VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF INFORMATION TECHNOLOGY

FACULTY OF COMPUTER NETWORKS AND COMMUNICATIONS

NGUYEN KHANH LINH - 22520769

TRAN THIEN THANH - 22521367

PHAM THI CAM TIEN - 22521473

THAI NGOC DIEM TRINH - 22521541

PROJECT REPORT

NETWORK AND SYSTEM ADMINISTRATION

TOPIC: DEPLOY WEB APPLICATION WITH DOCKER CONTAINER

SUPERVISOR

MS. TRAN THI DUNG

HO CHI MINH CITY, 2024

ACKNOWLEDGMENTS

First, we would like to express our sincere gratitude to Ms. Tran Thi Dung - our project supervisor for the Network and System Administration course, who has directly guided and supported us throughout the process of implementing and completing this project.

With all our efforts, our group has completed the project according to the proposed plan and perfected it to the best of our ability. However, as we are still young and inexperienced, there may be shortcomings. We hope to receive feedback and guidance from our supervisor to further improve this project. We also thank all group members for their efforts in contributing and performing their work to the best of their ability to complete this project.

We sincerely thank you!

Ho Chi Minh City, November 2024

Group 6

SUPERVISOR'S COMMENTS

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. Overview

1.1. Introduction

Docker is an open platform that allows users to develop, transport, and run applications easily on an independent virtualization platform. It enables users to isolate applications from infrastructure to quickly distribute software through containers. By leveraging Docker's methods for shipping, testing, and deployment, users can significantly reduce the latency between code writing and application deployment.

Docker is considered an operating system specifically for containers. Similar to how virtual machines virtualize server hardware, containers virtualize the server's operating system. Once Docker is installed on each server, it provides basic commands that can be used to create, start, and stop containers. Notably, containers are portable, allowing users to move their applications from one environment to another without concerns about differences in operating systems or hardware configurations.

1.2. Components

To have a more comprehensive view of web application deployment using Docker, we need to understand Docker's components in detail:

- Docker Client: is the command-line interface for users to interact with Docker. Using Docker Client, users can send requests to Docker Daemon to perform operations such as creating, running, managing containers, etc.
- Docker Engine: is the core component of Docker. This is considered a software platform used for creating containers, managing containers on a computer or server. Docker Engine includes:
    - Docker Daemon: used to create and manage objects like images, containers, networks, volumes.
    - Docker API: is a controller for Docker Daemon, helping with specific tasks from docker client that Docker Daemon must perform.
    - Docker Client: uses Docker API to send commands to Docker Daemon.

- Docker Registry: is a cloud service capable of automating continuous tasks and allowing application sharing between machines. It allows users to perform operations like push and pull with images.
- Dockerfile: is a text file containing a set of instructions for software execution steps and used as a structure to build docker images.
- Docker Image: is an immutable file containing source code, libraries, dependencies, tools, and other files necessary for an application to run. Docker Images can be shared and deployed independently on different machines.
- Docker Container: is a lightweight run-time environment containing all necessary components of an application such as source code, libraries, and tools, ensuring the application can run independently and consistently across all environments. Notably, containers share system resources, making them very lightweight and helping connection and interaction operations occur quickly and more conveniently.
- Docker Compose: is a tool supporting the definition and running of multi-container applications. It can manage multiple containers simultaneously in production, staging, development, testing, and CI.

1.3. Operations

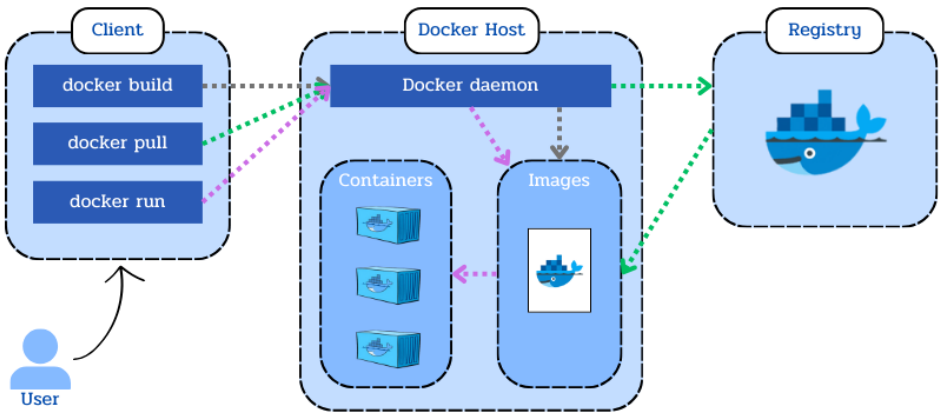

Figure 1.1. Docker operations

Docker Client and Docker Daemon can run on the same system, or Docker Client can connect to Docker Daemon remotely.

Docker Client sends requests to Docker Daemon to perform actions such as building containers, running containers, downloading images, etc., and the client can communicate with more than one daemon.

Docker Daemon manages docker objects such as images, containers, networks, etc., and listens for and executes Docker API requests like "docker build", "docker run", connecting with Docker registry to perform actions such as downloading or pushing Docker images.

After an image is successfully created, users can push it to repositories on docker using the "docker push" command to store in the Registry. Users can retrieve and use Docker images from the Registry by running "docker pull", "docker run" commands.

1.4. Writing Dockerfile

In Dockerfile, users first need to specify the base image that Docker will use to build the container by adding the following line: *FROM <image>*

- Example: FROM ubuntu:20.04 ⇒ The user selects Ubuntu version 20.04 as the base image.

Next, set up the working directory for the container using the command: *WORKDIR /path/to/directory*. This command specifies where subsequent commands will run.

- Example: WORKDIR /app ⇒ Subsequent commands will be executed in the /app directory.

Copy all files and directories from the user's project on their machine to the container image using the *COPY <source> <destination>* command.

- Example: COPY . . ⇒ Copies all files and directories from the current directory on the user's machine (where the Dockerfile resides) to the current working directory in the container.

Next, use *RUN <command>* to execute commands inside the container during the image-building process. This is used for installing software, configuring the environment, or performing any necessary setup steps before running the application.

- Example: RUN npm install -g yarn ⇒ Installs Yarn globally inside the container.

After that, users can inform Docker of the specific network ports the container will listen to at runtime using *EXPOSE <port>* (or *EXPOSE <port>/<protocol>*). However, when combined with the -p flag in the docker run command, these exposed ports can be mapped to ports on the host machine.

- Example: EXPOSE 80 ⇒ Indicates that the application inside the container will listen on port 80.

Finally, use *CMD ["executable", "param1", "param2"]* to specify the default command that will be executed when the container starts.

- Example: CMD ["node", "server.js"] ⇒ Instructs Docker to run the node server.js command when the container starts.

Comparison of Docker Deployment and Manual Deployment

Docker deployment	Manual deployment
1. Create a Dockerfile.	1. Set up the environment and required software (e.g., programming language, web server, database, and tools like npm, python3, etc.) on the developer's machine.
2. Build a Docker image from the Dockerfile.	2. Transfer the source code and install dependencies using tools like npm install, pip install, etc.
3. Run a Docker container from the Docker image on the developer's machine.	3. Manually start the application on the developer's machine using tools or direct commands.
4. Push the Docker image to a Docker registry for storage.	4. Source code and dependencies are typically copied directly from the developer's machine to the host without

	centralized storage.
5. Pull the Docker image from the Docker registry to the production machine.	5. Download the source code, set up the environment, and install required software on the production machine.
6. Run a Docker container from the Docker image on the production machine.	6. Manually start the application on the production machine using tools or direct commands.
7. Update the application by rebuilding the Docker image and pushing it to the registry. The production machine only pulls the new image and restarts the container.	7. Update the source code on the developer's machine, restart necessary services (e.g., web server, database), then pull the updated code to the production machine and rerun any setup commands as required.

Table 1.1. Comparison of Docker Deployment and Manual Deployment

1.5. Writing docker-compose.yml

The basic structure of a docker-compose.yml file looks as follows:

```
services:
  <service_name>:
    container_name: <container_name>
    restart: always
    build:
      context: <path_to_build_context>
    ports:
      - "<host_port>:<container_port>"
    volumes:
      - "<host_path>:<container_path>"
    depends_on:
      - <dependent_service>
```

Figure 1.2. Structure of a docker-compose.yml file

Explanation:

- services <service_name>: Define the services (containers) to be created.
- container-name: Specifies the name of the container for the service.
- restart: Configure the restart policy, e.g., always, on-failure, or no. In this example, it ensures the container always restarts if stopped.
- build: context: Point to the directory containing the Dockerfile and related files for building the Docker image.
- ports: Map ports between the host and the container in the format <host_port>:<container_port>.
- volumes: Link directories or volumes between the host and container in the format <host_path>:<container_path>.
- depends_on: Specify other services that this service depends on to ensure proper startup order.

Chapter 2. **Implementation**

2.1. Architecture

2.1.1. Objectives:

- Basic: Successfully deploy a static website - frontend only.
- Advanced: Successfully deploy a complete website with frontend, backend, and database.

2.1.2. Static web

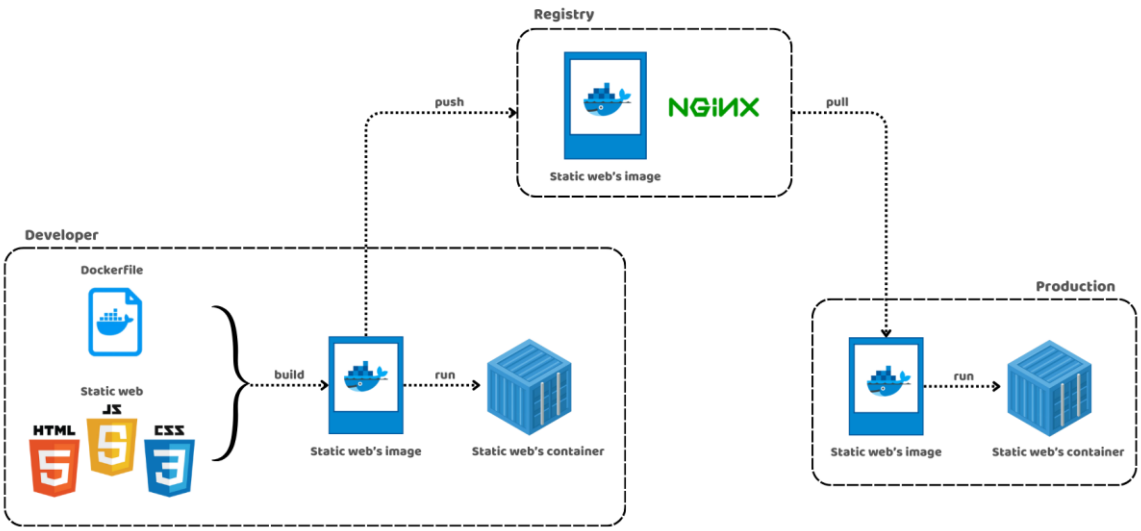

Figure 2.1. Static website architecture

- Registry: A storage and distribution location for Docker images that helps users manage their images and allows other users to easily search and use shared images. Docker Hub is the default registry used by Docker, though users can also use private registries.
- Developer: Here, the Developer creates a Dockerfile to specify how to build a Docker image from the static web source code they developed. Then, they proceed to build an image from the created Dockerfile. After creating the image, the Developer can run containers on their local machine. Alternatively, they can push the newly created Docker image to a Docker registry for storage and sharing.
- Production: When Production wants to use an image stored on the Registry for deployment elsewhere, they only need to pull that image to their machine using the "docker pull" command, then run containers from the retrieved image.

2.1.3. Web Application

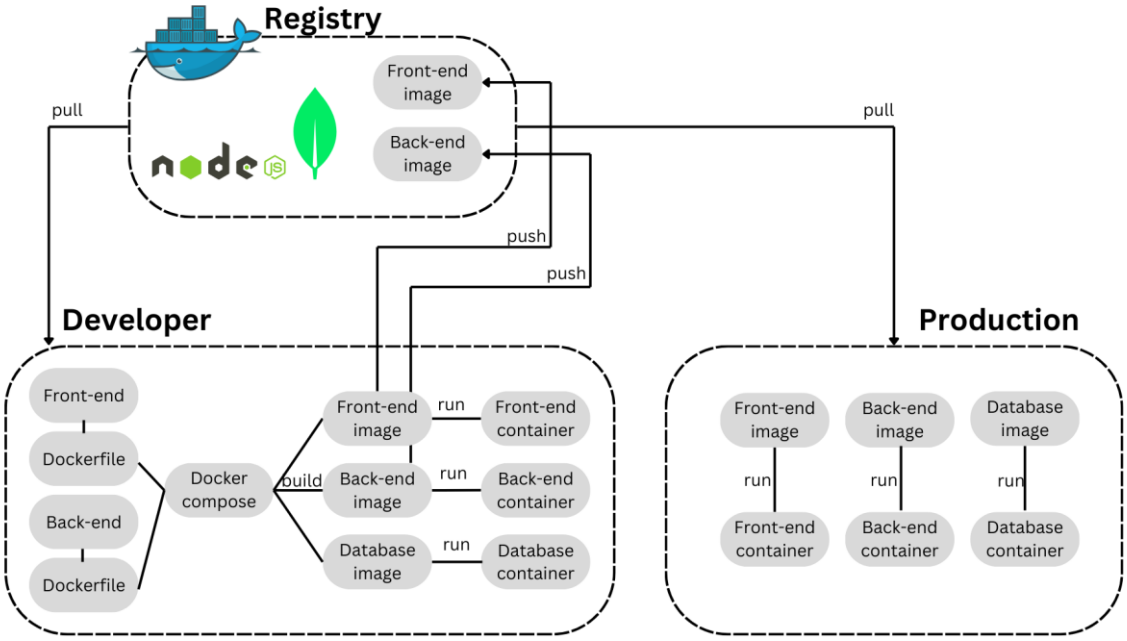

Figure 2.2. Web application architecture

- Registry: A storage location for Docker images after they are created, facilitating easy sharing and deployment. Docker Hub is the most popular registry, but other registries like GitLab Container Registry or Azure Container Registry can also be used.
- Developer: Dockerfiles for both frontend and backend services are created to define the necessary environments. Notably, these Dockerfiles pull Node.js images from Docker Hub as base images for both frontend and backend since both run on JavaScript and require appropriate runtime environments to ensure proper application functionality. Docker Compose is used to define and manage the entire multi-container application, including the MongoDB database. Instead of having to create, run, and connect each container with separate Docker commands, Docker Compose allows defining all containers and their related configurations in a single docker-compose.yml file. When Docker Compose runs, it pulls the MongoDB image from the registry to create the database container. After building frontend, backend, and database images, logging into Docker Hub and pushed to the Registry for storage and deployment purposes. When testing is needed, these images can be run to create corresponding containers, thereby running the web application on Docker.
- Production: This is the official application deployment environment where images from the Registry are pulled and run to serve end users.

2.2. Installation

Our team chose to run Docker on Windows environment, therefore WSL2 needs to be installed to provide the foundation for Docker Desktop to function as intended.

2.2.1. Installing WSL2

- First, launch the command prompt.
- Next, enter the command **wsl --install** to install.
- After the installation process is complete, restart and enter some required information to finish.

2.2.2. Install Docker Desktop

- Step 1: Visit the Docker homepage and download the Docker Desktop Installer.exe file
- Step 2: Run the installer with administrator privileges.
- Step 3: Keep the default options and click OK.

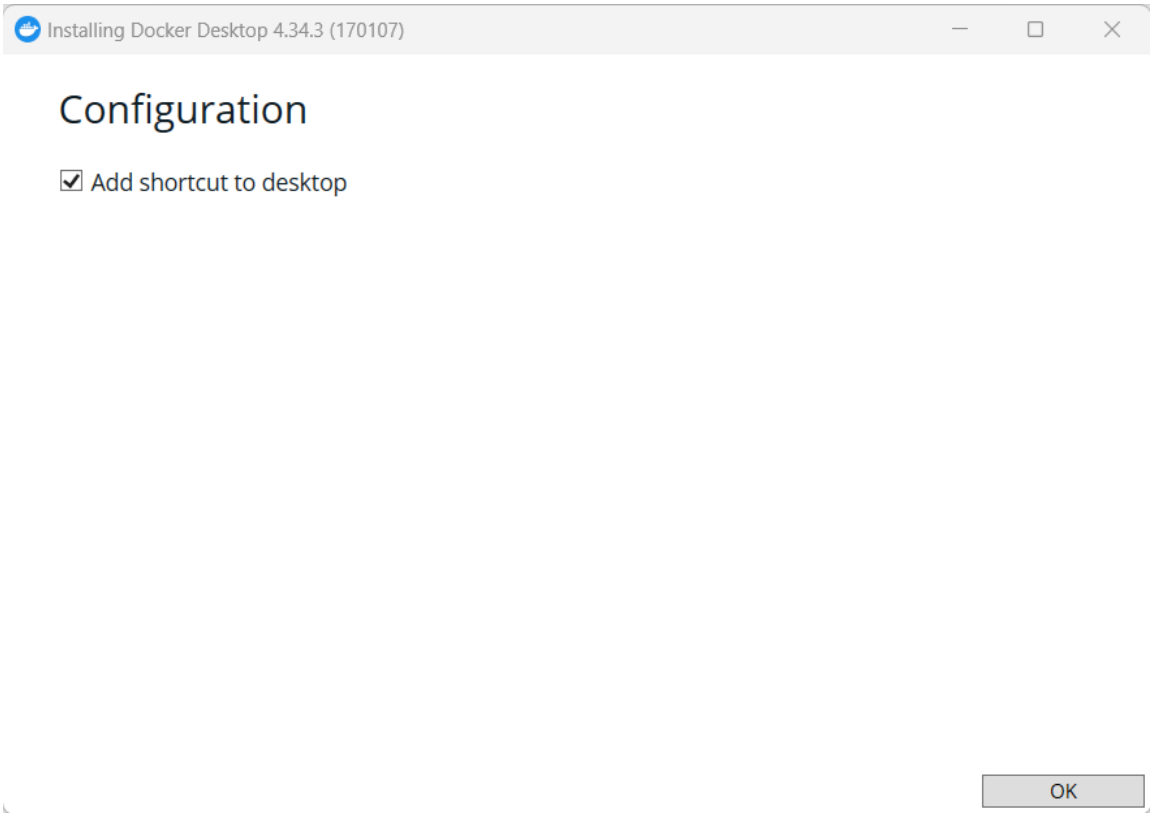

Figure 2.3. Docker Desktop installation interface (1)

- Step 4: Wait for the installer to install Docker for the system.

- Step 5: After the installer finishes running, click "Close and restart" to complete the installation process.

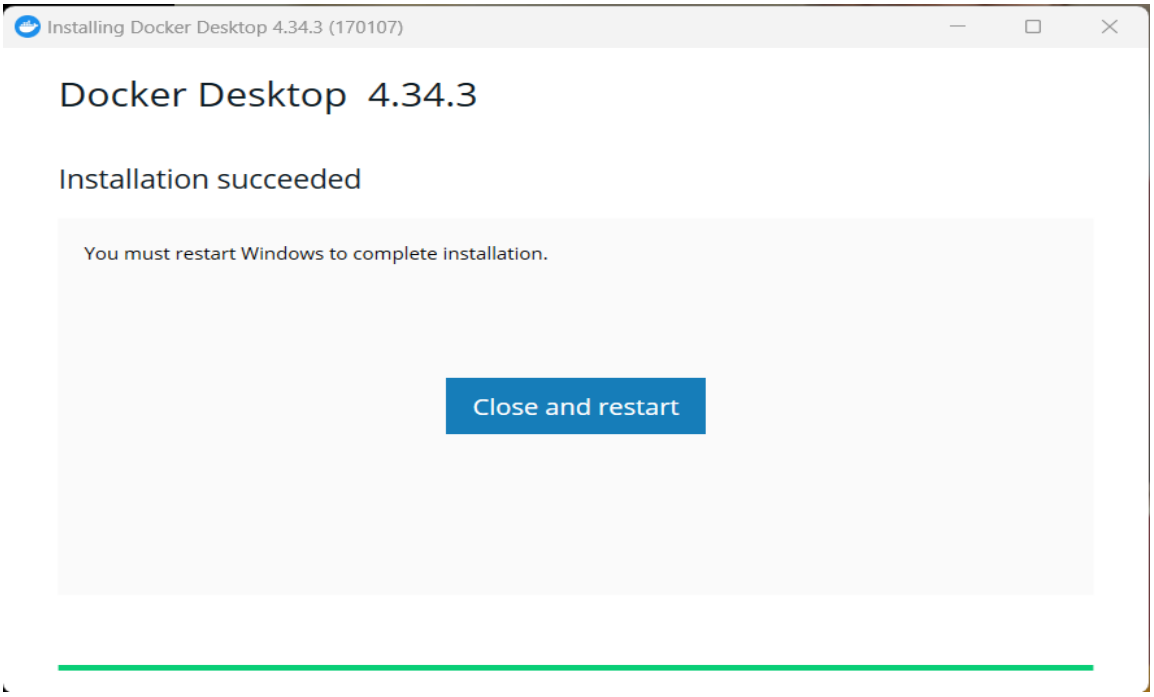

Figure 2.4. Docker Desktop installation interface (2)

- Step 6: Click "Accept" to accept the Docker Subscription Service Agreement. Docker Desktop has been successfully installed.

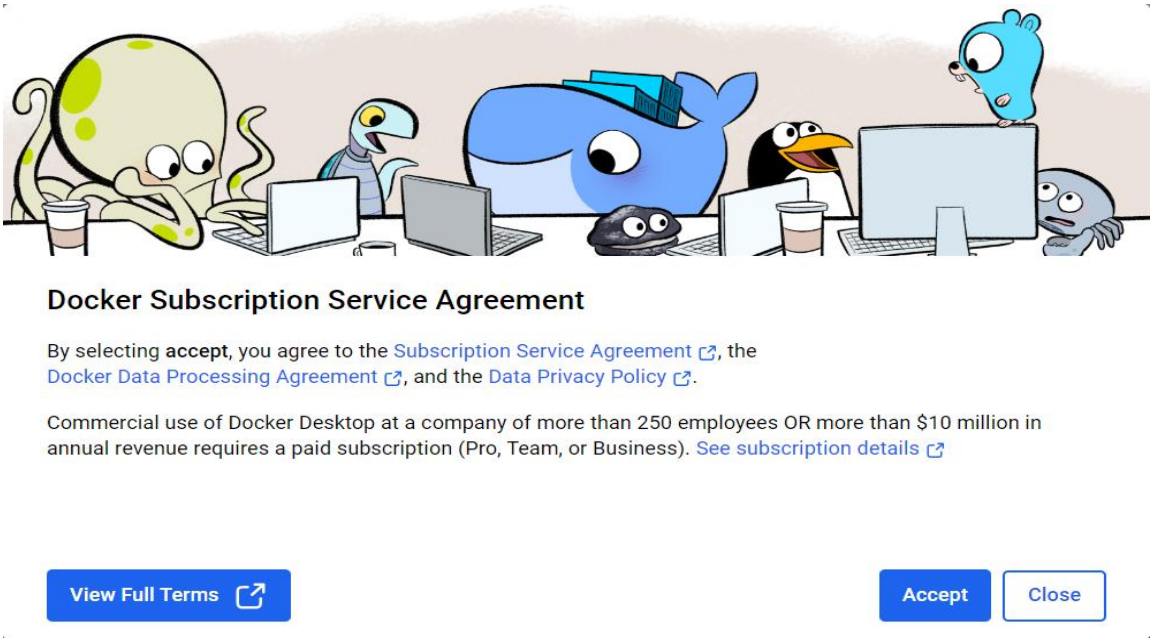

Figure 2.5. Docker Desktop Subscription Service Agreement interface

2.3. Configuration

2.3.1. Basic Configuration

Source code: https://github.com/solivaquaant/landing-page

To deploy a basic static webpage, our team referenced a static website template using only basic languages like HTML, CSS, and JavaScript. Our team started deploying the website by writing a Dockerfile, with contents as shown below:

```
FROM nginx:alpine

COPY ./ /usr/share/nginx/html

EXPOSE 80
```

Figure 2.6. Dockerfile for deploying a static website

Our team will explain the detailed contents of the written Dockerfile:

- FROM nginx:alpine - Sets the base docker image as nginx with the alpine
- COPY . /usr/share/nginx/html - Copies all files and directories from the current directory to nginx's root directory.
- EXPOSE 80 - Specifies that the container will listen for connections on port 80.

After writing the Dockerfile, our team began building the docker image using the command **docker build -t landing-page .**

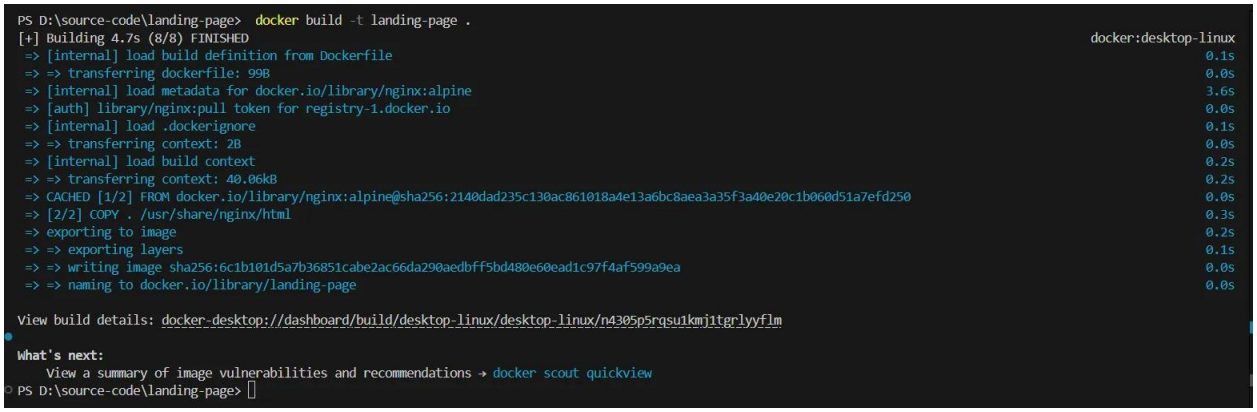

Figure 2.7. Creating Docker image

Explanation:

- docker build: command to create a new image based on instructions defined in the Dockerfile
- -t: used to name and tag the new image
- landing-page: the name given to the image being built. Users can optionally tag the image using the syntax <image_name>:<tag_name>; if no tag is specified, it defaults to "latest"
- .: specifies the path to the current directory where Docker will look for the Dockerfile

After completing the image build, our team verified the created image using **docker image ls** to list all available images on the machine.

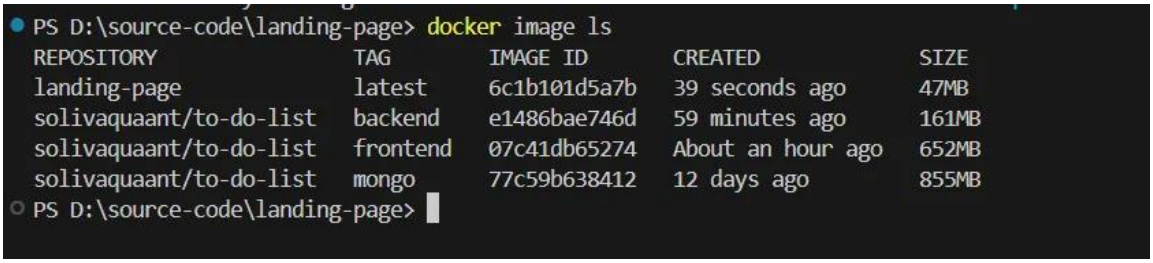

Figure 2.8. List of existing images on the computer

Next, our team launched a container with the command **docker run -d --rm -p 80:80 landing-page**.

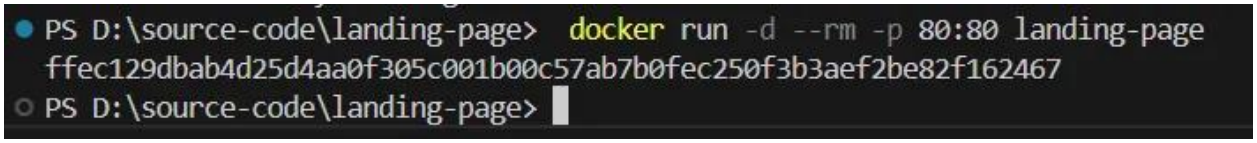

Figure 2.9. Running Docker container

The command includes these components:

- docker run: command to create and run a container
- -d: runs in detached mode - container runs in the background without occupying the current command window
- -p 80:80: opens port mapping from host to container. Here, requests from the host on port 80 will be forwarded to port 80 of the container

The website can now be accessed at http://localhost:80/

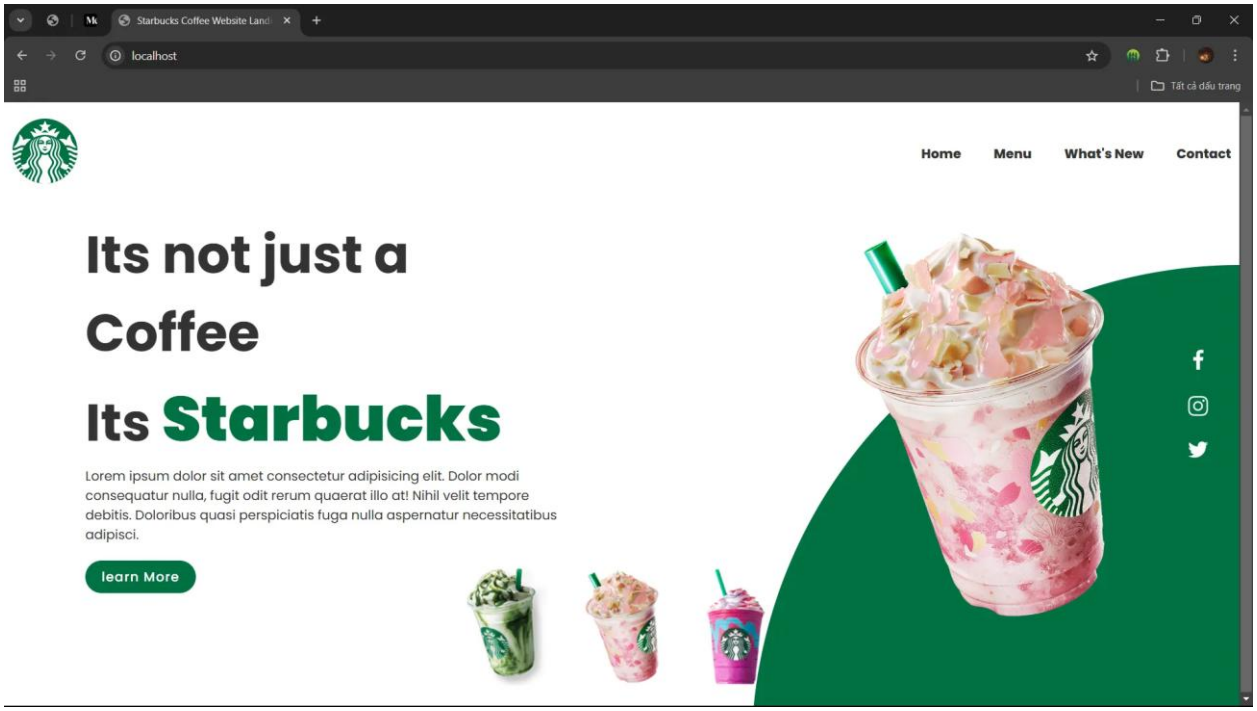

Figure 2.10. The deployed website interface

To facilitate the "transport" and deployment on other computers, our team will push the project's image to Docker Hub. First, our team creates a repository on Docker Hub.

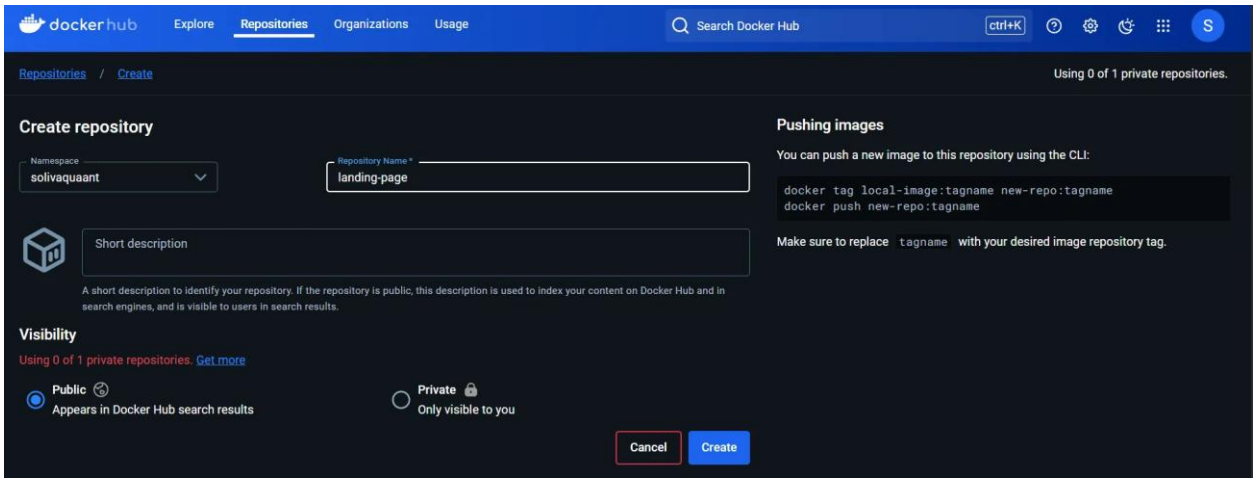

Figure 2.11. Creating repository on Docker Hub

In the terminal, our team logs into Docker Hub using the command **docker login -u** *<username>* and enters the password.

Figure 2.12. Logging into Docker Hub

Before pushing to Docker Hub, our team tags the image using the command **docker tag** *<specified_image_name> <username>/<image_name>:<tag_name>*. This tagging helps with user management. After tagging, our team uses the **docker image ls** command to view the list of images on the machine and sees that a copy of the specified image has been created.

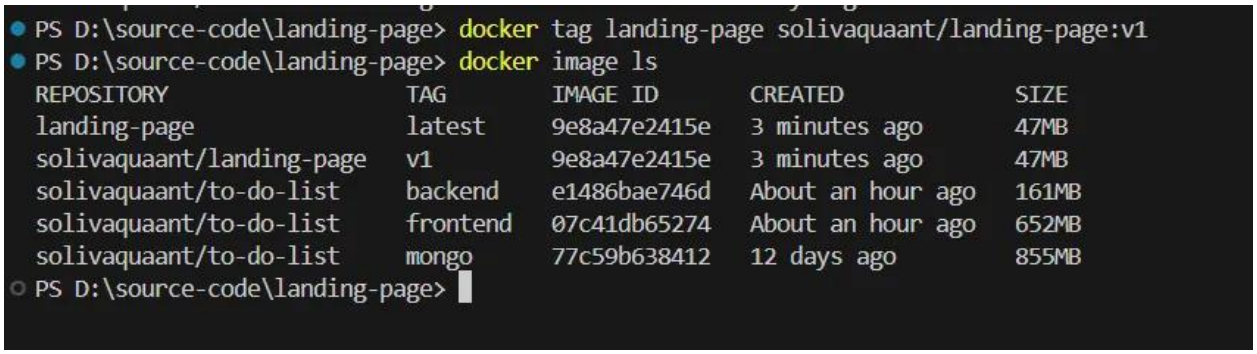

Figure 2.13. Tagging the image and verification

After completing the tagging process, our team pushes the images to Docker Hub using the command **docker push** *<username>/<image_name>:<tag_name>*

Figure 2.14. Pushing image to Docker Hub

When the image is successfully pushed, the Repository section in the Docker Hub interface will display the image's information.

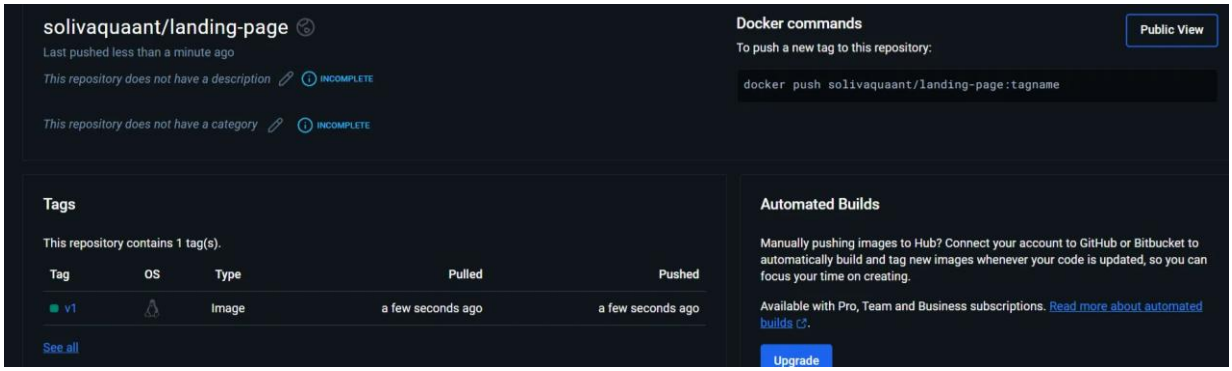

Figure 2.15. List of images pushed to Docker Hub

At this point, other machines can easily pull and run the image anywhere using the command **docker pull -a** <*username*>/<*image_name*>.

```
C:\Users\ASUS> docker pull -a solivaquaant/landing-page
v1: Pulling from solivaquaant/landing-page
43c4264eed91: Already exists
d1171b13e412: Already exists
596d53a7de88: Already exists
f99ac9ba1313: Already exists
fd072e74e282: Already exists
379754eea6a7: Already exists
45eb579d59b2: Already exists
472934715761: Already exists
002439eecfb8: Already exists
Digest: sha256:b1f458eb91a3326a1053d3859f3b511becc19e61bdec4b8c3e6bcb91acf5e6ec
Status: Downloaded newer image for solivaquaant/landing-page
docker.io/solivaquaant/landing-page

What's next:
    View a summary of image vulnerabilities and recommendations → docker scout quickview solivaquaant/landing-page
```

Figure 2.16. Pulling image to machine

After pulling the image to the machine, everyone can use the **docker image ls** command to list existing images on the machine.

```
C:\Users\ASUS>docker image ls
REPOSITORY                TAG        IMAGE ID       CREATED             SIZE
solivaquaant/landing-page   v1         9e8a47e2415e   7 minutes ago       47MB
solivaquaant/to-do-list     backend    e1486bae746d   About an hour ago   161MB
solivaquaant/to-do-list     frontend   07c41db65274   About an hour ago   652MB
```

Figure 2.17. Listing image inventory

Run the container based on the image just pulled from Docker Hub using the docker run command as mentioned above.

```
C:\Users\ASUS>docker run -d --rm -p 80:80 solivaquaant/landing-page:v1
71a1cf4b6fe2289f6356d265825fe8e67571fd48b9bbe7ebf56774df0fe00004
```

Figure 2.18. Running container

Access the deployed website by navigating to http://localhost:80/

2.3.2. Advanced Configuration

Source code: https://github.com/solivaquaant/simple-to-do-list

To deploy a complete website with frontend, backend, and database, our team will write two separate Dockerfiles for frontend and backend, and one docker-compose.yml file to connect the services.

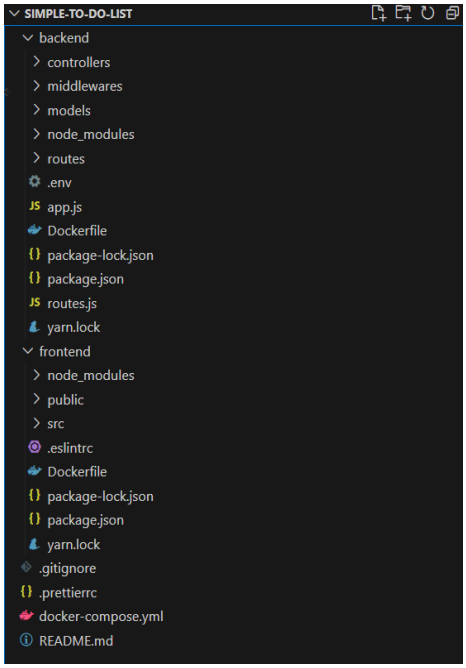

Figure 2.19. Web application directory structure

First, writing the Dockerfile for frontend:

- FROM node:18-alpine: Use Node image version 18 with Alpine Linux as the container base, helping reduce container size.
- WORKDIR /frontend: Set the working directory in the container to /frontend. Subsequent commands will be executed in this directory.
- COPY package*.json ./: Copy package.json and package-lock.json files from the computer to the current directory (/frontend) in the container.
- RUN npm install: Run npm install to install dependencies for the frontend application.
- COPY . .: Copy all project directory content from the computer to the current directory (/frontend) in the container.
- RUN npm run build: Run npm run build to build the application into static files in the build directory.
- RUN npm install -g serve: Install the serve package globally, a tool for serving static files, helping deploy the frontend application statically.

- EXPOSE 3000: Open port 3000 for the container to receive connections on this port.
- CMD ["serve", "-s", "build"]: Set the default command when the container starts to serve -s build, running the static frontend application from the build directory.

```
FROM node:18-alpine

WORKDIR /frontend

COPY package*.json ./

RUN npm install

COPY . .

RUN npm run build

RUN npm install -g serve

EXPOSE 3000

CMD ["serve", "-s", "build"]
```

Figure 2.20. Frontend Dockerfile

Next, writing the Dockerfile for backend:

- FROM node:18-alpine: Use Node image version 18 with Alpine Linux, a lightweight Linux distribution, as the container base.
- WORKDIR /backend: Set the working directory in the container to /backend.
- COPY package*.json ./: Copy package.json and package-lock.json files to the current directory (/backend) in the container.
- RUN npm install: Run npm install to install dependencies listed in package.json.
- COPY . .: Copy all project directory content from the computer to the current directory (/backend) in the container.
- EXPOSE 3001: Open port 3001 for the container to receive connections on this port.
- CMD ["node", "app.js"]: Set the default command when the container starts to node app.js, launching the application.

```
FROM node:18-alpine

WORKDIR /backend

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3001

CMD ["node", "app.js"]
```

Figure 2.21. Backend Dockerfile

Then, our team will create a docker-compose.yml file:

- services: Define services, where each service corresponds to a docker image. In this case, we have 3 services: backend, frontend, and mongo.
- container_name: Assign names to containers for easier management.
- restart: always: Ensures containers automatically restart if they encounter issues.
- build: context: <directory>: Specify the source directory for Docker to find and build the frontend's Dockerfile.
- ports: Open connection ports from the host machine to the container.
- depends_on: Indicate that this service requires a prerequisite service to start first. In this case, the "backend" service only starts if the "mongo" service has started, and the "frontend" service only starts if the "backend" service has started.
- env_file: Read environment variables from the .env file in the backend directory. This is because the MongoDB connection string is stored in the .env file in the backend.
- volumes: Mount the mongo-data volume to the /data/db directory in the container, storing MongoDB data to ensure data persistence when the container is removed.
- volumes: mongo-data: Defines the mongo-data volume mounted to the mongo container to store MongoDB data, ensuring data persistence when the container restarts.

```yaml
services:
  frontend:
    container_name: frontend-container
    restart: always
    build:
      context: ./frontend
    ports:
      - "3000:3000"
    depends_on:
      - backend

  backend:
    container_name: backend-container
    restart: always
    build:
      context: ./backend
    ports:
      - "3001:3001"
    env_file:
      - ./backend/.env
    depends_on:
      - mongo

  mongo:
    container_name: mongo-db
    image: mongo:latest
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db

volumes:
  mongo-data:
```

Figure 2.22. Content of docker-compose.yml

To build images and start containers, use the command **docker compose up -d**

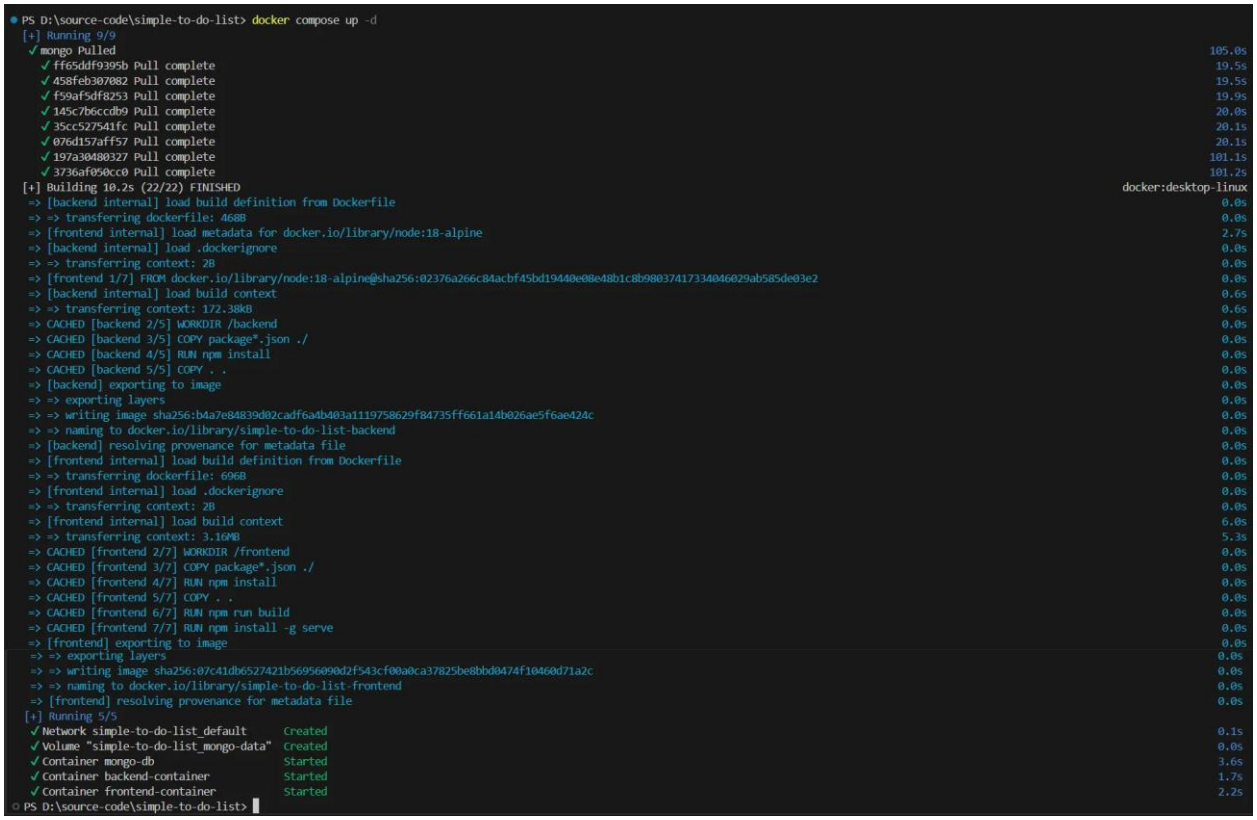

Figure 2.23. Building images and starting containers

After executing the above command, Docker will build the images declared in the services, use the built or pulled images, and start the containers. At this point, we can list running containers on computer using the **docker ps** command.

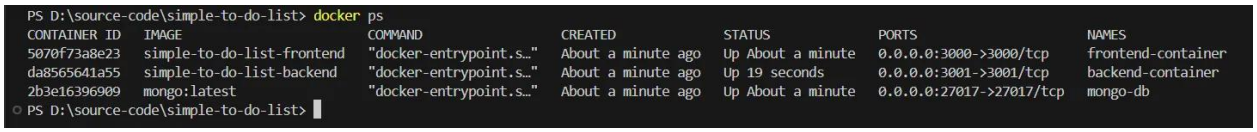

Figure 2.24. List of existing containers on the machine

Use a browser to access http://localhost:3000/ to access the deployed website.

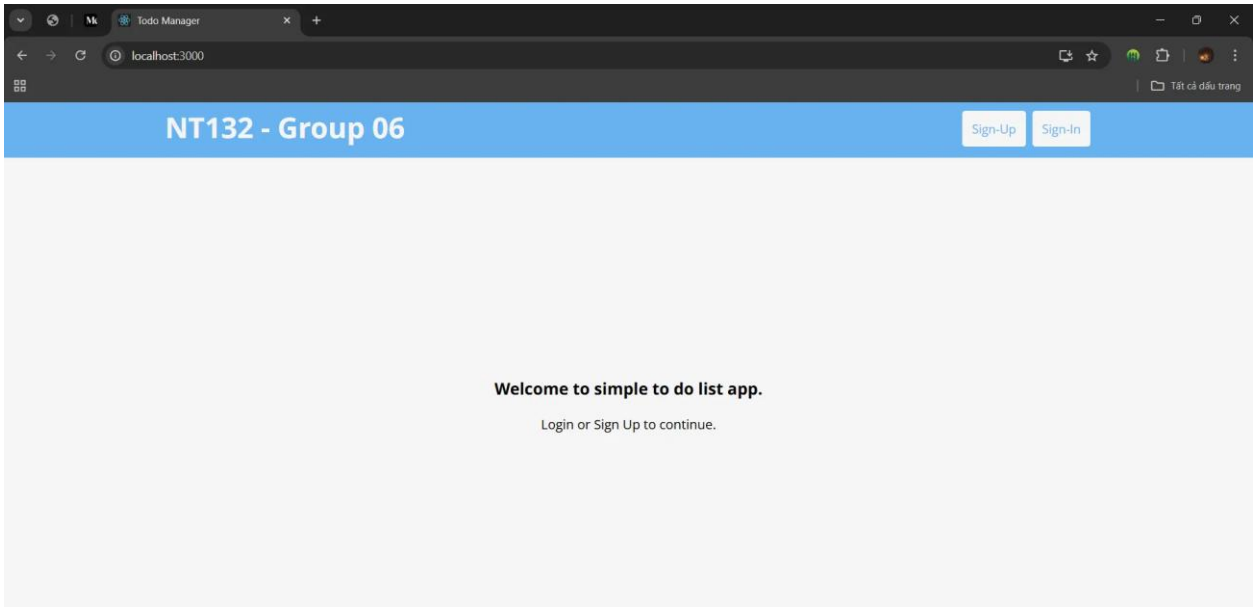

Figure 2.25. Web application interface

When necessary, can use the **docker compose down** command to stop and remove running containers.

```
PS D:\source-code\mern-todo-main\simple-to-do-list> docker compose down
[+] Running 4/4
 ✓ Container frontend-container       Removed
 ✓ Container backend-container        Removed
 ✓ Container mongo-db                 Removed
 ✓ Network simple-to-do-list_default  Removed
PS D:\source-code\mern-todo-main\simple-to-do-list>
```

Figure 2.26. Stopping or removing containers

To facilitate deployment on other machines, our team will push the images to Docker Hub. First, create a repository on Docker Hub.

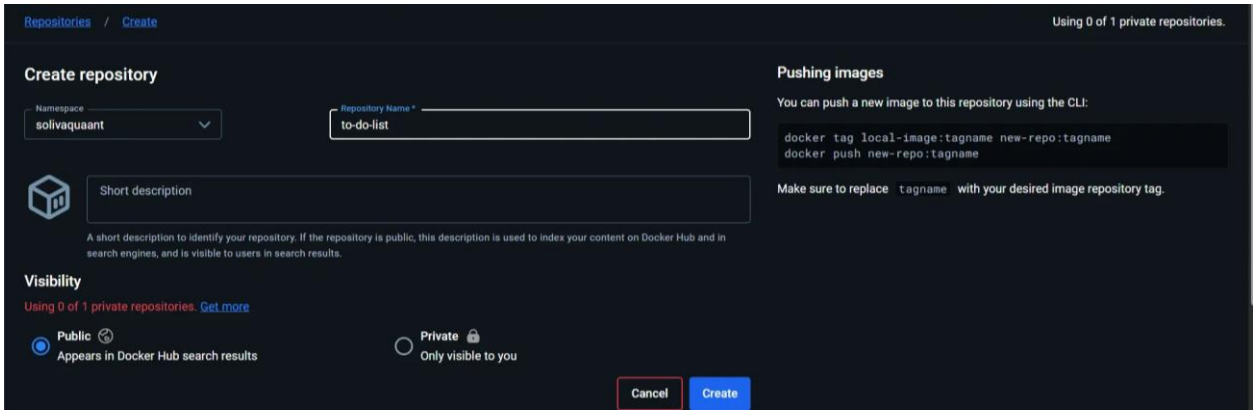

Figure 2.27. Creating repository on Docker Hub

Login to Docker account using **docker login -u** *<username>,* then enter password.

Figure 2.28. Logging into Docker

For better and easier management, our team tagged the images using **docker tag** *<image_name> <username>/<repo_name>:<tag>*. Then can use **docker image ls** to check the tags assigned to images.

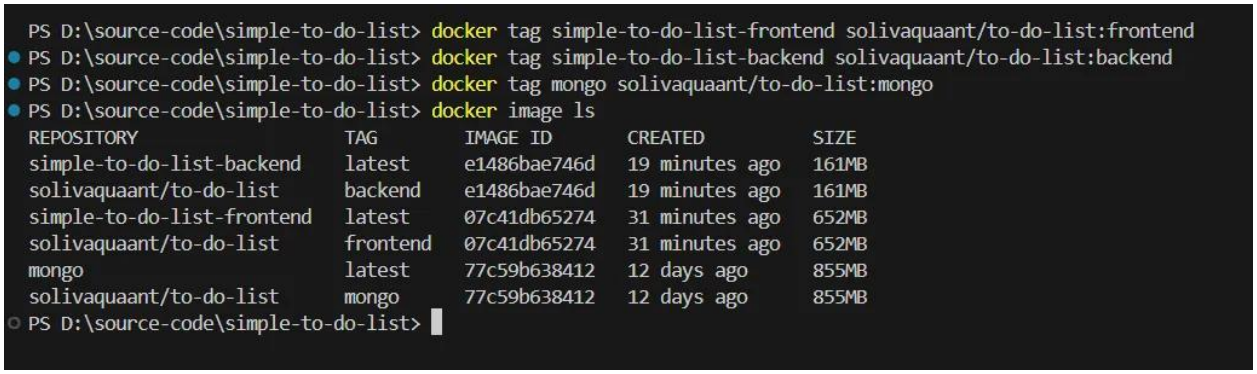

Figure 2.29. Tagging images and listing images

After tagging, push images from computer to Docker Hub using **docker push** *<username>/<repo_name>:<tag>*

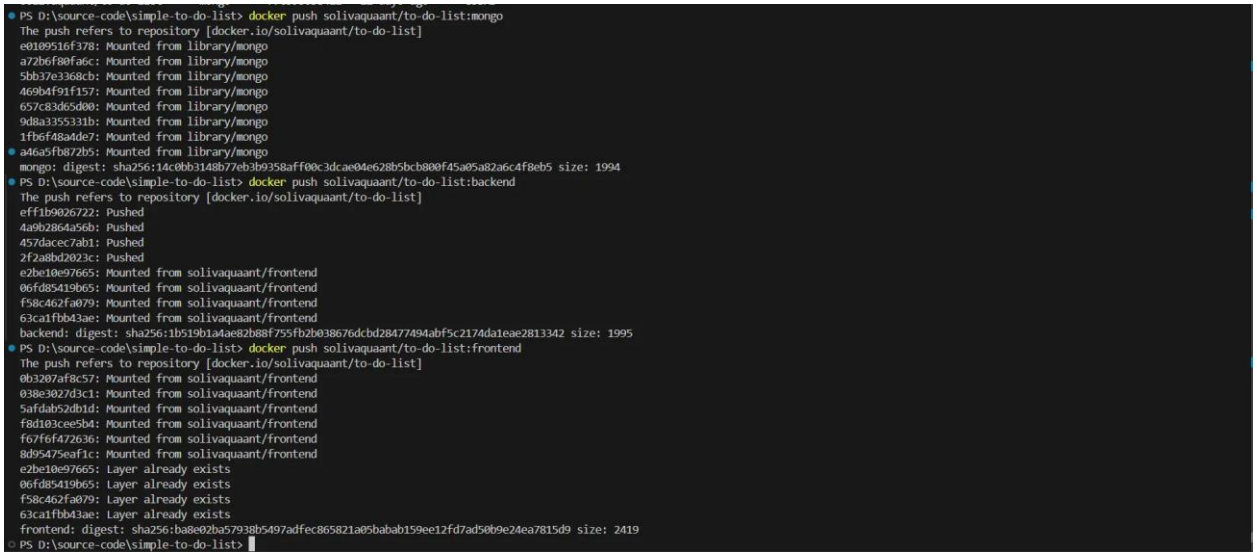

Figure 2.30. Pushing images to Docker Hub

To verify if images were successfully pushed, check Docker Hub. The image pushing process went smoothly.

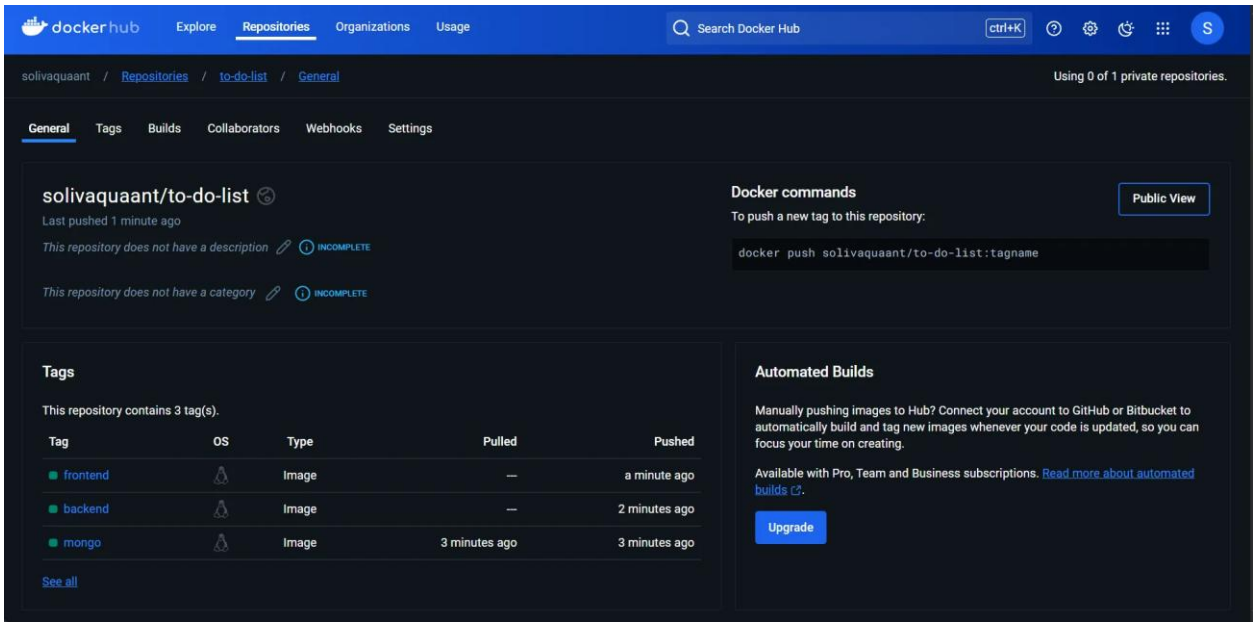

Figure 2.31. List of images on Docker Hub

For another computer to deploy the website, download the repository using **docker pull -a** *<username>/<repo_name>*.

```
C:\Users\ASUS>docker pull -a solivaquaant/to-do-list
backend: Pulling from solivaquaant/to-do-list
Digest: sha256:1b519b1a4ae82b88f755fb2b038676dcbd28477494abf5c2174da1eae2813342
frontend: Pulling from solivaquaant/to-do-list
Digest: sha256:ba8e02ba57938b5497adfec865821a05babab159ee12fd7ad50b9e24ea7815d9
mongo: Pulling from solivaquaant/to-do-list
Digest: sha256:14c0bb3148b77eb3b9358aff00c3dcae04e628b5bcb800f45a05a82a6c4f8eb5
Status: Downloaded newer image for solivaquaant/to-do-list
docker.io/solivaquaant/to-do-list

What's next:
    View a summary of image vulnerabilities and recommendations → docker scout quickview solivaquaant/to-do-list

C:\Users\ASUS>
```

Figure 2.32. Pulling images from Docker Hub

Using **docker image ls** to verify all images were downloaded.

```
C:\Users\ASUS>docker image ls
REPOSITORY                  TAG       IMAGE ID       CREATED          SIZE
solivaquaant/to-do-list     backend   e1486bae746d   47 minutes ago   161MB
solivaquaant/to-do-list     frontend  07c41db65274   59 minutes ago   652MB
solivaquaant/to-do-list     mongo     77c59b638412   12 days ago      855MB
```

Figure 2.33. List of existing images on machine

Start containers from images using **docker run -d --rm -p** *<host_port>:<container_port> <username>/<image_name>:<tag>*

```
C:\Users\ASUS>docker run -d --rm -p 3001:3001 solivaquaant/to-do-list:backend
af882aa6683a100d79653da27634da4a68f75e9deb83389c5544bf636a43ca51

C:\Users\ASUS>docker run -d --rm -p 3000:3000 solivaquaant/to-do-list:frontend
49696810afd0ffc18ee210ca5aa50f86a808c97b7dfc4625d27e9b70f315cf7b

C:\Users\ASUS>docker run -d --rm -p 27017:27017 solivaquaant/to-do-list:mongo
2eaaec791266a712756649fef4c493cef681d84a08700ee855024a7e9bdbfbc7
```

Figure 2.34. Starting containers sequentially

Verify by accessing http://localhost:3000/

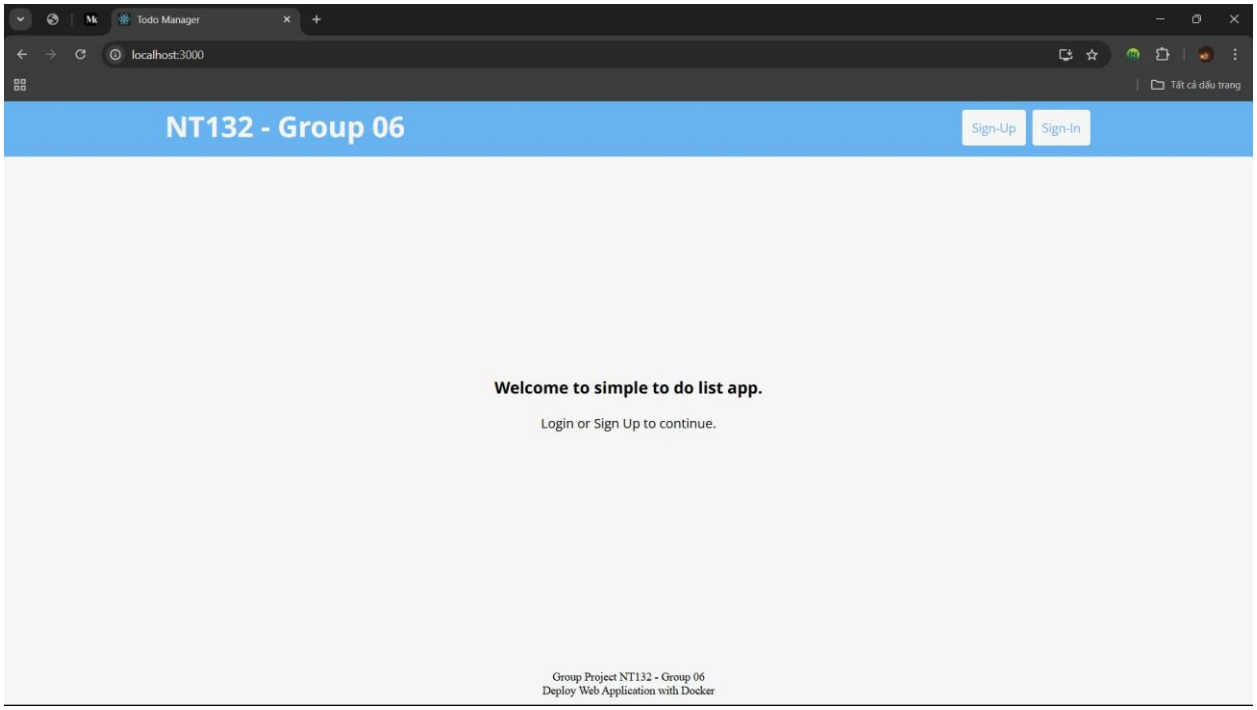

Figure 2.35. Welcome interface

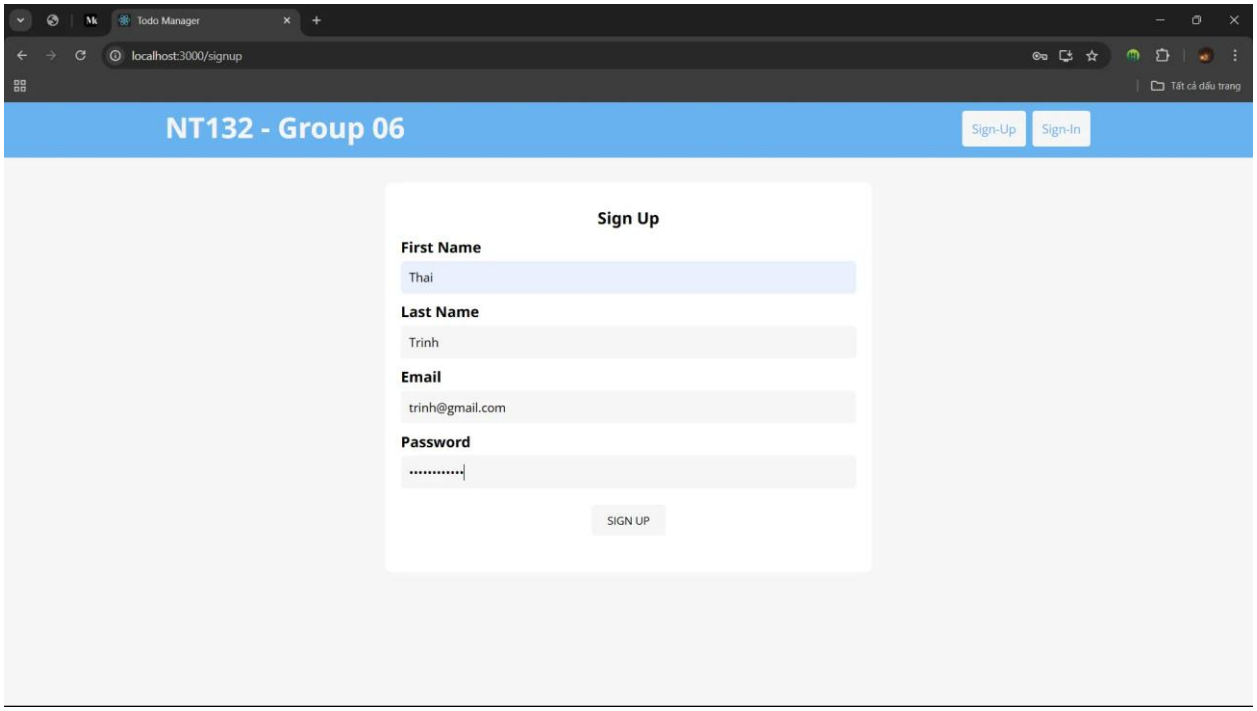

Figure 2.36. Registration page

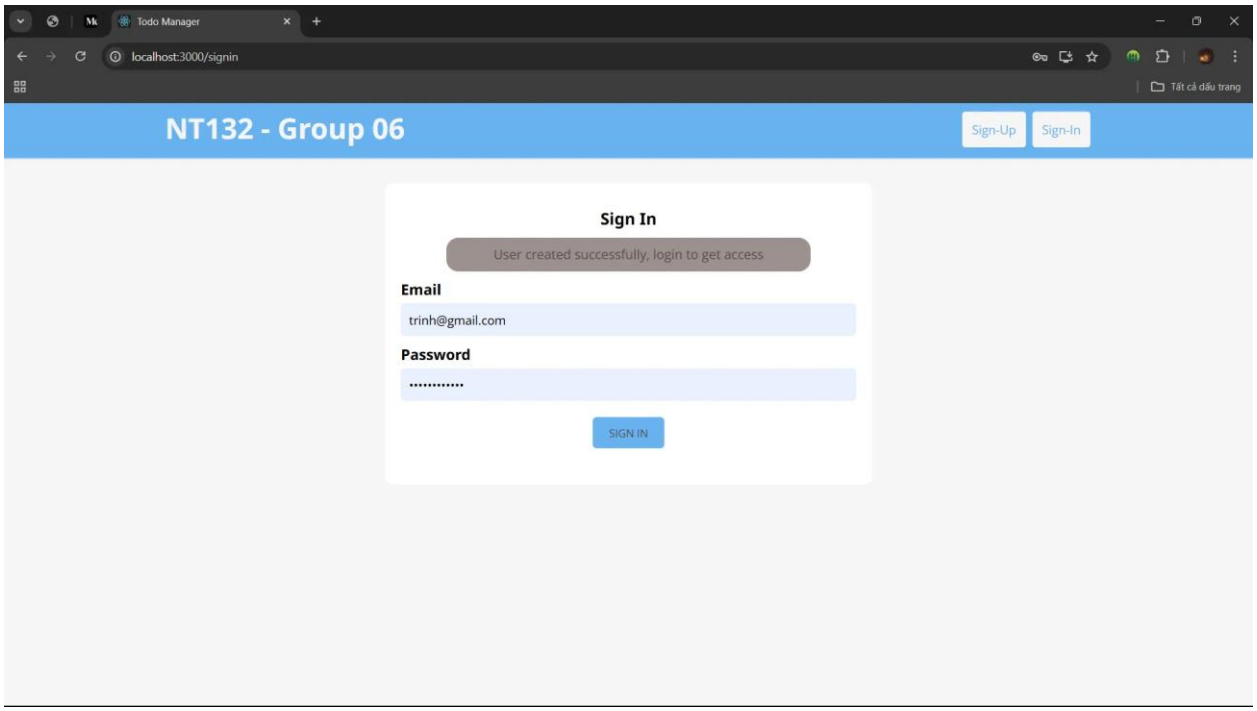

Figure 2.37. Login page

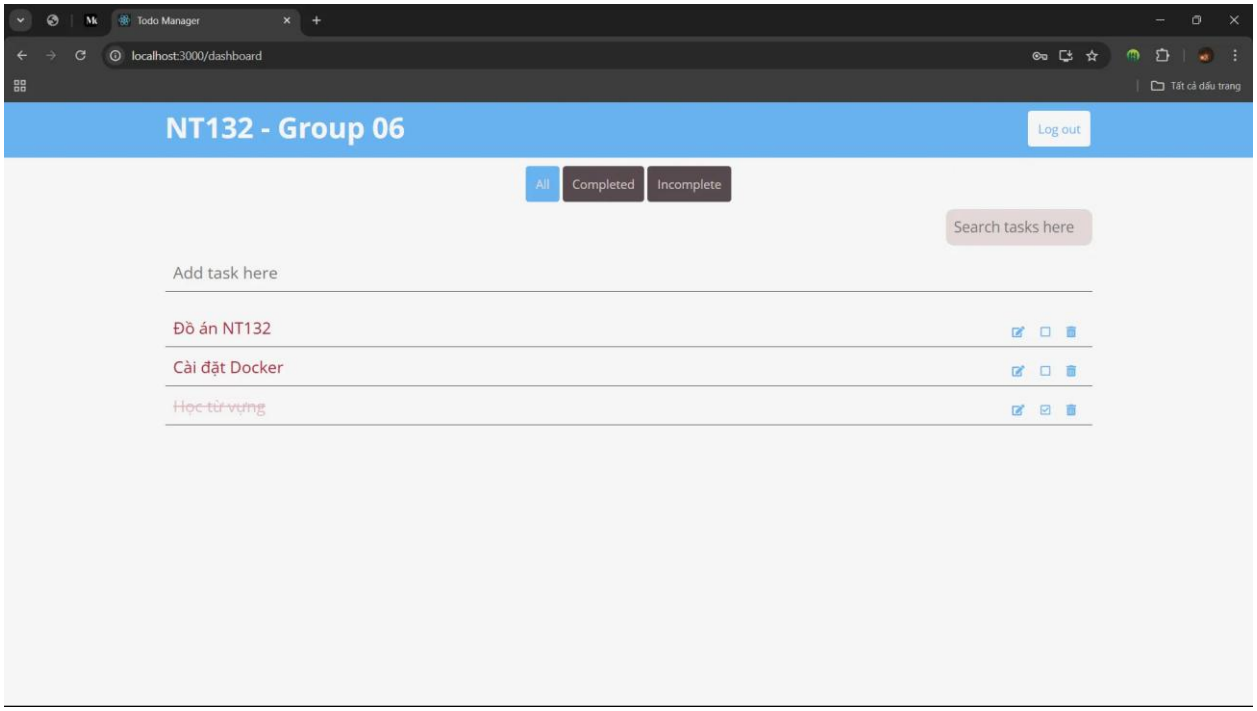

Figure 2.38. Post-login interface

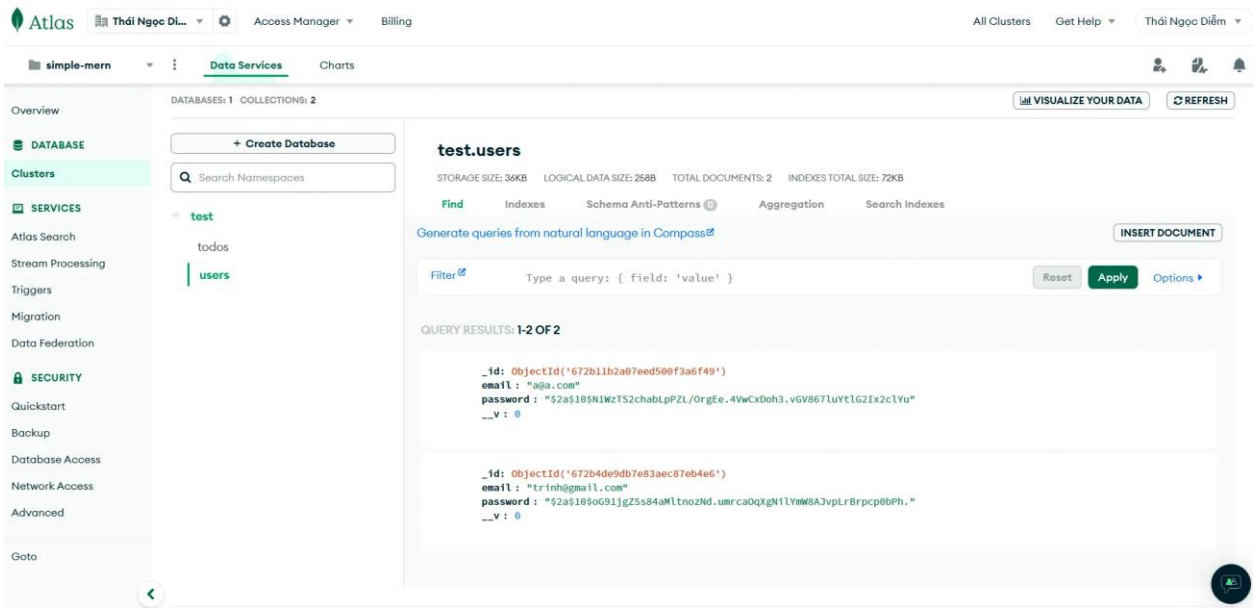

Figure 2.39. MongoDB users collection

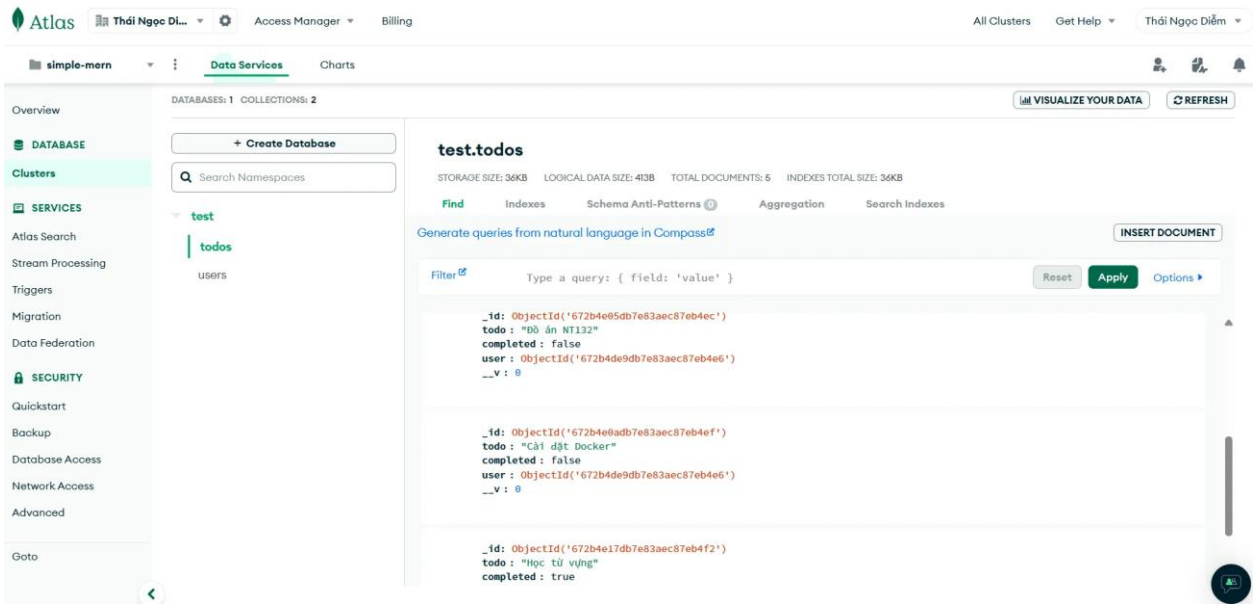

Figure 2.40. MongoDB todos collection

Chapter 3. **Result and Conclusion**

3.1. Result

Achievements:

- Successfully understood and deployed web applications using Docker. Mastered basic concepts like Dockerfile, Docker image, Docker container, Docker Compose, etc.
- Understanding of Docker's packaging process and operations
- Successfully deployed two websites using Docker:
    - Static website
    - Web application (frontend, backend, and Database)

Advantages:

- High portability: Containers can run on any machine with Docker installed
- Efficient resource management: Containers share the same kernel, reducing system resource consumption
- Fast deployment: Lightweight containers deploy quickly in seconds
- Security: Each container is an isolated environment, minimizing attack risks

Disadvantages:

- Docker Containers cannot automatically update when application resources change, requiring rebuilds after each update
- Risk of attacks when using untrusted images
- Docker can consume significant resources when running multiple containers and large applications

3.2. Conclusion

After 3 weeks from the project started, our team accomplished most basic and advanced requirements for deploying static websites and web applications using Docker.

Using Docker Compose made web deployment easier and faster, improving performance by minimizing startup time and increasing response speed, while simplifying management and deployment of multiple services. Additionally, the container mechanism supports web application scalability.

As a result, new web deployment time reduced from 2 hours to 15 minutes, while increasing system fault tolerance. This improved product quality and optimized web development time.

During research and implementation, our team faced some challenges related to network management and data storage. The project experience helped our team gain experience in detailed project planning, learn Docker and web knowledge, and strengthen teamwork, mutual support, coordination, and task division.

This project revealed Docker Compose's potential. With acquired knowledge and experience, our team believes they can apply Docker to future projects to enhance security and scalability.

REFERENCES

1. *LearnDocker*. (n.d.). LearnDocker. https://learndocker.online/
2. *Play with Docker Classroom*. (n.d.). https://training.play-with-docker.com/
3. Potnuru, R. (2023, February 4). *Dockerizing Your MERN Stack App: A Step-by-Step Guide.* DEV Community. https://dev.to/itsrakesh/dockerizing-your-mern-stack-app-a-step-by-step-guide-19nh
4. *"What is Docker?"* (2024, September 10). Docker Documentation. https://docs.docker.com/get-started/docker-overview/