Introduction to Cloud Computing Report

Lab 5: Docker Core Concepts Tutorial

Name: Yin Yuang
Student ID: 202383930027
Class: Software Engineering Class 1

Introduction to Cloud Computing
(Autumn,2025)

Nanjing University of Information Science and Technology, China
Waterford Institute

I. Objective

- Understand the fundamental concepts of containers and how they differ from traditional virtual machines, including their advantages in application delivery and resource isolation.
- Learn the structure, build process, and layered mechanism of Docker images, enabling the creation and execution of container instances based on images.
- Become familiar with the purpose of Docker Compose and its value in orchestrating multi-container applications. Acquire the ability to write and use `docker-compose.yml` files.
- Master key runtime operations of containers, including publishing ports, overriding default configurations, persisting data with volumes, and sharing local files via bind mounts.
- Build a basic multi-container application and understand network communication and cooperation between services.
- Establish a foundation for future development using Docker Desktop on a Windows + WSL2 environment.

II. Content

1. Basic Concepts of Containers

- Learn the definition, runtime mechanism, and lifecycle of containers.
- Understand how containers achieve isolation while sharing the host OS kernel.
- Distinguish between container environments and full virtual machine environments.

2. Docker Image Structure and Usage

- Study the layered filesystem architecture of images.
- Use official images to run containers and understand the relationship between images and containers.
- (Optional) Learn image-building workflows through Dockerfiles.

3. Basic Operations of Docker Compose

- Understand the motivation behind using Docker Compose.
- Write and execute `docker-compose.yml` files for multi-service application management.
- Learn common commands such as `docker compose up` and `docker compose down`.

4. Publishing Container Ports

- Understand the mapping relationship between container internal ports and host ports.
- Use the `-p` flag to expose containerized applications for external access.

5. Overriding Container Defaults

- Use command-line parameters or Compose configurations to override the default `CMD` or `ENTRYPOINT`.
- Observe the impact of different override methods on container behavior.

6. Persisting Container Data

- Create and use Docker volumes.
- Understand the significance of data volumes being independent of the container lifecycle.

7. Sharing Local Files (Bind Mounts)

- Learn how to use bind mounts to map host directories directly into containers.
- Master the use of bind mounts for code hot-reloading during development.

8. Building and Running Multi-Container Applications

- Build combined applications such as a web service + database service using Docker Compose.
- Observe inter-service networking and cooperative execution.
- Use logs and management commands to monitor the overall system state.

III. Results

1.What is a Container?

This experiment focuses on the fundamental question of "What is a container in Docker?" Through reading documentation and examining provided examples, the following experimental results were obtained.

1.1 Problems Solved by Containers

The experiment confirms that containers effectively address several common issues encountered in software development, such as:

- Conflicts arising from applications requiring different versions of Python, Node.js, or databases.
- Inconsistent execution results when running the same application on different machines (the well-known "It works on my machine" problem).

Containers provide an isolated and consistent runtime environment, ensuring reproducibility across systems.

1.2 Definition and Characteristics of Containers

Based on the experimental documentation, the following characteristics were summarized:

- A container is a lightweight and isolated technology for packaging and running software.
- It provides an independent process-level runtime environment for applications.
- Components such as the frontend, backend, and database can run in separate containers.
- Each container bundles its required dependencies, configuration, and files, preventing issues caused by inconsistent host environments.

The examples used in the experiment demonstrated this separation clearly:

- The React frontend runs inside its own container.
- The Python backend runs in another independent container.
- The database service runs in a third container.

1.3 Comparison Between Containers and Virtual Machines

The experiment produced the following comparison between containers and virtual machines:

Category	Container	Virtual Machine
Startup Speed	Seconds	Minutes
Resource Usage	Lightweight, shares host kernel	Heavy, each VM includes a full OS
Isolation Level	Process-level	System-level
Typical Use Cases	Microservices, CI/CD	Multi-OS environments

The results demonstrate that containers start faster and consume fewer system resources compared to virtual machines.

1.4 Combined Use of Containers and Virtual Machines

The experimental material also emphasizes the complementary relationship between containers and virtual machines:

- Virtual machines can serve as a hosting platform for container runtimes.
- In cloud environments, Docker is often deployed on virtual machines, which then run multiple containers.

This layered architecture is widely used in modern cloud infrastructure.

1.5 Results of Running Containers via Docker Desktop (GUI)

First, the `welcome-to-docker` image was successfully located inside the Docker Desktop *Images* panel. The interface displays the image size, creation date, and availability, confirming that the Docker environment is capable of retrieving and managing container images properly.

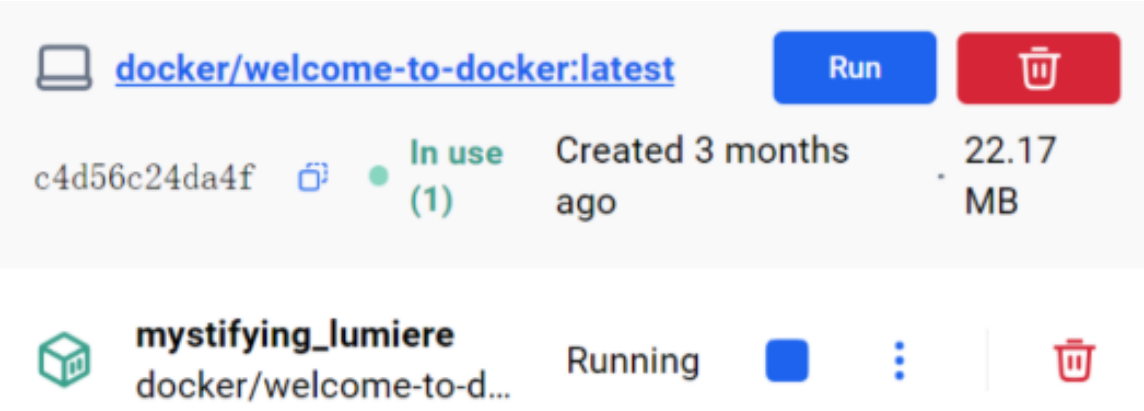

Figure 1: The Docker Desktop Images panel displaying the `welcome-to-docker` image.

Next, the container was configured using Docker Desktop's graphical interface before being launched.In this step, a container name was assigned, and port mapping was specified (host port `8080` mapped to container port `80`), ensuring that the application's web interface could be accessed through the local machine. This confirms that Docker Desktop provides an intuitive GUI for adjusting runtime parameters without relying on command-line operations.

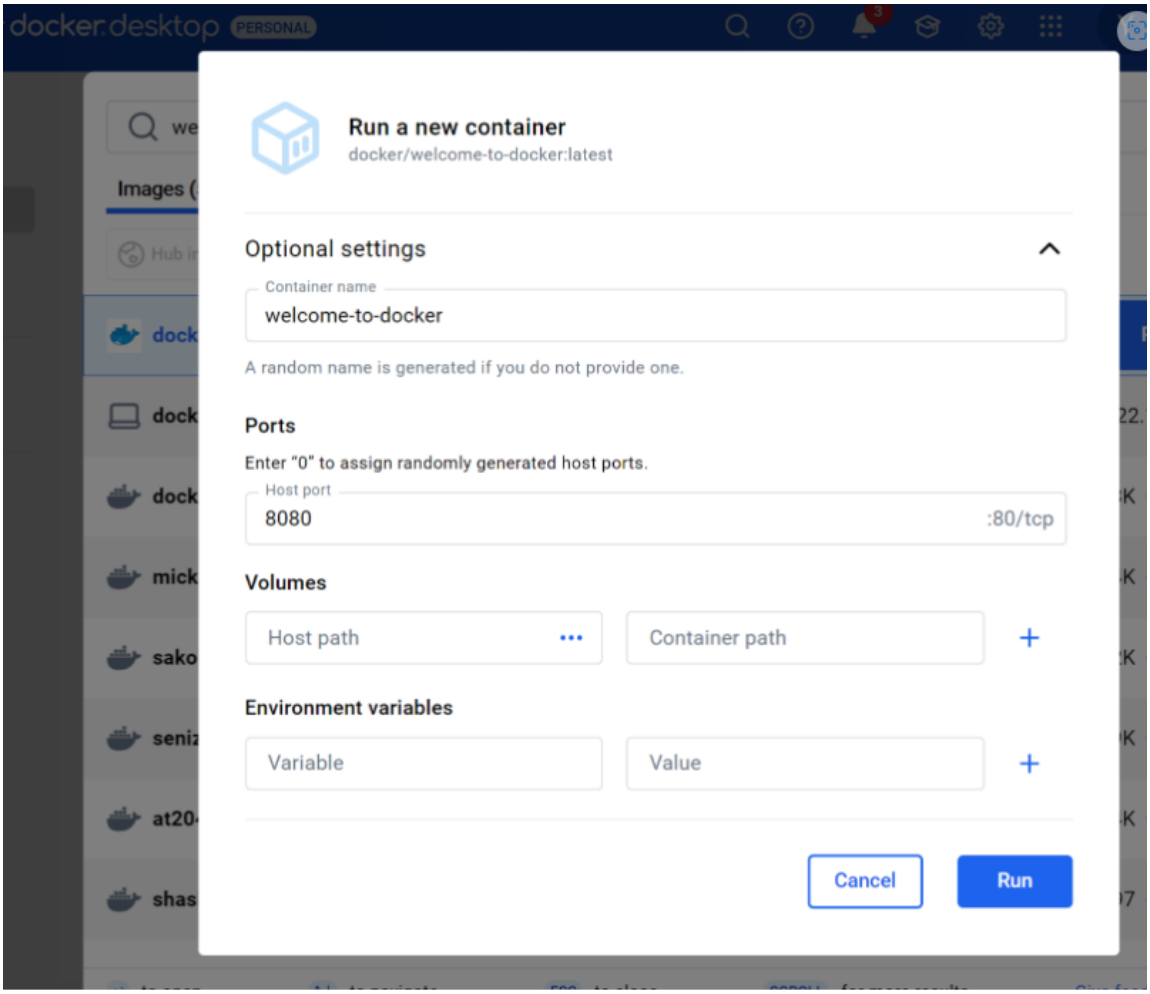

Figure 2: Configuration dialog for launching the container with port mapping and optional parameters.

After launching the container, navigating to `http://localhost:8080` successfully rendered the welcome page. The page displays the message “Congratulations!!! You ran your first container.” This confirms the container started correctly, the service inside the container was running, and the port mapping from host to container worked as expected.

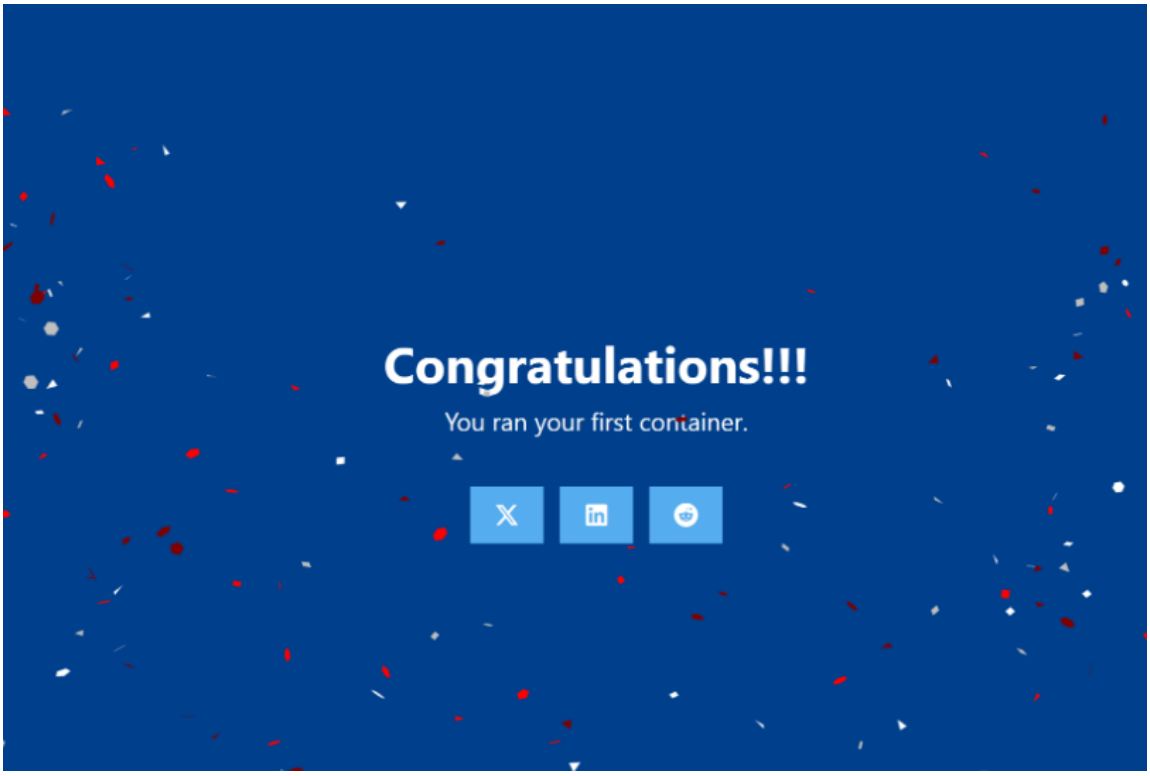

Figure 3: Browser output confirming successful container execution.

Finally, the running container was stopped using the Docker CLI command `docker stop <container_id>`. The successful return of the container ID indicates that the container was terminated correctly. This step demonstrates that both Docker Desktop and the CLI can be used interchangeably for container lifecycle management.

```
C:\Users\21534>docker stop 556fb88e819f
556fb88e819f
```

Figure 4: Stopping the running container using Docker CLI.

Overall, the experiment verified the correct functioning of Docker Desktop, including image retrieval, container configuration, container execution, web access to the running application, and lifecycle management via CLI. All operations executed successfully, confirming that the Docker environment is fully prepared for subsequent experiments involving image building, data persistence, and multi-container deployments.

2.What is an Image?

This part of the experiment focuses on the concept of Docker images and their role in containerized applications. Based on the official Docker documentation and experimental observations, the following results were obtained.

2.1 Definition of a Container Image

A container image is a standardized and portable package that contains everything required to run an application inside a container. It serves as a blueprint from which containers are created. According to the reference materials, a container image typically includes:

- Application files and binaries
- System libraries and dependencies
- Configuration files

In essence, an image defines the runtime environment of the application, ensuring consistent behavior across different systems.

2.2 Example: The PostgreSQL Image

The PostgreSQL image used in the experiment illustrates the structure and function of a typical container image. A PostgreSQL image contains:

- Database binaries such as `postgres`, `initdb`, and `psql`
- Default configuration files including `postgresql.conf` and `pg_hba.conf`
- Required system libraries such as `libpq`

When a container is started from this image, it behaves like a lightweight server with PostgreSQL preinstalled and ready for use. This demonstrates how images encapsulate a complete runtime environment.

2.3 Core Principles of Container Images

The experiment highlights two fundamental principles underlying Docker images.

2.3.1 Images Are Immutable

Once an image is built, it cannot be modified directly. Any change results in the creation of a new layer on top of the existing image. This immutability ensures reliability and prevents unintended alterations.

2.3.2 Images Are Layered

Docker images follow a layered filesystem structure rather than existing as a single monolithic file. Each layer corresponds to a specific change, operation, or addition. A typical multi-layer image may include:

Layer	Description
Layer 1	Base image (e.g., Ubuntu)
Layer 2	Install Python
Layer 3	Install Flask dependencies
Layer 4	Copy project source code
Layer 5	Set startup command `CMD["python", "app.py"]`

These layers stack to form the final complete image. This layered design improves efficiency and storage utilization because:

- Layers can be cached and reused.
- Different images can share common base layers.

2.4 Hands-On: Pulling and Analyzing a Docker Image

In this experiment, Docker Desktop was used to search for and inspect a container image from Docker Hub. The purpose was to verify that the environment supports image retrieval, metadata inspection, and layer visualization—essential steps before running a container.

The figure below shows the Docker Desktop *Images* panel after searching for the `welcome-to-docker` image. The interface displays the image name, tag, image ID, creation time, size, and available actions. The successful appearance of the image in the list confirms that Docker Desktop can correctly communicate with Docker Hub and pull remote images into the local system.

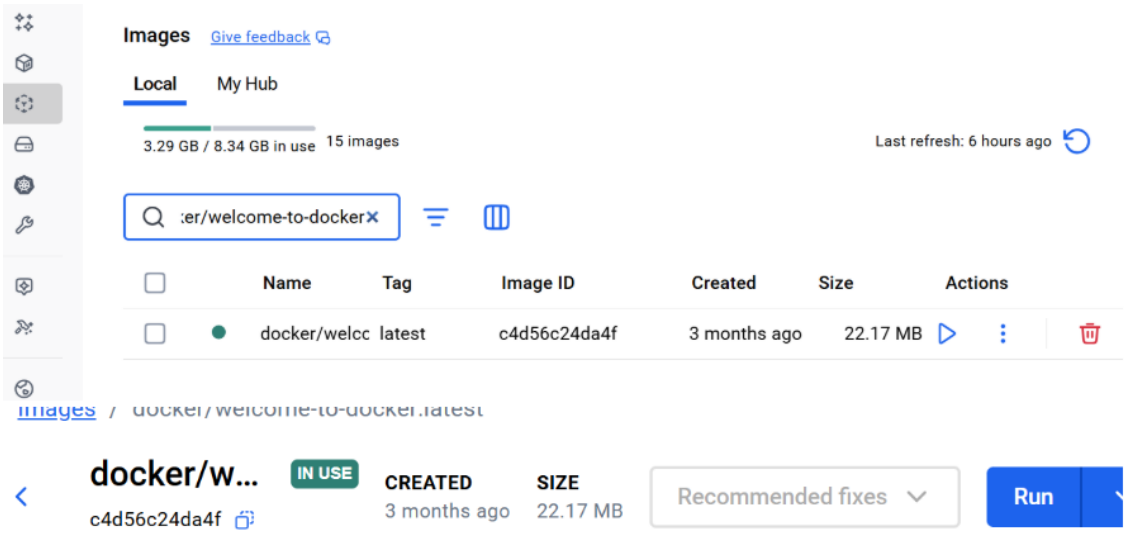

Figure 5: Searching and locating the `welcome-to-docker` image in Docker Desktop.

After selecting the image, Docker Desktop provides detailed information including its layered filesystem, as shown below. The panel lists each layer of the image—such as the base filesystem, configuration commands, and application dependencies—along with their corresponding file sizes. This validates the layered structure of Docker images, where each layer represents a specific change and contributes to the final immutable image. The screenshot also confirms that Docker Desktop can analyze vulnerabilities and metadata related to the image, although analysis has not yet been run.

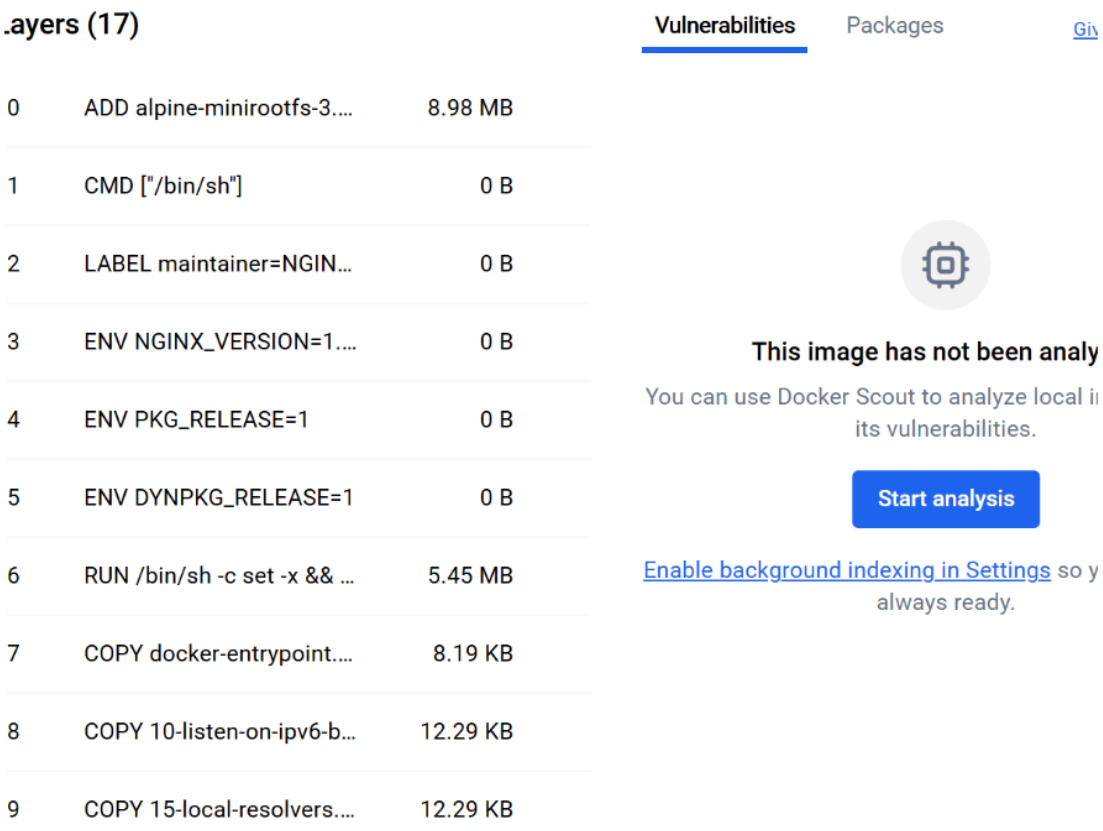

Figure 6: Detailed layer view of the container image, showing the immutable filesystem layers.

This experiment confirmed that Docker Desktop can successfully pull images from Docker Hub and display their essential metadata. By inspecting the `welcome-to-docker` image and its layered structure, we verified that

the environment correctly supports image retrieval and basic image analysis.

3.What is Docker Compose?

3.1 Background: Multi-Container Applications

Real-world applications often consist of multiple components such as databases, caching systems, message queues, backend services, and frontend applications. Each component is typically run in its own container, following the best practice: "Each container should do one thing, and do it well."

3.2 Purpose and Benefits of Docker Compose

Docker Compose is designed to simplify the deployment and management of multi-container applications. It allows all services and their configurations to be defined in a single YAML file, enabling:

- Automatic network creation and service linking
- Management of environment variables and port mappings
- Coordination of container lifecycles with a single command

3.3 Dockerfile vs. docker-compose.yml

File Type	Purpose
`Dockerfile`	Defines how to build a container image, including dependencies, source code, and startup command.
`docker-compose.yml`	Defines how to run multiple containers, specifying networks, volumes, environment variables, and inter-service dependencies.

3.4 Running a Multi-Container Application with Docker Compose

In this experiment, Docker Compose was used to deploy a multi-container application consisting of a Node.js web service and a MySQL database. The purpose of this exercise was to validate the ability of Docker Compose to orchestrate multiple services, manage networks, and coordinate container lifecycles with a single declarative configuration file.

Before deploying the multi-container application, the contents of the cloned project directory were examined. As shown below, the project includes essential components such as the `src` directory, a `Dockerfile`, and a `compose.yaml` file. The presence of the Compose file confirms that the application is designed to be orchestrated using Docker Compose, and the directory layout aligns with typical Node.js application structures.

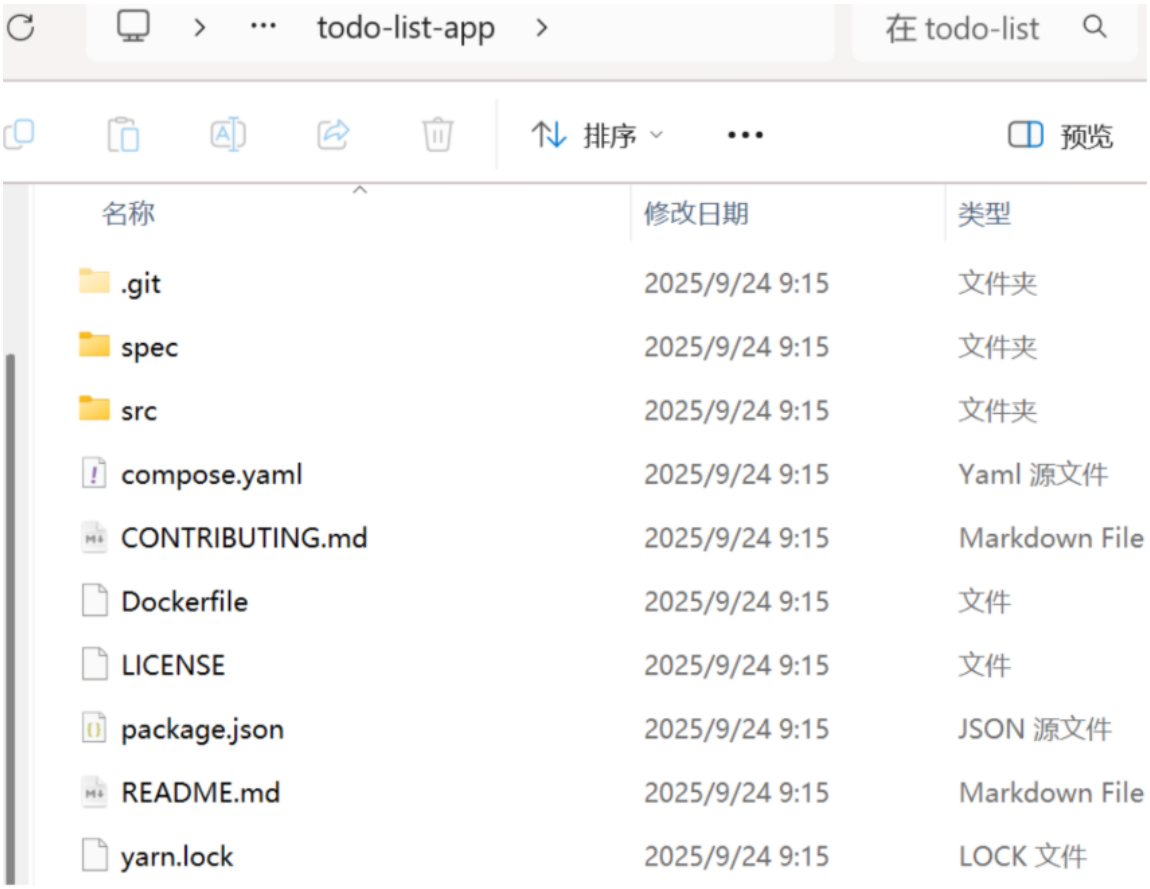

Figure 7: Project directory structure of the cloned Todo List application

Next, the `compose.yaml` file was reviewed to understand how the services are defined within this multi-container application. Figure below displays the configuration for both the `app` service (Node.js frontend) and the `mysql` service (database backend). The configuration specifies image sources, port mappings, volume mounts, working directories, and environment variables. Notably, the Node.js application connects to the MySQL container using Compose-managed networking, and a named volume (`todo-mysql-data`) is used to persist database data across container restarts. This confirms that the application leverages Compose features such as service linking, environment injection, and persistent storage.

```yaml
services:
  app:
    image: node:18-alpine
    command: sh -c "yarn install && yarn run dev"
    ports:
      - 127.0.0.1:3000:3000
    working_dir: /app
    volumes:
      - ./:/app
    environment:
      MYSQL_HOST: mysql
      MYSQL_USER: root
      MYSQL_PASSWORD: secret
      MYSQL_DB: todos

  mysql:
    image: mysql:8.0
    volumes:
      - todo-mysql-data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: todos

volumes:
  todo-mysql-data:
```

Figure 8: Service definitions in the `compose.yaml` file

After navigating into the project directory, the application was started using the `docker compose up -d` command. Figure below shows the terminal output, which indicates that both the Node.js service and MySQL service were created and started successfully. Docker Compose automatically created the required network, pulled missing images, and initialized the containers—demonstrating correct interpretation of the Compose configuration.

```
(base) PS D:\cloudcomputing\todo-list-app> docker compose up -d --build
[+] Running 3/3
 ✔Network todo-list-app_default    Created
 ✔Container todo-list-app-app-1    Started
 ✔Container todo-list-app-mysql-1  Started
```

Figure 9: Starting the multi-container application using `docker compose up -d`.

Once the stack was running, the web interface was accessed at `http://localhost:3000`. Figure below shows the To-Do List application successfully rendered in the browser. This confirms that the Node.js container is accepting HTTP requests and that it can communicate with the MySQL database container over the Docker network.

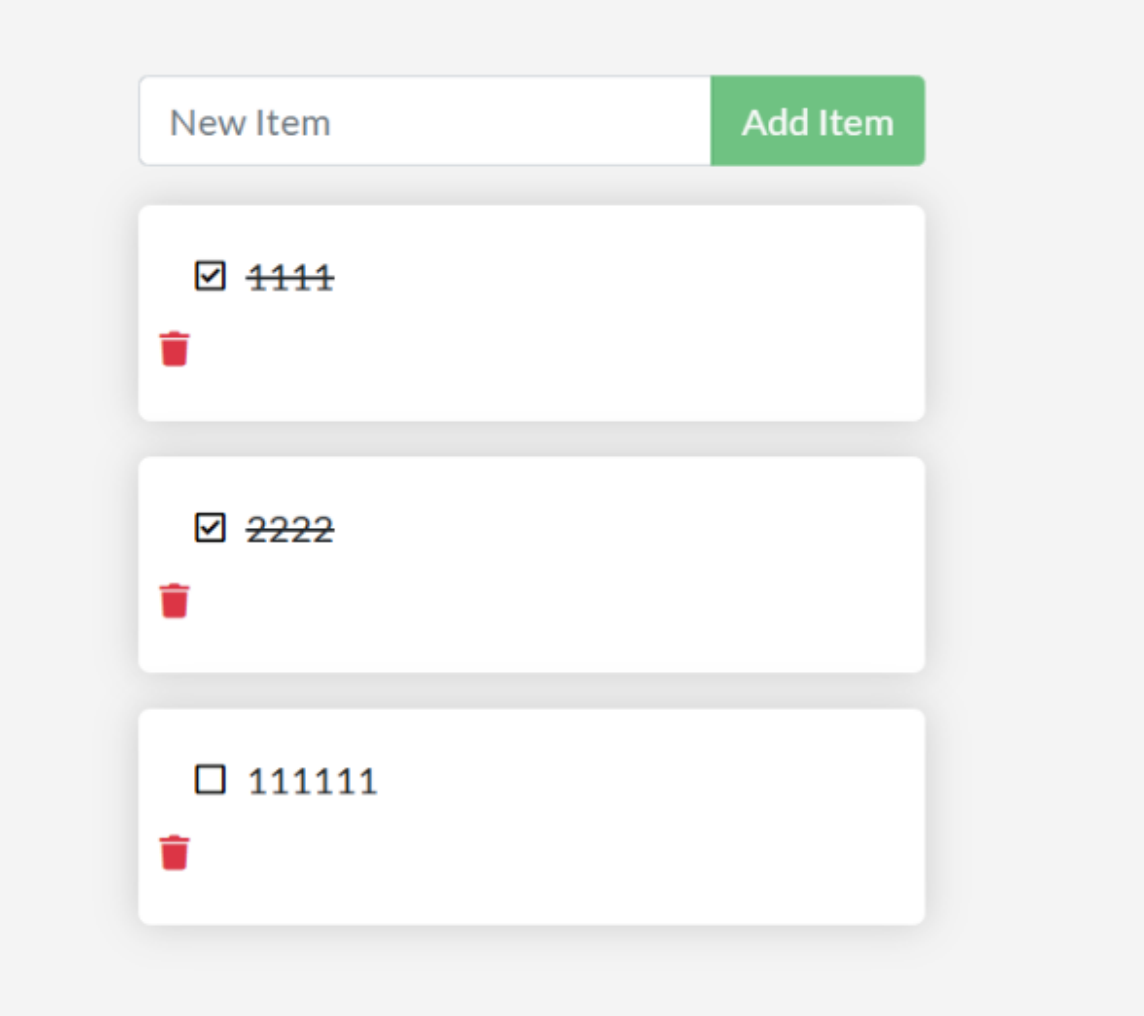

Figure 10: Browser showing the running To-Do List application at `localhost:3000`.

Docker Desktop was then used to visually inspect the running multi-container application. As shown below, the interface displays the two running services grouped together under the same Compose project. This confirms that Docker Desktop correctly interprets Compose deployments, allowing users to view logs, inspect container details, and monitor application health through the GUI.

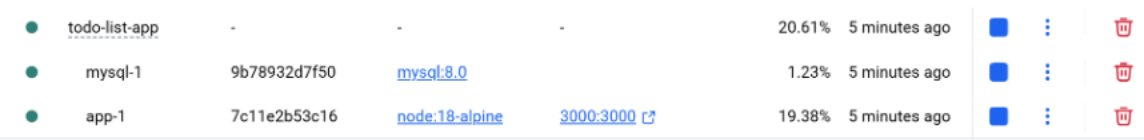

Figure 11: Docker Desktop displaying the running multi-container Compose project.

The environment was then removed using the `docker compose down` command. Figure below shows that all containers were stopped and removed, but persistent storage volumes remained intact. This behavior aligns with Docker Compose's default policy of preserving volumes unless explicitly removed.

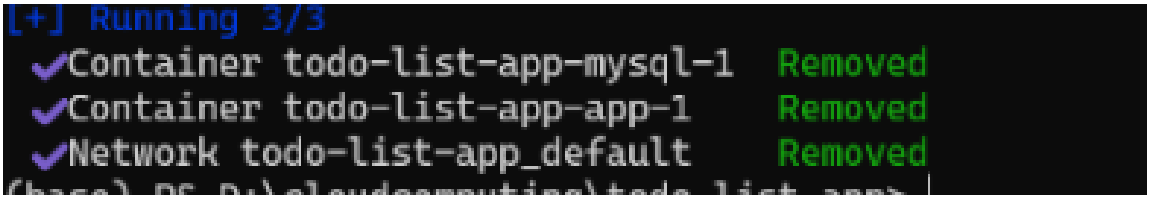

Figure 12: Stopping and removing the application using `docker compose down`

Finally, the volumes were removed using the `docker compose down --volumes` command . This deleted the MySQL data directory and ensured a fully clean teardown of the application state. The result verifies that Docker Compose supports complete cleanup when required, including containers, networks, and persistent data.

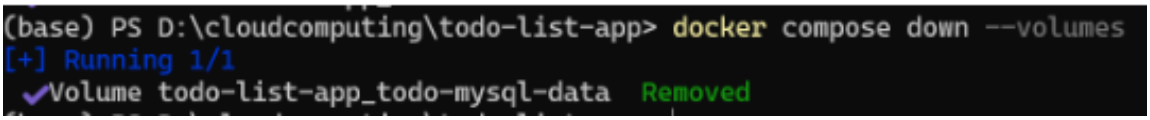

Figure 13: Removing containers, network, and persistent volumes using `docker compose down --volumes`

4.Publishing and exposing ports

4.1 Port Publishing Overview

Containers provide isolated runtime environments for different components of an application, such as frontends, backend APIs, and databases. Each container runs in its own network namespace, which isolates its network stack from the host machine and other containers. This isolation improves security, prevents dependency conflicts, and allows consistent behavior across environments.

However, the isolation also means that services inside the container are not directly accessible from the host or external networks. For example, a web application running in a container cannot be accessed through a browser on the host machine by default.

Port publishing is a mechanism that selectively exposes container ports to the host machine by creating a forwarding rule. This allows external traffic sent to a host port to be redirected to a specified port inside the container. Through this mechanism, containerized applications become reachable while still maintaining the benefits of isolation for other services.

4.2 Port Mapping Methods

Docker provides several ways to publish container ports, each suitable for different scenarios. The following table summarizes the most common methods:

Method	**Command Example**	**Host Port**	**Container Port**	**Description / Use Case**
Specific Port	`docker run -d -p 8080:80 nginx`	8080	80	Explicitly maps a container port to a specific host port. Allows predictable access (e.g., web apps on known ports). Suitable for production or services needing stable endpoints.
Ephemeral Port	`docker run -p 80 nginx`	Auto-assigned	80	Docker automatically assigns an available host port. Useful in development or testing when the exact host port is not important, avoiding conflicts with other services. The assigned port can be verified with docker ps.
Publish All Exposed	`docker run -P nginx`	Auto-assigned	As defined in EXPOSE	Publishes all ports declared with EXPOSE to ephemeral host ports. Convenient for multi-service containers or when multiple ports need exposure, especially in development and test environments.

Table 1: Summary of Docker port mapping methods and their use cases

Important Notes:

- By default, published ports are bound to all network interfaces (`0.0.0.0`), meaning that any system capable of reaching the host can access the service. Be cautious when exposing sensitive services such as databases.
- Port mapping does not break container isolation for other services; only the explicitly published ports are exposed.
- Ephemeral ports are useful for avoiding conflicts when running multiple containers simultaneously on the same host.
- For production deployment, it is recommended to use specific port mapping with firewall rules or reverse proxies to control external access.

4.3 Publishing container ports using both the CLI and Docker Compose

4.3.1 Running a Container Using Docker CLI

A new container was started using the following command:

```
docker run -d -p 8080:80 docker/welcome-to-docker
```

This command binds port 8080 on the host machine to port 80 inside the container, allowing external access to the web application.

```
C:\Users\yin>docker run -d -p 8080:80 docker/welcome-to-docker
73887f3be709572b818aca25a79842cf85392723ad52f0dc3231f12e4ec248d3

C:\Users\yin>docker ps
CONTAINER ID   IMAGE                     COMMAND                  CREATED         STATUS
PORTS                                    NAMES
73887f3be709   docker/welcome-to-docker   "/docker-entrypoint.…"   9 seconds ago   Up 8 seconds   0
.0.0.0:8080->80/tcp, [::]:8080->80/tcp   vibrant_shamir
```

Figure 14: Starting a container and publishing port 8080 using the Docker CLI

4.3.2 Verifying the Published Port

After starting the container, the published port was verified using the **Containers** view in the Docker Desktop Dashboard.

The dashboard displays the running container along with its published port mapping (`8080:80`) in the **Ports** column.

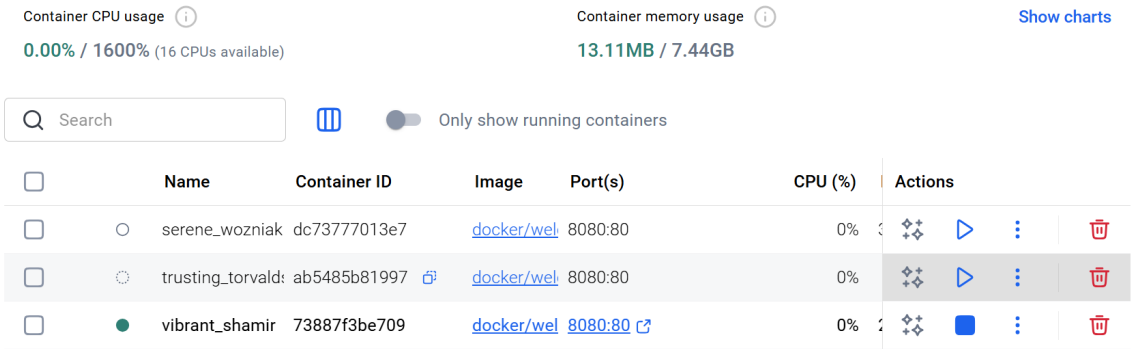

Figure 15: Docker Desktop Dashboard showing the published port mapping (8080:80)

The application was then accessed by opening the following URL in a web browser:http://localhost:8080

The web page loaded successfully, confirming that traffic sent to the host port was correctly forwarded to the container.

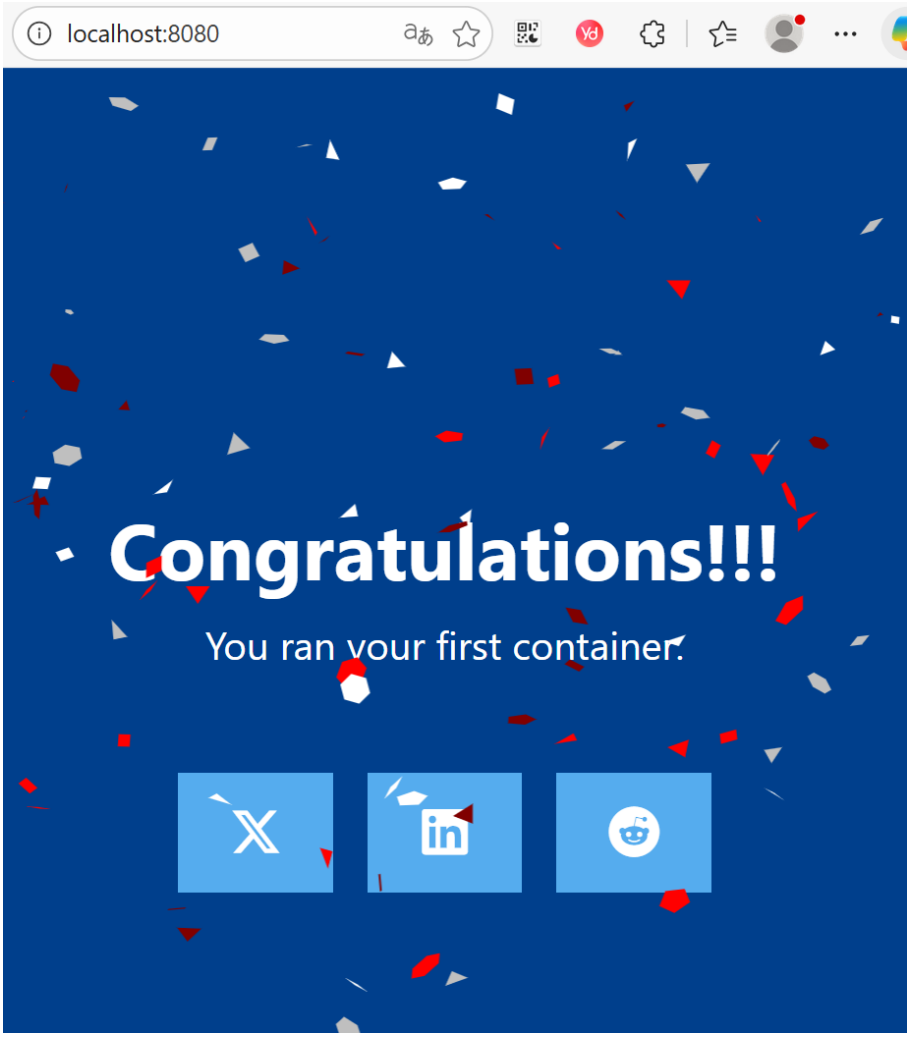

Figure 16: Web application accessed via `http://localhost:8080` after Docker CLI deployment

.

4.3.4 Running the Application Using Docker Compose

To launch the same application using Docker Compose, a new directory was created. Inside this directory, a file named `compose.yaml` was added with the following content:

```
services:
  app:
    image: docker/welcome-to-docker
    ports:
      - 8080:80
```

The `ports` configuration uses the same `HOST_PORT:CONTAINER_PORT` syntax as the `docker run` command. After navigating to the directory containing the `compose.yaml` file, the application was started using:

```
docker compose up
```

Docker Compose automatically pulled the image (if not already available), created the container, and published the specified port.

```
C:\Users\yin\docker-compose-demo>docker compose up -d
[+] Running 2/2
 ✔Network docker-compose-demo_default  Created
 ✔Container docker-compose-demo-app-1  Started

C:\Users\yin\docker-compose-demo>docker compose ps
NAME                       IMAGE                          COMMAND
        STATUS          PORTS
docker-compose-demo-app-1   docker/welcome-to-docker   "/docker-entrypoint.…"
 ago    Up 8 seconds    0.0.0.0:8080->80/tcp, [::]:8080->80/tcp
```

Figure 17: Launching the application using `docker compose up`

4.3.5 Result Verification

Once Docker Compose started successfully, the application was accessed again via:

`http://localhost:8080`

The same web page was displayed in the browser, indicating that the container was running correctly and the port publishing configuration was effective.

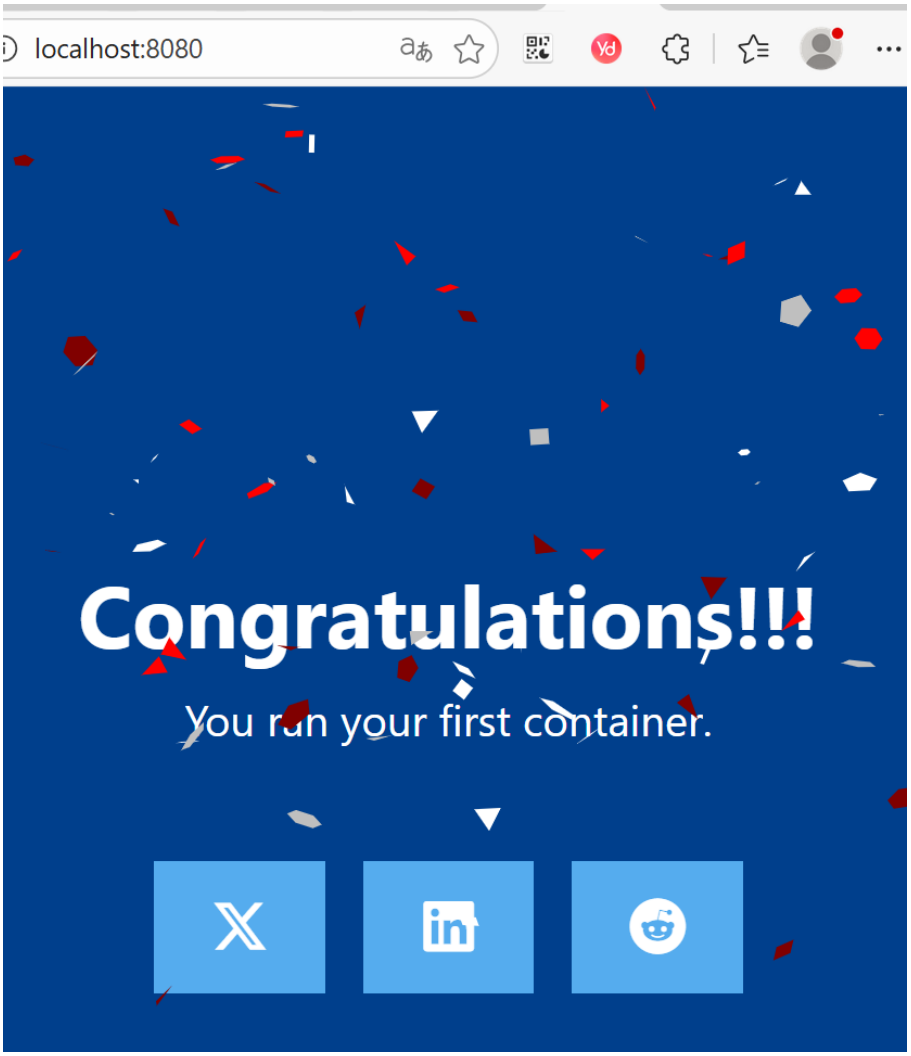

Figure 18: Web application accessed via `http://localhost:8080` after Docker Compose deployment

5.Overriding container defaults

5.1 Running Multiple Postgres Containers and Verification

Step 1: Start the First Postgres Container
Run the first Postgres container with default port mapping:

```
docker run -d -e POSTGRES_PASSWORD=secret -p 5432:5432 --name postgres1 postgres
```

Explanation:

- `-d`: Runs the container in detached mode.
- `-e POSTGRES_PASSWORD=secret`: Sets the Postgres superuser password.
- `-p 5432:5432`: Maps host port 5432 to container port 5432.
- `--name postgres1`: Names the container for easy identification.

Step 2: Start a Second Postgres Container
Run a second Postgres container on a different host port to avoid conflicts:

```
docker run -d -e POSTGRES_PASSWORD=secret -p 5433:5432 --name postgres2 postgres
```

Explanation:

- The container still uses the default internal Postgres port 5432.
- The host port is set to 5433 to ensure it does not conflict with the first container.
- Container naming helps distinguish between instances.

Step 3: Verify Both Containers are Running
Verify the containers are running:

```
C:\Users\yin>docker run -d -e POSTGRES_PASSWORD=secret -p 5432:5432 --name postgres1 postgres
3782865e64ea23cc91a510359eae3045fcbe115d4bf8e9da1d5ad3f71ee55659

C:\Users\yin>docker run -d -e POSTGRES_PASSWORD=secret -p 5433:5432 --name postgres2 postgres
bfb13255d60a2033fc863ece5f9968a5c0cc189b2821d95f68b40a71fc9a03e0

C:\Users\yin>docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED              STATUS              PORTS
                  NAMES
bfb13255d60a   postgres   "docker-entrypoint.s…"   8 seconds ago        Up 8 seconds        0.0.0.0:5433->5432/tcp, [::]
:5433->5432/tcp   postgres2
3782865e64ea   postgres   "docker-entrypoint.s…"   About a minute ago   Up About a minute   0.0.0.0:5432->5432/tcp, [::]
:5432->5432/tcp   postgres1
```

Figure 19: Two Postgres containers running simultaneously, verified with `docker ps` command

Verification Results: The verification screenshot shows:

- **postgres1**: Container running with host port 5432 mapped to container port 5432
- **postgres2**: Container running with host port 5433 mapped to container port 5432
- Both containers are in ”Up” status, confirming they are running properly

5.2 Running a Postgres Container in a Controlled Network

The objective of this experiment is to demonstrate how to run a Postgres container inside a custom Docker network. This allows for controlled communication between containers and isolates the database from external networks for better security and network management.

Step 1: Create a Custom Network

By default, containers connect to the default bridge network. For fine-grained control, we create a custom network:

```
docker network create mynetwork
```

Verify that the network has been created:

```
docker network ls
```

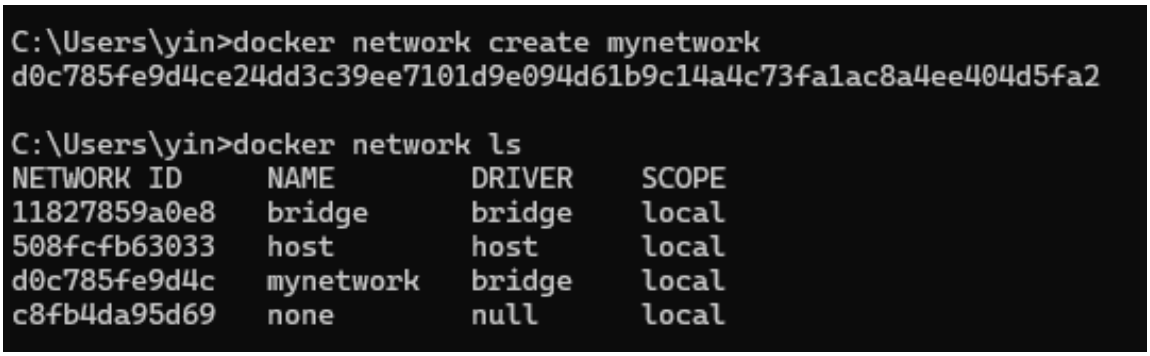

Figure 20: Listing all Docker networks to verify the creation of `mynetwork`

Step 2: Run Postgres Container in the Custom Network

Start the Postgres container and connect it to the custom network:

```
docker run -d -e POSTGRES_PASSWORD=secret -p 5434:5432 --network mynetwork postgres
```

Explanation of the command:

- `-d`: Run the container in detached mode.
- `-e POSTGRES_PASSWORD=secret`: Set the password for the Postgres superuser.
- `-p 5434:5432`: Map the host port 5434 to the container port 5432.
- `--network mynetwork`: Connect the container to the custom network `mynetwork`.

Figure 21: Starting Postgres container connected to the custom network `mynetwork`

Step 3: Verify Network Connection

To confirm that the container is attached to the custom network:

```
docker network inspect mynetwork
```

Figure 22: Inspecting the custom network to verify the Postgres container connection

5.3 Managing Container Resources

Run a Container with Resource Limits

Run a Postgres container with CPU and memory restrictions:

```
docker run -d -e POSTGRES_PASSWORD=secret --memory="512m" --cpus=".5" postgres
```

Explanation:

- `--memory="512m"`: Limits the container to use a maximum of 512 MB of RAM.
- `--cpus=".5"`: Limits the container to use a maximum of half a CPU core.
- `-e POSTGRES_PASSWORD=secret`: Sets the Postgres superuser password.
- `-d`: Runs the container in detached mode.

Figure 23: Starting a Postgres container with CPU and memory limits applied

5.4 Override the default CMD and ENTRYPOINT in Docker Compose

The objective of this experiment is to demonstrate how to run multiple instances of the Postgres database on the same host machine using Docker. This involves mapping each container to a unique host port to avoid conflicts, allowing simultaneous operation of multiple database instances.

Step 1: Create Docker Compose File

Create a `compose.yml` file with the following content:

```
services:
  postgres:
    image: postgres:18
    entrypoint: ["docker-entrypoint.sh", "postgres"]
    command: ["-h", "localhost", "-p", "5432"]
    environment:
      POSTGRES_PASSWORD: secret
```

Explanation of key parameters:

- `entrypoint`: Overrides the container's default entrypoint. Ensures the Postgres initialization script runs correctly.
- `command`: Overrides the default CMD to provide custom startup arguments (hostname and port).
- `environment`: Sets environment variables, such as Postgres password authentication.

Step 2: Start the Service

Bring up the Postgres service defined in Docker Compose:

```
docker compose up -d
```

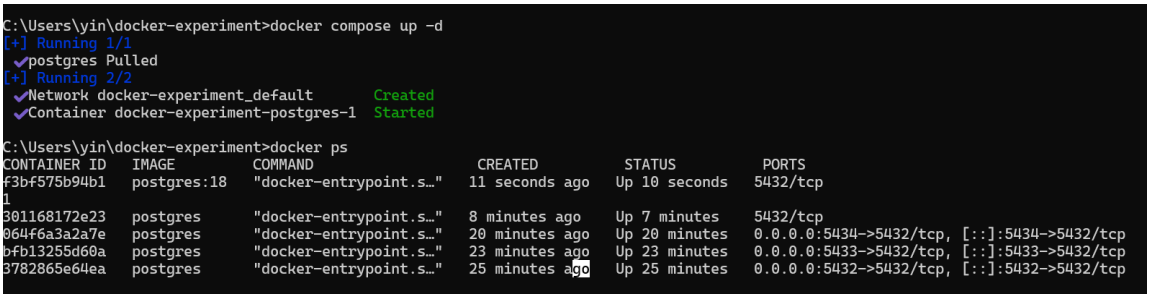

Figure 24: Starting the Postgres container with overridden CMD and ENTRYPOINT using Docker Compose

Step 3: Enter Container and Connect to Database

Connect to the Postgres database inside the container:

```
psql -U postgres
```

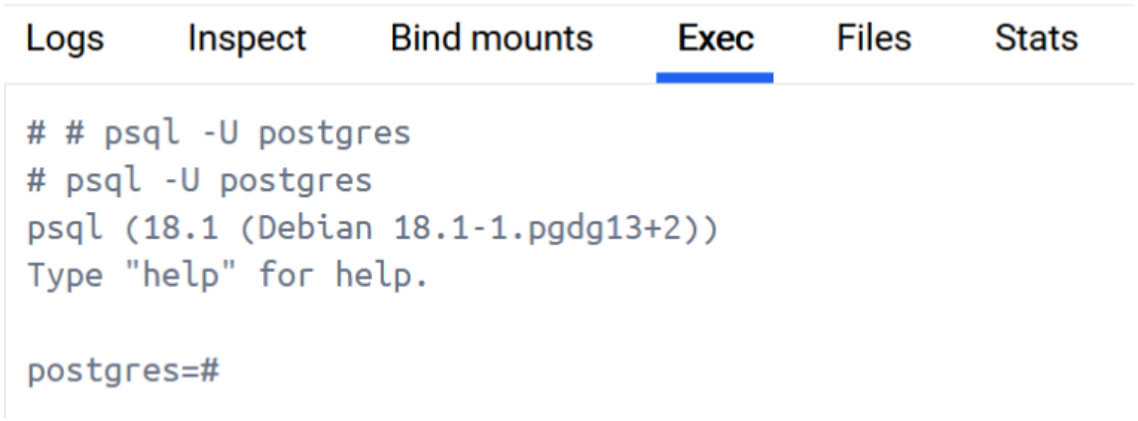

Figure 25: Entering the container shell and connecting to Postgres using psql

Note: Inside the container, password may not be required due to trust authentication. However, connections from other hosts or containers will require the password set via `POSTGRES_PASSWORD`.

6.Persisting container data

This part of the experiment focuses on Docker's persistent storage mechanisms. According to the Docker documentation, a **volume** is a storage method that allows data to persist beyond the lifecycle of a single container. Conceptually, a volume can be viewed as a shortcut or symbolic link from inside the container to a persistent storage location outside it.

In the experiment, a new volume was created and attached to a container, allowing data to remain intact even after the container was removed. This demonstrates the role of volumes in providing long-term, container-independent storage.

6.1 Create PostgreSQL Container with Volume

Start a PostgreSQL container with a named volume for data persistence:

```
docker run --name=db -e POSTGRES_PASSWORD=secret -d  -v postgres_data:/var/lib/postgresql postgres:18
```

Explanation:

- `--name=db`: Names the container for easy identification.
- `-e POSTGRES_PASSWORD=secret`: Sets the PostgreSQL superuser password.
- `-d`: Runs the container in detached mode.
- `-v postgres_data:/var/lib/postgresql`: Mounts a volume named `postgres_data` to the container's PostgreSQL data directory.
- `postgres:18`: Uses PostgreSQL version 18 image.

```
C:\Users\yin\docker-experiment>docker run --name=db -e POSTGRES_PASSWORD=secret -d -v postgres_data:/var/lib/postgresql
postgres:18
7c14a8a51ffb2c878460114d9a6f8640d61dd0ef9f05eb5bbfb5ec859fe81168

C:\Users\yin\docker-experiment>docker exec -ti db psql -U postgres
psql (18.1 (Debian 18.1-1.pgdg13+2))
Type "help" for help.

postgres=# CREATE TABLE tasks (
    id SERIAL PRIMARY KEY,
    description VARCHAR(100)
);
INSERT INTO tasks (description) VALUES ('Finish work'), ('Have fun');
CREATE TABLE
INSERT 0 2
postgres=# SELECT * FROM tasks;
 id | description
----+-------------
  1 | Finish work
  2 | Have fun
(2 rows)
```

Figure 26: Creating PostgreSQL container with volume and initializing database with test data

The container is successfully created with container ID: `7c14a8a51ffb2c878460114d9a6f8640d61dd0ef9f05eb5bbfb5ec859f`
Inside the container, a `tasks` table is created and populated with two records: (1, 'Finish work') and (2, 'Have fun').

6.2 Stop and Remove Container

Stop and delete the container while preserving the volume:

```
docker stop db
docker rm db
docker ps -a
```

```
C:\Users\yin\docker-experiment>docker stop db
db

C:\Users\yin\docker-experiment>docker rm db
db

C:\Users\yin\docker-experiment>docker ps -a
CONTAINER ID   IMAGE         COMMAND                  CREATED       STATUS       PORTS
                      NAMES
f3bf575b94b1   postgres:18   "docker-entrypoint.s…"   3 hours ago   Up 3 hours   5432/tcp
                      docker-experiment-postgres-1
301168172e23   postgres      "docker-entrypoint.s…"   3 hours ago   Up 3 hours   5432/tcp
                      quirky_colden
064f6a3a2a7e   postgres      "docker-entrypoint.s…"   3 hours ago   Up 3 hours   0.0.0.0:5434->5432/tc
p, [::]:5434->5432/tcp   condescending_murdock
bfb13255d60a   postgres      "docker-entrypoint.s…"   3 hours ago   Up 3 hours   0.0.0.0:5433->5432/tc
p, [::]:5433->5432/tcp   postgres2
3782865e64ea   postgres      "docker-entrypoint.s…"   3 hours ago   Up 3 hours   0.0.0.0:5432->5432/tc
p, [::]:5432->5432/tcp   postgres1
```

Figure 27: Stopping and removing the PostgreSQL container while preserving the volume

The container named db is successfully stopped and removed. The `docker ps -a` command confirms that the db container is no longer present, while other PostgreSQL containers continue running independently.

6.3 Verify Volume Persistence

Check that the volume persists after container deletion:

```
docker volume ls
```

```
C:\Users\yin\docker-experiment>docker volume ls
DRIVER    VOLUME NAME
local     68f16b7468dd98181bb0cc36333905342de543e2e879bc7088f34fc621edac58
local     1813efdbb7019335b17a074b9f3a5359adfc168f42ec450997f39b0c116d966a
local     7838b7b344ae98aa42ad2915b6bf7ebe40d7057c1e688644fef5778b8c2019a6
local     b4baf5e0646b8568df8fc2594c3e6447bf27bda3c7435a39af8b4dcc58300bc5
local     e54cc3797986c3c454a8c8b7ea14248c076d0cbe5338f31203cd395d072b9577
local     postgres_data
```

Figure 28: Verifying that the `postgres_data` volume persists after container removal

The `docker volume ls` output shows that the `postgres_data` volume (last entry) remains intact alongside other volumes. This confirms that Docker volumes have independent lifecycles from containers.

6.4 Create New Container with Existing Volume

Start a new container using the same persisted volume:

```
docker run --name=new-db -d -v postgres_data:/var/lib/postgresql postgres:18
```

Note: The `POSTGRES_PASSWORD` environment variable is omitted because it's only required during initial database bootstrap. The existing volume already contains the database configuration.

```
C:\Users\yin\docker-experiment>docker run --name=new-db -d -v postgres_data:/var/lib/postgresql postgr
es:18
141287df1b347be0eb7a5b1ef7a611124dde2f3cba3758834891b3ff31fc4c46

C:\Users\yin\docker-experiment>docker exec -ti new-db psql -U postgres -c "SELECT * FROM tasks"
 id | description
----+-------------
  1 | Finish work
  2 | Have fun
(2 rows)
```

Figure 29: Creating new container with existing volume and verifying data recovery

A new container named `new-db` is created with container ID: `141287df1b347be0eb7a5b1ef7a611124dde2f3cba3758834891b3`. The `SELECT * FROM tasks` query successfully retrieves both records from the previous container, confirming complete data recovery.

6.5 View Volume in Docker Desktop

Access Docker Desktop to view volume details and statistics:

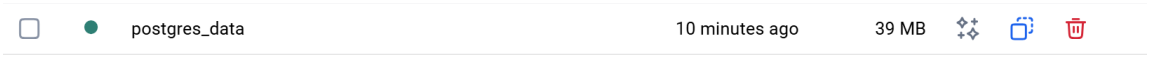

Figure 30: Viewing volume details in Docker Desktop interface

Results: Docker Desktop shows the `postgres_data` volume with the following properties:

- **Age**: 10 minutes (since creation)
- **Size**: 39 MB (PostgreSQL data storage usage)
- **Interface**: Provides management options including edit, copy, and delete

6.6 Cleanup - Remove Container and Volume

Clean up the testing environment by removing the container and volume:

```
docker rm -f new-db
docker volume rm postgres_data
```

Explanation:

- `docker rm -f new-db`: Force stops and removes the `new-db` container.
- `docker volume rm postgres_data`: Removes the `postgres_data` volume.

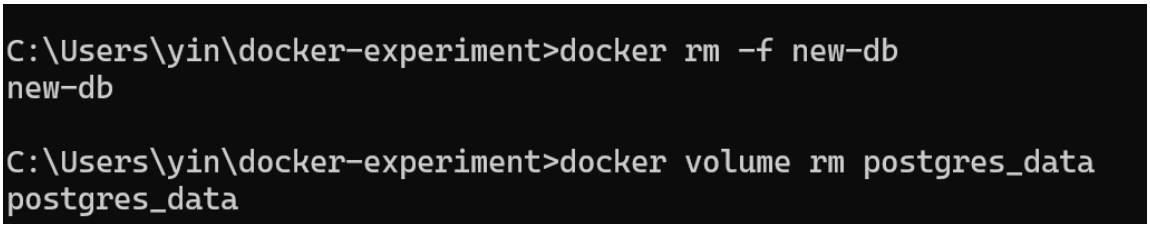

Figure 31: Cleaning up the testing environment by removing container and volume

Results: Both the `new-db` container and `postgres_data` volume are successfully removed, completing the full volume lifecycle demonstration.

The experiment confirms that Docker volumes provide reliable data persistence for stateful applications like databases, ensuring data integrity across container lifecycles.

7. Sharing Local Files with Containers

7.1 Overview

Containers provide filesystem isolation by default, which improves security and portability. However, many real-world applications require access to external files, such as configuration files, static web content, or persistent application data.

Storing such files directly inside container images is often undesirable, as it increases image size and may expose sensitive data when images are shared. To address this limitation, Docker provides mechanisms that allow controlled sharing of files between the host system and containers while maintaining isolation boundaries.

This experiment focuses on two Docker-supported file sharing mechanisms:

- Docker-managed **volumes**
- Host-managed **bind mounts**

7.2 Volumes vs Bind Mounts

Although Docker volumes and bind mounts both enable file sharing, they differ significantly in management, isolation, and intended use cases.

- **Docker Volumes** are managed entirely by Docker and stored in dedicated locations on the host. They are decoupled from the host directory structure and are well suited for persistent data such as databases and application state.
- **Bind Mounts** map a specific directory from the host filesystem directly into the container. This provides real-time synchronization and is particularly useful during development.

While bind mounts offer flexibility, they also introduce risks:

- Host file permission issues may prevent container access
- Accidental file deletion or modification on the host immediately affects the container

7.3 Experimental Design

This experiment investigates the behavioral differences between containers running with and without bind mounts. An Apache HTTP server (`httpd:2.4`) is used to observe how file persistence and synchronization behave across container lifecycle events.

7.3.1 Running an `httpd` Container Without a Bind Mount

The experiment begins by launching an Apache HTTP server container without any shared storage:

```
docker run -d -p 8080:80 --name my_site httpd:2.4
```

In this configuration, all website files reside exclusively inside the container's writable layer. Any changes made to the filesystem are lost once the container is stopped or removed.

Figure 32: Launching an Apache `httpd` container without a bind mount

—

7.3.2 Removing the Existing Container

To ensure a clean environment for the next test, the container is stopped and removed:

- `docker stop my_site`
- `docker rm my_site`

This step highlights the ephemeral nature of container filesystems when no external storage mechanism is used.

Figure 33: Stopping and removing the existing `my_site` container

7.3.3 Recreating the Container Using a Bind Mount

The container is then recreated with a bind mount that maps the current host directory to the Apache document root inside the container:

```
docker run -d --name my_site -p 8080:80  -v .:/usr/local/apache2/htdocs/ httpd:2.4
```

With this configuration, the container directly serves files from the host filesystem, enabling immediate reflection of file changes.

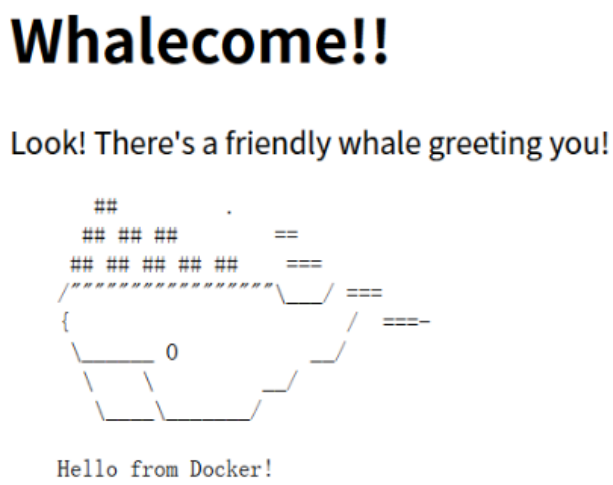

Figure 34: Launching the `httpd` container with a bind mount applied

7.3.4 Inspecting Container Files via Docker Desktop

Docker Desktop provides a visual interface to inspect container files. Figure 35 shows the container's filesystem as viewed through the `Files` panel.

Because a bind mount is used, the container's document root mirrors the host directory exactly, confirming that files are not copied but directly shared.

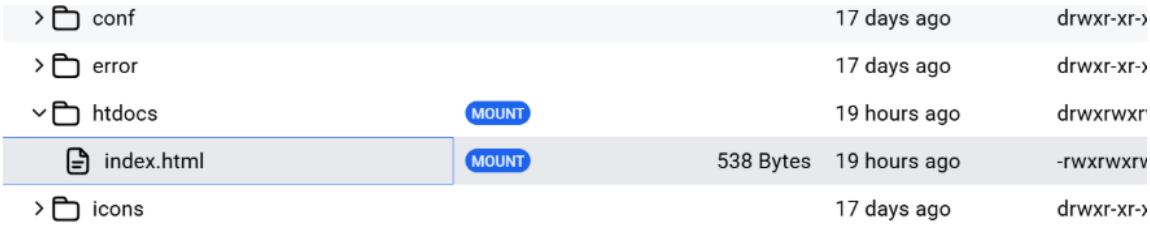

Figure 35: Viewing container files in Docker Desktop while using a bind mount

7.3.5 Demonstrating Real-Time Synchronization

To demonstrate real-time synchronization, a file (`index.html`) is deleted from the host directory. The same file disappears instantly from the container's filesystem view in Docker Desktop.

This behavior illustrates two fundamental properties of bind mounts:

- **Real-time bi-directional synchronization**: changes on the host or inside the container are immediately reflected on the other side
- **Shared filesystem linkage**: files are linked rather than duplicated, causing deletions and modifications to propagate instantly

Figure 36: Real-time file deletion synchronization between host and container

7.4 Docker Volumes for Persistent Storage

In contrast to bind mounts, Docker volumes provide a safer and more controlled approach to persistent storage.

A Docker volume is managed entirely by Docker and exists independently of any specific container. From the container's perspective, the volume behaves like a mounted directory, but its data remains intact even if the container is removed or recreated.

Figure 37 shows the creation of a Docker volume used to persist data across container lifecycles.

Figure 37: Creating a Docker volume to persist container data across restarts

8. Multi-container Applications

8.1 Source Code Acquisition

The example project was cloned from GitHub using the following command:

Figure 38 shows the successful cloning of the repository.

```
C:\Users\yin\Desktop>git clone https://github.com/dockersamples/nginx-node-redis
Cloning into 'nginx-node-redis'...
remote: Enumerating objects: 82, done.
remote: Counting objects: 100% (28/28), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 82 (delta 22), reused 18 (delta 18), pack-reused 54 (from 1)
Receiving objects: 100% (82/82), 76.62 KiB | 75.00 KiB/s, done.
Resolving deltas: 100% (26/26), done.

C:\Users\yin\Desktop>cd nginx-node-redis
```

Figure 38: Successful Git Repository Cloning

8.2 Manual Image Building

8.2.1 Building the Nginx Image

The Nginx image was built from the provided Dockerfile.

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

The build process is shown in Figure 39.

```
C:\Users\yin\Desktop\nginx-node-redis\nginx>docker build -t nginx .
[+] Building 1.0s (8/8) FINISHED                                                    docker:desktop-linux
 => [internal] load build definition from Dockerfile                                               0.0s
 => => transferring dockerfile: 143B                                                               0.0s
 => [internal] load metadata for docker.io/library/nginx:latest                                    0.0s
 => [internal] load .dockerignore                                                                  0.0s
 => => transferring context: 2B                                                                    0.0s
 => [1/3] FROM docker.io/library/nginx:latest@sha256:fb01117203ff38c2f9af91db1a7409459182a37c87cced5cb442d1d8fcc6  0.1s
 => => resolve docker.io/library/nginx:latest@sha256:fb01117203ff38c2f9af91db1a7409459182a37c87cced5cb442d1d8fcc6  0.0s
 => [internal] load build context                                                                  0.1s
 => => transferring context: 222B                                                                  0.0s
 => [2/3] RUN rm /etc/nginx/conf.d/default.conf                                                    0.4s
 => [3/3] COPY nginx.conf /etc/nginx/conf.d/default.conf                                           0.1s
 => exporting to image                                                                             0.2s
 => => exporting layers                                                                            0.1s
 => => exporting manifest sha256:4ecf0f55a6f9cb4dc13bef712c508a838f45e52dc40a153631df8a54d01a7691  0.0s
 => => exporting config sha256:8e2f22731696c3c0b91fcf14e7b3c062228c05c3275d7d631e54fbf5088f0b65    0.0s
 => => exporting attestation manifest sha256:7ba2ae4f91efcdc30b3f7faf80db7c2e3781af39473422c82e78a95a2c26aee4  0.0s
 => => exporting manifest list sha256:4f5871ae1b6f2e3403e34263bfa5389e687373a73c7795f82d7dfdae4ac5c9e3  0.0s
 => => naming to docker.io/library/nginx:latest                                                    0.0s
 => => unpacking to docker.io/library/nginx:latest                                                 0.0s
```

Figure 39: Nginx Image Build Process

8.2.2 Building the Web Application Image

Similarly, the Node.js web application image was built.

```
C:\Users\yin\Desktop\nginx-node-redis\web>docker build -t web .
[+] Building 3.6s (10/10) FINISHED                                                  docker:desktop-linux
 => [internal] load build definition from Dockerfile                                               0.0s
 => => transferring dockerfile: 177B                                                               0.0s
 => [internal] load metadata for docker.io/library/node:20                                         0.0s
 => [internal] load .dockerignore                                                                  0.0s
 => => transferring context: 2B                                                                    0.0s
 => [1/5] FROM docker.io/library/node:20@sha256:4b4e58e59c5e042928790c6fccd8ad16da6296bcc2e9924c56fba84a8e5ff662  0.1s
 => => resolve docker.io/library/node:20@sha256:4b4e58e59c5e042928790c6fccd8ad16da6296bcc2e9924c56fba84a8e5ff662  0.0s
 => [internal] load build context                                                                  0.1s
 => => transferring context: 17.30kB                                                               0.0s
 => [2/5] WORKDIR /usr/src/app                                                                     0.3s
 => [3/5] COPY package.json package-lock.json ./                                                   0.0s
 => [4/5] RUN npm ci                                                                               2.4s
 => [5/5] COPY ./server.js ./                                                                      0.1s
 => exporting to image                                                                             0.5s
 => => exporting layers                                                                            0.2s
 => => exporting manifest sha256:44ff66932ec3c3493068a53fcdb2655e15457093f156acd0263e83234bc5c924  0.0s
 => => exporting config sha256:6b876502a62daf34e7ca4fd576981f1a07437ec3cfb4bcfb00b2a1ebef9bd1f2    0.0s
 => => exporting attestation manifest sha256:acac323b1a382e96d2add2f96fe064e4aaaea3696a438f8bf4f380dcd131f3c2  0.0s
 => => exporting manifest list sha256:c5e4a75eadb907745111afe2cd0a8e22cb8a4a6d28256b7b7eb8eac56a4b804f  0.0s
 => => naming to docker.io/library/web:latest                                                      0.0s
 => => unpacking to docker.io/library/web:latest                                                   0.2s
```

Figure 40: Web Application Image Build Process

8.2.3 Image Verification

The successfully built images are verified using `docker images`, as shown in Figure 41.

```
C:\Users\yin\Desktop\nginx-node-redis\web>docker images
                                                                    i Info →  U  In Use
IMAGE                                ID             DISK USAGE   CONTENT SIZE   EXTRA
docker/welcome-to-docker:latest      c4d56c24da4f       22.2MB         6.03MB
nginx:latest                         4f5871ae1b6f        225MB         59.8MB
node:20                              4b4e58e59c5e       1.59GB          414MB
postgres:18                          38d5c9d52203        649MB          168MB    U
postgres:latest                      38d5c9d52203        649MB          168MB    U
web:latest                           c5e4a75eadb9       1.59GB          400MB
```

Figure 41: Docker Image List

8.3 Manual Container Deployment

8.3.1 Docker Network Creation

A custom bridge network was created to enable container communication.

```
C:\Users\yin\Desktop\nginx-node-redis>docker network create sample-app
6c9a94aff124b800c1d650117a2594b194a2a2d826a2db0169051a8beb832ef6
```

Figure 42: Docker Network Creation

8.3.2 Running Redis and Web Containers

Redis and two web containers were started manually. The running containers are shown in Figure 43.

```
C:\Users\yin\Desktop\nginx-node-redis>docker ps
CONTAINER ID   IMAGE         COMMAND                  CREATED              STATUS              PORTS
     NAMES
60722d1db9f2   nginx         "/docker-entrypoint.…"   10 seconds ago       Up 9 seconds        0.0.0.0:80->80/tcp, [::]:80->80/tcp
     nginx
a8f730b2a317   web           "docker-entrypoint.s…"   26 seconds ago       Up 26 seconds
     web2
987efaa11e55   web           "docker-entrypoint.s…"   47 seconds ago       Up 46 seconds
     web1
481df46b8d82   redis         "docker-entrypoint.s…"   About a minute ago   Up About a minute   6379/tcp
     redis
```

Figure 43: Running Containers Verification

8.3.3 Starting Nginx

Nginx was started with port mapping from container port 80 to host port 80.

```
C:\Users\yin\Desktop\nginx-node-redis>docker run -d  --name redis --network sample-app --network-alias redis redis
Unable to find image 'redis:latest' locally
latest: Pulling from library/redis
0c16123e68fb: Pull complete
1d9a95b931a8: Pull complete
30e994012175: Pull complete
2cedcff21aff: Pull complete
ae4ce04d0e1c: Pull complete
60f52e32d520: Pull complete
4f4fb700ef54: Pull complete
db91eab6df36: Download complete
5cfcc0f8ac58: Download complete
Digest: sha256:3906b477e4b60250660573105110c28bfce93b01243eab37610a484daebceb04
Status: Downloaded newer image for redis:latest
481df46b8d823b75a7a947d0d47d0e3620044adb527a79f23993b4ded723de33

C:\Users\yin\Desktop\nginx-node-redis>docker run -d --name web1 -h web1 --network sample-app --network-alias web1 web
987efaa11e55df543893c76b304279a06519d6aa79aca19bf0fcfe28d209f704

C:\Users\yin\Desktop\nginx-node-redis>docker run -d --name web2 -h web2 --network sample-app --network-alias web2 web
a8f730b2a3179de9a2d683f597a292dedab5bda20d3dd7a7c3ca27d515f05a6b

C:\Users\yin\Desktop\nginx-node-redis>docker run -d --name nginx --network sample-app  -p 80:80 nginx
60722d1db9f26be44eb5fecc2e4745383d5d21aaf1f367da3270828e6394c98d
```

Figure 44: Nginx Container Startup

8.4 Application Testing

8.4.1 Browser Access Test

The application was accessed via `http://localhost`. The page displayed the instance name and an incrementing visit counter stored in Redis.

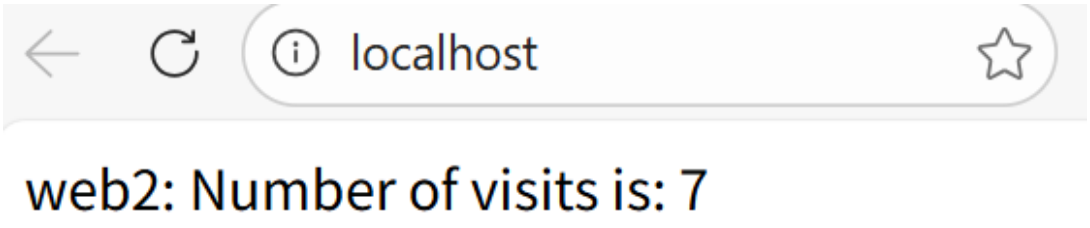

Figure 45: Browser Access Result

8.5 Docker Compose Deployment

8.5.1 Compose Configuration

The `compose.yml` file defines all services, networks, and dependencies in a centralized configuration.

```yaml
services:
  redis:
    image: redis
    ports:
      - '6379:6379'
  web1:
    restart: on-failure
    build: ./web
    hostname: web1
    ports:
      - '81:5000'
  web2:
    restart: on-failure
    build: ./web
    hostname: web2
    ports:
      - '82:5000'
  nginx:
    build: ./nginx
    ports:
    - '80:80'
    depends_on:
    - web1
    - web2
```

Figure 46: Docker Compose Configuration

8.5.2 One-Command Deployment

All services were built and started using a single command:

```
docker compose up -d --build
```

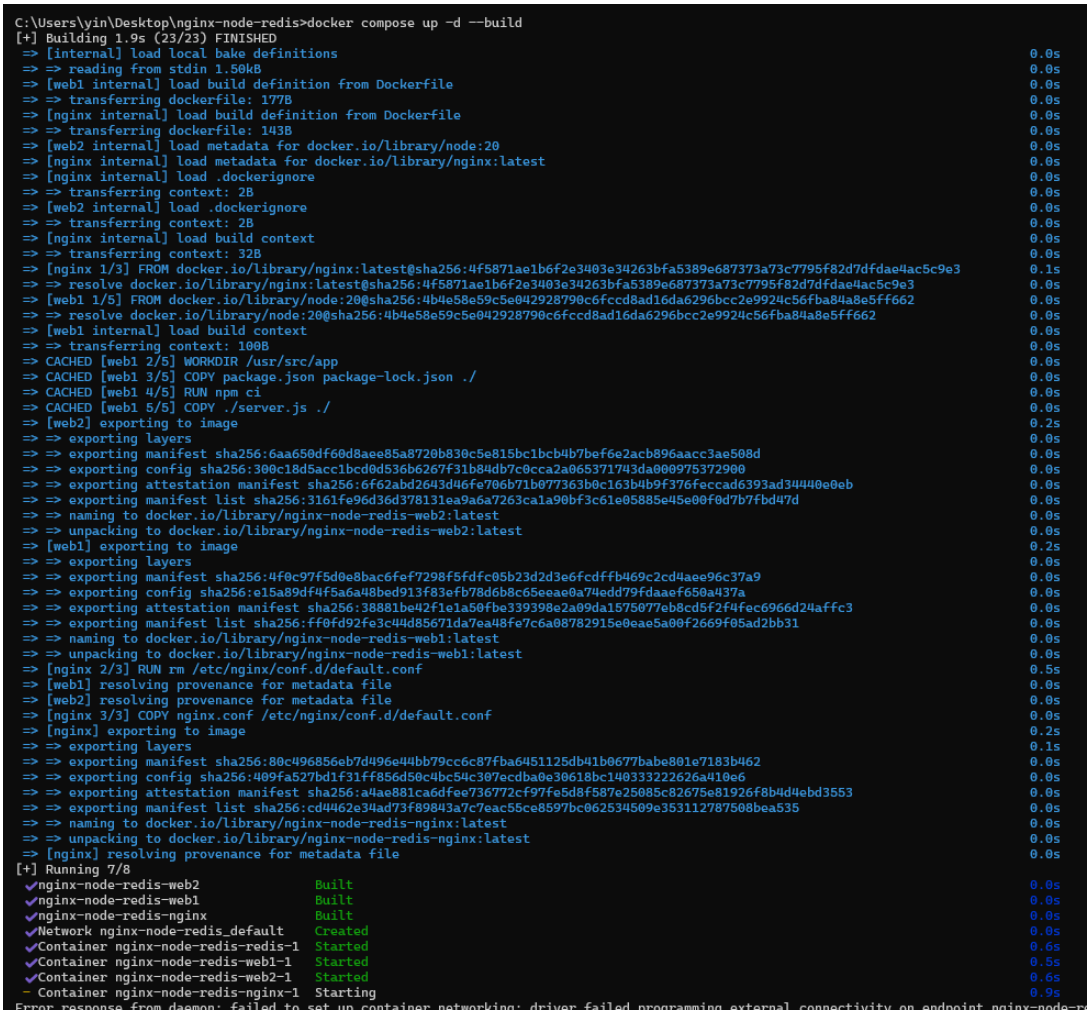

Figure 47: Docker Compose Startup

8.5.3 Compose Service Verification

The running services were verified using `docker compose ps`.

Figure 48: Docker Compose Service Status

IV. Conclusion

This experiment provided hands-on experience with core Docker concepts and tools, emphasizing containerization, multi-container management, and persistent data handling. By exploring the fundamentals of Docker images and containers, we gained a deeper understanding of how containers isolate applications while sharing the host OS kernel, offering significant advantages over traditional virtual machines.

Through Docker Compose, we learned to manage multi-service applications efficiently, reducing the complexity of running multiple containers and ensuring seamless inter-service communication. Additionally, the experiment highlighted the importance of container data persistence using volumes and bind mounts, which allows containers to maintain state across reboots or updates.

The ability to override default container behaviors and publish ports for external access were crucial skills in customizing containerized environments to fit specific application needs. Finally, the process of building and deploying multi-container applications demonstrated the flexibility and power of Docker in modern software development, enabling developers to easily set up and scale complex systems with minimal overhead.

Overall, this experiment reinforced the foundational knowledge required to effectively use Docker in both development and production environments, positioning us to build more scalable, portable, and efficient applications.