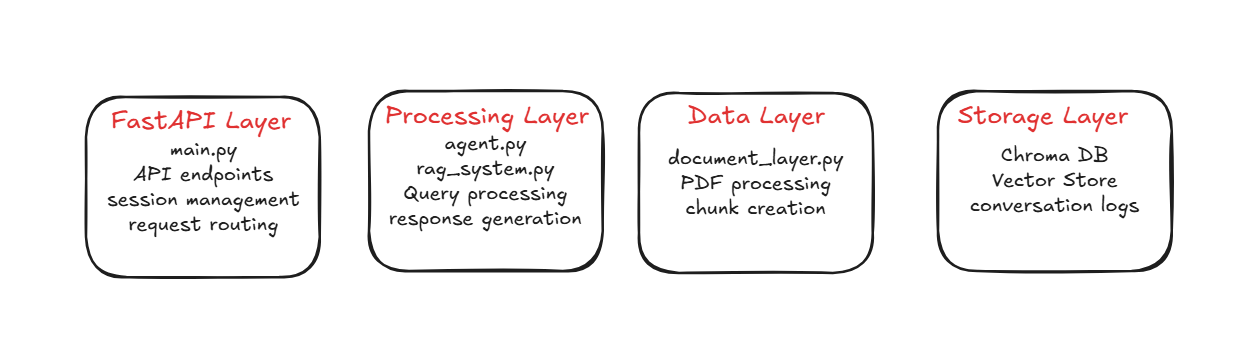
**Overview of the Customer Service (General Inquiry) Chatbot Architecture**

This customer support chatbot is based on the Retrival-Augumented-Generation **(RAG)** system integrated with Langchain and designed to provide customer support for NepaWholesale.

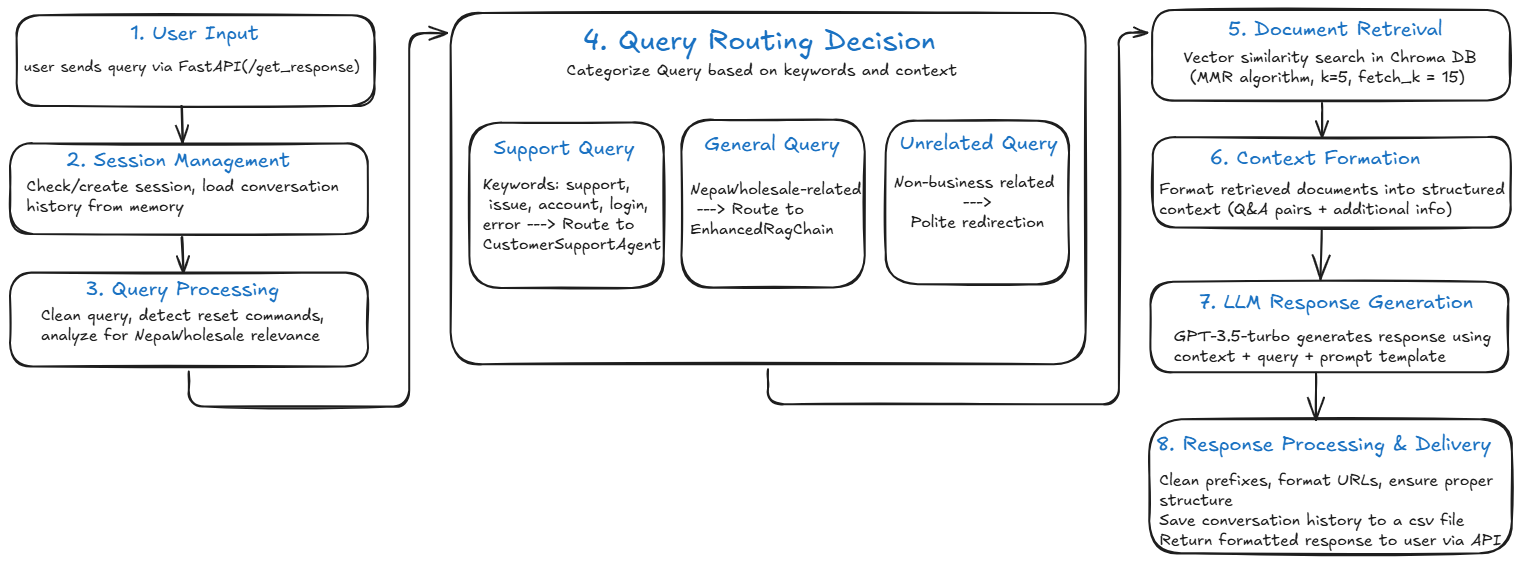
Here’s the high-level workflow:

1. **User Query:** The user sends a message through the FastAPI interface (handled by main.py)
2. **Document Loading:** Documents which are prepared before hand like Customer Support Knowledge Base documents are loaded and processed into chunks (document\_loader.py)
3. **Vector Store:** The chunks are embedded into vectors and stored in a Chroma database for retrieval (rag\_system.py)
4. **Query Processing:** The query is categorized and routed to either RAG or chain or the support agent.
5. **Response Generation:** The system retrieves relevant document chunks, combines them with the query, and uses an LLM (GPT-3.5 Turbo) to generate a response
6. **Output:** The response is returned to the user via the API (main.py).

**System Architecture Overview**

****

**Complete System Flow**

****

**Detailed Explanation of Each File**

**1. main.py**

Purpose: Acts as the main entry point, handling API requests, coordinating components, and managing sessions and database storage.

**Key Components:**

* **FastAPI Application**: Defines the API endpoints (/get\_response, /reset\_session, /health, /) using FastAPI, serving the chat interface (static/home.html) and processing user inputs.

**CustomerSupportChatbot Class**:

* **Initialization**:
  + Loads the document (docs/Customer Support KB Updated.pdf) using load\_single\_nepa\_document from document\_loader.py.
  + Initializes the LLM (ChatOpenAI with GPT-3.5-turbo) and embeddings (HuggingFaceEmbeddings).
  + Creates a vector store using create\_vector\_store from rag\_system.py.
  + Sets up the RAG chain (EnhancedRAGChain) and support agent (CustomerSupportAgent).
  + Initializes a SQLite database (chatbot\_conversations.db) to store conversations.
* **Session Management**:
  + Uses a dictionary (self.sessions) to track user sessions, storing conversation history with ConversationBufferMemory.
  + Generates unique session\_ids and handles session expiration (30 minutes inactivity).
* **Message Processing (process\_message)**:
  + Cleans the user query and checks if it’s a session reset request (e.g., "reset").
  + Detects if the query is NepaWholesale-related using keywords from NEPADocumentProcessor.
  + Routes queries:
    - **Support Queries**: Queries with keywords like "support", "issue", "account", or "login" go to CustomerSupportAgent.handle\_support\_query (agent.py).
    - **Unrelated Queries**: Non-NepaWholesale queries (e.g., "can you tell me something about Britain?") get a polite redirection.
    - **General Queries**: Other queries are processed by EnhancedRAGChain.invoke (rag\_system.py).
  + Saves the query and response to the SQLite database using save\_conversation.
* **Database Storage**:
  + The \_initialize\_database method creates a conversations table.
  + The save\_conversation method logs each interaction (session ID, query, response, timestamp, IP, user agent).

**How main.py contributes to Output:**

* This is the central system, which coordinates the entire workflow, ensuring queries are routed correctly and responses are stored.
* Manages conversation history, which helps maintain context (e.g., understanding follow-up questions)
* Provides a robust API interface for user interaction.

**Example in Output:**

* For "hello", main.py routes to the RAG chain, resulting in a generic greeting: "Hello! How can I assist you today with NepaWholesale's products and services?"
* For "I'm getting authentication errors...", it detects "error" and routes to the agent, which generates a tailored response.

**2. document\_loader.py**

**Purpose**: Loads and processes the knowledge base document (a PDF) into chunks suitable for the vector store.

**Key Components**:

* **NEPADocumentLoader Class**:
  + Supports loading PDF and text files (currently only PDF is used).
  + Uses PyPDFLoader from LangChain to extract text from PDFs.
  + Validates PDFs using validate\_pdf (checks for extractable text with PyPDF2).
  + Adds metadata (e.g., source, file name, file type) to each document chunk.
* **NEPADocumentProcessor Class**:
  + Processes raw documents into Q&A pairs or text chunks.
  + Splits text using RecursiveCharacterTextSplitter with a chunk size of 1000 and overlap of 200.
  + Identifies Q&A pairs using regex (Q: and A:) and assigns metadata (e.g., content\_type: qa\_pair).
  + Removes repetitive patterns (e.g., boilerplate text) using remove\_nepa\_repetitive\_patterns.
* **Functions**:
  + load\_single\_nepa\_document: Loads a single file and processes it into chunks.
  + load\_nepa\_documents: Loads multiple files from a directory (not used in your case).

**How It Works**:

1. The PDF (Customer Support KB Updated.pdf) is loaded using PyPDFLoader.
2. The text is split into chunks (e.g., 17 documents, as per our earlier test).
3. Q&A pairs are identified and tagged; other text is stored as general content.
4. Metadata is added to help the RAG system understand context (e.g., nepa\_category, priority).

**How It Contributes to Output**:

* Provides the knowledge base that the RAG system searches to answer queries.
* Ensures the document is broken into manageable chunks, improving retrieval accuracy.
* Metadata helps prioritize relevant information (e.g., Q&A pairs for direct answers).

**Example in Output**:

* For "Can you provide a comparison of nicotine levels...", the loader provides chunks containing vape juice information, which the RAG system retrieves.

**3. rag\_system.py**

**Purpose**: Implements the RAG pipeline, managing the vector store and generating responses using retrieved documents.

**Key Components**:

* **Vector Store Creation (create\_vector\_store)**:
  + Takes document chunks from document\_loader.py and embeddings from HuggingFaceEmbeddings.
  + Creates a Chroma vector store, storing document embeddings in ./chroma\_db.
  + Removes duplicates using remove\_duplicate\_documents based on content hash.
* **RAG Chain Setup (setup\_rag\_chain)**:
  + Configures a RetrievalQA chain with a retriever using Maximum Marginal Relevance (MMR) search (k=5, fetch\_k=15).
  + Uses a PromptTemplate to instruct the LLM to be professional, concise, and redirect unrelated queries.
* **EnhancedRAGChain Class**:
  + **Initialization**:
    - Sets up a custom retriever and prompt template similar to setup\_rag\_chain.
  + **Invoke Method**:
    - Retrieves relevant documents using MMR.
    - Formats context with format\_unique\_context, grouping Q&A pairs and other content.
    - Generates a response using the LLM with the formatted prompt.
    - Cleans the response to remove prefixes (Agent: Customer:) and ensure proper formatting.
  + **Context Formatting**:
    - Deduplicates retrieved documents.
    - Organizes content into "RELEVANT Q&A" and "ADDITIONAL INFORMATION" sections.

**How It Works**:

1. The vector store embeds document chunks into vectors.
2. For a query, the retriever fetches the top 5 most relevant chunks (after considering 15 candidates).
3. The chunks are formatted into a context string.
4. The LLM generates a response based on the context and query, guided by the prompt.

**How It Contributes to Output**:

* Retrieves relevant document chunks to ground the LLM’s response in factual data.
* Ensures responses are specific by including details from the knowledge base.
* Handles general queries by providing context-aware answers.

**Example in Output**:

* For "Can you provide a comparison of nicotine levels...", the RAG system retrieves chunks about vape juice flavors, but since specific comparison data is missing, it provides a general response with a support contact.

**4. agent.py**

**Purpose**: Handles specialized support queries (e.g., technical or account issues) with tailored responses, acting as a fallback when the RAG chain is insufficient.

**Key Components**:

* **CustomerSupportAgent Class**:
  + **Initialization**:
    - Takes an LLM and optional vector store.
    - Defines a PromptTemplate with guidelines for:
      * Clear, concise, friendly responses.
      * Step-by-step guidance for technical issues.
      * Specific handling of account issues (e.g., locked accounts).
      * Fallback to support contact if information is missing.
  + **Handle Support Query (handle\_support\_query)**:
    - Categorizes the query using categorize\_query (e.g., account, technical, order).
    - Retrieves up to 3 relevant documents from the vector store (if available).
    - Adds specific guidance for account queries (e.g., contact support for locked accounts).
    - Formats the prompt with context and query.
    - Generates a response using the LLM.
    - Cleans the response with clean\_response.
  + **Clean Response (clean\_response)**:
    - Removes Agent: and Customer: prefixes.
    - Ensures proper formatting (e.g., multiple newlines, ending punctuation).
    - Adds category-specific guidance if the response is too short (e.g., support contact for account issues).
  + **Categorize Query (categorize\_query)**:
    - Assigns categories based on keywords (e.g., "login" → account, "error" → technical).
    - **Error Handling**:
    - Provides tailored fallbacks for account issues (e.g., locked account advice).
    - Falls back to a generic support contact for other errors.

Finally, after each conversation from the user then the chatbot process and provides the output and also for each conversation it is saved in csv file for references and follow-up for what kind of questions are asked and how they are handled.

**How It Works**:

1. **Query Routing**:
   * In main.py, queries with support keywords (support, "issue", "account, etc.) are routed to handle\_support\_query`.
2. **Context Retrieval**:
   * The agent retrieves up to 3 documents using similarity search, providing context from the knowledge base.
3. **Category-Specific Logic**:
   * For account queries, it appends guidance about contacting support or resetting passwords.
   * For other categories, it relies on retrieved documents or general advice.
4. **Response Generation**:
   * The LLM generates a response using a prompt that emphasizes professionalism and problem-solving.
   * The response is cleaned to ensure it’s natural.
5. **Error Fallback**:
   * If document retrieval or the LLM fails, it provides a specific fallback for account issues or a generic support contact.

**Why It Produces High-Quality Output**:

* **Targeted Handling**:
  + The agent is designed for support queries, ensuring responses are practical and action-oriented (e.g., troubleshooting steps for authentication errors).
* **Context Awareness**:
  + By retrieving documents, it grounds responses in the knowledge base, increasing accuracy.
* **Tailored Guidance**:
  + Account-specific logic ensures relevant advice (e.g., contact support for locked accounts).
* **Robustness**:
  + Error handling ensures users always get a helpful response, even if data is missing.

**Example in Output**:

* **Query**: "I'm getting authentication errors when trying to access my tax-exempt certificates. Is this a known issue?"
  + **Process**:
    - main.py detects "error" and routes to agent.py.
    - categorize\_query labels it as technical.
    - The agent retrieves documents (likely none specific to this issue).
    - The prompt instructs the LLM to provide troubleshooting steps and a support contact.
    - The response suggests refreshing the page, clearing cache, and contacting support, which is practical and professional.
  + **Output**: "I'm sorry to hear about the authentication errors... try refreshing the page or clearing your browser cache and cookies... contact our technical support team at 561-684-1107..."

### How the Chatbot Produces the Output

Let’s trace the flow for the example queries:

1. **Query: "hello"**
   * **main.py**: Detects no support keywords, considers it NepaWholesale-related (greeting), and routes to EnhancedRAGChain.
   * **rag\_system.py**: Retrieves general documents (e.g., company overview), formats context, and generates a greeting: "Hello! How can I assist you today with NepaWholesale's products and services?"
   * **main.py**: Saves the query and response to the SQLite database.
   * **Output**: Friendly and appropriate greeting.
2. **Query: "Can you provide a comparison of nicotine levels across your top 3 best-selling vape juice flavors?"**
   * **main.py**: Detects it’s NepaWholesale-related (keywords like "vape juice") and routes to EnhancedRAGChain.
   * **rag\_system.py**: Retrieves documents about vape products, but lacks specific comparison data. The LLM generates a general response with a support contact.
   * **main.py**: Saves to the database.
   * **Output**: Informative but generic due to limited data, with a call to contact support.
3. **Query: "I'm getting authentication errors when trying to access my tax-exempt certificates. Is this a known issue?"**
   * **main.py**: Detects "error" and routes to CustomerSupportAgent.
   * **agent.py**: Labels as technical, retrieves documents (likely none specific), and generates troubleshooting steps with a support contact.
   * **main.py**: Saves to the database.
   * **Output**: Actionable advice (refresh page, clear cache) with a support contact, showing the agent’s strength in technical queries.

**Future Enhancements**

* Enhance Unrelated Query handling
* Improving Knowledge Base
* Expand Agent Capabilites
* Dynamic Response Length

**Outputs:**

