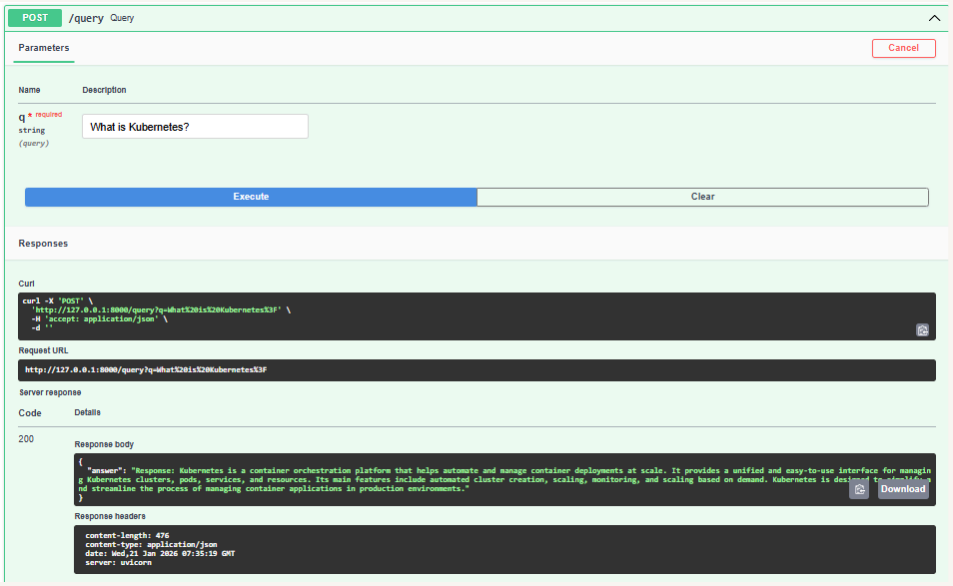
Build a RAG API with FastAPI

BY: MAXMILLA NJAHIRA



Introducing Today's Project!

Today we're building a RAG API from scratch that lets us query documents using natural language. We'll learn how RAG systems work by combining document search with AI-generated answers, understand how Ollama runs local LLMs, and discover how to build and run production-ready APIs using FastAPI. By the end, you'll have a working AI-powered question-answering system running entirely on your laptop, complete with automatic API documentation.

Key services and concepts

Services I used were FastAPI for building the web API with automatic documentation, Uvicorn for running the server, ChromaDB as a vector database for storing and searching document embeddings, and Ollama for running the tinyllama language model locally on my computer. Key concepts I learnt include RAG (Retrieval-Augmented Generation) which combines document search with AI-generated responses, embeddings as numerical representations of text that capture semantic meaning, vector databases for semantic search based on meaning rather than keyword matching, API endpoints and the request/response pattern, virtual environments for isolating project dependencies, and how production APIs enable dynamic content updates through RESTful endpoints like the /add endpoint I created.

Challenges and wins

This project took me approximately 4 hours to complete. The most challenging part was dealing with Python version compatibilities since it strictly required version 3.11, and I initially had Python 3.14 installed which caused several packages like ChromaDB to fail during installation.

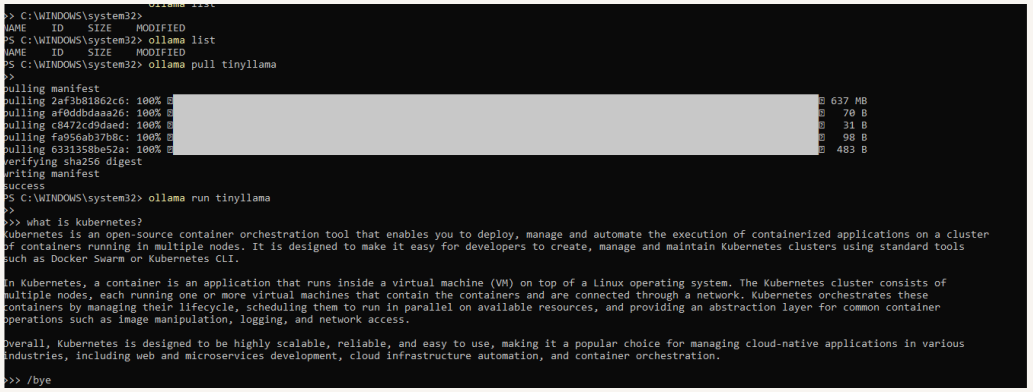
It was most rewarding to learn new concepts in AI, particularly understanding how RAG systems work under the hood, how embeddings enable semantic search, how vector databases differ from traditional databases, and how all these components integrate together to create an intelligent API. This was an exciting learning journey for me and I'm eager to continue learning more about AI and DevOps.

Why I did this project

I did this project because I want to deepen my knowledge in AI and DevOps for the future, and one of the best ways of doing so is by starting this learning journey with high expectations of learning as much as possible. This project absolutely met my goals by giving me hands-on experience building a production-ready RAG API from scratch, understanding how AI-powered applications work behind the scenes, and learning foundational DevOps concepts like virtual environments, API design, and local development workflows that will prepare me for the upcoming projects in containerization, Kubernetes orchestration, and CI/CD automation.

Setting Up Python and Ollama

In this step, I'm setting up Python and Ollama. Python is the language that we will be using to develop the RAG API and Ollama is a local AI model manager that allows us to run and interact with AI models directly on our machine. I need these tools because the RAG API requires Python to handle the backend logic and Ollama to serve the AI models for generating answers based on retrieved documents.



Ollama and tinyllama ready

Ollama is a local AI model manager and server that allows you to run, manage, and interact with large language models on your own machine without relying on external cloud services. I downloaded the tinyllama model because it is lightweight, fast, and efficient for local experimentation while still providing strong language understanding capabilities.

The model will help my RAG API by enabling quick and cost-effective embeddings, retrieval, and generation of relevant responses directly on my system, reducing latency and dependency on external APIs.

Setting Up a Python Workspace

In this step, I'm setting up a Python project workspace by creating a project folder, creating and activating a virtual environment, and installing the required Python dependencies. I need it because it keeps the RAG API code, libraries, and configurations organized and isolated, making development, dependency management, and future maintenance clean and predictable.

Virtual environment

A virtual environment is an isolated Python setup that has its own Python interpreter and installed packages, separate from the rest of the system. I created one for this project to keep all the dependencies for the RAG API contained so they do not interfere with other Python projects on my machine. Once I activate it, any Python commands and package installations apply only to this project. To create a virtual environment, I used Python’s built in venv module to generate a dedicated environment folder for the project.

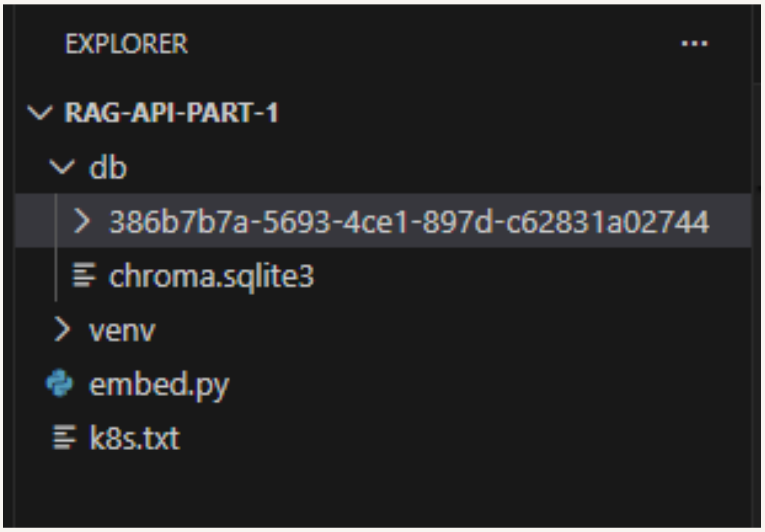
Dependencies

The packages I installed are FastAPI, ChromaDB, Uvicorn, and Ollama. FastAPI is used for building the API that receives user requests and routes them to the correct logic in the application. Chroma is used for storing and searching vector embeddings so the system can retrieve the most relevant documents for a query. Uvicorn is used as the ASGI server that runs the FastAPI application and handles incoming HTTP requests. Ollama is used as the Python client that allows the application to send prompts to a local language model and receive generated responses.

```
chromadb                                 1.4.1
fastapi                                  0.128.0
ollama                                   0.6.1
uvicorn                                  0.40.0
```

Setting Up a Knowledge Base

In this step, I'm creating a knowledge base and converting its content into embeddings so it can be searched by the RAG system. A knowledge base is a collection of written information that the application can retrieve from when answering questions. I need it because the AI model has limited built-in knowledge, and retrieving relevant, up-to-date information first allows it to generate accurate and context-aware answers.



Embeddings created

Embeddings are numerical vector representations of text that capture the meaning of the content rather than the exact words. I created them by running the embedding script, which took my knowledge document and converted each piece of text into vectors using Chroma. The db/ folder contains the stored embeddings, metadata, and index files that Chroma uses to quickly look up and retrieve relevant information.

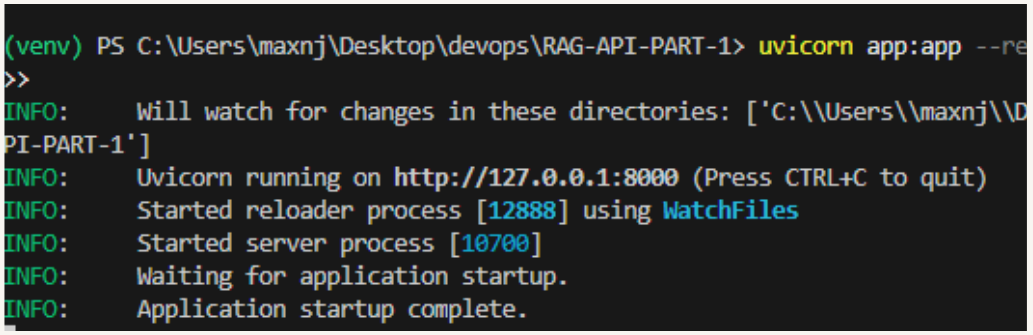
This is important for RAG because the system must retrieve the most relevant knowledge before generating an answer, otherwise the model would be guessing instead of responding based on real data..

Building the RAG API

In this step, I'm building a RAG API. An API is an interface that allows software to send requests and receive responses over the web. FastAPI is a Python web framework that makes it easy to build fast, structured APIs for handling those requests. I'm creating this because exposing the RAG system as an API allows questions to be sent in, relevant documents to be retrieved from Chroma, and AI-generated answers to be returned in a way that other apps or users can easily use.

How the RAG API works

My RAG API works by receiving a question at the /query endpoint, using Chroma to search my knowledge base for the most relevant text, and then sending both the question and that retrieved context to the language model to generate an answer. The main components are FastAPI, which exposes the web endpoint and handles requests, ChromaDB, which retrieves relevant documents from embeddings, and the language model, which uses the retrieved context to generate an accurate answer before sending it back to the user.

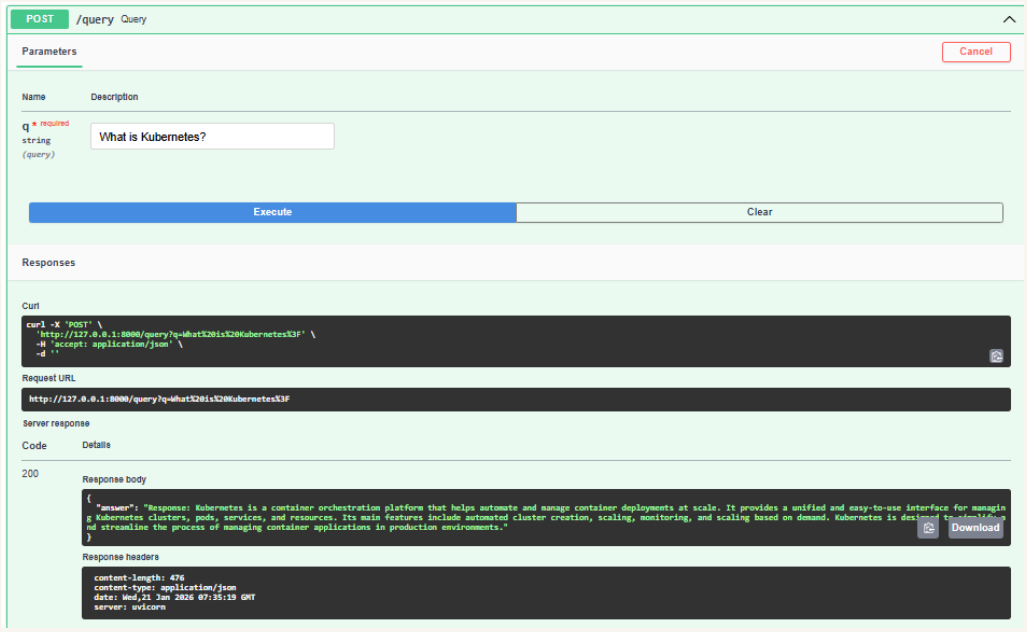

(venv) PS C:\Users\maxnj\Desktop\devops\RAG-API-PART-1> uvicorn app:app --re
>>
INFO: Will watch for changes in these directories: ['C:\\Users\\maxnj\\D
PI-PART-1']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [12888] using WatchFiles
INFO: Started server process [10700]
INFO: Waiting for application startup.
INFO: Application startup complete.

Testing the RAG API

In this step, I'm testing my RAG API. I'll test it using the command line and the FastAPI web interface. Swagger UI is an interactive documentation page automatically generated by FastAPI that shows my API endpoints and lets me send requests from the browser. I'll use it to submit questions to my /query endpoint and confirm that the API returns correct, AI-generated answers from my knowledge base.

API query breakdown

I queried my API by running a POST request from a new terminal using curl or Invoke-RestMethod, targeting the /query endpoint with a question like “What is Kubernetes?”. The command uses the POST method, which means I am sending data to the server so it can process the question instead of just retrieving static data. The API responded with an AI-generated answer that was created by first retrieving relevant context from the Chroma database and then using Ollama’s tinyllama model to generate a response based on that retrieved information.

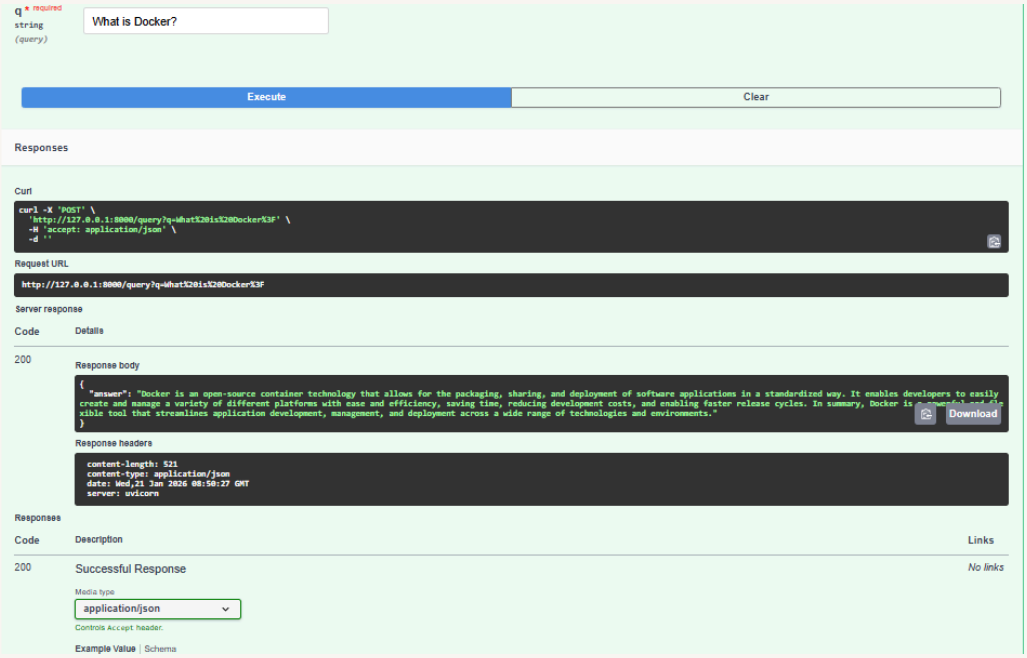


Swagger UI exploration

Swagger UI is an interactive documentation interface automatically generated by FastAPI that displays all your API endpoints, their parameters, and response formats in a visual, user-friendly way. I used it to test my RAG API by clicking "Try it out" on the POST /query endpoint, entering questions like "What is Kubernetes?" directly in the browser, and clicking "Execute" to see formatted JSON responses without writing any curl commands or test scripts. The best part about using Swagger UI was being able to explore and test my API endpoints interactively in real-time, making debugging and experimentation much easier than manually crafting requests.

Adding Dynamic Content

In this project extension, I'm creating a new `/add` POST endpoint that allows dynamic content addition to the knowledge base through the API, enabling real-time updates to Chroma without manually editing files or restarting the application. This makes the RAG system more flexible and production-ready by allowing other applications to expand the knowledge base automatically, users to add information through web interfaces, and the system to stay current without manual intervention, showcasing advanced API design skills that mirror how professional production systems handle dynamic data updates.



Dynamic content endpoint working

The /add endpoint allows me to dynamically add new content to the knowledge base through an API call without manually editing files or restarting the server, by accepting text as a parameter, generating a unique ID using uuid, and storing it directly in the Chroma collection as embeddings. This is useful because it makes the RAG system production-ready by enabling real-time updates, allowing other applications to expand the knowledge base automatically, and making new information immediately searchable through the /query endpoint without any manual intervention or downtime.