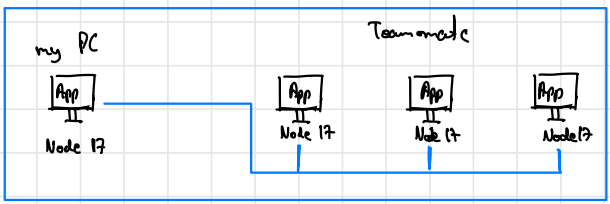
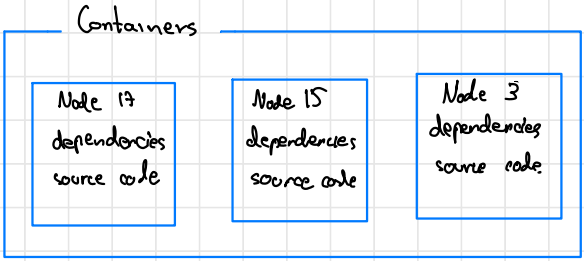
- **Docker**: is a tools to manage a container
- **Container**: is an object that contain all necessary environment setup for a specific application

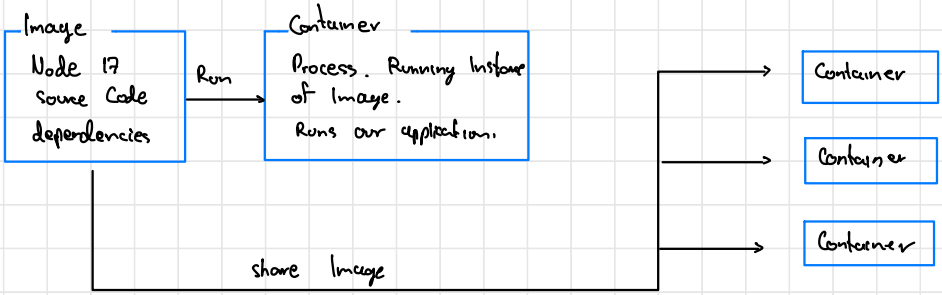


example: I have an app that requires a specific version of Node 17 to run, if I want to share my application to my friend, they also need Node 17 setup.

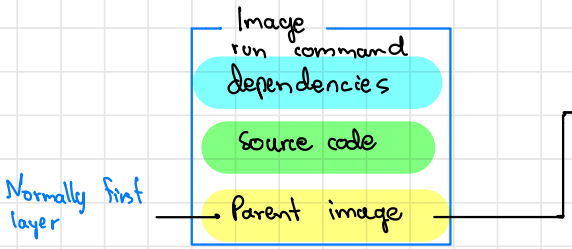


+ Container is similar to Virtual Machine but share host's operating system & faster.

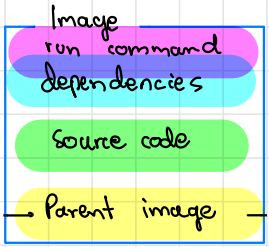
- **Image**: It is like a blueprints for containers.



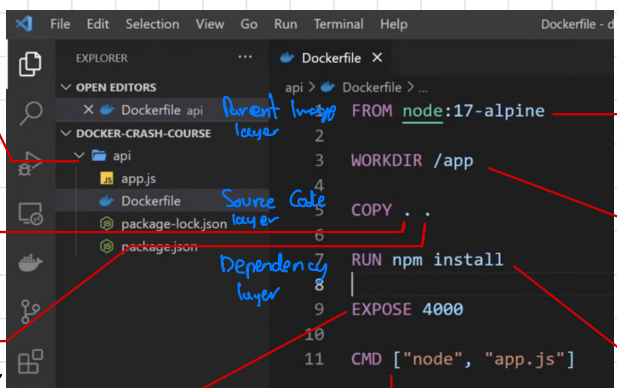
+ Layer of Image: Image is built of a lot of layer



- **Docker file** : is a file contain all instruction to Docker how to create an image with all the layer that we want (that we specified)



+ Creating Dockfile



directory of our application

this means copy api directory (current dir is api')

Copy to image root directory (we specified it as WORKDIR)

Install parent image: Node 17 on alpine linux distribution.

like macro. telling that every command will be run/install to this directory (this is image directory)

install all dependencies (all dependencies is specified in package.json)

- Specify a port
  (this is due to a specified port in api app')
  - this will be used to make port mapping

Specify what command to run at runtime (when this image is built)

+ Building Image from Dockfile

On terminal :

```
C:\Users\meng> cd api
C:\Users\meng\api> docker build -t myapp .
```

name of image

stand for tag

relative path to the docker file

- **Dockerignore**

If we have all dependencies / module installed in project tree, we don't want to copy these file to the image because we will use `RUN npm install` to install all dependencies any way.

Solution: .dockerignore

1. add .dockerignore to the project tree
2. add a name of a file or folder that we want to exclude in .dockerignore file.

- **Docker's command**

```
docker images                                   show all images that we have
docker run --name myapp_c1 -p 4000:4000 -d myapp:v1   run the image
```

`--name myapp_c1`: tag to give our container a name
`-p 4000:4000`: publish a container's port
`-d`: detach mode (won't block terminal)
`myapp:v1`: image's name

```
docker ps -a                                    show running container
                                                — show a non-running container as well
docker stop myapp_c1
```

either container's ID or name

```
docker start myapp_c2                           run a existing container.
docker build -t myapp2:v1 .                     build an image
```

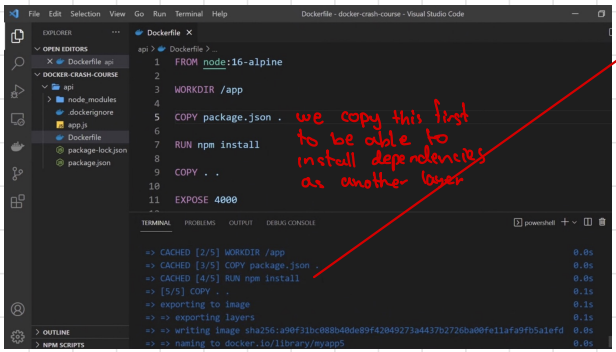
version — relative path to the docker file

```
docker image rm myapp -f                        delete image (work if no container use them)
```

myapp: image's name
-f: force to delete eventhough some container is using them

```
docker container rm myapp_c2                    delete a container.
docker system prune -a                          delete all image, container, volume.
```

Layer Caching



we made some change in app.js but no change in other place so the cached image from previous build is used

Volumns

- after building an image, you cannot change the image. If you modify some source code then you need to build a new image, This is not convenient because we cannot see a change suddenly after our modification.

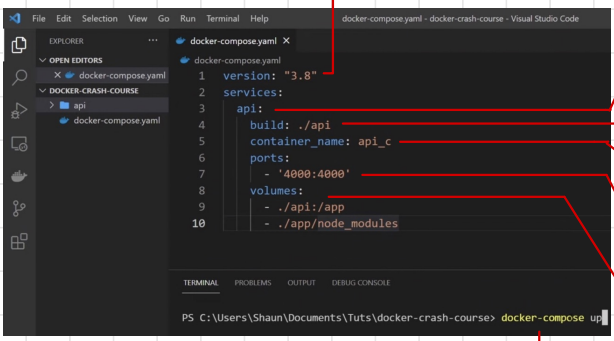
- Volumn can map a folder in our computer to a folder in the container so if we make changes on the source file, the container will response to that change too. but the image won't be changed any way. we still need to build a new image.

-v

```
docker run --name myapp-c_nodemon -p 4000:4000 --rm -v c:\users\api:/app -v /app/node_modules myapp:nodemon
```

- `--name myapp-c_nodemon`: name tag
- `-p 4000:4000`: port mapping for
- `--rm`: remove container after stop
- `-v c:\users\api:/app`: absolute path on our PC : container path — Volumn
- `-v /app/node_modules`: map this folder in container to a folder managed by docker. so it is persist.
- `myapp:nodemon`

- **Docker Compose** : similar to makefile, use to store all instruction how to run an image or multiple image.



`docker-compose down --rmi all -v`

stop a docker compose. it remove container automatically

`--rmi all` : remove all image created by it

`-v` : remove all volume created by it

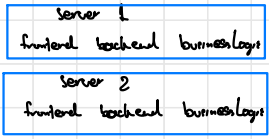
- **Dockerizing a React App**

step :
1. make docker file how an image should be built
2. make dockerignore file to tell docker to ignore our node_module folder when building an image.
3. Edit our docker-compose file to add the new project or service

Microservice Vs Monolithic

- smaller unit
- scale independently.
- smaller chance that if one of them goes down, the whole system goes down.
- It runs on open-source technology so no vender lock-in
- Since they are smaller in most case, they are easier to understand
- smaller, faster to deploy
- easier to scale.
- Calls between microservices will go through API

- Develop as one unit
- Deploy as one unit
- hard to scale.



Cloud Native