High-Level Design (HLD)

1. Architecture Overview

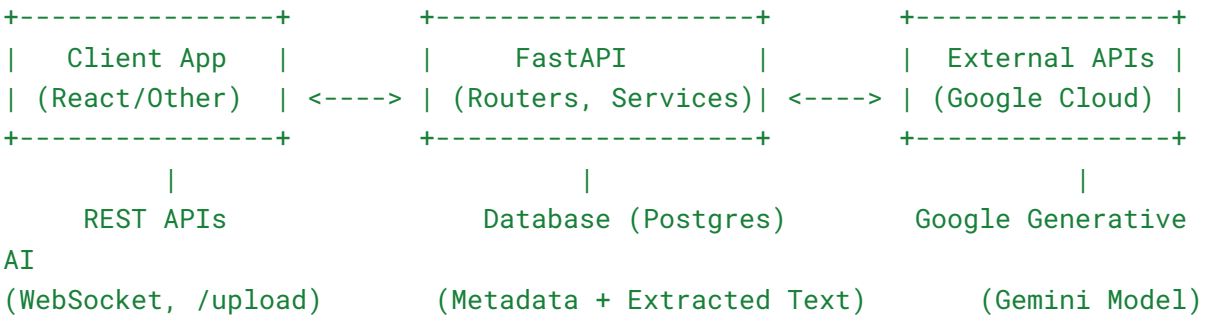
This system is a **backend service** built with FastAPI, designed to handle:

- **PDF Processing**: Upload PDFs, extract content, and store metadata and extracted text.
- **Real-Time Q&A**: Use Google Generative AI to provide real-time answers to user questions based on uploaded documents.
- **Rate Limiting**: Prevent abuse of WebSocket connections by restricting request frequency.
- **Google Cloud Integration**: Store PDFs in Google Cloud Storage and use Generative AI (Gemini model) for NLP.

Key Components:

1. **User**: The client (frontend or API consumer) interacts with the system via API endpoints or WebSocket connections.
2. **FastAPI Backend**:
    - **Routers**: Handle API endpoints for file uploads and WebSocket communication.
    - **Services**: Contain business logic (e.g., PDF processing, NLP integration).
    - **Database**: PostgreSQL for storing document metadata and content.
3. **External Services**:
    - Google Cloud Storage: For storing uploaded PDF files.
    - Google Generative AI: For NLP and real-time Q&A.

Architecture Diagram:



---

2. Functional Modules

1. **PDF Upload Module**:
    - API Endpoint: `/upload`
    - **Functions**:

        - Save PDF locally.
        - Extract text using PyMuPDF.
        - Upload the PDF to Google Cloud Storage.
        - Store metadata (e.g., filename, upload date) and extracted text in PostgreSQL.
2. **Real-Time Q&A Module**:
    - WebSocket Endpoint: `/ws`
    - **Functions**:
        - Establish WebSocket connection.
        - Fetch document content from PostgreSQL.
        - Pass questions and document content to Google Generative AI for answers.
        - Maintain session history for contextual Q&A.
        - Implement rate limiting to prevent abuse.
3. **Rate Limiting**:
    - Track user requests using client IP or session ID.
    - Enforce a request interval (e.g., one request every 5 seconds).
4. **Storage Integration**:
    - Use Google Cloud Storage for persistent PDF file storage.
    - Store public URLs of uploaded PDFs in the database.

---

3. Data Flow

1. **PDF Upload**:
    - Client uploads a PDF via `/upload`.
    - Backend processes the PDF (extracts text and uploads to GCS).
    - Metadata and extracted text are stored in PostgreSQL.
2. **Real-Time Q&A**:
    - Client connects to WebSocket (`/ws`) with a document ID.
    - Backend fetches the document's text content from PostgreSQL.
    - Questions are sent to Google Generative AI along with the document content.
    - AI generates and returns responses in real-time.

Low-Level Design (LLD)

1. Key Components

1.1 Database Schema

- **Table: `documents`**

```
CREATE TABLE documents (
    id SERIAL PRIMARY KEY,
```

```
    filename VARCHAR(255),
    upload_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    gcs_url TEXT,
    content TEXT
);
```

Fields:

- `id`: Unique identifier for each document.
- `filename`: Name of the uploaded PDF.
- `upload_date`: Timestamp of upload.
- `gcs_url`: URL of the PDF stored in Google Cloud Storage.
- `content`: Extracted text content from the PDF.

1.2. NLP Service

File: `app/services/nlp.py`

- **Responsibilities**:
    - Generate answers to user questions using Google Generative AI.
    - Maintain contextual session history.

1.3. WebSocket Router

File: `app/routers/ws.py`

- **Responsibilities**:
    - Establish and manage WebSocket connections.
    - Enforce rate limiting.
    - Handle real-time Q&A with session history.

---

2. Workflow for Major Functionalities

2.1. PDF Upload

1. **User Action**:
    - Uploads a PDF file via `/upload`.
2. **Backend**:
    - Save the file locally.
    - Extract text from the PDF.
    - Upload the file to Google Cloud Storage.

    - Store metadata and extracted text in PostgreSQL.
3. **Response**:
    - Returns the document ID and success message.

2.2. Real-Time Q&A

1. **User Action**:
    - Connects to WebSocket (`/ws`) and provides a document ID.
2. **Backend**:
    - Fetch document content from PostgreSQL.
    - Process questions using Google Generative AI.
    - Maintain session history for contextual responses.
3. **Response**:
    - Sends real-time answers to the client.

---

3. Implementation Details

3.1. Rate Limiting

Implemented in `app/routers/ws.py`:

```
rate_limits = {}

def is_rate_limited(user_id: str, interval: int = 5) -> bool:
    current_time = time.time()
    if user_id in rate_limits:
        last_request_time = rate_limits[user_id]
        if current_time - last_request_time < interval:
            return True
    rate_limits[user_id] = current_time
    return False
```

3.2. Contextual Q&A

Implemented in `app/services/nlp.py`:

```
def get_answer(question: str, document_content: str, history_text:
str) -> str:
    template = """
    You are an intelligent assistant. Based on the content provided
and conversation history, answer the following question with a clear
```

```
format giving appropriate html tags respectively starting with
heading or paragraph tag.
    History:
    {history}
    Content: {content}
    Question: {question}
    Answer:
    """
    formatted_prompt = template.format(history=history_text,
content=document_content, question=question)
    llm = GoogleGenerativeAIWrapper()
    return llm(formatted_prompt)
```

---

4. Error Handling

4.1. PDF Upload

- **Invalid File Type**:
  - Return a `400 Bad Request` error for unsupported file types.
- **Database Errors**:
  - Catch and log database errors.

4.2. WebSocket Q&A

- **Invalid Document ID**:
  - Return an error message if the document is not found in the database.
- **Rate Limiting**:
  - Inform users when they exceed the request limit.

---

5. Monitoring and Logging

- Log all WebSocket events (connections, disconnections, errors).
- Log all API calls (e.g., file uploads, database operations).
