# ChatBot API Tutorial - Part 3

## Requirements

Before starting this tutorial, make sure you have the following packages installed:

pip install flask langchain langchain\_openai python-dotenv

All other dependencies we’ll use in this tutorial are either part of the Python standard library or included with these packages:

* werkzeug.security (for password hashing) is included with Flask
* functools.wraps is part of the Python standard library
* flask.session, flask.redirect, and flask.url\_for are all part of Flask

Make sure you have your Azure OpenAI credentials set as environment variables:

export AZURE\_OPENAI\_API\_KEY=your\_api\_key\_here  
export AZURE\_OPENAI\_ENDPOINT=your\_endpoint\_here  
export AZURE\_OPENAI\_API\_VERSION=2025-01-01-preview

## Review of Tutorial Part 2

In Tutorial Part 2, we enhanced our Flask-based ChatBot API by adding conversation memory and multi-user support. We implemented a chat history feature that stores the 10 most recent interactions for context-aware conversations. We then expanded this to support multiple users through session IDs, where each user has their own conversation history. The application now includes endpoints for managing chat histories, listing active sessions, and generating unique session IDs. These improvements transformed our stateless API into a more interactive and personalized chatbot experience.

## Section 1: Adding Conversation Persistence Between Server Restarts

Currently, our chatbot loses all conversation history when the server restarts. In this section, we’ll implement a simple file-based persistence mechanism to save and load chat histories.

### Step 1: Add necessary imports for file operations

First, let’s add the necessary imports for file operations. Add these to the imports section of your app.py file:

# Add these imports to the existing imports  
import json  
import os  
from datetime import datetime  
from dotenv import load\_dotenv

These imports provide the functionality we need for file operations. The json module allows us to serialize and deserialize Python objects to and from JSON format. The os module provides functions for interacting with the operating system, including file and directory operations. The datetime module provides classes for manipulating dates and times. The load\_dotenv function loads environment variables from a .env file, which is where we’ll store our Azure OpenAI credentials.

### Step 2: Initialize Azure OpenAI client

Before we implement conversation persistence, let’s make sure our Azure OpenAI client is properly initialized. Add or update the following code in your app.py file:

# Load environment variables from .env file  
load\_dotenv()  
  
# Validate Azure OpenAI Environment Variables  
required\_vars = {  
 "AZURE\_OPENAI\_API\_KEY": "API key",  
 "AZURE\_OPENAI\_API\_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.")  
  
# 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 does several important things:

1. It loads environment variables from your .env file, which should contain your Azure OpenAI credentials.
2. It validates that all required environment variables are present, providing helpful error messages if any are missing.
3. It initializes the AzureChatOpenAI client with the appropriate configuration, including the API version, deployment name, temperature (which controls randomness), and maximum token length for responses.

Make sure you have a .env file in your project directory with the following variables:

AZURE\_OPENAI\_API\_KEY=your\_api\_key\_here  
AZURE\_OPENAI\_API\_ENDPOINT=your\_endpoint\_here  
AZURE\_OPENAI\_API\_VERSION=2025-01-01-preview  
AZURE\_OPENAI\_CHAT\_DEPLOYMENT\_NAME=your\_deployment\_name\_here

### Step 3: Define constants for persistence

Now, let’s define constants for our persistence mechanism. Add these after the imports section:

# Add these constants after the imports  
HISTORY\_DIR = "chat\_histories"  
HISTORY\_FILE = os.path.join(HISTORY\_DIR, "chat\_histories.json")  
  
# Create the history directory if it doesn't exist  
os.makedirs(HISTORY\_DIR, exist\_ok=True)

Here we define two constants: HISTORY\_DIR specifies the directory where we’ll store our chat history files, and HISTORY\_FILE is the full path to our JSON file. The os.makedirs() function creates the directory if it doesn’t exist, with exist\_ok=True preventing errors if the directory already exists.

### Step 4: Add functions to save and load chat histories

Next, let’s add functions to save and load chat histories. Add these functions after the constants:

def save\_chat\_histories():  
 """  
 Save all chat histories to a JSON file.  
 """  
 try:  
 # Convert datetime objects to strings for JSON serialization  
 serializable\_histories = {}  
 for session\_id, history in chat\_histories.items():  
 serializable\_histories[session\_id] = history  
  
 with open(HISTORY\_FILE, 'w') as f:  
 json.dump(serializable\_histories, f, indent=2)  
 print(f"Chat histories saved to {HISTORY\_FILE}")  
 return True  
 except Exception as e:  
 print(f"Error saving chat histories: {str(e)}")  
 return False

The save\_chat\_histories() function serializes our chat histories dictionary to JSON and writes it to the file. It creates a copy of the chat histories to ensure proper JSON serialization. The function uses a try-except block to handle any errors that might occur during the saving process, returning True if successful and False otherwise.

def load\_chat\_histories():  
 """  
 Load all chat histories from a JSON file.  
 Returns a dictionary of chat histories.  
 """  
 global chat\_histories  
 try:  
 if os.path.exists(HISTORY\_FILE):  
 with open(HISTORY\_FILE, 'r') as f:  
 chat\_histories = json.load(f)  
 print(f"Chat histories loaded from {HISTORY\_FILE}")  
 return True  
 else:  
 print(f"No chat history file found at {HISTORY\_FILE}")  
 return False  
 except Exception as e:  
 print(f"Error loading chat histories: {str(e)}")  
 return False

The load\_chat\_histories() function reads the JSON file and loads the chat histories into our global chat\_histories variable. It first checks if the file exists, and if not, it simply returns False. Like the save function, it uses a try-except block to handle any errors during loading.

### Step 4: Load chat histories on startup

Now, let’s load chat histories when the application starts. Add this code after the chat\_histories initialization:

# Replace the existing chat\_histories initialization with this  
chat\_histories = {}  
MAX\_HISTORY\_LENGTH = 10  
  
# Load chat histories on startup  
load\_chat\_histories()

This code initializes our chat\_histories dictionary and sets the maximum history length. Then it calls our load\_chat\_histories() function to load any previously saved chat histories when the application starts. This ensures that conversations persist across server restarts.

### Step 5: Update endpoints to save chat histories

Next, let’s update our endpoints to save chat histories after modifications. We’ll need to modify the ask\_question, clear\_history, and clear\_all\_history functions.

First, update the ask\_question function by adding a call to save\_chat\_histories() just before the return statement:

# Add this line before the return statement in ask\_question  
save\_chat\_histories()

This ensures that whenever a new question is asked and answered, the updated chat history is saved to the file.

Next, update the clear\_history function by adding a call to save\_chat\_histories() just before the return statement:

# Add this line before the return statement in clear\_history  
save\_chat\_histories()

This ensures that when a specific chat history is cleared, the change is saved to the file.

Finally, update the clear\_all\_history function by adding a call to save\_chat\_histories() just before the return statement:

# Add this line before the return statement in clear\_all\_history  
save\_chat\_histories()

This ensures that when all chat histories are cleared, the change is saved to the file.

### Step 7: Add an endpoint to manually save chat histories

Let’s add an endpoint to manually save chat histories:

@app.route('/save-histories', methods=['POST'])  
def save\_histories\_endpoint():  
 """  
 REST API endpoint to manually save all chat histories.  
 Returns a confirmation message.  
 """  
 success = save\_chat\_histories()  
 if success:  
 return jsonify({  
 "message": "Chat histories saved successfully",  
 "status": "success"  
 }), 200  
 else:  
 return jsonify({  
 "message": "Failed to save chat histories",  
 "status": "error"  
 }), 500

This endpoint provides a way to manually trigger the saving of chat histories. It calls our save\_chat\_histories() function and returns an appropriate response based on whether the save was successful. This can be useful for ensuring that chat histories are saved at specific points, such as before a planned server shutdown.

### Testing Conversation Persistence

Now that we’ve implemented conversation persistence, let’s test it using curl commands:

1. Start your Flask application:

python app.py

1. Create a new session and ask some questions:

# Generate a new session ID  
curl -X GET http://localhost:3000/generate-session

1. Use the session ID to ask questions:

# Replace YOUR\_SESSION\_ID with the session ID from the previous command  
curl -X POST http://localhost:3000/ask \  
 -H "Content-Type: application/json" \  
 -d '{"question":"What is artificial intelligence?", "session\_id":"YOUR\_SESSION\_ID"}'

1. Restart your Flask application:

# Press Ctrl+C to stop the application, then start it again  
python app.py

1. Verify that your chat history is still available:

# Replace YOUR\_SESSION\_ID with the same session ID  
curl -X GET "http://localhost:3000/history?session\_id=YOUR\_SESSION\_ID"

1. Ask a follow-up question to verify that context is maintained:

# Replace YOUR\_SESSION\_ID with the same session ID  
curl -X POST http://localhost:3000/ask \  
 -H "Content-Type: application/json" \  
 -d '{"question":"What are its main applications?", "session\_id":"YOUR\_SESSION\_ID"}'

## Section 2: Implementing User Identification and Authentication

In this section, we’ll enhance our chatbot with basic user identification and authentication. We’ll implement a simple username/password authentication system and associate sessions with specific users.

### Step 1: Add necessary imports and constants

First, let’s add the necessary imports and constants for user authentication. Add these to your app.py file:

# Add these imports to the existing imports  
from werkzeug.security import generate\_password\_hash, check\_password\_hash  
from functools import wraps  
from flask import session, redirect, url\_for  
  
# Add these constants after the existing constants  
USERS\_FILE = os.path.join(HISTORY\_DIR, "users.json")  
SECRET\_KEY = os.urandom(24) # Generate a random secret key for session management  
  
# Update Flask app configuration  
app.config['SECRET\_KEY'] = SECRET\_KEY

These imports provide the functionality we need for user authentication. The werkzeug.security module provides functions for securely hashing and checking passwords. The wraps decorator from functools helps us create route decorators. The additional Flask imports allow us to use sessions for maintaining user login state. We also define a path for storing user data and generate a random secret key for securing Flask sessions.

### Step 2: Add functions to manage users

Next, let’s add functions to manage users. Add these functions after the chat history functions:

def save\_users(users):  
 """  
 Save users dictionary to a JSON file.  
 """  
 try:  
 with open(USERS\_FILE, 'w') as f:  
 # Don't save the actual password, only the hash  
 serializable\_users = {}  
 for username, user\_data in users.items():  
 serializable\_users[username] = {  
 'password\_hash': user\_data['password\_hash'],  
 'sessions': user\_data['sessions']  
 }  
 json.dump(serializable\_users, f, indent=2)  
 print(f"Users saved to {USERS\_FILE}")  
 return True  
 except Exception as e:  
 print(f"Error saving users: {str(e)}")  
 return False

The save\_users() function serializes our users dictionary to a JSON file. It creates a copy of the users data to ensure we’re only saving the password hash and session IDs, not any other potentially sensitive information. Like our chat history functions, it uses a try-except block to handle any errors during saving.

def load\_users():  
 """  
 Load users from a JSON file.  
 Returns a dictionary of users.  
 """  
 try:  
 if os.path.exists(USERS\_FILE):  
 with open(USERS\_FILE, 'r') as f:  
 users = json.load(f)  
 print(f"Users loaded from {USERS\_FILE}")  
 return users  
 else:  
 print(f"No users file found at {USERS\_FILE}, creating default users")  
 # Create a default admin user  
 default\_users = {  
 'admin': {  
 'password\_hash': generate\_password\_hash('admin'),  
 'sessions': []  
 }  
 }  
 save\_users(default\_users)  
 return default\_users  
 except Exception as e:  
 print(f"Error loading users: {str(e)}")  
 return {'admin': {'password\_hash': generate\_password\_hash('admin'), 'sessions': []}}  
  
# Initialize users dictionary  
users = load\_users()

The load\_users() function reads the users from the JSON file. If the file doesn’t exist, it creates a default admin user with username ‘admin’ and password ‘admin’. This ensures there’s always at least one user account available. We then initialize our global users dictionary by calling this function.

def login\_required(f):  
 """  
 Decorator to require login for specific routes.  
 """  
 @wraps(f)  
 def decorated\_function(\*args, \*\*kwargs):  
 if 'username' not in session:  
 return jsonify({"error": "Authentication required"}), 401  
 return f(\*args, \*\*kwargs)  
 return decorated\_function

The login\_required decorator function will be used to protect routes that require authentication. It checks if a username exists in the current session, and if not, it returns a 401 Unauthorized response. This decorator can be applied to any route that should only be accessible to logged-in users.

### Step 3: Add authentication endpoints

Now, let’s add endpoints for user registration, login, and logout:

@app.route('/register', methods=['POST'])  
def register():  
 """  
 REST API endpoint to register a new user.  
 Expects a JSON payload with 'username' and 'password'.  
 Returns a confirmation message.  
 """  
 global users  
  
 try:  
 data = request.get\_json()  
 if not data or 'username' not in data or 'password' not in data:  
 return jsonify({"error": "Missing username or password in request body"}), 400  
  
 username = data['username']  
 password = data['password']  
  
 if username in users:  
 return jsonify({"error": "Username already exists"}), 400  
  
 users[username] = {  
 'password\_hash': generate\_password\_hash(password),  
 'sessions': []  
 }  
  
 save\_users(users)  
  
 return jsonify({  
 "message": f"User {username} registered successfully",  
 "status": "success"  
 }), 201  
  
 except Exception as e:  
 return jsonify({"error": f"Unexpected error: {type(e).\_\_name\_\_}: {str(e)}"}), 500

The /register endpoint allows new users to create an account. It expects a JSON payload with username and password fields. After validating the input, it checks if the username already exists. If not, it creates a new user entry with a hashed password (never storing the actual password) and an empty sessions list. Finally, it saves the updated users dictionary to the file.

@app.route('/login', methods=['POST'])  
def login():  
 """  
 REST API endpoint to log in a user.  
 Expects a JSON payload with 'username' and 'password'.  
 Returns a confirmation message and session ID.  
 """  
 global users  
  
 try:  
 data = request.get\_json()  
 if not data or 'username' not in data or 'password' not in data:  
 return jsonify({"error": "Missing username or password in request body"}), 400  
  
 username = data['username']  
 password = data['password']  
  
 if username not in users:  
 return jsonify({"error": "Invalid username or password"}), 401  
  
 if not check\_password\_hash(users[username]['password\_hash'], password):  
 return jsonify({"error": "Invalid username or password"}), 401  
  
 # Generate a new session ID for this user  
 new\_session\_id = str(uuid.uuid4())  
 users[username]['sessions'].append(new\_session\_id)  
  
 # Store username in session  
 session['username'] = username  
 session['session\_id'] = new\_session\_id  
  
 save\_users(users)  
  
 return jsonify({  
 "message": f"User {username} logged in successfully",  
 "status": "success",  
 "session\_id": new\_session\_id  
 }), 200  
  
 except Exception as e:  
 return jsonify({"error": f"Unexpected error: {type(e).\_\_name\_\_}: {str(e)}"}), 500

The /login endpoint authenticates users. It validates the username and password against our stored users. For security, we use the same error message for both invalid username and invalid password to prevent username enumeration. If authentication is successful, we generate a new session ID, add it to the user’s sessions list, and store the username and session ID in the Flask session. We then save the updated users dictionary and return the session ID to the client.

@app.route('/logout', methods=['POST'])  
@login\_required  
def logout():  
 """  
 REST API endpoint to log out a user.  
 Expects a JSON payload with 'session\_id'.  
 Returns a confirmation message.  
 """  
 global users  
  
 try:  
 data = request.get\_json() or {}  
 session\_id = data.get('session\_id', session.get('session\_id'))  
 username = session.get('username')  
  
 if username in users and session\_id in users[username]['sessions']:  
 users[username]['sessions'].remove(session\_id)  
 save\_users(users)  
  
 session.pop('username', None)  
 session.pop('session\_id', None)  
  
 return jsonify({  
 "message": "Logged out successfully",  
 "status": "success"  
 }), 200  
  
 except Exception as e:  
 return jsonify({"error": f"Unexpected error: {type(e).\_\_name\_\_}: {str(e)}"}), 500

The /logout endpoint handles user logout. It’s protected by our @login\_required decorator to ensure only authenticated users can access it. The endpoint removes the session ID from the user’s sessions list and clears the Flask session. This effectively terminates the user’s session, requiring them to log in again for future authenticated requests.

### Step 4: Update the ask\_question endpoint to check authentication

Now, let’s update the ask\_question endpoint to check if the session ID is associated with an authenticated user:

@app.route('/ask', methods=['POST'])  
def ask\_question():  
 """  
 REST API endpoint to ask a question to Azure OpenAI.  
 Expects a JSON payload with 'question' field and optional 'session\_id'.  
 Returns the model's response as JSON along with the chat history for that session.  
 """  
 global chat\_histories, users  
  
 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 and session\_id from the request  
 question = data['question']  
 session\_id = data.get('session\_id', session.get('session\_id', 'default\_session'))  
  
 # Check if this is an authenticated session  
 is\_authenticated = False  
 session\_username = None  
  
 # Check if the session is in the flask session  
 if 'username' in session and session.get('session\_id') == session\_id:  
 is\_authenticated = True  
 session\_username = session.get('username')  
 else:  
 # Check if the session is associated with any user  
 for username, user\_data in users.items():  
 if session\_id in user\_data['sessions']:  
 is\_authenticated = True  
 session\_username = username  
 break  
  
 print(f"Question received from {'authenticated ' + session\_username if is\_authenticated else 'unauthenticated'} session {session\_id}: {question}")

In this first part of the updated endpoint, we extract the question and session ID from the request. We then check if the session is authenticated by first looking at the Flask session (for browser-based clients) and then checking if the session ID is associated with any user in our users dictionary. This allows both browser-based and API-based authentication to work.

# Initialize session history if it doesn't exist  
 if session\_id not in chat\_histories:  
 chat\_histories[session\_id] = []  
  
 # Get the chat history for this session  
 session\_history = chat\_histories[session\_id]  
  
 # Create context from chat history  
 context = ""  
 if session\_history:  
 context = "Previous conversation:\n"  
 for entry in session\_history:  
 context += f"Human: {entry['question']}\nAI: {entry['answer']}\n"  
  
 # Prepare the question with context if there's history  
 contextualized\_question = question  
 if context:  
 contextualized\_question = f"{context}\nHuman: {question}"  
  
 # Invoke the chain with the user's question  
 response = chain.invoke({  
 "question": contextualized\_question  
 })  
 print(f"Response for session {session\_id}: {response.content}")

In this middle section, we prepare and process the question. We get the chat history for the session, create a context from previous conversations, and invoke the language model with the contextualized question. This part is similar to our previous implementation but will now track which user asked each question.

# Update chat history for this session  
 chat\_histories[session\_id].append({  
 "question": question,  
 "answer": response.content,  
 "timestamp": str(datetime.now()),  
 "user": session\_username if is\_authenticated else "anonymous"  
 })  
  
 # Limit chat history to MAX\_HISTORY\_LENGTH entries  
 if len(chat\_histories[session\_id]) > MAX\_HISTORY\_LENGTH:  
 chat\_histories[session\_id] = chat\_histories[session\_id][-MAX\_HISTORY\_LENGTH:]  
  
 # Save chat histories  
 save\_chat\_histories()  
  
 # Return the response with chat history for this session  
 return jsonify({  
 "answer": response.content,  
 "status": "success",  
 "session\_id": session\_id,  
 "authenticated": is\_authenticated,  
 "username": session\_username if is\_authenticated else None,  
 "history": chat\_histories[session\_id]  
 }), 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

In the final part, we update the chat history with the new question and answer, including the username if authenticated or “anonymous” if not. We then limit the history to the maximum length, save the updated histories, and return the response. The response now includes authentication information, letting the client know if the session is authenticated and which username it belongs to.

### Testing User Identification and Authentication

Now that we’ve implemented user identification and authentication, let’s test it using curl commands:

1. Start your Flask application:

python app.py

This command starts your Flask server, making all the endpoints we’ve defined available for testing. Make sure you’re in the directory containing your app.py file when you run this command.

1. Register a new user:

curl -X POST http://localhost:3000/register \  
 -H "Content-Type: application/json" \  
 -d '{"username":"testuser", "password":"password123"}'

This command sends a POST request to the /register endpoint with a JSON payload containing a username and password. If successful, you should receive a JSON response confirming that the user was registered. The server will hash the password before storing it in the users.json file.

1. Log in with the new user:

curl -X POST http://localhost:3000/login \  
 -H "Content-Type: application/json" \  
 -d '{"username":"testuser", "password":"password123"}'

This command authenticates the user by sending their credentials to the /login endpoint. If successful, you’ll receive a JSON response containing a session ID. This session ID is a UUID that uniquely identifies this login session and will be used for subsequent authenticated requests. Make note of this session ID for the next steps.

1. Use the session ID to ask questions:

# Replace YOUR\_SESSION\_ID with the session ID from the login response  
curl -X POST http://localhost:3000/ask \  
 -H "Content-Type: application/json" \  
 -d '{"question":"What is machine learning?", "session\_id":"YOUR\_SESSION\_ID"}'

This command sends a question to the chatbot, including the session ID to identify the user. The server will recognize this as an authenticated session and associate the conversation with the specific user. The response will include the answer to your question, along with information about the authentication status and the username associated with the session.

1. Log out:

# Replace YOUR\_SESSION\_ID with the session ID from the login response  
curl -X POST http://localhost:3000/logout \  
 -H "Content-Type: application/json" \  
 -d '{"session\_id":"YOUR\_SESSION\_ID"}'

This command logs the user out by invalidating their session ID. The server removes the session ID from the user’s list of active sessions and clears the Flask session. After logging out, any requests using this session ID will be treated as unauthenticated.

## Section 3: Advanced Prompt Engineering Techniques

In this section, we’ll enhance our chatbot with advanced prompt engineering techniques to improve the quality and relevance of responses. We’ll implement system prompts for different personas, add few-shot examples, and create a structured output format.

### Step 1: Add constants for different personas

First, let’s add constants for different chatbot personas. Add these after the existing constants:

# Add these constants for different personas  
PERSONAS = {  
 "default": "You are a helpful assistant providing concise and accurate answers. Maintain context from the conversation history.",  
 "expert": "You are an expert AI assistant with deep knowledge across many fields. Provide detailed, technical, and accurate information. Use academic language and cite sources when possible.",  
 "friendly": "You are a friendly and approachable assistant. Use casual language, be conversational, and add occasional humor. Keep explanations simple and relatable.",  
 "concise": "You are a concise assistant that values brevity. Provide short, direct answers with minimal elaboration. Use bullet points when appropriate."  
}

This code defines different personas that our chatbot can adopt. Each persona has a unique system prompt that guides the language model’s response style. The “default” persona provides balanced responses, the “expert” persona gives detailed technical information, the “friendly” persona is more casual and conversational, and the “concise” persona provides brief, to-the-point answers. These personas allow users to customize the chatbot’s tone and style based on their preferences or the specific context of their questions.

# Add these few-shot examples for different types of questions  
FEW\_SHOT\_EXAMPLES = {  
 "factual": [  
 {"question": "What is the capital of France?", "answer": "The capital of France is Paris."},  
 {"question": "Who wrote 'Romeo and Juliet'?", "answer": "William Shakespeare wrote 'Romeo and Juliet'."}  
 ],  
 "opinion": [  
 {"question": "What's the best programming language for beginners?", "answer": "For beginners, Python is often recommended due to its readable syntax and gentle learning curve. However, the 'best' language depends on your goals. JavaScript is great for web development, Swift for iOS apps, etc."},  
 {"question": "Is artificial intelligence dangerous?", "answer": "AI presents both opportunities and risks. While it can solve complex problems, concerns exist about job displacement, privacy, and autonomous weapons. Responsible development with ethical guidelines is important."}  
 ],  
 "instruction": [  
 {"question": "How do I make pancakes?", "answer": "Basic pancake recipe: Mix 1 cup flour, 2 tbsp sugar, 2 tsp baking powder, and a pinch of salt. Add 1 cup milk, 2 tbsp melted butter, and 1 egg. Stir until just combined (lumps are okay). Cook on a hot greased pan until bubbles form, then flip and cook until golden."},  
 {"question": "How can I improve my public speaking?", "answer": "To improve public speaking: 1) Practice regularly, 2) Record yourself and review, 3) Join a group like Toastmasters, 4) Know your material thoroughly, 5) Start with small audiences, and 6) Focus on breathing and posture."}  
 ]  
}

Here we define few-shot examples for different types of questions. Few-shot learning is a technique where we provide the language model with examples of the type of responses we want. We’ve categorized questions into three types: factual (seeking objective information), opinion (seeking subjective views), and instruction (seeking how-to guidance). For each category, we provide sample question-answer pairs that demonstrate the appropriate response format and depth. This helps the model understand how to structure its responses for similar questions.

### Step 2: Add a function to classify question types

Next, let’s add a function to classify question types for better prompt engineering:

def classify\_question\_type(question):  
 """  
 Classify the type of question to select appropriate few-shot examples.  
 Returns one of: 'factual', 'opinion', 'instruction', or None.  
 """  
 # Simple rule-based classification  
 question = question.lower()  
  
 # Check for factual questions (who, what, when, where)  
 if any(q in question for q in ["what is", "who is", "when did", "where is", "how many"]):  
 return "factual"  
  
 # Check for opinion questions  
 if any(q in question for q in ["what do you think", "opinion", "best", "worst", "should i", "better"]):  
 return "opinion"  
  
 # Check for instruction questions (how to)  
 if any(q in question for q in ["how do i", "how to", "steps", "guide", "tutorial", "instructions"]):  
 return "instruction"  
  
 # Default to None if no clear classification  
 return None

This function implements a simple rule-based approach to classify user questions into one of three categories: factual, opinion, or instruction. It works by converting the question to lowercase and then checking for specific phrases or keywords that typically indicate each question type. For example, “what is” or “who is” often indicates a factual question, while “what do you think” or “best” suggests an opinion question. If the question doesn’t match any of the defined patterns, the function returns None, and we’ll use the default prompt without specific few-shot examples. While this approach is straightforward, it could be enhanced with more sophisticated natural language processing techniques in a production environment.

### Step 3: Update the prompt template to use personas and few-shot examples

Now, let’s update our prompt template to use different personas and few-shot examples. Replace the existing prompt\_template with a function to create dynamic prompts:

def create\_prompt\_template(persona="default", question\_type=None):  
 """  
 Create a dynamic prompt template based on the persona and question type.  
 """  
 system\_message = PERSONAS.get(persona, PERSONAS["default"])  
  
 messages = [  
 ("system", system\_message),  
 ]

The first part of this function retrieves the system message for the requested persona (or falls back to the default persona if the requested one doesn’t exist). It then initializes a messages list with this system message. The system message sets the overall tone and behavior for the language model.

# Add few-shot examples if we have a question type  
 if question\_type and question\_type in FEW\_SHOT\_EXAMPLES:  
 examples = FEW\_SHOT\_EXAMPLES[question\_type]  
  
 # Add examples to the system message  
 examples\_text = "\n\nHere are some examples of how to answer this type of question:\n"  
 for example in examples:  
 examples\_text += f"\nQuestion: {example['question']}\nAnswer: {example['answer']}\n"  
  
 messages = [  
 ("system", system\_message + examples\_text),  
 ]

In this section, we check if we have a valid question type and if we have examples for that type. If so, we format those examples as additional text to append to the system message. This provides the model with concrete examples of how to answer similar questions, improving response quality and consistency.

# Add the actual question placeholder  
 messages.append(("human", "{question}"))  
  
 return ChatPromptTemplate.from\_messages(messages)

Finally, we add a placeholder for the user’s question and create a ChatPromptTemplate from our messages list. This template will be used to format the actual prompt sent to the language model, with the {question} placeholder replaced by the user’s actual question (including conversation history if available).

### Step 4: Update the ask\_question endpoint to use dynamic prompts

Now, let’s update the ask\_question endpoint to use our dynamic prompts. We’ll need to modify the function to accept a persona parameter and use the question classification:

@app.route('/ask', methods=['POST'])  
def ask\_question():  
 """  
 REST API endpoint to ask a question to Azure OpenAI.  
 Expects a JSON payload with 'question' field, optional 'session\_id', and optional 'persona'.  
 Returns the model's response as JSON along with the chat history for that session.  
 """  
 global chat\_histories, users  
  
 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, session\_id, and persona from the request  
 question = data['question']  
 session\_id = data.get('session\_id', session.get('session\_id', 'default\_session'))  
 persona = data.get('persona', 'default')

In this first part of the updated endpoint, we extract not only the question and session ID but also a new parameter: the persona. This allows clients to specify which persona they want the chatbot to use when answering their question. If no persona is specified, we default to the “default” persona.

# Check if this is an authenticated session  
 is\_authenticated = False  
 session\_username = None  
  
 # Check if the session is in the flask session  
 if 'username' in session and session.get('session\_id') == session\_id:  
 is\_authenticated = True  
 session\_username = session.get('username')  
 else:  
 # Check if the session is associated with any user  
 for username, user\_data in users.items():  
 if session\_id in user\_data['sessions']:  
 is\_authenticated = True  
 session\_username = username  
 break  
  
 print(f"Question received from {'authenticated ' + session\_username if is\_authenticated else 'unauthenticated'} session {session\_id}: {question}")  
  
 # Initialize session history if it doesn't exist  
 if session\_id not in chat\_histories:  
 chat\_histories[session\_id] = []  
  
 # Get the chat history for this session  
 session\_history = chat\_histories[session\_id]  
  
 # Create context from chat history  
 context = ""  
 if session\_history:  
 context = "Previous conversation:\n"  
 for entry in session\_history:  
 context += f"Human: {entry['question']}\nAI: {entry['answer']}\n"  
  
 # Prepare the question with context if there's history  
 contextualized\_question = question  
 if context:  
 contextualized\_question = f"{context}\nHuman: {question}"

This middle section is similar to our previous implementation. We check if the session is authenticated, initialize the chat history if needed, and prepare the contextualized question with any previous conversation history.

# Classify the question type for few-shot examples  
 question\_type = classify\_question\_type(question)  
  
 # Create a dynamic prompt template based on persona and question type  
 prompt\_template = create\_prompt\_template(persona, question\_type)  
  
 # Create a new chain with the dynamic prompt template  
 dynamic\_chain = prompt\_template | llm  
  
 # Invoke the chain with the user's question  
 response = dynamic\_chain.invoke({  
 "question": contextualized\_question  
 })  
 print(f"Response for session {session\_id}: {response.content}")

Here’s where the new prompt engineering comes in. We first classify the question type using our classify\_question\_type function. Then we create a dynamic prompt template based on the requested persona and the classified question type. We create a new chain with this template and invoke it with the contextualized question. This approach allows us to dynamically adjust the prompt based on both the user’s preferences (persona) and the nature of their question.

# Update chat history for this session  
 chat\_histories[session\_id].append({  
 "question": question,  
 "answer": response.content,  
 "timestamp": str(datetime.now()),  
 "user": session\_username if is\_authenticated else "anonymous",  
 "persona": persona  
 })  
  
 # Limit chat history to MAX\_HISTORY\_LENGTH entries  
 if len(chat\_histories[session\_id]) > MAX\_HISTORY\_LENGTH:  
 chat\_histories[session\_id] = chat\_histories[session\_id][-MAX\_HISTORY\_LENGTH:]  
  
 # Save chat histories  
 save\_chat\_histories()  
  
 # Return the response with chat history for this session  
 return jsonify({  
 "answer": response.content,  
 "status": "success",  
 "session\_id": session\_id,  
 "authenticated": is\_authenticated,  
 "username": session\_username if is\_authenticated else None,  
 "persona": persona,  
 "question\_type": question\_type,  
 "history": chat\_histories[session\_id]  
 }), 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

In the final part, we update the chat history with the new question and answer, including the persona used. We then limit the history to the maximum length, save the updated histories, and return the response. The response now includes additional information: the persona used and the classified question type. This provides more transparency to the client about how the response was generated.

### Step 5: Add an endpoint to list available personas

Finally, let’s add an endpoint to list all available personas:

@app.route('/personas', methods=['GET'])  
def get\_personas():  
 """  
 REST API endpoint to retrieve all available personas.  
 Returns a list of all personas as JSON.  
 """  
 return jsonify({  
 "personas": list(PERSONAS.keys()),  
 "descriptions": PERSONAS  
 }), 200

This endpoint provides a way for clients to discover what personas are available in the system. It returns both a list of persona names (keys) and their full descriptions. This allows client applications to present users with options for selecting a persona, along with explanations of what each persona does. This kind of discoverability is important for API usability, as it allows clients to adapt to changes in the available personas without hardcoding values.

### Testing Advanced Prompt Engineering

Now that we’ve implemented advanced prompt engineering techniques, let’s test them using curl commands:

1. Start your Flask application:

python app.py

If your application is already running from the previous tests, you can skip this step. Otherwise, start the Flask server to make all endpoints available.

1. List available personas:

curl -X GET http://localhost:3000/personas

This command retrieves a list of all available personas from the /personas endpoint. The response will include both the names of the personas (default, expert, friendly, concise) and their descriptions. This is useful for client applications that want to offer users a choice of personas.

1. Ask a factual question with the expert persona:

curl -X POST http://localhost:3000/ask \  
 -H "Content-Type: application/json" \  
 -d '{"question":"What is quantum computing?", "persona":"expert"}'

This command sends a factual question to the chatbot, specifying the “expert” persona. The response should be detailed and technical, using academic language and possibly citing sources. The question will be automatically classified as “factual” by our classification function, and the appropriate few-shot examples will be included in the prompt.

1. Ask an opinion question with the friendly persona:

curl -X POST http://localhost:3000/ask \  
 -H "Content-Type: application/json" \  
 -d '{"question":"What do you think about artificial intelligence?", "persona":"friendly"}'

This command sends an opinion question to the chatbot, specifying the “friendly” persona. The response should be casual, conversational, and possibly include some humor. The question will be classified as “opinion” based on the phrase “what do you think,” and the opinion-related few-shot examples will be included in the prompt.

1. Ask an instruction question with the concise persona:

curl -X POST http://localhost:3000/ask \  
 -H "Content-Type: application/json" \  
 -d '{"question":"How do I learn programming?", "persona":"concise"}'

This command sends an instruction question to the chatbot, specifying the “concise” persona. The response should be brief and to the point, possibly using bullet points. The question will be classified as “instruction” based on the phrase “how do I,” and the instruction-related few-shot examples will be included in the prompt.

You can experiment with different combinations of questions and personas to see how the responses vary. You can also try questions that don’t fit neatly into one of our defined categories to see how the system handles them.

## Conclusion

In this tutorial, we’ve enhanced our ChatBot API with three powerful features:

1. **Conversation Persistence**: We implemented a file-based persistence mechanism that saves chat histories between server restarts, ensuring continuity in user interactions.

* This feature allows our chatbot to remember past conversations, even if the server is restarted. By storing chat histories in a JSON file, we’ve created a simple but effective persistence layer that maintains the context of conversations over time. This greatly improves the user experience, as users don’t have to repeat information they’ve already provided.

1. **User Identification and Authentication**: We added a simple username/password authentication system that associates sessions with specific users, enabling personalized experiences and secure access.

* With this authentication system, we can now identify individual users and associate their conversations with their accounts. This opens up possibilities for personalization and user-specific features. The system securely stores password hashes rather than plaintext passwords, and it manages user sessions with unique session IDs, allowing for secure multi-device access.

1. **Advanced Prompt Engineering**: We implemented dynamic prompts with different personas and few-shot examples, improving the quality and relevance of responses for different types of questions.

* By using different personas and few-shot examples, we’ve given our chatbot the ability to adapt its tone and response style based on the context and user preferences. The question classification system helps the chatbot understand what type of question is being asked and provide appropriate responses. This makes the chatbot more versatile and capable of handling a wider range of user needs.

These enhancements have transformed our simple chatbot into a more robust, personalized, and intelligent conversational agent. The modular design we’ve used makes it easy to extend and modify these features as needed.

## Next Steps

Here are some ideas for further enhancements:

1. **Database Integration**: Implement a database backend (e.g., SQLite, PostgreSQL) for more robust storage of chat histories and user data.

* While our file-based approach works well for demonstration purposes, a database would provide better scalability, concurrent access, and data integrity. This would be especially important in a production environment with many users.

1. **Role-Based Access Control**: Add role-based access control (RBAC) for different types of users.

* Extending our authentication system to include user roles (e.g., admin, regular user) would allow for more granular control over who can access certain features or information. This is crucial for applications where different users should have different levels of access.

1. **Advanced Prompt Engineering**: Implement more sophisticated prompt engineering techniques, such as chain-of-thought reasoning or structured outputs.

* There are many advanced prompt engineering techniques that could further improve our chatbot’s responses. Chain-of-thought prompting can help with complex reasoning tasks, while structured output prompting can ensure responses follow a specific format.

1. **Multimedia Support**: Add support for multimedia inputs and outputs (images, audio, etc.).

* Modern chatbots often need to handle more than just text. Adding support for images, audio, or other media types would make our chatbot more versatile and useful in a wider range of scenarios.

1. **Web Interface**: Implement a web interface for easier interaction with the chatbot.

* While our REST API is great for programmatic access, a web interface would make the chatbot more accessible to non-technical users. This could be a simple chat interface or a more complex dashboard with additional features.

By implementing these enhancements, you can continue to build on the foundation we’ve created and develop an even more powerful and versatile chatbot application.