Talk2Books :: A Multi-lingual Conversational Application

This document outlines a detailed specification for building a conversational application that allows users to interact with a corpus of multi-lingual text content, leveraging Retrieval-Augmented Generation (RAG). It breaks down the problem into smaller, manageable modules, detailing the design for each.

I. Overall Goal:

To create a system where a user can ask questions in any supported language (English, Devanagari, Gurmukhi initially) and receive relevant, coherent answers generated from a provided corpus of text, also in those supported languages.

II. Supported Languages:

- **Initial:** English, Devanagari, Gurmukhi
- **Future Expansion:** Scalable to support additional languages with minimal code changes.

III. System Architecture:

The system will follow a RAG architecture comprising the following components:

1. **Data Ingestion & Processing:** Loads and preprocesses text data.
2. **Document Splitting:** Divides documents into manageable chunks.
3. **Embedding Generation:** Creates vector embeddings of text chunks.
4. **Vector Database:** Stores and indexes embeddings for efficient similarity search.
5. **Query Processing:** Embeds user queries and retrieves relevant document chunks.
6. **LLM Integration:** Generates answers using a Large Language Model based on query and context.
7. **API & Frontend:** Provides an interface for user interaction.

IV. Major Modules (Detailed Design Specifications):

Module 1: Data Ingestion & Preprocessing

- **Problem:** Load text data from various sources (files, databases, APIs) and prepare it for further processing.
- **Inputs:** File paths, database connection strings, API endpoints.
- **Outputs:** Cleaned, normalized text content.
- **Design:**
    - Implement data loaders for common formats (txt, pdf, docx, csv).
    - Normalize text:
        - Encoding: Ensure UTF-8 encoding.
        - Whitespace normalization.
        - Lowercasing (optional, can be controlled by a config parameter).
        - Handle language-specific punctuation.
    - Language Detection: Automatically detect the language of each document. Utilize a library like `langdetect`. Store language information with the document metadata.
    - Error Handling: Robust error handling for file access, database connections, and API requests.
- **Testing:** Unit tests for each loader and normalization step, including tests with various edge cases and invalid data.

Module 2: Document Splitting

- **Problem:** Divide large documents into smaller chunks suitable for embedding and retrieval.
- **Inputs:** Cleaned text content.
- **Outputs:** List of text chunks.
- **Design:**
  - Implement recursive character text splitter.
  - Configure chunk size and overlap. Allow these to be configurable parameters.
  - Consider semantic chunking. Attempt to split documents along sentence or paragraph boundaries to preserve context.
  - Handle language-specific sentence/paragraph boundary detection.
- **Testing:** Tests to verify that documents are split correctly, chunk sizes are within the expected range, and overlap is applied as configured. Tests with documents containing complex formatting and multiple languages.

Module 3: Embedding Generation

- **Problem:** Create vector embeddings of text chunks.
- **Inputs:** List of text chunks.
- **Outputs:** List of vector embeddings.
- **Design:**
  - Utilize a pre-trained multi-lingual sentence transformer model (e.g., `all-mpnet-base-v2` from sentence-transformers).
  - Cache embeddings to reduce computational cost.
  - Handle language-specific tokenization requirements of the embedding model.
- **Testing:** Verify that embeddings are generated correctly for text in all supported languages. Compare embeddings for similar and dissimilar text to ensure semantic accuracy.

Module 4: Vector Database Integration

- **Problem:** Store and index vector embeddings for efficient similarity search.
- **Inputs:** List of vector embeddings, associated document metadata.
- **Outputs:** Indexed embeddings in a vector database.
- **Design:**
  - Choose a vector database (Pinecone, ChromaDB, Weaviate). Consider scalability, cost, and ease of integration.
  - Implement indexing and querying functionality.
  - Store document metadata with each embedding for context retrieval.
- **Testing:** Verify that embeddings are stored and retrieved correctly. Measure query performance and scalability with a large number of embeddings.

Module 5: Query Processing & Retrieval

- **Problem:** Process user queries, retrieve relevant document chunks, and construct context for the LLM.
- **Inputs:** User query (text).
- **Outputs:** List of relevant document chunks.
- **Design:**
  - Embed the user query using the same embedding model as the document chunks.

    - Perform similarity search in the vector database to retrieve top-k most relevant chunks.
    - Implement filtering based on language. Allow the system to prioritize results in the same language as the query, if available.
- **Testing:** Verify that relevant chunks are retrieved for various queries in all supported languages. Measure retrieval accuracy and performance.

Module 6: LLM Integration & Response Generation

- **Problem:** Generate coherent and relevant answers using an LLM based on the user query and retrieved context.
- **Inputs:** User query, retrieved document chunks.
- **Outputs:** Generated answer (text).
- **Design:**
    - Choose an LLM (we will use OSS models such as Phi4, Gemma3, etc. via Ollama).
    - Construct a prompt that includes the user query and the retrieved context.
    - Configure LLM parameters (temperature, max tokens) for optimal response quality.
    - Implement error handling for LLM API requests.
- **Testing:** Evaluate the quality and relevance of generated answers for various queries and contexts. Use both automated metrics (e.g., ROUGE) and human evaluation.

Module 7: API & Frontend Development

- **Problem:** Provide an API and frontend interface for user interaction.
- **Inputs:** User queries.
- **Outputs:** Generated answers.
- **Design:**
    - Develop a REST API using Flask or Django.
    - Create a simple frontend interface using HTML, CSS, and JavaScript.
    - Implement input validation and error handling.
    - Support multi-lingual input and output.
- **Testing:** Unit tests for API endpoints, integration tests for frontend functionality, and user acceptance testing.

V. Technologies:

- **Programming Language:** Python
- **LLM:** We will use a local/on-premise instance of Ollama to start with.
- **Embedding Model:** `all-mpnet-base-v2` (sentence-transformers)
- **Vector Database:** ChromaDB, or Qdrant
- **API Framework:** Quart
- **Frontend:** HTML, CSS, JavaScript

VI. Future Considerations:

- **Language Translation:** Integrate a translation API to support queries in languages not natively supported.
- **Contextual Awareness:** Improve the system's ability to understand and maintain context across multiple turns of conversation.
- **User Feedback:** Collect user feedback to improve the accuracy and relevance of generated answers.

- **Scalability:** Design the system to handle a large number of users and a growing corpus of text data.

Setting up the development environment

You will need a PC/laptop with Ubuntu or macOS. In case you are using Windows, you can use WSL. Follow this guide to setup VS Code for working with WSL on a Windows machine.

Here's a step-by-step guide to start building the application using the specified stack. The instructions are simplified, organized, and designed for incremental progress.

---

1. Install Required Tools

1. **Ubuntu (or WSL)**
   Ensure you're using Ubuntu (or a compatible Linux distro).
   Install tools like `git`, `docker`, and `python3` (3.9 or higher) on your Ubuntu OS:

   ```
   sudo apt update && sudo apt install git docker.io python3-pip -y
   ```

   We need docker for deploying Qdrant vector DB.

2. **Python 3 and Virtual Environment**
   In the project folder, create a virtual environment for the backend:

   ```
   bss@host1:~/$ mkdir talk2books
   bss@host1:~/$ cd talk2books
   bss@host1:~/talk2books$ python3 -m venv .venv
   bss@host1:~/talk2books$ source .venv/bin/activate  # On Windows:
   .venv\Scripts\activate
   (.venv) bss@host1:~/talk2books$
   ```

3. **Install Python Dependencies**
   Create a `requirements.txt` file for the backend:

   ```
   quart
   langchain
   langchain-ollama
   langgraph
   qdrant-client
   sentence-transformers
   openai  # Optional if using OpenAI for LLM
   ```

   Install dependencies:

   ```
   (.venv) bss@host1:~/talk2books$ pip install -r requirements.txt
   ```

IMPORTANT: When in doubt, refer to the respective framework's documentation (e.g., this tutorial for langchain) for how to install and configure it. The code snippets that I show in the following sections are oversimplified to give you an idea how the code may look like. You should create these programs by referring to the latest documentation.

4. **React for Frontend**
   Install Node.js and npm:

   ```
   sudo apt install nodejs npm -y
   ```

   Create a React app:

   ```
   bss@host1:~/talk2books$ npx create-react-app frontend
   bss@host1:~/talk2books$ cd frontend
   bss@host1:~/talk2books/frontend$
   bss@host1:~/talk2books/frontend$ npm start  # Test React app
   ```

5. **Installing Ollama for local LLM serving** You need a locally running instance of Ollama to work with LLMs.

   1. Follow the instructions for your OS as described here: https://github.com/ollama/ollama
   2. Download a smaller model for running locally. Phi4-mini may be a reasonable choice.

---

2. Project Structure

Organize your project like this:

```
talk2books/
├── backend/              # Quart (Python) backend
│   ├── app.py            # Quart API
│   ├── rag_chain.py      # LangChain/LangGraph logic
│   └── requirements.txt
├── frontend/             # React frontend
│   ├── src/
│   │   └── App.js        # Main React component
│   └── package.json
├── qdrant/               # Qdrant config (optional)
└── .gitignore            # GitHub version control
```

---

3. Start with the Backend (Quart + LangChain)

a. Create a Simple Quart API

Refer to the Quart Tutorial to learn.

1. **backend/app.py**
   Write a basic Quart app:

```
from quart import Quart, request, jsonify

app = Quart(__name__)

@app.route('/api/query', methods=['POST'])
async def handle_query():
    data = await request.json
    query = data.get('query')
    # TODO: Add LangChain logic here
    return jsonify({"response": f"Received query: {query}"})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

2. **Run the Backend**

```
cd backend
python app.py
```

   Test it with `curl` or Postman:

```
curl -X POST http://localhost:5000/api/query -H "Content-Type:
application/json" -d '{"query": "Hello"}'
```

b. Add LangChain for RAG

Following are the high-level basic scripts. You should refer to the RAG Tutorials of LangChain to learn more.

1. **Install LangChain**
   Ensure `langchain` and `langgraph` are in `requirements.txt`.

2. **backend/rag_chain.py**
   Create a basic RAG chain:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Qdrant
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain.chains import RetrievalQA
from langchain_community.llms import OpenAI  # Or use a local model
```

```
# Example: Load documents and split
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500,
chunk_overlap=50)
# Add your document loading logic here

# Initialize Qdrant vector store
qdrant_client = Qdrant(
    client=QdrantClient("http://localhost:6333"),  # Qdrant URL
    collection_name="my_collection",
    embeddings=HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
)

# Create a retrieval QA chain
qa_chain = RetrievalQA.from_chain_type(
    llm=OpenAI(model_name="gpt-3.5-turbo"),  # Replace with your LLM
    chain_type="stuff",
    retriever=qdrant_client.as_retriever()
)

def answer_query(query: str) -> str:
    return qa_chain.invoke({"query": query})["result"]
```

3. **Integrate RAG into Quart API**
   Update `backend/app.py` to use `answer_query`:

```
from .rag_chain import answer_query

@app.route('/api/query', methods=['POST'])
async def handle_query():
    data = await request.json
    query = data.get('query')
    response = answer_query(query)
    return jsonify({"response": response})
```

---

4. Set Up Qdrant (Vector Database)

1. **Install Qdrant via Docker**
   Run Qdrant in a Docker container:

```
docker run -p 6333:6333 -p 6334:6334 qdrant/qdrant
```

   Verify Qdrant is running:

```
curl http://localhost:6333
```

2. **Create a Collection in Qdrant**
   Use the Qdrant Python client:

```
from qdrant_client import QdrantClient
from qdrant_client.models import VectorParams

client = QdrantClient("http://localhost:6333")
client.create_collection(
    collection_name="my_collection",
    vectors_config=VectorParams(size=384, distance="Cosine")  # Size
depends on your embedding model
)
```

---

5. Build the Frontend (React)

Again, this is a very simple code to get to started. Refer to the React Router Tutorial to learn.

1. **Create a Simple UI**
   Update `frontend/src/App.js`:

```
import React, { useState } from 'react';

function App() {
  const [query, setQuery] = useState('');
  const [response, setResponse] = useState('');

  const handleSubmit = async (e) => {
    e.preventDefault();
    const res = await fetch('http://localhost:5000/api/query', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ query })
    });
    const data = await res.json();
    setResponse(data.response);
  };

  return (
    <div>
      <h1>My RAG App</h1>
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          value={query}
          onChange={(e) => setQuery(e.target.value)}
          placeholder="Ask a question"
        />
        <button type="submit">Submit</button>
      </form>
```

```
        <div><strong>Response:</strong> {response}</div>
      </div>
    );
}

export default App;
```

2. **Run the React App**

```
cd frontend
npm start
```

---

6. Connect Everything

1. **Test the Full Flow**
   - Start Qdrant: `docker run -p 6333:6333 qdrant/qdrant`
   - Run the backend: `python backend/app.py`
   - Run the frontend: `npm start` in the `frontend/` directory.

2. **Add Documents to Qdrant**
   Update `backend/rag_chain.py` to load and index documents:

```
from langchain_community.document_loaders import TextLoader
loader = TextLoader("path/to/your/document.txt")
documents = loader.load_and_split(text_splitter)
qdrant_client.add_documents(documents)
```

---

7. Version Control with GitHub

1. **Initialize Git**

```
cd talk2books
git init
git add .
git commit -m "Initial setup"
```

2. **Link to GitHub**
   Create a repo on GitHub and push:

```
git remote add origin https://github.com/your-username/your-repo.git
git push -u origin main
```

---

8. Troubleshooting Tips

- **CORS Errors**: Add CORS middleware to Quart:

```
pip install quart-cors
```

```
from quart_cors import cors
app = cors(app)
```

- **Qdrant Errors**: Ensure Docker is running and the Qdrant container is active.
- **LangChain Errors**: Check the embedding model size matches Qdrant's vector size.

Next Steps

- Add authentication (e.g., API keys).
- Use a production-ready database (e.g., PostgreSQL) for metadata.
- Deploy to a cloud platform (e.g., Render, Vercel, or AWS).