What is Docker?

Docker is a platform that lets you package an application together with everything it needs (libraries, runtime, dependencies) into a standardized unit called a container.
Think of a Docker container as a lightweight, portable box that contains your app and its entire environment, so it can run the same way on any machine.

What problem does it solve?

Solves -> “It works on my machine” problem
Before Docker, developers ran into a lot of issues when deploying or sharing applications:

For a team working on the same application,
DB -> SQL, its binaries
Messaging -> Redis, its binaries
And so on…

Different machines had:
- different OS versions
- these dependencies may give issues on different OS envs
- also, the app might depend on a specific version of some dependency like a specific node version.

PROBLEM:
It works on some local environments, but may not run on production env.
Also, each time someone setups their local env, these dependencies need to be carefully downloaded and installed.

Docker fixes this by packaging the environment with the app.
- Container = App + Its env dependencies
→ If a container works on one machine, it will work on every machine that can run Docker.

So, if machine A has node v20 and machine B has node v18, our app runs only on v18, in machine A, we can develop the app on a container with node v18.

Docker is also Consistent with CI/CD and microservices
As teams moved toward microservices, apps were harder to maintain and deploy together.
Docker allows each service to:
- run in its own container
- be versioned independently
- be deployed independently

Tools like Docker Compose and Kubernetes automate multi-container systems.

Before Docker

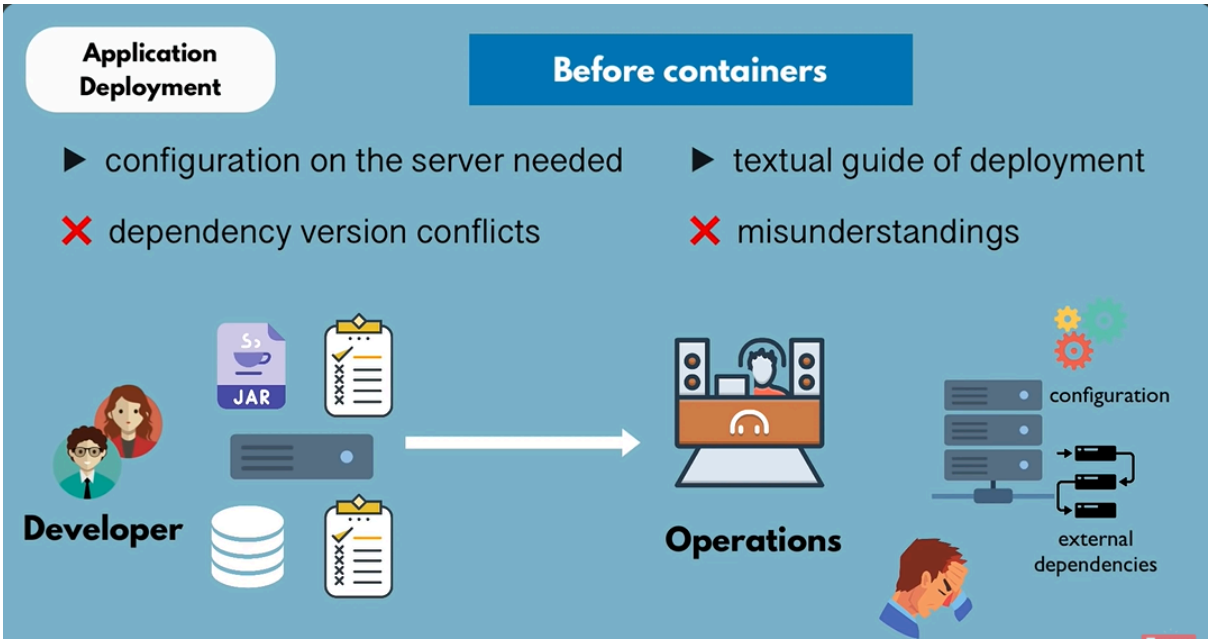
Before Docker, people used virtual machines (VMs) to isolate environments.

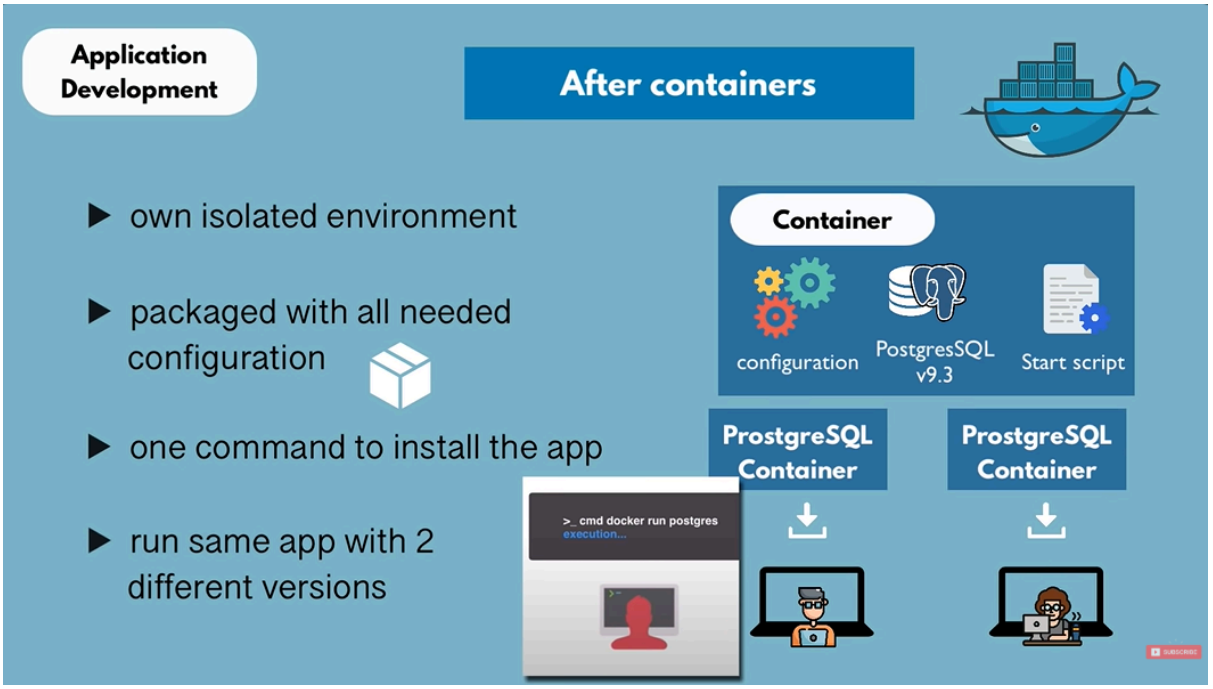
Problems with VMs:

- very heavy (gigabytes) since they need full OS
- slow to start
- Resource-intensive

Docker containers:

- use far less space (megabytes)
- start instantly
- share the host OS kernel → much lighter
- can run dozens of containers easily





Container

A way to package an application with all necessary dependencies and configurations.
Container = Application + Dependencies + Environment.

Where Environment includes:

- environment variables (like we have windows env variables)
- network settings
- isolated process space

It does not include a full OS, but it relies on the host's kernel.
Ex:
If you containerize a Node.js app, the container will include:

- your JavaScript files
- Node runtime
- npm libraries (dependencies)
- environment configs
- startup command (node app.js)
- Everything except the OS kernel.

Properties of Containers:

- Portable (from one machine to another)
- Lightweight (in MBs)

Where are these containers stored?

In the container repositories, just like github for code repos.
Private repos where organisations store their images for internal uses.

Public repos for lots of common images generated - Docker Hub.

What are images and how are they related to containers?

A Docker image is a read-only template/blueprint that contains:

- your application code
- dependencies (libraries, runtimes)
- startup instructions

Think of an image as a blueprint.

- You run the image to create a container, just like:
- a class becomes an object
- A blueprint becomes a house

A container is a running instance of an image.

When you run an image, Docker creates:

- a writable layer
- isolated environment
- running processes

So, Container = Image + Runtime Execution

images are built in layers.
Each line in a Dockerfile creates a new layer.

Example:

```
FROM python:3.10     <-- Base layer
COPY app/ /app       <-- Adds a layer
RUN pip install -r requirements.txt  <-- Another layer
CMD ["python", "main.py"]  <-- Startup instruction
```

These layers stack on top of each other to form the final image.

A container is not made from multiple images.
A container is made from one image that contains multiple layers.

Basic Visualization of image and container

Writable Layer (Container layer)
———————————————————

Image Layer N ex: Your app code (application image is generally on the top)
Image Layer 3 ex: Your app dependencies
Image Layer 2 ex: Python runtime layer
Base Image Layer 1 ex: Base OS (usually Alpine so that containers are lightweight)

Container = image (with read only layers inside) + one writable layer at the top

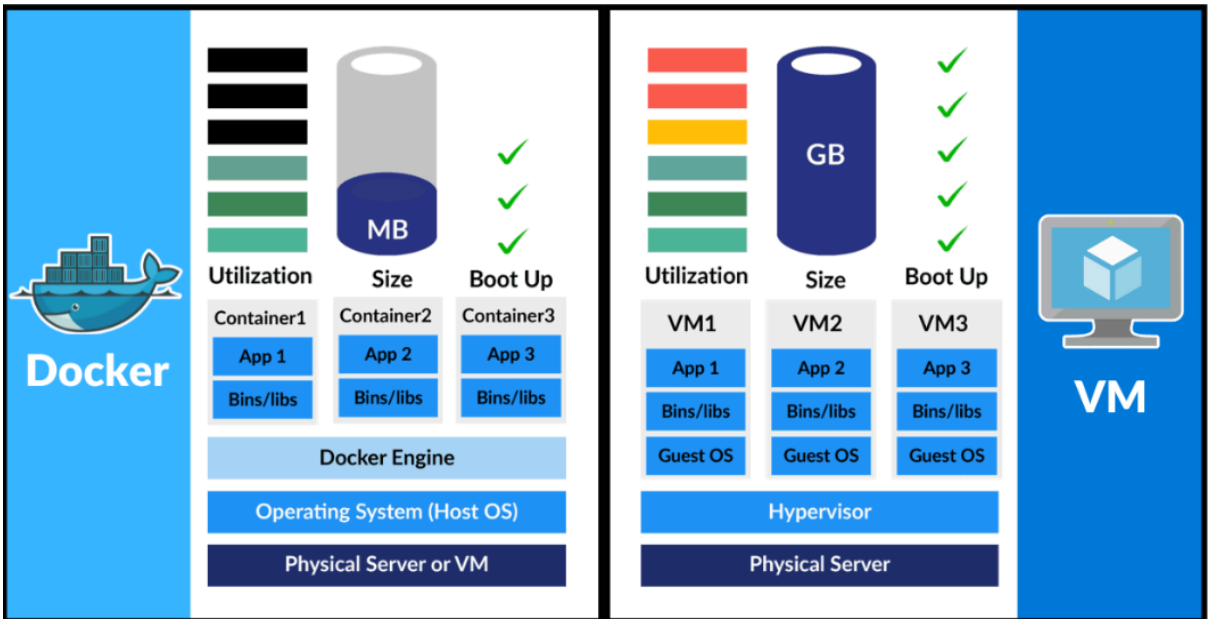
Docker Image

- Read-only template
- Built from layers
- Stored in registries
- Can be shared or pulled

Docker Container

- A running instance of an image
- Has a writable top layer
- Uses OS kernel from the host
- Lightweight and fast

Containers Vs Virtual Machines



Docker Containers

- Docker Containers use the Host OS Kernel (kernel = component of OS that interacts with the hardware), they don't have their own kernel.
  The container contains only the app and its dependencies, nothing more.
- Since the Host OS kernel is shared, docker images are light weight (in MBs).
- Fast bootup speed, in seconds.
- On an average machine, we can run many containers.
- Docker images can be version controlled.

- Since the host OS kernel is shared, OS compatible docker images need to be downloaded.
  Ex: You cannot run Linux docker image on old windows versions, before 11, check the compatibility of the image on docker hub before pulling it.

Virtual Machines

- VMs use their own Guest OS kernel on the top of Host OS kernel, it basically completely virtualises.
- Since the Host OS kernel is not shared, VM Images are large, in GBs.
- Slow bootup, in minutes.
- On an average machine, we can run one or two VMs max, since it's resource intensive.
- VMs cannot be version controlled.
- Since VMs have separate guest OS kernels, VMs are compatible on any OS.