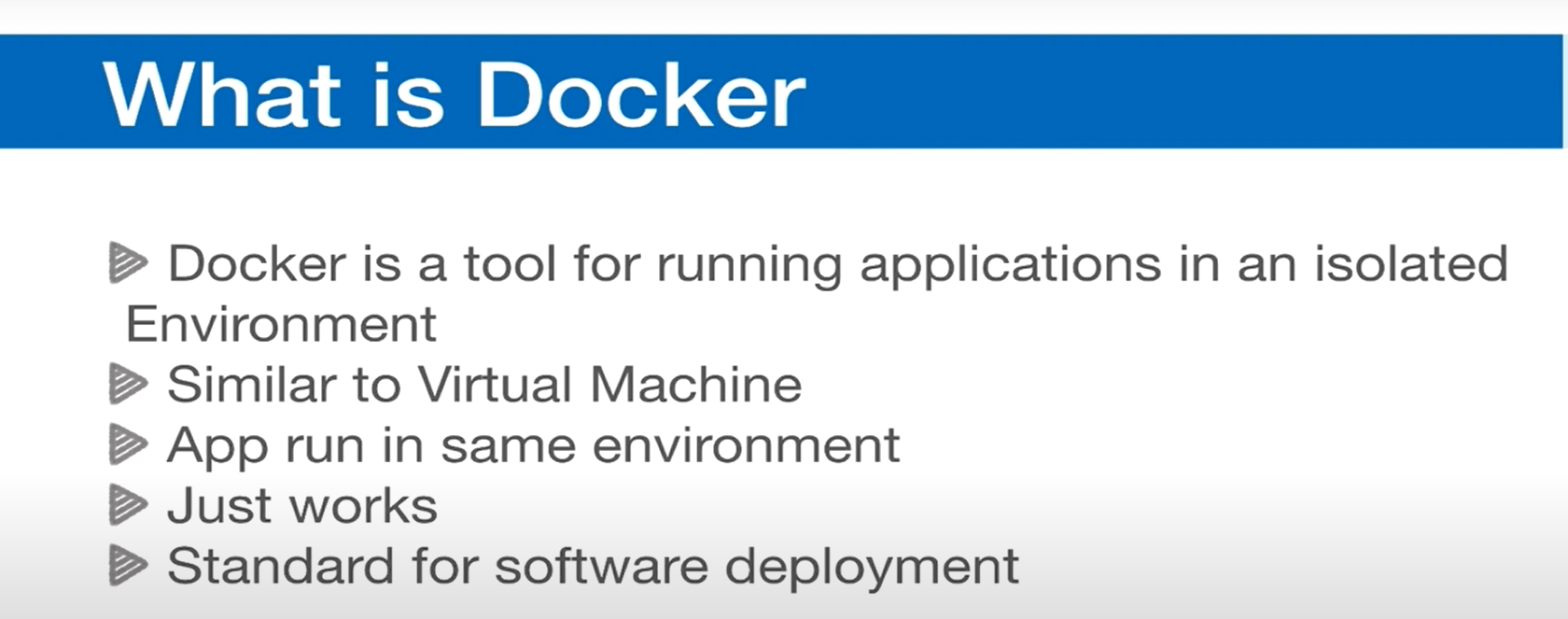
**Docker Interview Questions**

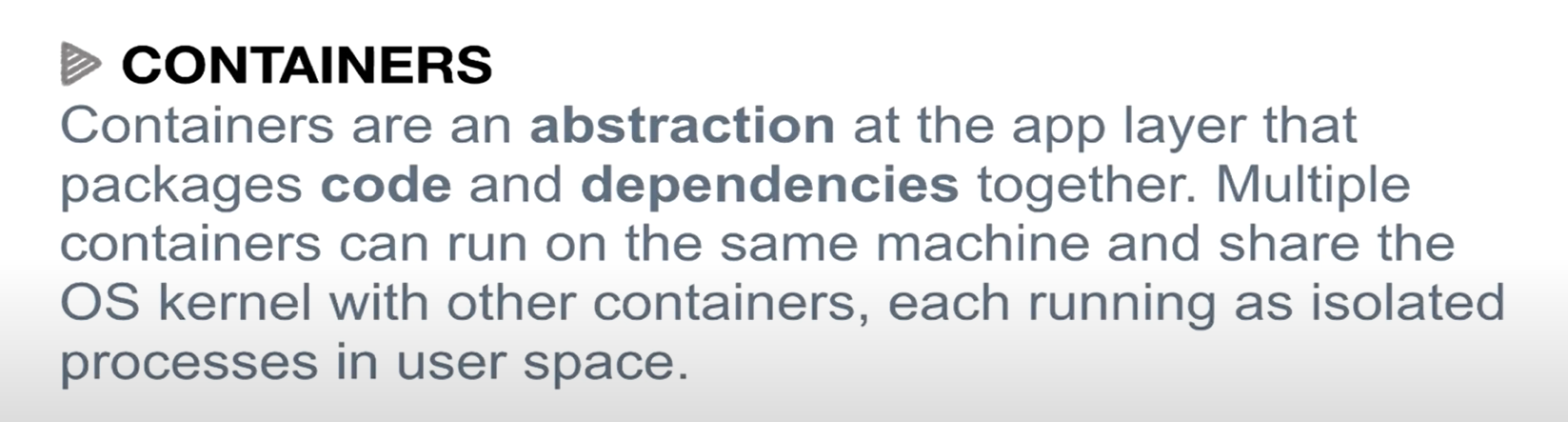
### What is Docker

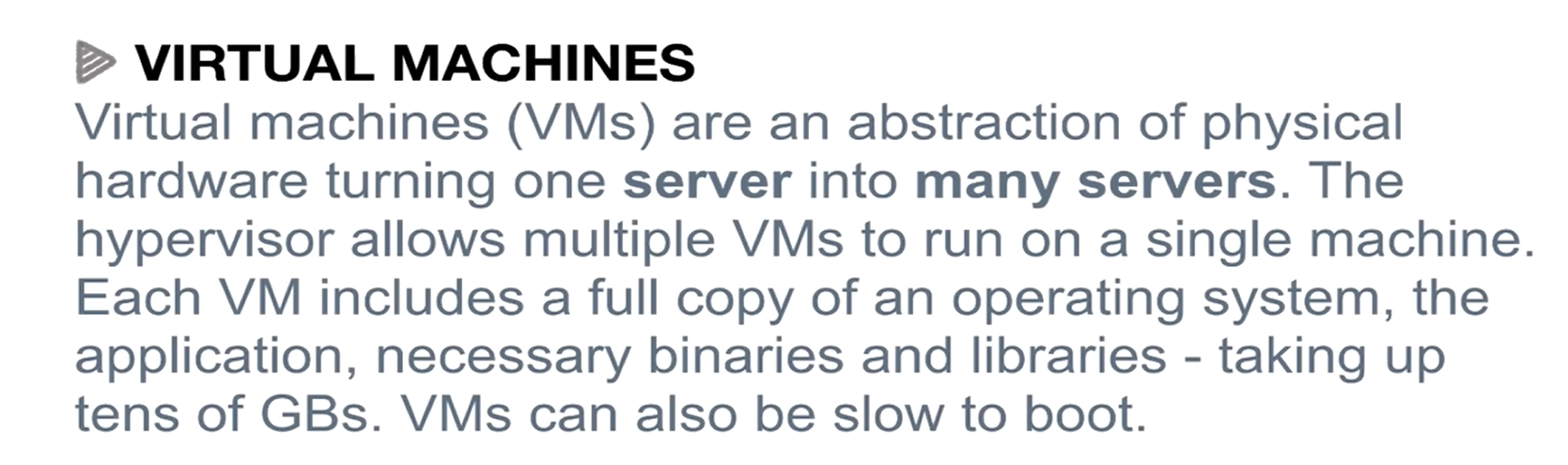
**Answer.** Docker packages applications into lightweight containers that include everything needed to run - code, libraries, and settings. Unlike bulky virtual machines (VMs) that require full operating systems, Docker containers share the host OS kernel, making them fast and portable.

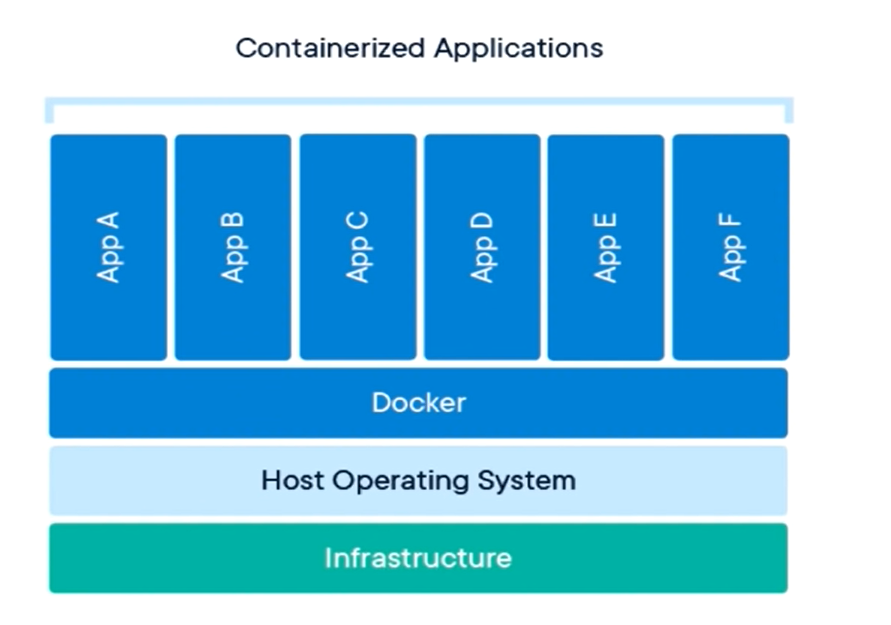
**Key Difference:**

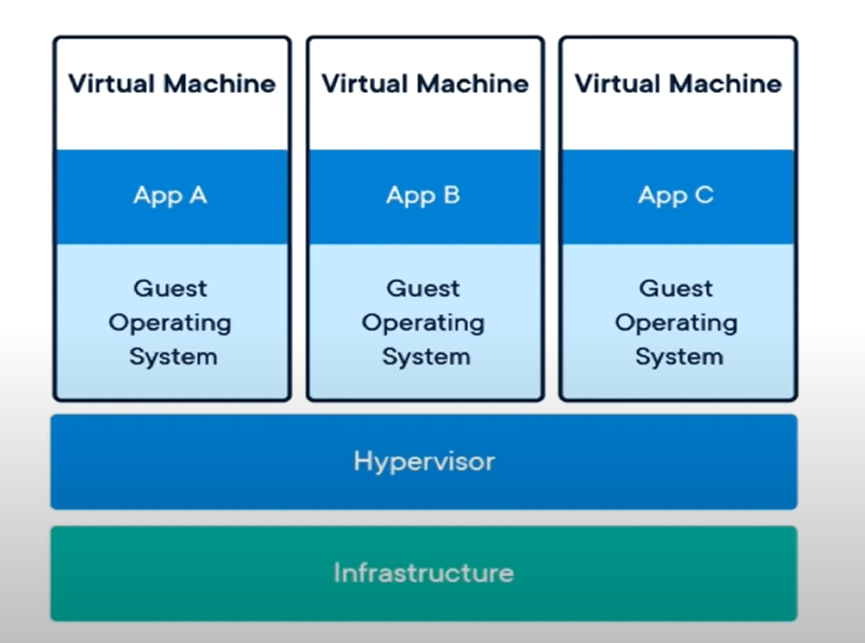
VMs are like shipping entire houses (OS + apps), while Docker ships pre-furnished rooms (just your app + dependencies) that share the building's foundation (host OS).

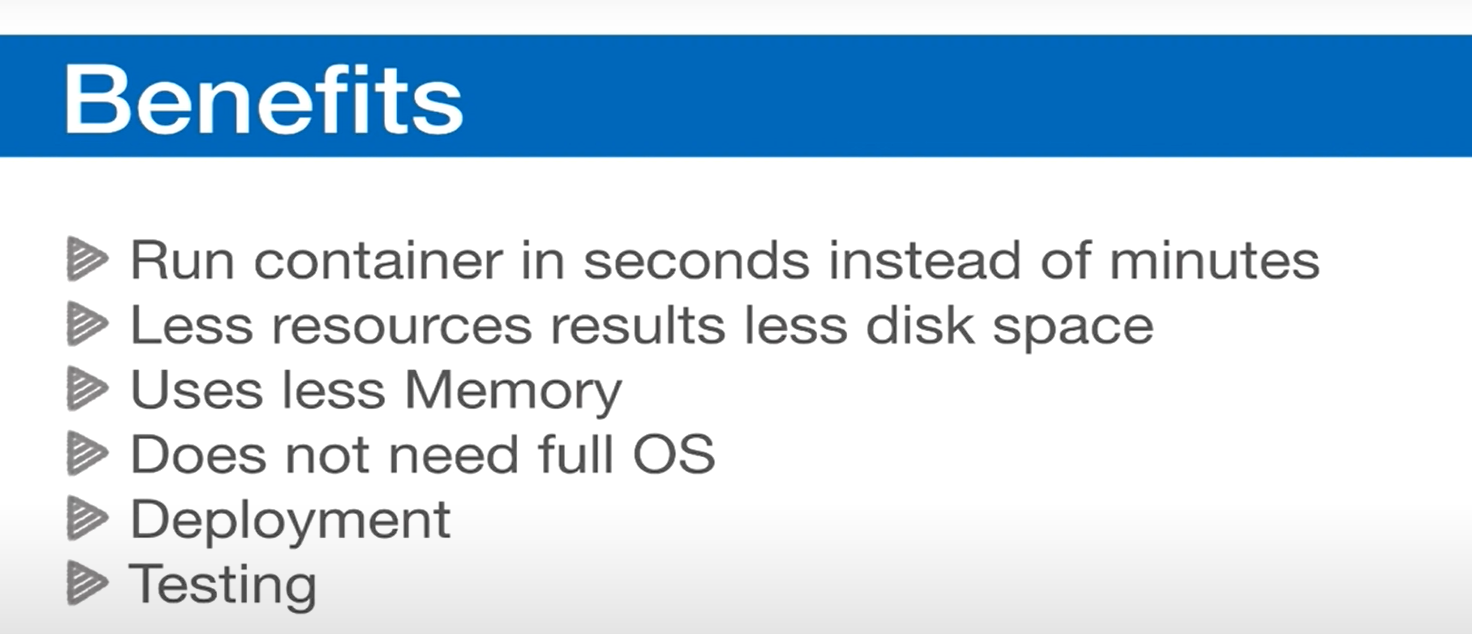








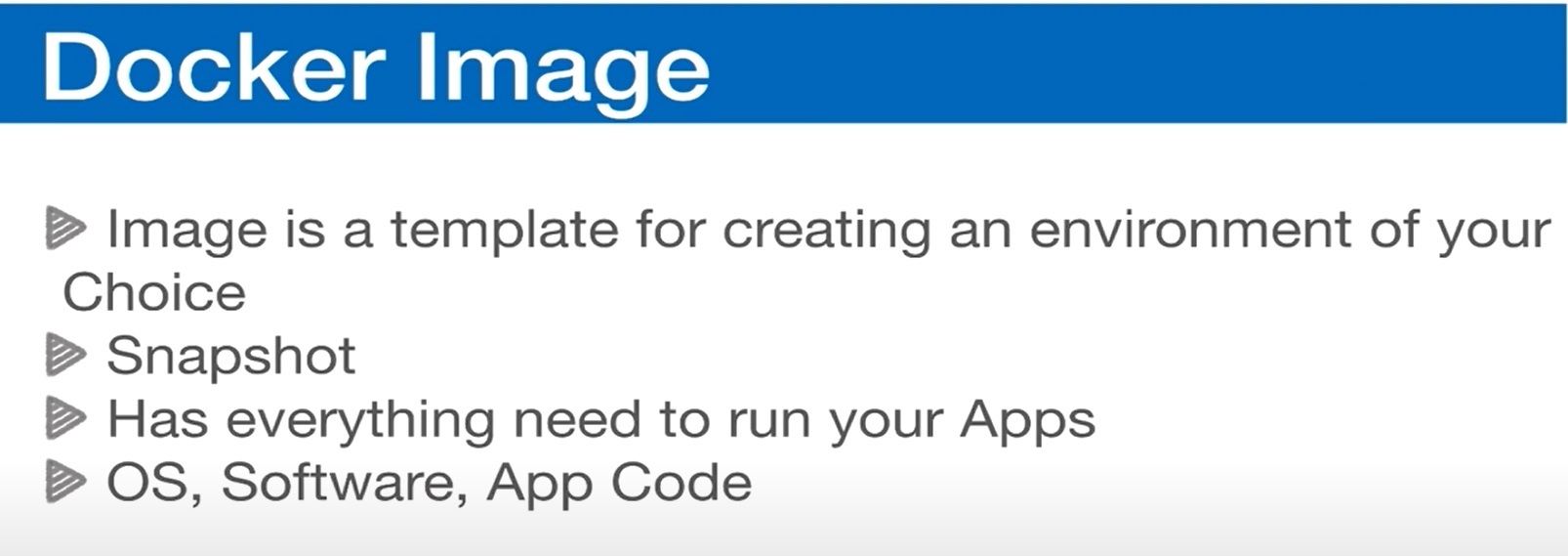


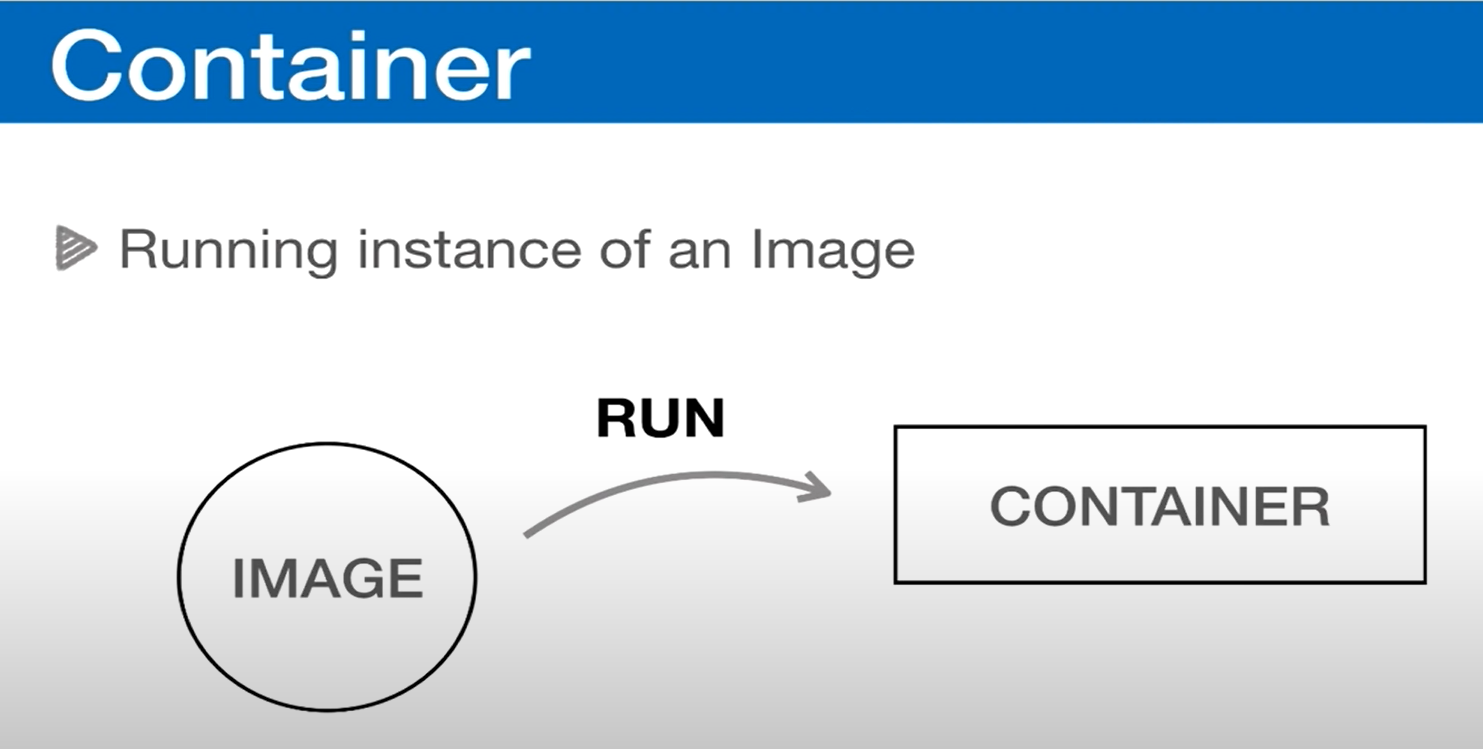


### 

### Docker Image

**Answer.** Docker images are static templates containing application code and dependencies, while containers are live, isolated instances of those images. Like how a cookie recipe (image) produces actual cookies (containers) when baked.





### 

### Docker Image & Containers

Docker is very important in today's time because as we move towards microservices architecture,

we create containers using Docker, inside these containers we do all our configuration, for example, what will be its operating system, what tools will be there in it, tools means

let us run Ubuntu in my Mac machine (docker run -it ubuntu)

Ubuntu this is the name of our image

what is the image so I will tell you but for now you understand that this is the name of our operating system so I am telling Docker that Docker run a container for me inside which what do I need I need Ubuntu image, Ubuntu operating system

as soon as I enter this here we will get an error the error says that it is not able to find image um it says that I do not have any operating system um so it is downloading it first right and don't worry these images are very light weight right so what did it do it downloaded it for me

Let us say that I have a MacBook, okay, which operating system is running in it, then you will say that Mac OS is running in it, right or for now let's take an example that Darwin OS is running in it, Correctly similarly, our images, right, our images are a kind of operating system. You need containers right containers can't do anything containers are empty they are just like machines so what can you do you take multiple containers like this is one container this is another container this is another container these containers can't do anything you have to do anything over these containers

### 

### Deleting Docker Volumes

* When you run docker ps, you see only active containers.
* docker volume ls lists **all volumes** (active + unused).
* Over time, as you **create and delete containers**, volumes can build up (especially anonymous volumes).
* Anonymous volumes are automatically created (e.g., -v /app/node\_modules) and **not deleted** when the container is removed.
* This can lead to **hundreds of unused volumes** cluttering your system.
* Volumes are designed to **persist data** (e.g., Postgres, MySQL).
  + In such cases, you **do not delete volumes**, since they store important data.
* But for anonymous volumes (like node\_modules), persisting them isn’t necessary.
* By default, docker rm <container> **does not delete volumes** attached to that container.
* To prevent buildup, use the -v option when removing containers.

When you run docker ps, it only shows the containers that are currently running. If you run docker volume ls, it lists all the volumes that exist on your system. Over time, as you create and delete containers, these volumes keep piling up, especially when anonymous volumes are created.

An **anonymous volume** is one that Docker creates automatically, for example when you mount something like /app/node\_modules. Even if you delete the container, the anonymous volume is not deleted. This can eventually lead to hundreds of unused volumes on your system.

Normally, volumes are meant to store **persistent data**. For example, if you have a Postgres or MySQL database, the data should survive even if the container is deleted. That’s why Docker does not delete volumes by default when you remove a container.

However, in cases like node\_modules, we don’t need the data to persist. To clean up these extra volumes you have a few options:

* Run docker volume rm <volume\_name> to delete a specific volume.
* Run docker volume prune to remove all unused volumes at once. (It will ask for confirmation.)
* When deleting a container, use docker rm -v <container\_id>. The -v flag makes sure the volumes attached to that container are also deleted. If you also add -f, like docker rm -fv <container\_id>, it forces the container to stop and removes it along with its volumes.

**Best practice:** For databases or anything that stores important information, never use -v because you want the data to stay. For temporary things like node\_modules, always use -v so your system doesn’t get cluttered with unnecessary volumes.

### 

### Docker Compose

When running containers with a long docker run command, it becomes tedious, especially when working on larger applications that require multiple containers (e.g., Node server, Postgres, Redis, Elasticsearch). Running many long commands is error-prone and inefficient.

**Docker Compose** solves this problem by letting us define everything in a single configuration file (docker-compose.yml) and manage multiple containers with one simple command.

The docker-compose.yml file is written in YAML format, where spacing and indentation are very important. Each container is defined as a **service** under the services section.

Example structure for one service (Node app):

| version: "3"  services:  node-app:  build: .  ports:  - "3000:3000"  volumes:  - ./:/app  - /app/node\_modules  environment:  - PORT=3000 |
| --- |

* version: "3" → Compose file format version.
* services → Defines containers. Each container is a service.
* node-app → Name of the service.
* build: . → Builds an image from the Dockerfile in the current directory.
* ports → Maps port 3000 on the host to port 3000 in the container.
* volumes →
  + ./:/app is a bind mount to sync local code with the container.
  + /app/node\_modules is an anonymous volume to prevent overwriting dependencies.
* environment → Passes environment variables like PORT=3000. You can also load from a .env file using env\_file.

### Important Commands (with explanations)

* **Start all services:** 👉 docker-compose up -d (the -d flag runs containers in background)
* **Stop and remove containers:** 👉 docker-compose down (removes containers but not volumes)  
   👉 docker-compose down -v (also deletes attached volumes)
* **Rebuild image after changes:** 👉 docker-compose up -d --build (forces a rebuild of the image before starting)
* **Check running containers:** 👉 docker ps
* **Check images built by Compose:** 👉 docker image ls (images will be named like project-folder\_service-name, e.g., node-docker\_node-app)

### Extra Notes

When you run docker-compose up, it not only creates your containers but also sets up a new **network** for them. This is done so that all the containers inside your project can talk to each other easily without interfering with containers from other projects. For example, your Node app can reach Postgres just by using the service name instead of worrying about IP addresses.

Another thing to keep in mind is that the way we set it up right now is for **development**. That’s why we’re using a bind mount (./:/app) so any code changes on your computer instantly reflect inside the container, and we’re running with npm run dev.

But in **production**, you wouldn’t do this. You don’t need to sync files from your computer, and instead of running in development mode, you’d run something like npm start or node index.js to start the app properly.

So the same Docker Compose setup can be adapted to both development and production, you just configure it a little differently depending on the environment.

### 

### Docker Compose Dev to Prod multiple files

In real projects, you usually need different setups for **development** and **production**. The main difference for our Node app is the command we run: in dev we use npm run dev (with nodemon), and in production we’d use npm start or just node index.js. Some people recommend skipping npm start in containers to avoid the extra layer, but it’s personal preference.

We could create two separate Dockerfiles, but a cleaner way is to use **one Dockerfile** and split our **docker-compose** configuration into multiple files.

Here’s how it works:

**1. Base file (docker-compose.yml)**This holds all **shared settings** that are the same across both environments (e.g., build context, ports, common environment variables).

| version: "3" services:  node-app:  build: .  ports:  - "3000:3000"  environment:  - PORT=3000 |
| --- |

2. **Dev file (docker-compose.dev.yml)** Adds dev-specific settings like bind mounts, nodemon, and environment variables.

| services:  node-app:  volumes:  - ./app:/app  - /app/node\_modules  environment:  - NODE\_ENV=development  command: npm run dev |
| --- |

**3. Prod file (docker-compose.prod.yml)** Removes bind mounts (since you don’t want code syncing in production), sets NODE\_ENV=production, and runs Node directly.

| services:  node-app:  environment:  - NODE\_ENV=production  command: node index.js |
| --- |

Now, when you run Docker Compose, you can stack files together:

* **Development:** 👉 docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d
* **Production:** 👉 docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d

The order matters: first the base file, then the override file (dev or prod). The second file can overwrite whatever is defined in the base.

### Handling devDependencies like nodemon

In dev, nodemon is needed, but in prod it’s not. By default, npm install installs everything, including dev dependencies, which wastes space in production.

To fix this, the Dockerfile uses a small script:

| ARG NODE\_ENV RUN if [ "$NODE\_ENV" = "development" ]; then \  npm install; \  else \  npm install --only=production; \  fi |
| --- |

This way:

* In **dev**, it installs everything (including nodemon).
* In **prod**, it installs only production dependencies.

The NODE\_ENV argument is passed from the compose files:

| build:  context: .  args:  NODE\_ENV: development # in dev |
| --- |

| build:  context: .  args:  NODE\_ENV: production # in prod |
| --- |

### Why this matters

* **In development:** you want live reloads (nodemon) and bind mounts so changes instantly reflect in the container.
* **In production:** you want only the necessary files, no dev dependencies, no bind mounts, and the container should run the app directly.

This setup makes it easy to switch between environments with just two commands:  
 👉 Dev: docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d  
 👉 Prod: docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d

And tear down with:  
 👉 docker-compose down -v

### Building images

In the base file, build: . tells Compose to build the image from the Dockerfile in the current folder. If you make changes, you need to rebuild with:  
 docker-compose up -d --build

Compose does not automatically detect code changes in the Dockerfile, so you must force a rebuild.