Data Flows

Document Upload Flow

This flow describes how a document is ingested into the system, stored, and automatically tagged using the CrewAI service. This action is restricted to users with the 'Admin' role.

1. **User Action (Frontend)**: An Admin user navigates to the upload section and selects a document file in the Gyst Next.js frontend.
2. **Upload Request (Frontend -> Next.js Backend)**: The frontend sends an HTTP POST request containing the document file and basic information (like original filename) to the Next.js Backend Document API (/api/documents/upload).
3. **File Handling & Metadata Extraction (Next.js Backend)**:
    - The Next.js backend receives the file stream.
    - It generates a hashed filename for storage.
    - It saves the file to the local ./uploads directory using the hashed filename.
    - It extracts basic metadata (file type, size) from the uploaded file.
    - It determines the local file path where the document was saved.
4. **Database Record Creation (Next.js Backend -> SQLite DB)**: The Next.js backend connects to the SQLite database (gyst.sqlite) and inserts a new record into the documents table. This record includes:
    - A unique document ID.
    - The user ID of the uploader.
    - The organization ID.
    - The hashed filename.
    - The original filename.
    - The relative file path in ./uploads.
    - File type and timestamp.
5. **AI Tagging Request (Next.js Backend -> Python FastAPI Service)**: The Next.js backend sends an HTTP POST request to the Python FastAPI service's /analyze_document endpoint. The request body includes the full local file path where the document is stored (./uploads/hashed_filename).
6. **Document Analysis (Python FastAPI Service)**:
    - The Python FastAPI service receives the request with the file path.
    - It initializes/activates a CrewAI task specifically designed for document analysis and tagging.
    - The CrewAI task utilizes the appropriate CrewAI RAG tool (e.g., PdfRagTool, DocxRagTool) to read the content directly from the provided file path in the local ./uploads directory.
    - CrewAI agents process the document content, identify key concepts, terms, or entities relevant for tagging.
7. **Tags Generation (Python FastAPI Service)**: CrewAI generates a list of suggested tags based on its analysis.
8. **Tags Response (Python FastAPI Service -> Next.js Backend)**: The Python FastAPI service returns the list of generated tags in the HTTP response body back to the Next.js backend.

9. **Database Tagging (Next.js Backend -> SQLite DB)**:
    - The Next.js backend receives the list of suggested tags.
    - For each suggested tag, it checks if the tag already exists in the tags table in the SQLite database. If not, it inserts the new tag.
    - It then inserts records into the document_tags junction table, linking the newly uploaded document ID to the relevant tag IDs (including the AI's confidence score if provided).
10. **Success Response (Next.js Backend -> Frontend)**: The Next.js backend sends a success response back to the frontend, potentially including the saved document's ID and the newly associated tags.
11. **UI Update (Frontend):** The frontend updates the file explorer and potentially displays the document and its automatically generated tags to the user.

AI Chat Flow

This flow describes how a user interacts with the AI via the chat interface to query information across documents, leveraging CrewAI and RAG. This functionality is available to all authenticated users within an organization.

1. **User Action (Frontend)**: An authenticated user types a question or command into the Chat Interface in the Gyst Next.js frontend and sends it.
2. **Chat Request (Frontend -> Next.js Backend)**: The frontend sends an HTTP POST request containing the user's query to the Next.js Backend Chat API (/api/chat).
3. **Context Gathering (Next.js Backend -> SQLite DB)**: The Next.js backend receives the query. To provide relevant context to the AI, it performs a lightweight search or retrieval from the SQLite database (gyst.sqlite). This could involve:
    - Identifying the currently viewed document's ID/path if the chat is context-aware.
    - Performing a quick keyword search on document metadata (filename, tags) to find potentially relevant documents within the user's organization.
    - Retrieving the file paths for these potentially relevant documents from the documents table.
4. **AI Chat Request (Next.js Backend -> Python FastAPI Service)**: The Next.js backend sends an HTTP POST request to the Python FastAPI service's /chat endpoint. The request body includes:
    - The user's query string.
    - A list of file paths for the relevant documents identified in the previous step.
    - (Optional) Any other relevant context (e.g., user role - though decided not to influence AI logic for core tasks, chat history snippets if stored).
5. **Contextual Processing (Python FastAPI Service)**:
    - The Python FastAPI service receives the request, query, and list of file paths.
    - It initializes/activates a CrewAI task designed for conversational AI.
    - The CrewAI task uses the appropriate RAG tools to read and process content from the provided list of document file paths in ./uploads. This allows the AI to understand the content of the documents relevant to the query.
    - CrewAI agents analyze the user's query in the context of the retrieved document content.
6. **Response Generation (Python FastAPI Service)**: CrewAI agents formulate a response based on their analysis, drawing information directly from the document

content provided via RAG. The response may include references to the documents used.

7. **AI Response (Python FastAPI Service -> Next.js Backend)**: The Python FastAPI service returns the AI-generated response (a text string) in the HTTP response body back to the Next.js backend. This response might contain markers or identifiers referring to the documents that were used.
8. **Response Formatting & Document Reference Mapping (Next.js Backend -> SQLite DB)**:
   - The Next.js backend receives the AI's text response.
   - It parses the response to identify any document references (based on paths or IDs provided by the Python service in a structured format, or via pattern matching).
   - For identified document references, it queries the SQLite database (gyst.sqlite) to retrieve corresponding metadata like the original filename or document ID to create clickable links for the frontend.
   - It formats the final response string for display in the chat interface.
9. **Display Response (Next.js Backend -> Frontend)**: The Next.js backend sends the formatted AI response (including mapped document links) to the frontend.
10. **UI Update (Frontend)**: The frontend displays the AI's response in the chat interface, rendering any document references as interactive links.