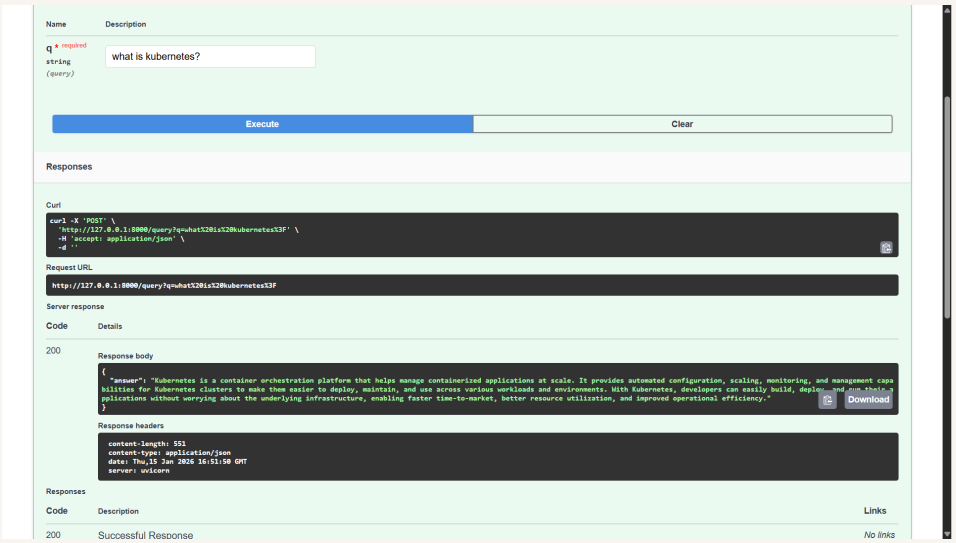
Build a RAG API with FastAPI

Abhinave P.B



Abhinave P.B
NextWork Student

Introducing Today's Project!

In this project, I will demonstrate how to build FastAPI I'm doing this project to learn about RAG bots

Key services and concepts

Services I used were Python, Ollama, FastAPI, Chroma, tinyllama, and Swagger UI. Key concepts I learnt include building a RAG API, integrating a local LLM, creating interactive API documentation, testing endpoints with curl and Swagger UI, and designing a dynamic knowledge base that updates and becomes searchable in real time.

Challenges and wins

This project took me approximately 2 hours. The most challenging part was understanding basics. It was most rewarding to build a FastAPI hands on project

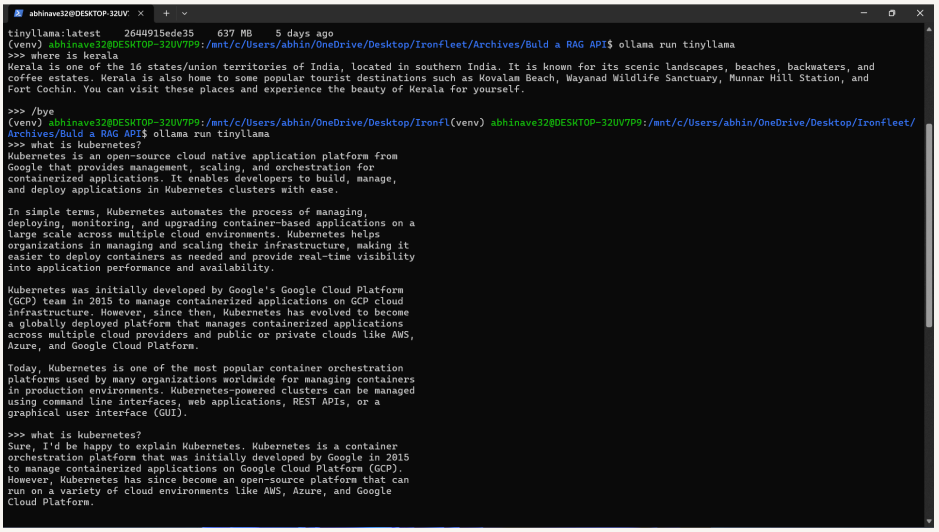
Why I did this project

I did this project because learn something new and yes it certainly meet my goals.

Abhinave P.B
NextWork Student

Setting Up Python and Ollama

In this step, I am setting up Python and Ollama. Python is a popular high-level programming language widely used for backend development and API creation. Ollama is a popular open-source platform that allows large language models (LLMs) to run locally on a device. These tools are required because the RAG (Retrieval-Augmented Generation) API depends on Python for application logic and Ollama for executing the language models locally.



Abhinave P.B
NextWork Student

Ollama and tinyllama ready

Ollama is a local LLM runtime that allows developers to easily download, manage, and run large language models on their own machines without relying on cloud-based services. I downloaded the TinyLLaMA model because it is a lightweight, resource-efficient language model that can run smoothly on systems with limited computational power while still providing reasonable natural language understanding and generation capabilities. The model will help my RAG (Retrieval-Augmented Generation) API by generating accurate, context-aware responses using the information retrieved from external knowledge sources, thereby improving response relevance, reducing hallucinations, and enabling faster, cost-effective local inference.

Abhinave P.B
NextWork Student

Setting Up a Python Workspace

In this step, I am setting up the Python development environment. This includes creating a project folder, creating and activating a Python virtual environment, and installing the required Python dependencies. I need this setup because it helps organize the project, manage dependencies efficiently, and ensure that the RAG API runs in an isolated and stable environment without version conflicts.

Virtual environment

A virtual environment is an isolated Python workspace that allows a project to use its own set of libraries and dependencies without affecting other Python projects on the same system. I created one for this project to avoid dependency conflicts and to ensure consistent package versions required for the RAG API. Once I activate it, any Python packages I install are limited to this project only, helping maintain a clean and stable development environment. To create a virtual environment, I used Python’s built-in venv module.

Dependencies

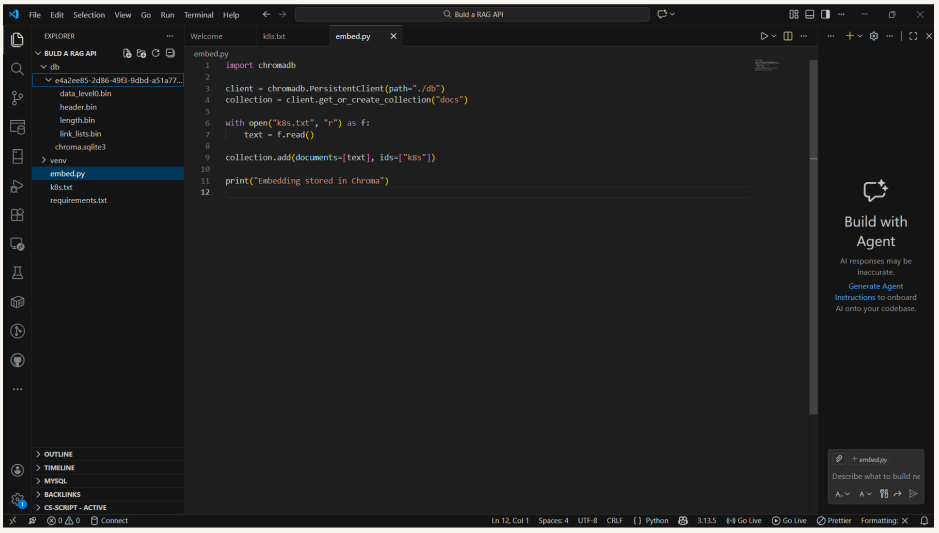
The packages I installed are FastAPI, Chroma, Uvicorn, and Ollama. FastAPI is used for building the backend API of the application, as it allows fast, efficient handling of HTTP requests and responses for the RAG system. Chroma is used as a vector database to store and retrieve embeddings, enabling efficient similarity search over documents for retrieval-augmented generation. Uvicorn is used as an ASGI server to run and serve the FastAPI application with high performance. Ollama is used to run large language models locally, which generate natural language responses based on the retrieved context in the RAG pipeline.

Abhinave P.B
NextWork Student

Abhinave P.B
NextWork Student

Setting Up a Knowledge Base

In this step, I'm creating a knowledge base that stores all the documents and information my RAG system will use to answer user queries. A knowledge base is a structured collection of text data (such as notes, documents, or FAQs) that can be processed, indexed, and searched by an AI model. I need it because the RAG API retrieves relevant information from this knowledge base and provides accurate, context-aware answers instead of relying only on the model's general knowledge.



Abhinave P.B
NextWork Student

Embeddings created

Embeddings are numerical vector representations of text that capture the semantic meaning of the content, allowing the AI to understand and compare text based on meaning rather than exact words. I created them by processing my knowledge base content using an embedding model and converting each text chunk into vectors. The db/ folder contains the stored embeddings and metadata in a vector database, which enables fast similarity search. This is important for RAG because it allows the system to efficiently retrieve the most relevant information from the knowledge base and pass it to the language model to generate accurate and context-aware responses.

Abhinave P.B
NextWork Student

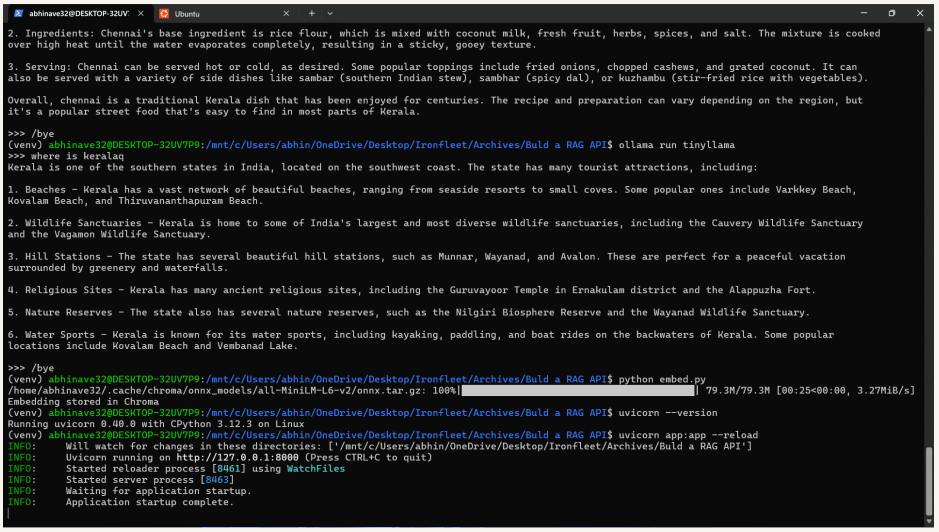
Building the RAG API

In this step, I’m building a RAG API that connects the user’s queries with the knowledge base and the language model. An API (Application Programming Interface) is a way for different software applications to communicate with each other using defined requests and responses. FastAPI is a modern, high-performance Python web framework used to build APIs quickly and efficiently, with automatic documentation and async support. I’m creating this because the RAG API acts as the backend service that receives user queries, retrieves relevant information from the knowledge base, and generates accurate answers using the language model.

How the RAG API works

My RAG API works by combining information retrieval with text generation to answer user queries accurately. When a user sends a question, the API first converts the query into an embedding. This embedding is then compared with the stored embeddings in the vector database to retrieve the most relevant documents from the knowledge base. The retrieved content is passed as context to the language model, which then generates a clear and relevant response based on both the user query and the retrieved data. Main components of my RAG API: Knowledge Base – Stores the source documents that contain domain-specific information. Embedding Model – Converts both documents and user queries into vector representations. Vector Database (Chroma / db folder) – Stores embeddings and enables fast similarity search. Retriever – Finds the most relevant documents based on the query embedding. Language Model (Ollama / TinyLlama) – Generates the final response using retrieved context.

Abhinave P.B
NextWork Student



Abhinave P.B
NextWork Student

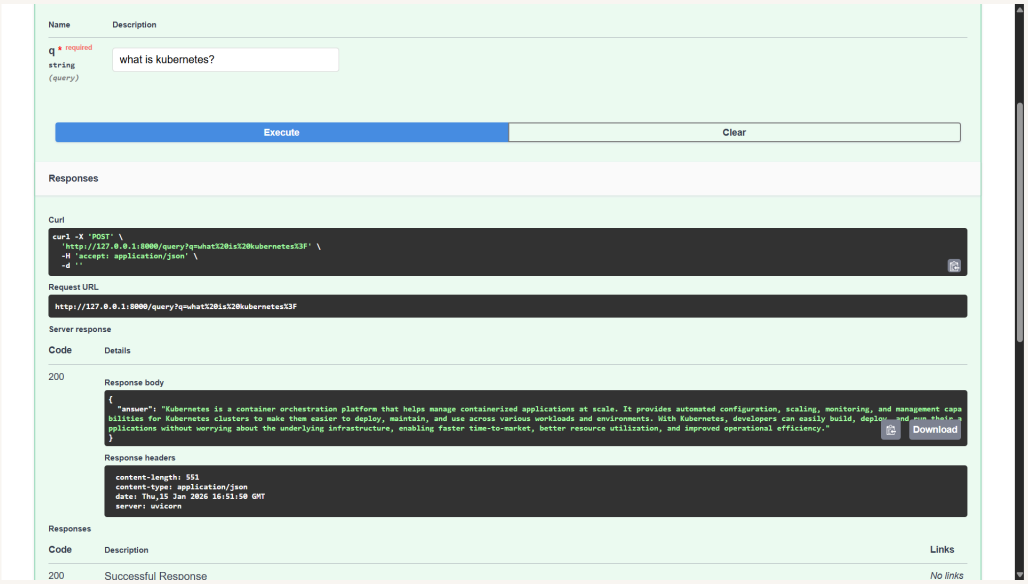
Testing the RAG API

In this step, I'm testing my RAG API to verify that it is working correctly and returning accurate responses. I'll test it using Swagger UI, which is an interactive web-based interface automatically generated by FastAPI. Swagger UI allows developers to view API endpoints, send sample requests, and inspect responses directly from the browser. I'll use it to send test queries to my RAG API endpoint, check whether relevant information is being retrieved from the knowledge base, and confirm that the API returns correct, meaningful answers.

API query breakdown

I queried my API by running the command using curl. The command uses the POST method, which means the request is sent to the server to process data and return a response rather than just retrieving static information. The API responded with the generated answer from the model based on the query provided, confirming that the endpoint is working correctly.

Abhinave P.B
NextWork Student



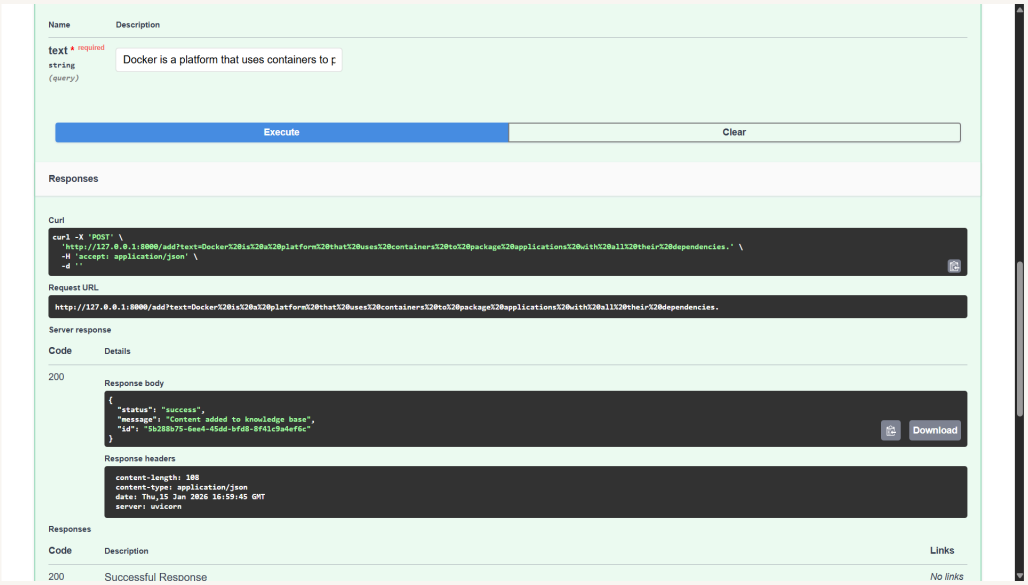
Swagger UI exploration

Swagger UI is an interactive web-based interface that automatically documents and visualizes REST APIs. I used it to test my API endpoints by sending requests directly from the browser and observing the responses without writing any client-side code. The best part about using Swagger UI was that it provided real-time API testing along with clear request and response formats, making it easy to debug and verify the API functionality

Abhinave P.B
NextWork Student

Adding Dynamic Content

In this project extension, I'm going to add a new endpoint to the API that lets us dynamically add content to the knowledge base - the same way production APIs allow users to update data in real-time!



Dynamic content endpoint working

The /add endpoint allows me to submit new content directly to the knowledge base via an API. This is useful because it lets me update the system in real time without manually editing files or restarting the server.

The place to learn & showcase your skills

Check out nextwork.org for more projects