AI-Powered Resume QA System

Naveen Prabakar

August 3, 2025

Overview

This document summarizes the design and implementation of a Q&A backend system built with FastAPI to serve as a career assistant. The assistant provides context-specific answers about **Naveen Prabakar** based on his resume, prior user interactions, and a continuously updating knowledge base. The application leverages state-of-the-art technologies including OpenAI's GPT-4, vector similarity search (FAISS), Redis session management, AWS S3 storage, and MongoDB archival.

1 Architecture and Components

1.1 Core Libraries and Dependencies

The system uses the following key external libraries and services:

- **FastAPI**: Web framework to handle HTTP requests.
- **Pydantic**: Input validation using data models.
- **LangChain**: For LLM prompt management, chain loading, embeddings, and document splitting.
- **OpenAI API**: GPT-4 model for natural language understanding and generation.
- **FAISS**: Efficient vector similarity search over embedded documents.
- **Redis**: Session storage for chat history with TTL-based expiry.
- **MongoDB**: Long-term archival of old Q&A interactions.
- **AWS S3**: Storage of Q&A logs in a text file backed by uploads/downloads.
- **Boto3**: AWS SDK for Python operations with S3.

1.2 Resume and Document Processing

- The resume PDF is loaded using `PyPDFLoader`, split into chunks with overlap using `RecursiveCharacterTextSplitter` to maintain semantic coherence.
- Text chunks are embedded using `OpenAIEmbeddings` and stored in a FAISS vector store with metadata indicating the `"resume"` source.

1.3 Prompt Engineering

- The system prompt defines the assistant's identity as a *career assistant* focused solely on answering questions about Naveen Prabakar based on the resume and prior conversation.

- User chat history is injected into the prompt template to preserve conversational context.

1.4 Session and Chat State Management

- Each user session is tracked via a UUID stored in a cookie.
- Redis stores a list of the last 5 messages (user and assistant) to maintain conversation context in memory with a TTL of 10 minutes.

1.5 Q&A Query Flow

Upon a user query:

1. Retrieve recent chat history from Redis and append the new user query.
2. Construct a conversational context string formatted by roles.
3. Use similarity search on FAISS separately for resume documents (max marginal relevance) and log documents (simple similarity search).
4. Combine retrieved documents as context for the QA chain.
5. Run the chain with GPT-4 to generate an answer.
6. Append the assistant response to Redis for session continuity.
7. Log the Q&A pair asynchronously to an S3 text file and periodically offload oldest entries to MongoDB for archival.
8. Re-embed updated logs into the vector store to enrich knowledge over time.

1.6 Background Logging and Vector Store Update

- The Q&A logs are maintained in an S3 text file, with local caching and merge logic.
- When the log exceeds 10 entries, oldest ones are moved to MongoDB, ensuring storage efficiency.
- Updated logs are re-embedded and added to the FAISS store to improve retrieval over time.

1.7 Session Clearing Endpoint

- The API provides an endpoint to clear the current session's chat memory from Redis.

2 Code Listing Excerpt

Below is an excerpt illustrating the `/ask` endpoint logic:

```
@app.post("/ask")
def ask_question(request: QueryRequest, background_tasks: BackgroundTasks,
    http_request: Request):

    # Retrieve session chat history
    session_id = http_request.cookies.get("session_id")
    redis_key = f"session:{session_id}"
    history_json = redis_client.lrange(redis_key, 0, -1)
    history = [json.loads(msg) for msg in history_json]

    # Update history with new query
    history.append({"role": "user", "content": request.query})
```

```
    trimmed_history = history[-5:]
    ...

    # Search and combine documents from resume and logs
    resume_docs = vectorstore.max_marginal_relevance_search(request.query, k=8,
        lambda_mult=0.5, filter={"source": "resume"})
    log_docs = vectorstore.similarity_search(request.query, k=2, filter={"
        source": "log"})
    combined_docs = resume_docs + log_docs

    # Generate answer via QA chain
    response = qa_chain.run({
        "input_documents": combined_docs,
        "question": injected_question
    })

    # Persist response and return
    ...
    return {"answer": response}
```

Conclusion

This backend service represents a well-designed, scalable Q&A system that effectively combines AI, vector search, and persistent storage to provide contextual responses. It demonstrates modern best practices in prompt engineering, session management, and data archival suitable for career assistant applications or similar custom knowledge bots.