Module 05

Purwadhika
Digital Technology School

AI Fullstack Software Development

Building and Running Apps in Containers with Docker

Job Connector Program

Outline

Purwadhika
Digital Technology School

- Intro to Docker
- Docker Setup

Intro to Docker

What is Docker ?

Docker is a software platform that allows you to quickly build, test, and deploy applications. Docker packages software into standard units called containers that have everything the software needs to function including libraries, system tools, code, and runtime. Using Docker, you can quickly deploy and scale your app to any environment and have confidence that your code will run.

Intro to Docker - Docker vs VM

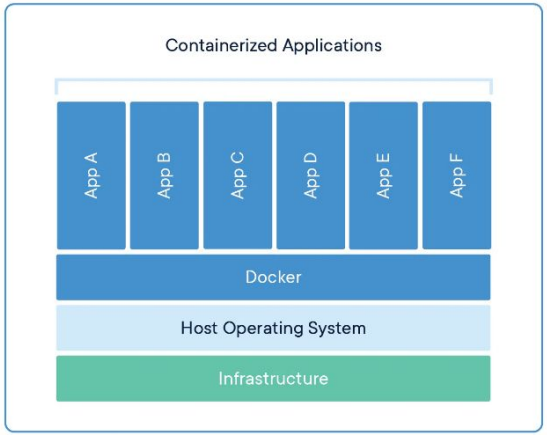
Docker

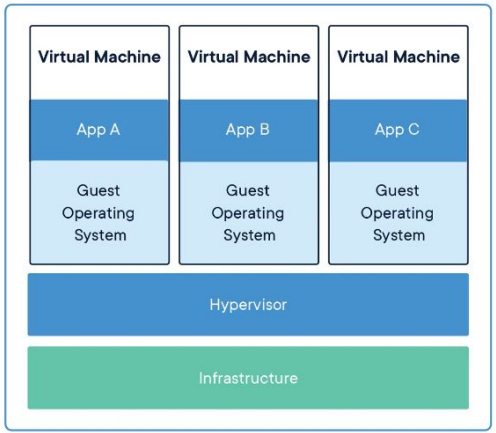
Allows developers to run applications that are owned in the form of containers on the operating system that is already installed on the server. So that on the server we no longer need to set up a VM, this is what causes Docker to be faster during the deployment process than a VM.

Virtual Machine

Every time we want to run an application on the server we have to prepare a virtual operating system first, and each application will usually be on a different virtual operating system on one computer. This is what causes it to take more time if we deploy the application to the VM.

Intro to Docker - Docker vs VM





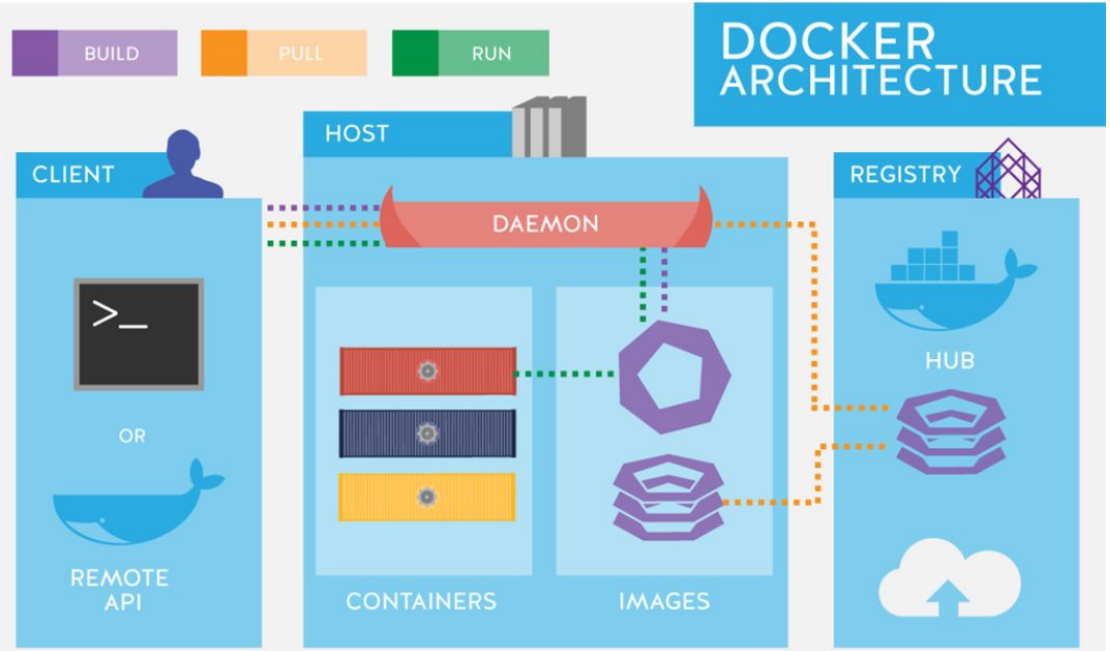
CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries – taking up tens of GBs. VMs can also be slow to boot.

Intro to Docker - Docker environment



Usually docker contains Docker Client, and Docker Server / Host.

Docker Client - contain terminal or command to execute Docker command

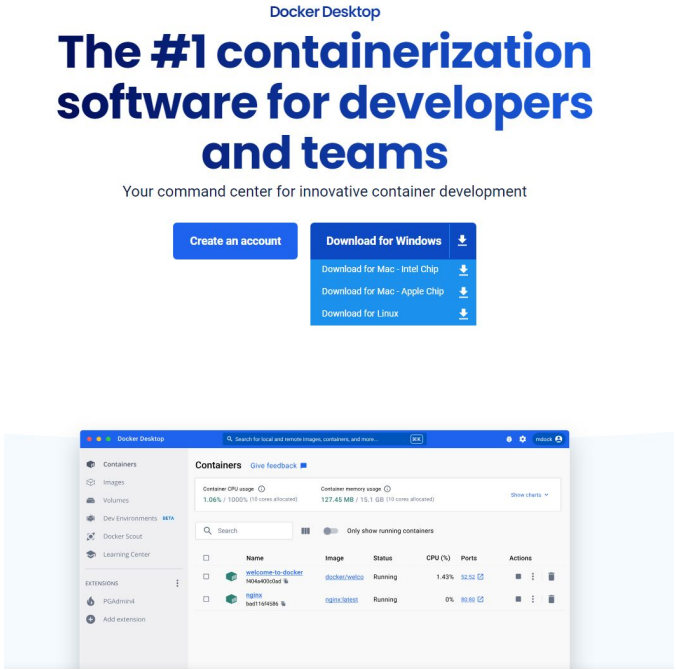
Docker Server - every request made by Docker Client, that would manage and executed using Docker Daemon.

Install Docker on Windows, Linux, Mac

Download from here :
https://www.docker.com/products/docker-desktop/

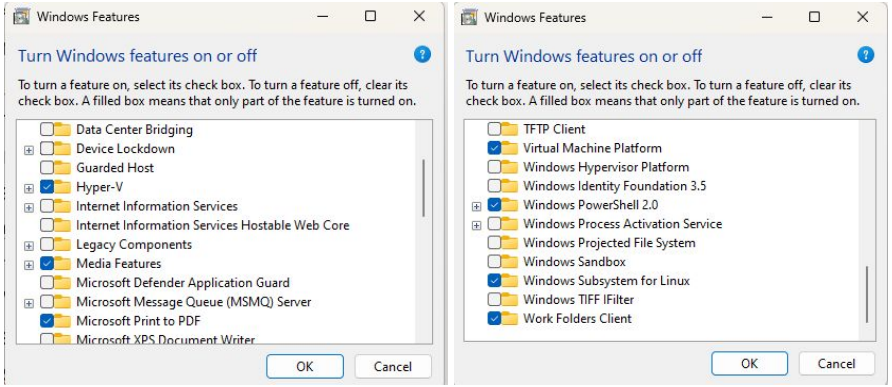
To run Docker in Windows we need to install **WSL** (windows subsystem linux) .

Why Docker needs Linux?
Docker is written in the Go programming language open_in_new and takes advantage of several features of the Linux kernel to deliver its functionality. Docker uses a technology called namespaces to provide the isolated workspace called the container.



Install WSL2 on Windows

- Open PowerShell as Administrator (Start menu > PowerShell > right-click > Run as Administrator) and enter this command: **dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart**
- Before installing WSL 2, you must enable the Virtual Machine Platform optional feature. Your machine will require virtualization capabilities to use this feature. Run : **dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart**
- Download the Linux kernel update package. here : *download here*
- Open PowerShell and run this command to set WSL 2 as the default version when installing a new Linux distribution: **wsl --set-default-version 2**
- Install linux distribution on powershell. Here : **wsl --install -d Ubuntu-22.04**
- Setup username and password to start using ubuntu on windows
- Enable **virtual machine platform**, **windows subsystem for linux**, **hyper v** on windows features



Ref :
https://learn.microsoft.com/en-us/windows/wsl/install-manual#step-3---enable-virtual-machine-feature

https://learn.microsoft.com/en-us/windows/wsl/install-manual#step-1---enable-the-windows-subsystem-for-linux

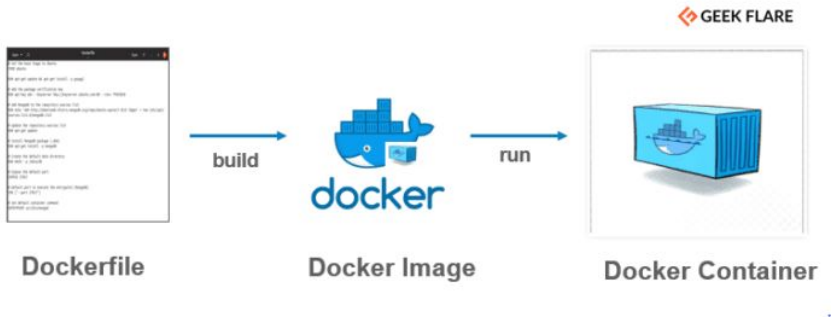
Install Docker Desktop on Windows

- Double-click Docker Desktop Installer.exe to run the installer.
- When prompted, ensure the Use WSL 2 instead of Hyper-V option on the Configuration page is selected or not depending on your choice of backend.
- If your system only supports one of the two options, you will not be able to select which backend to use.
- Follow the instructions on the installation wizard to authorize the installer and proceed with the install.
- When the installation is successful, select Close to complete the installation process.
- If your admin account is different to your user account, you must add the user to the docker-users group. Run Computer Management as an administrator and navigate to Local Users and Groups > Groups > docker-users. Right-click to add the user to the group. Sign out and sign back in for the changes to take effect.

Ref :
https://docs.docker.com/desktop/install/windows-install/#install-docker-desktop-on-windows

Things you need to know in Docker !

- **Containers :** A container is an isolated environment for your code. (**your apps**)
- **Images :** A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. (**instances of a container**)
- **Volumes :** In the context of Docker, a volume is a persistent storage location that exists outside of the container (**storage for your container**)



Docker Client

The Docker client provides a **command line interface (CLI)** that allows you to issue build, run, and stop application commands to a Docker daemon.

The main purpose of the Docker Client is to provide a means to direct the pull of images from a registry and to have it run on a Docker host.

Docker CLI - Common Commands

docker build

- Build an image from a Dockerfile

docker images

- List all images on a Docker host

docker run —name container_name images_name

- Run an image to create container

docker start container container_name

- Start existing container

Docker CLI - Common Commands

docker restart container container_name

- Restart existing container

docker rmi images_name

- Delete a local images

docker ps

- List all running

docker ps -a

- List all containers

Docker Server

Docker Daemon - In charge of managing images, either making changes or deleting. And also manage all the containers that are in docker.

Docker Host - The primary server that provides and runs the Docker Daemon.

Docker Registry - A place to store docker images, can be stored in the docker daemon for personal storage or stored in the Docker Hub so that it can be used by others.

Docker Objects

- Images Project templates that have been built into docker images.
- Networks Docker network that is used as a medium of communication between docker containers
- Containers Portable application that runs from docker images.

Docker Registry

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

- **Docker Hub :** https://hub.docker.com/
- **Google Container Registry**: https://cloud.google.com/container-registry

Docker Images

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Docker Container

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Dockerize your Project - Dockerfile

- Create a ‘dockerfile’ for your frontend/backend to stimulate the image of your apps

vite project

```
# Use the latest Node.js version
FROM node:22

# Set the working directory inside the container
WORKDIR /app

# Copy package.json and package-lock.json to the working directory
COPY package*.json ./

# Install app dependencies
RUN npm install

# Copy the rest of your application code to the working directory
COPY . .

# Expose a port to communicate with the React app
EXPOSE 5173

# Start the React app
CMD ["npm", "run", "dev"]
```

node js

```
FROM node:22

WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
EXPOSE 2000
VOLUME ["/app/node_modules"]
CMD ["npm", "run", "dev"]
```

Dockerize your Project - Vite Dockerfile

You have to setup server and preview on your vite-config.js to let the host access the project once it started as container.

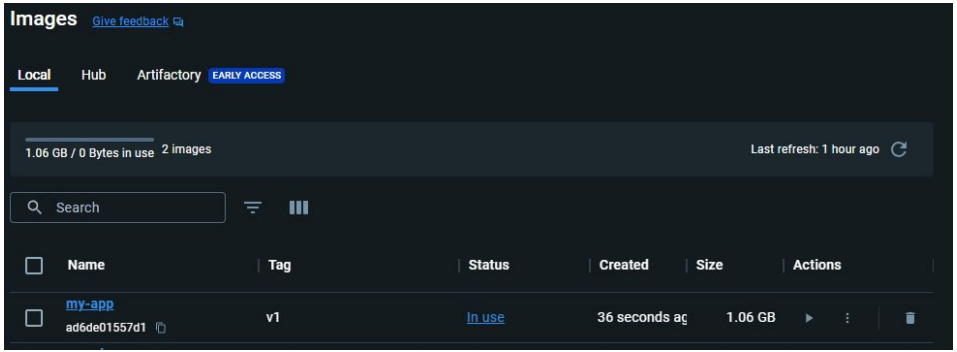
```
/** @format */

import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  server: {
    host: true,
    port: 5173,
  },
  preview: {
    host: true,
    port: 5173,
  },
});
```

Dockerize your Project - Build Docker Image

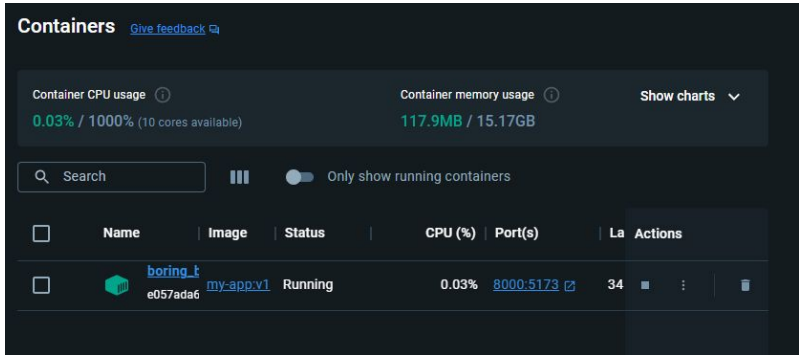
- After you create your dockerfile. Create your app image using this command :
- **docker build -t <Your App Image Name>:<version of the image app>** .
- Example : **docker build -t my-app:v1 .**

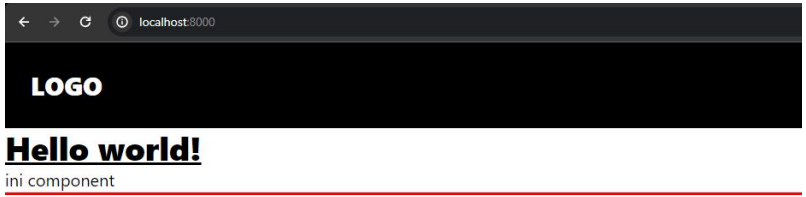


- Once it done, you can find your image app on your docker desktop

Crete and Run Docker Container

- Create your container from the images of your docker
- Run this command : **docker run -p <host port>:<docker port> <image_name>:<image_version> --name=container_name**
- **Note : host port is port accessible outside docker. Docker port is port that application running on docker**
- Example : **docker run -p 8000:5173 my-app:v1.** Means that your application port written in code is 5173.
- Now the container is running, so you can access your project through the host port.
- Check the logs to see if it works properly, by running this command : **docker logs <container_name>**
- **Make sure your app run command work properly. For example ; if you had prisma on you project write “prisma generate” on your dev starting point in package.json**





Find Docker Image in Docker Hub

- Find your images, for example search : postgresql
- Find your image version
- Use **docker pull <image_name>:<version>**
- Example : **docker pull postgres:16**

Docker Container - PostgreSQL

- **docker run --name <container_name> **
  **-e POSTGRES_USER=<username> **
  **-e POSTGRES_PASSWORD=<password> **
  **-e POSTGRES_DB=<database_name> **
  **-p <host_port>:5432 **
  postgres:<version>

Example : docker run --name POSTGRES_1 \ -e POSTGRES_USER=postgres \ -e POSTGRES_PASSWORD=password \ -e POSTGRES_DB=mydb \ -p 4404:5432 \ postgres:16

Access to Database

- Accessing database through host. Set your postgresql setting **host : “host.docker.internal”.**
- You can also access your postgresql through container name. **host: “container_name” .**

Docker Volume

Docker Volume is a **persistent storage** mechanism managed by **Docker**.

Why it matters:

- Containers are ephemeral (can be removed anytime)
- Database data **must not be lost**
- Volumes keep data outside the container lifecycle

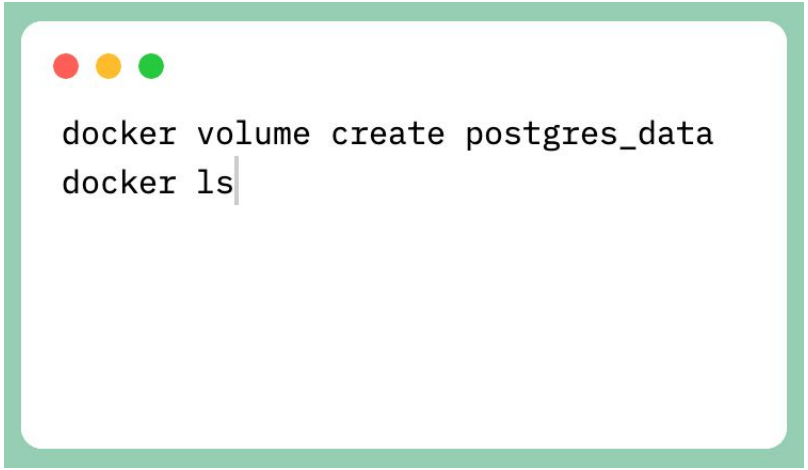
Docker Volume

```
$ docker volume create [OPTIONS] [VOLUME]
```

Command	Description
docker volume create	Create a volume
docker volume inspect	Display detailed information on one or more volumes
docker volume ls	List volumes
docker volume prune	Remove all unused local volumes
docker volume rm	Remove one or more volumes

Creating a Docker Volume

Using Docker Desktop (CLI or UI), we create and check a volume:



Running Database with Docker Volume

- **postgres:16** → PostgreSQL image
- **postgres_data** → Docker volume name
- **/var/lib/postgresql/data** → PostgreSQL data directory

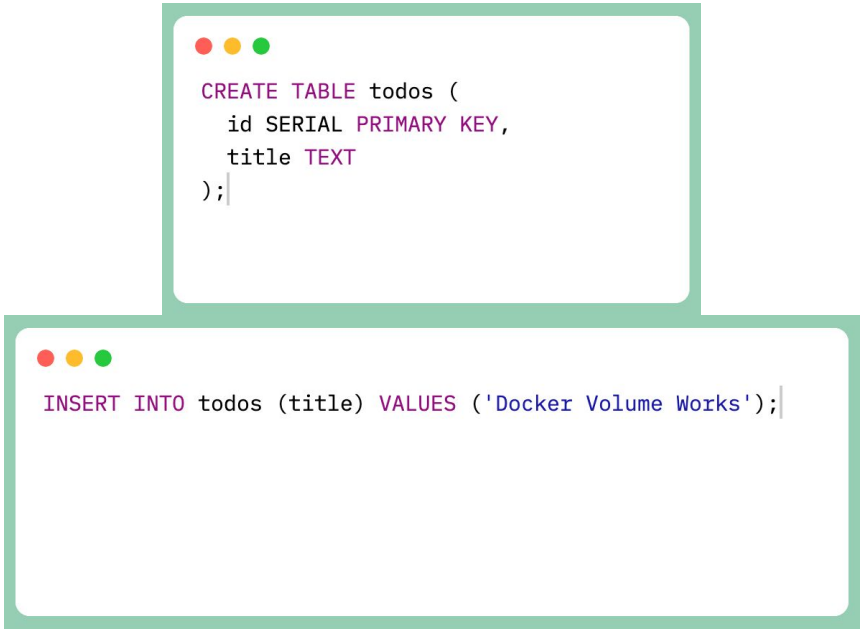
Now PostgreSQL is:

- Running inside Docker
- Persisting data using a volume

```
docker run -d \
  --name postgres-db \
  -p 5432:5432 \
  -e POSTGRES_USER=postgres \
  -e POSTGRES_PASSWORD=postgres \
  -e POSTGRES_DB=todo_db \
  --mount source=postgres_data,target=/var/lib/postgresql/data \
  postgres:16
```

Verifying the Database is Running

- Using Docker Desktop:
  - Container status: Running
  - Port 5432 exposed
  - Volume attached to container
- Using database client (example):
  - Connect to:
    - Host: localhost
    - Port: 5432
    - User: postgres
      Database: todo_db
- Create table and insert data



Proving Data Persistence (The Key Test)

1️⃣ Stop and remove the container:

```
docker stop postgres-db

docker rm postgres-db
```

2️⃣ Run PostgreSQL again using the same **volume**:

```
docker run -d \
  --name postgres-db \
  -p 5432:5432 \
  --mount source=postgres_data,target=/var/lib/postgresql/data \
  postgres:16
```

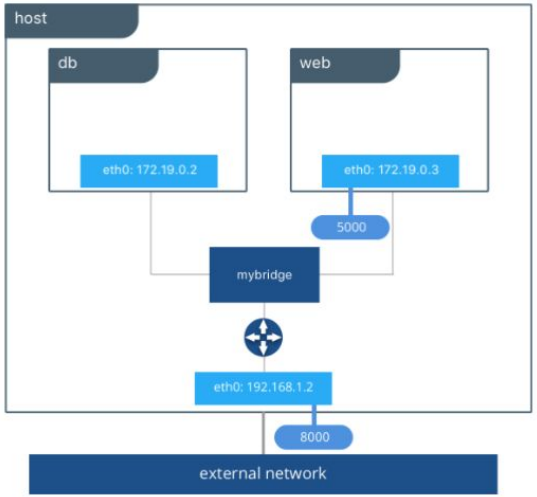
3️⃣ Reconnect to the database:

```
SELECT * FROM todos;
```

✅ The data is **still there**

Docker Network

Networking is about communication among processes, and Docker's networking is no different. Docker networking is primarily used to establish communication between Docker containers and the outside world via the host machine where the Docker daemon is running.



Docker Network

```
$ docker network COMMAND
```

Command	Description
docker network connect	Connect a container to a network
docker network create	Create a network
docker network disconnect	Disconnect a container from a network
docker network inspect	Display detailed information on one or more networks
docker network ls	List networks
docker network prune	Remove all unused networks
docker network rm	Remove one or more networks

In order to manage networks, you can use subcommands to create, inspect, list, remove, prune, connect, and disconnect networks.

Docker Network

Lets create a new network in our docker. Write down this command to create and check if that network created successfully.

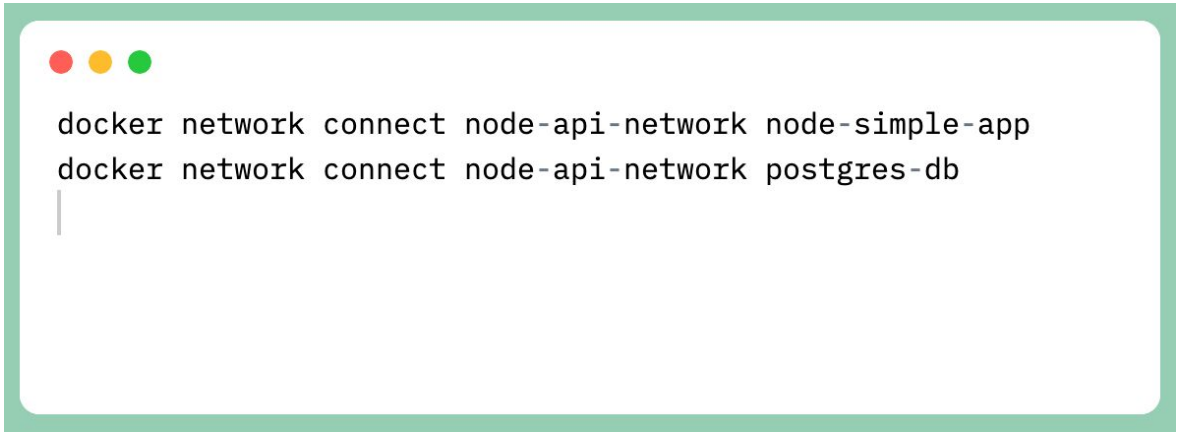
```
docker network create node-api-network
docker network ls
```

```
NETWORK ID     NAME                DRIVER    SCOPE
2c108776517e   be-sample_default   bridge    local
c3e7fb539144   bridge              bridge    local
795bdc01fc06   host                host      local
001479a01cd3   network-simple      bridge    local
dff78aaa1bf7   node-api-network    bridge    local
62c7e8e1f459   none                null      local
899324ac7e9f   stable-network      bridge    local
```

Docker Network

Add container name that you would like to connect into network you just made before.

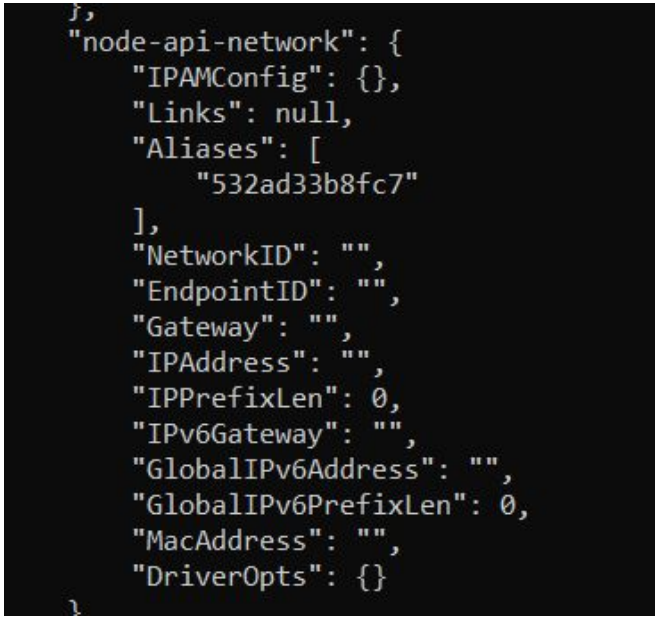
Lets connect node-simple-app and postgres_db into the same network.



Docker Network

Let's check if the container we just add is successfully connect with the network. Check each container using inspect argument.

```
docker container inspect postgres-db
docker container inspect node-simple-app
```



Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.

Compose works in all environments: production, staging, development, testing, as well as CI workflows.

Docker Compose

Using Compose is basically a three-step process:

- Define your app's environment with a Dockerfile so it can be reproduced anywhere.
- Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
- Run docker compose up and the Docker compose command starts and runs your entire app. You can alternatively run docker-compose up using the docker-compose binary.

Docker Compose

Compose has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Docker Compose - Compose Specification

The Compose file is a YAML file defining services, networks, and volumes for a Docker application. The latest and recommended version of the Compose file format is defined by the Compose Specification.

This document specifies the Compose file format used to define multi-containers applications. Distribution of this document is unlimited.

Docker Compose - Compose File

The Compose file is a YAML file defining:

- Version (DEPRECATED)
- Services (REQUIRED)
- Networks, volumes, configs and secrets.

The default path for a Compose file is compose.yaml (preferred) or compose.yml in working directory. Compose implementations SHOULD also support docker-compose.yaml and docker-compose.yml for backward compatibility. If both files exist, Compose implementations MUST prefer canonical compose.yaml one.

Docker Compose - Compose File

Lets go back to our projects, this time we will implement starting container node app and postgresql using docker compose. Lets create file name as docker-compose.yaml and put this code into your file.

```yaml
version: "3.8"

networks:
  node-api-network:
    name: node-api-network

volumes:
  postgres_data:

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: node-simple-app
    ports:
      - "3000:3000"
    volumes:
      - .:/app
      - /app/node_modules
    depends_on:
      - postgres_server
    networks:
      - node-api-network

  postgres_server:
    image: postgres:16
    container_name: postgres-db
    environment:
      POSTGRES_DB: todo_db
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - node-api-network
```

Docker Compose - Compose File

"If you are using **Windows**, modify the **`package.json`** file and change the script `"dev": "nodemon src/index.ts"` to `"dev": "nodemon --legacy-watch src/index.ts"`."

package.json

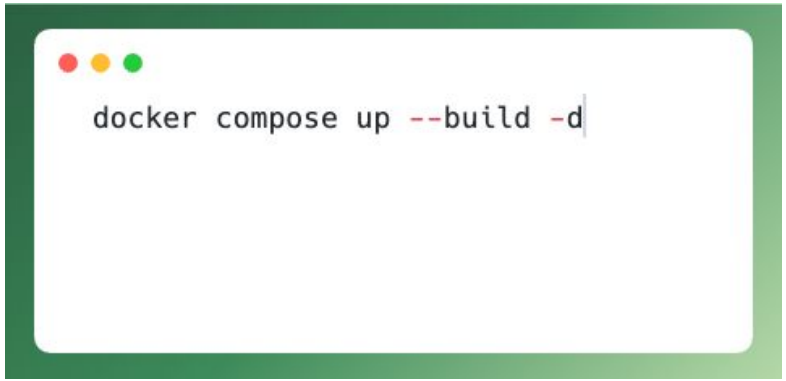
```
"scripts": {
    "build": "tsc",
    "start": "npm run build && node dist/index.js",
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev": "nodemon src/index.ts"
  }
```

package.json

```
"scripts": {
    "build": "tsc",
    "start": "npm run build && node dist/index.js",
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev": "nodemon --legacy-watch src/index.ts"
  }
```

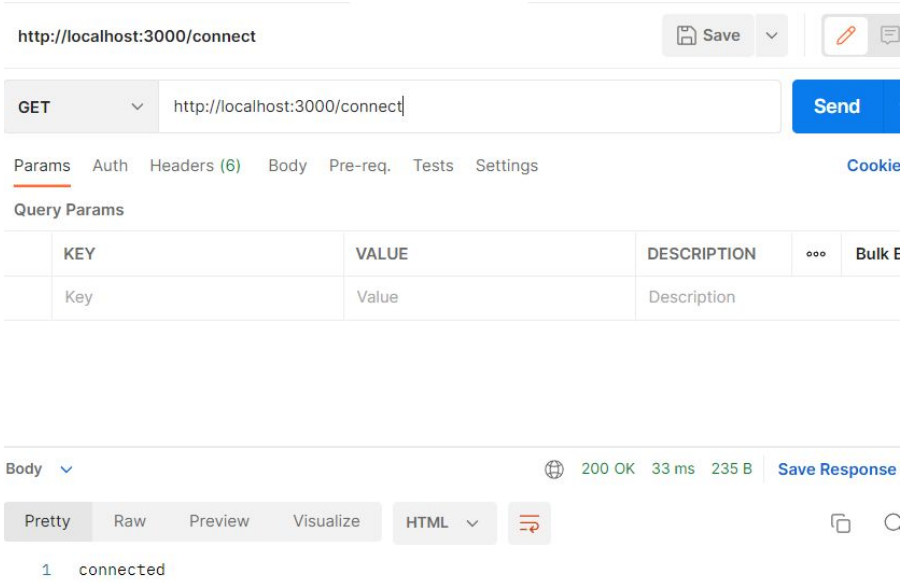
Docker Compose - Compose File

Lets run your container, use docker-compose command to take an action for docker-compose.yaml file. You can check through docker desktop for running container. But this time, there is more than one container running and wrapped by one container



Docker Compose - Compose File

Lets try and test our projects through postman!



Exercise

In this task, you will dockerize the **Blog App** you previously built by creating a **Dockerfile** that sets up a Node.js environment, installs all dependencies, copies the application code, exposes the required port, and runs the app inside a container.

Thank you