

Dockerfile) → lets you run processes in isolated environment

- Docker pull (think of it like downloading something)

e.g.: 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 which ran has a container ID

e.g.: CONTAINERID: 3F5447e9b298

- docker run -it ubuntu ↳ run the container
interactive attached.

↳ Docker port

↳ 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.

- 1) 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'

→ dockerfile

1) Create a python file in any directory

2) Create a 'dockerfile' in the same directory

ADD FROM Python

ADD percent.py /tree

CMD ["python", "/tree/percent.py"]

docker build -t project

11 11 4 " " :v2 {manchaitanya}

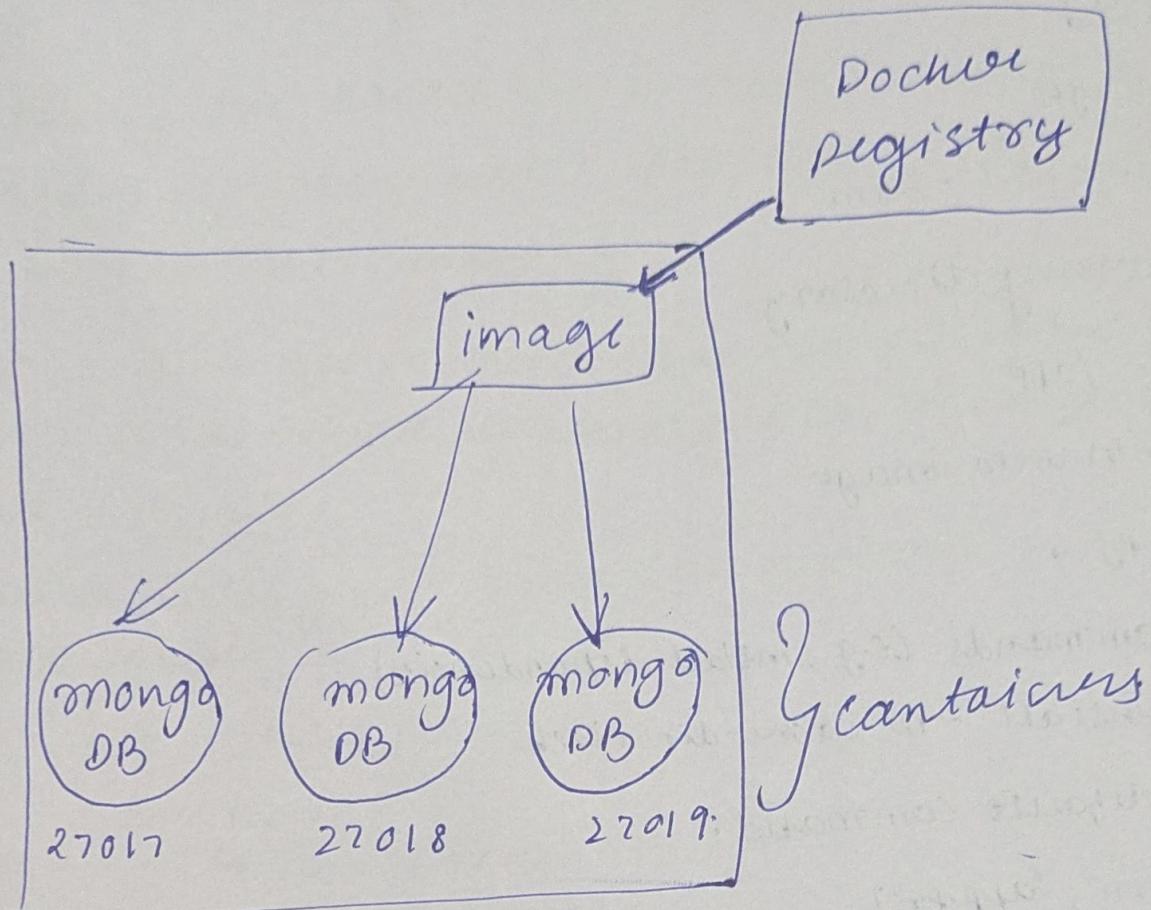
Containers are a way to package and distribute software applications in a way that makes them easy to deploy and run consistently across different environments. They allow you to package an application along with all its dependencies and libraries, into a single unit that can be run on any machine with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

Why containers?

- 1) every one has different operating systems.
- 2) steps to run a project can vary based on OS.
- 3) extremely 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 commands to build the code.

RUN npm run build

EXPOSE 3000 → expose ports.

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

Dockerfile

#1. base image.

FROM Python:3.11-bullseye

#2. Set working directory.

WORKDIR /app

#3. Copy file 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 → dockerfile 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, packages caches, and meta data.
- these inflates your image size and are not need in final image.

- it installs packages-
- But keeps
 - ↳ `/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 local deb file from
rm -rf '/var/lib/apt/lists/*' ↳ deletes downloaded package
list.`

Q) What is a Multi-Stage Docker Build?

Imagine 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 Dockerfile 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 stage,  
    docker build !").
```

{

Dockerfile (easy)

FROM golang:1.21-alpine AS builder.

WORKDIR /app

COPY main.go .

RUN go build --o myapp main.go

FROM ^{linux distribution} alpine:latest.

WORKDIR /app

COPY --from=builder /app/myapp .

CMD ["./myapp"]

STAGE 1

Build go app.

STAGE 2

Final runtime
Stage.

Q) 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 ./myapp
RUN npm install
COPY . ./myapp

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 any 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

node.js

python

java

go

rust

c/c++

react/
angular/vue

BUILDER IMAGE

node:18-alpine

python:3.11-slim

maven or gradle

golang:1.20

rust:1.70

gcc or clang

node + build tools.

FINAL IMAGE

alpine

alpine

openjdk:17-alpine

alpine

alpine

alpine

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;"]

EDOCKER 2)

Docker compose - it is basically a file which allows us to run multiple docker services 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 webapp + database + cache) docker compose lets you define all of them together and manage them as a single unit

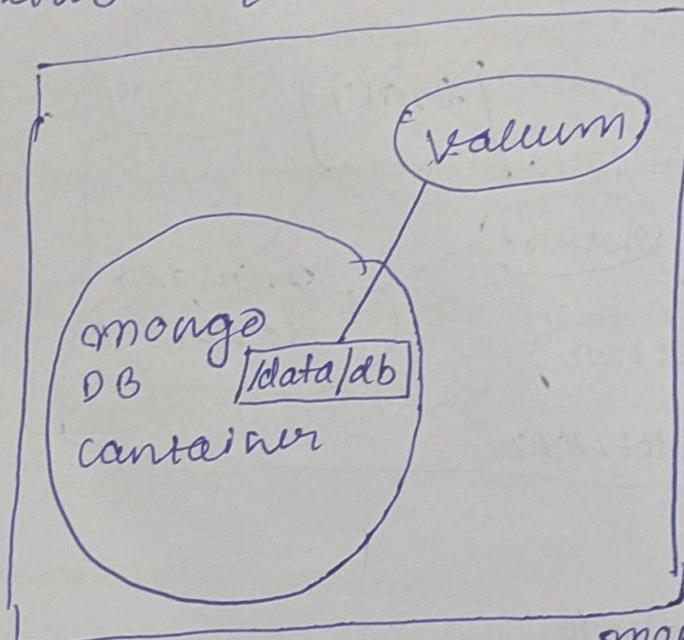
The docker-compose.yml file: this is where you define your services, networks, and volumes. It is the blueprint for your application stack

→ LAYERS IN DOCKER

Layers are fundamental part of the image that allows docker to be more efficient, fast and portable. A docker image is essentially built 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

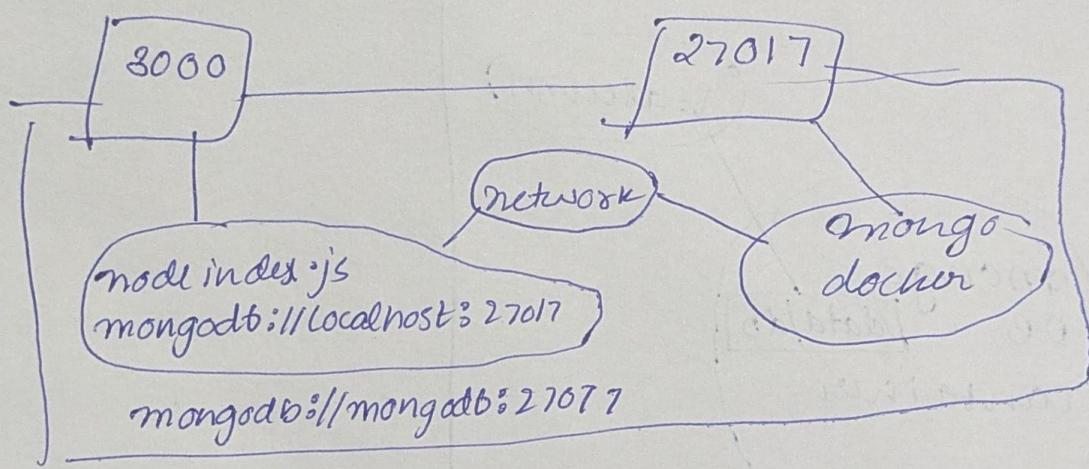
Now after running this command the data that was stored in ~~the~~ your mongo database will still exist after deleting ~~the~~ container. The data stays in /data/db for invalid

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 won'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 mongadb  
-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-convirt-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 attached to it
- ```
docker run -d -P 8000:8000 --name backend --network
 ↓
 detach mode (background) new my-custom-network
```
- \* 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 aadhar~~
- D volume bana Lo → docker volume create newdata
- Volume ka naam ↑
- 2) Container banao aur usme koi file banao.
- docker run -it --name test -v newdata:/app/data
- ubuntu bash.
- fir us container ke aadhar modification kرو.
- 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.' > \ /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-volume:/data → mount the named volume shared-volume 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: '8.8'

services:

web:

image: Python:3.9

working-dir: /app

volumes:

- ./app:/app

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

ports:

- "5000:5000"

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

## 2) challenge : 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.
- Postgres 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.py"

## 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: '8.8'

Services:

www:

build: .

~~volumes: /app:/app~~

working-dir: /app

volumes: ./app:/app

environment: • env

Expcse: 5000

depends-on:

- db

networks:

- backend

db:

image: postgres:13

environment: • env

volumes:

- Pgdata:/var/lib/postgresql/data

healthcheck: "I don'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 securely and by name.

Imagine you are building an app onion:

- `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 :-

|            | <u>When to use</u>                  | <u>Scope</u>        | <u>IP Assignment</u>     |
|------------|-------------------------------------|---------------------|--------------------------|
| 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

↳ 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.

Ports:

- "5432:5432" data' → location where PostgreSQL stores its data

volumes:

pgdata:

## Q. MySQL.

Services:

mysql:

image: mysql:8 → uses mysql image.

restart: always.

environment:

MYSQL\_ROOT\_PASSWORD: root

→ create database & user automatically.

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 use 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: .

fixes

Ports:

- "5000:5000" → "3000:3000"

volumes:

- .:/app

depends-on:

- mongo

networks:

- hehe

mongo:

image: mongo:mongodb → latest

volumes:

- dbdata :/var/lib/db → /data/db 3 mongo db stores data  
in /data/db

networks:

- hehe

networks:

hehe:

## Q) Practice round 2 of FLASK + POSTGRESQL APP.

Version: "8.8"

Services:

web:

build:

working-dir: /app

volumes: .:/app:/app

depends-on:

parts: "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:

- backend

### 3) Nodejs + Redis app

Version: "3.0.8"

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:

E K D U M S A H I

N A I L E D   I T

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

version: "8.8"

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: unicorn APP:app --host 0.0.0.0  
--port 8000 --reload.  
when you start this container  
run the FastAPI app using  
unicorn(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

(Q\*) 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.

## ⑪ cleaner Dockerfile.

lets the dockerfile be generic:

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

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

→ Node.js + MongoDB.

version: "3.8"

services:

node:

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

networks: backend

depends-on: db

X X X X

- db:

image: postgres:latest

Ports: "5432:5432"

volumes: pgdata:/var/lib/postgresql

networks: backend

volumes:

POSTGRES\_DB: mydb

POSTGRES\_USER: user

POSTGRES\_PASSWORD: pass

volumes:

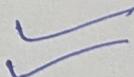
pgdata:

networks:

backend:

→ corrections

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"

~~networks~~

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: "6397:6397"

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

- ~~pg~~ 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:

- Q) How to make your docker container automatically restart if the server reboots  
→ docker run -d --restart unless-stopped <image-name>
- Q) Your container is crashing with the exit code 137.  
What does this mean and what could be the likely cause?  
How would you start debugging it?
- Exit code 137 means the container was killed by the system  
not by your app. Specifically it usually maps to:

$137 = 128 + 9 \rightarrow$  which means the container received 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 triggered.
- ③ Reduce app memory usage. or...
- ④ Increase memory limits for the container.

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

① Dockerfile for simple Node.js express app.

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

② React + multi-stage Build + Nginx.

```
FROM node:20-alpine as builder
```

```
WORKDIR /app
COPY package*.json .
```

```
RUN npm install
```

```
COPY . .
```

~~FROM alpine build~~

```
RUN npm run build
```

```
FROM nginx:stable-alpine
```

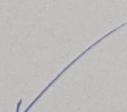
WORKDIR /app - no need

```
COPY --from=builder /app/myapp
```

~~FROM alpine build~~

```
EXPOSE 80
```

~~WORKDIR /app~~



→ FASTAPI + PostgreSQL compos

Version: "3.8"

Services:

web:

build:

working-dir: /app

volumes: .:/app:/app

Ports: "8000:8000"

command: unicorn app:app --host 0.0.0.0 --port 8000

~~depends-on~~

depends-on:

- db

networks: backend

db:

image: postgres:16

environment:

POSTGRES\_USER: user

" - POSTGRES\_PASSWORD: pass

" - DB: data-

volumes: pgdata:/var/lib/postgresql/data

networks: backend

volumes:

\*pgdata\*

networks:

backend:

```
FROM python:3.11-slim
WORKDIR /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
WORKDIR /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.
WORKDIR /app
COPY package*.json .
RUN npm install.
COPY . .
██████████
EXPOSE 3000
CMD ["npm", "start"]
```

### Part 3

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

ARG lets you pass the value at ~~runtime~~ build time.

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)

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~~

```
RUN pip install --no-cache-dir -r requirements.txt
```

```
COPY . .
```

```
FROM python:3.11-slim AS runner
```

```
WORKDIR /app
```

```
COPY --from=builder
```

~~EXPOSE~~

```
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 du 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 on 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 accessible 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.

pt9.

Boe

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 installs dependencies (build)
- 2) another stage copies only what's needed to run the app (runtime).

```
FROM python:3.11-slim as builder env-file
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 PYTHONDONTWRITEBYTECODE=1  
~~ENV PYTHONUNBUFFERED=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