Containers with Docker

Key Takeaways

TECHWORLD WITH NANA

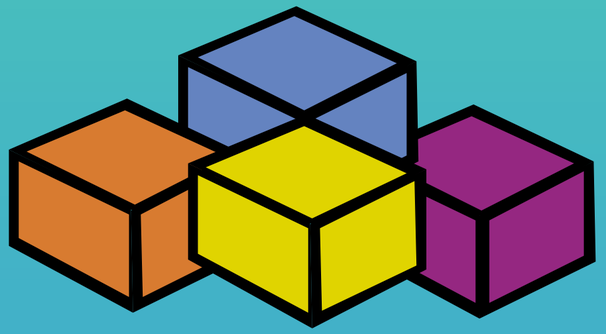
Introduction to Containers & Docker

TECHWORLD WITH NANA

What is Docker?

- Docker is an open source **containerization platform**
- Enables developers to **package applications into containers**
- Containers existed already before Docker
- Docker made containers popular

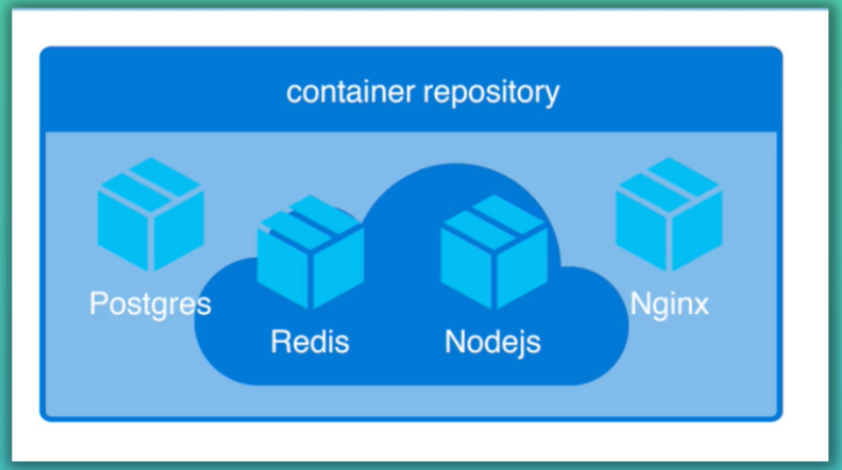
What is a container?



- A way to **package** application with **all** the **necessary dependencies** and **configuration**
- **Portable standardized artifact** for development, shipment and deployment
- Makes development and deployment **more efficient**

Where container artifacts are hosted

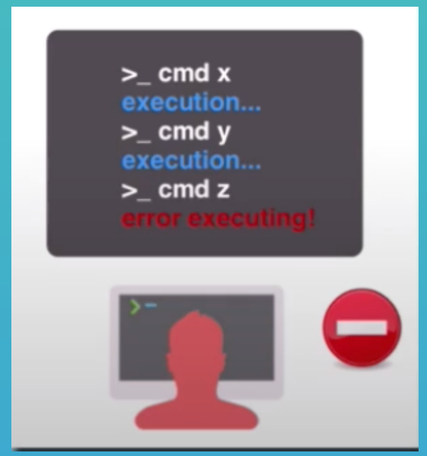
- Hosted in **container repositories**
- There are private and public repositories
- Public repository for Docker:

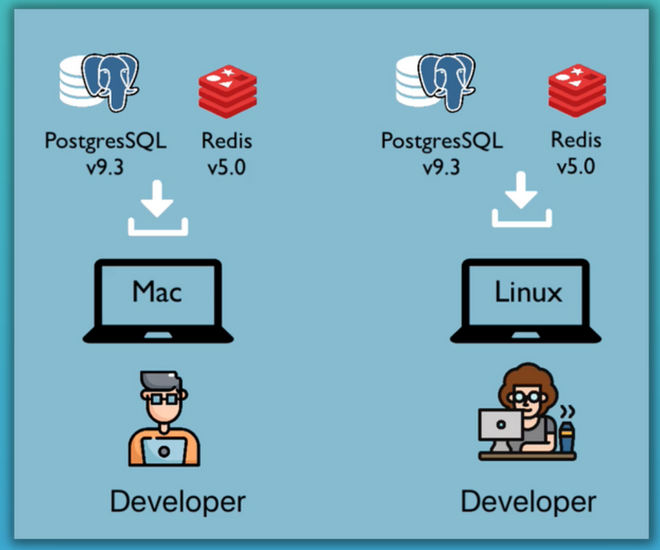


Application DEVELOPMENT before and after Docker

Before Containers

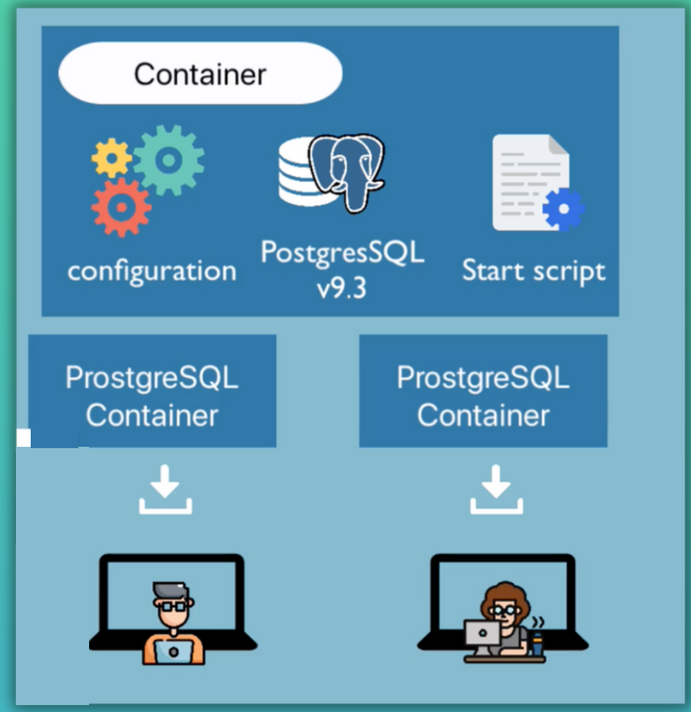
- **Installation process different** on each OS environment
- **Many steps** where something could go wrong



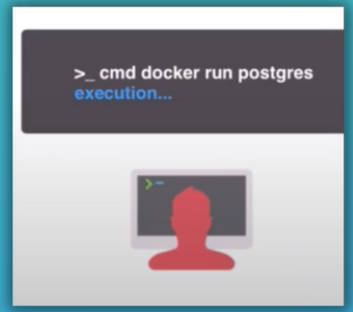


After Containers

- Own **isolated environment**
- Packaged with all needed configurations



- **1 command** to install the application
- Easily run same application with 2 different versions



TECHWORLD WITH NANA

Application DEPLOYMENT before and after Docker

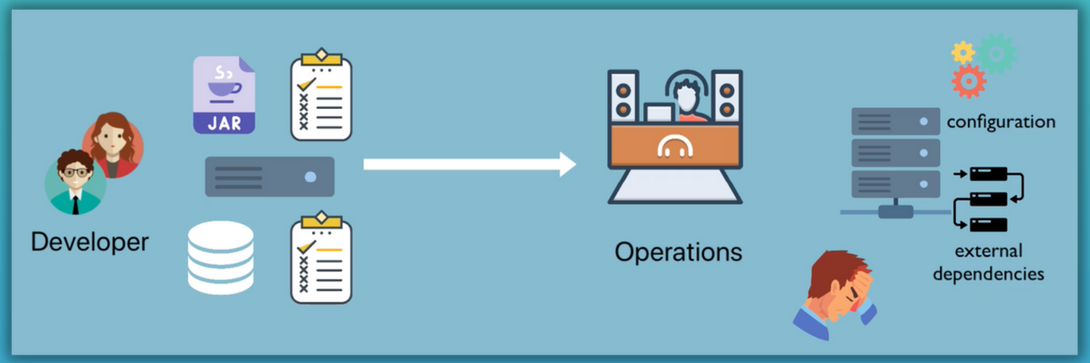
Before Containers

- Configuration on the server needed

❌ Dependency version conflicts

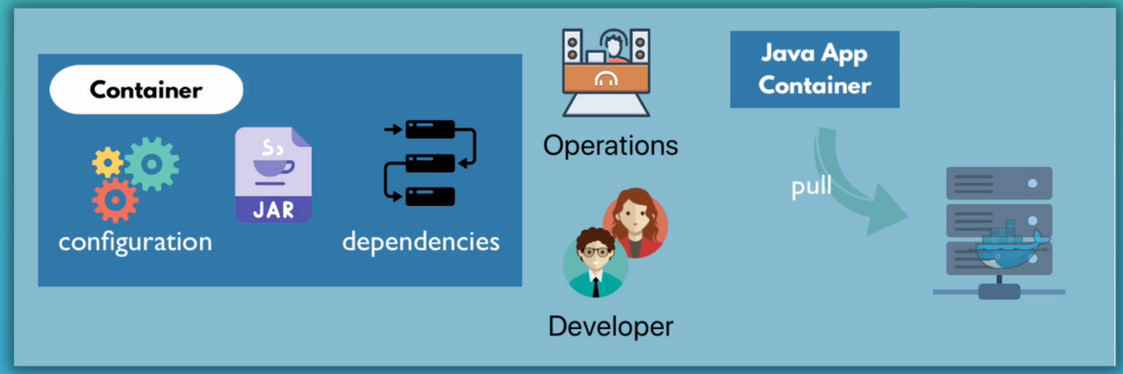
- Textual guide for deployment

❌ Misunderstanding



After Containers

- Devs & Ops work together to package the application in a container
- **No environment configuration needed** on server (except Container Runtime)



TECHWORLD WITH NANA

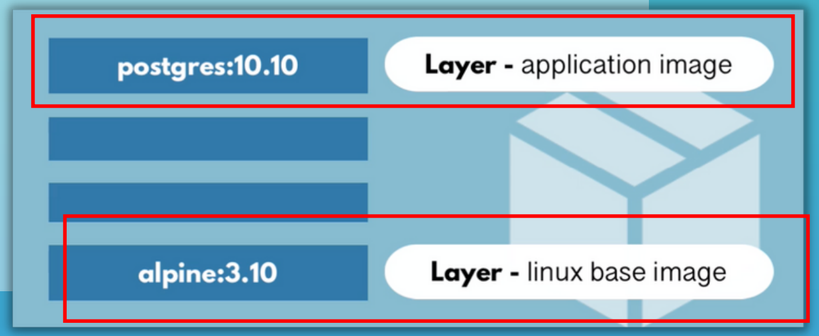
public repo- no login no authentication
docker run postgres:9.6 (first search container local. not found, then pull)
docker ps: all running containers
Operation system has two layers: OS kernel and application(ubuntu und Fedora)

Main command

Docker Image vs Docker Container

Docker **Image**

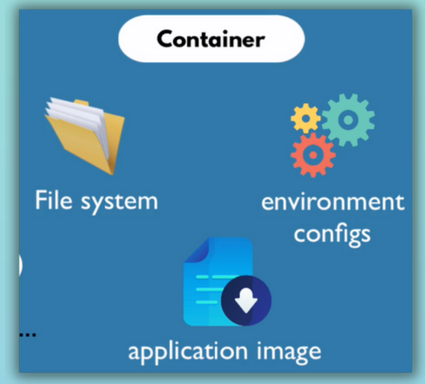
- The actual package (file)
- **Artifact** that can be moved around
- Not in "running" state
- Consists of several layers
- Mostly Linux Base Image, application image on top



Images become containers at runtime

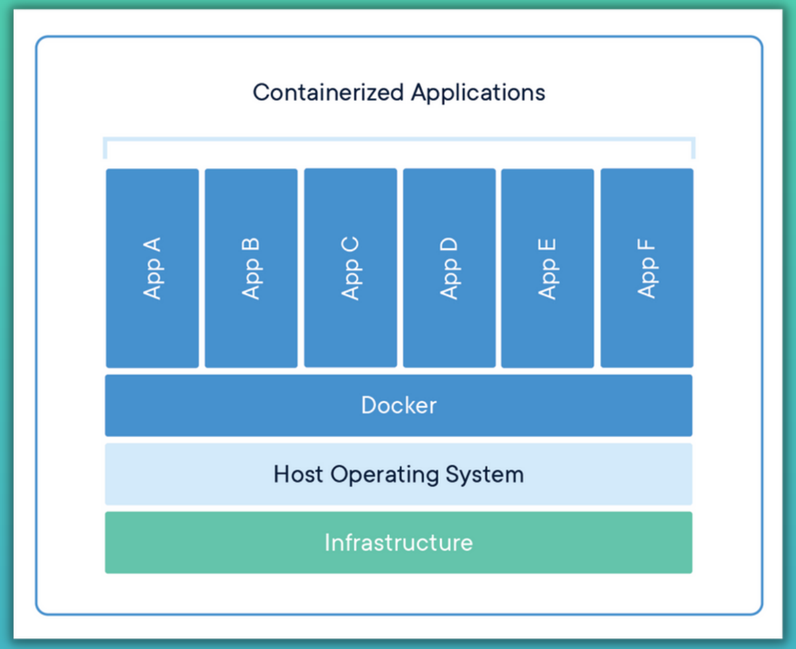
Docker **Container**

- Actually **starts the application**
- Is a **running environment** defined in the image



- Virtual file system
- Port binding: talk to application running inside of container

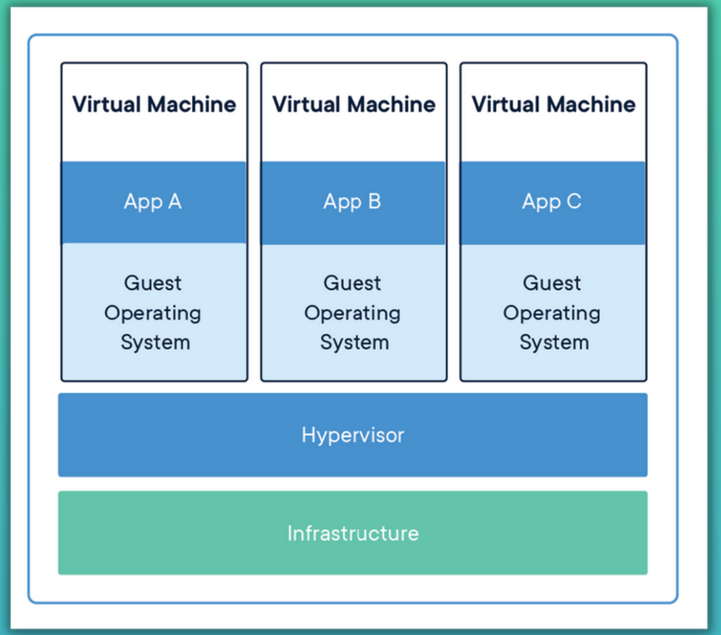
Containers vs Virtual Machines



Size: Docker Image much smaller

Speed: Docker containers start and run much faster

Compatibility: VM of any OS can run on any OS host



- Abstraction at the app layer
- Multiple containers share the OS kernel

- Abstraction of physical hardware
- Each VM includes a full copy of an OS

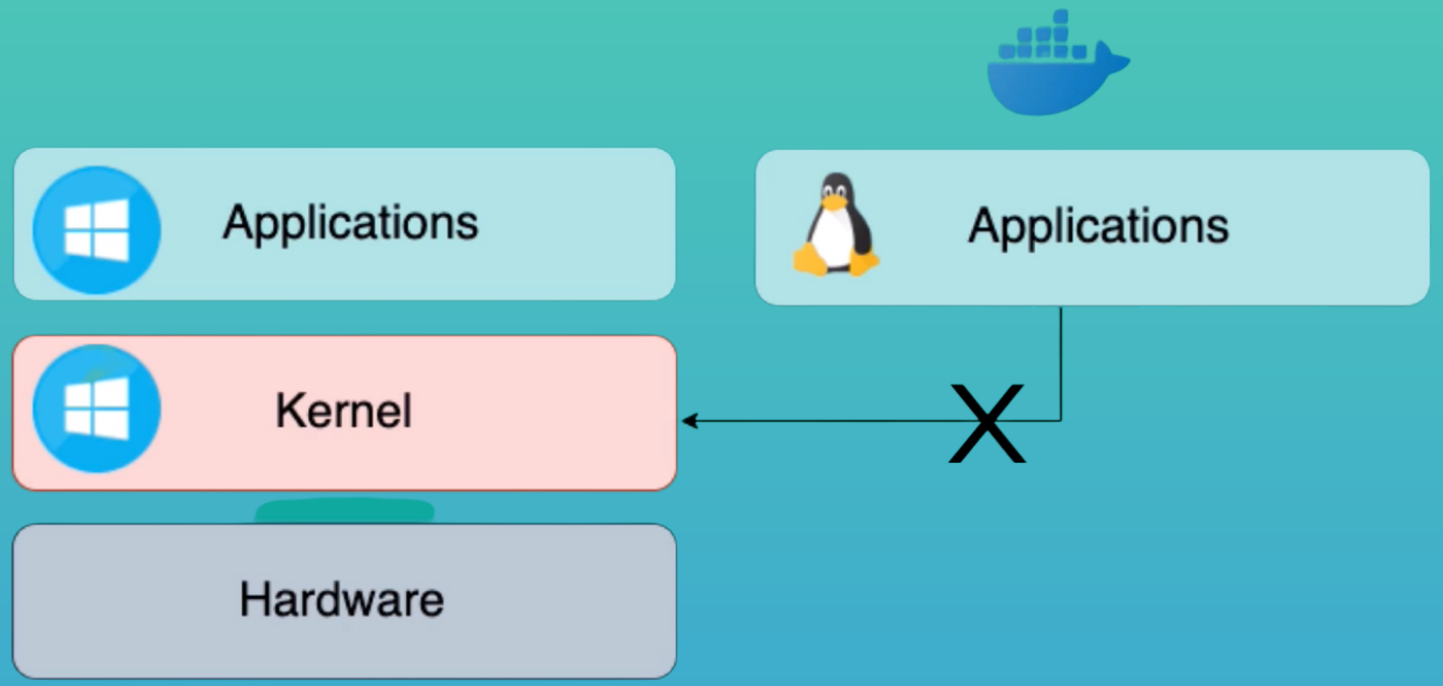
Both are **virtualization** tools

TECHWORLD WITH NANA

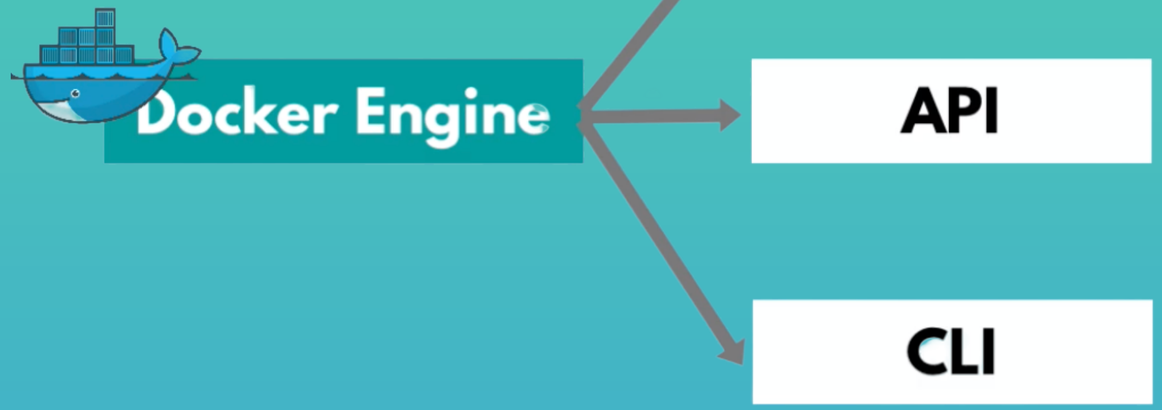
```
Container is a running environment for image
virtual file system / environment configs / application image
port binded: talk to application running inside  of container
```

You can't run Linux container on Windows host, but for that there is

Docker Desktop for Windows and Mac



Docker Architecture & its components - 1



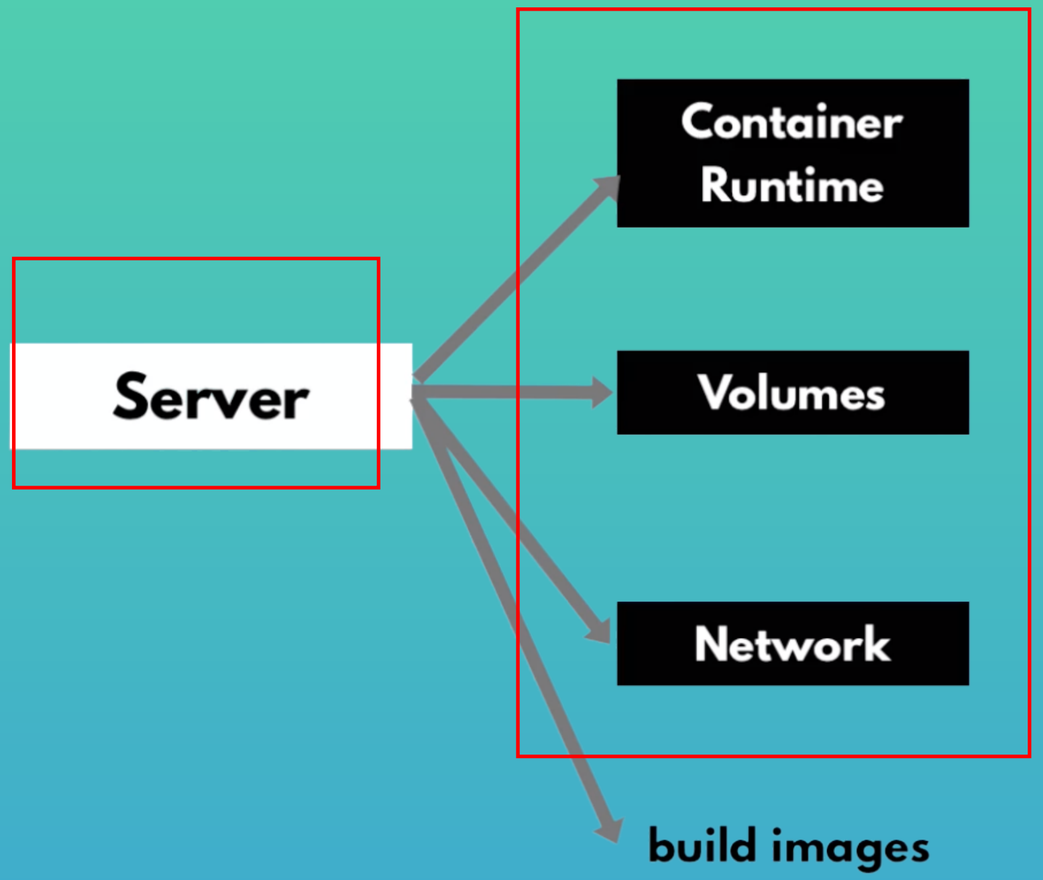
- Pulling images
- Managing images & containers

- Interacting with Docker Server

- Command Line Interface:

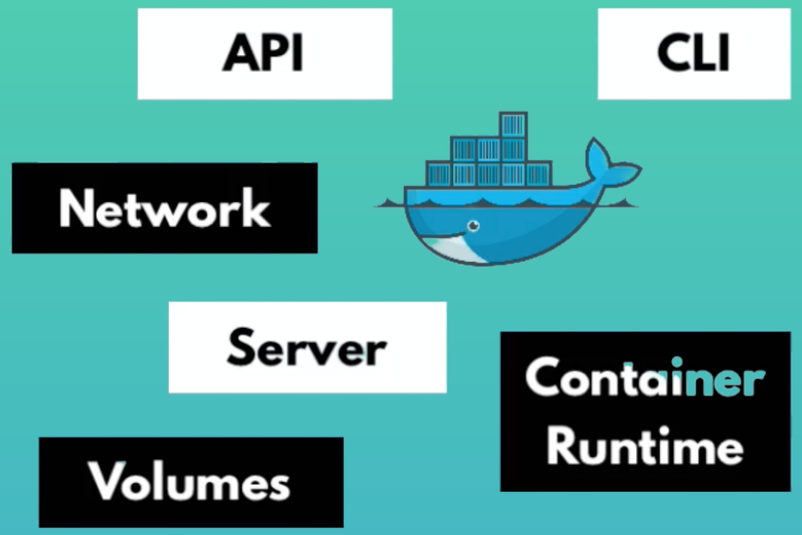
  Client to execute Docker commands

Docker Architecture & its components - 2



- Pulling images
- Managing container lifecycle
- Persisting data
- Configuring network for container communication

Docker Architecture & its components - 3



Docker has many functionalities in 1 application

Alternatives, if you need **only** a **container runtime**?

containerd

cri-o

Need to build an image?

buildah

Deep Dive into Docker

TECHWORLD WITH NANA

Docker Installation

Linux

native support

Mac + Windows

Docker Desktop

Get Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Link in Checklist

You can download and install Docker on multiple platforms. Refer to the following section and choose the best installation path for you.

Docker Desktop for Mac

A native application using the macOS sandbox security model which delivers all Docker tools to your Mac.

Docker Desktop for Windows

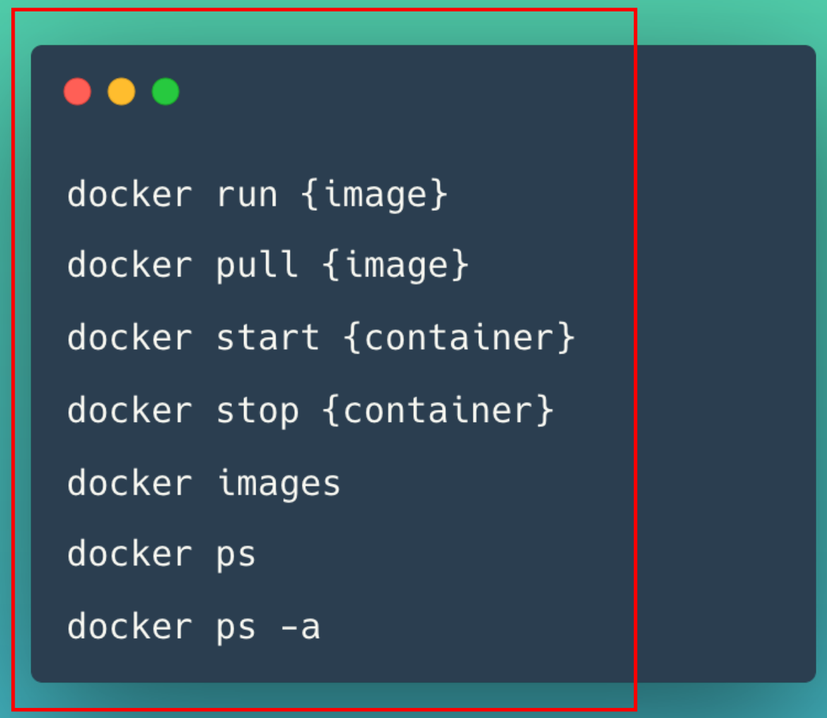
A native Windows application which delivers all Docker tools to your Windows computer.

Docker for Linux

Install Docker on a computer which already has a Linux distribution installed.

```
docker run -d redis
docker run = docker pull # docker start
```

Main Docker Commands



1. **docker run**: creates a container from an image
2. **docker pull**: pull images from the docker repository
3. **docker start**: starts one or more stopped container
4. **docker stop**: stops a running container + id
5. **docker images**: lists all the locally stored docker images
6. **docker ps:** lists the running containers
7. **docker ps -a:** show all the running and exited containers

Debug Commands

1. **docker logs**: fetch logs of a container `docker logs id/name`
2. **docker exec -it**: creates a new bash session in the container

`--name`

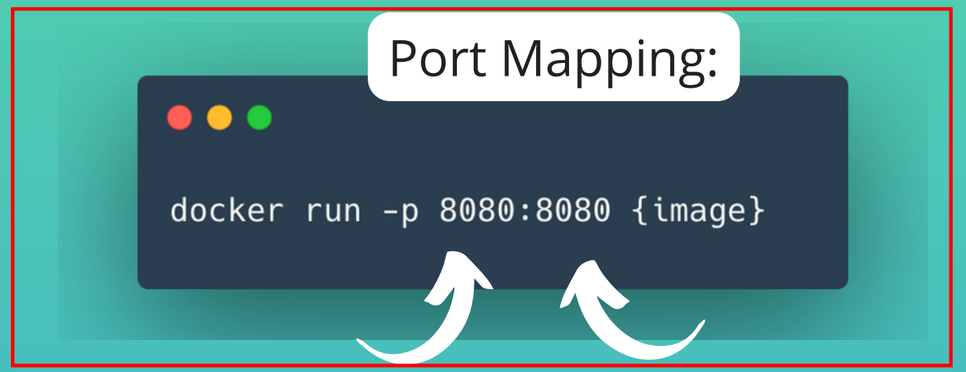
```
docker exec -it id /bin/bash
then you are the root use of a container
ls pwd printenv
```

Ports in Docker - 1

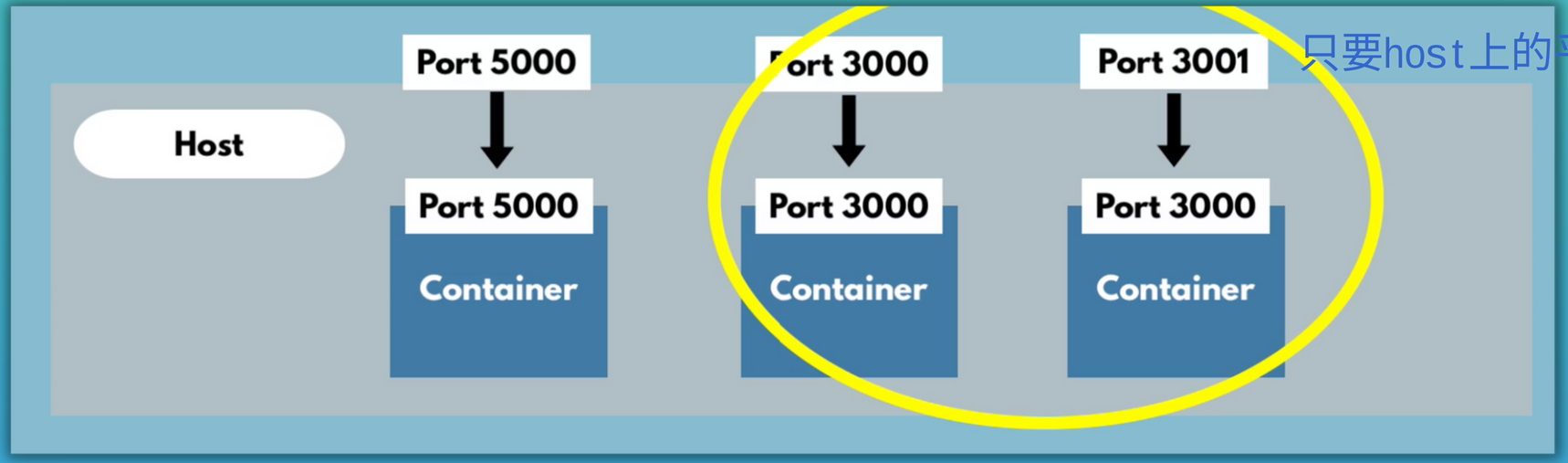
- **Multiple containers** can run on your host machine

Problem:

- But your laptop has only certain ports available
- **Conflict when same port** on host machine, so we need to map to a free port on host machine:



- host port
- container port

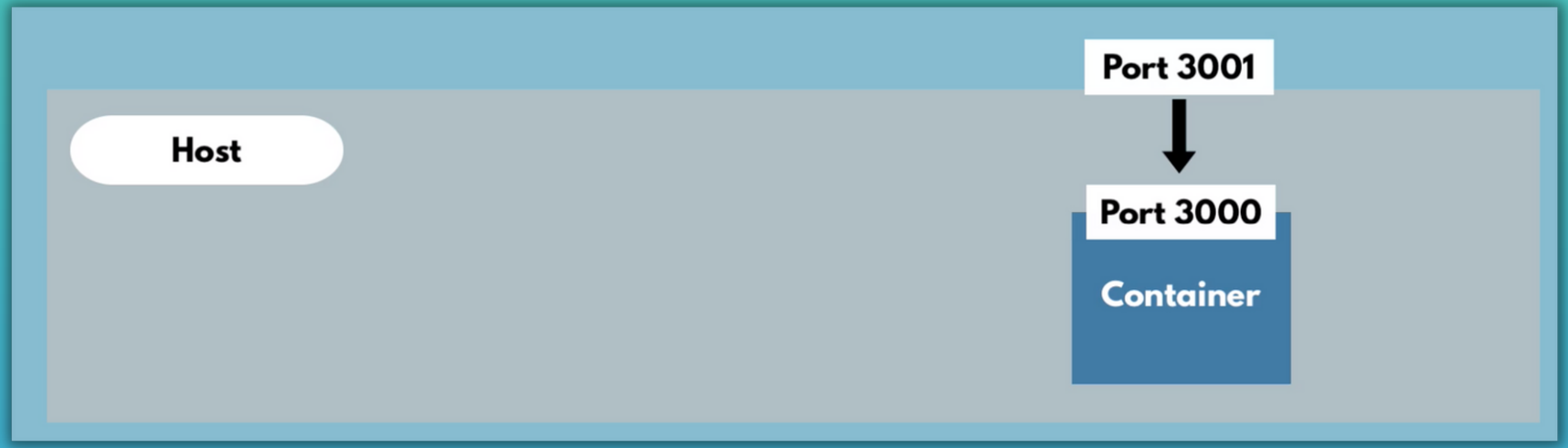


只要host上的平port不一样就好

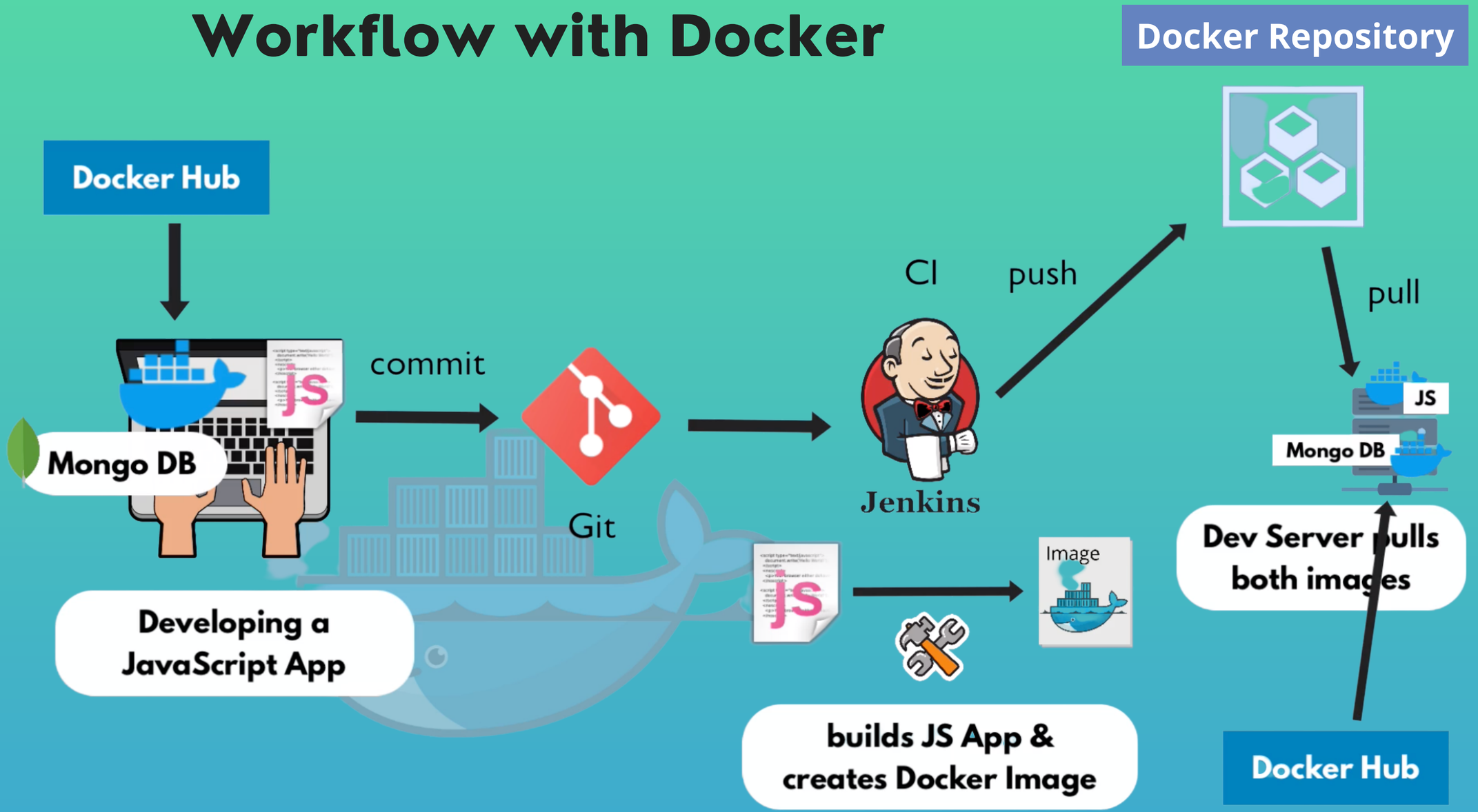
Ports in Docker - 2

- **Container Port** = Port used in container
- **Host Port** = Port on the host machine

some-app://localhost:3001



Workflow with Docker



Docker Compose - 1

Docker Compose is a tool for defining and **running multiple docker containers**

- YAML file to configure your application's services:
- You can **maintain and update configuration more easily** than with *docker run* command

docker run command

```
docker run -d \
--name mongo-express \
-p 8080:8080 \

-e ME_CONFIG_MONGODB_
   ADMINUSERNAME=admin \
-e ME_CONFIG_MONGODB_
   ADMINPASSWORD=password \
-e ME_CONFIG_MONGODB_
   SERVER=mongodb \

--net mongo-network \
mongo-express
```

mongo-docker-compose.yaml

```
version: '3'
services:
    mongodb:
        image: mongo
        ...
    mongo-express:
        image: mongo-express
        ports:
          - 8080:8080
      environment:
        - ME_CONFIG_MONGODB_A....
        ...
```

- Docker Compose automatically creates a **common docker network** for docker containers in it (*--net* option in docker run)

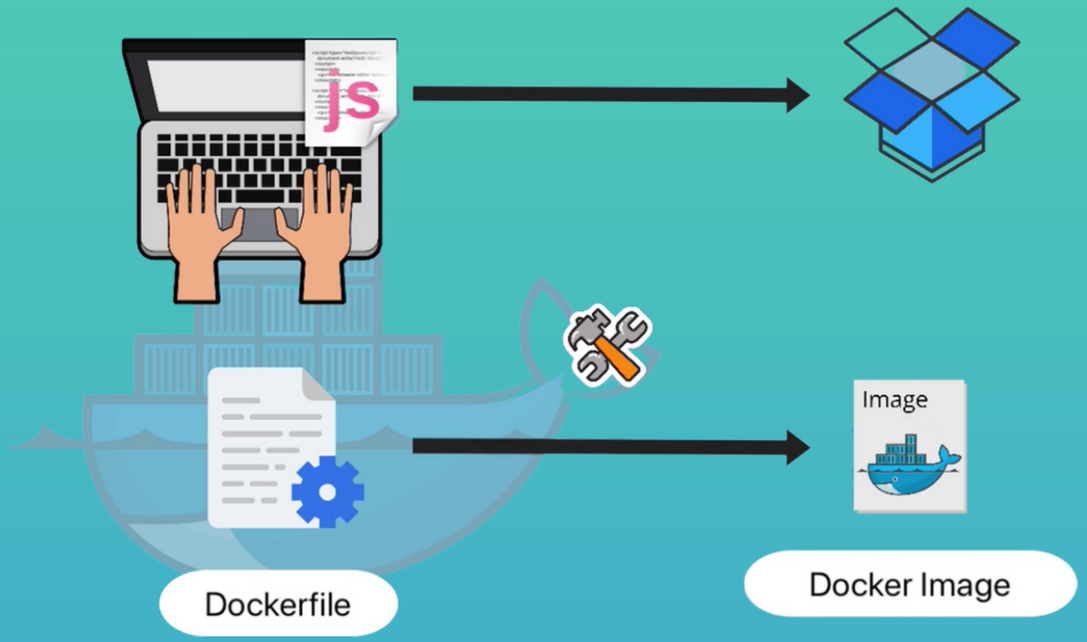
Docker Compose - 2

Example docker-compose.yaml file

```yaml
version: '3'
services:
  mongodb:
    image: mongo
    ports:
    - 27017:27017
    environment:
    - MONGO_INITDB_ROOT_USERNAME=admin
    - MONGO_INITDB_ROOT_PASSWORD=password
  mongo-express:
    image: mongo-express
    ports:
    - 8080:8081
    environment:
    - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
    - ME_CONFIG_MONGODB_ADMINPASSWORD=password
    - ME_CONFIG_MONGODB_SERVER=mongodb
```

Dockerfile - 1

A simple text file that consists of **instructions to build Docker images**



- Some common Dockerfile commands:

Image Environment Blueprint	DOCKERFILE
install node	**FROM** node
set MONGO_DB_USERNAME=admin set MONGO_DB_PWD=password	**ENV** MONGO_DB_USERNAME=admin \ MONGO_DB_PWD=password
create /home/app folder	**RUN** mkdir -p /home/app
copy current folder files to /home/app	**COPY** . /home/app
start the app with: "node server.js"	**CMD** ["node", "server.js"]

CMD = entrypoint command

You can have multiple RUN commands

blueprint for building images

TECHWORLD WITH NANA

Dockerfile - 2

- Each instruction in a Dockerfile results in an Image Layer:

- It's common to start with an existing base image in your application's Dockerfile:

- Command to build an image from a Dockerfile and a context
- The build's context is the set of files at a specified location

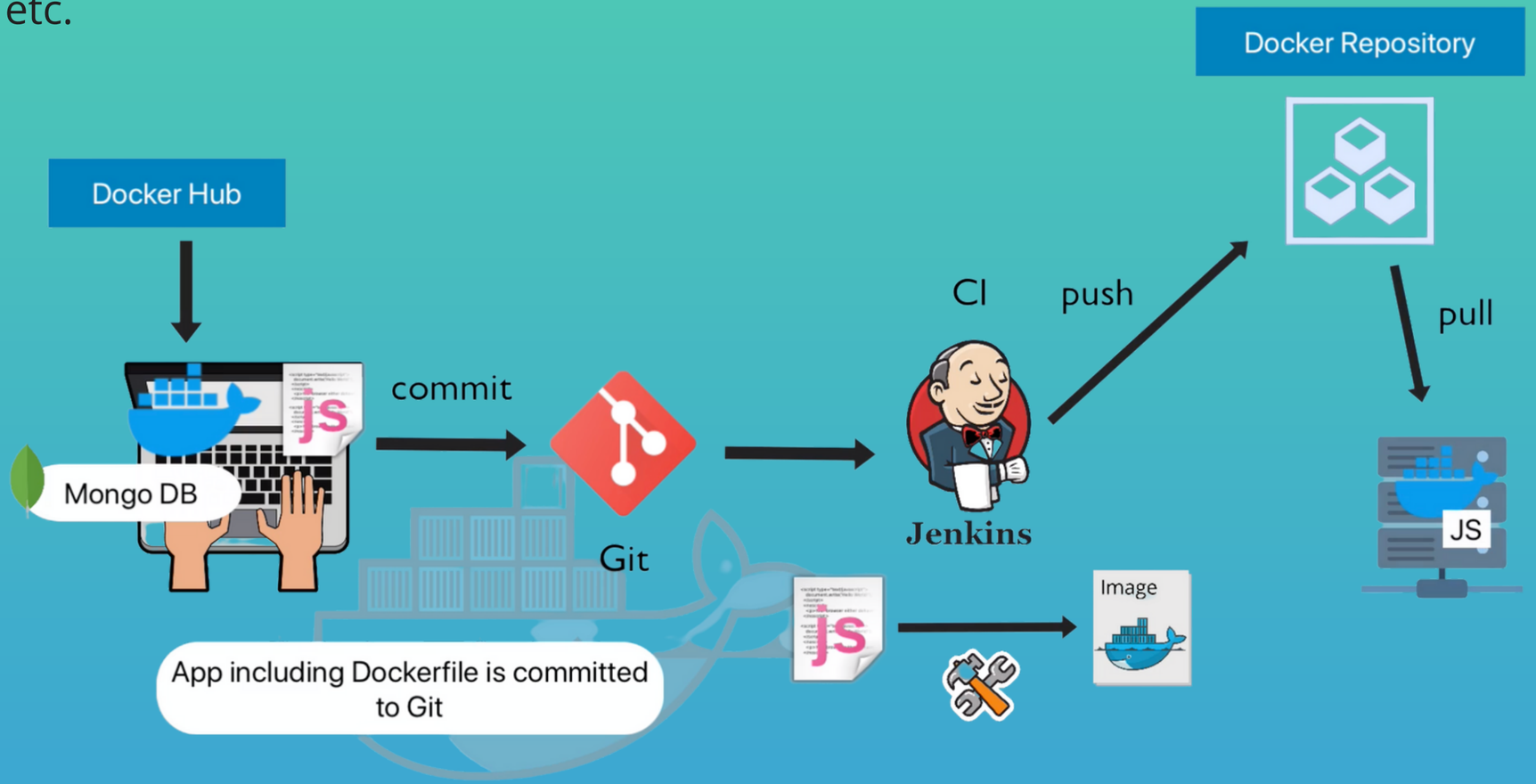
```
docker build -t my-app:1.0 .
```

Uses the current directory (.) as build context

TECHWORLD WITH NANA

Dockerfile - 3

- Dockerfile is **used in CI/CD** to build the docker image artifact, which will be pushed to Docker repo,
- Docker image can then be pull to multiple remote servers or pulled to locally for development and testing etc.



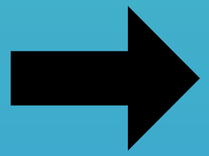
Private Docker Repository - 1

- When you work in a company, you will be working with a private docker registry
- **Public = DockerHub**, Example for **Private** = Amazon Elastic Container Registry "**AWS ECR**"

Difference to DockerHub

1. You need to **login,** so authenticate with the registry before fetching or pushing the image
2. **Tag** your image with the **registry address and name**,
3. Push the tagged image

```
docker login
- reponame
- username
- password

docker tag {repo-name}:{image-version}
docker push {tagged-image}
```



registryDomain/imageName:tag

Private Docker Repository - 2

In DockerHub

- **docker.io is default** repository

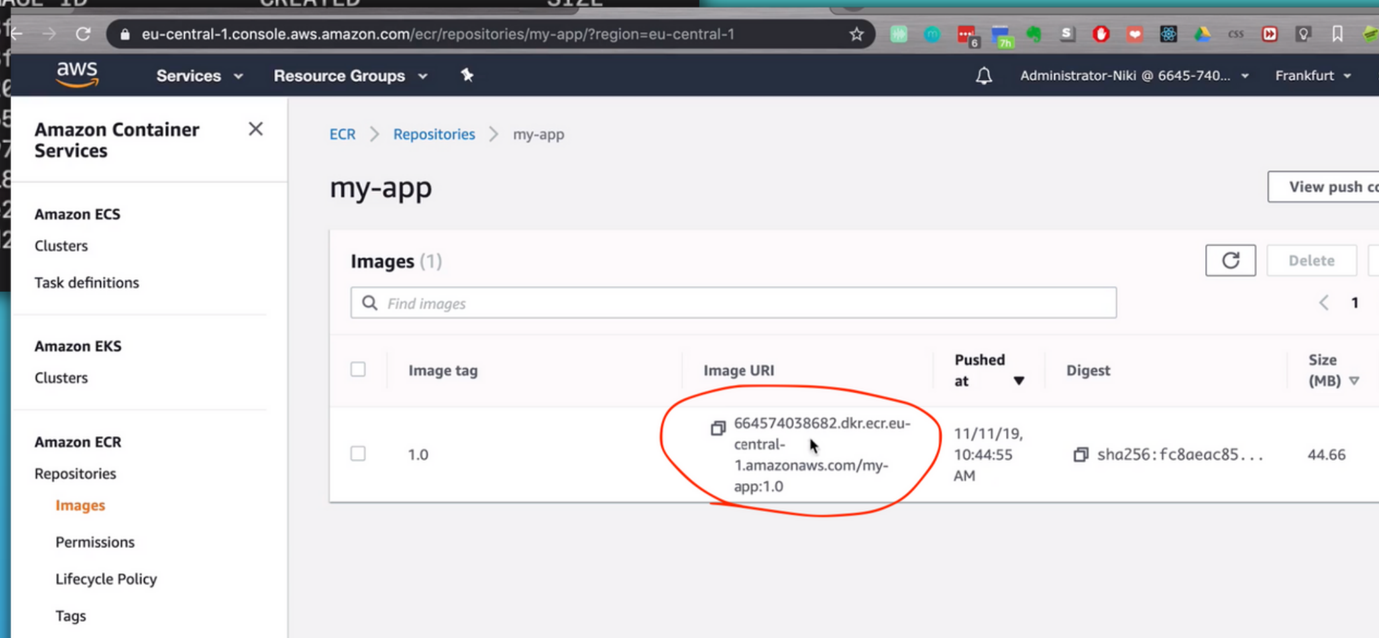
docker pull mongo:4.2 = *docker pull docker.io/library/mongo:4.2*

In AWS ECR or any other private registry

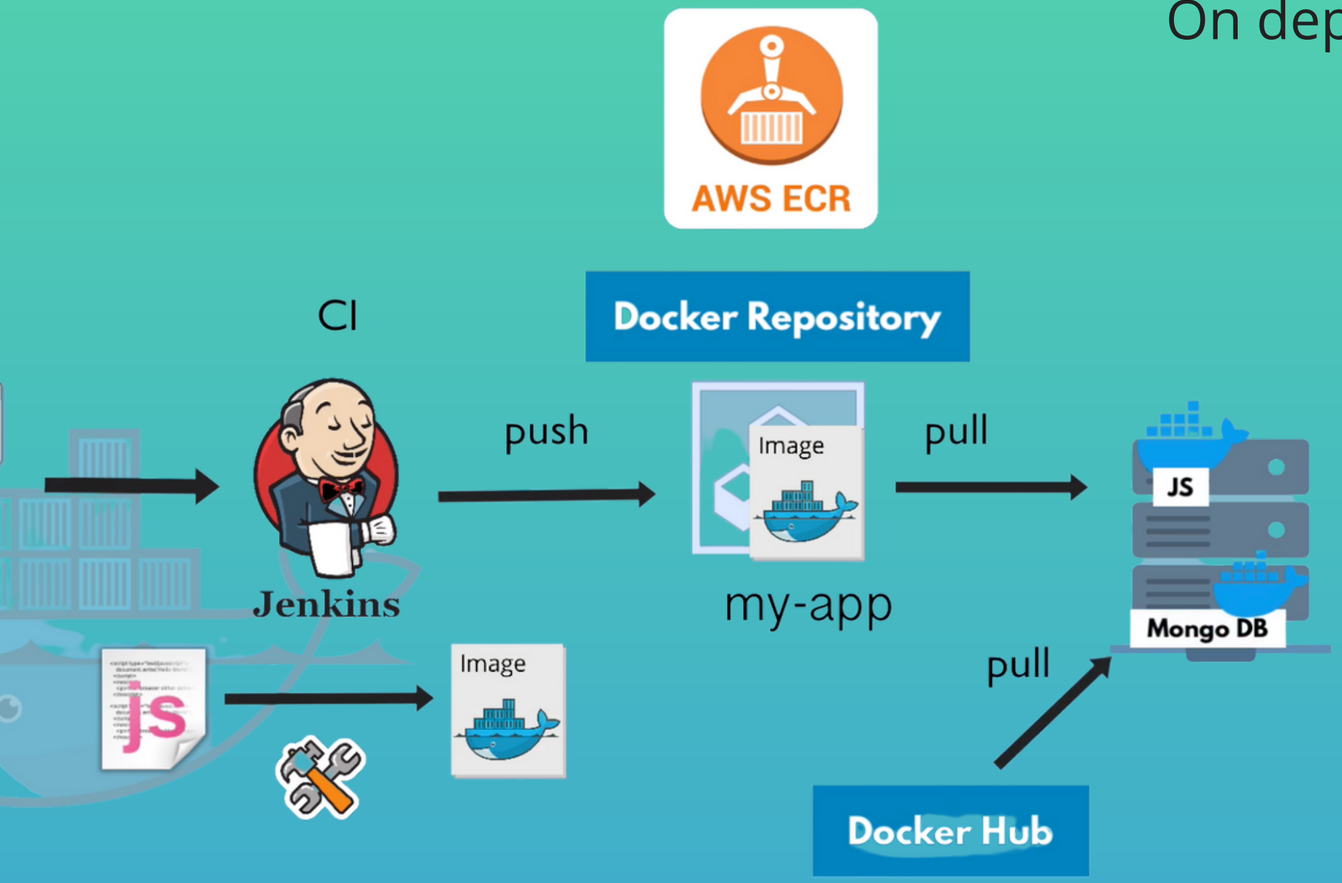
- Need to include the registry domain

docker pull 523450290.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.0

```
[~]$ docker tag my-app:1.0 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.0
[~]$ docker images
REPOSITORY                                                  TAG                 IMAGE ID            CREATED             SIZE
664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app      1.0                 83f
my-app                                                      1.0                 83f
node                                                        13-alpine           f20
mongo                                                       latest              965
mongo-express                                               latest              597
redis                                                       4.0                 e18
redis                                                       latest              de2
postgres                                                    9.6.5               bd2
[~]$ docker push 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.0
```



Deploy Docker Containers on a Remote Server



On deployment server, you can pull from both:

Private Images (Your JS application, company internal libraries)

↓

from **private docker repository** (AWS ECR)

Public Images (Mysql, MongoDB)

↓

from **public docker repository** (DockerHub)

TECHWORLD WITH NANA

Docker Volumes - 1

Volumes are the way to **persist data** generated by and used by Docker containers

Why Volume are needed for persistence?

- Data is stored on the virtual file system of the container. So when container is removed, the data is deleted as well.

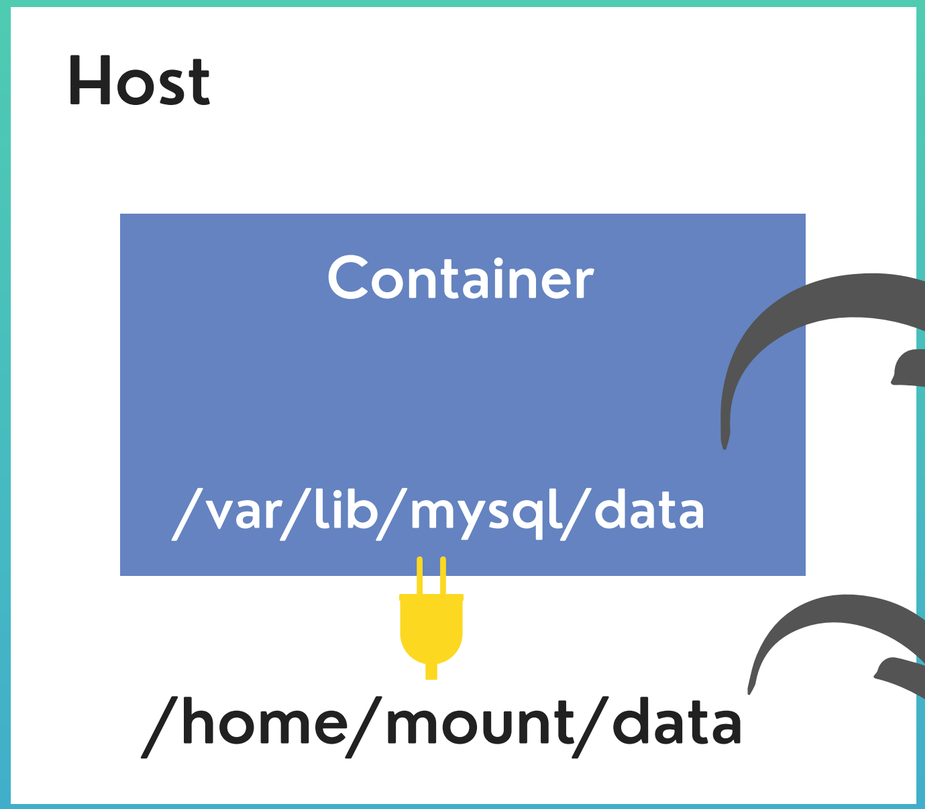
By default:
Data is gone when removing the container

Important for

- Databases
- Other stateful applications

Docker Volumes - 2

How Docker Volumes work?



- Folder in physical host file system is **mounted** into the virtual file system of Docker
- Data gets **automatically replicated**

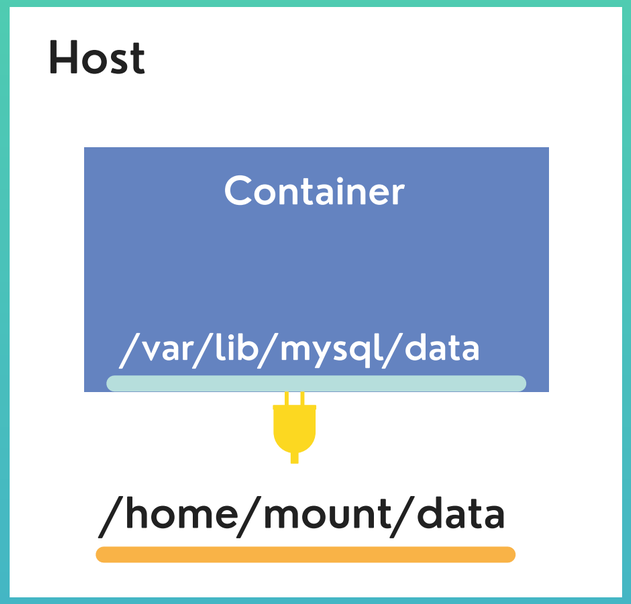
Virtual File System inside container

Host File System (physical)

Docker Volumes - 3

3 Volume Types

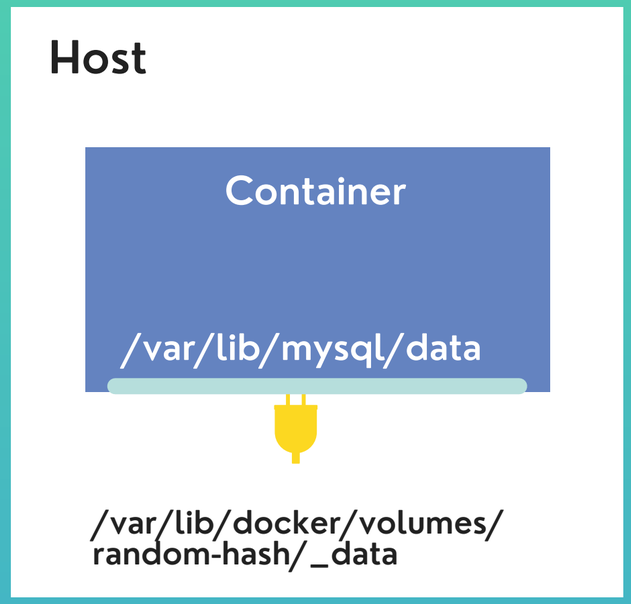
Host Volumes



- You decide **where on the host file system** the reference is made

```
docker run
-v /home/mount/data:/var/lib/mysql/data
```

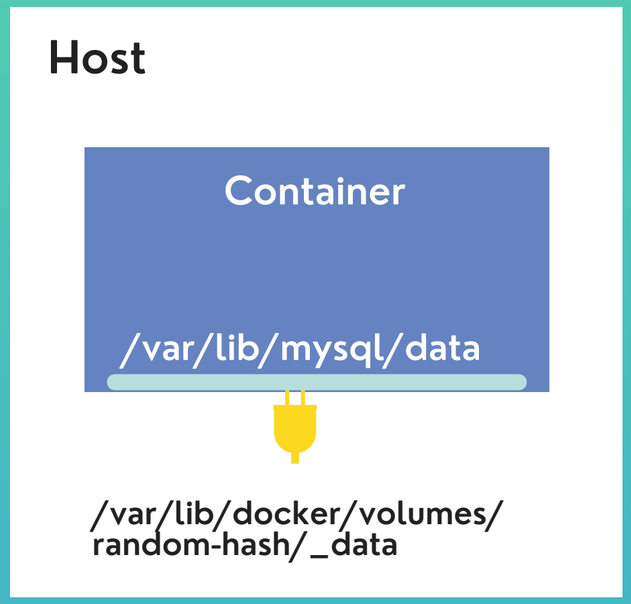
Anonymous Volumes



- For **each container a folder is generated** that gets mounted

```
docker run
-v /var/lib/mysql/data
```

Named Volumes



- You can **reference** the volume **by name**
- Should be used in production

```
docker run
-v name:/var/lib/mysql/data
```

Docker Volumes - 4

Docker Volumes in docker-compose file:

mongo-docker-compose.yaml

Docker Best Practices - 1

- **Security**: Only use official/trusted Docker images as base image to avoid malware
- **Security**: Use specific image versions
- **Size & Security**: Use minimal base images (e.g. prefer alpine-based images over full-fledged system OS images)
- **Size**: Optimize caching image layers
- **Size**: Use .dockerignore to exclude files and folders
- **Cleaner Dockerfile**: Make use of "Multi-Stage Builds"

Improve Security

Optimize Image Size

Write cleaner and more maintainable Dockerfiles

Docker Best Practices - 2

- **Security**: Use least privileged user (create a dedicated user and group with minimal permissions to run the application)
- **Security**: Scan your images for vulnerabilities
- **Security**: Don’t leak sensitive information to Docker images

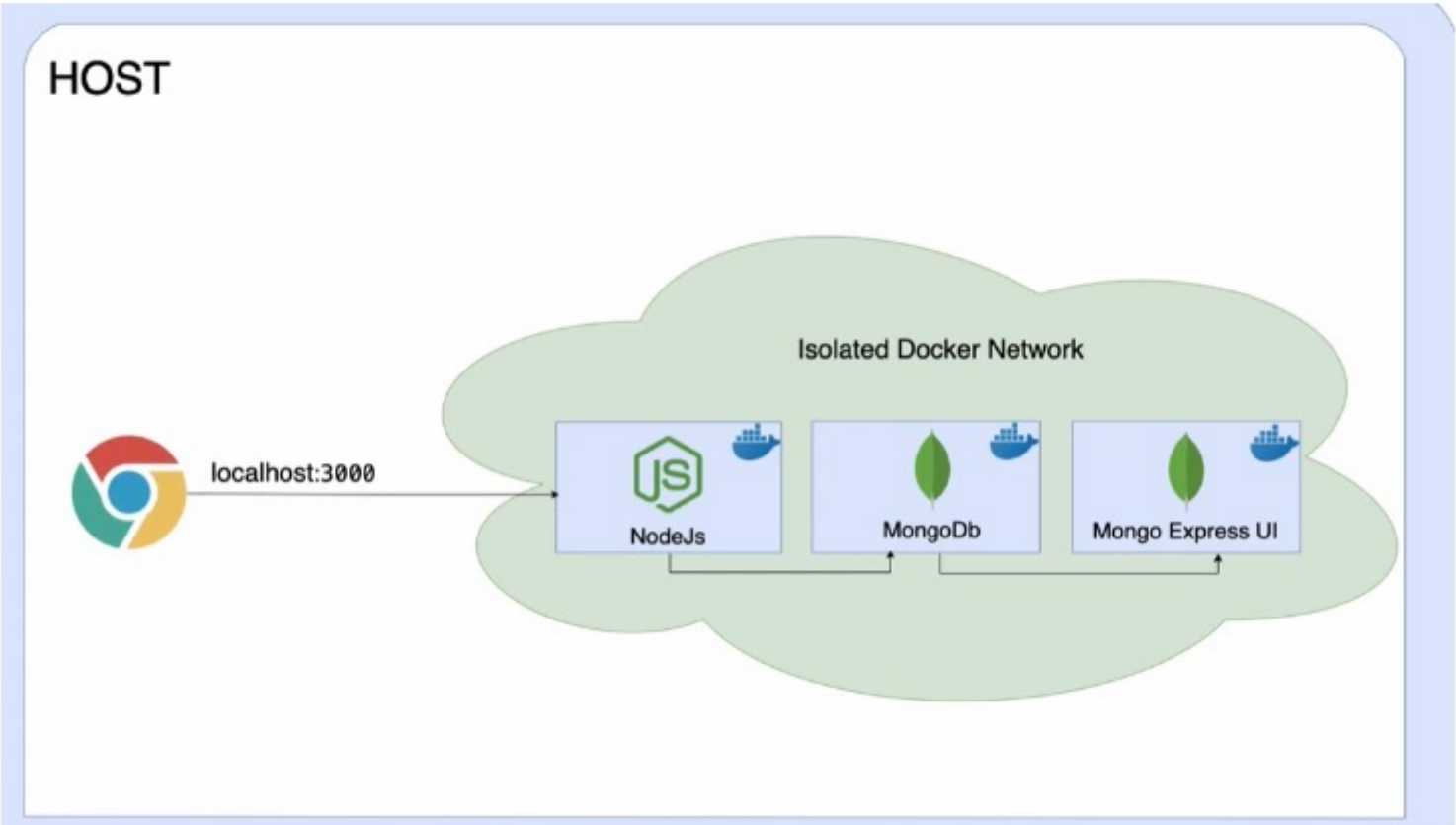
Improve Security

Optimize Image Size

Write cleaner and more maintainable Dockerfiles

Demo Project
Backend: nodjs with MongoDB

1. go to docker hub find mongoDB image and MongoExpress.
2. pull the mongoDB image : **docker pull mongo**
3. pull the mongoDB express: **docker pull mongo-express**
4. run both containers: **docker images (check what images you have)**



1. **Docker network ls** (list docker networks)
2. create own networks for mongoDB and mongoExpress: **docker network create mongo-network**
3. check if it created: **docker network ls**
4. Provide network condition wenn we run the container:

```
docker run -p 27017:27017 -d  \
-e MONGO_INITDB_ROOT_USERNAME=admin\
-e  MONGO_INITDB_ROOT_PASSWORD=password \
--name mongoDB\
 --net mongo-network \
mongo
```

1. start run mongo express:

```
docker run -d \
-p 8081:8081 \
-e ME_CONFIG_MONGODB_ADMINUSERNAME=admin \
-e ME_CONFIG_MONGODB_ADMINPASSWORD=password \
--net mongo-network \
--name mongo-express \
-e ME_CONFIG_MONGODB_SERVER=mongodb \
mongo-express
```

1. to the localhost:8081 where mongo express is and create a new database. Where your can then from nodejs to coneect to the databse.
2. for nodejs
3.

```
var express = require('express');
var path = require('path');
var fs = require('fs');
var MongoClient = require('mongodb').MongoClient;
var bodyParser = require('body-parser');
var app = express();
```

```
app.get('/get-profile', function (req, res) {
    var response = res;

    MongoClient.connect('mongodb://admin:password@localhost:27017', function (err, client) {
        if (err) throw err;

        var db = client.db('user-account');
        var query = { userid: 1 };
        db.collection('users').findOne(query, function (err, result) {
            if (err) throw err;
            client.close();
            response.send(result);
        });
    });
});

app.post('/update-profile', function (req, res) {
```

docker logs
docker logs idCOntainer tail (to check the logs)

summary: create docker network then run two docker containers (mongoDB and mongoExpress)
Then next step using docker compose to map the commands in a file

docker run command

```
docker run -d \

--name mongodb \

-p 27017:27017 \

-e MONGO-INITDB_ROOT_USERNAME
=admin \

-e MONGO-INITDB_ROOT_PASSWORD
=password \

--net mongo-network \

mongo
```

mongo-docker-compose.yaml

```
version: '3'
services:
    mongodb:
        image: mongo
        ports:
          - 27017:27017
        environment:
          - MONGO..._USERNAME=admin
```

docker run command

```
docker run -d \
--name mongo-express \
-p 8080:8080 \

-e ME_CONFIG_MONGODB_
   ADMINUSERNAME=admin \
-e ME_CONFIG_MONGODB_
   ADMINPASSWORD=password \
-e ME_CONFIG_MONGODB_
   SERVER=mongodb \

--net mongo-network \
mongo-express
```

mongo-docker-compose.yaml

```
version: '3'
services:
    mongodb:
        image: mongo
        ...
    mongo-express:
        image: mongo-express
        ports:
          - 8080:8080
        environment:
          - ME_CONFIG_MONGODB_A....
          ...
```

Docker comose takes care of creating a common network -> we don't need to create a network

```yaml
version: '3'
services:
  mongodb:
    image: mongo
    ports:
    - 27017:27017
    environment:
    - MONGO_INITDB_ROOT_USERNAME=admin
    - MONGO_INITDB_ROOT_PASSWORD=password
  mongo-express:
    image: mongo-express
    ports:
    - 8080:8081
    environment:
    - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
    - ME_CONFIG_MONGODB_ADMINPASSWORD=password
    - ME_CONFIG_MONGODB_SERVER=mongodb
```

how to start docker containers using docker compose?

1. using docker-compose commands
    a. **docker-compose -f mongo.yaml up** (f for file, up for starting all containers
        i. at the beginning the network is created
        ii. create two container with indextat the end
        iii. logs are mixed
    b. when you restart your container, everything is gone. Data is gone.
        i. Docker volumes for Data Persistence
    c. start the application
    d. stop: docker-compose -f mong.yaml down
        i. stop container
        ii. remove container
        iii. remove netwokr

code-test-build-deploy
from local to commit to git, then CI.
Jenkins: BUilds JS app and create docker image ~~and push to docker repository~~

Build a docker image:
copy artifact to **docker file is blueprint for building images**.

RUN executes inside of container
COPY executes on the HOST machine

Image Environment Blueprint	DOCKERFILE
install node	**FROM** node
set MONGO_DB_USERNAME=admin set MONGO_DB_PWD=password	**ENV** MONGO_DB_USERNAME=admin \ MONGO_DB_PWD=password
create /home/app folder	**RUN** mkdir -p /home/app
copy current folder files to /home/app	**COPY** . /home/app
start the app with: "node server.js"	**CMD** ["node", "server.js"]

CMD = entrypoint command

You can have multiple RUN commands

blueprint for building images

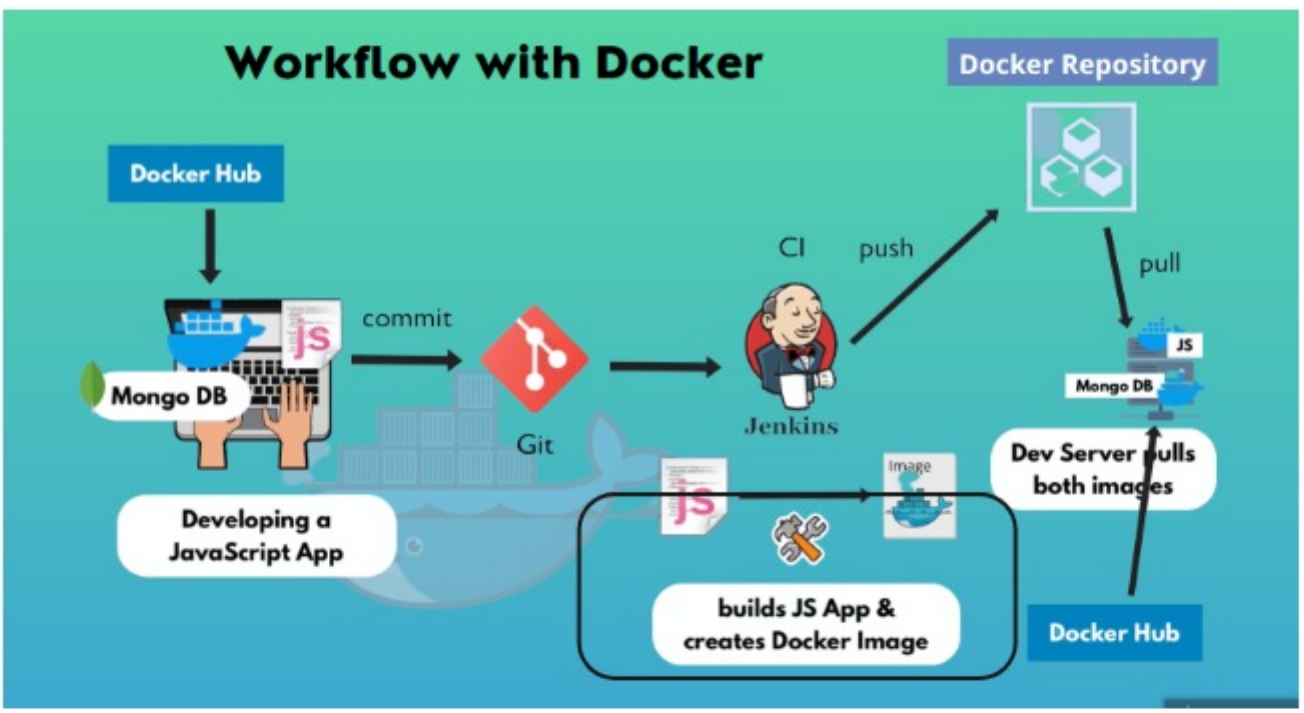
CMD just one , RUN can multiples. CMD is entrypoint.

```
FROM node:13-alpine

ENV MONGO_DB_USERNAME=admin \
    MONGO_DB_PWD=password

RUN mkdir -p /home/app

COPY . /home/app

CMD ["node", "server.js"]
```

"/home/app/server.js"

docker image from docker file

1. docker build -t my-app:1.0 .
2. docker images (check if the image created)



1. change focker file-> must rebuid docker image
2. docker stop +idContainer
3. if docker container stopped. must first cancel container then delete the image-
4. delete container: **docker ps -a |grep my-app** you ge the id of the container and use **docker rm +idContainer**.
5. then delete image: **docker rmi +idImage**
6. then build image again
7. if /bin/bash not work try /bin/sh

Overview

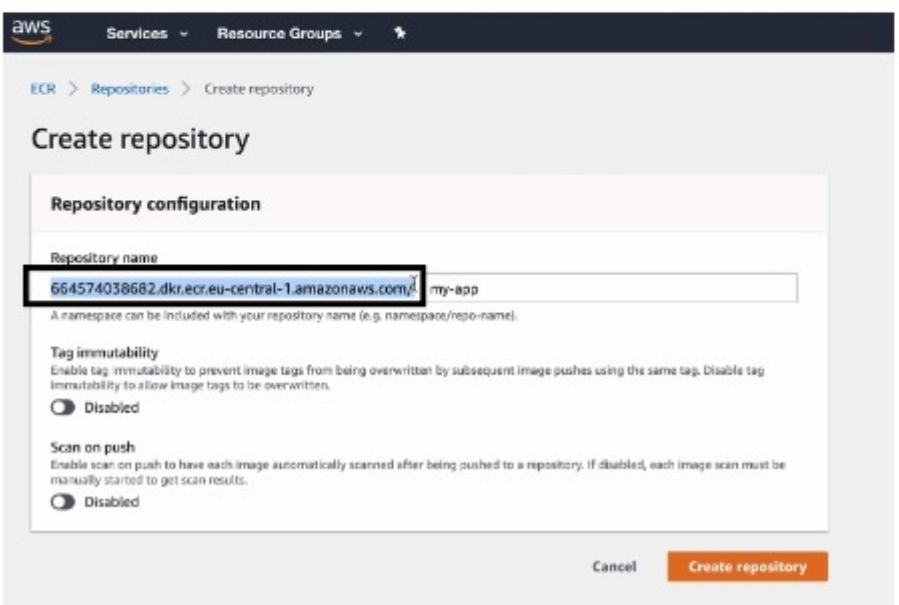
Docker Registry

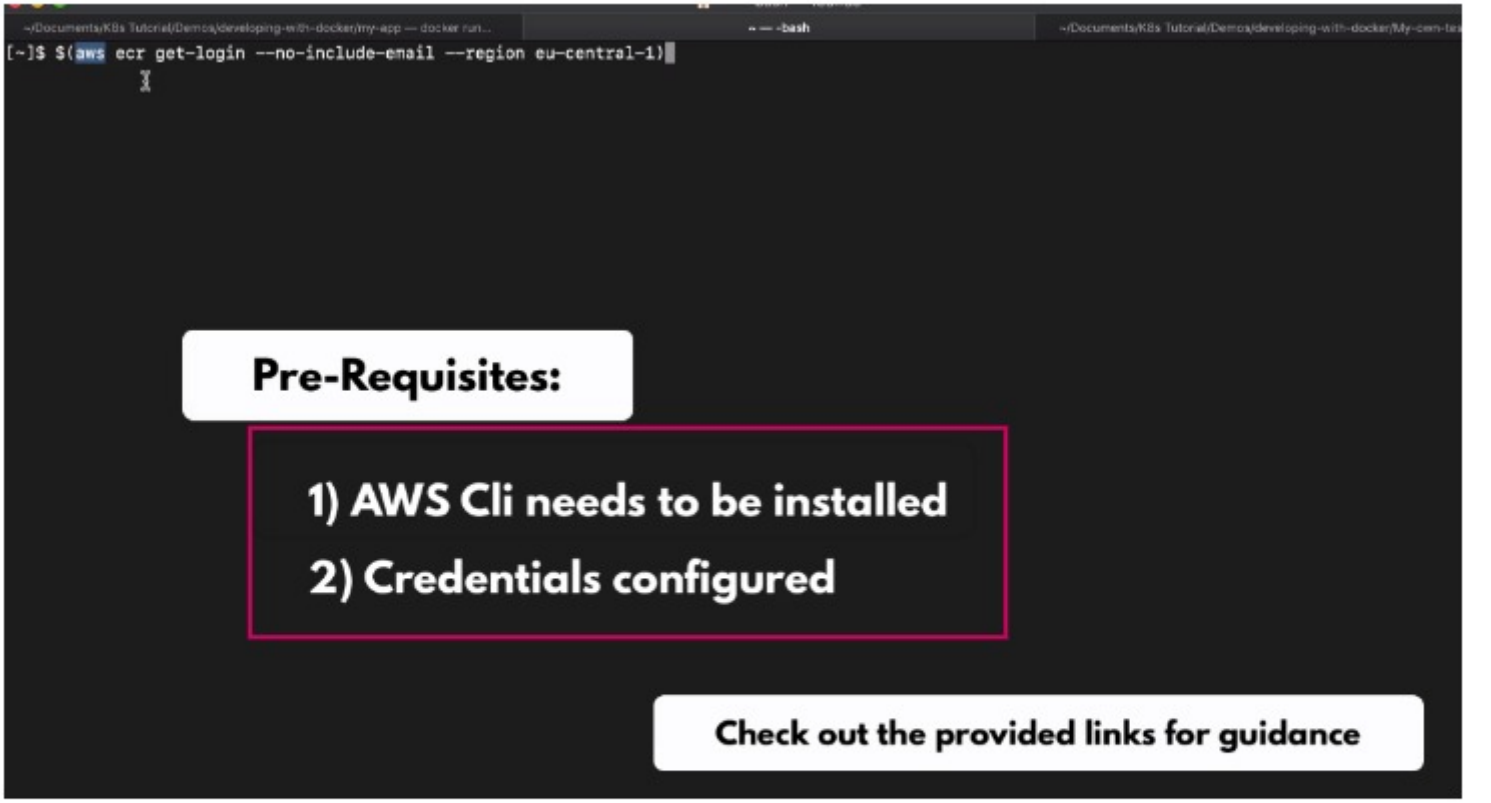
- [x] Docker private repository
- [x] Registry options
- [x] build & tag an image
- [x] docker login
- [x] docker push

Amazon ECR

create priavate registroy
-aws -> ECR (elastic container registry
get started-> create a registry -> give repository name -> create a repository
AMS registory: **Repository per Image. You save different tags of the same image**.
Push the image locally (docker images) to the repository
got to view push commandds.
Push:

1. You always have to **login to private repo**.
2.





Push commands for my-app

Use the AWS CLI:

```
$(aws ecr get-login --no-include-email --region eu-central-1)
```

Note: If you receive an "Unknown options: --no-include-email" error when using the AWS CLI, ensure that you have the latest version installed. **Learn more**

2. Build your Docker image using the following command. For information on building a Docker file from scratch see the instructions here. You can skip this step if your image is already built:

```
docker build -t my-app .
```

tag is actually rename

3. After the build completes, tag your image so you can push the image to this repository:

```
docker tag my-app:latest 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:latest
```

4. Run the following command to push this image to your newly created AWS repository:

```
docker push 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:latest
```

Close

Image Naming in Docker registries

registryDomain/imageName:tag

- In **DockerHub**:
  - docker pull mongo:4.2
  - docker pull docker.io/library/mongo:4.2
- In **AWS ECR**:
  - docker pull 520697001743.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.0

Deploy the application in the server -> Using **docker-compose** -> This docker-compose file would be used on the server to deploy all the allocations/services

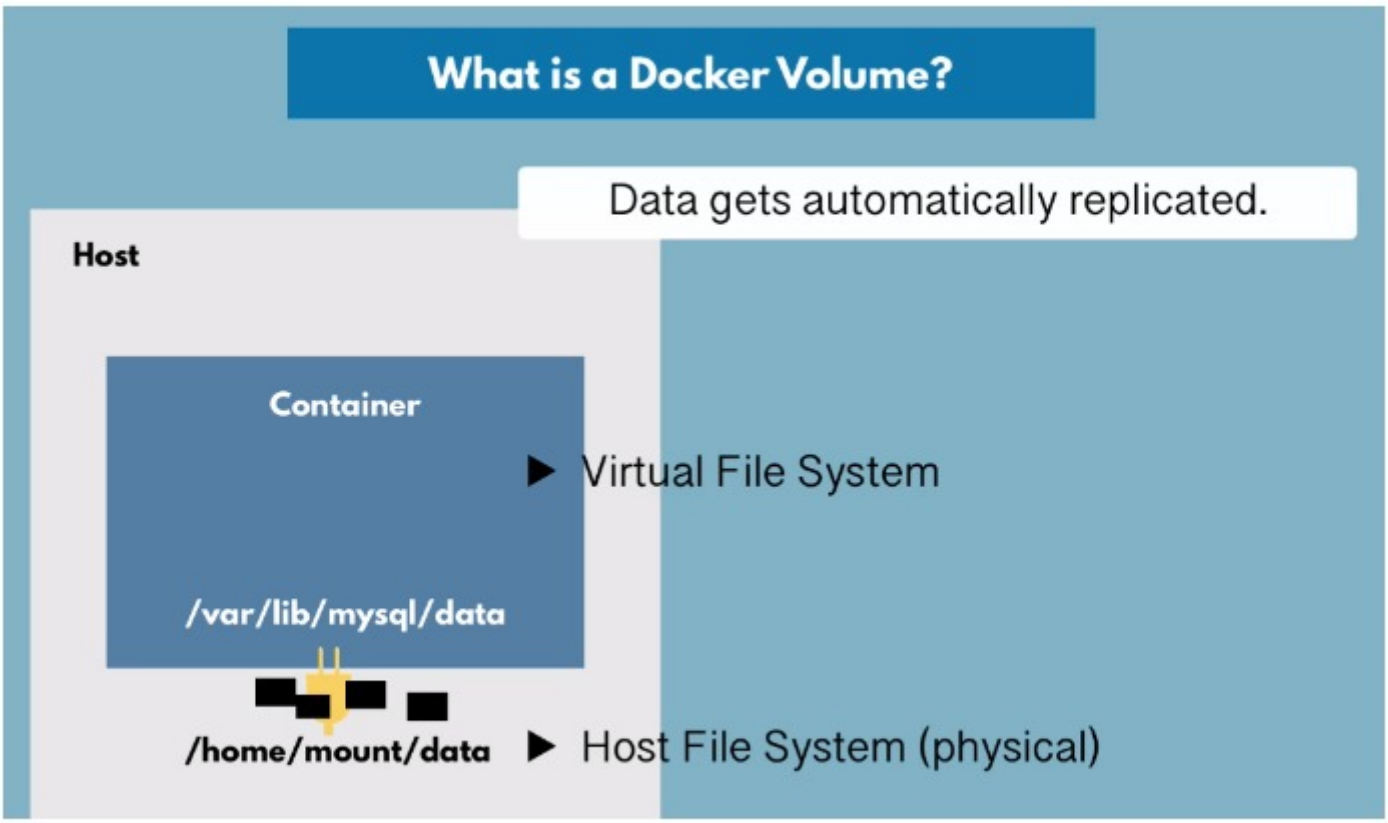
```
my-app:
  image: 664574038682.dkr.ecr.eu-central-1.amazonaws.com/my-app:1.0
```

1. add a new container in the list of docker-compose: get your image from aws
    a. my-app: image : repoNameInAWSECR
2. The server needs to login to pull from PRIVATE repo in AWS ECR
3. In the deploy server, create a mongo.yaml file as for docker compose. -> vim (copy all from local). then run docker-compse -f mongo.yml up
4. check the mongo express in the localhost:8080.

```
22
23  app.post('/update-profile', function (req, res) {
24    var userObj = req.body;
25
26    MongoClient.connect("mongodb://admin:password@mongodb", function (err, client) {
27      if (err) throw err;
28
29      var db = client.db('my-db');
30      userObj['userid'] = 1;
31
32      var myquery = { userid: 1 };
33      var newvalues = { $set: userObj };
34
35      db.collection("users").updateOne(myquery, newvalues, {upsert: true}, function(err, res) {
36        if (err) throw err;
37        client.close();
38      });
39
40    });
41    // Send response
42    res.send(userObj);
43  });
44
45  app.get('/get-profile', function (req, res) {
46    var response = {};
```

Before is @localhost:3000. But in the docker-compose. It will directly know MangoDB and the port is already configured. So that you can even not write

Volumes: data persistence



Volume types

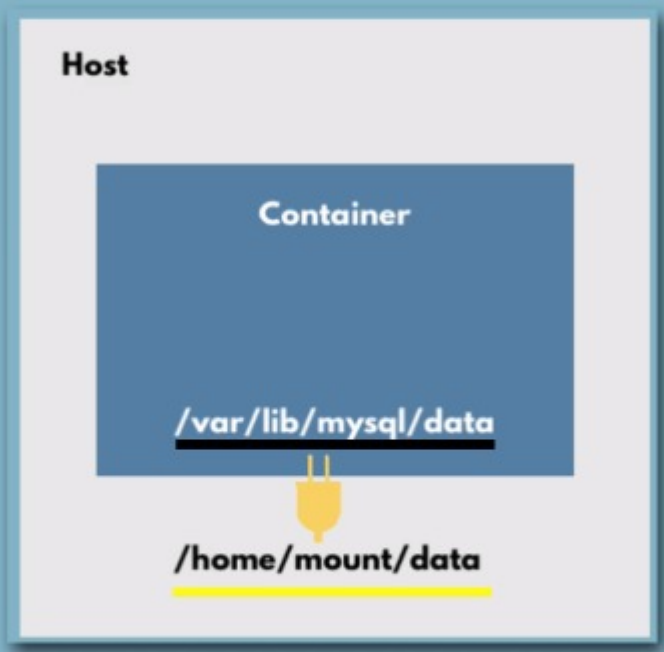
3 Volume Types

- docker run

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

Host Volumes

- you decide **where on the host file system** the reference is made

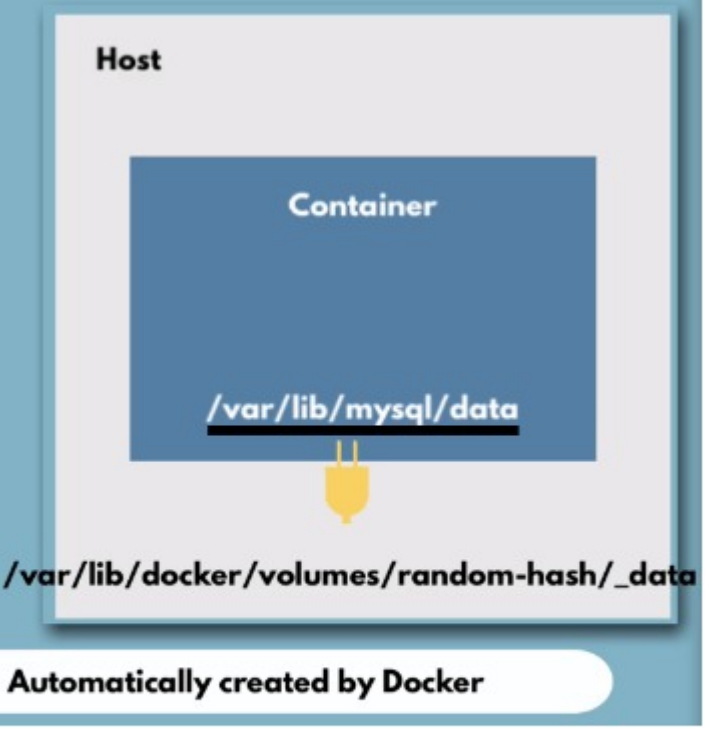


3 Volume Types

- docker run
  -v /var/lib/mysql/data

Anonymous Volumes

- for **each container a folder is generated** that gets mounted



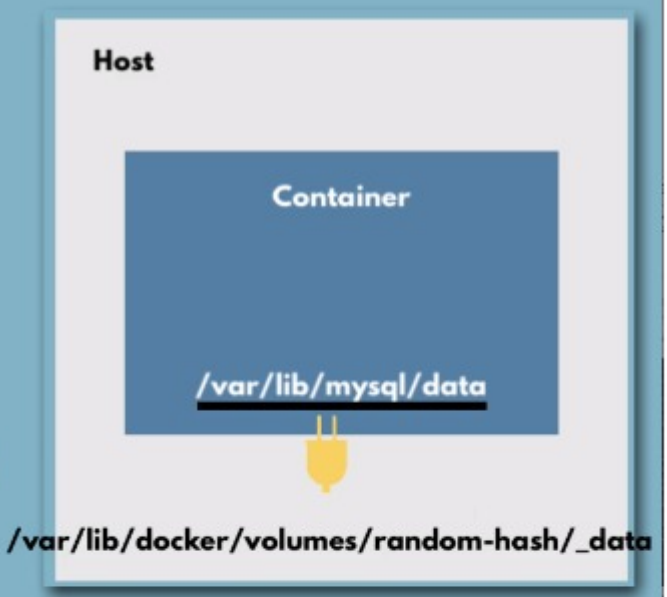
Automatically created by Docker

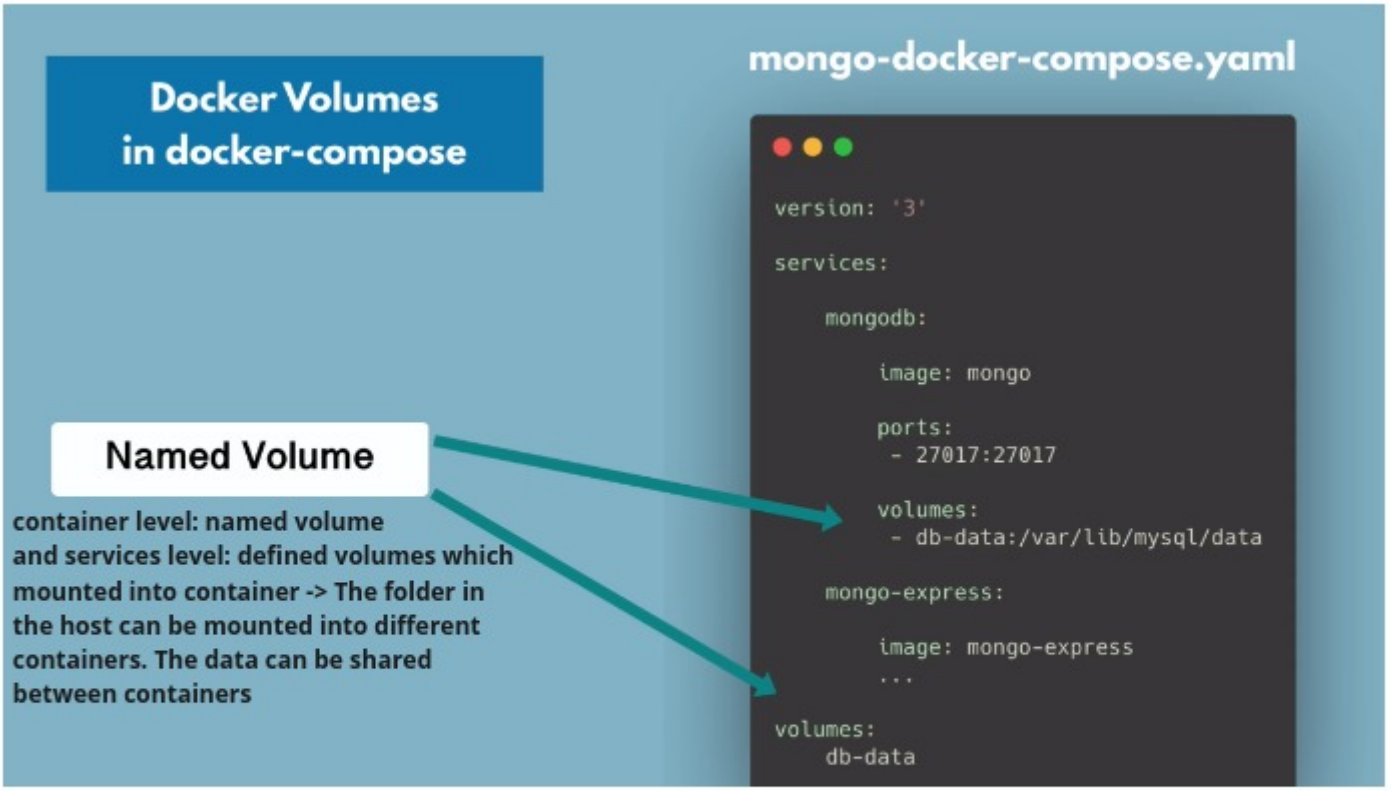
3 Volume Types

- docker run
  -v **name:**/var/lib/mysql/data

Named Volumes

- you can **reference** the volume **by name**




Docker Volumes
in docker-compose
mongo-docker-compose.yaml
version: '3'
services:
mongodb:
image: mongo
ports:
- 27017:27017
volumes:
- db-data:/var/lib/mysql/data
mongo-express:
image: mongo-express
...
volumes:
db-data
Named Volume
container level: named volume
and services level: defined volumes which
mounted into container -> The folder in
the host can be mounted into different
containers. The data can be shared
between containers

Using the docker-compose to start the two containers (docker-compose -f mongo.yml
Then check it in the mongoDB
start the application: npm start
create volumes:

- services level volume (give a name, provide a driver-> driver:local) Info for docker to create a storage file in the local
- container level: volumes: mongo-db:/data/db (/data/db default path in the mongoDB

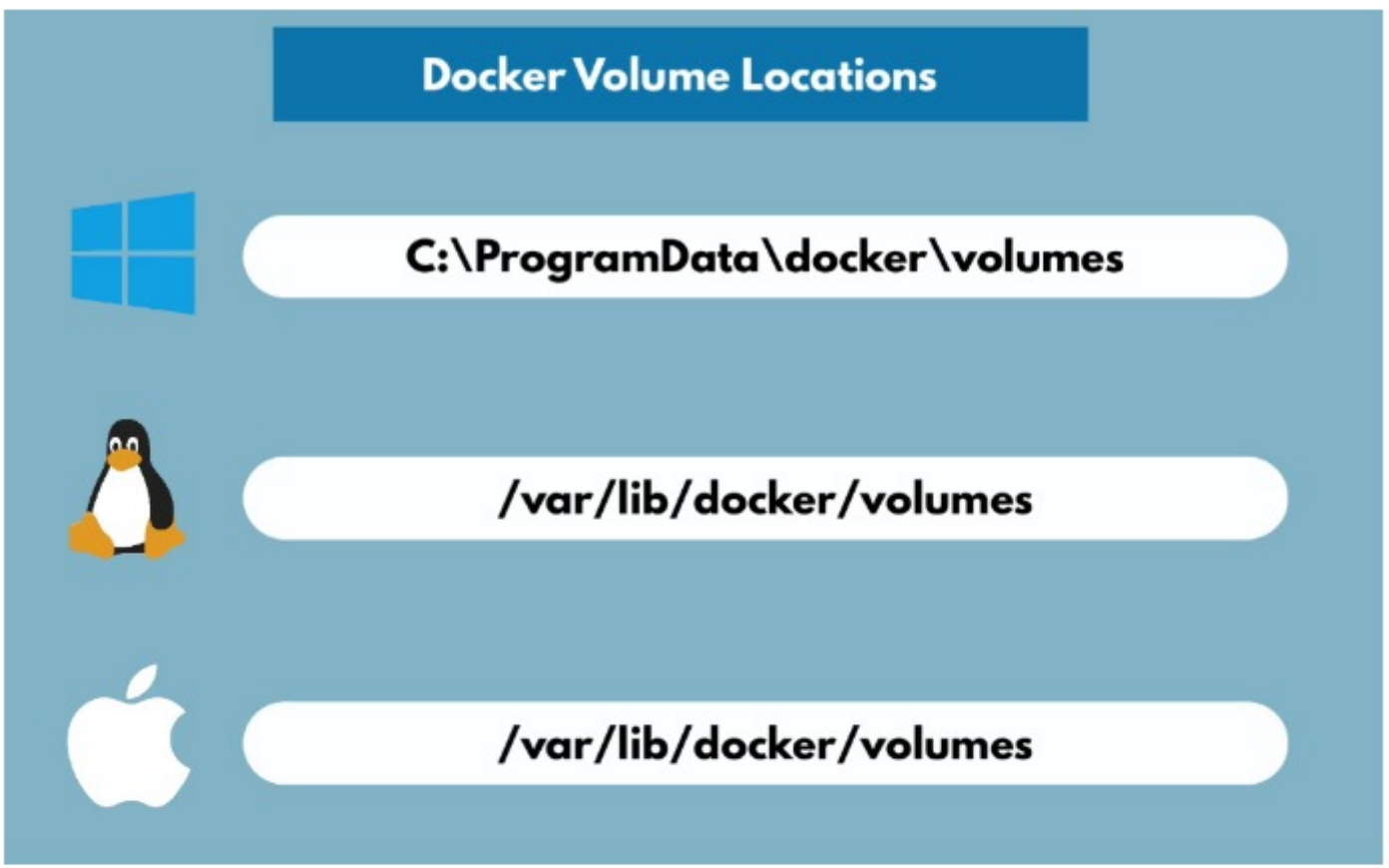
Then restart the docker-compose again

```
# - 3000:3000
mongodb:
  image: mongo
  ports:
    - 27017:27017
  environment:
    - MONGO_INITDB_ROOT_USERNAME=admin
    - MONGO_INITDB_ROOT_PASSWORD=password
  volumes:
    - mongo-data:/data/db
mongo-express:
  image: mongo-express
  ports:
    - 8080:8081
  environment:
    - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
```

mysql: var/lib/mysql

postgres: var/lib/postgresql/data

WHere docker volumes locations???



How to end screen session :

ctrl+a+k mac