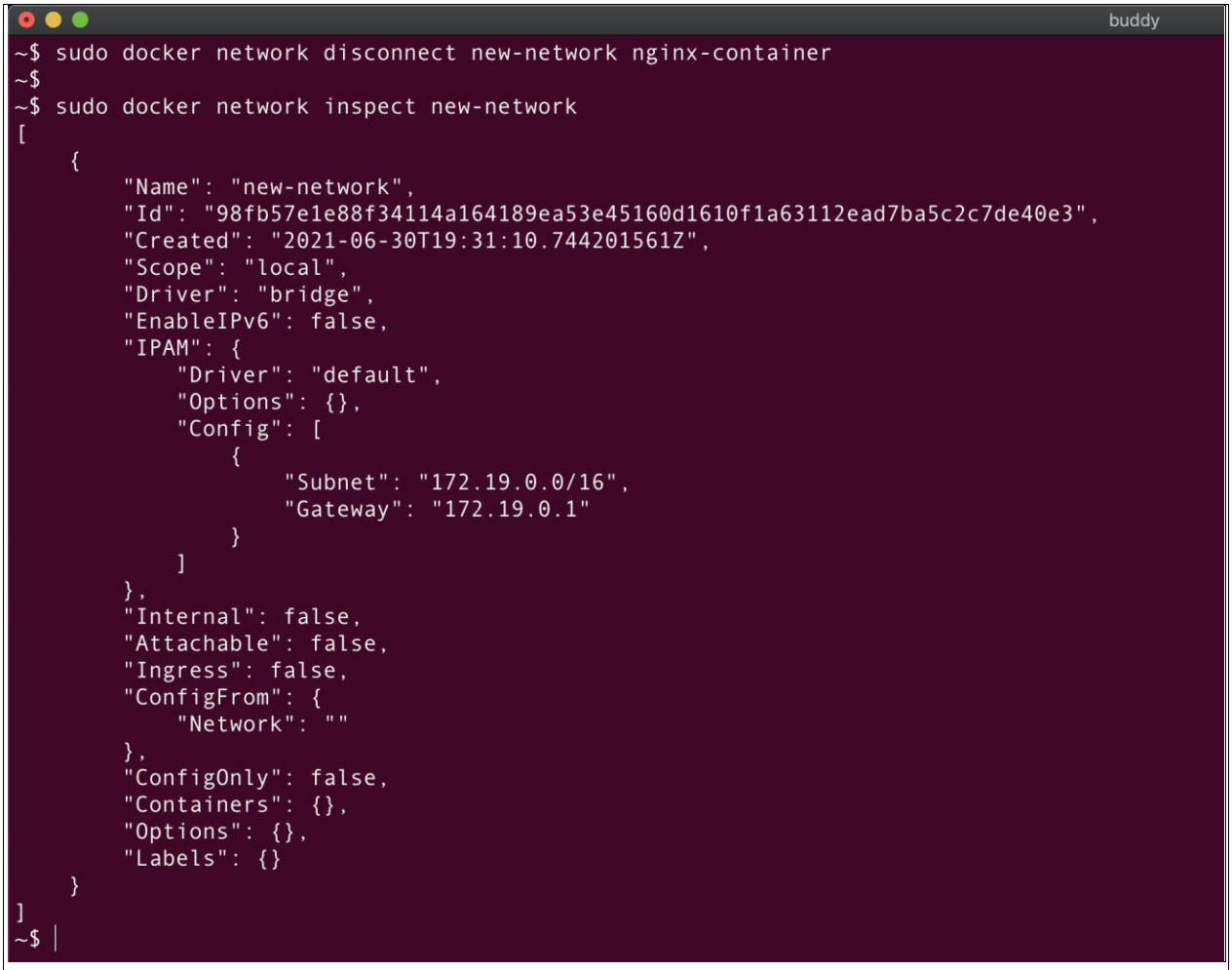
Connect a Docker Container to a Network on Start

This command connects a Docker container to a network as soon as it starts:

```
docker run -d --network=<network_name or id> <contaienr_name>
```

Disconnect a Docker container from a Network:

```
docker network disconnect <network_name_or_id> <container_name_or_id>
```



Remove a Network

```
docker network rm <network_id or network_name>
```

```
buddy
~$ sudo docker network ls
NETWORK ID     NAME          DRIVER    SCOPE
687204004d78   bridge        bridge    local
55a2e8c8af6b   host          host      local
98fb57e1e88f   new-network   bridge    local
096926ffad37   none          null      local
~$
~$ sudo docker network rm new-network
new-network
~$
~$ sudo docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
687204004d78   bridge    bridge    local
55a2e8c8af6b   host      host      local
096926ffad37   none      null      local
~$
```

Show Information about one or more Networks

```
docker network inspect <network_id or network_name>
```

Get the IP Address of the running Docker Container

```
sudo docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
<container_name or container_id>
```

```
buddy
~$ sudo docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS     NAMES
10a6a69a1376   nginx     "/docker-entrypoint.…"   11 seconds ago   Up 10 seconds   80/tcp    fervent_babbage
~$
~$ sudo sudo docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' 10a6a69a1376
172.17.0.2
~$
```

Docker Volumes

In this section, we'll look at some of the commonly used Docker volumes commands!

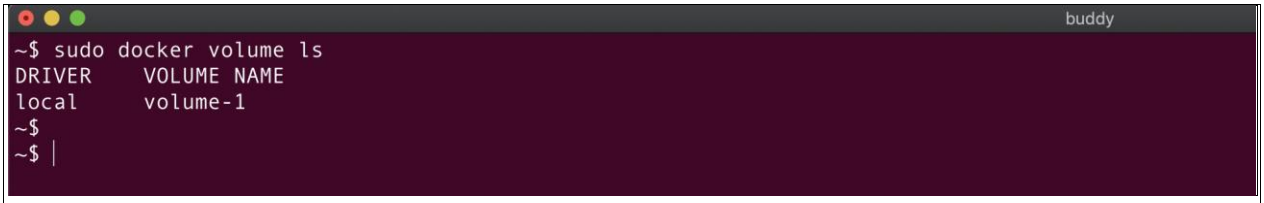
Create Docker Volume

```
docker volume create --name volume-name
```

```
buddy
~$ sudo docker volume create --name volume-1
volume-1
~$
~$
```

List Docker Volumes

```
docker volume ls
```



Mounting Docker Volume using the -v Flag

We can mount the volume inside the Docker container once it is created using the below command:

```
docker run -it --name <container-name> -v <volume-name>:<path-in-container-where-volume-is-mounted> <image-name>
```

We are creating a new container with the container name <container-name> using the image <image-name> and then mount the volume volume-name inside the container at path-in-container-where-volume-is-mounted.

Mounting Docker Volume using the --mount Flag

```
docker run -it --name <container-name> --mount soure=<volume-name>, destination=<path-in-container-where-volume-is-mounted> <image-name>
```

Get Details about a Docker Volume

```
docker volume inspect <volume-name>
```

Remove a Docker Volume

To remove the volume, we first have to remove the containers using that volume and then only we can remove the volume.

To remove the volume we can use the below command:

```
docker volume rm <volume-name>
```

Volume Mount using bind-mount

To mount any specific host directory inside the container, we have to use the below Docker run command:

```
docker run -it -v /path/on/host:/path/in/contianer/where/volume/has/to/be/mounted
<image-name>
```

Creating Bind Mount Volume using the --mount flag

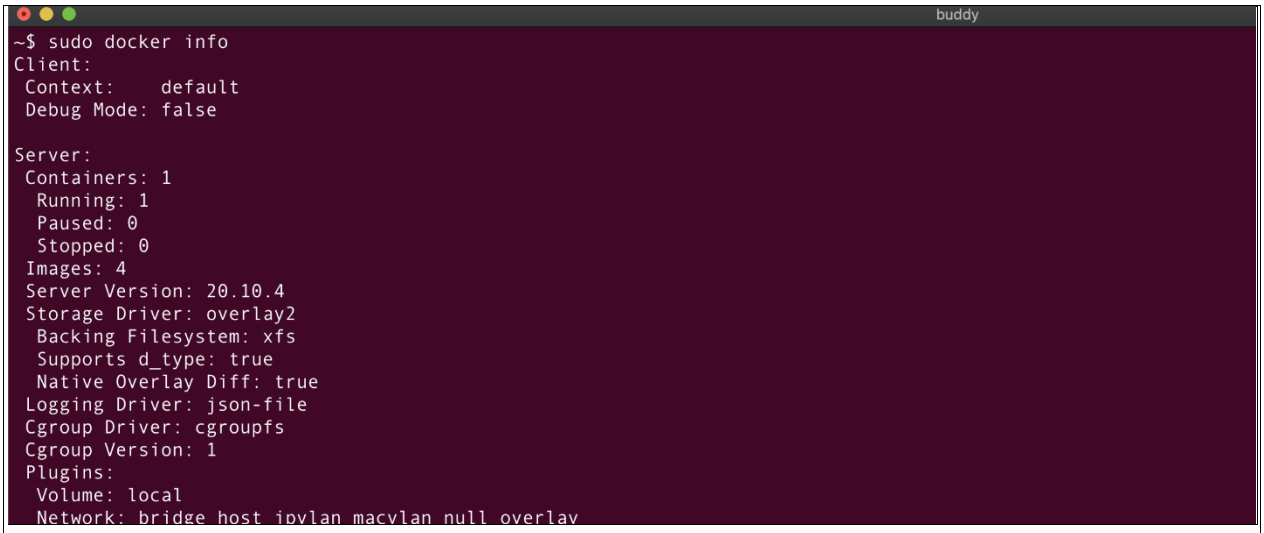
```
docker run -it --name <container_name> --mount
type=bind,source=/path/on/host/,target=/path/on/container first-image
```

If a directory in a container has some content and you mount the volume with type bind onto that directory, the existing content of that directory would be lost and you get an empty directory.

Docker System-wide Commands

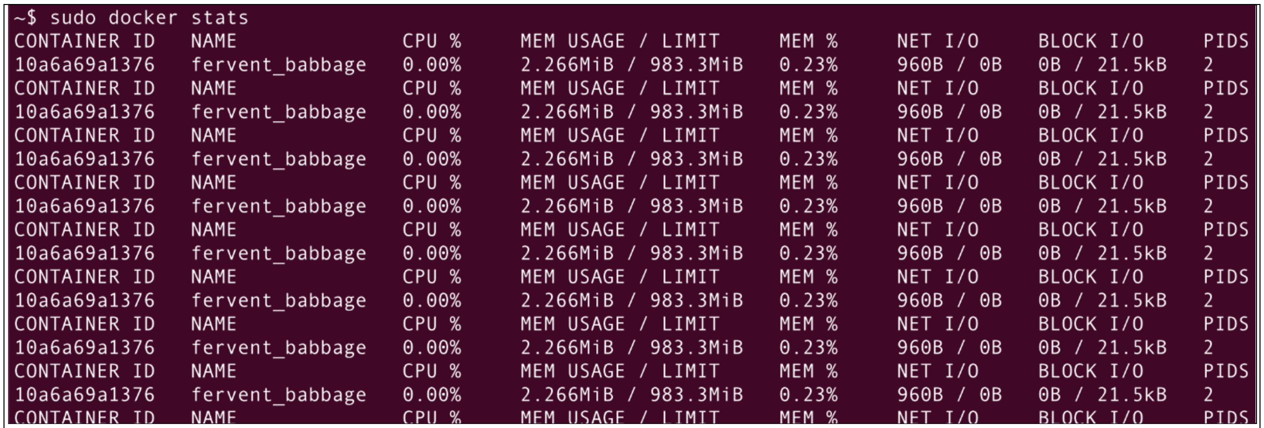
Docker Info

```
docker info
```



Docker Stats of the running Container

```
docker stats
```



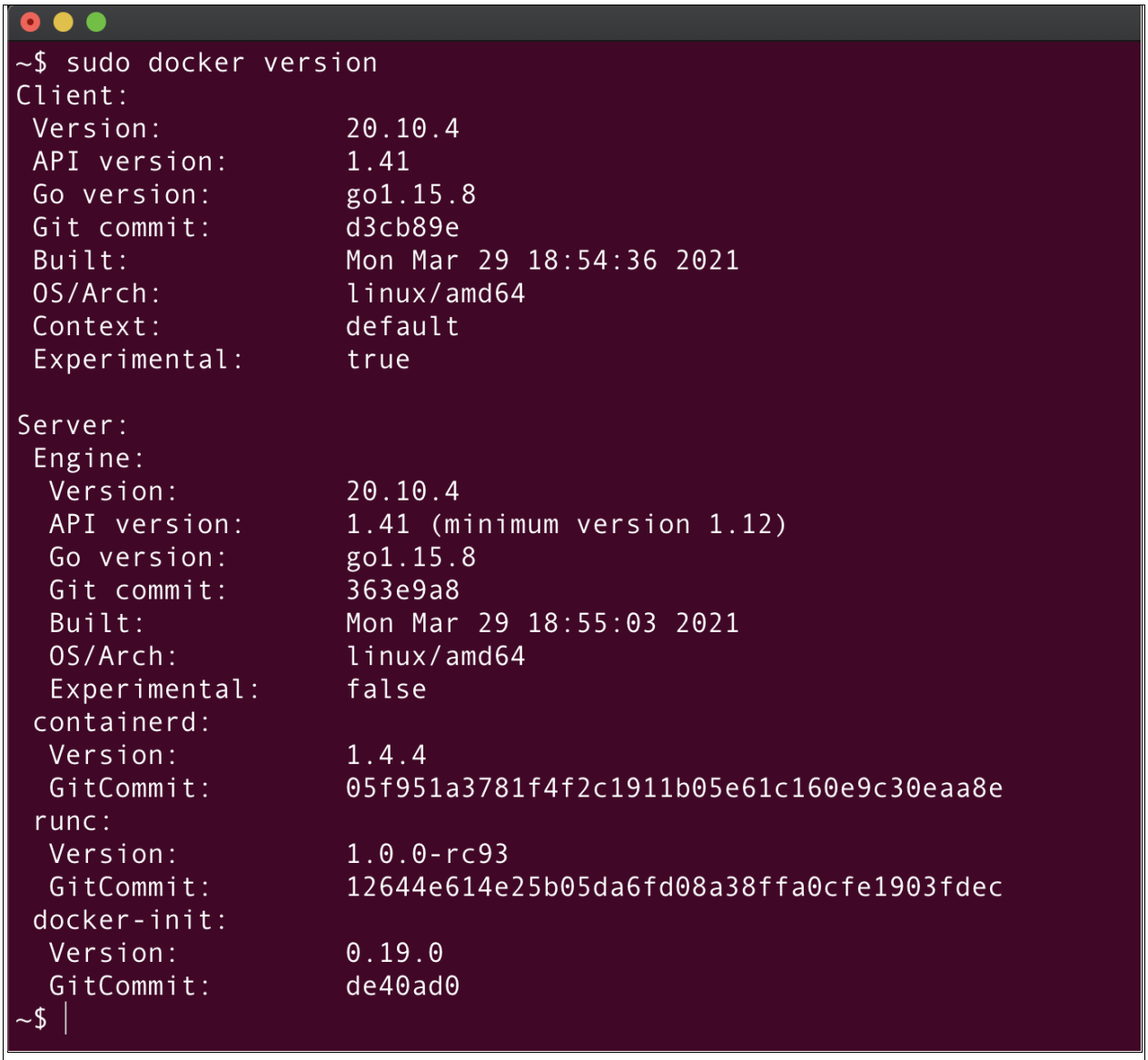
```
~$ sudo docker stats
CONTAINER ID   NAME              CPU %    MEM USAGE / LIMIT     MEM %    NET I/O      BLOCK I/O      PIDS
10a6a69a1376   fervent_babbage   0.00%    2.266MiB / 983.3MiB   0.23%    960B / 0B    0B / 21.5kB    2
CONTAINER ID   NAME              CPU %    MEM USAGE / LIMIT     MEM %    NET I/O      BLOCK I/O      PIDS
10a6a69a1376   fervent_babbage   0.00%    2.266MiB / 983.3MiB   0.23%    960B / 0B    0B / 21.5kB    2
CONTAINER ID   NAME              CPU %    MEM USAGE / LIMIT     MEM %    NET I/O      BLOCK I/O      PIDS
10a6a69a1376   fervent_babbage   0.00%    2.266MiB / 983.3MiB   0.23%    960B / 0B    0B / 21.5kB    2
CONTAINER ID   NAME              CPU %    MEM USAGE / LIMIT     MEM %    NET I/O      BLOCK I/O      PIDS
10a6a69a1376   fervent_babbage   0.00%    2.266MiB / 983.3MiB   0.23%    960B / 0B    0B / 21.5kB    2
CONTAINER ID   NAME              CPU %    MEM USAGE / LIMIT     MEM %    NET I/O      BLOCK I/O      PIDS
10a6a69a1376   fervent_babbage   0.00%    2.266MiB / 983.3MiB   0.23%    960B / 0B    0B / 21.5kB    2
CONTAINER ID   NAME              CPU %    MEM USAGE / LIMIT     MEM %    NET I/O      BLOCK I/O      PIDS
10a6a69a1376   fervent_babbage   0.00%    2.266MiB / 983.3MiB   0.23%    960B / 0B    0B / 21.5kB    2
CONTAINER ID   NAME              CPU %    MEM USAGE / LIMIT     MEM %    NET I/O      BLOCK I/O      PIDS
10a6a69a1376   fervent_babbage   0.00%    2.266MiB / 983.3MiB   0.23%    960B / 0B    0B / 21.5kB    2
CONTAINER ID   NAME              CPU %    MEM USAGE / LIMIT     MEM %    NET I/O      BLOCK I/O      PIDS
10a6a69a1376   fervent_babbage   0.00%    2.266MiB / 983.3MiB   0.23%    960B / 0B    0B / 21.5kB    2
CONTAINER ID   NAME              CPU %    MEM USAGE / LIMIT     MEM %    NET I/O      BLOCK I/O      PIDS
```

Docker Stats of all the Containers

```
docker stats --all
```

Show the Docker Version

```
docker version
```



Get Detailed Info about an Object (Container, Image, Volume, etc)

```
docker inspect <name or id>
docker inspect nginx
```

Get the Summary of Docker Usage

```
docker system df
```

```
buddy
~$ sudo docker system df
TYPE            TOTAL     ACTIVE    SIZE      RECLAIMABLE
Images          4         1         415.2MB   282.1MB (67%)
Containers      1         1         1.095kB   0B (0%)
Local Volumes   0         0         0B        0B
Build Cache     0         0         0B        0B
~$
```

This gives the information about:

1. The total size of all the containers
2. The total size of all the images
3. The total size of the volumes
4. Cache

Clean your Docker system

```
docker system prune
```

```
buddy
~$ sudo docker system prune
WARNING! This will remove:
  - all stopped containers
  - all networks not used by at least one container
  - all dangling images
  - all dangling build cache

Are you sure you want to continue? [y/N] y
Total reclaimed space: 0B
~$
~$
```

This command will clean:

1. All the stopped containers
2. All the networks not used by at least one container
3. All the dangling images
4. All the dangling build cache

Dockerfie

docker build -t image_apline https://github.com/githubdevsecops/jenkins_workshop.git

docker build -f Dockerfile https://github.com/githubdevsecops/jenkins_workshop.git

docker image tag dc447fee37aa image_test_1

FROM

Usage:

FROM <image>

FROM <image>:<tag>

FROM <image>@<digest>

Information:

FROM must be the first non-comment instruction in the Dockerfile.

FROM can appear multiple times within a single Dockerfile in order to create multiple images. Simply make a note of the last image ID output by the commit before each new FROM command.

The tag or digest values are optional. If you omit either of them, the builder assumes a latest by default. The builder returns an error if it cannot match the tag value.

Reference - Best Practices

MAINTAINER

Usage:

MAINTAINER <name>

The MAINTAINER instruction allows you to set the Author field of the generated images.

Reference

RUN

Usage:

RUN <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows)

RUN ["<executable>", "<param1>", "<param2>"] (exec form)

Information:

The exec form makes it possible to avoid shell string munging, and to RUN commands using a base image that does not contain the specified shell executable.

The default shell for the shell form can be changed using the SHELL command.

Normal shell processing does not occur when using the exec form. For example, RUN ["echo", "$HOME"] will not do variable substitution on $HOME.

Reference - Best Practices

CMD

Usage:

CMD ["<executable>","<param1>","<param2>"] (exec form, this is the preferred form)

CMD ["<param1>","<param2>"] (as default parameters to ENTRYPOINT)

CMD <command> <param1> <param2> (shell form)

Information:

The main purpose of a CMD is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an ENTRYPOINT instruction as well.

There can only be one CMD instruction in a Dockerfile. If you list more than one CMD then only the last CMD will take effect.

If CMD is used to provide default arguments for the ENTRYPOINT instruction, both the CMD and ENTRYPOINT instructions should be specified with the JSON array format.

If the user specifies arguments to docker run then they will override the default specified in CMD.

Normal shell processing does not occur when using the exec form. For example, CMD ["echo", "$HOME"] will not do variable substitution on $HOME.

Reference - Best Practices

LABEL

Usage:

LABEL <key>=<value> [<key>=<value> ...]

Information:

The LABEL instruction adds metadata to an image.

To include spaces within a LABEL value, use quotes and backslashes as you would in command-line parsing.

Labels are additive including LABELs in FROM images.

If Docker encounters a label/key that already exists, the new value overrides any previous labels with identical keys.

To view an image's labels, use the docker inspect command. They will be under the "Labels" JSON attribute.

Reference - Best Practices

EXPOSE

Usage:

EXPOSE <port> [<port> ...]

Information:

Informs Docker that the container listens on the specified network port(s) at runtime.

EXPOSE does not make the ports of the container accessible to the host.

Reference - Best Practices

ENV

Usage:

ENV <key> <value>

ENV <key>=<value> [<key>=<value> ...]

Information:

The ENV instruction sets the environment variable <key> to the value <value>.

The value will be in the environment of all “descendant” Dockerfile commands and can be replaced inline as well.

The environment variables set using ENV will persist when a container is run from the resulting image.

The first form will set a single variable to a value with the entire string after the first space being treated as the <value> - including characters such as spaces and quotes.

Reference - Best Practices

ADD

Usage:

ADD <src> [<src> ...] <dest>

ADD ["<src>", ... "<dest>"] (this form is required for paths containing whitespace)

Information:

Copies new files, directories, or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.

<src> may contain wildcards and matching will be done using Go’s filepath.Match rules.

If <src> is a file or directory, then they must be relative to the source directory that is being built (the context of the build).

<dest> is an absolute path, or a path relative to WORKDIR.

If <dest> doesn’t exist, it is created along with all missing directories in its path.

Reference - Best Practices

COPY

Usage:

COPY <src> [<src> ...] <dest>

COPY ["<src>", ... "<dest>"] (this form is required for paths containing whitespace)

Information:

Copies new files or directories from <src> and adds them to the filesystem of the image at the path <dest>.

<src> may contain wildcards and matching will be done using Go's filepath.Match rules.

<src> must be relative to the source directory that is being built (the context of the build).

<dest> is an absolute path, or a path relative to WORKDIR.

If <dest> doesn't exist, it is created along with all missing directories in its path.

Reference - Best Practices

ENTRYPOINT

Usage:

ENTRYPOINT ["<executable>", "<param1>", "<param2>"] (exec form, preferred)

ENTRYPOINT <command> <param1> <param2> (shell form)

Information:

Allows you to configure a container that will run as an executable.

Command line arguments to docker run <image> will be appended after all elements in an exec form ENTRYPOINT and will override all elements specified using CMD.

The shell form prevents any CMD or run command line arguments from being used, but the ENTRYPOINT will start via the shell. This means the executable will not be PID 1 nor will it receive UNIX signals. Prepend exec to get around this drawback.

Only the last ENTRYPOINT instruction in the Dockerfile will have an effect.

Reference - Best Practices

VOLUME

Usage:

VOLUME ["<path>", ...]

VOLUME <path> [<path> ...]

Creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

Reference - Best Practices

USER

Usage:

USER <username | UID>

The USER instruction sets the user name or UID to use when running the image and for any RUN, CMD and ENTRYPOINT instructions that follow it in the Dockerfile.

Reference - Best Practices

WORKDIR

Usage:

WORKDIR </path/to/workdir>

Information:

Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY, and ADD instructions that follow it.

It can be used multiple times in the one Dockerfile. If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction.

Reference - Best Practices

ARG

Usage:

ARG <name>[=<default value>]

Information:

Defines a variable that users can pass at build-time to the builder with the docker build command using the --build-arg <varname>=<value> flag.

Multiple variables may be defined by specifying ARG multiple times.

It is not recommended to use build-time variables for passing secrets like github keys, user credentials, etc. Build-time variable values are visible to any user of the image with the docker history command.

Environment variables defined using the ENV instruction always override an ARG instruction of the same name.

Docker has a set of predefined ARG variables that you can use without a corresponding ARG instruction in the Dockerfile.

HTTP_PROXY and http_proxy

HTTPS_PROXY and https_proxy

FTP_PROXY and ftp_proxy

NO_PROXY and no_proxy

Reference

ONBUILD

Usage:

ONBUILD <Dockerfile INSTRUCTION>

Information:

Adds to the image a trigger instruction to be executed at a later time, when the image is used as the base for another build. The trigger will be executed in the context of the downstream build, as if it had been inserted immediately after the FROM instruction in the downstream Dockerfile.

Any build instruction can be registered as a trigger.

Triggers are inherited by the "child" build only. In other words, they are not inherited by "grand-children" builds.

The ONBUILD instruction may not trigger FROM, MAINTAINER, or ONBUILD instructions.

Reference - Best Practices

STOPSIGNAL

Usage:

STOPSIGNAL <signal>

The STOPSIGNAL instruction sets the system call signal that will be sent to the container to exit. This signal can be a valid unsigned number that matches a position in the kernel's syscall table, for instance 9, or a signal name in the format SIGNAME, for instance SIGKILL.

Reference

HEALTHCHECK

Usage:

HEALTHCHECK [<options>] CMD <command> (check container health by running a command inside the container)

HEALTHCHECK NONE (disable any healthcheck inherited from the base image)

Information:

Tells Docker how to test a container to check that it is still working

Whenever a health check passes, it becomes healthy. After a certain number of consecutive failures, it becomes unhealthy.

The <options> that can appear are...

--interval=<duration> (default: 30s)

--timeout=<duration> (default: 30s)

--retries=<number> (default: 3)

The health check will first run interval seconds after the container is started, and then again interval seconds after each previous check completes. If a single run of the check takes longer than timeout seconds then the check is considered to have failed. It takes retries consecutive failures of the health check for the container to be considered unhealthy.

There can only be one HEALTHCHECK instruction in a Dockerfile. If you list more than one then only the last HEALTHCHECK will take effect.

<command> can be either a shell command or an exec JSON array.

The command's exit status indicates the health status of the container.

0: success - the container is healthy and ready for use

1: unhealthy - the container is not working correctly

2: reserved - do not use this exit code

The first 4096 bytes of stdout and stderr from the <command> are stored and can be queried with docker inspect.

When the health status of a container changes, a health_status event is generated with the new status.

Reference

SHELL

Usage:

SHELL ["<executable>", "<param1>", "<param2>"]

Information:

Allows the default shell used for the shell form of commands to be overridden.

Each SHELL instruction overrides all previous SHELL instructions, and affects all subsequent instructions.

Allows an alternate shell be used such as zsh, csh, tcsh, powershell, and others.