Docker is actually a platform that includes several tools working together which implements containerization concept.

Docker consists of:

* Docker Engine: The core tool that:

Creates and runs containers

Builds images

Manages Docker objects

* Docker Desktop: A user-friendly application that includes:

The Docker Engine

A graphical interface

Tools for managing containers and images

* Docker CLI (Command Line Interface):

Tools to interact with Docker through commands

Used to build, run, and manage containers

Let’s dive into containarization’s core concepts

1. What is Docker containarization?

It is like a lightweight, portable container for your application and all its dependencies. Think of it like a shipping container - everything your app needs is packed inside, and it works the same way no matter where you run it (your laptop, cloud, etc.).

1. Key concepts:

* Container: A running instance of your application with all necessary dependencies
* Image: A blueprint for creating containers (like a template)
* Dockerfile: A recipe that tells Docker how to build your image
* Registry: A place to store and share Docker images (like Docker Hub)

1. Explain Containers with a practical example related to your sentiment analysis service:

A Container is a running instance of an image - it's the actual live, running version of our application. Think of it this way:

* If an image is like a recipe book, a container is the actual cooked meal
* If an image is like a class in programming, a container is like an object instance

Key points about containers:

* Isolation: Each container runs in its own environment, with its own:
* File system
* Network interface
* Process space
* Resource limits (CPU, memory)
* Multiple Containers: You can run multiple containers from the same image. For example:
* One container for testing
* Another for production
* Each running independently

1. Explain Docker images with a practical example:

Think of a Docker image like a snapshot of a complete system setup. For our sentiment analysis service, an image would include:

* Python installation
* All required libraries (FastAPI, Hugging Face, etc.)
* application code
* Any configuration files
* The operating system components needed to run everything

We need images because:

Consistency: Everyone using the application gets exactly the same environment

Portability: The image can run anywhere Docker is installed (local machine, cloud, etc.)

Version Control: You can maintain different versions of our application setup

Quick Deployment: New instances can be started instantly from the image

1. Explain Dockerfile with a practical example:

A Dockerfile is like a set of instructions that tells Docker how to build your image. Think of it as :“A step-by-step recipe”.

Each line is a command that adds a layer to your image

For our Python sentiment analysis service, a Dockerfile would typically include instructions to:

* Start with a base Python image
* Copy your requirements.txt
* Install dependencies
* Copy your application code
* Set up the command to run your FastAPI service

6. Explain Docker Registry with a practical example:

A Registry is like a repository or library for Docker images - think of it like GitHub but for Docker images. The most common registry is Docker Hub.

Key points about registries:

Storage & Sharing:

* Store sentiment analysis service images
* Share images with team members or deploy to different servers
* Pull pre-built images (like Python or FastAPI base images)

Types of Registries:

* Public (like Docker Hub)
* Private (for sensitive code)
* Self-hosted (within your organization)

Common Operations:

* Push: Upload your images
* Pull: Download images
* Tag: Version your images

7. Lets start working on steps to create Docker container:

1. Install Docker Desktop: Go to <https://www.docker.com/products/docker-desktop/> .Download the version for your operating system (Windows/Mac/Linux). Install and follow the setup instructions.
2. Create a Dockerfile:In your project's root directory (where main.py is located)Create a new file named exactly **Dockerfile** (no file extension)You can create it using any text editor.
3. Add Dockerfile.

FROM python:3.12.4-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY ./app ./app

EXPOSE 8000

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

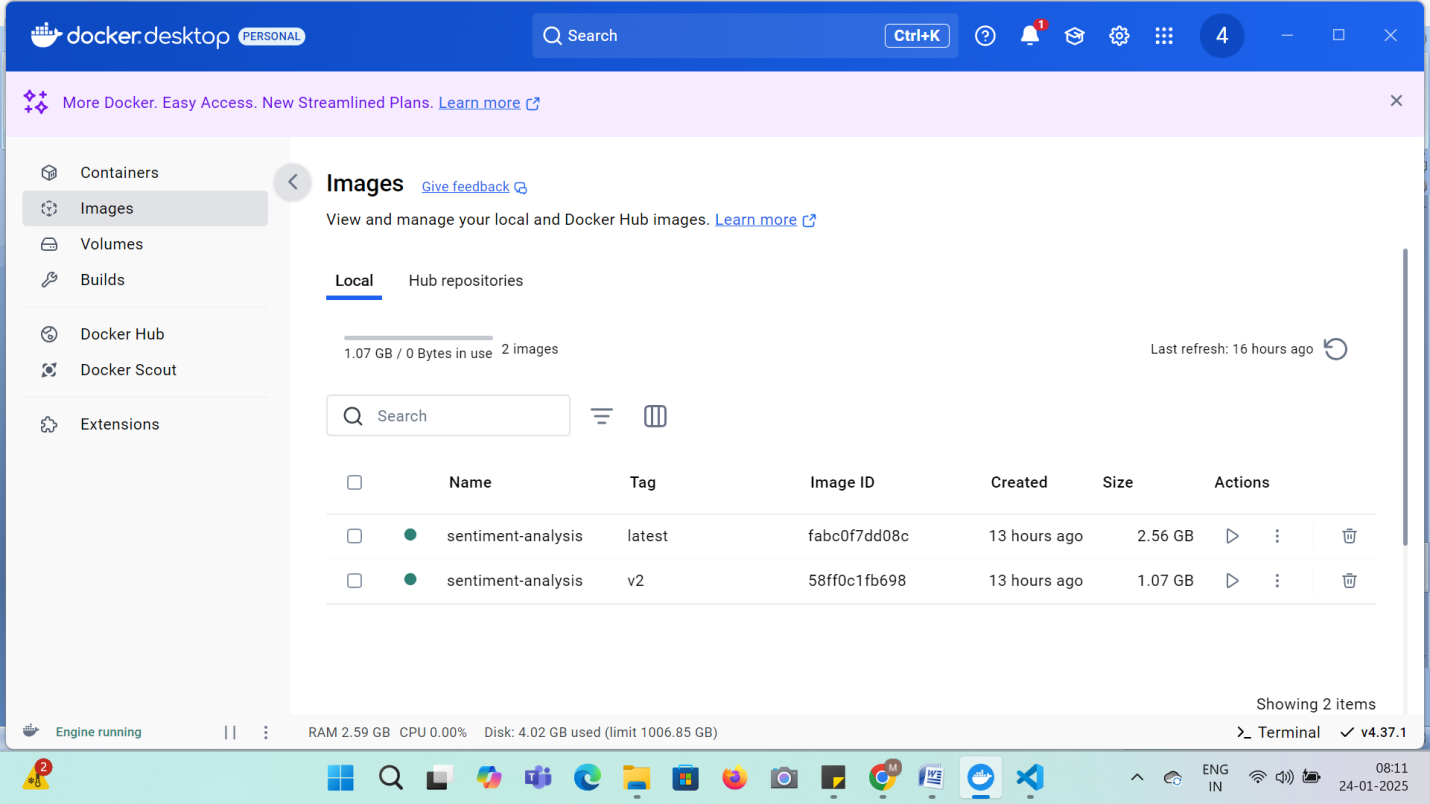
1. Build the image **docker build -t sentiment-analysis .** command.
2. Run the docker with the below command docker run -p 8000:8000 sentiment-analysis
3. Check if service is running:

Open browser to http://localhost:8000/docs

You should see the FastAPI Swagger UI

1. Images which are created locally can be seen on docker desktop🡪Images.

Docker Desktop runs a local Docker daemon (engine).When you run **docker build**, it creates the image in your local Docker registry. Docker Desktop automatically displays these local images



1. Steps to tag your image for Docker Hub (this is the next step for pushing):

* setting up a Docker Hub account. Go to hub.docker.com/signup
* verify and log in to Docker Hub through your terminal:First, log in using this command:
* **docker login**

you got "Login Succeeded", you're already authenticated and ready to proceed.

Tag your image for Docker Hub (this is the next step for pushing). The command format will be:

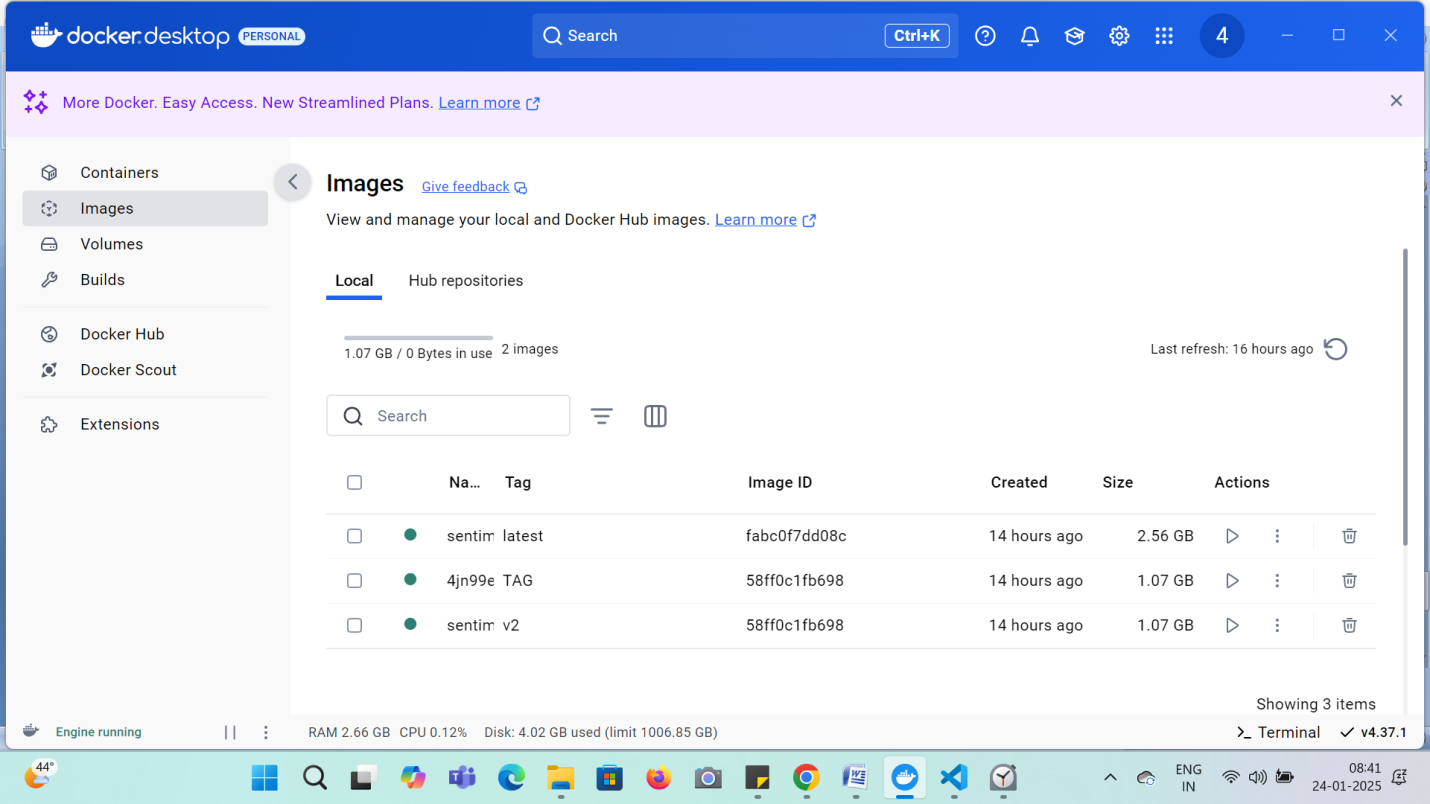
docker tag sentiment-analysis:v2 YOUR-USERNAME/sentiment-analysis:TAG

You can check if the tagging worked using these commands:

* **docker images**

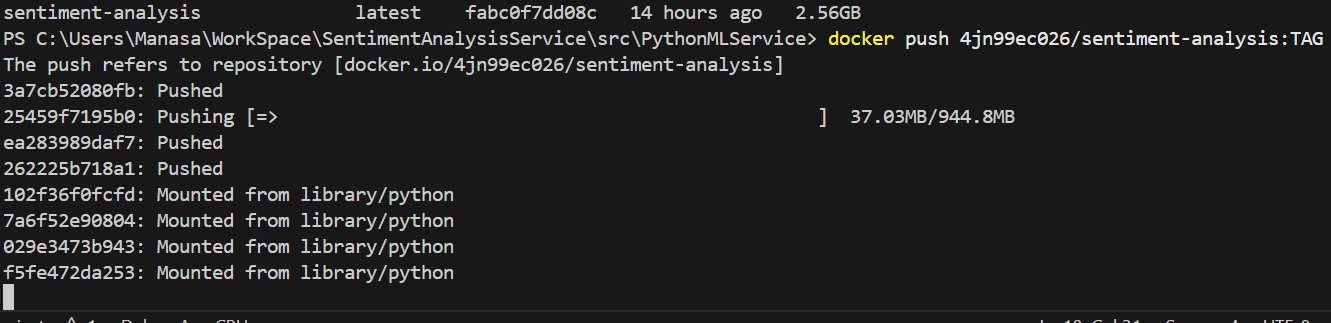
This will list all your local images. You should see both:

* Your original image (sentiment-analysis:v2)
* Your newly tagged image (username/sentiment-analysis:tag)

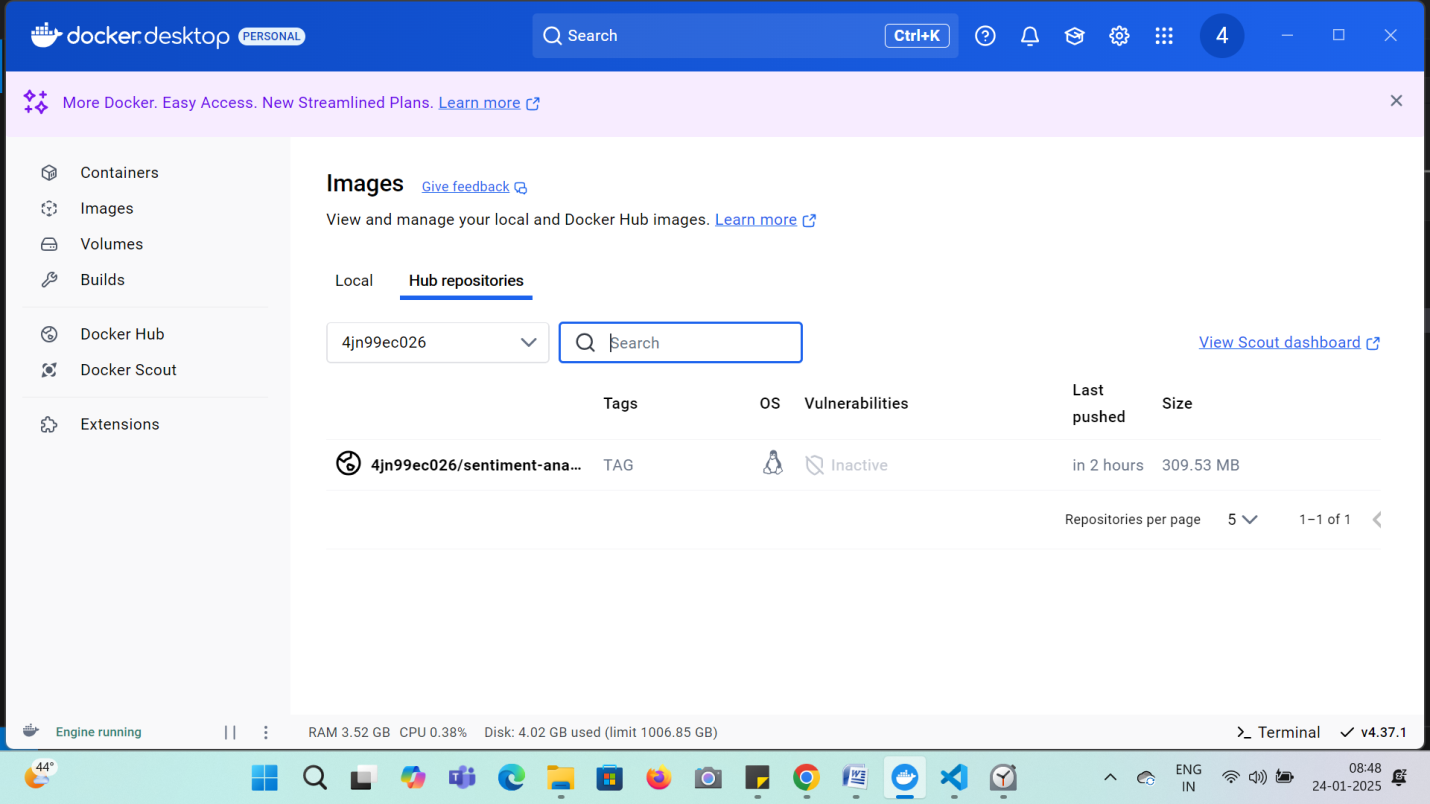


1. How to push your tagged image to Docker Hub

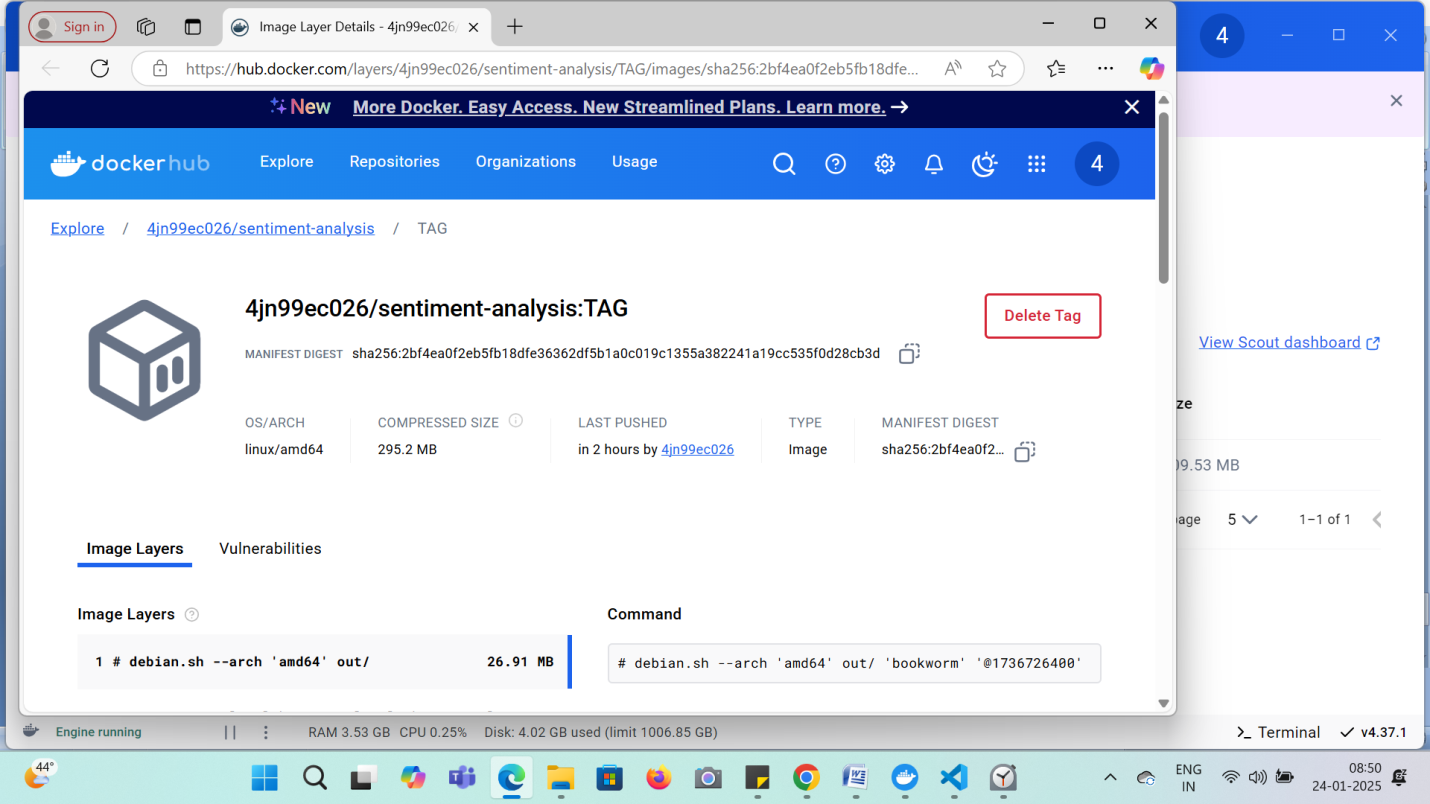
The command format is: docker push YOUR-USERNAME/sentiment-analysis:TAG



Once the Push completes . We can see it in the Hub repositories



View in Docker hub the image which is pushed



1. Now that your image is on Docker Hub, you can:

Test pulling it from another machine using: docker pull YOUR-USERNAME/sentiment-analysis:TAG

1. Share it with others by giving them: Your Docker Hub repository name,The tag you used

So the complete repository name would be: **YOUR-USERNAME/sentiment-analysis**

**Test your image which is pushed to Docker Hub**

* First, let's pull your image (as if you were a new user):

docker pull YOUR-USERNAME/sentiment-analysis:TAG

* Then run it:

docker run -p 8000:8000 YOUR-USERNAME/sentiment-analysis:TAG

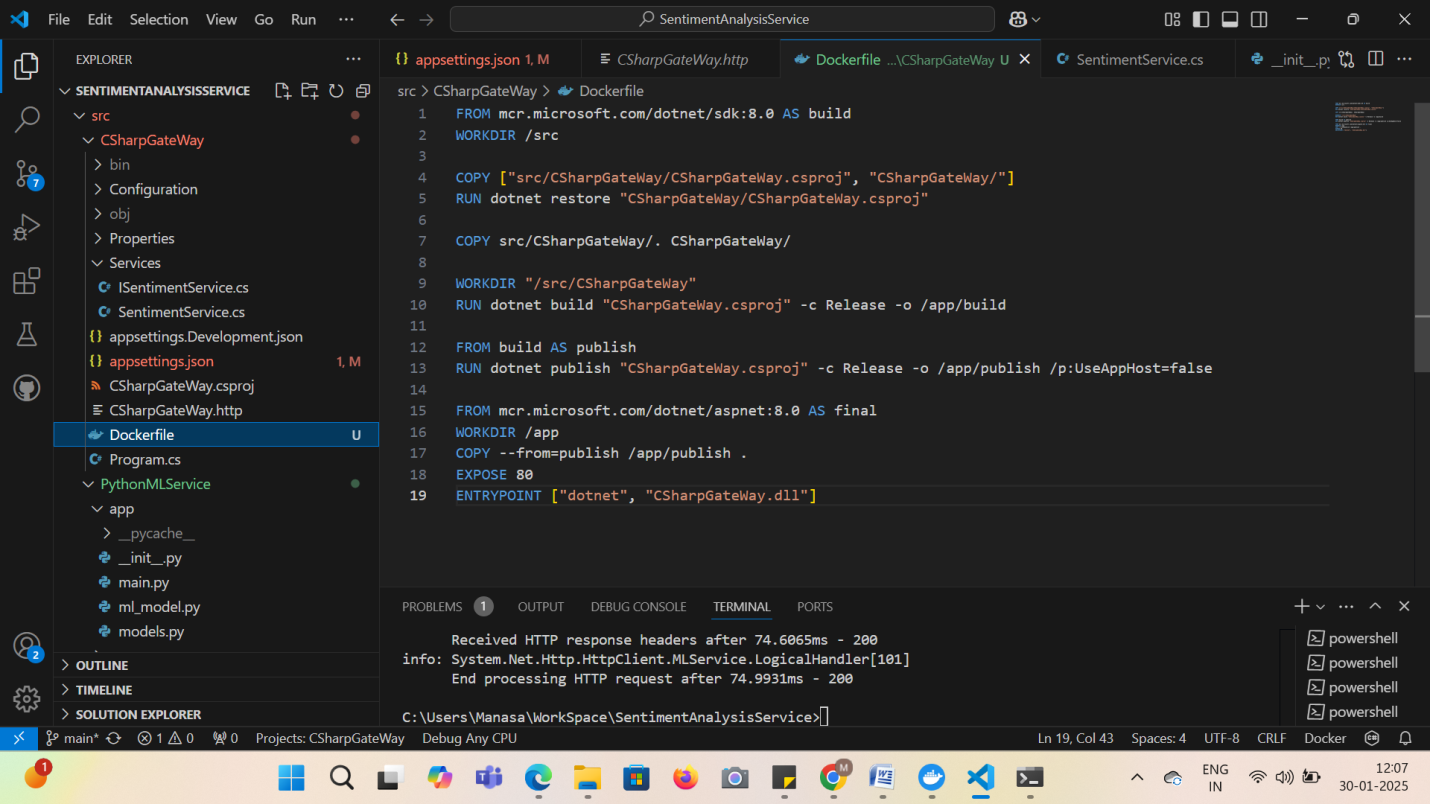
Test it by:

* Opening **http://localhost:8000/docs** in your browser
* Or trying a test request

Now, let’s how to create docker file for the C# part of the API

Create a new file named **Dockerfile** in your C# project directory:It should be in: **src/CSharpGateWay/Dockerfile**

once the docker file is created



First, let's build the C# API Dockerfile:

* Navigate to your project root directory, Build the image: **docker build -t csharp-gateway:v1 -f src/CSharpGateWay/Dockerfile .**
* Run it: **docker run -p 80:80 csharp-gateway:v1**

Let's push C# API container to Docker Hub. Similar to what we did with the Python service, we'll need to:

* Tag the image with your Docker Hub username:

docker tag csharp-gateway:v1 YOUR-USERNAME/csharp-gateway:v1

* Push to Docker Hub:

docker push YOUR-USERNAME/csharp-gateway:v1

Let's set up a **Docker Network for communication between your services**. We'll do this step by step:

First, create a new network:

**docker network create sentiment-network**

Let's verify the network creation using these commands:

List all networks:

docker network ls

Let's connect both containers to the network. First, we need to run our containers with network configuration:

* Run the Python FastAPI service:

docker run -d --name sentiment-service --network sentiment-network -p 8000:8000 YOUR-USERNAME/sentiment-analysis:TAG

* Run the C# Gateway:

docker run -d --name csharp-gateway --network sentiment-network -p 80:80 YOUR-USERNAME/csharp-gateway:v1

Key points:

**--name** gives each container a fixed name

**--network** connects to our network

**-d** runs in detached mode

verify the connections and test communication in steps:

* First, verify both containers are running and connected:

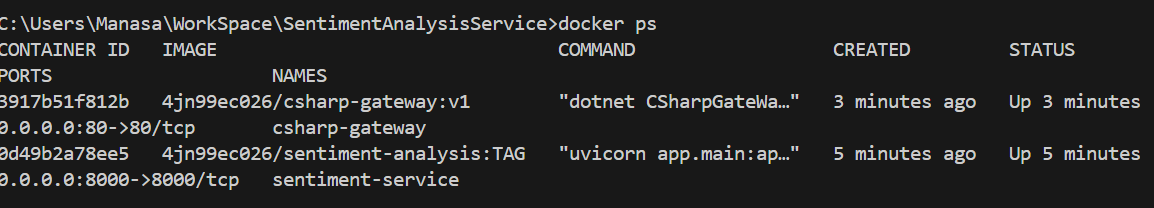
docker network inspect sentiment-network

Look for both containers in the "Containers" section.

* Check if both services are running:

docker ps

Should show both containers with status "Up"



1. To check the C# Gateway logs for error details:

docker logs csharp-gateway

1. Also check the Python service logs:

docker logs sentiment-service

To rebuild the C# service with the updated configuration if any. Here are the steps:

1. First, rebuild the image:

docker build -t YOUR-USERNAME/csharp-gateway:v1 -f src/CSharpGateWay/Dockerfile .

1. Stop and remove the old container:

docker rm -f csharp-gateway

1. Run the new container:

docker run -d --name csharp-gateway --network sentiment-network -p 8080:8080 -e "ASPNETCORE\_ENVIRONMENT=Development" YOUR-USERNAME/csharp-gateway:v1  


Create a Docker Compose file to manage both services. But before that what is Docker compose file?why do we need it?

Think of Docker Compose as a recipe book that:

* Defines all your services in one file
* Manages how they work together
* Makes it easy to start/stop everything at once

Benefits of using Docker Compose:

* Instead of running multiple docker commands, you use one command
* Keeps service configurations (ports, networks, volumes) in one place
* Makes it easier to share your setup with others
* Ensures services start in the correct order

Simple example of what it replaces:

* Instead of running separate commands for each service
* Instead of manually creating networks
* Instead of managing container dependencies

Here's a brief summary of creating and running services with Docker Compose:

Create **docker-compose.yml** in project root:

services:

sentiment-service:

image: YOUR-USERNAME/sentiment-analysis:TAG

ports:

- "8000:8000"

networks:

- sentiment-network

csharp-gateway:

image: YOUR-USERNAME/csharp-gateway:v1

ports:

- "8080:8080"

environment:

- ASPNETCORE\_ENVIRONMENT=Development

networks:

- sentiment-network

networks:

sentiment-network:

1. Key Steps:

* Build both images: **docker build**
* Start services: **docker-compose up -d**
* Check status: **docker-compose ps**
* View logs: **docker-compose logs**
* Stop services: **docker-compose down**

1. Test Services:

* Python API: http://localhost:8000/docs
* C# Gateway: http://localhost:8080/swagger/index.html

