Building a Flask API with LangChain and Azure OpenAI - Part 1

This tutorial will guide you through building a simple Flask API that uses LangChain and Azure OpenAI to create a conversational AI application. By the end of this tutorial, you'll have a working API that can respond to questions using the GPT-4o model hosted in Azure AI Foundry.

Prerequisites

Before starting, make sure you have:

- **Python 3.11** installed (recommended, though 3.9+ should work)
- **Azure OpenAI resource** with GPT-4o deployed
- **Azure OpenAI API key** and other credentials
- **Basic knowledge** of Python and REST APIs

Step 1: Setting Up Your Environment

First, let's create a new directory for our project and set up a virtual environment:

```
# Create a project directory
mkdir langchain-flask-api
cd langchain-flask-api

# Create a virtual environment
python -m venv venv

# Activate the virtual environment
# On Windows:
# venv\Scripts\activate
# On macOS/Linux:
source venv/bin/activate
```

Step 2: Creating the Requirements File

Create a file named `requirements.txt` with the following content:

```
flask==3.0.3
langchain==0.3.0
langchain-openai==0.2.0
python-dotenv==1.0.1
```

Install these dependencies:

```
pip install -r requirements.txt
```

Step 3: Setting Up Environment Variables

Create a `.env` file in your project directory to store your Azure OpenAI credentials:

```
AZURE_OPENAI_API_KEY=your-api-key
AZURE_OPENAI_ENDPOINT=https://your-resource-name.openai.azure.com/
AZURE_OPENAI_API_VERSION=2024-05-01-preview
AZURE_OPENAI_CHAT_DEPLOYMENT_NAME=your-deployment-name
```

Replace the placeholder values with your actual Azure OpenAI credentials: - `your-api-key`: Your Azure OpenAI API key from the Azure Portal - `your-resource-name`: The name of your Azure OpenAI resource - `your-deployment-name`: The name of your GPT-4o deployment

Make sure to add `.env` to your `.gitignore` file to prevent committing sensitive credentials:

```
echo ".env" > .gitignore
```

Step 4: Creating the Flask Application

Now, let's create our main application file. Create a file named `app.py` and we'll build it step by step:

Step 4.1: Import Necessary Libraries

First, let's import all the required libraries:

```
# Import necessary libraries
from flask import Flask, request, jsonify
from langchain_openai import AzureChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from dotenv import load_dotenv
import os
```

These imports provide: - `Flask`: The web framework for building our API - `request` and `jsonify`: For handling HTTP requests and responses - `AzureChatOpenAI`: LangChain's interface to Azure OpenAI - `ChatPromptTemplate`: For creating structured prompts - `load_dotenv`: For loading environment variables from our `.env` file - `os`: For accessing environment variables

Step 4.2: Load Environment Variables and Initialize Flask

Next, let's load our environment variables and initialize our Flask application:

```
# Load environment variables from .env file
load_dotenv()
```

```
# Initialize Flask application
app = Flask(__name__)
```

This code loads the environment variables from our `.env` file and creates a new Flask application instance.

Step 4.3: Configure and Validate Azure OpenAI Environment Variables

Let's add validation to ensure all required environment variables are present:

```
# Section 1: Configure and Validate Azure OpenAI Environment Variables
required_vars = {
    "AZURE_OPENAI_API_KEY": "API key",
    "AZURE_OPENAI_ENDPOINT": "endpoint",
    "AZURE_OPENAI_API_VERSION": "API version",
    "AZURE_OPENAI_CHAT_DEPLOYMENT_NAME": "deployment name"
}
for var, desc in required_vars.items():
    if not os.getenv(var):
        raise ValueError(f"Missing {desc} in environment variables. Check your .env file.")
```

This code checks if all required environment variables are set. If any are missing, it raises an error with a helpful message.

Step 4.4: Initialize the Azure OpenAI Model with LangChain

Now, let's initialize the Azure OpenAI model using LangChain:

```
# Section 2: Initialize the Azure OpenAI Model with LangChain
try:
    llm = AzureChatOpenAI(
        openai_api_version=os.getenv("AZURE_OPENAI_API_VERSION"),
        azure_deployment=os.getenv("AZURE_OPENAI_CHAT_DEPLOYMENT_NAME"),
        temperature=0.7,
        max_tokens=500
    )
except Exception as e:
    raise RuntimeError(f"Failed to initialize AzureChatOpenAI: {str(e)}")
```

This code: - Creates an instance of `AzureChatOpenAI` with our Azure OpenAI credentials - Sets `temperature` to 0.7 for a balance of creativity and consistency - Limits responses to 500 tokens - Includes error handling to provide clear feedback if initialization fails

Step 4.5: Define a Simple Prompt Template

Let's create a prompt template for our AI:

```
# Section 3: Define a Simple Prompt Template
prompt_template = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant providing concise and accurate answers."),
    ("human", "{question}")
])
```

This template: - Sets a system message that defines the AI's behavior - Creates a placeholder for the user's question

Step 4.6: Create a Chain

Now, let's create a simple chain that combines our prompt template and language model:

```
# Section 5: Create a Chain
chain = prompt_template | llm
```

This creates a processing pipeline that: 1. Takes the user's question 2. Formats it using our prompt template 3. Sends it to the language model 4. Returns the model's response

Step 4.7: Define the REST API Endpoint

Let's create an endpoint to handle user questions:

```
# Section 5: Define the REST API Endpoint
@app.route('/ask', methods=['POST'])
def ask_question():
    """
    REST API endpoint to ask a question to GPT-4o.
    Expects a JSON payload with 'question' field.
    Returns the model's response as JSON.
    """
    try:
        # Get JSON data from the request
        data = request.get_json()
        if not data or 'question' not in data:
            return jsonify({"error": "Missing 'question' in request body"}), 400

        # Extract question from the request
        question = data['question']
        print(f"Question received: {question}")

        # Invoke the chain with the user's question
        response = chain.invoke({
            "question": question
        })
        print(f"Response: {response.content}")
        # Return the response
        return jsonify({
            "answer": response.content,
            "status": "success"
        }), 200

    except KeyError as e:
        return jsonify({"error": f"KeyError: {str(e)}"}), 500
    except Exception as e:
        return jsonify({"error": f"Unexpected error: {type(e).__name__}: {str(e)}"}), 500
```

This endpoint: - Accepts POST requests to `/ask` - Extracts the question from the JSON request body - Validates that a question was provided - Sends the question to our LangChain pipeline - Returns the AI's response as JSON - Includes error handling for various failure scenarios

Step 4.8: Run the Flask Application

Finally, let's add the code to run our Flask application:

```
# Section 7: Run the Flask Application
if __name__ == '__main__':
    # Start the Flask server on port 3000
    app.run(host='0.0.0.0', port=3000, debug=True)
```

This code: - Starts the Flask server on port 3000 - Binds to all network interfaces (`0.0.0.0`) - Enables debug mode for easier development

Step 5: Testing the API

Now that we have built our API, let's test it using curl commands:

1. Start the application:

```
python app.py
```

2. Send a test question:

```
curl -X POST http://localhost:3000/ask -H "Content-Type: application/json" -d '{"question":
```

You should receive a response like:

```
{
  "answer": "The capital of France is Paris.",
  "status": "success"
}
```

3. Try another question:

```
curl -X POST http://localhost:3000/ask -H "Content-Type: application/json" -d '{"question":
```

Conclusion

Congratulations! You've successfully built a simple Flask API that uses LangChain and Azure OpenAI to answer questions. This API provides a foundation for more complex applications.

In the next tutorial (Part 2), we'll enhance this API by adding chat history functionality, allowing for multi-turn conversations and maintaining context across multiple interactions. We'll implement session management to support multiple users and add the ability to start new conversations.

Stay tuned for Part 2!