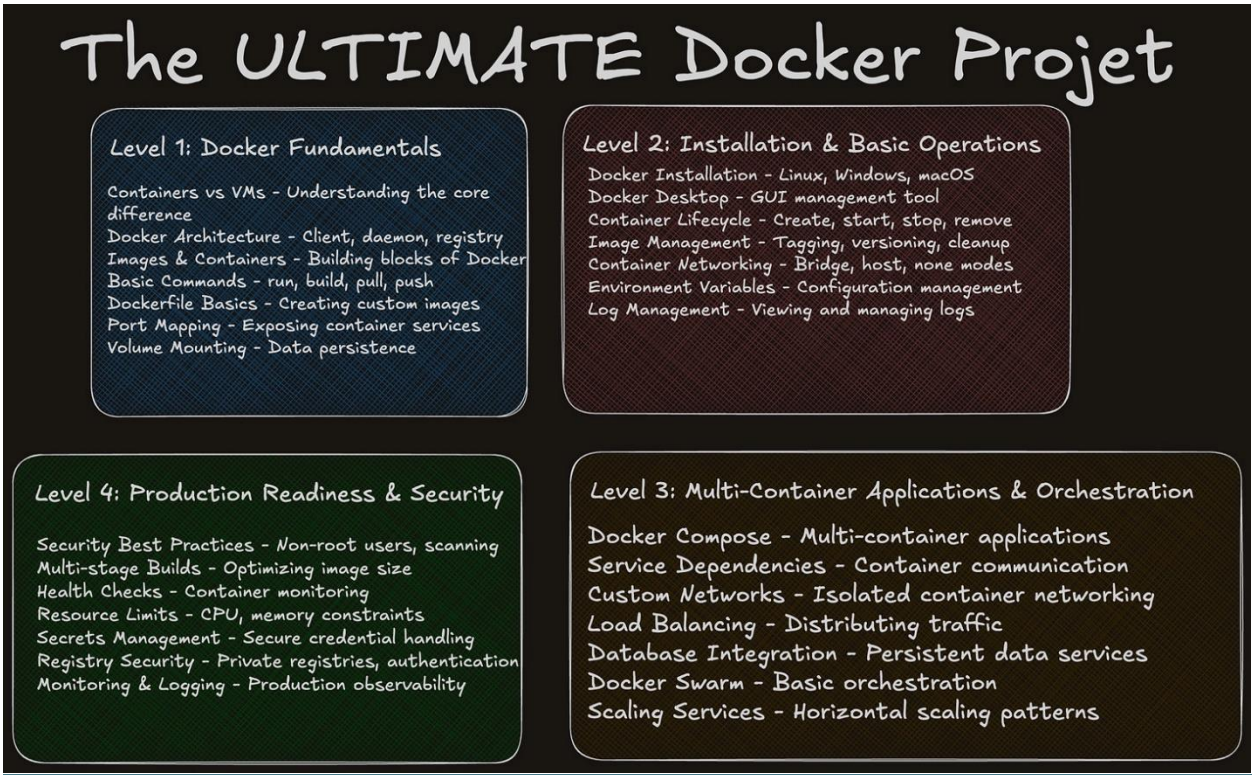
I wanted to sit down and create the **best** possible starting point for learning Docker.

This will be split into **three** parts, which we'll use to build what I'm calling "The Ultimate Docker Project 🐳". The first two parts will give you everything you need to follow along with the final project in part three.



Part One Covers:

Level 1: Docker Fundamentals

Part Two Covers:

Level 2: Installation and Basic Operations + **Level 3:** Multi-Container Applications & Orchestration

Part Two Covers:

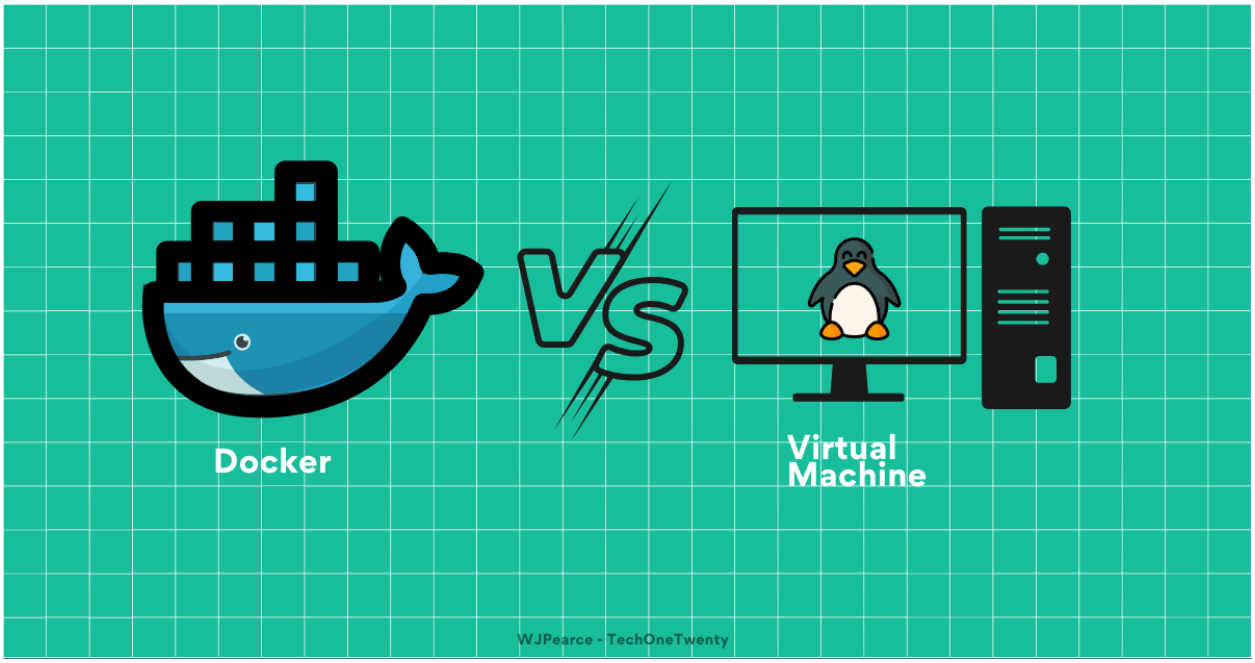
Level 4: Production Readiness & Security + **The Ultimate Docker Project**

Level One

Containers vs VMs: Understanding the core difference

Quick note: You may recognise some of this early foundational material from TechOneTwenty, after the initial few sections, it's all original material you will only find here on **Cyber Notes.**

Let's start by taking a look at how docker works when comparing it to a traditional virtual machine.
Docker and traditional virtual machines both allow for the isolation and deployment of applications, but they do so in fundamentally different ways.



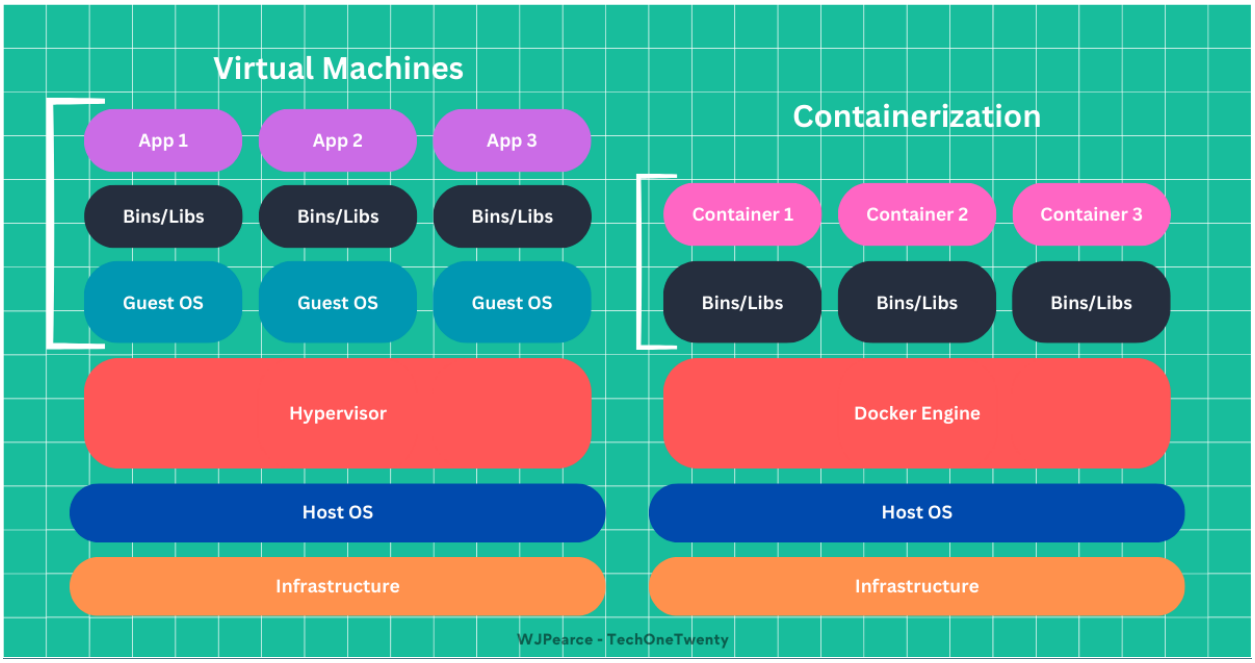
Docker uses containerisation, which packages an application and its dependencies into a single, lightweight container that shares the host operating system's kernel.

This makes Docker containers more resource efficient and faster to start up compared to VMs because they don't need to boot an entire OS.

In contrast, traditional VMs run full operating systems on top of a hypervisor, which is a software layer that manages multiple VMs on a host machine. Each VM includes its own OS, libraries, and application binaries, which can result in higher resource consumption and slower performance due to the overhead of running multiple OS instances.

If non of that made sense, don't panic 😵 I will cover it all here.

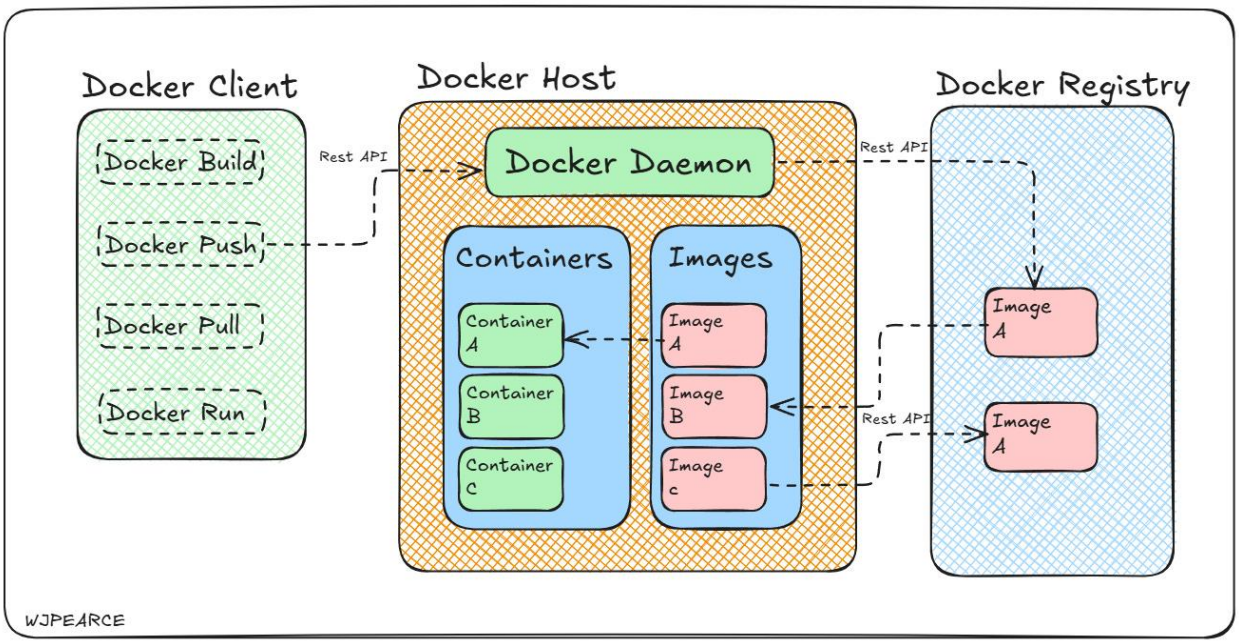
This diagram should help digest and understand the above:



Docker Architecture: The Three Components

Docker operates on a client server architecture with three main components that work together. Understanding how these pieces interact will help you grasp how Docker commands actually execute behind the scenes.

Take it from someone who loves jumping in and building stuff before they understand it. That's a valid approach, but it's not always the best. Take the time here to understand the architecture.



The Docker Client is what you interact with directly through the command line. When you type commands like docker run or docker build, you're using the Docker client. This client doesn't actually do the heavy lifting, instead, it sends your commands to the Docker daemon via a REST API. ***(If you aren't sure what is meant by REST API, let me know down below. I really love seeing what level my audience is at)***

The rest is for members only. **Cyber Notes is as affordable as Substack allows!**

I pour in hours of work crafting weekend projects and technical explainers. Most of them for **free**. Your support keeps this growing and means the world 🌎

Join one of the fastest growing Cloud & Cyber communities below ⬇️

If you're reading this, it means you're a member of Cyber Notes, I truly can't thank you enough for that. I absolutely love sharing my knowledge here in a clear, visual and practical way. I really hope it helps! Any questions, feel free to comment.

The Docker Daemon (or dockerd) is the powerhouse that runs continuously in the background on your host machine. It's responsible for managing Docker objects like images, containers, networks, and volumes. The daemon listens for API requests from the

Docker client and handles all the actual work of building, running, and managing your containers.

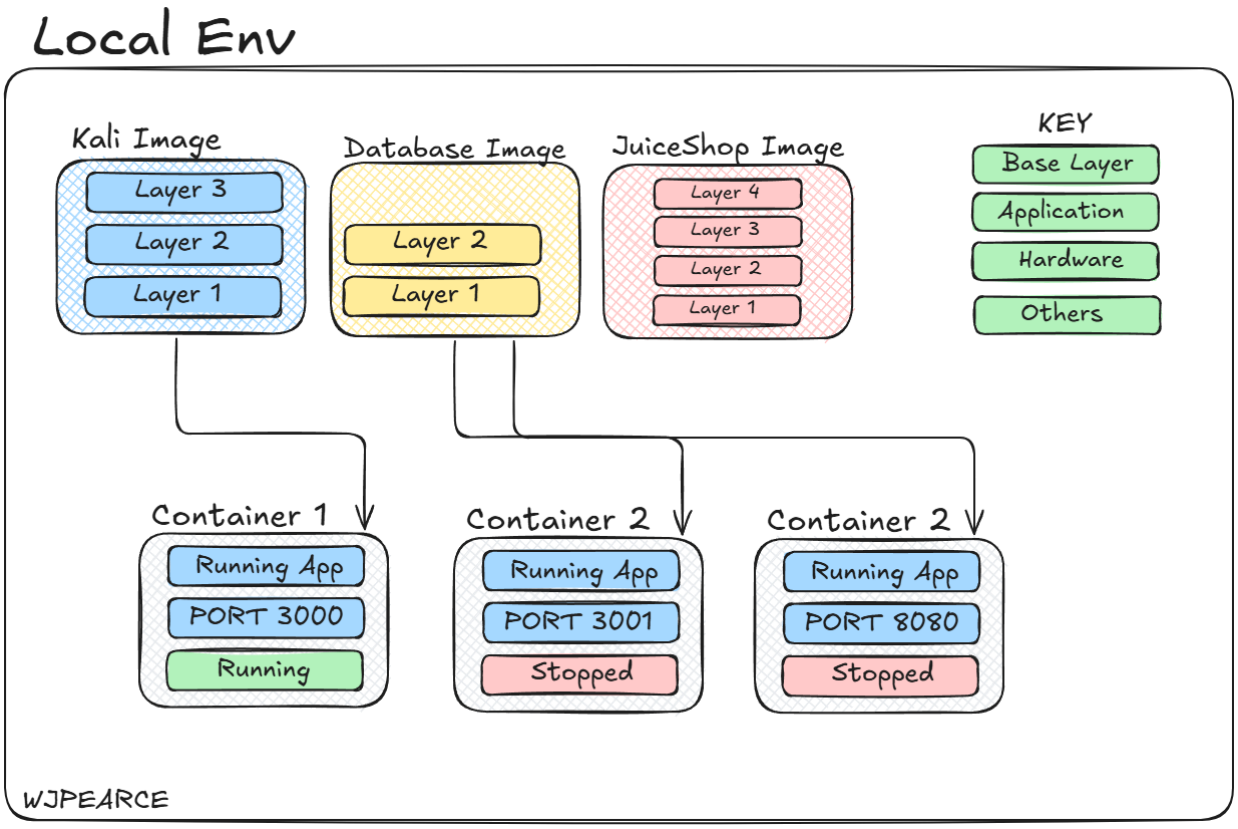
The Docker Host is simply the machine where the Docker daemon runs, this could be your local development machine, a server, or a cloud instance. The host provides the underlying resources (CPU, memory, storage) that containers use when they're running. I've recently ran into an issue where sharing images to and from M1 Macbooks, but more on that another time.

Docker Registry is where Docker images are stored and distributed. The most well known registry is Docker Hub, although I use AWS ECR more. Docker hub hosts millions of public images that you can pull and use. When you run docker pull nginx, you're downloading the nginx image from a registry. You can also set up private registries for your organisation's internal images.

These three components communicate.
You issue commands through the client > the daemon executes them > and images flow to and from registries as needed. This architecture allows for flexible deployment scenarios, as the client and daemon can even run on different machines.

Images & Containers: Building blocks of Docker

Think of Docker images and containers as the blueprint and the house, related but different concepts that make up Docker's foundation.



Docker Images are read only templates that contain everything needed to run an application: the code, runtime, system tools, libraries, and settings. Images are built in layers, with each layer representing a change or addition to the previous one: **Remember this!**

This layered approach makes images easier to manage, if multiple images share common layers (like the same base operating system), Docker only stores one copy of each shared layer... This is more forgiving on your host resources.

You can think of an image as a snapshot of a file system at a specific point in time, like when you're emulating a game and you save the ROM file at a point in time (Sorry if that example doesn't land.)

Images are immutable, meaning once created, they don't change. When you download an image with docker pull ubuntu, you're getting a complete Ubuntu environment packaged as an image.

Docker Containers are running instances of images. When you execute docker run, Docker takes an image and creates a container from it, basically bringing that static

blueprint to life. Containers are where your applications actually run, and unlike images, they're mutable. You can write files, install software, and make changes within a running container.

The key relationship: one image can spawn multiple containers. You might have a single Python application image but run tons of containers from it across different environments. Each container operates independently, even though they originated from the same image.

Basic Commands: run, build, pull, push

Don’t worry about needomg to commit these to memory they're used repeatedly when working with Docker and they’ll sink in with time and use.

Image Management:

- docker pull <image_name> - Downloads a Docker image from a registry (such as Docker Hub)
- docker build -t <tag_name> <path> - Builds a Docker image from a Dockerfile at the specified path and tags it with a name
- docker image ls - Lists all Docker images available on your local machine
- docker image rm <image_id> - Removes a specified Docker image from your local machine

Container Operations:

- docker run <options> <image_name> - Creates a new container from a specified image and starts it
- docker ps - Lists all running containers (add -a to show all containers, including stopped ones)
- docker stop <container_id> - Stops a running container
- docker rm <container_id> - Removes a container (must be stopped first)
- docker exec -it <container_id> <command> - Executes a command inside a running container, allowing interactive access

Beyond the Basics: These commands will get you started, but Docker has dozens more for specific tasks: docker logs for viewing container output, docker inspect for detailed

container information, docker network for managing container networking, and many others.

Rather than overwhelming you with an exhaustive list (which would make for a rather boring read!), focus on mastering these core commands first. As you work with Docker more, you'll naturally discover additional commands when you need them. If you ever need a specific command, a quick search or docker --help will point you in the right direction

...or honestly, just use **AI** if you forget.

Dockerfile Basics: Creating custom images

While pulling pre built images from Docker Hub is convenient, the real power of Docker comes from creating your own custom images tailored to your specific applications. This is where Dockerfiles come in, simple text files that contain step by step instructions for building your own images.

What is a Dockerfile? A Dockerfile is a recipe that tells Docker how to build an image. It's a plain text file (literally named Dockerfile with no extension) that contains a series of commands that Docker executes in order. Think of it as an automated script that installs software, copies files, and configures everything your application needs to run.

Needed Dockerfile Instructions:

- FROM - Specifies the base image to start from (e.g., FROM ubuntu:20.04)
- COPY - Copies files from your host machine into the image
- RUN - Executes commands during the build process (installing packages, etc.)
- WORKDIR - Sets the working directory inside the container
- EXPOSE - Documents which ports the container will use
- CMD - Specifies the default command to run when the container starts

Let’s test this

Create a project directory:

Open Command Prompt and run the following commands to create and navigate to your project directory:

mkdir docker-nginx-app

cd docker-nginx-app

Create a file named Dockerfile in the docker-nginx-app directory with the following content:

```
# Use the official NGINX image from the Docker Hub
FROM nginx:alpine

# Copy custom configuration file from the host to the container
COPY nginx.conf /etc/nginx/nginx.conf

# Copy static website files to the container
COPY html /usr/share/nginx/html
```

In the docker-nginx-app directory, create a file named nginx.conf with the following content:

```
events {}

http {
  server {
    listen 80;

    location / {
      root /usr/share/nginx/html;
      index index.html;
    }
  }
}
```

Let's test this

Create a project directory:

Open Command Prompt and run the following commands to create and navigate to your project directory:

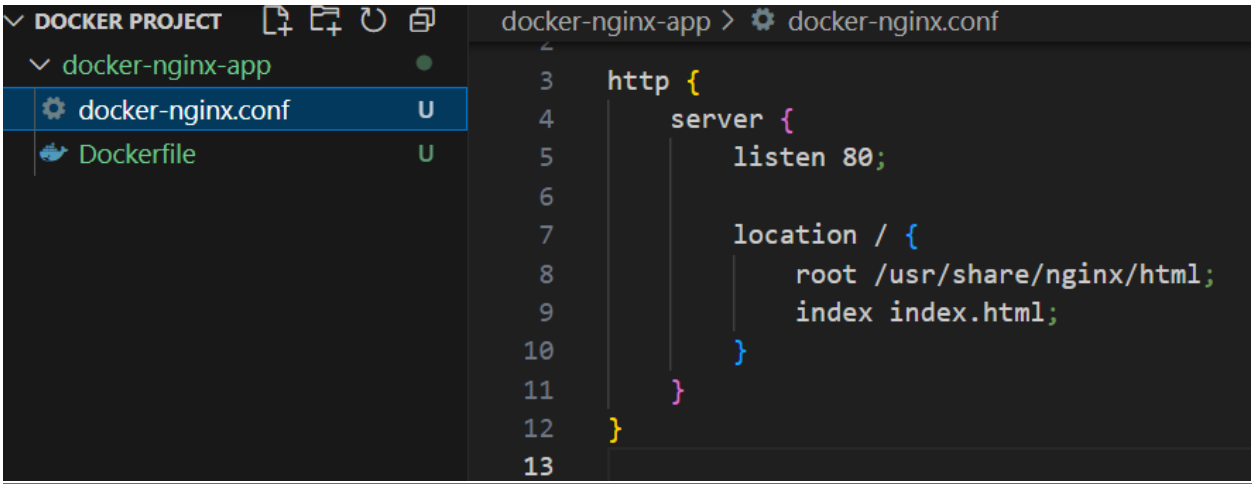
```
mkdir docker-nginx-app

cd docker-nginx-app
```

Create a file named Dockerfile in the docker-nginx-app directory with the following content:

```
# Use the official NGINX image from the Docker Hub
FROM nginx:alpine

# Copy custom configuration file from the host to the container
COPY nginx.conf /etc/nginx/nginx.conf

# Copy static website files to the container
COPY html /usr/share/nginx/html
```

In the docker-nginx-app directory, create a file named nginx.conf with the following content:

```
events {}

http {
    server {
        listen 80;

        location / {
            root /usr/share/nginx/html;
            index index.html;
        }
    }
}
```



Create HTML Directory and index.html File:

In the docker-nginx-app directory, create a folder named html, then create an index.html file inside the html folder with the following content:

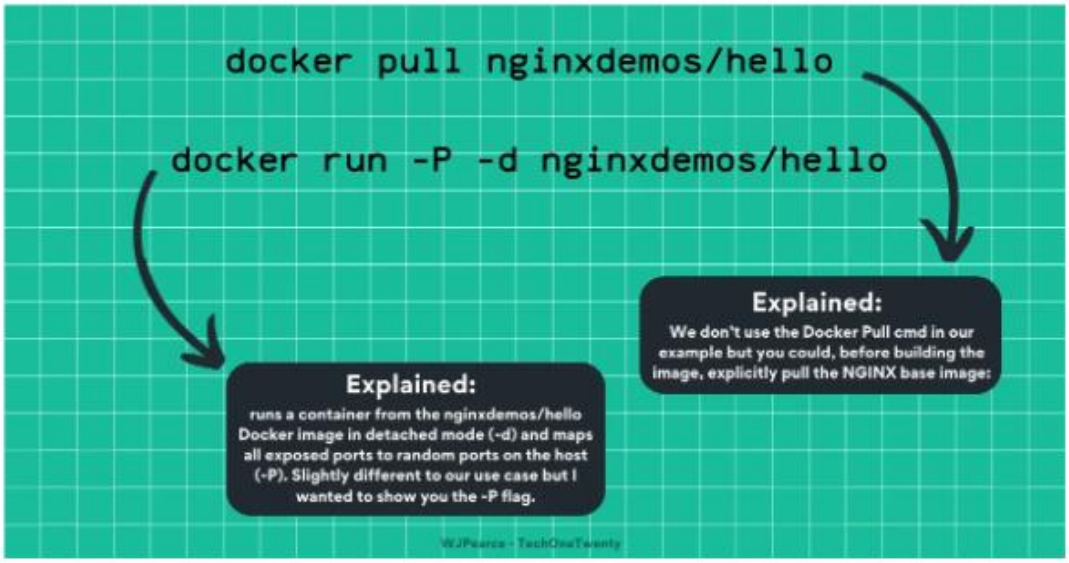
```
<h1>YOUR CUSTOM TEXT</h1>
```

Your project directory structure should look like this: docker-nginx-app

Open Command Prompt in the docker-nginx-app directory and run:

```
docker build -t my-nginx .
```

This command uses the Dockerfile in the current directory to build an image named my-nginx.



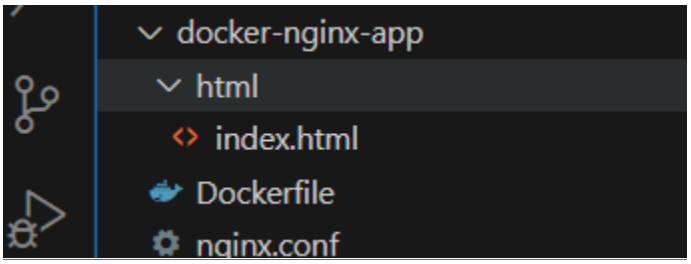
Run the Docker Container:

```
docker run -d -p 8080:80 --name my-nginx-container my-nginx
```

In the docker-nginx-app directory, create a folder named html, then create an index.html file inside the html folder with the following content:

<h1>YOUR CUSTOM TEXT</h1>

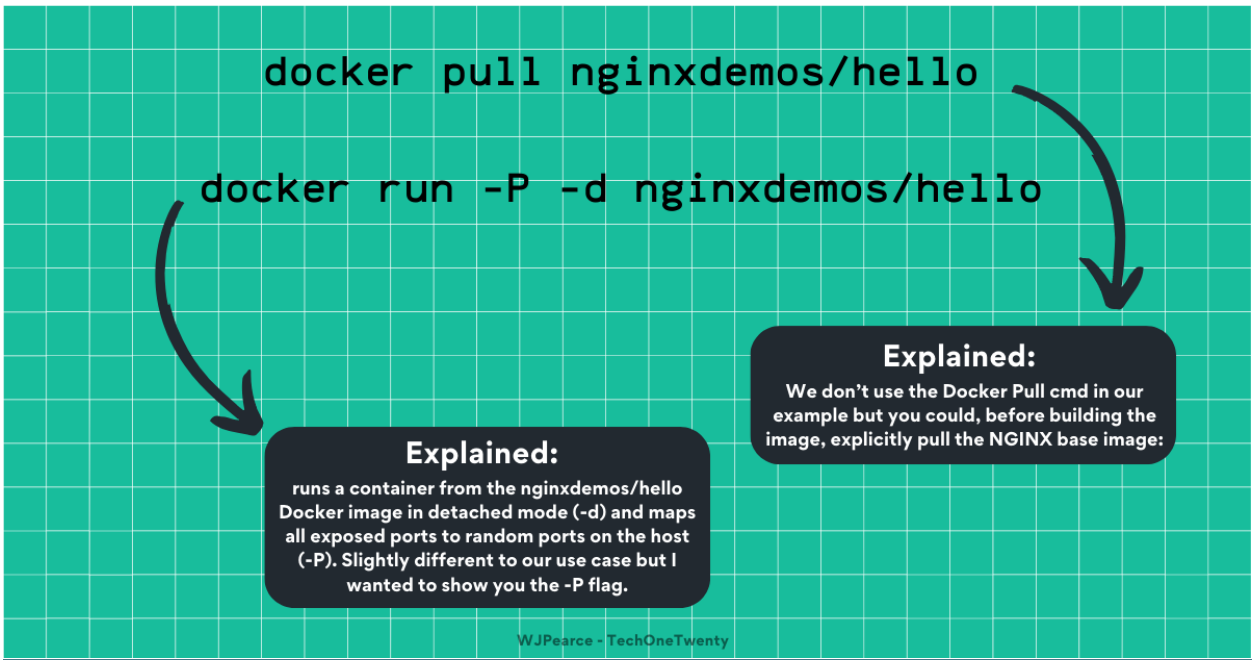
Your project directory structure should look like this: docker-nginx-app



Open Command Prompt in the docker-nginx-app directory and run:

docker build -t my-nginx .

This command uses the Dockerfile in the current directory to build an image named my-nginx.



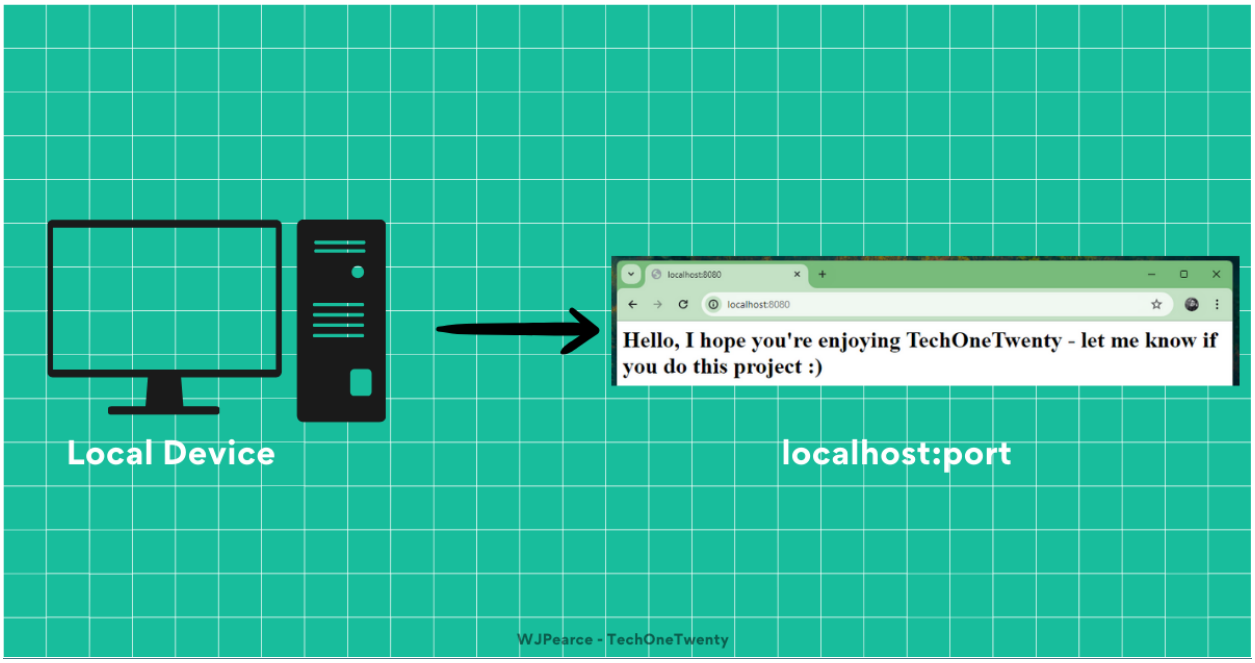
Run the Docker Container:

docker run -d -p 8080:80 --name my-nginx-container my-nginx

This command runs the container in detached mode (-d) - this is so we can still use the terminal. Maps port 8080 on your host to port 80 in the container (-p 8080:80), and names the container my-nginx-container

Check running containers with docker ps

You should see my-nginx-container running. Open a web browser and go to http://localhost:8080
You should see the message "<Your own message ;) >



Port Mapping: Exposing container services

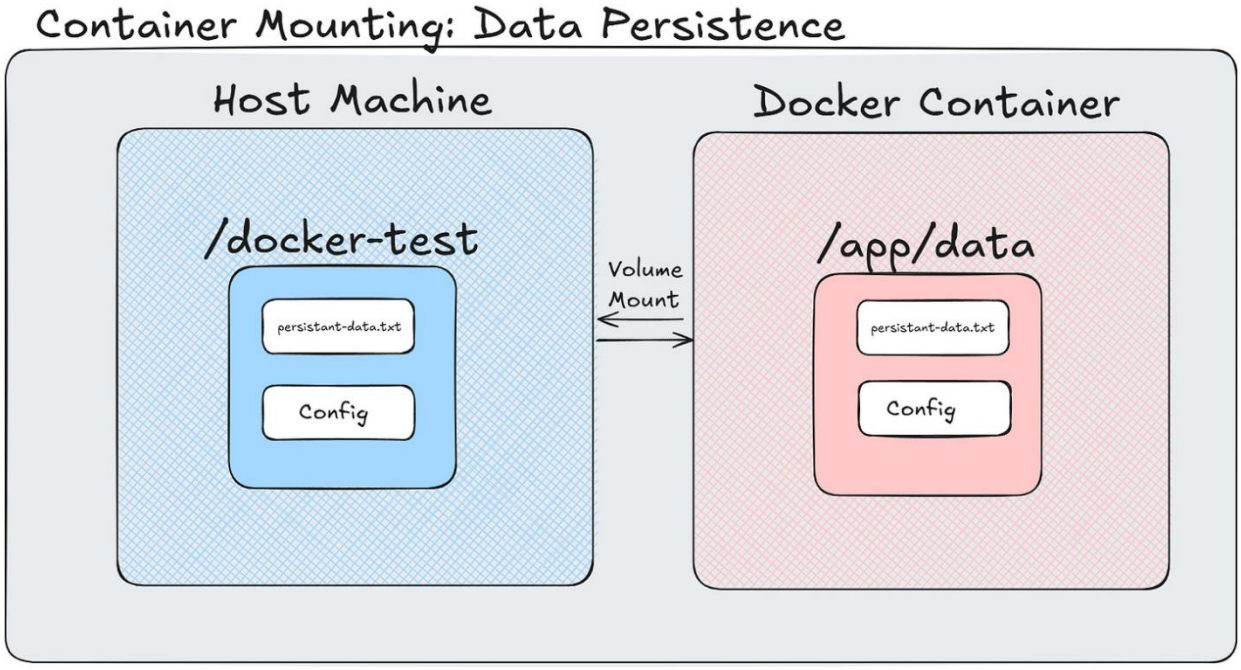
I've decided to move this section to part 3, I felt it made way more sense in that Level.

Volume Mounting: Data persistence

Volume Mounting: Data Persistence
By default, containers are ephemeral. When you stop and remove a container, everything inside it disappears. This poses a problem: what happens to your database files, user

uploads, or configuration data? Volume mounting solves this by creating a bridge between your container and the host machine's file system.

Why Volume Mounting Matters:
When a container writes data to its internal file system, that data lives and dies with the container. Delete the container, and your data vanishes. Volume mounting allows you to specify directories or files that should persist beyond the container's lifecycle, stored safely on your host machine.



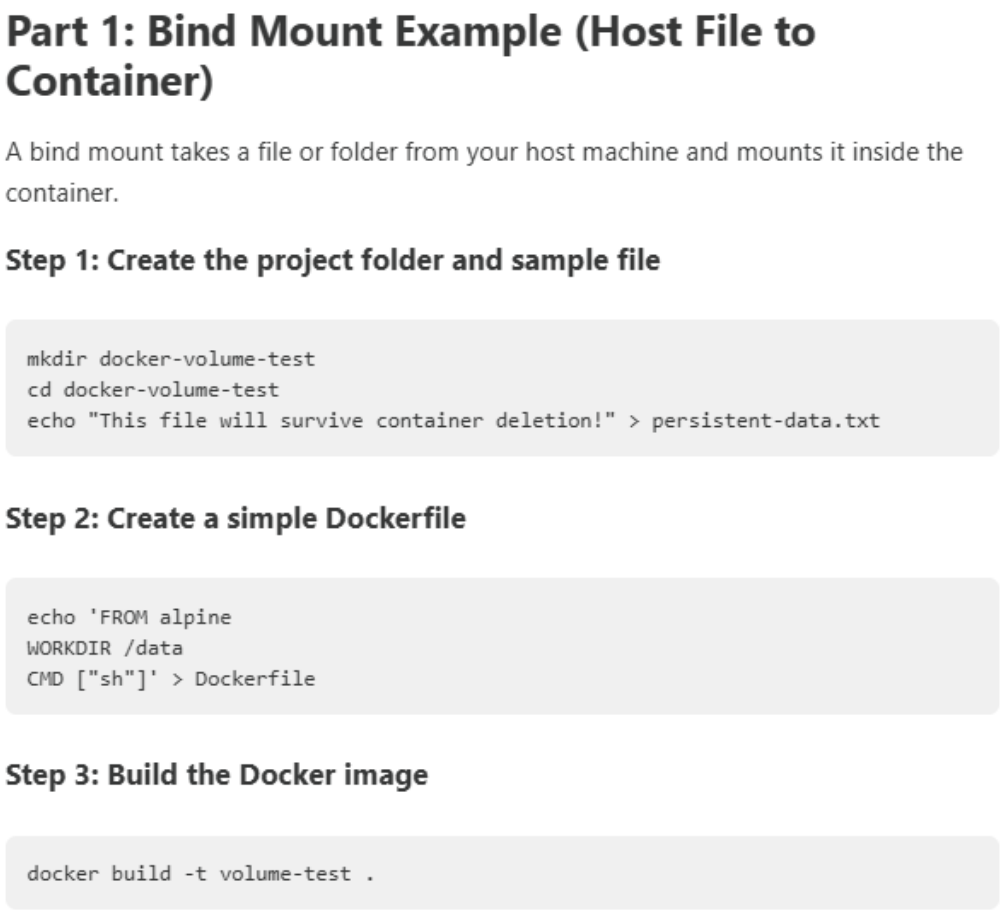
How It Works:
Volume mounting creates a connection between a directory on your host machine and a directory inside the container. Any changes made to files in the mounted directory are reflected in both locations immediately. Think of it as a shared folder that both your host and container can access.

Project 🚀

Okay lets finish up with small project you can do at home to help understand the above.

This tutorial shows you two ways to persist data in Docker containers: **bind mounts** and **Docker volumes**. You'll create both types, write data to them, delete the containers, and verify that your data survives.

Part 1: Bind Mount Example (Host File to Container)



Part 1: Bind Mount Example (Host File to Container)

A bind mount takes a file or folder from your host machine and mounts it inside the container.

Step 1: Create the project folder and sample file

```
mkdir docker-volume-test
cd docker-volume-test
echo "This file will survive container deletion!" > persistent-data.txt
```

Step 2: Create a simple Dockerfile

```
echo 'FROM alpine
WORKDIR /data
CMD ["sh"]' > Dockerfile
```

Step 3: Build the Docker image

```
docker build -t volume-test .
```

A bind mount takes a file or folder from your host machine and mounts it inside the container.

Step 1: Create the project folder and sample file

mkdir docker-volume-test

cd docker-volume-test

echo "This file will survive container deletion!" > persistent-data.txt

Step 2: Create a simple Dockerfile

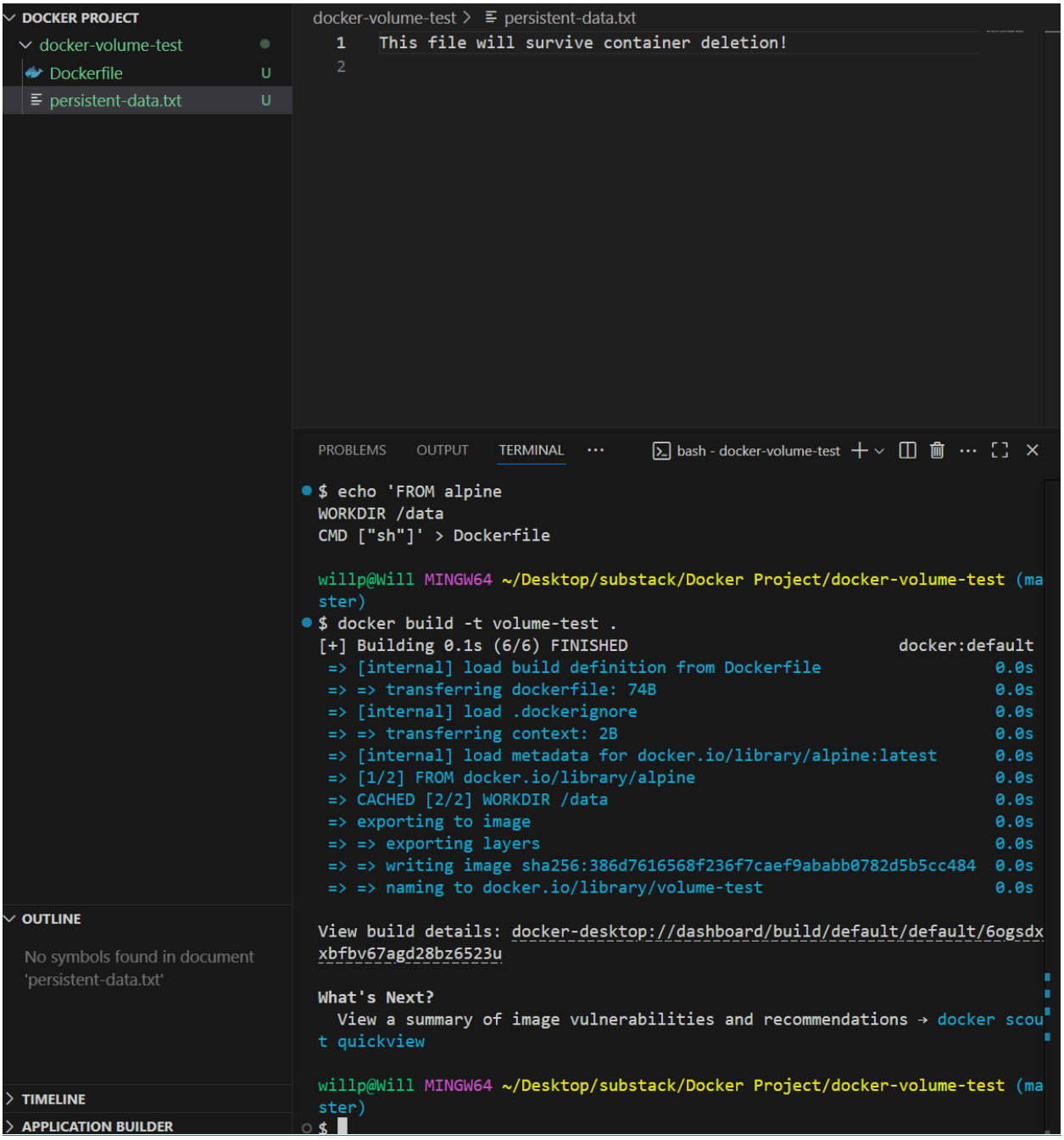
echo 'FROM alpine

WORKDIR /data

CMD ["sh"]' > Dockerfile

Step 3: Build the Docker image

docker build -t volume-test .



Step 4: Run container with bind mount

Step 4: Run container with bind mount

```
# Windows (Git Bash/WSL) - this will be different on mac
winpty docker run -it --name bind-container -v "$(pwd)/persistent-
data.txt:/data/persistent-data.txt" volume-test
```

This mounts your `persistent-data.txt` from your local folder into the container at `/data/persistent-data.txt`.

Step 5: Modify the file inside the container

```
cd /data
echo "Appended from inside container!" >> persistent-data.txt
cat persistent-data.txt
exit
```

Step 6: Delete the container

```
docker rm bind-container
```

Step 7: Verify persistence on your host

```
cat persistent-data.txt
```

You should see both the original text and the appended text. The file persisted because it lives on your host machine!

Windows (Git Bash/WSL) - this will be different on mac

winpty docker run -it --name bind-container -v "$(pwd)/persistent-data.txt:/data/persistent-data.txt" volume-test

This mounts your persistent-data.txt from your local folder into the container at /data/persistent-data.txt.

Step 5: Modify the file inside the container

cd /data

echo "Appended from inside container!" >> persistent-data.txt

cat persistent-data.txt

exit

Step 6: Delete the container

docker rm bind-container

Step 7: Verify persistence on your host

cat persistent-data.txt

You should see both the original text and the appended text. The file persisted because it lives on your host machine!

```
willp@Will MINGW64 ~/Desktop/substack/Docker Project/docker-volume-test
ster)
$ docker rm bind-container
bind-container

willp@Will MINGW64 ~/Desktop/substack/Docker Project/docker-volume-test
ster)
$ cat persistent-data.txt
This file will survive container deletion!
Appended from inside container!

willp@Will MINGW64 ~/Desktop/substack/Docker Project/docker-volume-test
ster)
$
```

Part 2: Docker Volume Example (Docker-Managed Storage)

Part 2: Docker Volume Example (Docker-Managed Storage)

A Docker volume is managed entirely by Docker and stored in Docker's internal storage area.

Step 1: Create a Docker volume

```
docker volume create myvolume
```

Step 2: Run container with Docker volume

```
docker run -it --name volume-container -v myvolume:/data volume-test
```

Step 3: Write data to the volume

```
cd /data
echo "This data is stored in a Docker volume!" > volume-data.txt
echo "Docker manages this storage location." >> volume-data.txt
cat volume-data.txt
exit
```

Step 4: Delete the container

```
docker rm volume-container
```

Step 5: Verify the volume still exists

```
docker volume ls
```

A Docker volume is managed entirely by Docker and stored in Docker's internal storage area.

Step 1: Create a Docker volume

docker volume create myvolume

Step 2: Run container with Docker volume

docker run -it --name volume-container -v myvolume:/data volume-test

Step 3: Write data to the volume

cd /data

echo "This data is stored in a Docker volume!" > volume-data.txt

echo "Docker manages this storage location." >> volume-data.txt

cat volume-data.txt

exit

Step 4: Delete the container

docker rm volume-container

Step 5: Verify the volume still exists

docker volume ls

You should see myvolume in the list.

The Challenge: Prove Volume Persistence

The Challenge: Prove Volume Persistence

Now let's prove that the Docker volume data survived the container deletion:

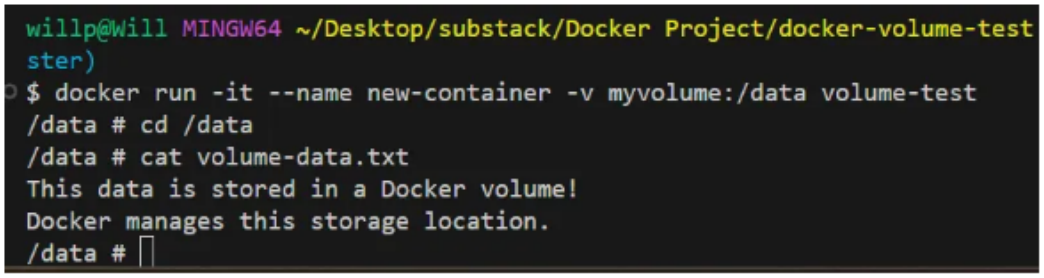
Step 1: Create a new container using the same volume

```
docker run -it --name new-container -v myvolume:/data volume-test
```

Step 2: Check if your data is still there

```
cd /data
cat volume-data.txt
```

Success! You should see the text you wrote earlier, even though you deleted the original container.



Key Differences Summary

Bind Mount:

- File/folder lives on your host machine
- You control the exact location
- Survives container deletion because it's on your host
- Good for development (editing files outside container)

Docker Volume:

- Managed entirely by Docker
- Stored in Docker's internal directory
- Survives container deletion because Docker manages it
- Good for production databases and persistent application data

Now let's prove that the Docker volume data survived the container deletion:

Step 1: Create a new container using the same volume

docker run -it --name new-container -v myvolume:/data volume-test

Step 2: Check if your data is still there

cd /data

cat volume-data.txt

Success! You should see the text you wrote earlier, even though you deleted the original container.

```
willp@Will MINGW64 ~/Desktop/substack/Docker Project/docker-volume-test
ster)
$ docker run -it --name new-container -v myvolume:/data volume-test
/data # cd /data
/data # cat volume-data.txt
This data is stored in a Docker volume!
Docker manages this storage location.
/data #
```

Key Differences Summary

Bind Mount:

- File/folder lives on your host machine
- You control the exact location
- Survives container deletion because it's on your host
- Good for development (editing files outside container)

Docker Volume:

- Managed entirely by Docker
- Stored in Docker's internal directory
- Survives container deletion because Docker manages it
- Good for production databases and persistent application data

- Can be shared between multiple containers

Both methods achieve data persistence, but they're managed differently and serve different use cases!