Frequent Answers and Questions app

Andrei Steau

July 2025

1 Introduction

Develop a solution that can provide answers to users' questions by matching their queries with semantically similar questions in a predefined FAQ database. This system should further be enhanced to interact with the OpenAI API when it can't find a close enough match within the local FAQ database.

2 Project Structure

The project follows a modular FastAPI architecture in accordance with industry best practices for the development of scalable web applications.The codebase is organized into four distinct layers:

- **core** module contains essential infrastructure components, application settings, rate limiting, and Celery configuration for background task processing.
- **models** layer houses both SQLAlchemy database models and Pydantic schemas for data validation and serialization.
- **services** layer encapsulates business logic with dedicated modules for embeddings generation, OpenAI API integration, and similarity search operations;
- **api** layer provides clean REST endpoints that leverage dependency injection.

The other files not associated with a directory are helper tools for the environment (**.env**) file, for deployment (**Dockerfile and docker-compose**), for initialization and populating database tables (**init_db and init.sql**) and **requirements.txt and .gitignore**.

2.1 Models

In this layer, database tables are declared, in the **db.py** file, respectively the DTOs used in the endpoints part, in **schemas.py**. Here **Vector** from **pgvector** is used for the embeddings part because it has a fixed size (that of the embeddings created by openai) and is useful for databases with a small volume of data.

```python
engine = create_engine(settings.database_url)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

class FAQEntry(Base):
    __tablename__ = "faq_entries"

    id = Column(Integer, primary_key=True, index=True)
    question = Column(Text, nullable=False)
    answer = Column(Text, nullable=False)
    embedding = Column(Vector(1536), nullable=True)
    collection = Column(String(50), default="default")
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())

class QueryLog(Base):
    __tablename__ = "query_logs"

    id = Column(Integer, primary_key=True, index=True)
    user_question = Column(Text, nullable=False)
    matched_question = Column(Text, nullable=True)
    answer = Column(Text, nullable=False)
    source = Column(String(20), nullable=False) # 'local' or 'openai'
    similarity_score = Column(Float, nullable=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())

def create_tables():
    Base.metadata.create_all(bind=engine)

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

2.2 Services

The **embeddings_service** provides the business logic for text transformation using OpenAI. It offers functions for both individual and batch processing of texts. To reduce the cost and use fewer requests, I implemented a separate limiter in the **core** directory that aims to reduce and manage how embedding is done.

```python
async def compute_embedding(self, text: str) -> List[float]:
    try:
        estimated_tokens = self.estimate_tokens(text)
        await openai_rate_limiter.acquire(estimated_tokens)
        embedding = await self.embeddings.aembed_query(text)
        return embedding
    except Exception as e:
        logger.error(f"Error computing embedding: {e}")
        raise
```

Celery also transforms this embedding computation operation from a synchronous process that would block the FastAPI application for minutes, into an asynchronous and scalable system that allows users to receive instant responses while processing is performed in the background on distributed workers, thus providing automatic retry on errors, advanced monitoring, OpenAI cost optimization through intelligent batch processing.

```python
@celery_app.task
def compute_embeddings_for_collection(collection: str = "default"):

    db = SessionLocal()
    try:
        entries = db.query(FAQEntry).filter(
            FAQEntry.collection == collection,
            FAQEntry.embedding.is_(None)
        ).all()

        if not entries:
            logger.info(f"No entries to process for collection: {collection}")
            return

        embedding_service = EmbeddingService()
        questions = [entry.question for entry in entries]

        embeddings = asyncio.run(embedding_service.compute_embeddings_batch(questions))
        for entry, embedding in zip(entries, embeddings):
            entry.embedding = embedding

        db.commit()
        logger.info(f"Computed embeddings for {len(entries)} entries in collection: {collection}")

    except Exception as e:
        logger.error(f"Error computing embeddings for collection {collection}: {e}")
        db.rollback()
        raise
    finally:
        db.close()
```

The **similarity_service** provides the capability to calculate the similarity between 2 embeddings using cosine similarity provided by pgvector and to return the closest answer associated with the embedding if it exceeds a threshold declared in the **.env** file.

The **openai_service** service uses the LLM to answer questions related to the range of questions found in the database if the level of similarity between the asked question and the database is below the threshold. If the question is not related to customer service at all, a basic answer is returned.

```python
async def get_answer(self, user_question: str, context: Optional[str] = None) -> str:

    if not user_question or user_question.strip() == "":
        return "Please provide a valid question."
    try:
        system_message = SystemMessage(content="""
        You are a helpful IT support assistant. Answer user questions about account management,
        password resets, profile settings, and general IT support topics. Answer only if you know the answer, don't guess.
        Be concise and provide actionable steps when possible.
        If you don't know the answer or the question is not related to IT support, politely redirect the user with
        "This is not really what I was trained for, therefore I cannot answer. Try again! ".
        """)

        human_message = HumanMessage(content=user_question)

        if context:
            human_message.content = f"Context: {context}\n\nQuestion: {user_question}"

        response = await self.llm.ainvoke([system_message, human_message])
        return response.content

    except Exception as e:
        logger.error(f"Error getting OpenAI response: {e}")
        return "I'm sorry, I'm having trouble processing your request right now. Please try again later."
```

2.3 API

This part of the project deals with handling REST endpoints via APIRouter, the main ones being **ask_question** and**compute_embeddings**. The first one uses the similarity services and openai to test whether the response will be generated by openai or returned from the database. The second composes embeddings for a batch of texts.

```python
@router.post("/embeddings")
async def compute_embeddings(
    collection: str = "default",
):
    try:
        task = compute_embeddings_for_collection.delay(collection)
        return {"message": f"Embedding computation started for collection: {collection}", "task_id": task.id}
    except Exception as e:
        logger.error(f"Error starting embedding computation: {e}")
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Error starting embedding computation"
        )
```

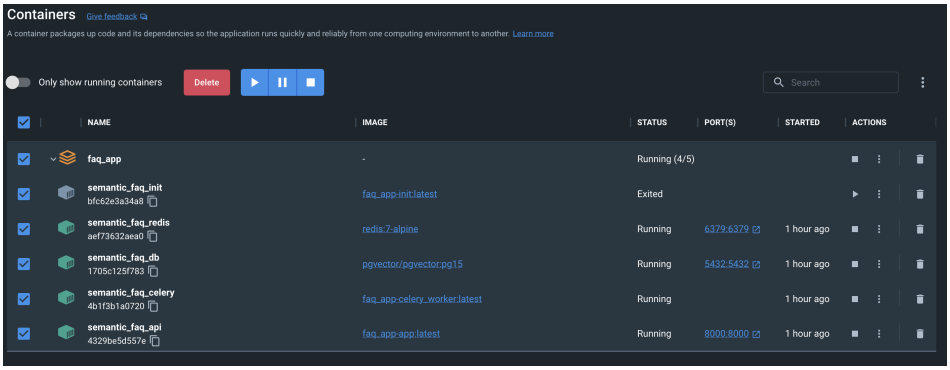
3 Deployment

The deployment process begins with the foundational infrastructure services and builds up to the complete application stack through a carefully orchestrated sequence of service initialization.

The database is configured with environment variables for privacy, under **5432** port, and includes persistent storage through a named volume and incorporates an initialization script that runs automatically when the container first starts. A comprehensive health check mechanism ensures the database is fully operational before dependent services attempt to connect, using **pg_isready** commands with configurable retry logic.

Redis serves as the message broker and caching layer, deployed using the lightweight Alpine-based Redis 7 image. This service handles task queuing for the **Celery** workers and provides high-performance caching capabilities for the application.

The main application service is built from a custom **Dockerfile** that creates a secure, production-ready Python environment. The build process starts with a Python 3.11 Alpine base image, chosen for its minimal footprint and security benefits. The container setup includes installing essential system dependencies like GCC, PostgreSQL development libraries, and other compilation tools needed for Python packages that require native extensions. The application follows security best practices by creating a dedicated non-root user called **appuser** and ensuring all application files have appropriate ownership and permissions.



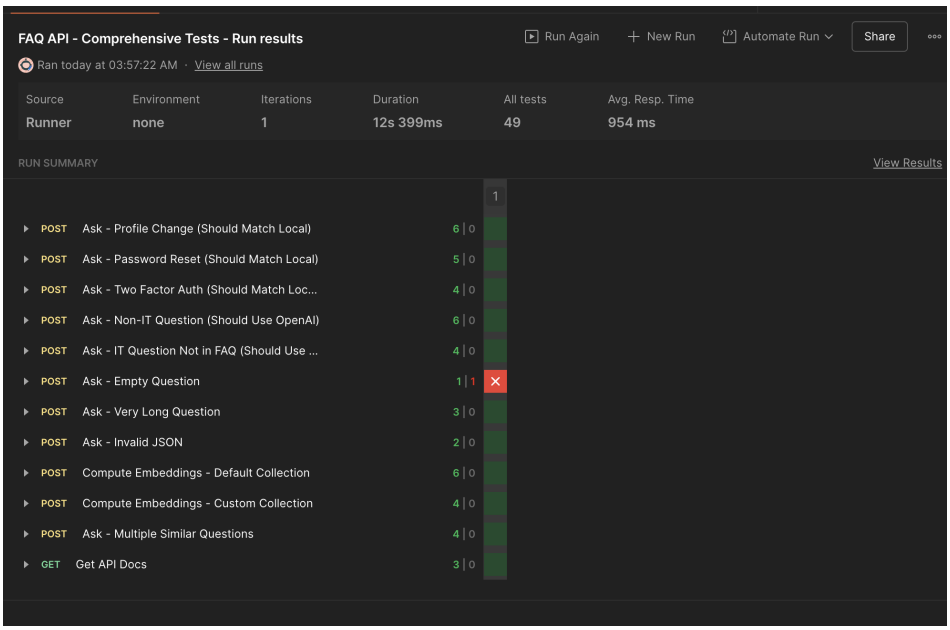
4 Testing

Postman using various inputs to see the quality of the responses and error handling.I tested the application in Postman in order to validate validates the API's routing system between local database matching and OpenAI fallback responses. The suite includes **Local DB Match Tests** that verify questions about profile changes, password resets, and two-factor authentication correctly match FAQ entries with high similarity scores. **OpenAI Fallback Tests** confirm non-IT questions (weather inquiries) trigger the configured redirect response "This is not really what I was trained for, therefore I cannot answer. Try again!" while IT questions outside the FAQ scope receive helpful troubleshooting guidance.

Critical Edge Cases Tested:

- **Empty/Invalid Requests**: Validates graceful error handling and appropriate user guidance
- **Long Complex Questions**: Multi-part queries with combined issues (profile + password + email problems) are successfully processed with structured responses
- **Out-of-Scope Questions**: Non-IT topics correctly trigger domain boundary enforcement
- **Performance**: Response times ranging 1.4-6.1 seconds remain within acceptable limits

All tests maintain consistent response structure validation, ensuring `source`, `matched_question`, `similarity_score`, and `answer` fields behave correctly across both local and OpenAI processing paths. The test results confirm robust API behavior with proper scope management and reliable fallback mechanisms.



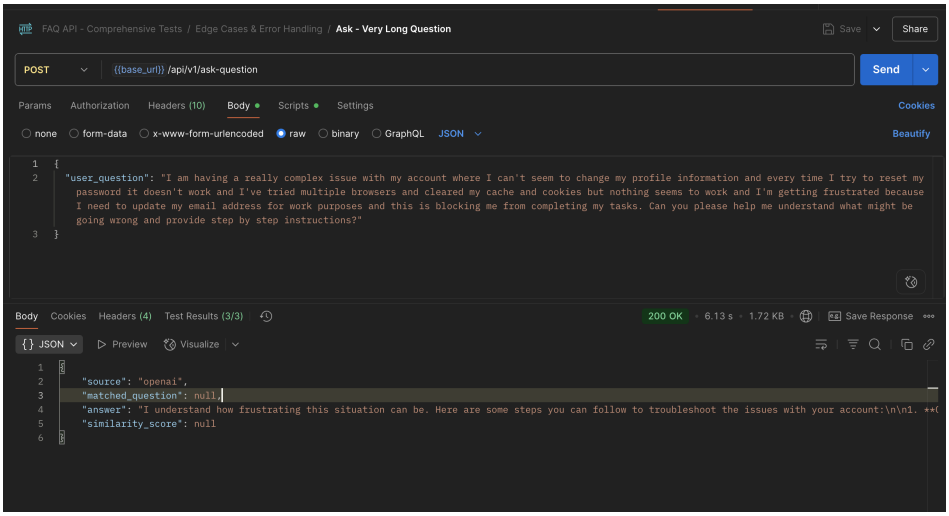


Figure 1: Long question

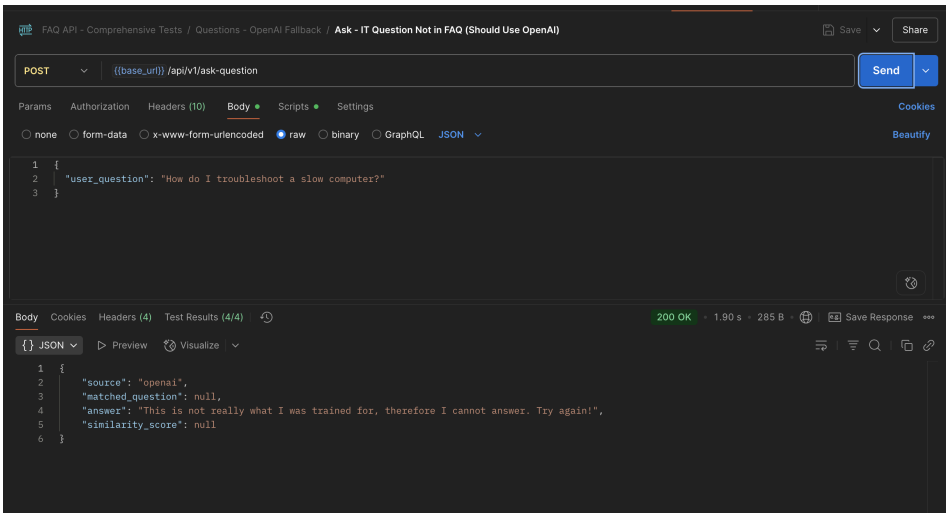


Figure 2: IT but not customer related

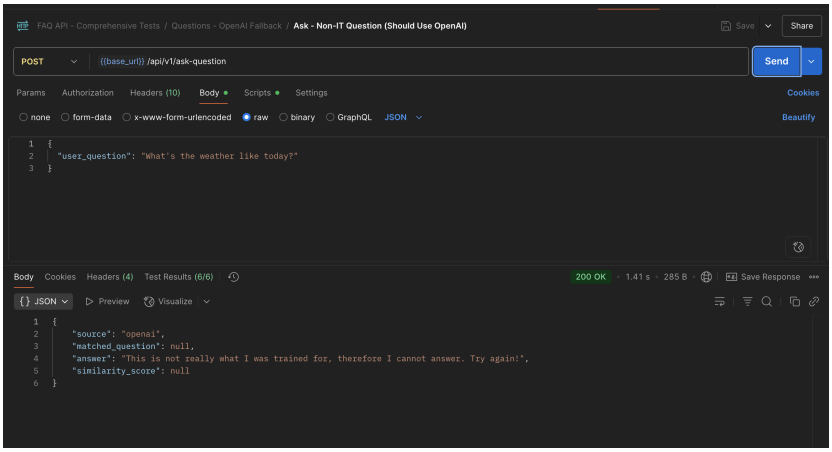


Figure 3: Unrelated question