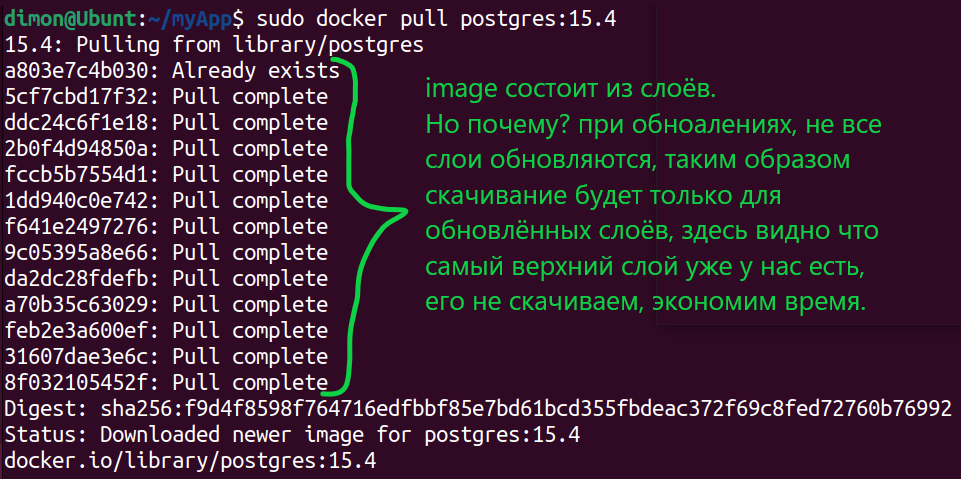
**Dockerfile** - does not have any extension, this file will have the instruction, in that file we provide all the instructions/dependency that we want to be in a VM, (it has similar purpose as .java files) .

**build** - it is a **process** of creating an image from the **Dockerfile** (it is similar as a run time in Java when the JVM converts .java files into .class files). In another words Docker will read **Dockerfile** and will create a VM accordingly.

**image** - it is a package that can be moved around. It will have the features that you choose in docker hub, or requested in Dockerfile**.** It is not a VM its something that is ready to use and describe how the VM should look like. It is not running. It consist of the layers (layers is also images), when we download new image for host machine image will look like this



Each image has a version of a tags, if you do not specify version the docker will run the latest version. Latest dos not mean the last one it means the last stable (lst)

**Tag** - it is something as a version of the image. For example you have an image with java 10 and then you add some minor changes and basically you did not change the whole image but steel made some updates so in thes case you can use tag.

**Container** - A way to package applications with all the necessary dependencies and configurations. it's an actual running VM, it's created from the image. It is possible to create multiple VM from the same image. For example we can run different versions of the application in different containers. It is similar to objects in Java that have been created from the same class (we can create 1 or more objects from 1 class).

**Docker Hub** - images repository.

**Port Mapping** - every application has its own port for communication. For example if to install Jenkins on a PC it will have its own port, for example 80, then we install for example Postman and it can not have the same port on this PC as Jenkins, port 80 is occupied, so the OS will assign port 70 to postman, in order to keep the communication safe.

In case of Docker we have VM and Jenkins installed on it and it will have port 80. Then on the same PC with Docker we activate one more VM with Postman and that VM with postman knows nothing about the first VM and gives to Postman port 80 as well. So there is a potential defect/problem.

In order to expose the application via the internet we have to do a port mapping. The host PC will give the VM#1 a port for example 80, and then Jenkins`s port would be 8080. Then, since the host PC can not give 80 to another VM it gives to it port 70, so Postman would have port 7080.

**docker run -p Host\_port:container\_port image** ->This is the format for port mapping.

For example:

docker run -p 4444:80 mongo

port mapping is used when you want to see that containers run correctly in a browser. In that case you assign a port of service to your port that you want, and after this you know which port is yours and you check this port manually. For port mapping example we can do **localHost\_port:container\_port.**

Чёта я читаю и нихера не понятно

В общем каждый сервер который мы раним в контейнере имеет свой порт, но он доступен в рамках сети самого контейнера, т.е. извне (в том числе и хост машине) к нему не подключится. Чтобы иметь возможность работать с сервером который работает в докер контейнере надо зделать что-то типа мостика, по которому мы привязываем свободный порт из хост машины к порту на котором работает сервер.

**docker run -p [HOST\_PORT]:[CONTAINER\_PORT] [IMAGE]**

Тогда обратившись к HOST\_PORT мы сможем работать с сервером на докере.

**volume mapping** - when we run the container, for example with tests, it has its own results/output, and we might want to use these results in the host machine, but since it's on the container/vm, when the container stops all the results will be lost.

And this is the time when volume mapping helps us to bind/connect the directories in the VM with the directory in a host machine. We can imagine it as a tunnel between these directories. Whatever you do in one directory (create files, or new directories…) shows in another, if you drop the file in one directory you can view it in another. So volume mapping is the concept through which we can share files between the host and the container. So when we run tests we can save the output in the directory in the container and that directory we can volume map to the directory in the host, so we can keep/use the output. After we stop the container, everything would be removed and the directory as well but we still have the host directory with all information.

Realization:

It may be good practice first to create directory on a host machine (this is the first path) and when you write path in the container it might not even exist, Docker will create it.

For example:

**docker run -it -v /C/for\_experements/docker\_experements/DockerFile/vins\_guru\_framwork/results:/home/selenium-docker/test-output some\_name\_of\_the\_prototip\_app**

**ещё пример :**

**docker run -it -v /C/for\_experements/docker\_experements/DockerFile/vins\_guru\_framwork/test\_output:/home/selenium-docker/test-output delete\_me\_i\_am\_for\_experement\_only**

обрати внимание, что после диска С нет двоеточий ‘:’ иначе возникает ошибка и папка не создаётся

**docker network** - we can run the multiple containers on a host machine and these containers are independent to iche other they do not know about each other they basically can not communicate via the internet. Docker network allows to establish internet connection between these VM.

Realization:

docker network create *name* -> to give the name to our network!

docker run --network=*name* nginx -> we attaching nginx to the network (adding vm to the network)

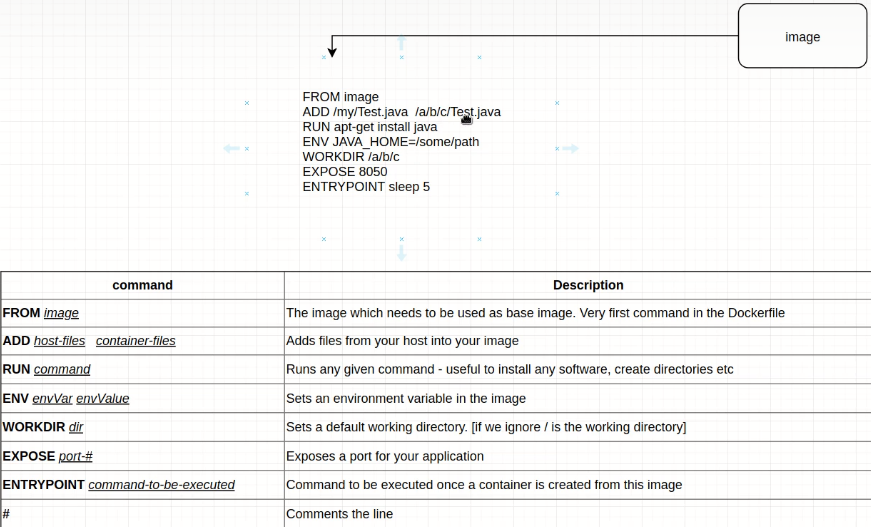
docker run --network=*name* ubuntu -> we attaching ubuntu to the network

**How to create Docker image**

Docker hub - is the place where people share images. Docker also maintenance it official repo called ‘library’.

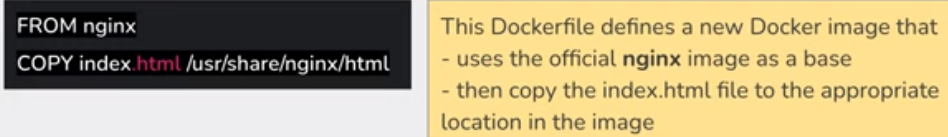
To create your own image we need to start with a Docker file (with the capital D) that has no extension. The very first line would be :

**from** *image* -> this is the very first command in the docker file. The mining is that we are using existing image as reference, as the base of our image. For example we want java, but we want java on top of ubuntu, so ubuntu is our **from** image. So we need some base image and on top of that we add our staff.



It looks like we have 2 ways to create image:

1)-**using Dockerfile**. Dockerfile is a text file (you can edit it in any text editor) with instructions to build a Docker image.

When we run a Dockerfile, Docker image is created, when we run Docker image containers are created.

Here is example of Dockerfile

Создадим image на основе ubuntu23.10 и поставим на него текстовый редактор nano который не входит изначально в эту OS, таким образом получим свою кастомную image. ДЛЯ ЭТОГО НАДО ЧТОБЫ была спулена нужная версия image ubuntu её укажем в FROM.

A) Открываем его в редакторе (можно открыть редактор NotePad++ при сохранении не указывать расширение)

B) Создадим image на основе ubuntu:23.10 и поставим на неё текстовый редактор которого нет по умолчанию для этого прописываем:

FROM ubuntu:23.10

RUN apt-get -y update && apt install nano

сохраняем файл.

C) docker build -t nazvanie\_nashei\_bydyschei\_image put\_k\_Dockerfile ->создаёт image, важно указать путь где лежит Dockerfile.

2)-**using Docker commit**. How it works -> when we create a container and for example install some software (for example Git) on top of it, when we delete the container all software will be gone, and next time we have to create it again from scratch. But it is possible to use Docker commit to save the architecture of the present container as an image, so in the future we don't need to install all the programs that we had, all we need is run the image.

Realization:

**docker commit nameOfContainerYouWantToSave nameForThatImage**