Use Case UC1: Retrieve Relevant Document Chunks

Scope: RAG Chatbot System (Backend Pipeline)

Level: Sub-function

Primary Actor: User (via System Trigger)

1. Stakeholders and Interests

- **User:** Wants to find text segments that answer their question accurately. Expects fast and relevant results.
- **Developer:** Wants to ensure correct query cleaning, effective STOP_WORDS filtering, and accurate ranking based on Term Frequency (TF).
- **Orchestrator (System Controller):** Expects the module to return data in a standard format (List<Hit>) without errors to proceed to the answering stage.

2. Preconditions

- The system is initialized, and the KeywordIndex is successfully loaded into memory.
- The user's question (question) has been received, and the user's intent (Intent) has been determined by the previous module (e.g., STAFF_LOOKUP).

3. Success Guarantee (Postconditions)

- The system identifies the top K (default 10) most relevant document chunks (Hit) based on the query, sorts them according to the scoring rules, and saves them into the Context object.

4. Main Success Scenario (Basic Flow)

1. The **Orchestrator** initiates the **QueryWriter** strategy with the user's question and the detected intent.
2. The **QueryWriter** normalizes the question by converting it to lowercase and removing punctuation marks.
3. The **QueryWriter** iterates through the tokens and filters out meaningless conjunctions and banned words defined in STOP_WORDS.
4. The **QueryWriter** checks the Intent and appends specific booster terms to the list (e.g., adding "office", "email" for STAFF_LOOKUP intent).
5. The **QueryWriter** returns the list of cleaned and expanded terms (List<String>) to the Orchestrator.
6. The **Orchestrator** calls the **Retriever** strategy using these terms and the loaded KeywordIndex.
7. The **Retriever** fetches matching records (IndexEntry) from the KeywordIndex for each query term.

8. The **Retriever** calculates a cumulative score for each document chunk based on the sum of Term Frequencies (TF).
9. The **Retriever** sorts the results based on the following deterministic tie-breaking rules:
    - Score (Descending order)
    - Document ID (Ascending order - if scores are equal)
    - Chunk ID (Ascending order - if scores and document IDs are equal)
10. The **Retriever** returns the top 10 highest-scoring results (List<Hit>) to the Orchestrator.

5. Extensions (Alternative Flows)

- **3a. The query consists entirely of "Stop Words" (e.g., "and or but"):**
    - **3a1.** The QueryWriter detects that the filtered list is empty.
    - **3a2.** The system uses the original words from the question (unfiltered) or falls back to default general terms.
- **7a. No matches found in the Index for the query terms:**
    - **7a1.** The Retriever creates an empty list of Hits.
    - **7a2.** The system logs a warning: "No hits found for terms: [terms]".
    - **7a3.** Returns the empty list (The AnswerAgent will handle the "No answer found" response in the next stage).
- **10a. Fewer than 10 results found:**
    - **10a1.** The Retriever returns all available results (e.g., 3 hits) without attempting to pad the list.

6. Special Requirements

- **Deterministic Behavior:** The same input (Question + Configuration) must always produce the exact same results in the exact same order. Tie-break rules must be strictly enforced.
- **Performance:** The retrieval process must complete in under 200ms, even with a fully loaded index.
- **Language Support:** Turkish and English characters (e.g., İ, ı, ş, ç) must be preserved during normalization.

7. Technology and Data Variations

- The index structure is read from a JSON file and stored in memory as a HashMap within the KeywordIndex class.
- Sorting algorithms are implemented using Java's built-in Comparator interface.
- Logging and tracing are handled via the TraceBus (Observer Pattern).