Semantic Contextual PDF Retrieval (SCPR)

Developer Documentation

Abstract

This document provides the complete technical documentation for the Semantic Contextual PDF Retrieval (SCPR) application. It details the project's architecture, core components, backend API, frontend integration, and testing strategy. This manual is intended for developers and system administrators responsible for maintaining, extending, or deploying the application.

Contents

1 Introduction

1.1 Project Vision

The Semantic Contextual PDF Retrieval (SCPR) project is a full-stack web application designed to transform static PDF documents into interactive, conversational resources. It addresses the challenge of information retrieval in large documents by allowing users to ask natural language questions and receive answers generated directly from the document's content.

1.2 Core Technology: RAG

The system is built on a **Retrieval-Augmented Generation (RAG)** pipeline. This hybrid AI model combines the strengths of two different AI systems:

1. **Retrieval:** A dense vector retrieval system (using Google's embedding model and ChromaDB) excels at finding semantically relevant passages of text from a large knowledge base (the PDF).
2. **Generation:** A powerful Large Language Model (LLM) (Google's Gemini) excels at understanding context and generating human-like text.

By first *retrieving* relevant context and then providing it to the LLM to *generate* an answer, the RAG pipeline ensures that the application's responses are accurate, factually grounded in the source document, and not prone to AI "hallucinations."

2 Software Architecture

The application operates on a decoupled client-server model.

- **Frontend (Client):** A React single-page application (SPA) that provides the user interface for file uploading and chat. It is a "dumb" client that holds no application logic.
- **Backend (Server):** A monolithic Python Flask server (`app.py`) that exposes a REST API. This server contains all the core logic, including PDF parsing, AI model integration, vector database management, and the RAG pipeline.
- **AI Services (External):** The system depends on the Google AI API for two distinct services:
  - `models/text-embedding-005` for text vectorization.
  - `gemini-2.5-flash` for answer generation.
- **Database (Local):** ChromaDB is used as a local, persistent vector store, saved to the `chroma_db` directory.

3 Backend API (app.py)

The Flask server is the core of the application. It exposes two primary endpoints.

3.1 Endpoint: POST /upload

This endpoint handles the ingestion and indexing of a new PDF document.

- **Description:** Accepts a `multipart/form-data` upload containing a single PDF file. It saves the file, triggers the complete `initialize_vectorstore` pipeline, and creates the persistent `chroma_db` on disk.
- **Request (form-data):**
  - `file`: The PDF file to be indexed.
- **Success Response (200 OK):**

```
{
  "status": "success",
  "file_name": "example_document.pdf",
  "total_chunks": 120,
  "indexing_duration": "4.56s",
  "timestamp": "2025-11-10T20:30:00.123Z"
}
```

- **Error Response (400 Bad Request):**

```
{
  "status": "error",
  "message": "No file part"
  // Or "No selected file"
  // Or "Invalid file type. Must be PDF."
}
```

- **Error Response (500 Server Error):**

```
{
  "status": "error",
  "message": "No readable text found in PDF (even after OCR)."
  // Or "Failed to index PDF: [Error details]"
}
```

3.2 Endpoint: POST /query

This endpoint handles all user questions and generates AI-powered answers.

- **Description:** Accepts a JSON object containing a user's question. It loads the persistent vector store, runs the full RAG pipeline, and returns a detailed JSON response.
- **Request (JSON):**

```
{
  "question": "What is the capital of France?"
}
```

- **Success Response (200 OK):**

```
{
  "status": "success",
  "question": "What is the capital of France?",
```

```
  "elapsed_time": {"overall": "1.23s"},
  "steps": [
    {"name": "Retrieval (Vector Search)", "status": "Completed", ...},
    {"name": "Context Augmentation (Prompt Engineering)", ...},
    {"name": "Generation (Gemini API Call)", ...}
  ],
  "final_answer": "According to the document, the capital of France is Paris.",
  "source_context": "The Eiffel Tower is in Paris... Paris is the capital..."
}
```

- **Error Response (400 Bad Request):**

```
{
  "status": "error",
  "message": "Vector store is empty. Please upload and index a PDF first."
  // Or "No question provided"
}
```

4 Core Backend Logic

The application's logic is contained within several key Python functions in `app.py`.

4.1 `extract_text_from_pdf(filepath)`

This function is responsible for robustly extracting all text from a PDF.

- It first attempts to parse the PDF as a digital-native document using `pdfplumber`, which is fast and accurate for text-based files.
- If `pdfplumber` returns no text (a sign of a scanned or image-only PDF), it triggers an OCR (Optical Character Recognition) fallback.
- The fallback uses `fitz` (PyMuPDF) to render each page as an image, `Pillow (PIL)` to process it, and `pytesseract` to extract the text from the image.
- This hybrid approach ensures both speed for simple PDFs and accuracy for complex, scanned documents.

4.2 `get_embedding_function()`

This is a singleton manager for the AI embedding model.

- It uses a global variable `embedding_function` to ensure the Google AI model is only initialized once per server session.
- On its first call, it reads the `GEMINI_API_KEY` from the environment and instantiates `GoogleGenerativeAIEmbeddings`.
- All subsequent calls immediately return the already-initialized instance, saving time and resources.

4.3 `initialize_vectorstore(file_path, file_name)`

This function orchestrates the entire indexing pipeline.

1. **Extract:** Calls `extract_text_from_pdf()` to get the raw text.

2. **Chunk:** Uses `RecursiveCharacterTextSplitter` to break the text into 1000-character chunks with a 200-character overlap (to maintain context between chunks).
3. **Embed & Store:** Calls `get_embedding_function()` and passes the text chunks and embedding function to `Chroma.from_texts`. This single LangChain command automatically handles embedding all chunks and saving them to the persistent directory specified by `DB_PATH`.

4.4 `query_document()`

This function contains the core RAG logic.

1. **Load Store:** It first checks if the `vectorstore` is in memory. If not, it loads it from disk using `Chroma(persist_directory=...)`.
2. **Search:** It performs a `vectorstore.similarity_search()` to find the top 5 text chunks most relevant to the user's question.
3. **Augment:** It constructs a detailed prompt, injecting the retrieved chunks (the "context") and a system instruction that forces the AI to answer *only* based on this context.
4. **Generate:** It passes this augmented prompt to the `ChatGoogleGenerativeAI` (Gemini) model to get the final answer.

5 Frontend (React)

The frontend is a standard React application bootstrapped with 'create-react-app'.

- **Communication:** It uses `axios` to make HTTP requests to the Flask backend. The backend URL is managed by an environment variable, `REACT_APP_API_BASE_URL`, in the `.env.local` file, which defaults to `http://127.0.0.1:5000`.
- **Components (Example Structure):**
  - `App.js`: Manages the main application state, including the list of chat messages.
  - `UploadComponent.js`: A component that renders the file upload form and calls the `POST /upload` endpoint.
  - `ChatInterface.js`: A component that renders the chat history and the input box, calling the `POST /query` endpoint on submit.

6 Testing Strategy

The project maintains a robust testing suite for the backend using `pytest`.

- **Test Suite:** `test_app.py`
- **Test Dependencies:** `pytest`, `pytest-mock` (from `requirements-dev.txt`)
- **Key Fixture (`client()`)**: The `client` fixture in `test_app.py` is **session-scoped** (`@pytest.fixture(scope="session")`). This is a critical design choice to prevent `PermissionError` on Windows. It creates a single test client and a single set of test database/upload folders for the entire test run, rather than per-test.

- **Mocking:** The `test_full_workflow_happy_path` test uses `mocker` (from `pytest-mock`) to patch all external dependencies, including:
  - `app.extract_text_from_pdf`
  - `app.GoogleGenerativeAIEmbeddings`
  - `app.ChatGoogleGenerativeAI`

  This makes the test extremely fast and ensures it doesn't make real, billable API calls.
- **Test Coverage:** The test suite covers:
  - API Error handling (400 errors for bad input).
  - The "Happy Path" (successful upload followed by a successful query).
  - Logical edge cases (querying before a file is uploaded).