System Documentation: RAG Indexing Workflow

Workflow Name: RAG Indexing Workflow for Documentation
Source File: RAG_Indexing_Workflow_for_Documentation.json
Project Lead: Mike Holland
System Architect: Gemini Pro

1. Overall Goal

This workflow serves as the dedicated data pipeline for bootstrapping a new project with an existing knowledge base. Its purpose is to receive a URL to a JSON file and a new project name from the UI, create the necessary project records in the database, and then systematically process the contents of the JSON file. It chunks the text, generates vector embeddings for each chunk, and saves them to the long-term memory store (rag_store), permanently associating them with the newly created project.

2. Key Components & Architecture

- **Trigger:** A unique n8n **Webhook** (...303639ed...) listens for POST requests from the Chat 8 UI's "Create from URL" feature.
- **Database:** It interacts with both the project_contexts table (to create the new project) and the rag_store table (to save the indexed memories).
- **External Services:** It makes calls to Google Cloud Storage (GCS) to fetch the source JSON and to the OpenAI API to generate embeddings.
- **Logic:** The workflow is a sequential pipeline: create the project, fetch the data, chunk it, create embeddings, and save everything to the database.

3. Step-by-Step Data Flow

The workflow executes in a clear, linear sequence:

1. **Webhook2:** The workflow is triggered when the user submits a JSON URL and a new project name from the UI. The payload contains { "projectName": "...", "jsonUrl": "..." }.
2. **Create Project Entry1:** A Postgres node immediately executes a complex SQL query. This single query is responsible for:
    - Generating a new, unique project_id (numeric), chat_session_id, and rag_session_id.

- Inserting a new record into the project_contexts table with these new IDs and the provided project name.
- Returning the newly generated IDs for use in subsequent steps.

3. **HTTP Request GCS Fetch1:** An HTTPRequest node takes the jsonUrl from the initial webhook and fetches the raw text content of the JSON file from Google Cloud Storage.
4. **Chunk Code1:** A sophisticated Code node takes the raw text from the file. It intelligently splits the text into smaller, manageable chunks suitable for embedding, ensuring no chunk exceeds a maximum character limit. It outputs a list of items, each containing a text chunk.
5. **Filter1:** A simple Filter node ensures that no empty chunks proceed, preventing wasted API calls.
6. **Create Embedding1:** (Runs for each chunk) An HTTPRequest node takes each text chunk and sends it to the OpenAI embeddings API (text-embedding-ada-002) to generate a vector embedding.
7. **Merge Embedding with Chunk Data1:** A Merge node combines the original chunk data with the newly generated embedding vector from the previous step.
8. **Prepare for DB1:** A Set node takes the merged data and restructures it, creating clean fields (session_id_for_insert, original_content_for_db, etc.) ready for the next step.
9. **Format Vector for Postgres1:** A Code node takes the embedding vector (which is a JavaScript array) and converts it into the specific string format (e.g., "[0.1, 0.2, ...]") required by the pgvector database type.
10. **Insert Embedding to Xata1:** (Runs for each chunk) This is the final, critical database operation. A Postgres node takes the fully prepared data for each chunk and executes an INSERT query to save it to the rag_store table. It correctly links each memory to the project_id and rag_session_id that were generated in Step 2.
11. **Prepare Final Response1:** After all chunks have been successfully inserted, a Set node runs once to format a clean success message.
12. **Respond to Webhook3:** The final Respond to Webhook node sends the success object (containing the new project_id, session_id, and rag_session_id) back to the UI, signaling that the project has been created and indexed, and a new chat session can begin.