>_

Docker Compose

Toàn Tập

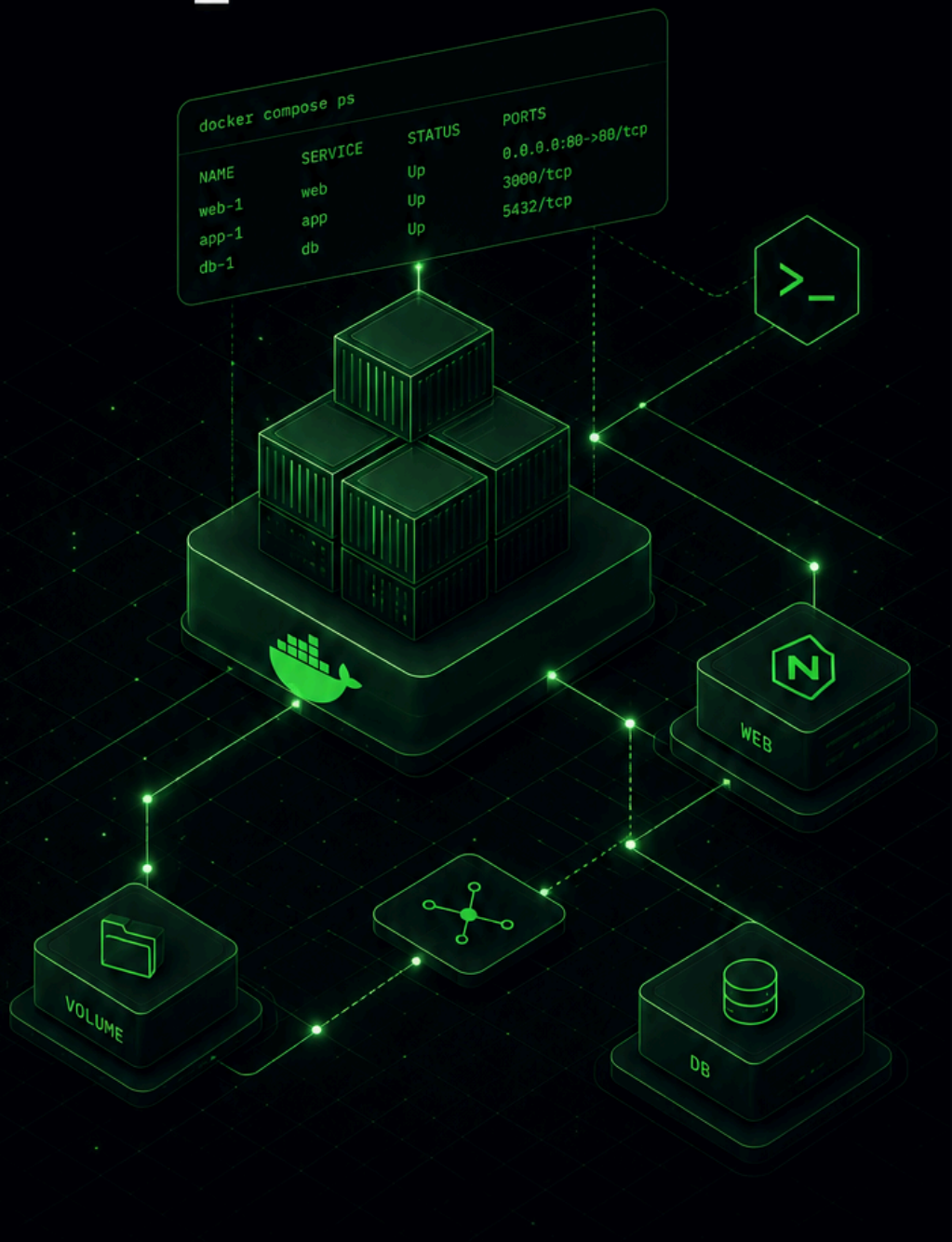
compose.yaml

```yaml
services:
  web:
    image: nginx:alpine
    ports:
      - "80:80"
    depends_on:
      - app
  app:
    build: ./app
    environment:
      - NODE_ENV=production
    depends_on:
      - db
  db:
    image: postgres:15-alpine
    volumes:
      - db_data/var/lib/postgresql/data

volumes:
  db_data:
```

CONTAINERS	7	RUNNING
SERVICES	12	HEALTHY
IMAGES	9	LOCAL
VOLUMES	5	ATTACHED
NETWORKS	4	ACTIVE

```
$ docker compose up -d
$ docker compose ps
$ docker compose logs -f
$ docker compose down
$ docker compose config
```



Do Cao Hieu

Table of Contents

PHẦN IV — AI & VẬN HÀNH

PHẦN

Phần I — Nền tảng

01

Nền tảng — Docker & Compose

Trước khi đụng vào file `compose.yaml`, phải nắm chắc vài khái niệm gốc. Nếu phần này vững thì mọi từ khóa phía sau đều dễ hiểu, vì chúng chỉ là cách *khai báo* lại những thứ ông vốn làm bằng tay với lệnh `docker`.

Container, Image, Dockerfile — bộ ba phải phân biệt

Nhiều người mới học hay lẫn ba thứ này. Hiểu bằng ẩn dụ nấu ăn:

Thuật ngữ	Là gì	Ẩn dụ
Dockerfile	File text chứa *công thức* build nên một image (cài gói, copy code, set lệnh chạy)	Công thức món ăn
Image	Gói đóng băng (read-only) gồm OS tối giản + app + thư viện, build ra từ Dockerfile	Nguyên liệu sơ chế đông lạnh
Container	Một *tiến trình đang chạy* tạo ra từ image — có thể tạo/xóa/nhân bản thoải mái	Món ăn nấu xong đang bốc khói

Điểm cốt lõi: **1 image → tạo ra N container**. Image bất biến; container thì sống/chết liên tục. Khi container chết, mọi dữ liệu ghi bên trong nó *mất hết* — đây chính là lý do sau này phải học `volumes`.

Docker Engine, Daemon, CLI

- **Docker Engine** — toàn bộ phần mềm Docker chạy trên máy.
- **Docker Daemon** (`dockerd`) — tiến trình nền thực sự quản lý container, image, network. Ông không gõ lệnh trực tiếp với nó.
- **Docker CLI** (`docker`) — công cụ dòng lệnh ông gõ; nó gửi yêu cầu cho daemon qua API.

Khi gõ `docker run nginx`, CLI bảo daemon: "lấy image nginx, tạo container, chạy đi". Compose cũng chỉ là một lớp bọc gọi xuống cùng daemon đó.

Docker Compose là gì, vì sao cần

Một app thật hiếm khi chỉ có 1 container. Ví dụ web app điển hình:

- 1 container **web** (nginx/frontend)
- 1 container **api** (backend)
- 1 container **db** (PostgreSQL)
- 1 mạng riêng để chúng nói chuyện
- 1 volume để DB không mất dữ liệu

Nếu làm tay, ông phải gõ 3-4 lệnh `docker run` dài loằng ngoằng, tự tạo network, tự nhớ thứ tự. Cực và dễ sai.

> **Docker Compose** giải quyết bằng cách: ông *khai báo* toàn bộ "dàn nhạc" đó trong **một file YAML duy nhất** (`compose.yaml`), rồi chỉ cần một lệnh `docker compose up` là tất cả tự dựng lên đúng thứ tự.

Declarative vs Imperative — tư duy quan trọng nhất

Kiểu	Nghĩa	Ví dụ
Imperative (ra lệnh)	Ông nói *từng bước phải làm gì*	Gõ tay `docker run`, `docker network create`...
Declarative (khai báo)	Ông mô tả *kết quả mong muốn*, để Docker tự lo cách đạt được	Viết `compose.yaml` rồi `up`

Compose là **declarative**. Ông viết "tôi muốn có web + api + db như thế này", Compose so sánh với trạng thái hiện tại rồi tự tạo/xóa/sửa cho khớp. Đây là lý do chạy `docker compose up` lần hai sẽ không tạo trùng — nó thấy "đã khớp rồi" thì thôi.

`compose.yaml` — file trung tâm

Tên file chuẩn hiện nay là `compose.yaml` (hoặc `compose.yml`). Tên cũ `docker-compose.yml` vẫn chạy được nhưng là quy ước cũ. Compose tự tìm file theo thứ tự ưu tiên trong thư mục hiện tại.

YAML dùng **thụt lề bằng dấu cách** (KHÔNG dùng tab) để biểu diễn cấu trúc lồng nhau. Sai thụt lề = lỗi cú pháp. Đây là cái bẫy phổ biến nhất với người mới.

```yaml
services:        # top-level key
  web:           # tên service (thụt 2 dấu cách)
    image: nginx # thuộc tính của service (thụt 4 dấu cách)
```

Vòng đời cơ bản — những lệnh phải thuộc lòng

Lệnh	Tác dụng
`docker compose up`	Dựng & chạy toàn bộ stack (thêm `-d` để chạy nền)
`docker compose down`	Dừng & xóa container + network (volume giữ lại trừ khi thêm `-v`)
`docker compose ps`	Liệt kê container đang chạy trong stack
`docker compose logs`	Xem log (thêm `-f` để theo dõi realtime)
`docker compose build`	Build lại image từ `build:`
`docker compose config`	In ra file đã được merge/validate — cực hữu ích để debug

> **Mẹo vàng:** Trước khi `up`, luôn chạy `docker compose config` để xem file sau khi gộp anchor/include/biến môi trường trông ra sao. Nó bắt lỗi cú pháp trước khi ông tốn thời gian.

Bức tranh tổng thể của một file Compose

Một file đầy đủ gồm các **top-level object** (đối tượng cấp cao nhất) — đây chính là thứ ông hỏi ban đầu:

Top-level	Vai trò	Mới/Cũ
`services`	Các container — trái tim của file	Cũ
`networks`	Mạng ảo để container nói chuyện	Cũ
`volumes`	Lưu trữ bền, không mất khi container chết	Cũ
`configs`	File cấu hình non-sensitive bơm vào container	Cũ
`secrets`	Dữ liệu nhạy cảm (mật khẩu, key)	Cũ
`models`	Model AI (Docker Model Runner)	**MỚI**

Cộng thêm vài key bổ trợ: `name`, `include`, `x-*` (extension). Các chương sau sẽ mổ xẻ *từng key, từng thuộc tính con* một.

02

Cấu trúc file & các key cấp cao

Chương này giải thích bộ khung của `compose.yaml`: các key cấp cao nhất *không phải* là object container (`name`, `include`, `x-*`) và quy tắc YAML cần biết để không vấp lỗi.

`version` — và vì sao đừng dùng nó nữa

Ngày xưa file luôn mở đầu bằng `version: "3.8"`. **Giờ bỏ đi.** Từ Compose v2, key này *obsolete* (lỗi thời) — Compose tự dùng schema mới nhất. Nếu ông để lại, Compose còn in cảnh báo. Quy tắc: **không viết** `version:`.

`name` — đặt tên project

```
name: my-stack
```

`name` đặt tên cho cả project. Tên này thành *tiền tố* cho mọi tài nguyên Compose tạo ra: container `my-stack-web-1`, network `my-stack_default`, volume `my-stack_db_data`. Nếu không khai báo, Compose lấy tên thư mục chứa file làm project name.

Vì sao quan trọng: nó là *không gian tên* (namespace) cô lập các stack với nhau. Hai project khác tên chạy song song không đụng nhau. Tương đương cờ `-p` của CLI.

`include` — tách stack lớn ra nhiều file

```
include:
  - ./common/compose.yaml
  - path:
      - ./base.yaml
      - ./override.yaml
    env_file: ./.env
    project_directory: ..
```

`include` kéo nội dung file Compose khác vào như thể nó nằm sẵn trong file này. Khác với `extends` (kế thừa từng service), `include` gộp *nguyên cả file*. Dùng khi stack lớn, muốn chia theo nhóm (vd: `compose.db.yaml`, `compose.monitoring.yaml`) cho dễ quản lý.

Các thuộc tính con:

Thuộc tính	Ý nghĩa
`path`	Đường dẫn 1 file, hoặc list nhiều file (file sau override file trước)
`env_file`	File `.env` áp dụng riêng cho phần include này
`project_directory`	Thư mục gốc để giải các đường dẫn tương đối bên trong file được include

`x-*` — extension fields (mở rộng)

Mọi key bắt đầu bằng `x-` được Compose **bỏ qua khi chạy**, nhưng vẫn parse được. Nó là chỗ để ông nhét dữ liệu tái sử dụng. Kết hợp với *YAML anchor* tạo ra sức mạnh DRY (Don't Repeat Yourself — đừng lặp lại).

```yaml
x-common-env: &common-env       # &common-env = "anchor" (cái mỏ neo)
  TZ: Asia/Ho_Chi_Minh
  LOG_LEVEL: info

services:
  web:
    environment:
      <<: *common-env            # *common-env = "alias", <<: = "merge vào đây"
      NODE_ENV: production        # vẫn thêm/đè key riêng được
```

Giải thích từng ký hiệu YAML:

Ký hiệu	Tên	Tác dụng
`&tên`	Anchor (mỏ neo)	Đánh dấu một khối để tái dùng
`*tên`	Alias (tham chiếu)	Chèn lại khối đã neo
`<<:`	Merge key	Gộp các key của khối được tham chiếu vào map hiện tại

Kết quả: `web.environment` sẽ có `TZ`, `LOG_LEVEL` (từ anchor) + `NODE_ENV`. Sửa anchor một chỗ, mọi nơi dùng nó đổi theo. Cực hợp cho cấu hình lặp như `logging`, `restart`, `environment` chung.

Nội suy biến môi trường — `${VAR}`

Trong file Compose, ông có thể chèn biến môi trường từ shell hoặc file `.env`:

```yaml
services:
  web:
    image: "myapp:${TAG}"            # lấy TAG từ môi trường
    ports:
      - "${HOST_PORT:-8080}:80"      # nếu HOST_PORT trống thì mặc định 8080
```

Các dạng cú pháp:

Cú pháp	Nghĩa
`${VAR}`	Thay bằng giá trị VAR (rỗng nếu không có)
`${VAR:-default}`	Dùng `default` nếu VAR không set *hoặc* rỗng
`${VAR-default}`	Dùng `default` chỉ khi VAR *không set*
`${VAR:?error msg}`	Báo lỗi và dừng nếu VAR trống — ép buộc phải khai báo
`$$`	Một dấu `$` thật (thoát, để Compose không hiểu là biến)

File `.env` đặt cạnh `compose.yaml` được Compose tự nạp để điền các biến này. Đây là cách tách cấu hình thay đổi (port, tag, mật khẩu dev) ra khỏi file chính.

> **Phân biệt dễ nhầm:** `.env` ở cấp project (điền biến *trong* file YAML) **khác** với thuộc tính `env_file:` của service (nạp biến *vào trong container*). Chương Services sẽ nói rõ.

Thứ tự đọc của Compose

Khi chạy `docker compose up`, Compose ghép cấu hình theo lớp, đè dần:

1. File `compose.yaml` chính
2. File override `compose.override.yaml` (nếu có — tự động merge)

3. Các file `-f` truyền thêm trên CLI (đè theo thứ tự)
4. Biến từ `.env` và môi trường shell được nội suy

Muốn xem kết quả cuối sau khi gộp hết: `docker compose config`. Luôn dùng lệnh này khi nghi ngờ.

PHẦN

Phần II — Services (lõi)

03

Services — Image, Build, Lệnh & Biến môi trường

`services` là top-level quan trọng nhất: mỗi mục con bên dưới nó là *một service* (thường tương ứng một container). Chương này mổ nhóm thuộc tính đầu tiên: **service lấy image ở đâu, chạy lệnh gì, nhận biến môi trường nào**.

```
services:
  web:            # "web" là TÊN service — dùng để service khác gọi tới qua DNS
    image: nginx
```

Tên service (`web`) đồng thời là *hostname* trong mạng nội bộ: service `api` có thể gọi `http://web` là tới. Nhớ kỹ điểm này.

`image` — lấy image có sẵn

```
image: postgres:16-alpine
```

`image` chỉ định image dùng cho service. Định dạng đầy đủ: `[registry/]repository:tag`.

Phần	Ví dụ	Ý nghĩa
registry	`docker.io`, `ghcr.io`	Kho chứa (mặc định Docker Hub)
repository	`library/postgres`	Tên image
`tag`	`:16-alpine`, `:latest`	Phiên bản

> **Cảnh báo `:latest`**: nó *không* nghĩa là "mới nhất luôn cập nhật". Nó chỉ là một nhãn như mọi nhãn khác, dễ gây "máy tôi chạy được, máy anh thì không". Production nên **pin tag cụ thể** (`:16.2`) để tái lập được.

`build` — tự build image từ Dockerfile

Khi không có image dựng sẵn, ông build từ source. Dạng ngắn:

```
build: ./web        # thư mục chứa Dockerfile
```

Dạng đầy đủ (long syntax) — nắm hết các con key này rất hữu ích:

```
build:
  context: ./web            # thư mục gốc để gửi cho daemon + giải đường dẫn COPY
  dockerfile: Dockerfile    # tên file (đổi nếu dùng Dockerfile.prod)
  target: production        # build tới đúng stage này (multi-stage build)
  args:                     # biến dùng LÚC BUILD (ARG trong Dockerfile)
    APP_ENV: prod
  cache_from:               # mượn layer cache từ image này cho nhanh
    - myapp:cache
  platforms:                # build đa kiến trúc
    - linux/amd64
    - linux/arm64
```

```
  no_cache: false           # true = bỏ cache, build sạch
  pull: true                # luôn kéo base image mới nhất
  secrets:                  # secret chỉ tồn tại lúc build, không nằm trong image
    - build_token
```

Giải thích các con key dễ nhầm:

Con key	Giải thích
`context`	Thư mục "bối cảnh" — toàn bộ file ở đây được gửi cho Docker daemon; lệnh `COPY` trong Dockerfile lấy từ đây
`dockerfile`	Đường dẫn Dockerfile *tương đối với context*
`target`	Trong Dockerfile nhiều `FROM ... AS <stage>`, chọn dừng ở stage nào (vd build dev vs prod)
`args`	Truyền giá trị cho `ARG` trong Dockerfile — chỉ có lúc build, KHÔNG còn khi container chạy
`cache_from`	Tái dùng layer đã build sẵn để build nhanh hơn (hay dùng trong CI)

> **Phân biệt `args` (build) vs `environment` (runtime):** `args` là biến *lúc đóng gói image*; `environment` là biến *lúc container chạy*. Mật khẩu KHÔNG bao giờ để trong `args` vì nó dính vào lịch sử image.

Có thể khai cả `image` lẫn `build`: Compose build rồi *đặt tên* image kết quả theo `image`.

`command` — đè CMD

```
command: ["nginx", "-g", "daemon off;"]
```

`command` ghi đè chỉ thị `CMD` trong Dockerfile — tức *lệnh chính* container chạy. Viết được 2 kiểu:

- **Exec form** (khuyến nghị): `["nginx", "-g", "daemon off;"]` — mảng, không qua shell, xử lý tín hiệu dừng đúng.
- **Shell form**: `command: nginx -g 'daemon off;'` — chạy qua `/bin/sh -c`, tiện nhưng dễ vướng vấn đề tín hiệu.

`entrypoint` — đè ENTRYPOINT

```
entrypoint: /docker-entrypoint.sh
```

`entrypoint` ghi đè `ENTRYPOINT` — *chương trình* luôn chạy, và `command` trở thành *tham số* truyền cho nó. Quan hệ: `ENTRYPOINT` + `CMD` = lệnh hoàn chỉnh. Đa số trường hợp chỉ cần đụng `command`.

`environment` — biến môi trường runtime

```
environment:            # kiểu map
  NODE_ENV: production
  API_URL: "http://api:8080"
# hoặc kiểu list:
# environment:
#   - NODE_ENV=production
```

`environment` đặt biến môi trường *bên trong container đang chạy*. App đọc qua `process.env`, `os.environ`... Đây là cách chuẩn cấu hình app 12-factor.

`env_file` — nạp biến từ file

```
env_file:
  - path: ./.env.app
    required: false      # không có file cũng không lỗi
  - ./secrets.env
```

`env_file` nạp hàng loạt biến từ file `KEY=VALUE` vào container. Tiện khi có nhiều biến, không muốn liệt kê dài trong YAML.

> **Ba cái "env" dễ loạn — chốt lại:** - `.env` (cấp project, cạnh compose) → điền `${VAR}` *trong* file YAML. - `env_file:` (thuộc tính service) → nạp biến *vào trong* container. - `environment:` → khai từng biến *vào trong* container, đè được `env_file`.

Định danh và ngữ cảnh chạy

```
container_name: my-web     # tên container cố định
hostname: web-host         # hostname BÊN TRONG container
working_dir: /app          # thư mục làm việc khi chạy command
user: "1000:1000"          # chạy với uid:gid này (bảo mật: tránh root)
```

Thuộc tính	Giải thích	Lưu ý
`container_name`	Đặt tên cố định cho container	**Mất khả năng scale** — không nhân được nhiều bản; cẩn thận
`hostname`	Tên máy nội bộ container tự thấy	Khác với tên service dùng để service khác gọi
`working_dir`	Thư mục mặc định khi chạy lệnh	Như `cd` trước khi chạy
`user`	UID:GID tiến trình chạy dưới	Đặt non-root để giảm rủi ro bảo mật

04

Services — Cổng, Mạng, Phụ thuộc & Healthcheck

Nhóm thuộc tính này quyết định service *kết nối với thế giới* ra sao: mở cổng nào, vào mạng nào, đợi service nào, và làm sao biết nó "khỏe".

`ports` — ánh xạ cổng ra host

Đây là chỗ container "thò" dịch vụ ra ngoài cho máy host (và internet) truy cập.

```
ports:
  - "8080:80"                # HOST:CONTAINER
  - "127.0.0.1:8443:443"     # chỉ bind vào 1 IP host
```

Cú pháp ngắn `"HOST:CONTAINER"`: lưu lượng đến cổng `8080` của máy host → chuyển vào cổng `80` của container.

> **Bẫy kinh điển:** số bên trái là cổng host, số bên phải là cổng container. Nhớ sai là app "chạy mà không vào được".

Cú pháp dài (long syntax) — rõ ràng, kiểm soát tốt hơn:

```
ports:
  - target: 9000          # cổng BÊN TRONG container
    published: "9000"     # cổng PHƠI RA host
    host_ip: 0.0.0.0      # IP host để bind (0.0.0.0 = mọi interface)
    protocol: tcp         # tcp | udp
    mode: host            # host | ingress (chỉ liên quan Swarm)
```

Con key	Ý nghĩa
`target`	Cổng app lắng nghe bên trong container
`published`	Cổng người ngoài gõ vào trên host
`host_ip`	Giới hạn bind vào IP cụ thể — `127.0.0.1` = chỉ máy local truy cập được
`protocol`	`tcp` (mặc định) hay `udp`

`expose` — mở cổng nội bộ, KHÔNG ra host

```
expose:
  - "3000"
```

`expose` chỉ tuyên bố "service này lắng nghe cổng 3000" cho các service *cùng mạng* gọi tới. Nó **không** publish ra host — người ngoài không truy cập được. Dùng cho DB, cache nội bộ không nên lộ ra ngoài.

> **`ports` vs `expose`:** `ports` = mở cửa ra đường phố (host/internet). `expose` = mở cửa thông phòng nội bộ. DB nên `expose`, web nên `ports`.

`networks` (ở cấp service) — gắn service vào mạng

```yaml
networks:
  frontend: {}
  backend:
    aliases:
      - web-internal        # tên gọi thay thế trong mạng này
    ipv4_address: 172.28.1.10   # IP tĩnh (cần ipam config ở top-level)
```

`networks` liệt kê service này thuộc những mạng nào (mạng định nghĩa ở top-level `networks`). Một service vào nhiều mạng để phân tách lưu lượng (vd: web ở cả `frontend` lẫn `backend`, db chỉ ở `backend`).

Con key	Ý nghĩa
`aliases`	Tên DNS phụ — service khác gọi được bằng các tên này
`ipv4_address`	Gán IP cố định trong mạng (mạng phải khai `ipam` subnet)

Nếu không khai `networks`, service tự vào mạng `default` mà Compose tạo sẵn — và mọi service trong đó thấy nhau qua tên service. Đây là lý do "không cấu hình gì mà api vẫn gọi được db".

`depends_on` — thứ tự và điều kiện khởi động

Dạng ngắn (chỉ đảm bảo *thứ tự khởi động*, không đảm bảo "sẵn sàng"):

```yaml
depends_on:
  - db
  - api
```

Dạng dài (mạnh hơn nhiều — đợi *điều kiện*):

```yaml
depends_on:
  db:
    condition: service_healthy   # đợi healthcheck của db báo OK
    restart: true                # db restart thì service này cũng restart
  api:
    condition: service_started   # chỉ cần api đã khởi động
```

Các giá trị `condition`:

Giá trị	Nghĩa
`service_started`	Container phụ thuộc đã *bắt đầu chạy* (chưa chắc sẵn sàng)
`service_healthy`	Container phụ thuộc đã *pass healthcheck* — an toàn nhất
`service_completed_successfully`	Container phụ thuộc đã *chạy xong và thoát mã 0* (dùng cho job khởi tạo)

> **Hiểu lầm tai hại:** `depends_on` dạng ngắn chỉ xếp *thứ tự bật*, KHÔNG đợi app sẵn sàng. DB "đã chạy" nhưng có thể chưa nhận kết nối. Muốn thật sự đợi → phải `condition: service_healthy` + có `healthcheck` ở service db.

`healthcheck` — định nghĩa "thế nào là khỏe"

```yaml
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost/health"]
  interval: 30s        # bao lâu kiểm tra một lần
  timeout: 10s         # mỗi lần chờ tối đa bao lâu
  retries: 3           # sai liên tiếp mấy lần thì coi là "unhealthy"
  start_period: 40s    # thời gian ân hạn lúc mới khởi động, sai không tính
  start_interval: 5s   # trong start_period thì kiểm tra dày hơn
```

`healthcheck` dạy Docker cách tự kiểm tra sức khỏe container. Kết quả hiện ở cột STATUS (`healthy` / `unhealthy`) và là điều kiện cho `depends_on: service_healthy`.

Con key	Ý nghĩa
`test`	Lệnh kiểm tra; thoát mã 0 = khỏe, khác 0 = bệnh. `CMD` = chạy trực tiếp, `CMD-SHELL` = qua shell
`interval`	Chu kỳ giữa các lần kiểm tra
`timeout`	Quá thời gian này coi như lần kiểm tra thất bại
`retries`	Số lần thất bại liên tiếp trước khi gắn nhãn `unhealthy`
`start_period`	Giai đoạn khởi động được "tha" — thất bại không bị tính
`start_interval`	Tần suất kiểm tra trong `start_period` (Compose mới)

Tắt healthcheck của image kế thừa: `test: ["NONE"]`.

`restart` — chính sách tự khởi động lại

```yaml
restart: unless-stopped
```

`restart` quyết định Docker làm gì khi container thoát.

Giá trị	Hành vi
`no`	Không bao giờ tự restart (mặc định)
`always`	Luôn restart, kể cả khi ông chủ động dừng rồi máy khởi động lại
`on-failure`	Chỉ restart khi thoát với mã lỗi (khác 0)
`unless-stopped`	Restart trừ khi ông *chủ động* dừng nó — lựa chọn phổ biến cho service nền

`dns` và `extra_hosts` — phân giải tên

```yaml
dns:
  - 1.1.1.1
extra_hosts:
  - "host.docker.internal:host-gateway"   # trỏ về máy host
  - "myhost:192.168.1.100"
```

- `dns` — đặt máy chủ DNS riêng cho container.
- `extra_hosts` — thêm dòng vào `/etc/hosts` của container, ánh xạ tên → IP thủ công. `host-gateway` là từ khóa đặc biệt trỏ về máy host (hữu ích khi container cần gọi service chạy trên host).

05

Services — Mount dữ liệu, Configs, Secrets & Logging

Chương này nói về cách service *truy cập dữ liệu bền* và *nhận file cấu hình/bí mật*. Đây là nhóm hay nhầm lẫn nhất giữa người mới.

`volumes` (ở cấp service) — gắn lưu trữ vào container

Nhớ lại: container chết là mất dữ liệu bên trong. `volumes` giải quyết bằng cách mount một vùng lưu trữ *sống lâu hơn container* vào một đường dẫn trong container.

```
volumes:
  - db_data:/var/lib/postgresql/data    # named volume
  - ./html:/usr/share/nginx/html:ro     # bind mount, read-only
```

Có **3 loại mount** — phải phân biệt rõ:

Loại	Cú pháp	Bản chất	Dùng khi
Named volume	`db_data:/path`	Docker quản lý, khai ở top-level `volumes`	Dữ liệu app/DB cần bền
Bind mount	`./local:/path`	Trỏ thẳng tới thư mục trên host	Code dev, file config, mount qua lại realtime
tmpfs	(xem dưới)	Nằm trong RAM, mất khi container dừng	Dữ liệu tạm, nhạy cảm không muốn ghi đĩa

Hậu tố quyền:

Hậu tố	Nghĩa
`:ro`	Read-only — container chỉ đọc, không ghi
`:rw`	Read-write (mặc định)
`:z` / `:Z`	Nhãn SELinux (chia sẻ / riêng) — cần trên hệ có SELinux

Cú pháp dài (long syntax) cho mount, rõ ràng hơn:

```
volumes:
  - type: bind            # bind | volume | tmpfs
    source: ./config
    target: /etc/app
    read_only: true
  - type: volume
    source: cache_vol
    target: /cache
    volume:
      nocopy: true        # đừng copy dữ liệu sẵn có của image vào volume
```

Con key	Ý nghĩa
`type`	`volume` (named), `bind` (host path), `tmpfs` (RAM)
`source`	Tên volume hoặc đường dẫn host

Con key	Ý nghĩa
`target`	Đường dẫn bên trong container
`read_only`	`true` = chỉ đọc
`nocopy`	Với named volume: không copy nội dung gốc của image vào

`tmpfs` — vùng nhớ tạm trong RAM

```yaml
tmpfs:
  - /run
  - /tmp:size=100m
```

`tmpfs` mount một thư mục nằm hoàn toàn trong RAM. Nhanh, và *biến mất* khi container dừng. Hợp cho file tạm, dữ liệu nhạy cảm không muốn chạm đĩa.

`configs` (ở cấp service) — bơm file cấu hình

```yaml
configs:
  - source: app_config       # tên config khai ở top-level
    target: /etc/app/app.conf
    uid: "1000"
    gid: "1000"
    mode: 0440
```

`configs` lấy một config (định nghĩa ở top-level) và đặt nó thành *file* bên trong container. Khác với bind mount file: config được Docker quản lý, có thể version, và (trong Swarm) phân phối tới nhiều node.

Con key	Ý nghĩa
`source`	Tên config ở top-level `configs`
`target`	Đường dẫn file trong container (mặc định `/<source>`)
`uid` / `gid`	Chủ sở hữu file
`mode`	Quyền file kiểu octal (`0440` = chủ đọc, nhóm đọc)

Dạng ngắn: chỉ ghi `- app_config` → mount vào `/app_config`.

`secrets` (ở cấp service) — bơm dữ liệu nhạy cảm

```yaml
secrets:
  - db_password              # ngắn → /run/secrets/db_password
  - source: api_key          # dài
    target: api_key
    mode: 0400
```

`secrets` giống `configs` nhưng dành cho *bí mật* (mật khẩu, token, key). Secret được mount vào `/run/secrets/<tên>` dưới dạng file, quyền chặt. Triết lý: app đọc mật khẩu từ *file*, không nhét vào biến môi trường (vì env dễ rò qua log, `docker inspect`).

> **Tại sao secret > environment cho mật khẩu:** biến môi trường lộ ra trong `docker inspect`, log, danh sách tiến trình. File secret ở `/run/secrets/` (tmpfs, RAM) kín hơn và quyền hạn chế hơn. Nhiều image hỗ trợ hậu tố `_FILE` (vd `POSTGRES_PASSWORD_FILE`) chính vì lý do này.

`logging` — cấu hình ghi log

```yaml
logging:
  driver: json-file       # cơ chế lưu log
  options:
    max-size: "10m"       # mỗi file log tối đa 10MB
    max-file: "3"         # giữ tối đa 3 file (xoay vòng)
```

`logging` quyết định log của container (những gì in ra stdout/stderr) được lưu thế nào.

Driver	Mô tả
`json-file`	Mặc định, lưu JSON trên host. *Phải đặt* `max-size` / `max-file` kẻo log phình đầy đĩa
`local`	Tối ưu hơn json-file, tự nén & xoay vòng
`syslog`	Đẩy log sang syslog
`none`	Tắt log hoàn toàn

> **Lỗi sản xuất thường gặp:** quên đặt `max-size` → file log của một container phình tới hàng chục GB, lấp đầy ổ đĩa server. Luôn giới hạn.

`labels` — gắn nhãn metadata

```yaml
labels:
  com.example.description: "Web frontend"
  traefik.enable: "true"
```

`labels` gắn cặp key-value tùy ý lên container. Bản thân Docker không dùng chúng, nhưng *công cụ khác* thì có: reverse proxy như Traefik đọc label để tự cấu hình route, hệ giám sát đọc label để gom nhóm. Quy ước đặt tên theo dạng tên miền ngược (`com.company.xxx`) để tránh trùng.

06

Services — Tài nguyên, Bảo mật, Deploy & Develop

Nhóm thuộc tính cuối của service: giới hạn tài nguyên, cấu hình bảo mật, khối `deploy` (kiểu Swarm), và `develop.watch` (hot-reload). Thêm `profiles` và `extends`.

Giới hạn tài nguyên (chế độ thường)

```yaml
mem_limit: 512m       # RAM tối đa
cpus: 0.5             # tối đa nửa nhân CPU
cpu_shares: 1024      # trọng số CPU khi tranh chấp
pids_limit: 200       # số tiến trình tối đa
shm_size: 128mb       # kích thước /dev/shm (bộ nhớ chia sẻ)
```

Thuộc tính	Ý nghĩa
`mem_limit`	Trần RAM; vượt → bị OOM-kill
`cpus`	Số nhân CPU tối đa (`0.5` = 50% một nhân)
`cpu_shares`	Trọng số tương đối khi nhiều container tranh CPU
`pids_limit`	Chặn fork bomb — giới hạn số tiến trình
`shm_size`	Một số DB/trình duyệt (Chrome) cần `/dev/shm` lớn

Runtime và vòng đời

```yaml
init: true                # chèn init nhỏ làm PID 1, dọn tiến trình zombie
read_only: true           # filesystem container chỉ-đọc (an toàn hơn)
stop_grace_period: 30s    # đợi bao lâu trước khi giết cứng
stop_signal: SIGTERM      # tín hiệu gửi khi dừng
```

Thuộc tính	Ý nghĩa
`init`	Bật tiến trình init tối giản làm PID 1 → thu hồi zombie, xử lý tín hiệu đúng
`read_only`	Khóa filesystem gốc; app chỉ ghi được vào volume/tmpfs đã mount
`stop_grace_period`	Khoảng "graceful" để app tự dọn trước khi bị SIGKILL
`stop_signal`	Tín hiệu dừng tùy chỉnh (mặc định SIGTERM)

Bảo mật — capabilities và quyền

```yaml
cap_add:
  - NET_ADMIN             # thêm năng lực kernel cụ thể
cap_drop:
  - ALL                  # bỏ HẾT rồi chỉ thêm cái cần (best practice)
security_opt:
  - no-new-privileges:true
sysctls:
  net.core.somaxconn: 1024
ulimits:
```

```
  nofile:
    soft: 20000
    hard: 40000
```

Thuộc tính	Ý nghĩa
`cap_add`	Cấp thêm "capability" Linux (quyền nhỏ của root) cho container
`cap_drop`	Tước capability; `ALL` rồi add lại = nguyên tắc đặc quyền tối thiểu
`security_opt`	Tùy chọn bảo mật; `no-new-privileges` chặn leo thang đặc quyền
`sysctls`	Chỉnh tham số kernel cho riêng container (vd hàng đợi kết nối)
`ulimits`	Giới hạn tài nguyên kiểu Unix; `nofile` = số file descriptor mở được

---

`devices` — gắn thiết bị phần cứng

```
devices:
  - "/dev/ttyUSB0:/dev/ttyUSB0"
```

`devices` cho container truy cập thiết bị host (cổng serial, GPU thô, ổ đĩa). Dạng `host:container`.

`profiles` — bật service theo môi trường

```
profiles:
  - dev
```

`profiles` gắn service vào một/nhiều hồ sơ. Service *không khởi động* trừ khi profile được bật:

```
docker compose --profile dev up      # mới chạy service có profile "dev"
```

Service *không có* `profiles` luôn chạy. Dùng để gom các service phụ (debug tool, seed db, monitoring) chỉ bật khi cần, tránh nặng máy lúc dev thường.

`extends` — kế thừa cấu hình

```
extends:
  service: db          # kế thừa từ service "db"
  # file: ./common.yaml  # có thể từ file khác
command: ["postgres", "-c", "hot_standby=on"]   # rồi đè/thêm
```

`extends` cho một service *thừa hưởng* cấu hình của service khác (cùng file hoặc file ngoài), rồi đè/bổ sung. Khác với anchor YAML (chỉ copy text), `extends` hiểu ngữ nghĩa service và merge thông minh hơn. Tránh lặp khi nhiều service gần giống nhau.

`deploy` — cấu hình triển khai (chủ yếu cho Swarm)

```
deploy:
  mode: replicated
  replicas: 3                # chạy 3 bản giống nhau
  endpoint_mode: vip         # vip | dnsrr
  placement:
    constraints:
      - node.role == worker   # chỉ đặt trên node worker
  resources:
```

```yaml
    limits:
      cpus: "0.50"
      memory: 512M
    reservations:
      cpus: "0.25"
      memory: 256M
      devices:                # GPU
        - driver: nvidia
          count: all
          capabilities: ["gpu"]
  restart_policy:
    condition: on-failure
    delay: 5s
    max_attempts: 3
  update_config:
    parallelism: 2
    delay: 10s
    order: start-first        # start-first | stop-first
```

`deploy` mô tả *cách triển khai quy mô*: số bản sao, đặt ở node nào, tài nguyên, chiến lược cập nhật rolling. Phần lớn key này chỉ có hiệu lực với **Docker Swarm** (`docker stack deploy`); chạy `docker compose up` thường thì Compose chỉ tôn trọng một số (như `resources`, `replicas`).

Con key	Ý nghĩa
`replicas`	Số bản sao của service chạy song song
`placement`	Ràng buộc chọn node để đặt container
`resources.limits`	Trần CPU/RAM cứng (kiểu deploy)
`resources.reservations`	Lượng CPU/RAM/GPU *giữ chỗ* tối thiểu — chỗ khai GPU
`restart_policy`	Chính sách restart kiểu Swarm (chi tiết hơn `restart`)
`update_config`	Cách cập nhật cuốn chiếu (rolling update): bao nhiêu bản một lúc, thứ tự

> **GPU lưu ý:** khai GPU qua `deploy.resources.reservations.devices` với `capabilities: ["gpu"]`. Cần NVIDIA Container Toolkit cài trên host.

`develop.watch` — hot-reload khi sửa code

```yaml
develop:
  watch:
    - action: sync              # đồng bộ file vào container
      path: ./src
      target: /app/src
      ignore:
        - node_modules/
    - action: rebuild           # build lại image khi file này đổi
      path: ./package.json
    - action: sync+restart      # đồng bộ rồi restart container
      path: ./nginx.conf
      target: /etc/nginx/nginx.conf
```

`develop.watch` (Compose mới) theo dõi file trên host và phản ứng khi chúng đổi — không cần build/up lại tay. Bật bằng `docker compose up --watch` (hoặc `docker compose watch`).

`action`	Hành vi
`sync`	Copy file đã đổi thẳng vào container đang chạy (nhanh, hợp code interpret)
`rebuild`	Build lại image & tạo container mới (khi đổi dependency, Dockerfile)

`action`	**Hành vi**
`sync+restart`	Đồng bộ file rồi restart tiến trình (khi đổi file config app cần đọc lại)

Đây là tính năng làm trải nghiệm dev với Docker gần như "live reload" của framework hiện đại.

PHẦN

Phần III — Hạ tầng & Tài nguyên

07

Networks — Mạng (top-level)

Top-level `networks` *định nghĩa* các mạng ảo; service rồi *tham chiếu* tới chúng. Hiểu mạng là hiểu cách container tìm và nói chuyện với nhau.

Mạng mặc định — thứ ông được tặng miễn phí

Nếu file không khai mạng nào, Compose tự tạo một mạng tên `default` và bỏ mọi service vào đó. Bên trong, **mỗi service gọi service khác bằng chính tên service** nhờ DNS nội bộ của Docker.

```yaml
services:
  api:
    image: my-api
  db:
    image: postgres
# api gọi "db:5432" là tới, không cần IP, không cần cấu hình gì.
```

Đây là điểm "thần kỳ" làm Compose tiện: *service discovery* (tự tìm nhau qua tên) hoạt động sẵn.

Khai báo mạng tùy chỉnh

```yaml
networks:
  frontend:
    driver: bridge

  backend:
    driver: bridge
    attachable: true
    internal: false
    enable_ipv6: false
    driver_opts:
      com.docker.network.bridge.name: br-backend
    ipam:
      driver: default
      config:
        - subnet: 172.28.0.0/16
          gateway: 172.28.0.1
          ip_range: 172.28.5.0/24
    labels:
      com.example.description: "internal backend net"
```

Giải thích từng thuộc tính:

Thuộc tính	Ý nghĩa
`driver`	Loại mạng. `bridge` (mặc định, 1 host), `overlay` (nhiều host/Swarm), `host` (dùng chung stack mạng host), `none` (cô lập hoàn toàn), `macvlan` (gán MAC riêng)
`attachable`	`true` = cho phép container *rời* (không thuộc Compose) gắn vào mạng này
`internal`	`true` = mạng **không ra internet** — cô lập, chỉ nội bộ container nói chuyện
`enable_ipv6`	Bật IPv6 cho mạng
`driver_opts`	Tùy chọn truyền thẳng cho driver mạng (vd đặt tên bridge interface)
`ipam`	IP Address Management — quản lý dải IP (xem dưới)

Thuộc tính	Ý nghĩa
`labels`	Nhãn metadata cho công cụ ngoài

`driver` — các loại mạng chi tiết

Driver	Khi nào dùng
`bridge`	Mặc định, mạng ảo trên *một* host. 99% trường hợp dev/single-server
`overlay`	Nối container *qua nhiều host* (Docker Swarm). Cần cho cụm
`host`	Container dùng chung mạng của host — không cô lập, nhanh nhất, mất tính tách biệt
`macvlan`	Container có MAC/IP riêng như máy vật lý trên LAN
`none`	Tắt mạng hoàn toàn — cô lập tuyệt đối

`ipam` — quản lý dải IP

```yaml
ipam:
  driver: default
  config:
    - subnet: 172.28.0.0/16     # dải IP của mạng
      gateway: 172.28.0.1       # cổng ra
      ip_range: 172.28.5.0/24   # giới hạn cấp IP trong dải con này
```

`ipam` (IP Address Management) cho ông kiểm soát dải địa chỉ. Cần khai khi muốn gán **IP tĩnh** cho service (`ipv4_address` ở cấp service phải nằm trong subnet này). Đa số trường hợp để Docker tự lo, không cần đụng.

Con key	Ý nghĩa
`subnet`	Dải IP CIDR của mạng (vd `172.28.0.0/16`)
`gateway`	Địa chỉ cổng ra của mạng
`ip_range`	Thu hẹp dải mà Docker tự cấp phát (chừa chỗ cho IP tĩnh)

`internal` — mạng kín không ra internet

```yaml
networks:
  secure_backend:
    internal: true
```

Cờ `internal: true` cắt đường ra internet của mạng. Container trong đó nói chuyện với nhau được nhưng *không gọi ra ngoài* được. Dùng để cô lập tầng DB/nội bộ khỏi internet — một lớp phòng thủ tốt.

`external` — dùng mạng đã có sẵn

```yaml
networks:
  shared_net:
    external: true
    name: some-existing-network
```

`external: true` bảo Compose: "mạng này *đã tồn tại*, đừng tạo, chỉ gắn vào". Dùng khi nhiều project khác nhau cần chung một mạng (vd reverse proxy Traefik ở stack riêng, các app khác join vào mạng của nó). `name` chỉ ra tên mạng thật bên ngoài.

> `external` **là pattern liên-stack quan trọng:** tạo trước mạng bằng `docker network create shared`, rồi mọi compose project tham chiếu `external: true` để cùng vào. Đó là cách cho các stack độc lập "thấy" nhau.

Tóm tắt vòng đời mạng

- `docker compose up` → tạo mạng (trừ external).
- Service join mạng qua key `networks` cấp service.
- `docker compose down` → xóa mạng Compose tạo (external thì giữ).
- Xem mạng: `docker network ls`, soi chi tiết: `docker network inspect <tên>`.

08

Volumes — Lưu trữ bền (top-level)

Top-level `volumes` *định nghĩa* các named volume; service rồi mount chúng. Đây là cách dữ liệu *sống sót* qua những lần container tạo/xóa.

Vì sao cần volume — vấn đề gốc

Filesystem của container là *tạm thời* (ephemeral). Xóa container → mất sạch. Với database, file người dùng upload, cache... mất là thảm họa. Volume tách dữ liệu ra một vùng do Docker quản lý, *độc lập với vòng đời container*.

```yaml
services:
  db:
    image: postgres
    volumes:
      - db_data:/var/lib/postgresql/data   # mount named volume

volumes:
  db_data:        # khai báo named volume (tối giản — Docker tự lo phần còn lại)
```

Xóa và tạo lại container `db` bao nhiêu lần, dữ liệu trong `db_data` vẫn còn.

Named volume tối giản

```yaml
volumes:
  db_data:
  cache_vol:
```

Chỉ cần ghi tên là đủ. Docker tạo volume với driver `local`, lưu ở `/var/lib/docker/volumes/<project>_db_data/` trên host. Phần lớn nhu cầu dừng ở đây.

Named volume đầy đủ

```yaml
volumes:
  cache_vol:
    driver: local
    driver_opts:
      type: none
    labels:
      com.example.role: "cache"
```

Thuộc tính	Ý nghĩa
`driver`	Loại volume. `local` (mặc định, trên host); plugin khác cho NFS, cloud, v.v.
`driver_opts`	Tùy chọn truyền cho driver — chỗ cấu hình bind/NFS (xem dưới)
`labels`	Nhãn metadata
`name`	Đặt tên thật tùy ý (bỏ qua tiền tố project)

Bind một thư mục host thành volume

```yaml
volumes:
  bind_vol:
    driver: local
    driver_opts:
      type: none
      o: bind
      device: /data/app      # thư mục thật trên host
```

Cách này biến một thư mục host cụ thể thành "named volume". Khác với bind mount trực tiếp (`./local:/path` ở cấp service), cách này quản lý tập trung ở top-level và tái dùng cho nhiều service.

`driver_opts`	**Ý nghĩa**
`type`	`none` cho bind, `nfs` cho NFS
`o`	Tùy chọn mount (vd `bind`, hoặc chuỗi option NFS)
`device`	Đường dẫn host hoặc địa chỉ NFS export

Volume NFS — lưu trên máy chủ mạng

```yaml
volumes:
  nfs_vol:
    driver: local
    driver_opts:
      type: nfs
      o: "addr=10.0.0.10,rw,nfsvers=4"
      device: ":/exported/path"
```

Cho phép volume nằm trên một **NFS server** trong mạng, thay vì đĩa local. Hữu ích khi nhiều host cần chung dữ liệu, hoặc muốn lưu trên NAS.

`external` — dùng volume có sẵn

```yaml
volumes:
  legacy_data:
    external: true
    name: existing-volume
```

`external: true` nói "volume này đã tồn tại, đừng tạo mới". Compose chỉ gắn vào. Quan trọng để **bảo vệ dữ liệu**: volume external *không bị xóa* khi `docker compose down -v`. Dùng cho dữ liệu quý do người khác/quy trình khác tạo.

> **Bẫy mất dữ liệu:** `docker compose down -v` xóa *mọi* volume Compose quản lý. Nếu volume DB không phải external và ông lỡ thêm `-v`, dữ liệu bay. Volume production quan trọng nên cân nhắc để `external: true`.

Vòng đời volume — khác hẳn network

Hành động	**Ảnh hưởng volume**
`docker compose up`	Tạo volume nếu chưa có

Hành động	Ảnh hưởng volume
`docker compose down`	**Giữ** volume (dữ liệu an toàn)
`docker compose down -v`	**Xóa** volume Compose quản lý (mất dữ liệu!)
`docker volume ls`	Liệt kê volume
`docker volume rm <tên>`	Xóa thủ công
`docker volume prune`	Dọn các volume không ai dùng

Điểm khác network: `down` thường *không* đụng volume — Docker cố tình thận trọng với dữ liệu. Phải chủ động `-v` mới xóa.

Backup volume — mẹo thực tế

Volume `local` nằm ở `/var/lib/docker/volumes/`. Cách backup gọn là chạy một container tạm mount cả volume lẫn thư mục host rồi `tar`:

```
docker run --rm \
  -v my-stack_db_data:/data \
  -v $(pwd):/backup \
  alpine tar czf /backup/db_backup.tar.gz -C /data .
```

Đây là pattern chuẩn để sao lưu dữ liệu trong volume ra file nén trên host.

09

Configs & Secrets — Cấu hình và Bí mật (top-level)

Hai top-level này giải cùng một bài toán — "đưa dữ liệu vào container dưới dạng file" — nhưng tách theo độ nhạy cảm: `configs` cho dữ liệu *thường*, `secrets` cho dữ liệu *bí mật*.

`configs` (top-level) — định nghĩa file cấu hình

```
configs:
  app_config:
    file: ./config/app.conf        # nội dung lấy từ file trên host

  inline_config:
    content: |                     # nội dung viết thẳng trong compose
      server.port=8080
      server.host=0.0.0.0

  external_config:
    external: true                 # đã tồn tại sẵn (Swarm), đừng tạo
    name: prod_app_config
```

`configs` khai báo các mẩu cấu hình để service bơm vào dưới dạng file (qua key `configs` cấp service đã học ở chương 5). Ba nguồn nội dung:

Nguồn	Cú pháp	Khi nào
`file`	`file: ./app.conf`	Cấu hình nằm sẵn trong repo
`content`	`content: | ...`	Cấu hình ngắn, muốn gói gọn trong compose
`external`	`external: true`	Config đã nạp sẵn vào Swarm/Docker

> `config` **vs bind mount file:** đều cho file vào container. Khác biệt: config được Docker *quản lý như đối tượng* — version được, phân phối được qua nhiều node trong Swarm, và `content` cho phép nhúng thẳng không cần file rời. Bind mount thì đơn giản, trực tiếp, hợp dev.

`secrets` (top-level) — định nghĩa bí mật

```
secrets:
  db_password:
    file: ./secrets/db_password.txt    # đọc từ file

  api_key:
    environment: API_KEY               # đọc từ biến môi trường host

  external_secret:
    external: true                     # secret đã có trong Swarm
    name: prod_db_password
```

`secrets` giống `configs` về cơ chế (bơm thành file) nhưng dành cho dữ liệu nhạy cảm và được mount vào `/run/secrets/<tên>` — một vùng tmpfs (RAM), quyền hạn chế.

Nguồn	Cú pháp	Ghi chú
`file`	`file: ./pw.txt`	Nguồn phổ biến nhất; file *đừng commit lên git*
`environment`	`environment: API_KEY`	Lấy từ env host — tiện cho CI bơm secret qua biến
`external`	`external: true`	Secret quản lý bởi Swarm/orchestrator

Cơ chế triết lý: đọc bí mật từ FILE

Mô hình secret của Docker dựa trên một nguyên tắc: **app đọc bí mật từ file, không từ biến môi trường**. Lý do:

Rủi ro của `environment`	`secrets` (file) khắc phục
Lộ qua `docker inspect`	File không hiện trong inspect
Lộ qua log/crash dump (in cả env)	File không bị in cùng env
Tiến trình con thừa kế hết env	File chỉ ai đọc đúng đường dẫn mới thấy
Nằm trong layer image (nếu qua ARG)	Mount lúc chạy, không vào image

Nhiều image chính thức hỗ trợ hậu tố `_FILE` đúng vì mô hình này:

```
services:
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password   # trỏ tới FILE
    secrets:
      - db_password

secrets:
  db_password:
    file: ./secrets/db_password.txt
```

Postgres thấy biến `..._FILE`, tự đọc nội dung file đó làm mật khẩu — mật khẩu thật không bao giờ nằm trong biến môi trường.

Khác biệt thật giữa chế độ thường và Swarm

Cần nói thẳng để khỏi ảo tưởng bảo mật:

- Trong **`docker compose` thường (1 host)**, secret thực chất được mount như một file bind/tmpfs. Nó *gọn gàng và tách biệt* hơn env, nhưng không có mã hóa khi lưu (encryption at rest).
- Trong **Docker Swarm**, secret được mã hóa trong Raft store, phân phối an toàn tới node, chỉ giải mã trong RAM của container cần. Đây mới là bảo mật "thật".

Dù vậy, ngay cả ở chế độ thường, dùng `secrets` vẫn **tốt hơn** nhét mật khẩu vào `environment` vì tránh được các đường rò qua inspect/log.

So sánh nhanh configs vs secrets

	configs	secrets
Mục đích	Dữ liệu cấu hình thường	Mật khẩu, token, key, chứng chỉ
Mount tại	Đường dẫn ông chỉ định	`/run/secrets/<tên>` (mặc định)
Lưu ở đâu	Đĩa thường	tmpfs (RAM), quyền chặt

	configs	**secrets**
Nguồn `content` inline	Có	Không (chỉ file/env/external)
Mã hóa (Swarm)	Không	Có

Quy tắc vàng về secret

> **Không bao giờ commit file secret lên git.** Thêm `secrets/` và `*.env` vào `.gitignore`. Đây cũng là điều development-rules ông đang theo nhấn mạnh: không đẩy thông tin nhạy cảm (dotenv, API key, credential) lên repo.

PHẦN

Phần IV — AI & Vận hành

10

Models & AI — Docker Model Runner, Provider, Agents

Đây là phần *mới* khiến ông thắc mắc ban đầu: "giờ có thêm agents, model nữa hả?". Chương này tách bạch cái nào là **top-level object thật** (`models`), cái nào là **tooling xung quanh** (agents, cagent, MCP).

Bối cảnh: Docker Model Runner

Docker những năm gần đây đẩy mạnh AI. Sản phẩm lõi là **Docker Model Runner** — một thành phần chạy LLM *cục bộ*, phơi ra **endpoint tương thích OpenAI**. App của ông trỏ vào endpoint đó như gọi OpenAI, nhưng model chạy trên máy ông, miễn phí, riêng tư.

Khái niệm	Giải thích
Docker Model Runner	Bộ máy chạy LLM local, expose API OpenAI-compatible. Trên Mac Apple Silicon dùng GPU qua Metal
OCI model artifact	Model được đóng gói & phân phối như image, kéo từ Docker Hub namespace `ai/` (vd `ai/smollm2`, `ai/llama3.2`)
`docker model`	Lệnh CLI mới: `docker model pull/run/ls/rm` — y hệt thao tác với image nhưng cho model

Lệnh CLI cơ bản:

```
docker model pull ai/smollm2     # kéo model về
docker model run ai/smollm2      # chạy thử tương tác
docker model ls                  # liệt kê model đã có
docker model rm ai/smollm2       # xóa
```

`models` (top-level) — object AI mới

Đây chính là top-level object mới, *ngang hàng* `services` / `volumes` / `networks`.

```
models:
  smollm:
    model: ai/smollm2            # bắt buộc: tên OCI artifact
    context_size: 8192           # cửa sổ ngữ cảnh (số token)
    runtime_flags:               # cờ truyền cho engine suy luận
      - "--no-prefill-assistant"

  llama:
    model: ai/llama3.2:1B-Q4_0   # pin tag + mức lượng tử hóa cụ thể
```

Thuộc tính	Ý nghĩa
`model`	(Bắt buộc) Tham chiếu OCI artifact của model, dạng `ai/<tên>[:tag]`
`context_size`	Kích thước context window — bao nhiêu token model "nhớ" được mỗi lần
`runtime_flags`	Cờ dòng lệnh truyền thẳng cho engine inference (llama.cpp...)

> **Lượng tử hóa (quantization):** tag kiểu `1B-Q4_0` nghĩa là model 1 tỉ tham số, nén xuống 4-bit. Càng nén (Q4 < Q8) thì nhẹ & nhanh hơn nhưng giảm chút chất lượng. Chọn theo RAM/CPU máy ông.

Service tiêu thụ model — 2 cách tham chiếu

Service thường (app chat) khai nó *dùng* model nào qua key `models` cấp service:

Cách 1 — short (chỉ liệt kê tên):

```yaml
services:
  chat-app:
    image: my-chat
    models:
      - smollm
```

Cách 2 — long (Compose tự inject biến môi trường):

```yaml
services:
  chat-app:
    image: my-chat
    models:
      smollm:
        endpoint_var: OPENAI_BASE_URL   # tên biến chứa URL endpoint
        model_var: OPENAI_MODEL         # tên biến chứa tên model
```

Cách 2 mạnh hơn: Compose tự đặt vào container hai biến môi trường — một chứa **URL endpoint** (dạng `http://...//engines/v1`, OpenAI-compatible), một chứa **tên model** để gọi API. App chỉ việc đọc biến và gọi như gọi OpenAI.

Con key	Ý nghĩa
`endpoint_var`	Tên biến môi trường Compose sẽ điền URL của Model Runner vào
`model_var`	Tên biến môi trường Compose sẽ điền tên model vào

`provider` — cách KHÁC khai model trong services

```yaml
services:
  model-runner:
    provider:
      type: model
      options:
        model: ai/smollm2
```

`provider` là một *kiểu service đặc biệt*: nó không phải container thường mà là điểm ủy quyền cho một capability bên ngoài. Với `type: model`, Compose giao việc cho Docker Model Runner. Service khác có thể `depends_on` nó.

Con key	Ý nghĩa
`type`	Loại provider — `model` cho Model Runner
`options`	Tùy chọn cho provider; với model: `model`, `context-size`, `runtime-flags`...

> `models` **(top-level) vs** `provider` **(service):** cùng kích hoạt Model Runner nhưng góc nhìn khác. `models` là *tài nguyên* tách riêng, nhiều service share được — gọn khi nhiều app dùng chung model. `provider` đặt

> model như *một service* trong đồ thị phụ thuộc — tiện khi muốn `depends_on` rõ ràng. Chọn 1 trong 2 tùy phong cách.

"Agents" — làm rõ: KHÔNG phải top-level object

Đây là điểm dễ hiểu lầm. Khác với `models`, **không có** top-level `agents:` ngang hàng `services`. "Agent" trong hệ Docker AI là:

Thành phần	Bản chất
Agent (khái niệm)	Thường là một *service app* (như `chat-app` ở trên) = code + model + công cụ (tools). Nó là service bình thường, không phải từ khóa Compose
`cagent`	Công cụ *riêng* của Docker để build/chạy agent. Một file YAML khai báo agent (model, hướng dẫn, tools/MCP) chạy bằng **binary** `cagent` **độc lập** — nằm NGOÀI compose
MCP (Model Context Protocol)	Giao thức chuẩn để agent gọi "tools" (web search, đọc file, gọi API). Docker có **MCP Toolkit / MCP Gateway** đóng gói tool dưới dạng container

Vậy nên khi nghe "Docker có agents", hiểu đúng là: Docker cung cấp *hệ sinh thái công cụ* (cagent + MCP Toolkit + Model Runner) để xây agent — chứ không phải thêm một block `agents:` vào `compose.yaml`.

`cagent` — phác qua (ngoài phạm vi Compose)

Để hình dung, file `cagent` trông như sau (KHÔNG phải compose.yaml):

```yaml
# agent.yaml — chạy bằng:  cagent run agent.yaml
agents:
  root:
    model: openai/gpt-4o          # hoặc model local qua Model Runner
    instruction: "Bạn là trợ lý DevOps."
    toolsets:
      - type: mcp                 # nối tới MCP server lấy tools
        ref: docker:duckduckgo
```

Điểm cần nhớ: cú pháp này thuộc *cagent*, không trộn vào `compose.yaml`. Nếu ông chỉ cần *chạy app dùng LLM*, thì `models` trong compose là đủ; còn muốn *xây con agent tự ra quyết định + gọi tool*, mới cần tới cagent.

Bản đồ nhanh hệ AI của Docker

Muốn làm	Dùng
Chạy LLM local cho app	`models` (top-level) + Model Runner
Khai model như service có phụ thuộc	`provider: {type: model}`
Kéo/chạy/xóa model bằng tay	`docker model ...`
Xây agent tự hành (tools, reasoning)	`cagent` (binary riêng)
Cấp tool cho agent (search, API...)	MCP Toolkit / MCP Gateway

11

CLI — Lệnh & Workflow thực tế

Biết cú pháp file chưa đủ; phải biết *lái* nó. Chương này gom các lệnh `docker compose` hay dùng và mạch làm việc thực tế.

Lệnh vòng đời cốt lõi

```
docker compose up -d            # dựng & chạy nền (detached)
docker compose down             # dừng & xóa container + network
docker compose down -v          # ... XÓA luôn volume (mất dữ liệu!)
docker compose restart          # khởi động lại các service
docker compose stop             # dừng nhưng giữ container
docker compose start            # chạy lại container đã stop
```

Lệnh	Tác dụng	Lưu ý
`up`	Tạo + chạy. `-d` chạy nền, `--build` build trước, `--watch` bật hot-reload	Lệnh dùng nhiều nhất
`down`	Hạ stack. Mặc định giữ volume	Thêm `-v` mới xóa volume
`stop` / `start`	Tạm dừng/chạy lại, *không* xóa container	Khác `down` ở chỗ giữ container
`restart`	stop rồi start	Không nạp lại file YAML đã đổi

> `restart` **không đọc lại file YAML.** Nếu ông sửa `compose.yaml` rồi `restart`, thay đổi *không* áp dụng. Phải `up -d` lại (Compose tự tạo lại container nào đổi cấu hình).

Build và image

```
docker compose build            # build mọi service có build:
docker compose build --no-cache # build sạch, bỏ cache
docker compose up -d --build    # build rồi up một phát
docker compose pull             # kéo image mới nhất từ registry
docker compose push             # đẩy image đã build lên registry
```

Quan sát và gỡ lỗi

```
docker compose ps               # container đang chạy + trạng thái
docker compose ps -a            # cả container đã dừng
docker compose logs -f          # log realtime toàn stack
docker compose logs -f web      # log riêng service web
docker compose logs --tail=100 web
docker compose top              # tiến trình trong các container
docker compose events           # luồng sự kiện realtime
```

Lệnh	Khi nào dùng
`ps`	Kiểm tra service nào chạy/chết/healthy
`logs -f`	Theo dõi log để debug. `-f` = follow (realtime)

Lệnh	Khi nào dùng
`config`	Validate + xem file đã merge — *luôn dùng khi nghi cú pháp*
`top`	Xem tiến trình bên trong container

`exec` và `run` — chạy lệnh trong container

```
docker compose exec web sh        # mở shell trong container web ĐANG chạy
docker compose exec db psql -U app appdb   # gõ psql thẳng vào db
docker compose run --rm web npm test       # tạo container MỚI tạm để chạy 1 lệnh
```

Lệnh	Khác biệt
`exec`	Chạy lệnh trong container *đang chạy sẵn*. Dùng để vào shell, soi, thao tác nóng
`run`	Tạo container *mới* (one-off) để chạy một tác vụ rồi xóa (`--rm`). Hợp migration, test, script

`config` — vũ khí debug số một

```
docker compose config             # in file đã gộp & validate
docker compose config --services  # liệt kê tên service
docker compose config --volumes   # liệt kê volume
```

`config` parse toàn bộ (gộp override, include, nội suy biến, mở anchor) rồi in ra YAML cuối cùng. Khi "sao nó không nhận biến của tôi?", "anchor merge đúng chưa?", "include vào chưa?" — `config` trả lời ngay. Nếu file sai cú pháp, nó báo lỗi tại đây thay vì lúc `up`.

Profiles — bật nhóm service

```
docker compose --profile dev up -d          # bật profile "dev"
docker compose --profile dev --profile debug up
COMPOSE_PROFILES=dev,debug docker compose up # qua biến môi trường
```

Service không gắn `profiles` luôn chạy. Service gắn profile chỉ chạy khi profile đó được bật. Cách gom các service phụ trợ (seeder, monitoring, debug) tách khỏi luồng chính.

Watch — vòng lặp dev hiện đại

```
docker compose up --watch         # up + theo dõi file (cần khai develop.watch)
docker compose watch              # chỉ chạy chế độ watch
```

Kết hợp với khối `develop.watch` (chương 6): sửa code → tự `sync` / `rebuild` / `restart`. Gần như live-reload mà không rời Docker.

Scale — nhân bản service

```
docker compose up -d --scale api=3    # chạy 3 bản api
```

`--scale` nhân số container của một service. *Điều kiện*: service đó **không** được đặt `container_name` (tên cố định thì không nhân được), và `ports` nên để Docker tự cấp cổng host (hoặc dùng reverse proxy phía trước).

Mạch làm việc điển hình

Một vòng phát triển thường gặp:

```
# 1. Kiểm tra cú pháp trước
docker compose config

# 2. Dựng stack, build nếu cần
docker compose up -d --build

# 3. Theo dõi log xem có lỗi khởi động
docker compose logs -f

# 4. Vào container thao tác/migration
docker compose exec api npm run migrate

# 5. Sửa code với hot-reload
docker compose up --watch

# 6. Xong việc, hạ stack (giữ dữ liệu)
docker compose down
```

Nhiều file — tách dev/prod

```
docker compose -f compose.yaml -f compose.prod.yaml up -d
```

Compose gộp các file `-f` theo thứ tự, file sau đè file trước. Pattern phổ biến: `compose.yaml` (cấu hình chung) + `compose.override.yaml` (tự nạp, cho dev) + `compose.prod.yaml` (chỉ định danh khi deploy). Nhờ vậy một bộ định nghĩa phục vụ nhiều môi trường.

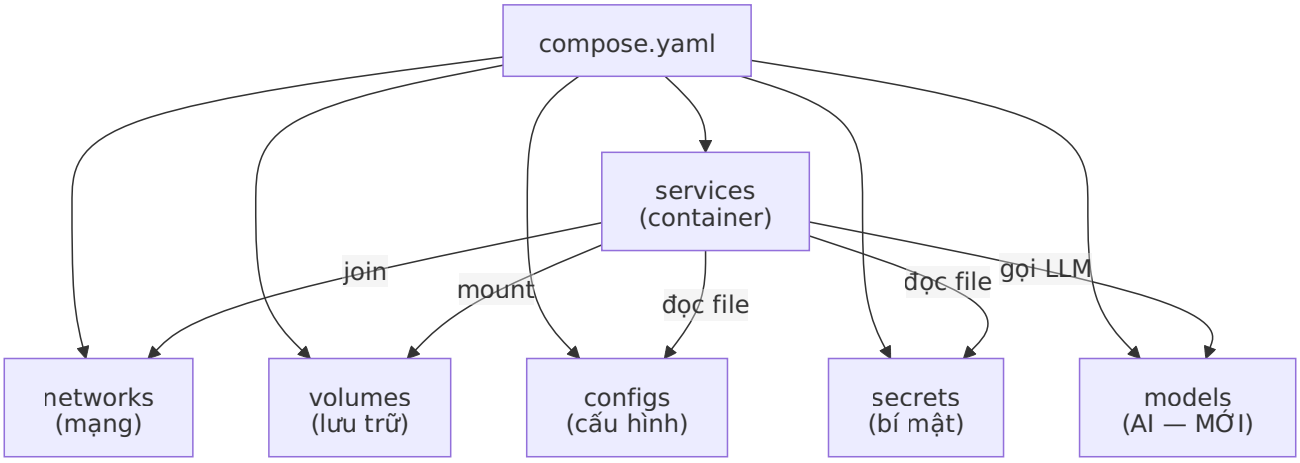
File	Vai trò
`compose.yaml`	Định nghĩa nền tảng, chung mọi môi trường
`compose.override.yaml`	Tự động merge — đặt cấu hình *dev* ở đây
`compose.prod.yaml`	Cấu hình production, chỉ nạp khi `-f` rõ ràng

12

Cheatsheet & Tổng kết

Chương tra cứu nhanh: bản đồ toàn cảnh, bảng "key này thuộc đâu", các bẫy hay gặp, và mini-template để copy.

Sơ đồ quan hệ các top-level object



Ý tưởng cốt lõi: các top-level *định nghĩa tài nguyên*; `services` *tiêu thụ* chúng bằng cách tham chiếu tên.

Bảng "thuộc tính này nằm ở đâu"

Thuộc tính	Cấp service	Cấp top-level	Ghi chú
`image` / `build`	✅	–	Nguồn image
`ports` / `expose`	✅	–	Mở cổng
`environment` / `env_file`	✅	–	Biến môi trường
`volumes`	✅ (mount)	✅ (định nghĩa)	Hai cấp khác nhau!
`networks`	✅ (join)	✅ (định nghĩa)	Hai cấp khác nhau!
`configs`	✅ (dùng)	✅ (định nghĩa)	Hai cấp khác nhau!
`secrets`	✅ (dùng)	✅ (định nghĩa)	Hai cấp khác nhau!
`models`	✅ (dùng)	✅ (định nghĩa)	Mới — hai cấp
`depends_on` / `healthcheck`	✅	–	Thứ tự & sức khỏe
`deploy`	✅	–	Chủ yếu Swarm
`develop.watch`	✅	–	Hot-reload

> **Quy luật chung:** `volumes`, `networks`, `configs`, `secrets`, `models` đều xuất hiện *hai lần* — một lần ở top-level để *định nghĩa*, một lần trong service để *dùng*. Đừng nhầm hai vai này.

Bảng "cũ vs mới"

Object	Có từ	Bản chất
`services`	Luôn có	Container — lõi
`networks`	Cũ	Mạng ảo
`volumes`	Cũ	Lưu trữ bền
`configs`	Cũ (phổ biến qua Swarm)	File cấu hình
`secrets`	Cũ	Dữ liệu nhạy cảm
`models`	**Mới (kỷ nguyên AI)**	LLM qua Model Runner
`provider`	Mới (cấp service)	Ủy quyền capability (vd model)
agents / cagent	Mới — *ngoài* compose	Tooling agent riêng, không phải top-level

Top 10 bẫy người mới

#	Bẫy	Cách tránh
1	Còn viết `version:`	Bỏ đi, đã obsolete
2	Dùng **tab** thụt lề YAML	Chỉ dùng dấu cách
3	Đảo ngược `ports` (host:container)	Trái = host, phải = container
4	Tưởng `depends_on` đợi "sẵn sàng"	Cần `condition: service_healthy` + healthcheck
5	Mất dữ liệu vì `down -v`	Volume quý để `external: true`
6	Quên `max-size` logging	Luôn giới hạn kẻo đầy đĩa
7	Nhét mật khẩu vào `environment`	Dùng `secrets` + `_FILE`
8	`restart` không nạp lại YAML	Sửa file thì `up -d` lại
9	Đặt `container_name` rồi muốn scale	Bỏ tên cố định mới `--scale` được
10	Commit **file secret/.env** lên git	Cho vào `.gitignore`

Mini-template chạy được ngay

```yaml
name: demo

services:
  web:
    build: ./web
    ports:
      - "8080:80"
    depends_on:
      api:
        condition: service_healthy
    networks: [frontend, backend]
    restart: unless-stopped

  api:
    image: my-api:latest
    environment:
      DB_HOST: db
    secrets:
      - db_password
    networks: [backend]
    healthcheck:
```

```
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 10s
      retries: 3

  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    volumes:
      - db_data:/var/lib/postgresql/data
    secrets:
      - db_password
    networks: [backend]

networks:
  frontend:
  backend:
    internal: true        # db tách khỏi internet

volumes:
  db_data:

secrets:
  db_password:
    file: ./secrets/db_password.txt
```

Stack này gói gần hết bài học: build, ports, depends_on có điều kiện, healthcheck, hai mạng (một mạng nội bộ kín), volume bền, và mật khẩu qua secret + `_FILE`.

Lộ trình ôn tập gợi ý

1. **Tuần 1** — Dựng được stack 2 service (web + db) với volume + network mặc định. Thuộc `up/down/ps/logs`.
2. **Tuần 2** — Thêm `healthcheck` + `depends_on: service_healthy`, `env_file`, `.env` nội suy biến.
3. **Tuần 3** — `secrets` + `_FILE`, mạng `internal`, giới hạn tài nguyên, `profiles`.
4. **Tuần 4** — `develop.watch` cho dev, nhiều file `-f` cho dev/prod, thử `models` chạy LLM local.

Câu hỏi còn mở (tự kiểm tra)

- Khác nhau thực chất giữa `secrets` ở chế độ Compose thường vs Swarm là gì? (Gợi ý: mã hóa at-rest)
- Khi nào chọn `models` top-level, khi nào chọn `provider`?
- `extends` và YAML anchor — cái nào merge "thông minh" hơn, vì sao?

Nắm được ba câu này tức là ông đã đi quá mức "biết cú pháp" sang "hiểu vì sao". Chúc ôn tập tốt! 🐳

Index