(Docker) → Lets you run processes in isolated environm

- Docker pull (think of it like downloading something)

  eg: docker pull ubuntu (basically download an ubuntu docker image in your local system).

- Docker run

  ↳ docker images
  → list all the images that you pulled onto your local system.

docker run ubuntu..

docker ps ↳ lists all the running containers.

docker ps -a ↳ list of containers stopped running in past

* each container when ran has a container ID

eg: CONTAINER ID : 3F5447e9b298

- docker run -it ubuntu ↳ run the container and use it.
  (-it: interactive, attatched.)

* ~~docker pull~~

→ Port mapping: gateway of your local machine to docker container.
  we will use curl to test mapping

* sudo docker run -it -p 7990:80 nginx.
  docker logs 'container_ID' ↳ to generate logs.

→ docker exec : allows you to execute commands from outside into your docker container.

i) Make Python file & docker file for that
2) open terminal in the #PWD.
3) docker build -t kuchbhi .
4) ~~docker run kuchbhi~~

to fix sudo problem

sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker

docker run -it -P 8000:80 -d nginx
↓
make it run in background

→ docker exec -it "container-ID" Bash
any command

→ docker inspect : it inspects and gives information about the container

docker inspect 'container-ID'

→ docker file

1) create a Python file in any directory
2) create a 'dockerfile' in the same directory

~~ADD~~ FROM Python

ADD percent.Py/test

CMD ["Python", "/test/percent.Py"]

docker Build -t project

" " " " " :V2 (manchaitya tag)

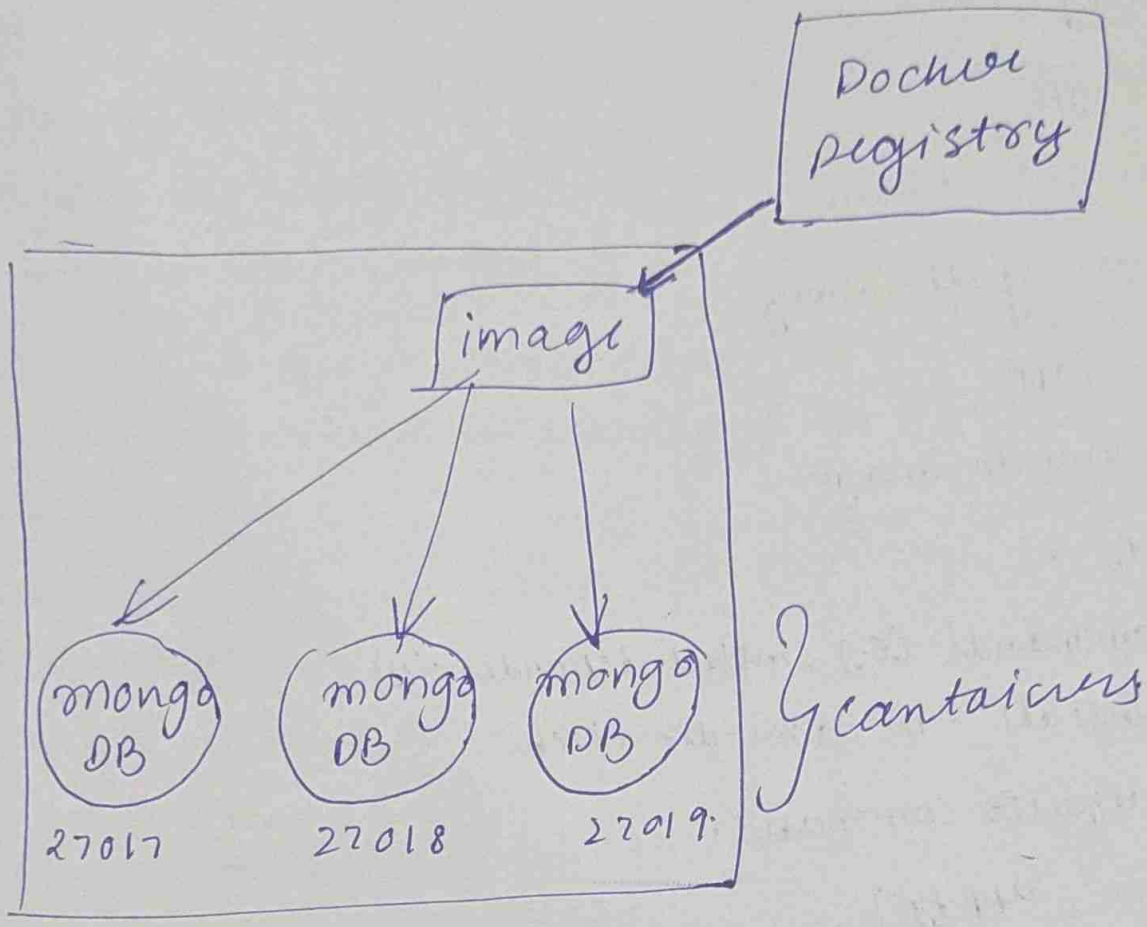
Containers are a way to package and distribute software applications in a way that makes them easy to deploy and run consistantly across different environment. They allow you to package an application along with all its dependencies and libraries, into a single unit that can be run on any machine with a container runtime such as docker

Why containers?

1) every one has different operating system.
2) steps to run a projects can vary based on OS.
3) Extremly harder to keep track of dependencies as project grows.

⇒ A docker image is a light weight, standalone, executable package that includes everything needed to run a software, including the code, a runtime, libraries, environment variables, and config files.

⇒ A docker container is a running instance of an image. It encapsulates the application or service and its dependencies, running in an isolated environment.



docker file.

FROM node:16-alpine ——→ Base image.

WORKDIR /app ——→ working directory

COPY . . ——→ copy over files.

RUN npm install
RUN npm run build ——→ Run commands to build the code.

EXPOSE 3000 ——→ Expose Ports.

CMD ["node", "dist/index.js"] → final command that runs ~~the~~ when ~~the~~ running the container

docker file

```
#1. Base image.
FROM Python:3.11-slim

#2. Set working directory.
WORKDIR /app

#3. Copy files into image
COPY app.py .

#4. Run commands (e.g., install dependencies)
RUN pip install --no-cache-dir flask

#5. Set default command.
CMD["Python", "app.py"]
```

Key instructions

FROM ⟶ specifies base image

WORKDIR ⟶ Set working directory

COPY ⟶ copies file into image

RUN ⟶ Executes command at build time

CMD ⟶ Set default command for container

EXPOSE ⟶ (optional) documents which port the app uses

Use RUN pip install --no-cache-dir flask
↓
to reduce image size

~~Level 2 → docker file for a flask web app~~

Q) Modify the dockerfile to reduce image size by cleaning apt cache after package installation in Ubuntu base image.

→ When using apt in docker (like apt-get install), it often leaves behind temporary files, package caches, and meta data.
these inflates your image size and are not need in the final image.

- it installs packages-
- Bat bups
  - ↳ /var/lib/apt/lists/*
  - ↳ /var/cache/apt/archives/

these stay in the image unless you manually remove them.

→ EXAMPLE OF BAD DOCKERFILE

```
FROM ubuntu:22.04.
RUN apt-get update && apt-get install -y curl git
```

→ CORRECTED DOCKERFILE

```
FROM ubuntu:22.04
RUN apt-get update && \ → Updates the package lists
    apt-get install -y curl git && \ → installs curl & git
    apt-get clean && \ → Removes ~~tag~~ local .deb file from apt cache.
    rm -rf /var/lib/apt/lists/* → Deletes downloaded package list.
```

∋ What is a Multi-Stage Docker Build?

Imagin building a cake:

* You need a big messy kitchen to prepare it
* But you only serve the clean slice on a small plate.

multistage builds works the same:

* First stage: messy tools, compilers, builds stuff.
* Final stage: only the output (binary), no extra junk

Q6) Multi-Stage build for a go application.

You are building a Go app. the first stage compiles the binary, and the second stage runs it.

Task:

write a docker file that uses multistage build to:

* Compile a Go program in one stage.
* Copy the binary to minimal scratch or alpine image.
* Keep the final image under 20MB

ANSWER).

→ Goal of this ques:

1) Build the app (Go compiler needed)
2) Run the app (no compiler needed).

→ Folder structure:

```
go-app/
├── main.go
└── Dockerfile
```

main.go - Simple go App.

```
Package main
import "fmt"
func main() {
    fmt.Println("Hello from a multi staged
        docker build !")
}
```

Dockerfile (easy)

```
FROM golang:1.21-alpine As builder
WORKDIR /app
COPY main.go .
RUN go build -o myapp main.go

FROM alpine:latest
WORKDIR /app
COPY --from=builder /app/myapp .
CMD ["./myapp"]
```

STAGE 1 — Build go app.

alpine ↑ Linux distribution

STAGE 2 — Final runtime Stage.

(2) Docker file for a nodejs app using multi-stage build.

Create a multi stage docker file for a production ready node.js application

assume you have these files:

- Package.json
- Package-lock.json
- Server.js.

Practice on own :-

```
FROM node:18-alpine AS builder
WORKDIR /app
COPY server.js .
RUN node build -o myapp server.js
COPY package.json package-lock.json ./
RUN npm install
COPY . .

FROM alpine:latest.
RUN apk add --no-cache nodejs npm
WORKDIR /app.
COPY --from=builder /app/app.
EXPOSE 3000
CMD ["node", "server.js"]
```

FROM node:18-alpine AS builder → Starts build stage with node.js tool.

COPY package.json ... → copy only dependency info to cache install layer.

RUN npm install → install all needed modules.

COPY . . → copy all app files like app.py.

FROM alpine → Starts fresh lightweight image.

apk add nodejs npm → Add runtime only.

COPY --from=builder ... → copy app + modules from build stage.

CMD ["node", "server.js"] → start your app

LANGUAGE	BUILDER IMAGE	FINAL IMAGE
Node.js	node:18-alpine	alpine.
Python	python:3.11-slim	alpine.
Java	maven or gradle	openjdk:17-alpine
Go	golang:1.20	alpine.
Rust	rust:1.70	alpine.
C/C++	gcc or clang	alpine.
React/angular/vue	node + build tools.	nginx.

Q) a) Multi-stage docker file for react app

Task

You are building a react frontend app & you want to containerize it using multi-stage build

Your project contains

- Package.json
- Package-lock.json
- src/
- public/

```
FROM node:18-alpine AS builder
WORKDIR /app
COPY Package.json Package-lock.json ./
RUN npm install
~~RUN npm run build~~
COPY . .
RUN npm run build
FROM nginx:alpine
~~WORKDIR /app~~ RUN rm -rf /usr/share/nginx/html/*
RUN npm run nodejs
COPY --from=builder /app/~~app~~ build/ usr/share/nginx/html
EXPOSE ~~8000~~
CMD ["nginx", "-g", "daemon off;"]
```

DOCKER 2

Docker compose - It is basically a file which allows us to run multiple docker servers at once with a single command.

It is a tool for defining and running multi-container docker application. Instead of managing individual containers with long 'docker run' commands, you define your entire application stack in a YAML file & manage it with simple commands

What docker compose solves: when you have an application that needs multiple services (like a web app + database + cache) docker compose lets you define all of them together and manage them as a single unit

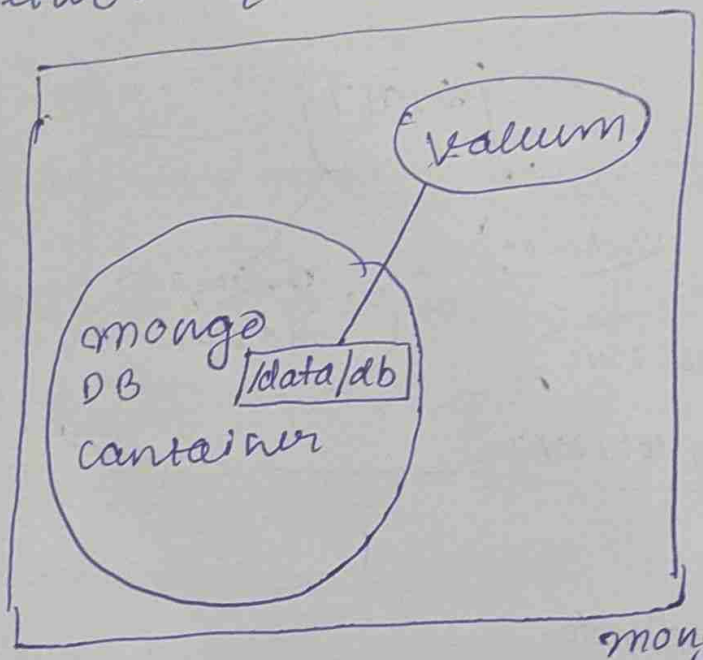
The docker_compose.yml file: This is where you define your services, networks, and volumes. It is the blueprint for your application stack

→ LAYERS IN DOCKER

Layers are a fundamental Part of the image that allows docker to be more efficient, fast and portable. a docker image is essentialy build up from a series of layers, each representing a set of differences from the previous layer.

why layers? → if you change your docker file, layers can get reused based on where the change was made.

→ Networks & volumes.



i) docker volume create ~~db-mongo-data~~ mongo-db-data,

name of volume.

ii) ~~docker run -P 27017:27017 -V /data/db mongo~~

→ docker run -P 27017:27017 -V mongo-db-data: /data/db mongo

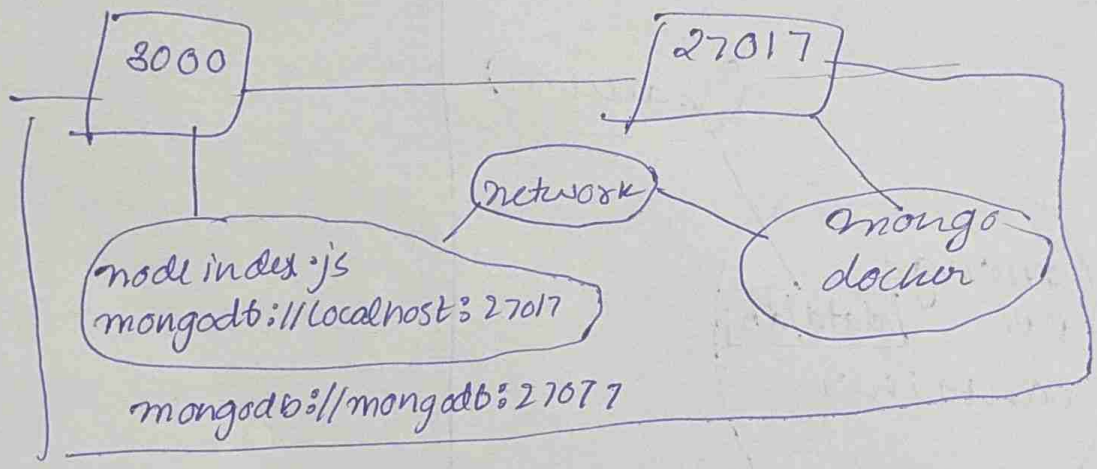
now after running this command the data that was stored in your mongo database will still exists after deleting the container. the data stays in /data/db of volume

~~after~~. now if you restart, as long as you are mounting that /data/db folder to your container your data will get back.

but if you ~~do~~ start the container without the volume mount you can't see any data.

Q) can we mount multiple mongo db container to the same volume?

when you want to talk to the other container running on the machine you use network




```
docker run
  --name mongodb
  -p 27017:27017
  -v mongo-db-data:/data/db
  mongo

docker run -p 3000:3000 user-app
```

How to make containers talk to each other.

1) Clone the repo - http://github.com/100xdevs-cohort-2/week-15

2) Build the network :
   - docker build -t image_tag.

3) Create a network :
   - docker network create my_custom_network

* Start the backend process with the network attatched to it

docker run -d -p 3000:3000 --name backend --network my_custom_network

(detach mode (background) me)

* Start mongo on the same network

docker run -d -v volume_database:/data/db --name mongo --network my_custom_network

* Check the logs to ensure the db connection is successful

docker logs <container_id>

Volume

~~1) container ke aandar~~

volume ka naam

1) volume bana lo → docker volume create newdata

2) container banao aur usme koi file banao.
   → docker run -it --name test-vol -v new:/app/data ubuntu bash.
   → fir us container ke aandar modification karo.

3) dusara container banao with the same volume
   - docker run -it --rm -v newdata:/app/data ubuntu bash

   this container will also have the same file

1) docker volume create shared_volume

2) docker run -d --name writer \
-v shared_volume:/data busybox \
sh -c "echo 'written by container 1' > \
/data/info.txt ; sleep 3600"

* docker run → run a new container
* -d → run the container in the detached mode (in the background).
* --name writer → give the container the name writer)
* -v shared_vol:/data → mount the named volume shared_vol inside the container at path /data
* busybox :→ use the lightweight busybox image.
* rest of the command from sh → this is the command that container runs.

DOCKER COMPOSE

It is a tool designed to help you define and run multi-container docker applications. With compose, you use a Yaml file to configure your application's services, networks, and volumes. Then, with a single command, you can create and start all the services from your configuration.

1) Basic Flask app.

Write a docker-compose.yml that:

- uses python:3.9 image.
- mounts code from ./app to /app.
- installs requirements
- runs python app.py.
- Exposes port 5000.

→

```
version: '3.8'
services:
  web:
    image: python:3.9
    working_dir: /app
    volumes:
      - ./app:/app
    ports:
      - "5000:5000"
    command: sh -c "pip install -r requirements.txt
                    && python app.py"
```

necessary because you won't be able to run it in your browser it will stay inside your container only.

2) Challange : Flask + PostgreSQL

You are building a docker-compose.yml with :

- A `web` service running a flask app (Python 3.9)
- a `db` service using `Postgres:13`
- Your flask app depends on the database.
- Postgre data is stored in a named volume called Pgdata
- App & DB should be on the same custom network.

```
version: '3.6'
services:
  web:
    image: python:3.9
    working-dir: /app
    volumes:
      - ./app:/app
    ports:
      - "5000:5000"
    command: sh -c "pip install -r requirements.txt && python app
  db:
    image: postgres:13
    volumes:
      - ./app:/pgdata
    Environment:
      - POSTGRES_DB=mydb, POSTGRES_USER=myuser,
        POSTGRES_PASSWORD = mypassword.
    network: backend.
```

Incorrect file

→ version: '3.8'

services:

web:

image: python:3.9

working_dir: /app

volumes: ./app:/app

ports:

- "5000:5000"

command: sh -c "pip install -r requirements.txt
&& python app.py"

depends_on:

- db

networks:

- backend

db:

image: postgres:13

environment:

POSTGRES_DB: mydb

POSTGRES_USER: myuser

POSTGRES_PASSWORD: mypassword

volumes:

- pgdata:/var/lib/postgresql/data

networks:

- backend

volumes:

pgdata:

networks:

backend:

```
version: '3.8'
services:
  web:
    build: .
    ~~volumes: .:/app~~
    working_dir: /app
    volumes: ./app:/app
    environment: .env
    Expose: 5000'
    depends_on:
      - db
    networks:
      - backend
  db:
    image: postgres:13
    environment: .env
    volumes:
      - Pgdata:/var/Lib/postgresql/data
    healthcheck: " Idon't know how to write"
    networks:
      - backend
volumes:
  Pgdata:
Networks:
  backend:
```

What is a docker network?

→ a docker network is like a private wifi that lets your containers talk to each other securly and by name.

imagin you are building an app where:

- web (flask app) needs to talk to db (postgres)
- without a network, they can't see each other.
- with a network, they can find each other using their service names (like db, web).

TYPES of docker networks :-

	When to use	Scope	IP Assignment
i) bridge	→ Default for single-host apps	Local only	Auto assigned & by default
ii) host	→ Advanced cases, faster networking	Shares host network	No isolation.
iii) none	→ no network at all	—	—
iv) custom	→ for multi container apps.	Local	Named.

We always use Custom Networks.

Benefit 1) container name-based DNS.

- web can talk to db just by using db:5432.
- No IPs, no guesswork.

Benefit 2) only containers in the same network can talk, keeps external apps outside unless you expose ports.

→ docker-compose.yml

1. PostgreSQL.

```
version: "3.8"
Services:
  db:
    image: postgres:15
    restart: always. ——→ restarts the container if it crashes.
    environment: ——→ Set environment variable for postgresql
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydb

    Volumes:
      - pgdata:/var/lib/postgresql/data.
                          ↳ location where postgresql stores its
    Ports:                    data.
      - "5432:5432"

volumes:
  pgdata;
```

2. MYSQL.

```
Services:
  mysql:
    image: mysql:8  ⟶ uses mysql image.
    restart: always.
    environment:  ⟶ create database & user automatically.
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: appdb.
      MYSQL_USER: user.
      MYSQL_PASSWORD: pass.
    ports:
      - "3306:3306"
    volumes:
      - mysqldata:/var/lib/mysql
                  ↳ location where mysql stores the data.
volumes:
  mysqldata:
```

3) Redis

```
Services:
  redis:
    image: redis:alpine
    ports:
      - "6379:6379"
```

* Runs redis from the tiny alpine image.
* Exposes redis port 6379 so apps can connect to it

4. Node.js App.

```
Services:
  nodeapp:
    build:
      context: .
    ports: "3000:3000"
    volumes:
      - .:/app
    command: ["node", "app.js"]
```

When to use `build: .`

You uses this when you have your own source code and a docker file that defines how to build the image.

When to use `image: node`

You just want to run a node container. You're not building an image with your own app.

→ Node.js + MongoDB App.

```
version: "3.8"
services:
  node:
    build: .                                  first
    ports:
     - "5000:5000" ----> "3000:3000"
    volumes:
     - .:/app
    depends_on:
     - mongo
    networks:
     - here

  mongo:
    image: mongo:nanoserver → Latest
    volumes:
     - dbdata:/var/lib/db → /data/db } mongo db stores data in /data/db.
    networks:
     - here
networks:
  here:
```

2) Practice round 2 ⇒ FLASK + POSTGRESQL APP.

```
version: "3.8"
services:
  web:
    build: .
    working_dir: /app
    volumes: ./app:/app
    depends_on:
      - db
    ports: "5000:5000"
    depends_on:
      - db
    command: sh -c "pip install -r requirements.txt && python app.py"
    networks:
      - Backend
  db:
    image: postgreSQL:latest
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: Password
      POSTGRES_DB: mydb
    volumes:
      - pgdata:/var/lib/postgresql/data
    ports: "5432:5432" → Postgres default Port.
    networks:
      - Backend
  volumes:
    pgdata:
  networks:
    Bbackend:
```

no need bcz you already install in Dockerfile.

3) Nodejs + Redis app

```
Version: "3.8"
Services:
  api:
    build: .
    working_dir: /app
    Ports: "3000:3000"
    depends_on:
      - redis
    network: backend
  redis:
    image: redis:alpine
    Ports: "6379:6379"
    network: backend.

networks:
    backend:
```

EKDUM SAHI

NAILED IT

(3) Create docker-compose.yml for python FastAPI + MySQL app.

```
version: "3.8"
services:
  web:
    build: .
    working_dir: /app
    ports: "8000:8000"
    volumes: ./app:/app
    depends_on:
      - mysql
    networks:
      - backend
  mysql:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: mydb
      MYSQL_USER: user
      MYSQL_PASSWORD: pass
    ports: "3306:3306"
    volumes: mysqldata:/var/lib/mysql
    networks: backend
volumes:
  mysqldata:
networks:
  backend:
```

command: uvicorn app:app --host 0.0.0.0 --port 8000 --reload.

server used to run FASTAPI

when you start this container run the FastAPI app using uvicorn (a lightweight server)

uvicorn → the server used to run FastAPI

app:app → First app → filename app.py

second app → your FastAPI object : app = FastAPI()

--host 0.0.0.0 → expose it to outside the container

--Port 8000 → run it on port 8000

--reload → Auto-reload on code changes.

COMMAND IN FAST API

uvicorn app:app --host 0.0.0.0 --port 8000 --reload.

---

python manage.py runserver 0.0.0.0:8000 } for terminal.
↳ in django to start the development server, we run this command in terminal.

So. in docker-compose.yml, you write :

↳ command: python manage.py runserver 0.0.0.0:8000.

0.0.0.0 → It tells Django to listen all network interfaces, so it works inside docker.

manage.py → special helper script that Django automatically creates when you start a new django project

---

to run redis with persistence enabled, we usually start redis server with a flag.

command: redis-server --appendonly yes

(2*) Why would someone want to override CMD from the docker file.

→

(1) different environment needs different commands.

→ In development, you may want auto-reloading and debug mode.

In Performance, you want performance & stability.

Example :-

command: flask run --host=0.0.0.0 --reload.

command: gunicorn main:app --bind 0.0.0.0:5000.

(2) Testing or temporary behaviour.

→ Sometimes you want to run one-off commands in the same container.

example:-

command: python manage.py migrate.

command: python manage.py runserver 0.0.0.0:8000

(3) container reuse across projects.

→ you might use same dockerfile for API apps, background worker, database seeder. each one may need a different command.

command: celery -A app.tasks worker.

command: python seed-db.py.

(4) cleaner Dockerfile.

Lets the dockerfile be generic.

CMD ["python", "app.py"]

then use command: to inject more advanced behaviour per environment without editing dockerfile.

→ Nodejs + MongoDB.

```
version: "3.8"
services:
  web:
    build: .
    ports: "3000:3000"
    depends_on:
      - mongo
    command: node server.js.
    environment:
      - MONGO_URL = mongodb://mongo:27017/myapp.

  mongo:
    image: mongo:6
    volumes:
      - mongo_data:/data/db.

volumes:
  mongo_data:
```

→ Django + PostgreSQL

```
version: "3.8"
services:
  web:
    ~~image~~
    build: .
    working_dir: /app
    ports: "8000:8000"
    volumes: ~~./app~~ ./app:/app
    command: uvicorn app:app --host 0.0.0.0 --port 8000 --reload
    networks: backend
    depends_on: db
  db:
    image: postgres:latest
    ports: "5432:5432"
    volumes: pgdata:/var/lib/postgresql
    networks: backend
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
~~networks:~~
volumes:
  pgdata:
networks:
  backend:
```

⇒ correction:

command: python manage.py runserver 0.0.0.0:8000

→ Flask + Redis.

```
version: "3.6"
services:
    web:
      build: .
      working_dir: /app
      ports: "5000:5000"
      volumes: ./app:/app
      command: python app.py
      networks: backend
      depends_on: redis

    redis:
       image: redis:alpine
       ports: "6379:6379"
       ~~volumes~~
       networks: backend

networks:
    backend:
```

→ Flask + Redis + PostgreSQL

```
version: "3.8"
services:
  web:
    build: .
    working_dir: /app
    ports: "5000:5000"
    volumes: .:/app
    command: python app.py
    depends_on:
      - redis
      - db
    network:
      - backend

  redis:
    image: redis:alpine
    ports: "6897:6397"
    network: backend

  db:
    image: postgres:14
    ports: "5432:5432"
    environment:
      POSTGRES_USER = Flaskuser
      POSTGRES_PASSWORD = flaskpass
      POSTGRES_DB = flaskdb
    volumes: pgdata:/var/lib/postgresql/data
    networks: backend

volumes:
  pgdata:
networks:
  backend:
```

→ Fastapi + Redis + Postgresql

```
version: "3.8"
services:
    web: /
      build: .
      working-dir: /app
      Ports: "8000:8000"
      volumes: :/app
      command: uvicorn app:app --host 0.0.0.0 --Port 8000 --reload
      depends_on:
        - redis
        - db
      network: Backend.

    Redis:
      image: redis:alpine
      Ports: "6397:6397"
      networks: backend.

    db:
      image: postgres:14
      Ports: "5432:5432"
      environment:
          POSTGRES:USER = user
          POSTGRES_DB = mydata
          POSTGRES_Password = Pass.
      volumes: Pgdata:/ var/lib/ Postgresql/data.
      network: backend.

volumes:
   Pgdata:
networks:
   backend:
```

(8) How to make your docker container automatically restart if the server reboots

→ docker run -d --restart unless-stopped <image_name>

(9) Your container is crashing with the exit code 137. what does this mean and what could be the likly cause? how would you start debugging it?

→ Exit code 137 means the container was killed by the system not by your app. specifically it usualy maps to.

137 = 128 + 9 → which means the container recieved signal 9 (SIGKILL)

MOST COMMON CAUSE:

OUT OF MEMORY.
when your container tries to use more RAM than is available, the Linux OOM killer terminates it with signal 9. that leads to exit code 137.

HOW TO DEBUG AND FIX?

(1) Check logs : docker logs <container_id>

(2) Look at system memory:
- Run `dmesg | grep -i kill` on the host to check if the kernel OOM killer trigg-ered.

(3) Reduce app memory usage. or....

(4) increase memory limits for the container.

docker run -m 512m --memory-swap 1g ...

Q) Dockerfile for simple Node.js express app.

```
FROM node:20.-alpine
WORK_DIR /app.
COPY Package.json Package.json
COPY Package-lock.json Package-lock.json
RUN npm install
COPY . .
EXPOSE 8000.
CMD ["node", "index.js"]
```

Q) React + multi-stage Build + Nginx.

```
FROM node:20-alpine as builder
WORK_DIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM nginx: stable-alpine
WORK_DIR /app . no need
COPY --from=builder /app/myapp
EXPOSE 80
```

→ FASTAPI + POSTGRESQL compos

```
version: "3.8"
services:
  web:
    build: .
    working_dir: /app
    volumes: ./app:/app
    ports: "8000:8000"
    command: uvicorn app:app --host 0.0.0.0 --port 8000
    [redacted]
    depends_on:
      - db
    networks: backend

  db:
    image: postgres:16
    environment:
      POSTGRES_USER: user
      "          _PASSWORD: pass
      "          _DB: data
    volume: pgdata:/var/lib/postgresql/data
    networks: backend

volumes:
  pgdata:

networks:
  backend:
```

```
FROM python:3.11-slim
WORK_DIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

Dockerfile

```
FROM python:3.11-slim
WORK_DIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

↓

for development only

instead use:

```
CMD ["gunicorn", "myproject.wsgi:application",
     "--bind", "0.0.0.0:8000"]
```

```
FROM node:latest-alpine.
WORK_DIR /app
COPY Package*.json .
RUN npm install.
COPY . .
[redacted]
EXPOSE 3000
CMD["npm", "start"]
```

Part 3

```
FROM node:18.
WORK_DIR /app.
COPY Package*.json .
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

ARG lets you pass the value at ~~runtime~~ buildtime.

```
ARG MY_NAME=Guest
RUN echo "Hello $MY_NAME"
```

Build like this:
→ docker build --build-arg MY_NAME=Alice.
It will print: Hello Alice.

But you cannot access ARG in a running container unless you copy it to ENV

ENV is a runtime variable. it sets a variable that's inside the container when it runs.

```
ENV APP_MODE = development
CMD ["sh", "-c", "echo Mode is $APP_MODE"]
```

when you docker run, it will say:
- Mode is development

you can also override it at runtime.

docker run -e APP_MODE=production my_image.

Syntax in shell (and Dockerfile RUN)

```
if [condition] ; then
  do something
else
  do something
fi # end of the list.
```

```
FROM python:3.11-slim AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .

FROM python:3.11-slim AS runner
WORKDIR /app
COPY --from=builder
EXPOSE 8000
CMD ["python", "app.py"]
```

---

to scale a service (e.g., web) to 3 instances in docker compose:

1) use CLI command.

docker-compose up --scale web=3

this tells docker to launch 3 replicas of web service.

```
name: Docker CI

on:
  push:
    branches: [main]
              ↳ change the actual branch.

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    steps:
      - name: checkout code.
        uses: actions/checkout@v3

      - name: Login to Docker Hub
        uses: docker/login-action@ v3
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build Docker image.
        run: docker build -t ${{ secrets.DOCKER_USERNAME }}/
                                                myapp:latest
                                change the actual name

      - name: Push Docker image.
        run: docker push ${{ secrets.DOCKER_USERNAME }}/
                                                myapp:latest
```

PROJECT

↦ Structure :-

```
node-docker-hello/
├── Dockerfile
├── .github/
│   └── workflows/
│       └── docker-ci.yml
├── package.json
└── app.js
```

1) APP SETUP

- Language: Node.js
- File: app.js with simple http server or express app returning "Hello from docker"
- Dependencies: listed in package.json.

2) Dockerfile

```
FROM node:alpine
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
CMD ["npm", "start"]
```

③ Local docker test.

> docker build -t node-hello .

> docker run -p 3000:3000 node-hello.

app accessable at http://localhost:3000

④ Github repository.

- create a GitHub repository and push the folder to github.

MukulB0412/node-docker-hello.

⑤ GitHub Secrets.

-Added to repo settings.

- DOCKER_USERNAME
- DOCKER_PASSWORD.

pto.

Why we need multistage builds.?

→ If you install all dependencies and tool in a single dockerfile, the final image became bloated, hard to maintain, and slow to start.

Solution:

↳ Split the build process into stages:

1) Stage one install dependencies (builder)

2) another stage copies only what needed to run the app (runtime).

```
FROM python:3.11-slim as builder.
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

FROM python:3.11-slim as runner
WORKDIR /app
COPY --from=builder /root/.local /root/.local
COPY . .
CMD ["python", "app.py"]
```

env file

```
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONBUFFERED=1
ENV PYTHONUNBUFFERED=1
```

nginx

RUN rm -rf /usr/share/nginx/html/* } clears out the default Nginx webpage.

Vite → Localhost:5173

Dockerfile.backend

FROM node:18-alpine as builder
WORKDIR /app

COPY frontend/package*.json ./
RUN npm install

COPY frontend/ ./
RUN npm run build

FROM nginx:alpine as runner
COPY --from=builder /app/dist /usr/share/nginx/html
EXPOSE 80