**Docker**

Docker fundamentals :

* Docker is linux based.
* Docker handles containers which contains applications (Dependencies…)
* It is lightweight compared to a VM.
* It does not work well for every app out there for example, there may be cases where an app runs better if it is not containerised.

Install Docker on Ubuntu :

sudo apt install docker.io

systemctl enable docker

systemctl start docker

**Run a container :**

sudo docker run hello-world

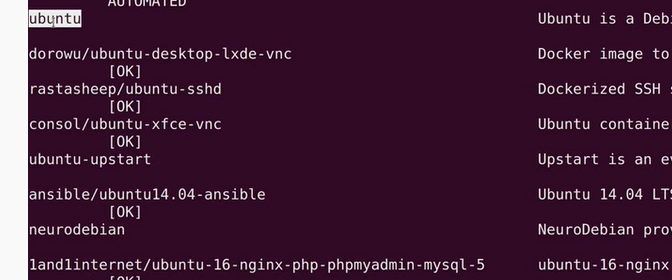
**Remove the sudo for using docker :** It would be nicer if we remove the sudo demand for our docker commands, to do that :

sudo usermod -aG docker elie

So here, we added the docker group to elie. A logout and login must be done.

Docker images :

**See container images :** The command below will show us images downloaded and stored locally on our machines. docker images

**Search an image :** To seach for a specific container image we can search by using a keyword docker search ubuntu

Here we can see that there are some repositories like ‘dorowu’ where inside of it is the actual contrainer image. We can also see in the first line, the container ‘ubuntu’ which is directly the image that we can use so we will install it.

**Download images :** The command below will let us download a container image

docker pull ubuntu

docker images

Now, we will see the the image ‘ubuntu’ that is stored locally on our machine.

We could have skip this step by using the following :

docker run ubuntu

This will automatically download the container and then run it but we learned the above step so that if we wanted to download a container image and run it later.

N.B :

In fact, docker containers quit if they have nothing to do so that there will not be a waste of ressources etc… So for example, when we ran the ‘ubuntu’ container, we will stay in the bash shell and the container will quit after running because there is no application that is running inside of this container.

**Running images :** To display the running images on our system we can use the command :

docker ps

We can can also have more details and see the history of containers, so we can see which container images where running and when it stopped :

docker ps -a

**Access images shell :** We can access the shell of a specific container so that we could launch commands on that container…

docker run -it ubuntu /bin/bash

or

docker run -it ubuntu

This will return a bash shell for that specific container so we can launch commands. The ‘-it’ creates an interactive shell inside a container.

N.B :

Containers are not stateful, so when the container dies or quits, everything added to this container will be erased, so when we relaunch the container (The container will be built from the image), only the apps or things that comes by default with the container image will be present. So the image is the blueprint.

**Make containers persist :** In this section, we will learn how we can keep containers running in the background so that it never quits and make access to shell of container.

Method 1 :

docker run -it -d ubuntu

The ‘-d’ means deamon mode and deamon means a service that runs in the background. However, here we will not have the acces to the shell of the container because we used the

‘-d’ option that will send the container to the background directly.

So now, we need to find a way to interact with the shell and to do that we first launch the following command to see the container ID for our container :

docker ps

Then we will do the following :

docker attach 454dz4z484

So here we attached the container with the specific ID to the shell so we gain access to the shell of the container. We could also only put the first two numbers of the container ID because docker is smart enough to know that we would want to reference this container.

However, here when we quit from the shell of the container, the container will still quit so to fix that we access to the shell of the container with the attach command and then we hold ctrl and pres p and while you’re still holding ctrl press q and now when we quit from the shell and run ‘docker ps’ we will see that our container image is always running in the background.

So evertime we attach to the container shell and would want that the container stays running we should use the combination of keys described above.

Method 2 :

docker exec -it container\_id /bin/bash

We will have access to the shell of the container and when we quit we will still have the container running so no need to do the combination keys.

We can also run a command without accessing the shell of the container :

Docker exec -d container\_name touch /etc/TEST

So here, a file named ‘TEST’ will be created in the /etc directory of the container.

Access containerized apps :

docker run nginx

Here, we launched the nginx container image and we are stuck to the command line of nginx because by default nginx stays running when we first run it.

docker run -it -d -p 8080:80 nginx

So here, we created the interactive mode (shell) for our container and we sent it to the background. Moreover, the nginx listens on the port 80 by default but it is not accessible from outside of the container so we will link the port 80 of nginx to the port 8080 of our machine. So we are opening the port 8080 of our physical machine and now we can access nginx server by going to the browser of our machine and accessing ‘localhost:8080’

N.B :

For the case of nginx container, we do not need to use the combination of key clicks to say to this container to stay running even if we quit from it because nginx by default stays running by default.

**Stop a container :** To stop a container we can do the following :

docker stop 45ce56de

**Restart a container :** To

docker run -it -d -–restart unless-stopped -p 8080:80 nginx

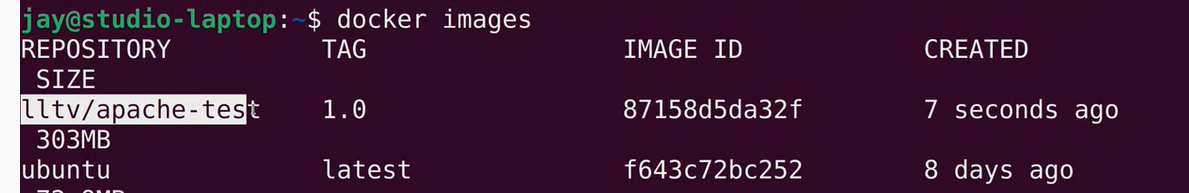
The following example starts a nginx container and configures it to always restart unless it is explicitly stopped or Docker is restarted. When the container is stopped (manually or otherwise), it is not restarted even after Docker daemon restarts.

A) Create images :

Now let’s say that we have a container image that is running and we have done some changes to that image, now we want to create a new image that clones our running image so that our changes will be saved in a new image that we can reuse later.

docker commit 15dzz5dc1 lltv/apache-test:1.0

So here, we created a new image under the lltv repo and we give it a tag which is the version of our apache so we can see the image that is stored locally on our machine.



docker run -it -d -p 8080:80 lltv/apache-test:1.0

Here after we run this new image and stopped the other image because we want that port 8080 be free and used by this container, we will notice that when we try to go ‘localhost:8080’ it will not work because this container image is an image that we created and not the one that we downloaded so we need to manually add an entrypoint for this container image.

docker commit –-change=’ENTRYPOINT [“apachectl”, “-DFOREGROUND”]’ 15dzz5dc1 lltv/apache-test:1.1

docker run -it -d -p 8080:80 lltv/apache-test:1.1

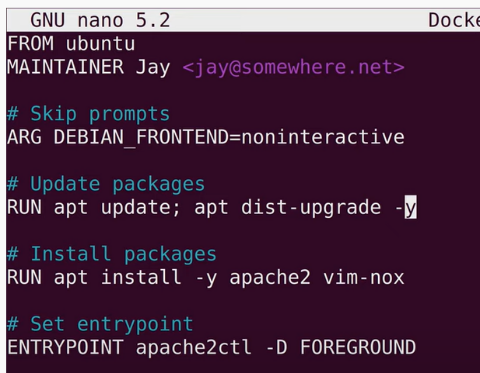
So now, when we launch the default page of the apache server we will see the page.

**Automise the process : It is the role of jenkins**

We will create a docker file

nano DockerFile

We will write inside of the following text file the following :



Now we will build the image by using this DockerFile :

docker build -t lltv/apache-test:1.2 .

N.B :

- In the photo just above we can see the use of ‘RUN’. In fact, ‘RUN’ commands will be executed on the container. So for example, if we want to copy something from our local machine to the container while building our image, we can use ‘RUN’ and then ‘cp’ but it won’t have access to our local machine because ‘RUN’ refers always to the container so in that case we can use ‘COPY’ instead of ‘RUN’.

- The docker file must always be called ‘Dockerfile’.

**Delete an image :**

docker rmi 15dzz5dc1

It will not work because we had a container that uses this image. Even if the container is stopped, we can not delete the image before removing the container in the first place.

docker rm 6ddgz45j

docker rmi 15dzz5dc1

Dockerfile parameters :

**RUN :**

Always executes in a new layer and created a new image layer on top of our base image that we created. Used to install packages to the container when building from the base image.

**CMD :**

CMD is just one command and it is the entrypoint, so we can not enter multiple commands to CMD like we do with RUN.

**ENRTYPOINT :**

ENTRYPOINT is different from CMD where the value that we enter in ENTRYPOINT will be the parameter of the value we will pass with the ‘docker run’ command.

Example:

ENTRYPOINT [“/bin/echo”]

docker run -it image\_name hello

So in this example, there will be an output of hello because hello will be the parameter of the ENTRYPOINT value.

**WORKDIR :**

RUN mkdir new\_folder

WORKDIR new\_folder

RUN touch new\_file\_in\_new\_folder

So the WORKDIR will let us execute the commands in that specific directory

**ENV :**

Is used to define environment variables when creating the image

ENV TARGET\_DIR /etc/new\_folder

WORKDIR $TARGET\_DIR

**USER :**

Specify the user that we want to launch the image with (We can create users in a container image so for example we create the image then we connect to the shell of the image, we create a new user, we commit to create a new image) and then we add the following to our Dockerfile before building it :

FROM elie/ubuntu

USER elie

**ADD :**

Let us add files/directories from our environment to the image we want to create

FROM elie/ubuntu

ADD <http://wordpress.org/latest.zip> .

So the following will copy the zip to the home directory (We specified that with the .). In fact, ADD decompress by default the compressed files that we downloaded.

**COPY :**

This instruction is like ADD instruction but can only do one thing (copy things)

**LABEL :**

This instruction let us add metadata to the image that is being constructed. These metadata are key/value pairs

FROM elie/ubuntu

LABEL location=”Grenoble” type=”ubuntu-server” role=”web-server”

**ONBUILD :**

This is executed only when we are building a new image from an old image.

old image :

FROM elie/ubuntu

MAINTANER elie

RUN apt-get install -y apache2

ONBUILD RUN apt-get update

new image :

FROM elie/ubuntu

RUN echo hello

Here when the new image will be built, the ‘apt-get update’ will be executed.

2 types of forms while using Dockerfile parameters

**Shell form :** RUN apt-get update

**Executable form :** RUN [“apt-get”, “install”]

**Shell form :** CMD echo Hello

**Executable form :** CMD [“/bin/echo”, “Hello”]

B) Docker logs :

docker logs 15dzz5dc1

or

docker logs container\_name

Container name :

We will change the name of the container when we create it with the ‘docker run’ command

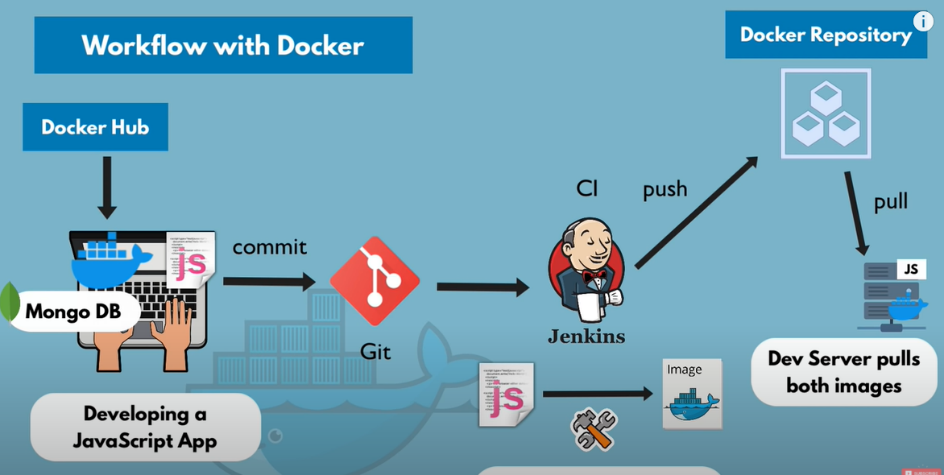
docker run -it -d -p 8080:80 –-name redis-older redis:4.0

If we do not enter manually the name of the container while creating it, it will get a random name.

Docker run vs Docker start :

So with ‘docker run’ we are pulling the image and then we are creating and starting the container but with ‘docker start’ we are just starting the container that we already created with docker run.

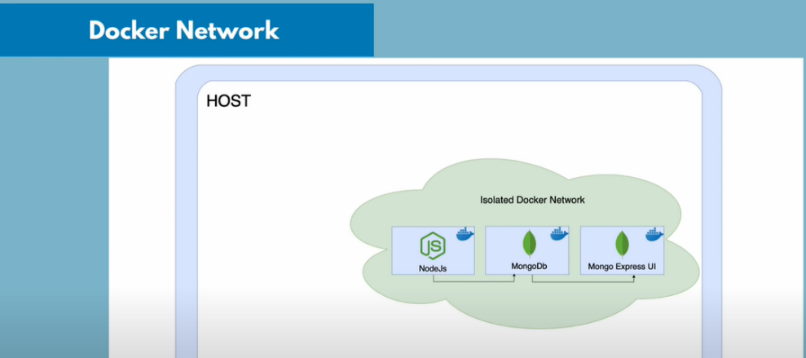
Docker in a real environement :



So in a real environnement, let’s say the developpers are working on a project, the files or source code of the project will be pushed to git. Then jenkins will take these files and build a docker image for that specific project with the needed dependencies… This docker image will be pushed to a docker repository and then the server on which we want to run this application will pull the docker image from the docker repo to run it. The server will use also docker hub to pull for example mongo db which is necessary for our application to run so it uses both repo (local repo for our app and docker hub for the other dependencies)

Docker network :

Docker creates its isolated docker network so when I create 2 containers in the same docker network. Containers inside a docker network can talk to each other by just using the container name so without port mapping or anything. And the applications outside the docker network will be able to communicate with them by using port mapping. So in our case when we will create an container for our javascript app for example, it will be placed in the same docker network.



So docker by default generates docker networks and we can see these networks by doing :

docker network ls

In fact, by default 3 networks are created when launching a new container

The network ‘bridge’ is the default docker network which is the ‘docker0’ that we visualize when executing ‘ip a’ on our physical machine. So when we create multiple containers without specifying the network of those containers, these containers will be in the same network so will have same subnet but each one of them has his own ip address.

**Create our own docker network :**

We can create our own docker network (we can specify what type of network we want with the ‘-d’ option but if we do not specify, it will by default create a bridged network)

docker network create mongo-network

**Remove our own docker network :**

docker network rm mongo-network

**Create our own subnet for our docker network :**

docker network create -d bridge –-subnet 172.25.0.0/16 my\_network

ou

docker network create –-subnet 172.25.0.0/16 my\_network

**Run our container inside of the created network :**

docker run -it -d -p 27017:27017 –-name mongodb –-net mongo-network mongo

**Connecting docker networks :**

docker network connect new\_network container\_name

So here we have 2 networks each one of them has a container inside of it, we connected a container from a network to another network. In this case, when we launch the command ‘ip adr show’ on the container we will see 2 interfaces, one linked to the default bridge network and the other one to the other network.

**DNS integration:**

DNS can make name resolution of containers in the same network. However, this fonctionnality does not work for the network bridge by default. That’s why when we don’t use user created networks, we must use the ‘link functionality’ to link the containers in the same network.

**Disconnecting docker networks :**

docker network disconnect new\_network container\_name

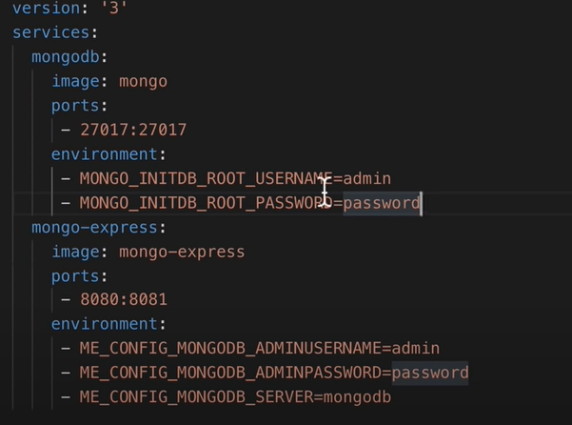
**Connecting container in same network :**

We can use the functionality ‘link’ to connect containers within the same network but this is only used for the bridge networks that is created by default. To explain, if we use user defined networks we do not need to use the ‘link’ option because the DNS will be able to resolve the names of the containers inside the network.

Automating creating multiple docker containers :

So that we don’t write the full command line to create a container from an image with all the parameters needed, we can create a yaml file to automate that and use ‘docker compose’ to build the container needed from this yaml file. So here, we added our first container which is adressed by a variable that we called‘mongodb’(It is not the name of the container that will be created, in fact when we run this file, a name will be created for this container by taking the variable name that we added here and just adding prefixes and sufixes) and we added some parameters to it, we can add more containers by adding it just below the mongodb which is the first service(container) and we don’t need to take care of the docker network because by default the ‘docker compose’ will put them inside of the same network.

So here, we added ‘mongo-express’ which our second container in the same yaml file.



Now, we will actually run the yaml file to create and launch these containers :

docker-compose -f mongo.yaml up

We specified our yaml file name and ‘up’ is to run these containers. So here a default docker network will be created with a random name.

**Shutdown multiple services :**

docker-compose -f mongo.yaml down

Here, we shut down the different containers that were running and also the docker network that was created for them.

Wrap Up :

So that we do not confuse between the docker file that we created and docker compose we will explain the following :

In fact, we will add to our yaml file the name of the image that we build by using the Dockerfile. So docker compose will create for us several containers and one of them is a container for the image that we have created.

Docker Volumes :

docker run -it -d \

> -p 27017:27017

> --name mongodb

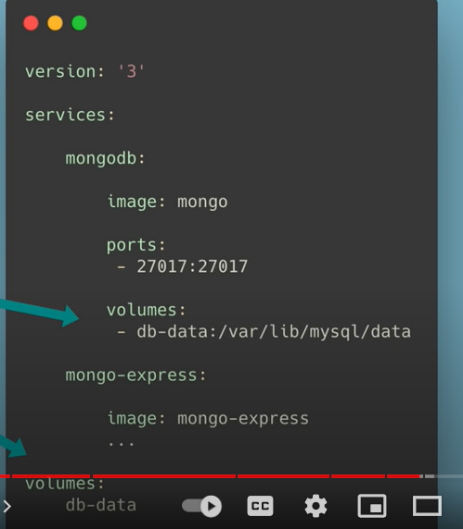
> -v /home/mount/data:/var/lib/mysql/data

> mongo

So here, the ‘-v’ let us do this link so the ‘/home/mount/data’ is the emplacement on our local machine and the ‘/var/lib/mysql/data’ is the emplacement on the container.

This type of volume is called host volume and the main caracteristique is that you decide where on the host file system the reference is made.

The other method is to define that with the docker compose.



The ‘db-data’ is the name of the refernce and after that we put the path of container

And in the end we put the volumes that we defined.

The ‘db-data’ is the name of the refernce and after that we put the path of container

And in the end we put the volumes that we defined.

**Create volumes:** We can create volumes with the following command

docker volume create new\_volume

**List volumes:** We can list volumes created by using the command

docker volume ls

**More details on volumes:** We can have more details of volume

docker volume inspect new\_volume

**Docker volume location:** By default, Docker create automatically volumes in /var/lib/docker/volumes

**Copy from container to volume:** We can copy files from a specific container to a volume

cp docker-nginx/html/index.html /var/lib/docker/volumes/new\_volume/

**Mount our container to the created volume:**

docker run -it -d \

> -p 27017:27017

> --name mongodb

> --mount source=new\_volume

> mongo

Containers more info :

We can get more info on a container rather than just using the ‘docker ps -a’

docker inspect container\_name

See working processes :

We can see working processes on a container by using following command :

docker top container\_name

Check commands executed on image :

We can see what commands were executed on a specific image :

docker history image\_name

See what ports container is using :

docker port container\_id

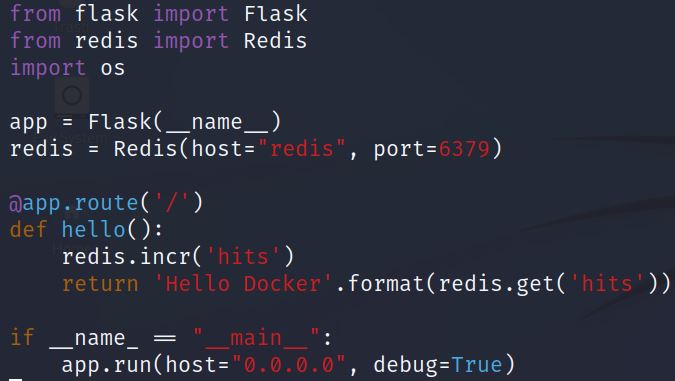
Container Forced stop :

If you are having troubles stopping a specific container, you can use the following command:

docker-compose kill container\_name

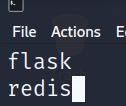
Example of creating an app with containers :

**a) Create the web application with python :** We will create a python file named ‘app.py’ in /composeapp/app and write the following code inside of it :



In this example, ‘redis’ is the hostname of the container redis

Now we will create the ‘requirments.txt’ file inside the same directory /composeapp/app

****

**b) Create the Dockerfile :** We will create a python image with the directory of the app



So here we will be copying the ‘app’ directory files from our machine to the ‘composeapp’ directory in our container then we will go to this directory and install the ‘requirments.txt’ file.

Then we will build our image :

docker build -t python/composeapp .

**c) Create the docker compose file :**

****

So here the docker-compose yaml file define 2 services : web and redis

- The service web use the image that we created that contains the application files and is based on python. Then we launch the command “python app.py”. We associate our directory “/composeapp/app” that is residing on our local machine to be the volume for the “/composeapp” directory on the container that contains the application files. We are also making a link to the ‘redis’ image that we defined also in this yaml file.

Example of installing Netbox with all needed packages

