**Concepte generale**

* **Docker** – tool pentru a rula aplicatii intr-un container. Aplicatia din container are un full level OS-virtualization
* **Container** – un mediu in care este rulata aplicatia si in care se creaza un fel de zona izolata pentru acea aplicatie. Aplicatiile pot comunica intre ele, adica containerele pot comunica. Containerul e anume mediul in care aplicatia ruleaza. Containerul creaza propriile librarii, fisiere de configuratie pentru aplicatie, de care ea are nevoie. Deci, asa containerul creaza tot mediul necesar pentru o aplciatie ca sa ruleze, de aceea pe orice pc, aplicatia va rula mereu la fel pe un container.
* **Docker Desktop** functioneaza doar pe Linux, iata de ce, cand instalam docker pe windows, el instaleaza si kernel de la Linux.Deci, ea se foloseste de Kernel la Linux
* **Docker nu este o virtual machine. El comunica cu kernel la linux, si creaza containere izolate**
* **Atentie!Daca nu vrem sa instalam Docker Desktop, trebuie sa instalam manual Docker dupa mai multi pasi**

**De ce containere**

Atunci cand instalam o aplicatie direct pe sistemul de operare, apar niste probleme:

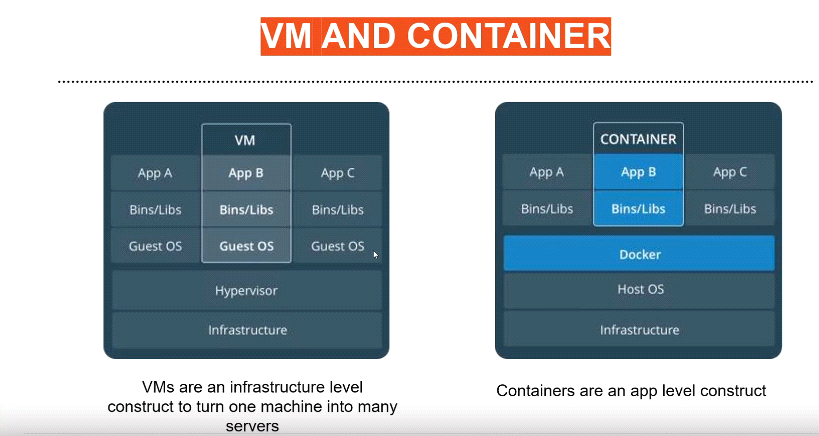
* Are acces nelimitat la CPU si RAM
* Full view la files system
* aceleasi environment variables, si deci poate 2 app au nevoie de a rula pe diferite Java Version, dar toate acceseaza aceeas JAVA\_HOME

Avantajele la virtual machine:

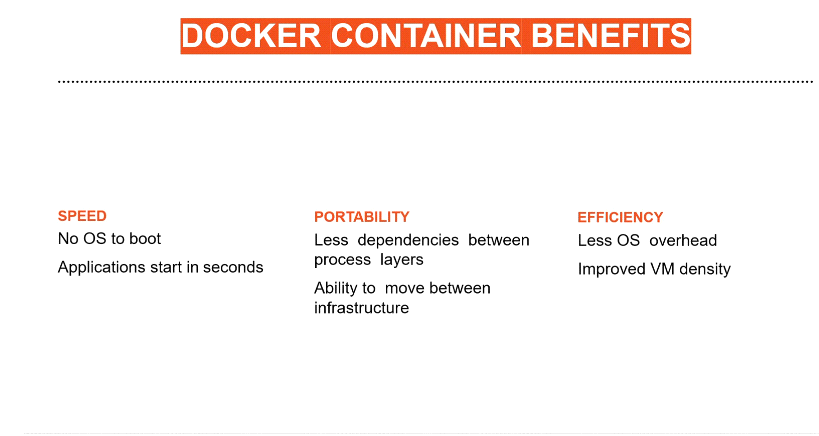
* fiecare aplicatie ruleaza in mod independent
* resurse limitate

Dar, nu e prea comod sa tot instalam un sistem de operare aparte pentru fiecare aplicatie, fie el si virtual.

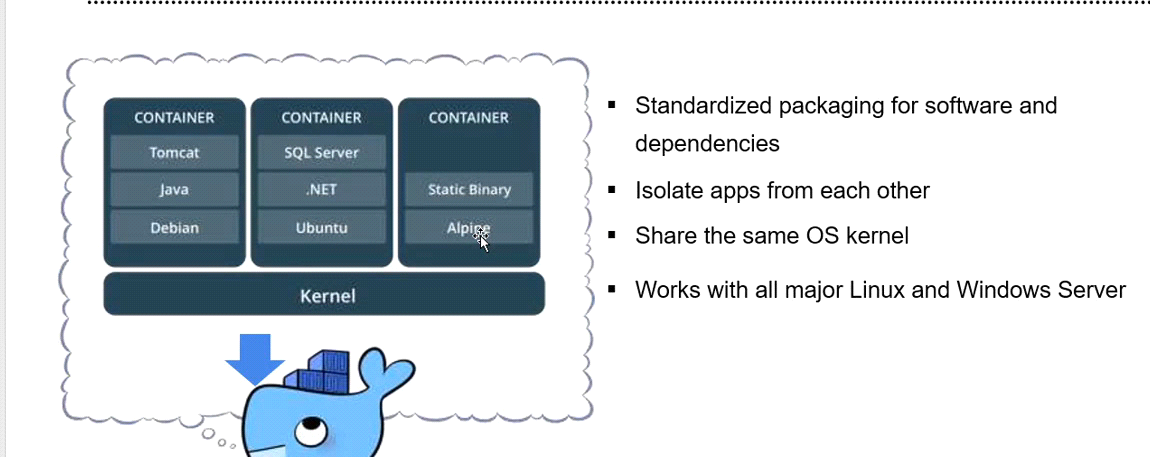
* Multe aplicatii sunt greu de configurat si instalat, dar cu un container, putem crea o imagine, ce contine toate aceste informatii, si avem deodata aplicatia ce va rula la toti la fel cu aceasta imagine



Vedem ca fiecare VM are neaparat un OS, dar fiecare Container nu are un OS separat. Ele toate rueleaza pe acelasi Linux, doar ca sunt gestionate de Docker asa ca sa fie ca si cum ele ar rula pe OS separate

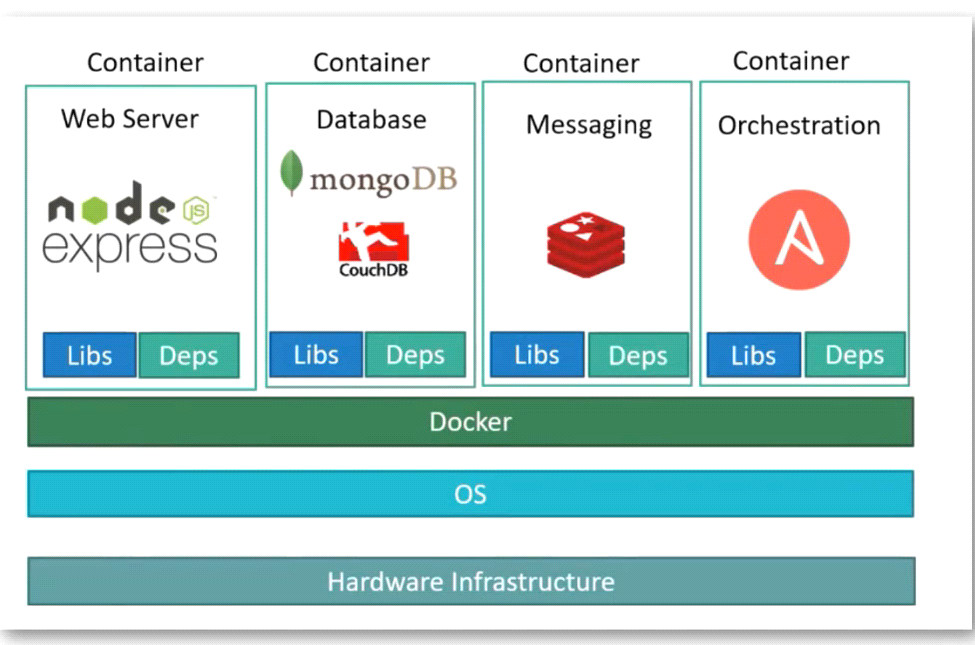


* Docker poate crea un virtual network pentru a comunica intre containere



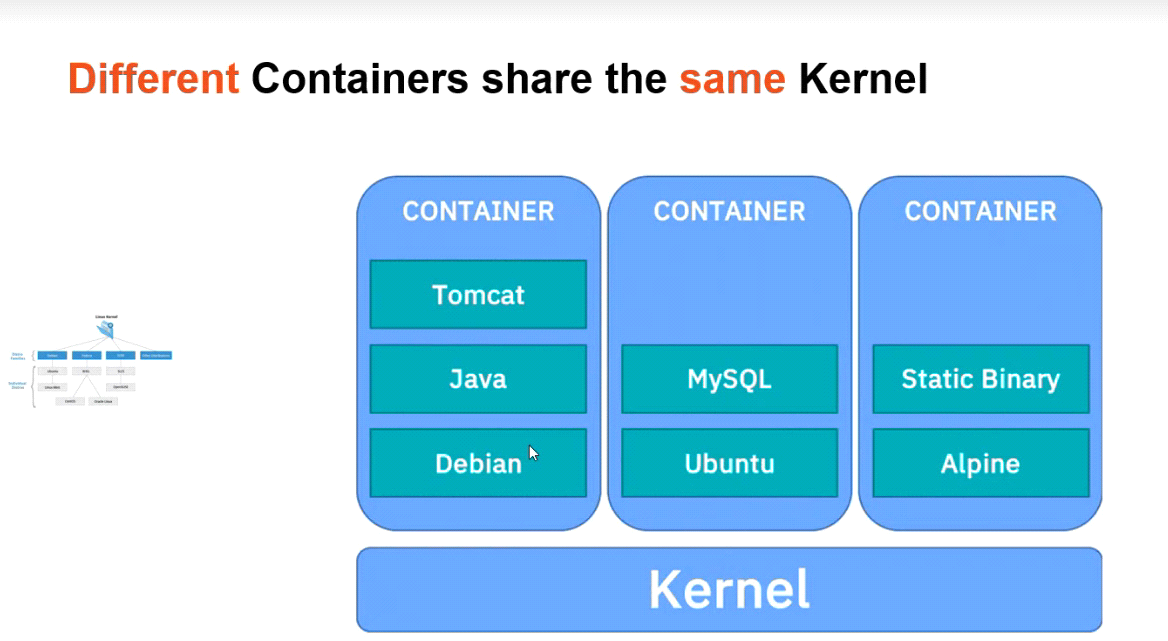
Fiecare Container poate avea instalat un fel de OS. Dar acesta nu va fi un OS separat. Toate aceste OS, ca Debian, Ubuntu etc. au acelasi kernel, Linux, insa ele mai vin cu ceva pachete, programe suplimentare care anume si le definesc ca OS. In Docker, cand ca si cum instalam un OS pe un container, noi pur si simplu instalam aplicatiile si pachetele lui caracteristice, nu instalam iar kernel, caci docker are deja kernel de la linux oferit.

* **Kernelul nu este in container!**
* Cu Docker nu trebuie mereu sa ne chinuim sa tot instalam aplicatii de care avem nevoie, dar ele vor fi direct instalate intr-un container si vor rula acolo separat:



**Cum funtioneaza Docker**

* Docker functioneaza doar pe Linux
* Windows are WSL(Windows Subsytem for Linux)
* WSL – este o chestie pe Windows care permite rularea la Linux environment fara a fi necesar de a instala o VM sau dual boot
* Deci, Docker va folosi WSL, fara a fi nevoie sa creeze vreun Virtual Machine, ci va folosi toate chestiile de la Linux
* WSL e ca o VM, dar nu chiar.
* Docker foloseste o **Client-Server** arhitecture
* **Client-Server** – un client(noi de ex) facem requesturi la un server, si serverul raspunde cu un raspuns
* Requesturile le facem in command line



Deci, daca instalam de ex Debian in Container, se vor instala doar acele aplicatii specifice lui Debian, dar Containerul va folosi oricum Linux Kernel, adica se va instala doar ceea ce face ca Debian sa aiba in plus fata de Kernel

**Docker components**

* **Image** – este ca un template sau ca si cum un jar. Ea contine coduri, configuratii, fisiere, comenzi etc. pentru a construi containerul si deci aplciatia
* **Container** – imaginea cand ruleaza, Tot ce face aplicatia din image se va stoca in Container. Containerul nu poate rula fara image!
* **Engine** – software care executa comenzile pentru containere(cele ce incep cu docker). Networking si volumes tot fac parte din el. Un component important e **Docker Daemon.**Anume Docker Daemon raspunde la requesturi
* **Registru** – stocheaza imaginile. Ca un maven repository
* **Control Panel** – ofera management pentru containere

**Comenzi**

* **docker run imagine**

Comanda run face 4 chestii(executa 3 comenzi):

* Cauta daca imaginea locala este prezenta
* Daca nu e, e luata de pe remote repo (docker pull)
* Apoi imaginea e folosita pentru a crea containerul pentru ea si a rula aplicatia ce o creaza (docker create)
* executa docker start

Daca o aplicatie termina executia, containerul e oprit, dar continua sa existe si poate fi pornit

Vom fi deodata conectati in container

**docker run helloworld** testeaza doar

* **service docker start** – porneste docker
* **service docker status** – vedem daca docker ruleaza sau nu
* **docker run --rm Image** – creaza containerul si la terminarea ciclului sau de existenta, el e sters
* **docker run -it Image** – creaza container si deschide deodata un bash shell pentru a interactiona cu el
* **docker ps** – arata containerele ce ruleaza. ps(process status)
* **docker ps -a** – arata toate containerele, ca ruleaza sau nu
* **docker container start numeContainer** – porneste un container existent
* **docker start numeContainer** – tot porneste un container existent. Deci, cuvantul container e deprisos, dar logurile sale nu sunt afisate
* **docker start -i container** – activeaza interactive mode pentru container
* **docker rm Nume** – sterge container
* **docker images** – lista de images instalate
* **docker rmi Nume/Id** – sterge imagine
* **docker pull NumeImagine** – preia imaginea din remote repo, dar nu creaza un container
* **docker pull NumeImagine**:latest – anume acest :latest punce cea mai noua versiune, dar nu e recomandat, caci versiunea se poate schimba
* **docker container create --name <Nume> --network <NumeRetea> -e CUSTOM\_ENV\_VAR=NumeAles -p <PortDin:PortIn> <NumeImagine>**

Cu portul de ex 8080:2021, adica portul 8080, cand va fi folosit, va fi inlocuit cu 2021

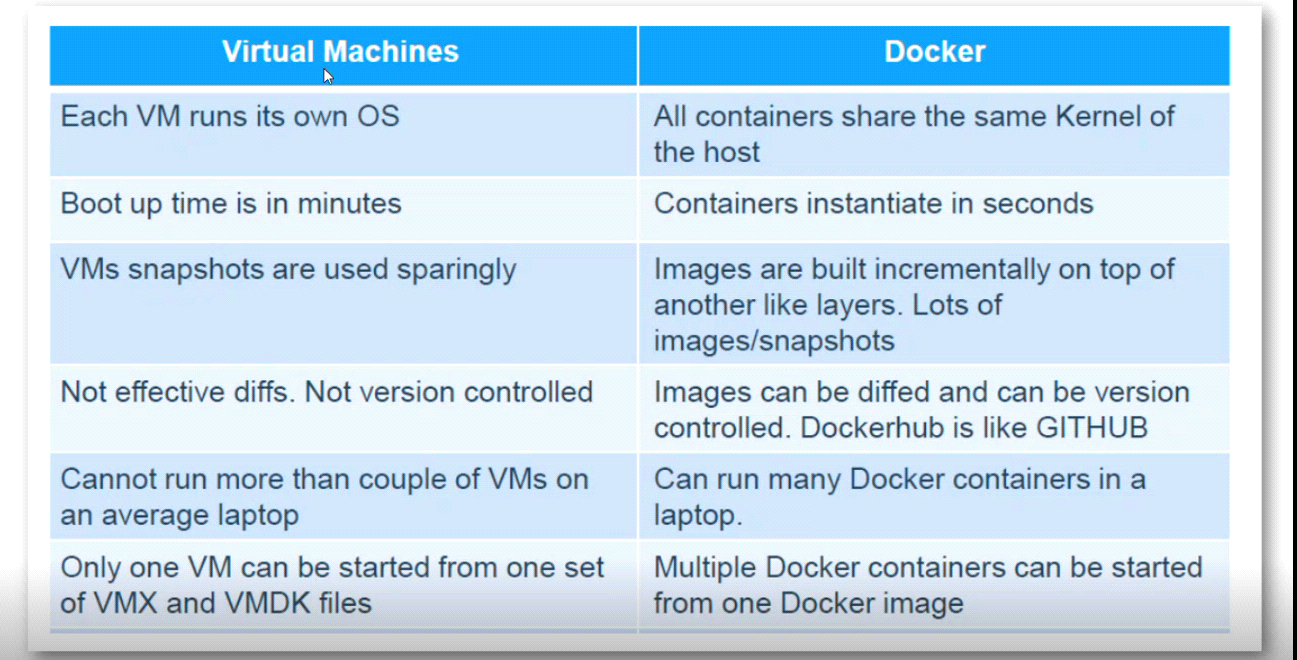
-e CUSTOM\_ENV\_VAR=NumeAles creaza o variabila, dar putem pune ce vrem, si MYVAR=Ceva

* **docker container inspect NumeContainer** – arata info despre container
* **docker image inspect Imagine** – arata infor despre imagine
* **exit** – iesim din container
* **docker diff NumeContainer** – arata un fel de loguri la container, ce s-a pus, ce s-a sters etc. Acum, chiar daca containerul va crea o imagine pe baza la ce are in el, noul container creat cu aceasta imagine va vedea logurile vechi, de la container parinte
* **docker commit ContainerName NumeNouaImage** – creaza o noua image pe baza la container
* **docker exec ContainerName <command>** - executa o comanda intr-un container ce ruleaza
* **docker tag OldImgId newNameImg** - modifica numele imaginii
* **docker login** - ne logam pe docker hub
* **docer push** - pune img remote

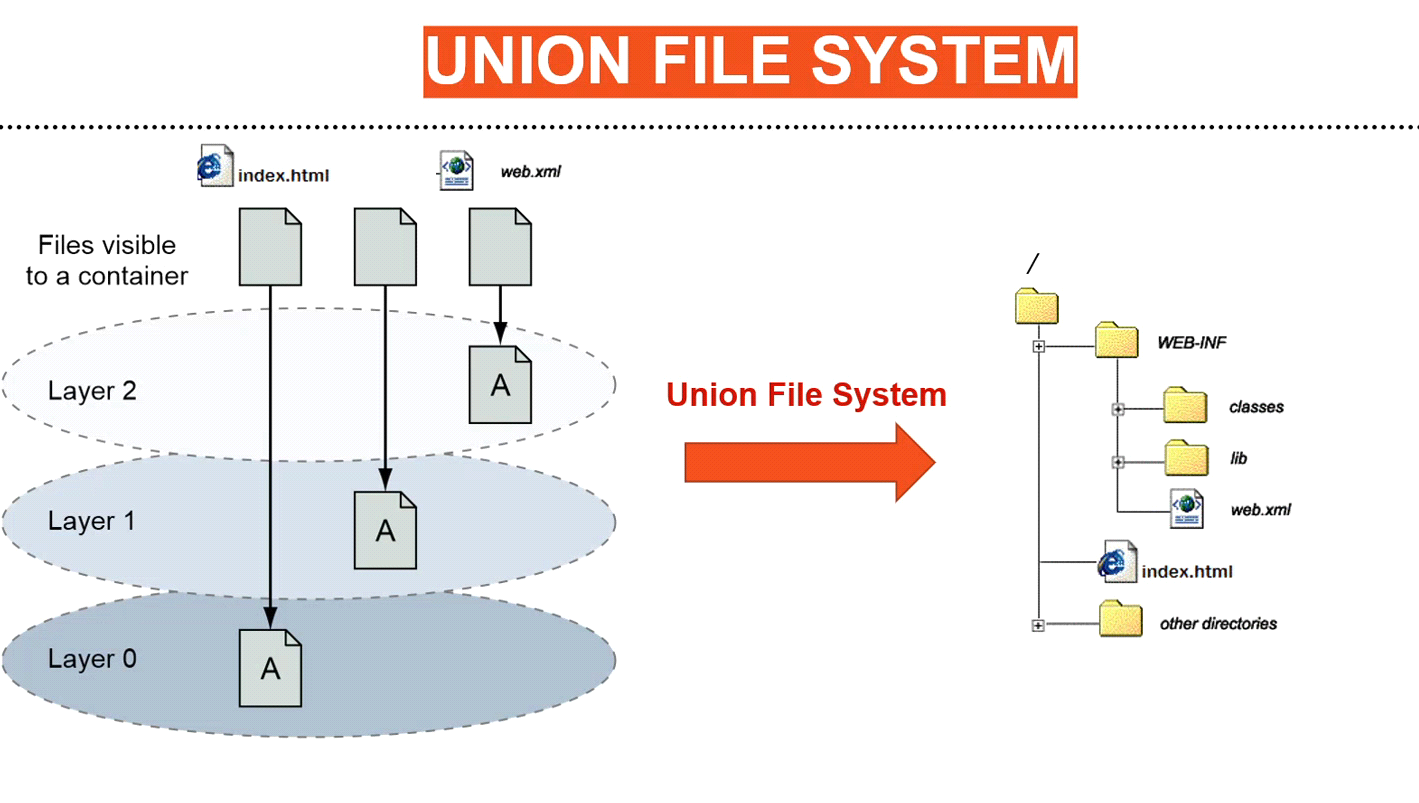
**Image vs Container**

* Image este ca o clasa, un model pentru crearea la container
* Container este ca o instanta la image, care ruleaza chiar
* Putem crea o imagine pe baza la alta imagine chiar

**Docker vs VM**



**Union File System**

* 

Fie ca avem o imagine ce contine fisierul index.html. Ea e la Layer 0

Acum, pe baza la imaginea de la Layer 0 mai cream o imagine, ce mai contine de ex Java

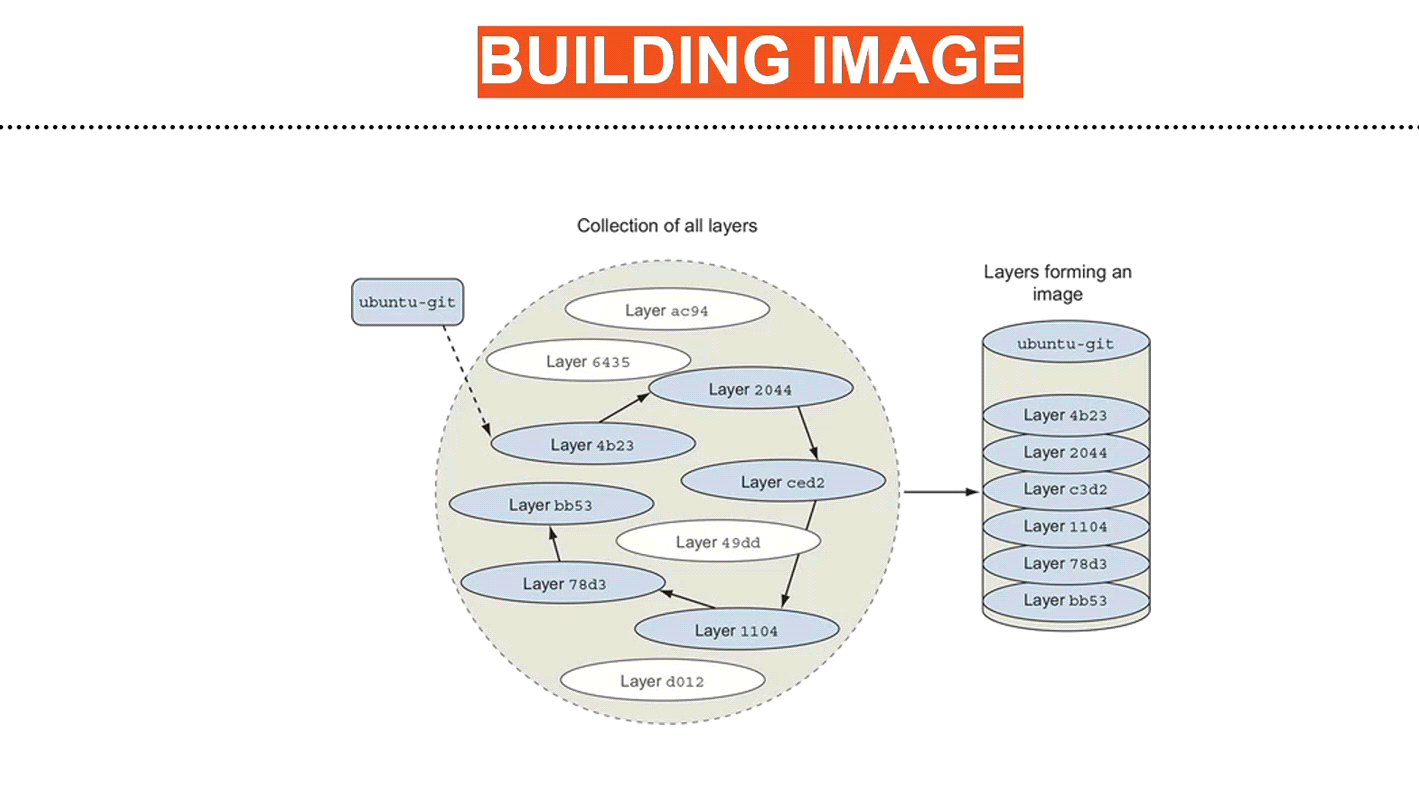
Acum pe baza la imaginea anterioara cu Java, mai cream o imagine pe baza ei si mai adaugam un web.xml

* Totusi, e interesant ca acum nu o sa avem 3 fisiere de index.html sau 2 fisiere de Java, ci doar unul de fiecare. Nu s-a facut cate o copie a elementelor din imaginea precedenta in cea noua, dar ele se folosesc direct din ea.
* Deci, Layer 1 depinde de layer 0, si Layer 2 de 1 si 0. Iata de ce, toate aceste fisierele nu se copie de mai multe ori, ci exista doar ca unitati singure
* Docker si Linux pur si simplu are grija sa preia toate aceste fisierele de care depind si sa genereze o multime cu toate
* Acum, sa zicem ca vrem sa stergem fisierul index.html in Layer 2.Anume din imagine. Nu vom putea sa il stergem din imaginea din Layer 0, ci putem doar crea o noua imagine pe baza la Layer 2 si sa specificam ca nu il vrem pe index.html, si acest fisier nu va fi sters din sistem, dar pur si simplu Layer 3 va avea informatia ca el trebuie sa fie ascuns, sa nu fie preluat din Layer 0 si ca si cum sa para ca e inexistent
* Deci, in concluzie, Docker aduna tot de prin imagine si imaginele dependente si creaza **File System la Container**
* **Un container nu are size egal cu cel al imaginii! El preia direct ce are nevoie din imagine, si daca modifica ceva, salveaza doar modificarile la nivel de Layer**

**Docker Container Lifecycle**

* build – container creat de la imagine
* commit – cream noua imagine de la cea de baza
* run – pornire in mai multi pasi
* pause – il punem pe pauza cu tot cu procesele sale
* stop – oprim container
* kill – oprim fortat
* rm – stergem container
* rmi – stergem imagine

**Building Images(Optional pentru un developer)**

* 
* Putem crea un file ce sa contina coduri pentru a genera o imagine cu container, adica contine comenzi ce ar trebui sa le scriem in consola

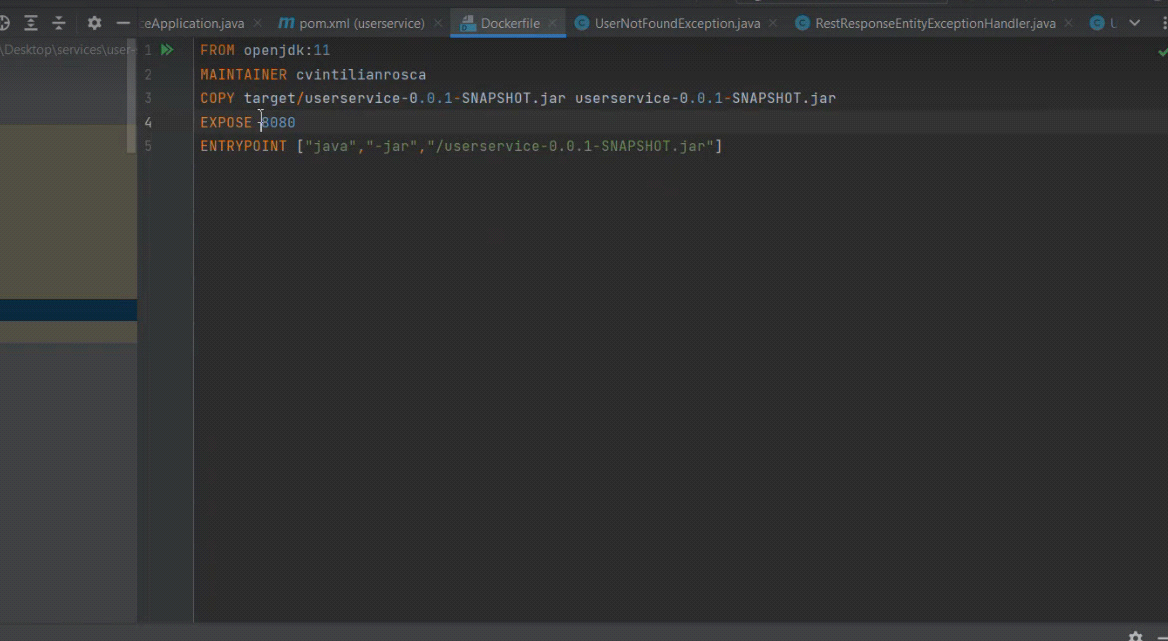
.................

**docker-compose.yml**

* Nu e prea comod sa tot pornim fiecare container
* In Intellij, folosim fisierul docker-compose.yml pentru a porni toate containerele necesare la pornirea aplicatiei

.......

**Docker File**



From – numele imaginii

MAINTAINER – creatorul fisierului

COPY – locatia intai o punem cu fisierul si apoi numele pe care il va avea fisierul in container

EXPOSE este portul

ENTRYPOINT – comanda care se executa la docker run

**Docker-compose**

* Daca docker-compose nu functioneaza,o instalam
* Docker compose ne permite sa lucram cu multiple containere
* El foloseste un fisier docker-compose.yaml
* Un fisier docker-compose.yaml are urmatoarea structura:version:

version:

services:

web:

build:

ports:

volumes:

database:

image:

environment:

version - versiunea de docker compose folosita

service - de aici se definesc configuratiile containerului

web - service name

build - locatia la docker file

ports - porturile aplicatiei

version: '3.9' # docker-compose

services:

web: # web services

build: . # builds containers in the root directory

ports:

- "8080:8080" # app to port 8080

# Mount volumes in home directory on the away directory

volumes:

- "/home:/away"

database:

image: "redis:alpine"

environment: #environment variables

- "USERNAME=yourUser"

- "PASSWORD=p@wen"

* Acum, navigam cu cd pana la folderul unde e fisieru yaml si rulam **docker-compose build** pentru a crea imaginea
* **docker-compose run** - porneste containerul pe baza imaginii create
* **docker-compose up** - combinatia la build si run
* **docker-compose ps** - vedem toate containerele asociate cu fisierul yaml
* **docker-compose down** - sterge imaginile asociate cu fisier si opreste si sterge containerele
* **docker-compose stop** - doar opreste containerele asociate la file