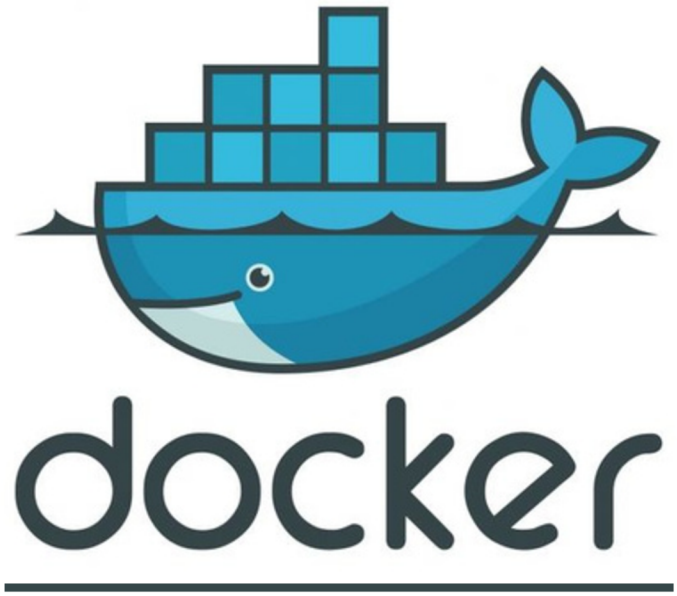


Written by :
Derradji Amine Abdelbasset

Content :

- Docker Core Conception
- Docker Architecture
- Dealing with images
- Dealing with containers
- Dealing with Dockerfile

Hello, I am **Derradji Amine Abdelbasset**, a cloud services associate and an ICT student. I wrote this documentation to help beginners understand Docker basics, especially the theoretical part of images, containers, and Dockerfile. This documentation also consists of the basic commands of each part.

This documentation is one of two parts that I will share on LinkedIn to simplify this tool as much as I can.

For more information, visit my website: www.derradjiamine.me

1 - Docker Core Conception :

To solve the resources waste problem in servers, IT engineers created VMs. Dividing one single server into multiple servers was a great idea. Although resource waste still happens, engineers developed a new technology, which is **containers**. Instead of deploying your application into a VM, you can run your application in a container inside the VM, which provides good isolation and an ideal environment for your apps while saving more resources and running multiple services on one server, which was impossible in the past. Docker is a tool that provides creating containers, deploying applications on them, provisioning the running process, and managing your containers as you like.

To create a container, you should build a necessary key that determines what your container should have and provide. This key is the **image**, and once you run it, your containers will be created and running.

To build an image for a specified project, you need to define another key that determines what the image should contain; it is the **Dockerfile**!

2 - Docker Architecture :

Docker has a specified architecture that provides the reliability of processes and maintains container operations. This architecture is based on some concepts:

Dockerfile: is a file that follows the source files in our project and is used for creating images for the project. It consists of some instructions that define what the image should contain, such as the OS, the required environment, etc.

Image: is a template used for creating and running containers. It describes the project environment such as the OS, libraries, and dependencies, and even the source code of the application. You can use a ready image, or you can build one using the Dockerfile of the specified project.

Container: is an isolated environment that runs the application. It provides all the dependencies and is based on an image. Docker will install all the necessary software for the application inside the container without installing it on the host server, which provides more lightweight, reliability, and security.

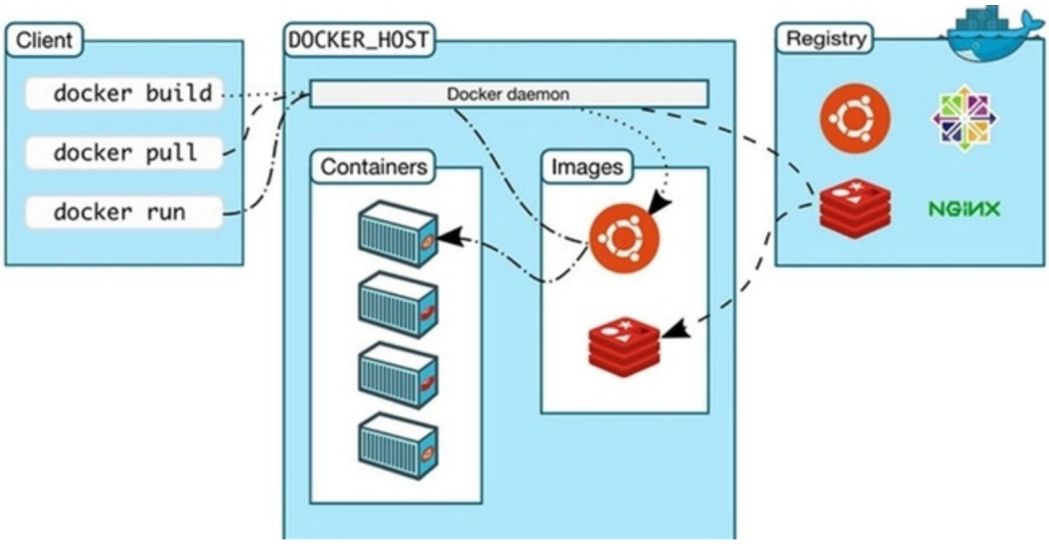
Registry: is a storage space where Docker images are stored. This option provides the ability to push, pull, and share images with developers. This registry can be public, like Docker Hub, or private, like enterprise registries. One registry can store more than one repository. Note that a repository is a collection of multiple images belonging to the same project.

Client: is the interface through which a user interacts with Docker. It can be CLI, when the user manages operations using commands, or GUI, when the user manages operations through a graphical interface such as Docker Desktop.

Docker Daemon: is the core component of Docker. It is an engine responsible for creating, managing, and linking containers. The daemon takes orders from the user via the API connection between the Client and the Daemon.

Namespaces: is a mechanism that Docker uses to achieve the isolation of containers on the host.

when a command running in the Client to create a container using an image, the Deamon receive this command and search for this image , if he finds it he will start the creating , if no he will pull it from the Registry (Docker Hub), then create the container and run it.



3 - Dealing with images :

This documentation focuses on CLI operations.
At this point, it is necessary to understand that the Dockerfile is the foundation of Docker images.
Each command in a Dockerfile, when executed, creates a new layer so as a result, the Dockerfile builds multiple layers .
To interacte with images using CLI you must be familiar with those commands :

* Building an image

```
docker build -t <image-name> <project-path>
```

-t : to give the image an indentifier tag
You must give the correct project path to this command or navigate the project than using this command with " . "

* Tagging an image

```
docker tag <source_img>[:tag] <target_img>[:tag]
```

* you can create 2 images tag the same image by changing the name , or keep the name and change the version number .
* before pushing an image into Docker Hub you have to tag the existing image with the tag <username/image-name>

* Running an image

```
docker run -d -p 8080:80 <image-name>
```

* This command runs a specified image by the name of the image.
* -d : this atribute provide deattaching the terminal after conatiners lunched which make user able to use the CLI .
- p : to attach the client port (ex : 8080) to the container port (ex: 80) to enabling communication with the container

* Pushing an image to Docker Hub

```
docker push <username/image-name>
```

* As we mentioned before , the image must be tagged before pushing

*** Pulling an image from Docker Hub**

```
docker pull <image-name>
```

* image name must includes provider's name sometimes

*** Search an image**

```
docker search <image-name>
```

* a list of images will appears , just choose the right image with image ID

***Seeing image list to**

```
docker image ls
```

***Seeing image list to**

```
docker image history <image-name>
```

* Docker will show all the layers of the selected image

*** Building an image from existing container**

```
docker commit -m <comment>  <container-name>  <new-image-name>
```

3- Dealing with containers

In contrast of images , there is not much to do with containers exept some opertion such as :

*** create & run a container**

```
docker run <image-name>
```

* This previous command run an image whch lead to create a new container than run it

*** viewing runnig containers**

```
docker ps
```

* use -a to see all the existing containers

*** Stop running container**

```
docker stop <container-name/id>
```

*** remove a container**

```
docker rm -f <container-name/id>
```

* - f : force the opertion to avoid problems

*** Name a container during creating**

```
docker run --name=<givenn-name> <image-name>
```

* By default Docker give a random name for containers , you can custom the name by this command

*** Access to the container**

```
docker run -ti <image-name>
```

* During the creation use this atribute to create a terminal for the container to maintain opertions on it

In sevreal cases , we interacte with applications requires more than 1 container (container for each service) , so to run multiple containers in the same time we use " compose " option :

*** Runnig multiple containers**

```
docker compose up
```

* Notice that you have to navigate the project than tap this command

*** Stop multiple containers**

```
docker compose down
```

*** Direct upload of changes**

```
docker compose watch
```

* we use this command in the development cases , when want to save changes automaticly and upload it to the container without building everytime

4- Dealing with Dockerfile

The Dockerfile is the foundation of image creation.
This file contains the application source code and includes instructions that define the application dependencies and the environment, such as operating systems, frameworks, and necessary libraries.
The Dockerfile is based on a shared image called the base image, which forms the base layer of the image.
After that, each instruction in the Dockerfile creates an additional layer.
The core instructions are:

FROM <imag>[:<tag>]

▪ Should be the first one, this instruction determine the base image to start with , provide the base image name then you can provide the version if you want , base image can be an OS (ubunt..etc) or basic environement as (Node, python .etc)

WORKDIR <path>

▪ should be the second one in order , determine the container's directory that instructions must applied in it , if the path does not correspond to any existing directory in the container , system will create a new one using that name

COPY <source-code> <destination-path>

▪ This instruction used to import source code into the container , it taken the source code path as parameter , you can upload one or more files (. for all) , then precis the destination directory path on the container , globaly WORKDIR will take effect if it used

RUN <command>

▪ it used to run a specified command in the containers during their creation , such as installing the npm packages , it used to setup softwares and configurations

CMD ["executable", "param1", "param2"]

▪ it used to defined a default command executed when container launched such as CMD ["node", "index.js"]

We can use the docker init option to create a Dockerfile dynamicaly using CLI , navigate the project then use the command "docker init"