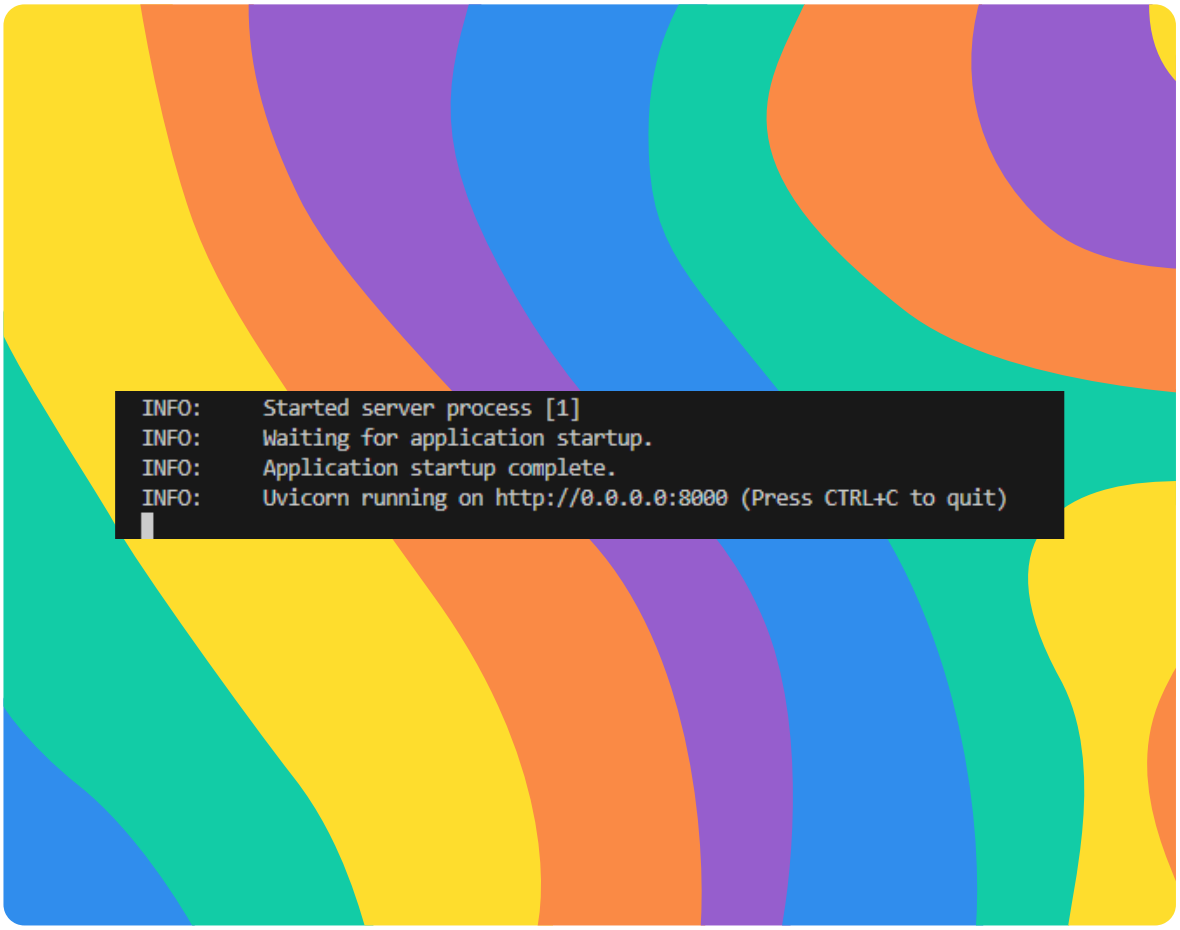
Containerize a RAG API with Docker



Introducing Today's Project!

In this project, I will demonstrate how to containerize my RAG API with Docker by packaging the FastAPI application, ChromaDB vector database, and Ollama local LLM into a Docker container that runs identically anywhere, and then share this Docker image to the world through Docker Hub. I'm doing this project to learn modern DevOps practices used by professional engineers at companies like Netflix, Uber, and Spotify to ensure applications work consistently across different environments, understand how containers solve the "it works on my machine" problem by bundling code and dependencies together, and gain hands-on experience with Docker as part of this DevOps × AI series that will eventually teach me Kubernetes orchestration and CI/CD automation with GitHub Actions.

Key services and concepts

Services I used were Docker Desktop, Docker CLI, Docker Hub, FastAPI, Uvicorn, Python, and curl or PowerShell Invoke-WebRequest. Key concepts I learnt include containerization, writing a Dockerfile, building Docker images, tagging images, pushing and pulling images from Docker Hub, running containers with port mapping, understanding image layers and caching, and verifying that a containerized RAG API behaves the same whether built locally or pulled from a registry.

Challenges and wins

This project took me approximately 3 hours to complete. The most challenging part was connecting my project correctly to Docker and uploading my image to Docker Hub due to authentication and tagging issues. It was most rewarding to resolve those challenges and successfully pull and run my API from Docker Hub, proving that my container works anywhere.

Why I did this project

This project met my goals because I now understand the full workflow from local development to containerization and public distribution, and overall it was a cool project that taught me a lot.

Setting Up the RAG API

In this step, I'm setting up my RAG API's code, database, dependencies, virtual environment, and ensuring Ollama is running so the API is fully functional before containerizing it with Docker. The RAG API is an Application Programming Interface that receives questions from users, searches through my documents stored in ChromaDB to find relevant information using embeddings, uses Ollama's local LLM (tinyllama) to generate accurate answers based on the retrieved context, and sends those answers back to users, making my RAG system accessible and interactive through a FastAPI web service

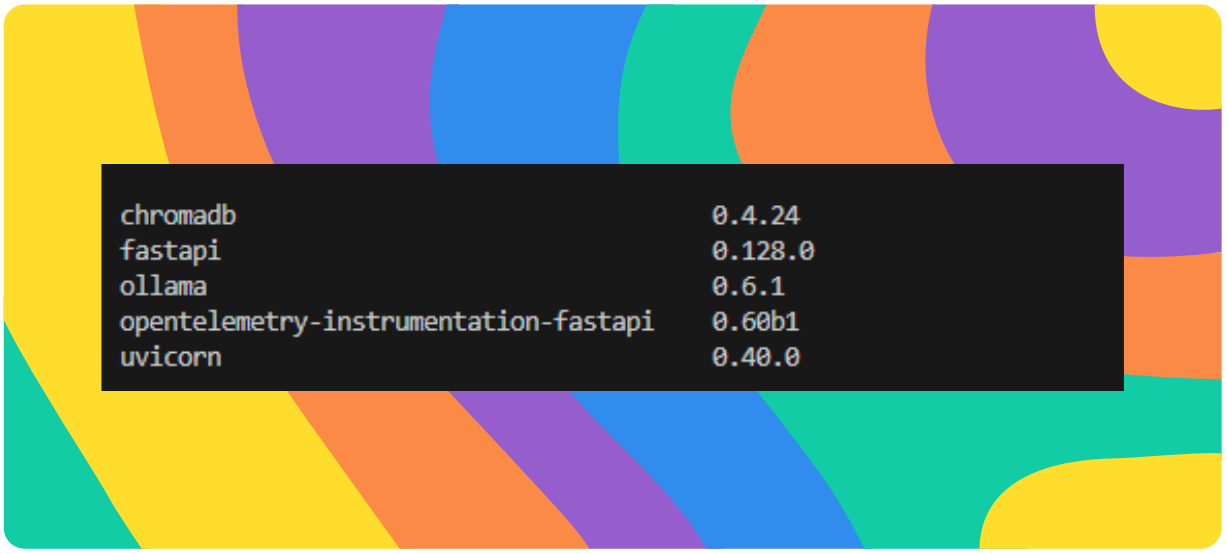
API setup and workspace

In this step, I’m creating a Docker based container environment for my RAG API so it can be packaged with its code, dependencies, and database and run identically on any machine. A virtual environment is an isolated runtime that contains the exact Python version, libraries, and configurations the API needs to function correctly. I need it because containerization removes manual setup, prevents dependency conflicts, and ensures the API works reliably on my machine, a teammate’s computer, or a production server without reconfiguration.

Dependencies installed

The packages I installed are FastAPI, Uvicorn, ChromaDB, and Ollama. FastAPI is used for creating the API itself and handling HTTP requests so users or applications can send questions and receive responses. ChromaDB is used for storing document embeddings and performing similarity search so the system can retrieve the most relevant information for a query.

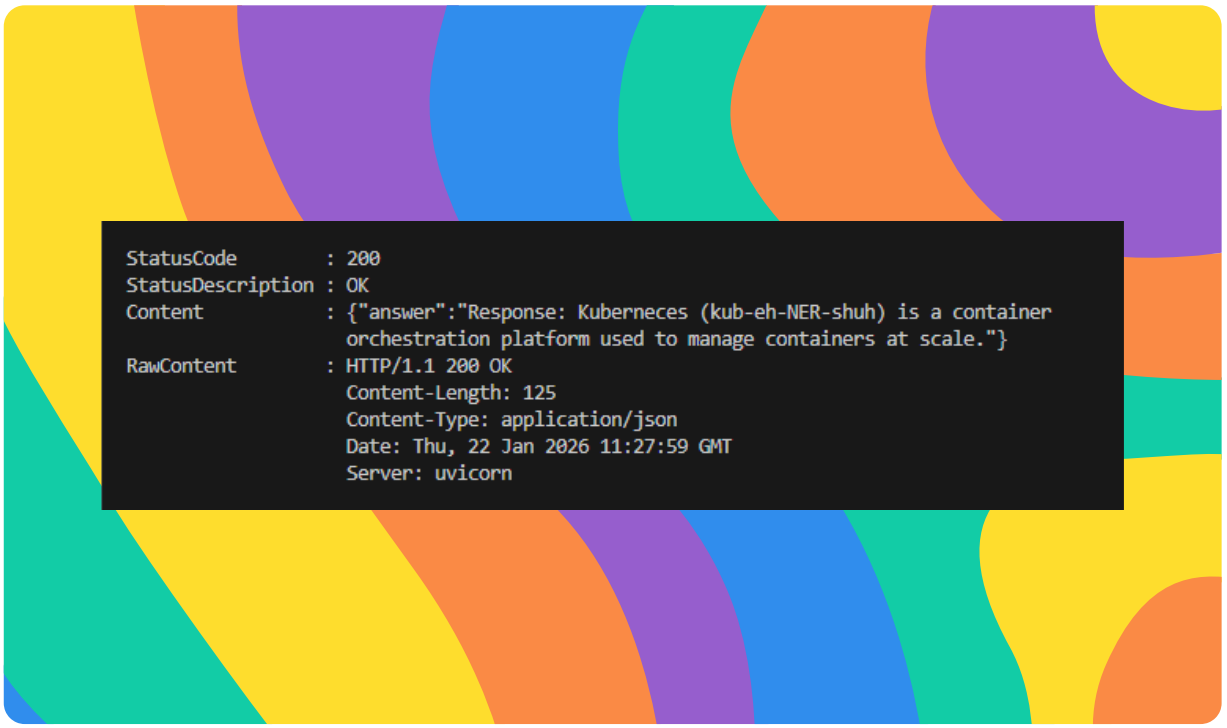
Uvicorn is used for running the FastAPI application as a server and listening for incoming requests. Ollama is used for communicating with the local Ollama service, allowing the API to send prompts to the language model and receive generated answers.



Local API working

I tested that my API works by starting the FastAPI server with Uvicorn and then sending a POST request to the /query endpoint using curl or Invoke-WebRequest with a test question about Kubernetes.

The local API responded with a JSON object containing an AI generated answer pulled from the Chroma database and produced by the TinyLlama model through Ollama. This confirms that the FastAPI server is running, the Chroma database is accessible, Ollama is active, the language model is available, and all components of the RAG pipeline are correctly integrated and functioning.



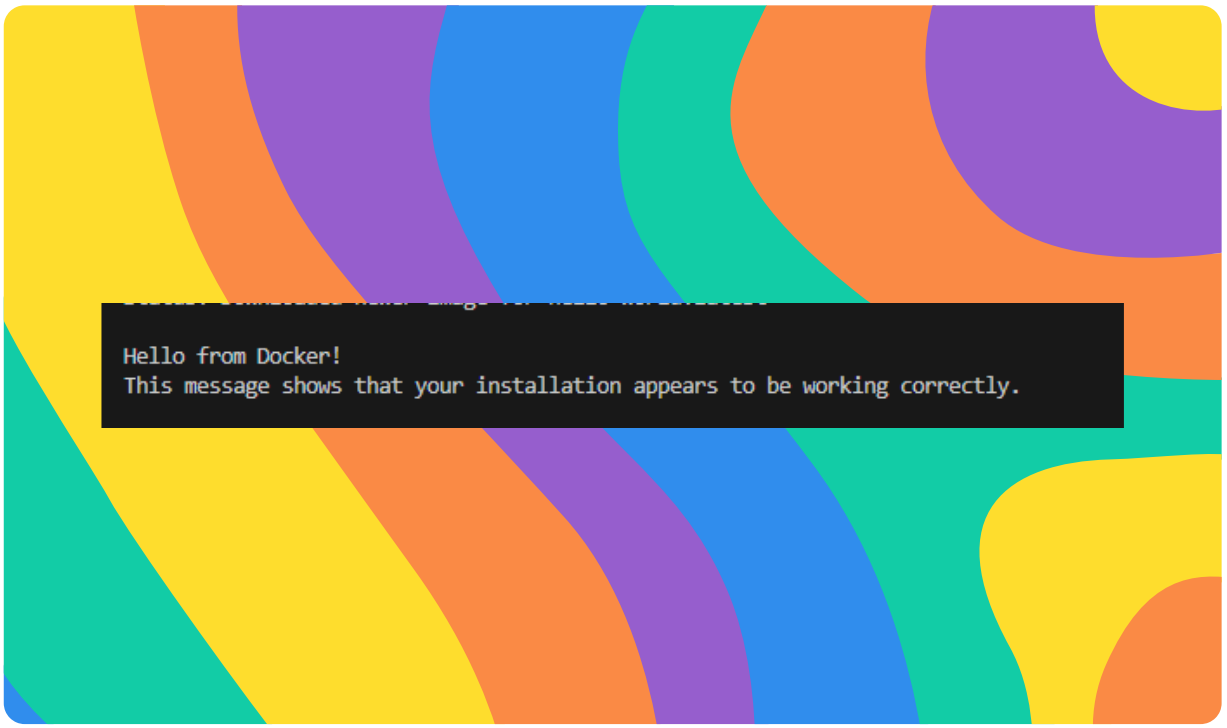
Installing Docker Desktop

Docker Desktop setup

Docker Desktop is a user friendly application that installs and runs Docker Engine, the Docker CLI, and the required virtualization layer so containers can run on my computer. I installed it because my RAG API depends on specific versions of Python, FastAPI, ChromaDB, the Ollama client, and a pre built embeddings database, and Docker Desktop allows me to package all of these into a single container. Containerization will help my project by eliminating environment mismatch, removing manual setup for other machines, and ensuring the API runs identically on my laptop, a teammate’s system, or a production server.

Docker verification

I verified Docker is working by running the command docker run hello-world in my terminal and waiting for it to execute. The hello-world container proves that Docker can pull images from the internet, create a container, and run it successfully on my system, which confirms that the Docker installation is working correctly.


Hello from Docker!
This message shows that your installation appears to be working correctly.

Creating the Dockerfile

In this step, I’m building a RAG API. RAG stands for Retrieval Augmented Generation, which means the API retrieves relevant context from a database and uses it to generate better answers with a language model. I’m creating files like a Dockerfile to define the environment and dependencies, then building a Docker image of the API so it can be run as a container and tested consistently anywhere.

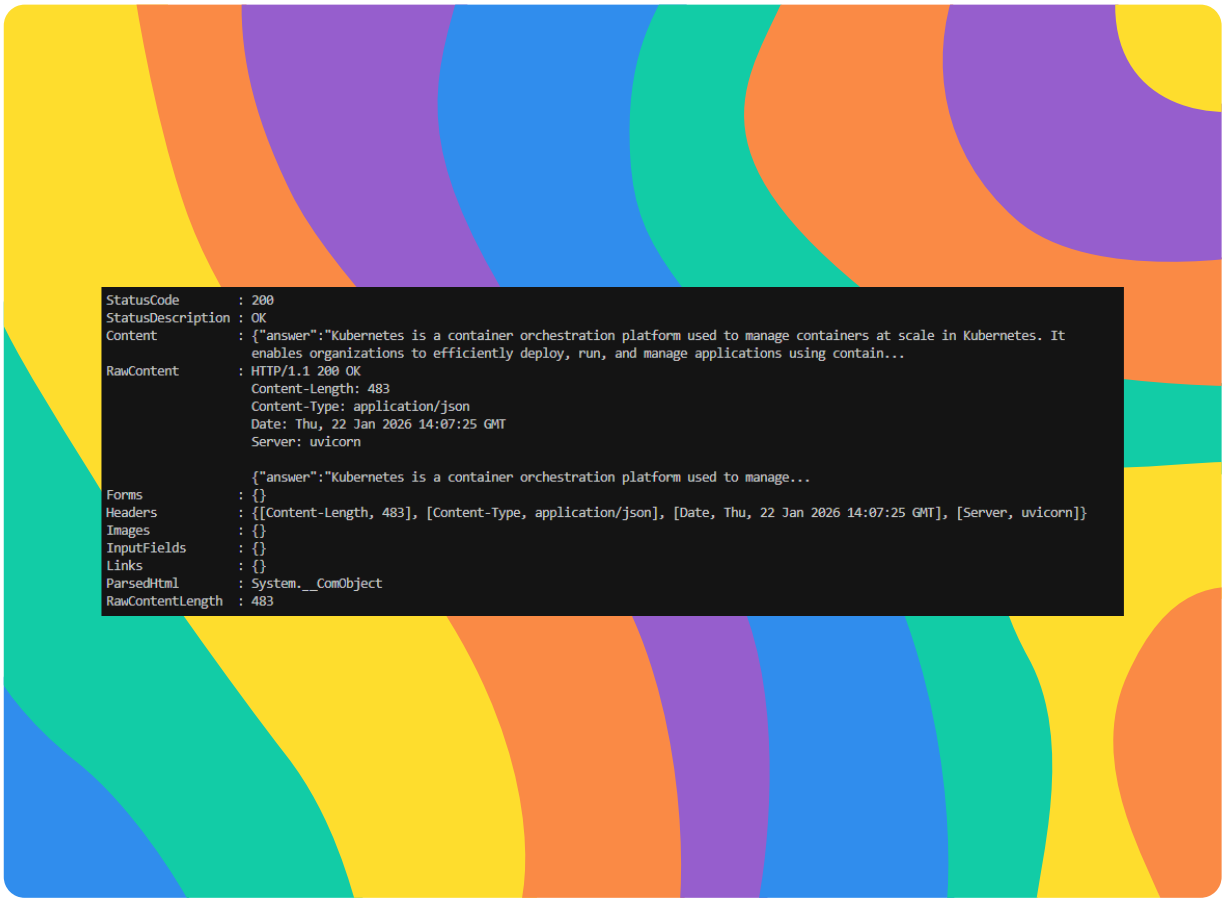
How the Dockerfile works

A Dockerfile is a text file with instructions that tell Docker how to build a container image. The key instructions in my Dockerfile are FROM, WORKDIR, COPY, RUN, EXPOSE, and CMD. FROM tells Docker to start from a Python 3.11 base image that already has Python installed. COPY is used for bringing my app files app.py, embed.py, and k8s.txt into the container. RUN executes commands during the build process, like installing FastAPI, ChromaDB, and Ollama and precomputing embeddings by running embed.py. CMD defines the command that runs when the container starts, which launches the FastAPI server with uvicorn so the RAG API is available on port 8000.

Containerized API test results

Testing the API after containerization proved that my RAG API works correctly inside a Docker container, all dependencies are packaged, and the container can reach Ollama on the host to generate AI answers. The difference between running locally and in Docker is that locally it relies on my system Python and installed packages, whereas Docker provides a consistent, isolated environment with everything included.

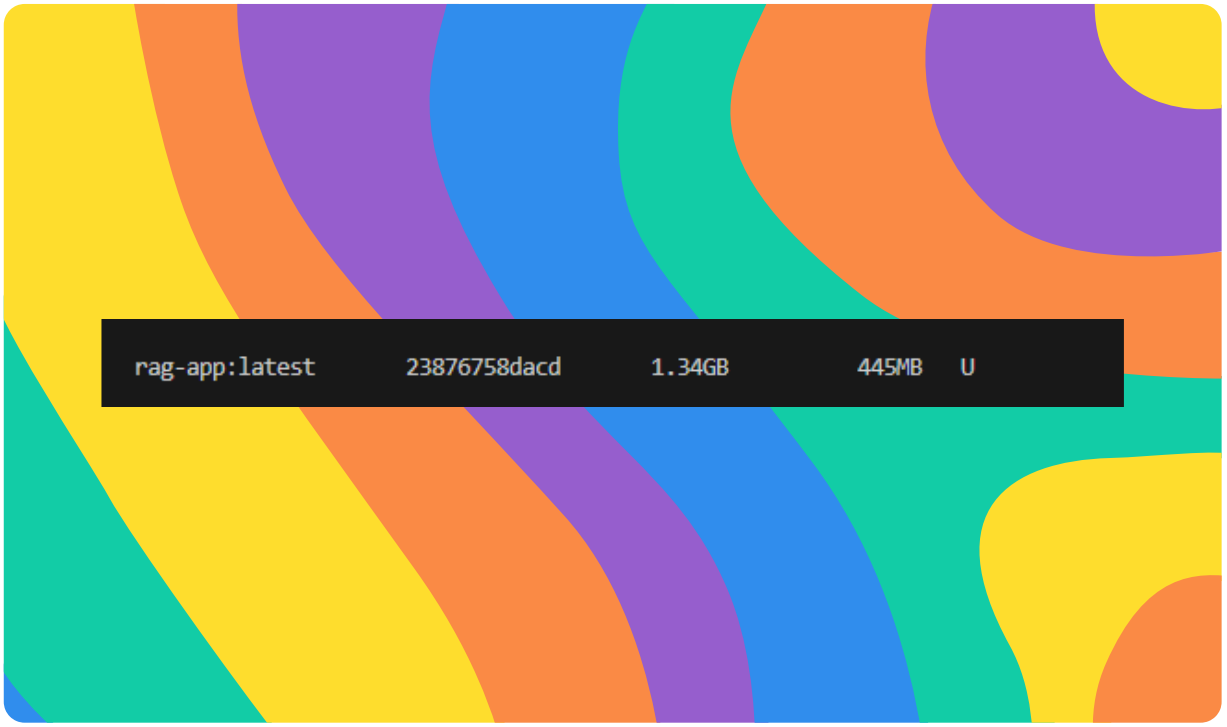
Containerization helps because it ensures the API behaves the same way on any machine, avoids conflicts with system dependencies, and makes deployment predictable.

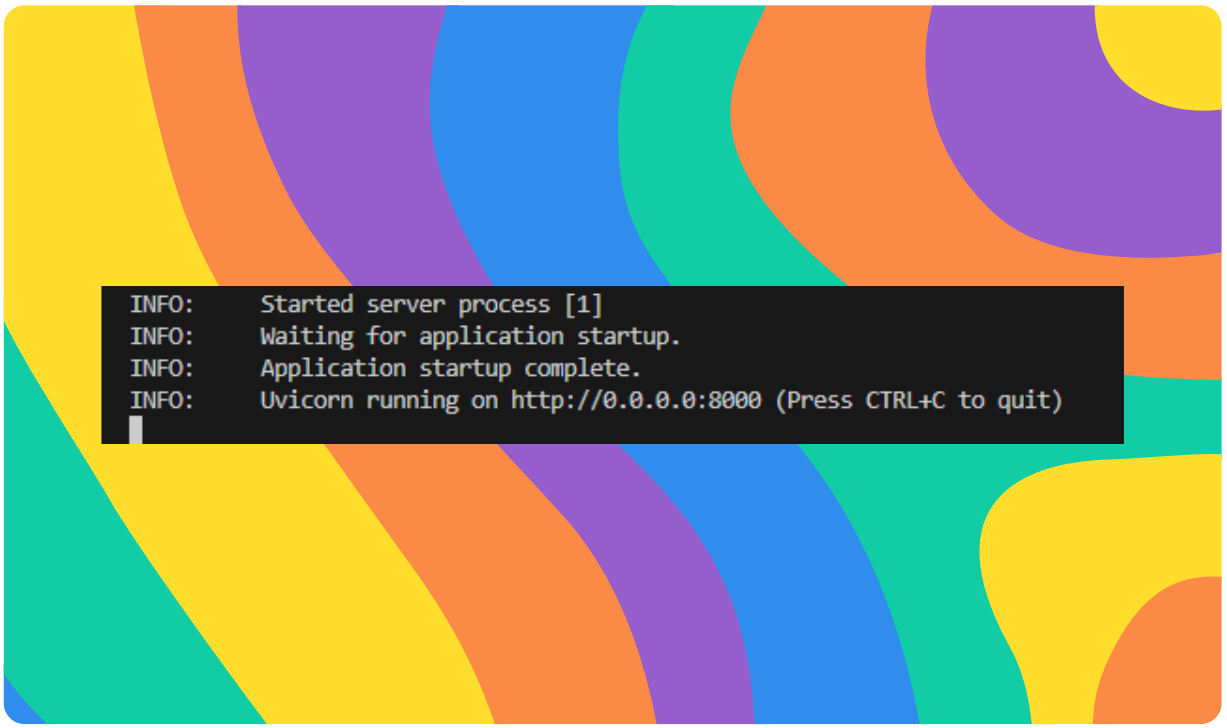


Building and Running the Container

Docker image build complete

Building a Docker image involves Docker reading the Dockerfile top to bottom, executing each instruction, and saving the results as a reusable, read-only image. I verified my Docker image was built successfully by running the docker build -t rag-app . command and waiting for it to complete without errors, then listing my images with docker images and confirming that rag-app appeared with the latest tag. This confirms that my API is now containerized because Docker has a named image that packages my application code, dependencies, and configuration and can be run consistently in a container anywhere Docker is installed.



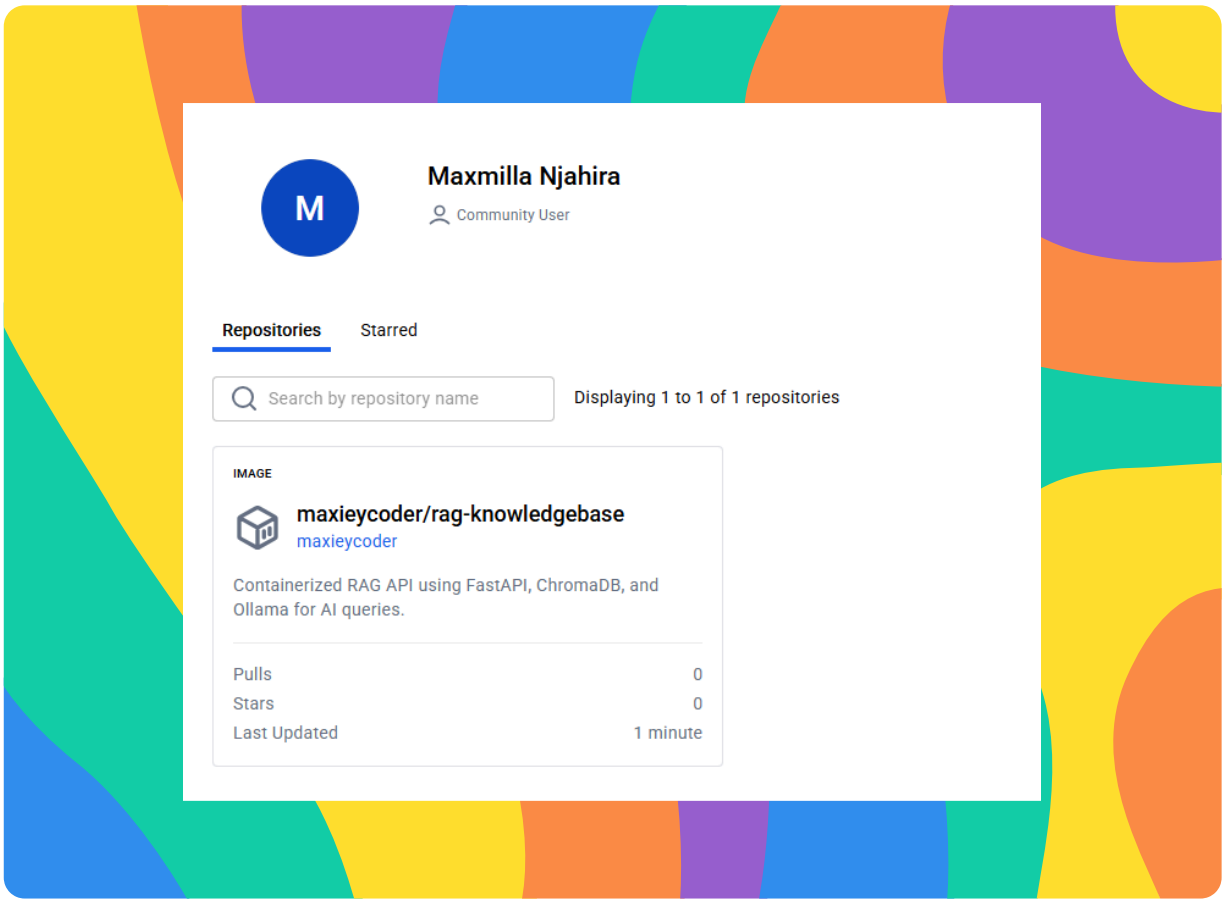

INFO: Started server process [1]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)

Pushing to Docker Hub

In this project extension, I'm pushing to Docker Hub. Docker Hub is a cloud-based registry for storing and sharing Docker images. I'm doing this because it lets me distribute my containerized API publicly, share it with teammates, deploy it to production or cloud platforms, and use it in CI/CD pipelines.

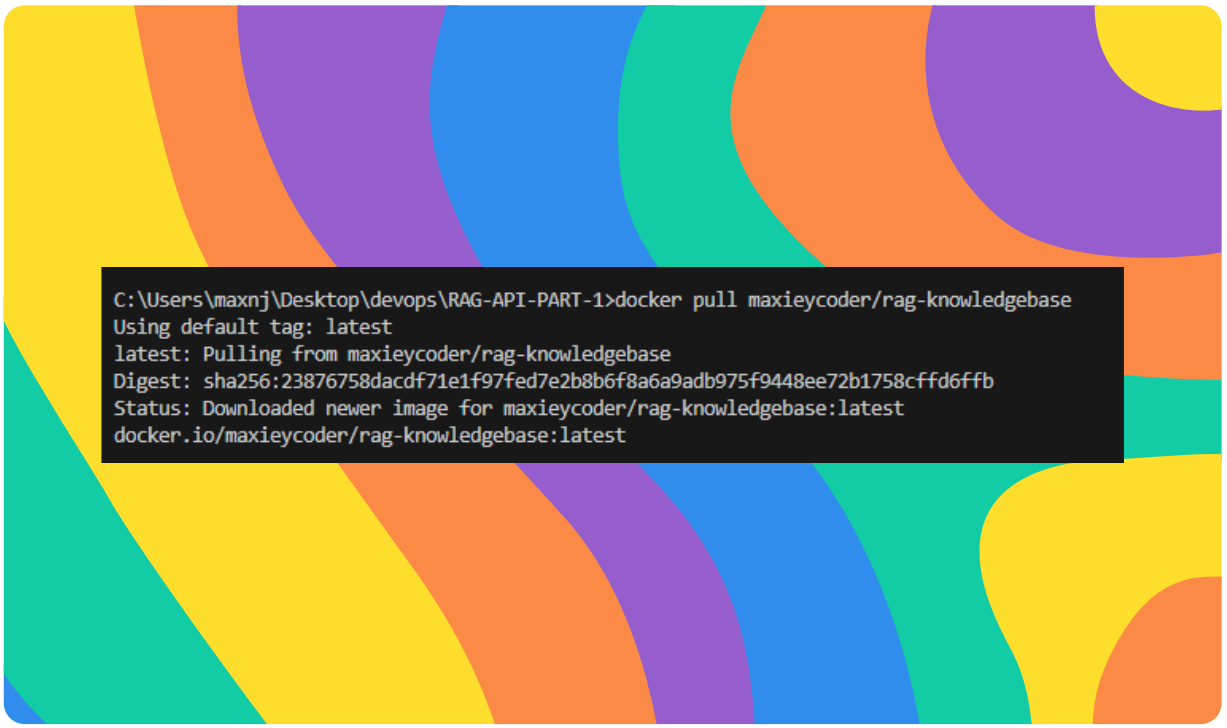
Docker Hub push complete

I pushed to Docker Hub by first logging in with `docker login`, then tagging my local image with my Docker Hub username using `docker tag rag-app maxieycoder/rag-app`, and finally running `docker push maxieycoder/rag-app` to upload the image. Docker Hub is useful because it acts as a cloud-based registry for storing and sharing Docker images, making them accessible from any machine. The advantage of pushing to a registry is that it allows me to share my containerized API with others, deploy it to production, and use it in CI/CD pipelines, ensuring consistent environments everywhere.



Pulling from Docker Hub

Pulling an image from Docker Hub means downloading a pre-built Docker image from the cloud so it can run on your machine without building it locally. When I ran `docker pull maxieycoder/rag-knowledgebase`, Docker downloaded all the image layers from my Docker Hub repository, showing progress for each layer and confirming the image was successfully retrieved. The difference between building locally and pulling from Docker Hub is that building locally requires running all Dockerfile instructions to create the image from scratch, whereas pulling retrieves a ready-to-use image from a remote registry, making it faster and ensuring consistency across machines.


C:\Users\maxnj\Desktop\devops\RAG-API-PART-1>docker pull maxieycoder/rag-knowledgebase
Using default tag: latest
latest: Pulling from maxieycoder/rag-knowledgebase
Digest: sha256:23876758dacdf71e1f97fed7e2b8b6f8a6a9adb975f9448ee72b1758cffd6ffb
Status: Downloaded newer image for maxieycoder/rag-knowledgebase:latest
docker.io/maxieycoder/rag-knowledgebase:latest