Diving with the Whale | Day II

Containers & Images
Management, and Hands On!

Diving with the Whale - Docker Day II - Containers & Images Management

Author : Ahmed Ayman

Stuff You Should know

- **ONLY ONE!**
    - Docker containers are designed to run only one process.
- **Ephemeral** (lasting for a very short time)
    - Docker containers are meant to be ephemeral, once their process completes the container goes to sleep, and it's memories about anything happened while it was running goes with it !
- **No persisting data!**
    - Because docker containers are ephemeral, any stored data are ephemeral as well!
    - But there is always a work around right?

Docker command structure

```
docker command [-options] [arguments]
```

Images Management

How to display all images you have locally ?

```
docker images
```

How to remove an image ?

```
# docker rmi <image_name/id>
docker rmi ubuntu
```

To remove an image. All containers running or stopped must first be removed!

How to Download/Pull new Images from Docker-Hub

```
# docker pull <image_name>
docker pull hello-world
```

docker pull by default pulls from Docker-Hub registry

How to pull from other registries ?

```
# docker pull <registry>/<image_name>
docker pull ctfd/ctfd
```

this will pull ctfd image from ctfd registry

How to build new image ?

```
# change your directory to the place containing your Dockerfile
cd project
docker build -t tag .
```

How to push images to registry ?

```
docker login
# then enter your docker-hub username and password
# docker push <image_name/id>
docker push todo-flask-v2
```

Containers Management

How to run a container!

```
# docker run <image_name/id>
docker run ubuntu
```

So what actually happened when we ran the above command ? nothing ?!!

as we said earlier docker containers are ephemeral so the container ran and exited once it's process is done!

How to display all running containers ?

```
docker ps
# CONTAINER ID   IMAGE           COMMAND                  CREATED          STATUS         PORTS     NAMES
# 131cf4b3c0f6   todo-flask-v2   "python3 -m flask ru…"   13 seconds ago   Up 9 seconds             ecstatic_cori
```

- Container ID
  - Each container has a unique ID, you can reference the container using it's ID
- Image
  - The image name the container is an instance of
- Command
  - The running command on the container
- Created
  - When was the container created
- Status
  - What is the status of the container (running - paused - exited)
- Ports
  - Exposed ports
- Name
  - Each container gets a unique random name unless you specified a unique name when running it

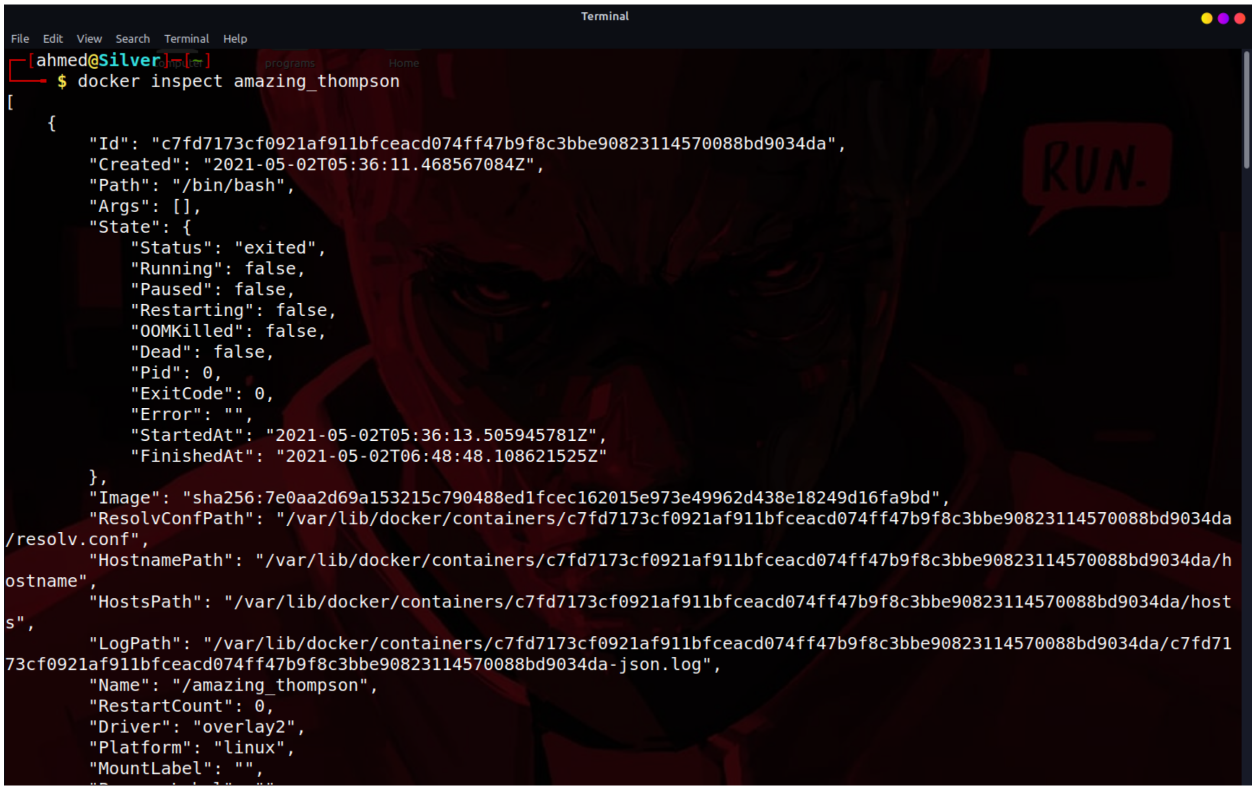
How to display all containers ?

```
docker ps -a
# CONTAINER ID   IMAGE           COMMAND                  CREATED       STATUS                     PORTS     NAMES
# 69702cf6c516   todo-flask-v2   "python3 -m flask ru…"   6 hours ago   Exited (0) 6 hours ago               ecstatic_lewin
# 6ae1e6519b0f   todo-flask-v2   "python3 -m flask ru…"   7 hours ago   Exited (0) 6 hours ago               priceless_elion
# 114bfd201f11   be7f076f4fb1    "/bin/sh -c 'pip ins…"   7 hours ago   Exited (2) 7 hours ago               sad_cohen
# c7fd7173cf09   ubuntu          "/bin/bash"              44 hours ago  Exited (0) 43 hours ago              amazing_thompson
```

Docker Inspect

Docker inspect will provide all information about your docker container in JSON format.

```
# docker inspect <container_id>
# docker inspect <container_name>
docker inspect 6970   # we can use the first few characters that makes the container id unique instade of using the whole id
```

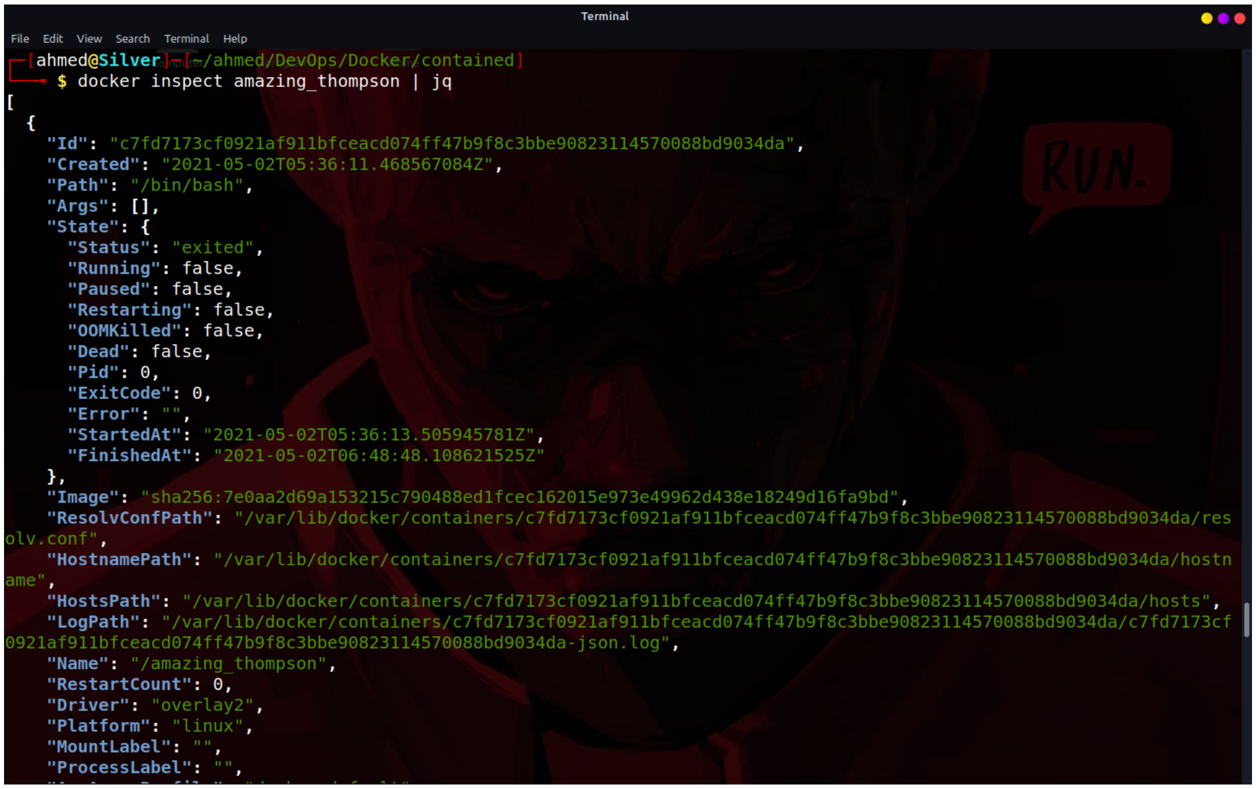


```
┌─[ahmed@Silver]─[~]
└──╼ $ docker inspect amazing_thompson
[
    {
        "Id": "c7fd7173cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da",
        "Created": "2021-05-02T05:36:11.468567084Z",
        "Path": "/bin/bash",
        "Args": [],
        "State": {
            "Status": "exited",
            "Running": false,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 0,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2021-05-02T05:36:13.505945781Z",
            "FinishedAt": "2021-05-02T06:48:48.108621525Z"
        },
        "Image": "sha256:7e0aa2d69a153215c790488ed1fcec162015e973e49962d438e18249d16fa9bd",
        "ResolvConfPath": "/var/lib/docker/containers/c7fd7173cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da
/resolv.conf",
        "HostnamePath": "/var/lib/docker/containers/c7fd7173cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da/h
ostname",
        "HostsPath": "/var/lib/docker/containers/c7fd7173cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da/host
s",
        "LogPath": "/var/lib/docker/containers/c7fd7173cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da/c7fd71
73cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da-json.log",
        "Name": "/amazing_thompson",
        "RestartCount": 0,
        "Driver": "overlay2",
        "Platform": "linux",
        "MountLabel": "",
```

and the list of info goes on, and on ..

lets tidy this file to be a little human-readable

```
# if you don't have jq install it using
sudo apt install jq
docker inspect amazing_thompson | jp
```



```
┌─[ahmed@Silver]─[~/ahmed/DevOps/Docker/contained]
└──╼ $ docker inspect amazing_thompson | jq
[
  {
    "Id": "c7fd7173cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da",
    "Created": "2021-05-02T05:36:11.468567084Z",
    "Path": "/bin/bash",
    "Args": [],
    "State": {
      "Status": "exited",
      "Running": false,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 0,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2021-05-02T05:36:13.505945781Z",
      "FinishedAt": "2021-05-02T06:48:48.108621525Z"
    },
    "Image": "sha256:7e0aa2d69a153215c790488ed1fcec162015e973e49962d438e18249d16fa9bd",
    "ResolvConfPath": "/var/lib/docker/containers/c7fd7173cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da/res
olv.conf",
    "HostnamePath": "/var/lib/docker/containers/c7fd7173cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da/hostn
ame",
    "HostsPath": "/var/lib/docker/containers/c7fd7173cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da/hosts",
    "LogPath": "/var/lib/docker/containers/c7fd7173cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da/c7fd7173cf
0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da-json.log",
    "Name": "/amazing_thompson",
    "RestartCount": 0,
    "Driver": "overlay2",
    "Platform": "linux",
    "MountLabel": "",
    "ProcessLabel": "",
```

How do we keep it keep running ?

The process that the container starts with MUST be one that can run indefinitely

```
docker run -it ubuntu /bin/bash
```

- - i , --interactive Keep STDIN open even if not
- -t , --tty Allocate a pseudo-TTY
- -d , --detach Run container in background

Containers Controlling

Create

```
# docker create <image_name/id>
docker create ubuntu
# 82b7efdb8b21753776208011090b033b0842257fa56169f65d0de597df00b1d9
```

creates the docker container from the giving image, but does not run it!

Start

```
# docker start <container_id/name>
docker start 82b7ef
```

this will actually start the container

Run

```
# docker run <image_name/id>
docker run ubuntu
```

this command actually is some think like an alias for

```
docker start $(docker create ubuntu)
```

Pause

```
# docker pause <container_id/name>
docker pause 82b7ef
```

This will stop a running container but will not make it exit i.e. change it's state from up to paused

UnPause

```
# docker pause <container_id/name>
docker unpause 82b7ef
```

This will start a paused container i.e. change it's state from paused to up

Stop

```
# docker stop <container_id/name>
docker stop 82b7ef
```

This will stop a running container i.e. change it's state from up to exited

stop is the gentle way of stopping a container, it's like asking the container politely "can you exit please?" , it's the same as pressing Ctrl+C

The main process inside the container will receive SIGTERM

Kill

```
# docker kill <container_id/name>
docker kill 82b7ef
```

This will force a running container to exit i.e. change it's state from up to exited

Remember saying about the stop command it's the polite way? well kill is the un-respectful one.

The main process inside the container will receive SIGKILL

Delete a Container

Guess the command name!

YES you guessed it right, rm!

```
# docker rm <container_id>
docker rm 82b7ef
```

Why should we delete a container ?

Stopped containers remain on the system and take up space. You can remove these.

TO REMOVE A CONTAINER IT MUST BE STOPPED.

here is a trick to remove all containers in a one-liner

```
docker rm $(docker ps -aq --filter "status=exited")
# $() starts a sub-shell
# -a means all the containers
# -q means quite mode -> outputs only the containers ids
# -f, --filter "key=value filter" -> filters the containers based on the filter
#  "status=exited" -> apply ( get only the exited containers ) as the filter for --filter
```

Mapping Exposed Ports

let's say your web application you're trying to dockeraize uses the port 80, how will you be able to preview the web app if the container is completely isolated from the host system?

mapping ports is the answer!

we will map the port from the container to the host port we want

my flask web app runs on port 5000 so because I have nothing running on port 5000 I will map it to the same port, but I could map it to any other unused port

```
# docker run -p <host_port>:<container_port> <image_name/id>
# maps these specific ports to each other
docker run -p 5000:5000 todo-flask-v2
```

let's say my app uses a lot of ports, so I want to map all of them at once!

```
# docker run -P <image_name/id>
docker run -P todo-flask-v2
```

this will map all exposed ports in the container to random unused ports in the host machine

Docker Exec

we can execute commands against a container using the exec command

```
# docker exec <conatiner_id/name> command
docker exec inspiring_dirac ls
```

this will execute the command ls inside the container inspiring_dirac and prints the output in STDOUT

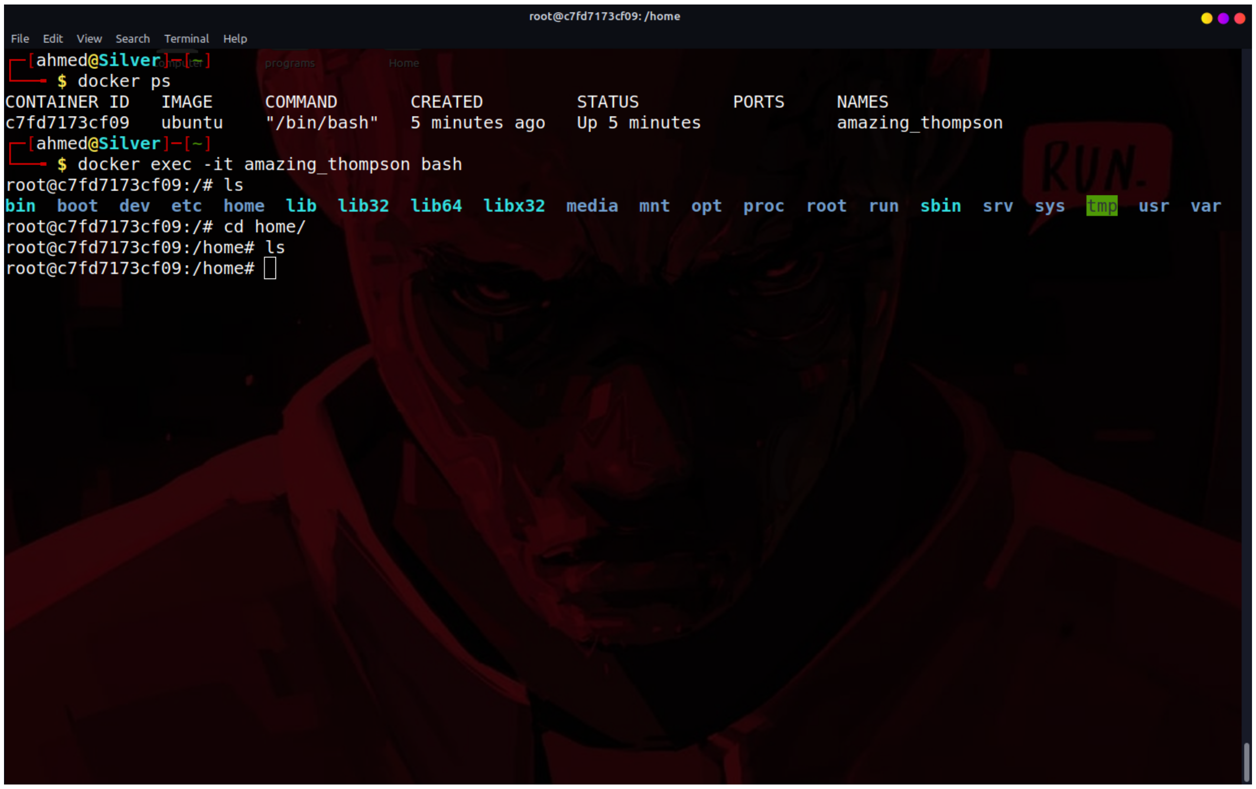
So can we log into a container ?

well yes, but not really!

think about it, now we can execute commands inside the container, and /bin/bash is a command right ? so we can get a shell session and walk around inside the container!

for example

```
# lets run an ubuntu container
docker run -itd ubuntu
# c7fd7173cf0921af911bfceacd074ff47b9f8c3bbe90823114570088bd9034da
docker exec -t 6aeb /bin/bash  # or we can simply use bash
```



```
┌─[ahmed@Silver]─[~]
└──╼ $ docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED         STATUS         PORTS     NAMES
c7fd7173cf09   ubuntu    "/bin/bash"   5 minutes ago   Up 5 minutes             amazing_thompson
┌─[ahmed@Silver]─[~]
└──╼ $ docker exec -it amazing_thompson bash
root@c7fd7173cf09:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@c7fd7173cf09:/# cd home/
root@c7fd7173cf09:/home# ls
root@c7fd7173cf09:/home#
```

"docker exec -it <container_id/name> bash" output

TASK

- Play around with docker images and container
- pull some images Ubuntu image for example or Nginx
- display all images
- choose an image and run it (try to detach it once and once don't)
- run any image, Ubuntu for example
- make it last (execute a command that will keep the container up and running)
- login into the container in the interactive mode

- play arround
- exit the container
- display all running containers
- stop your container using it's id
- display all containers
- remove your container using it's name