I see! You're asking for a **Low-Level Design (LLD)** for the given system.

Here’s an overview of the LLD for the document ingestion and QnA chat session management system based on your models, APIs, and WebSocket design.

## Low-Level Design (LLD) Overview

### 1. ****User Authentication & Authorization****

**Auth Middleware**:

* + Token-based authentication (API Key).
  + Verify that the provided token is valid.
  + AsyncAuthMiddleware verifies the token using Token.objects.aget().
  + On success, the user is added to the scope for further processing.
  + On failure, send error response and close the WebSocket connection.

**Django User Model**:

* + Uses the User model for authentication.

### 2. ****Document Ingestion & Metadata Storage****

**Models**:

* + DocumentInfo: Stores document metadata such as documentUUID, documentTitle, user, processStatus, etc.
  + ProcessStatus: Stores document processing status (e.g., chunked, processing, success).
  + **Ingestion Logic**:
    - Receive documents via the API (POST /documents/api/ingestion/).
    - Validate the token and check document content (name and text).
    - Split the document into smaller chunks (e.g., using spaCy or another method).
    - Save metadata into DocumentInfo and ProcessStatus.
    - Ingest document chunks into the vector database (Weaviate).

**Error Handling**:

* + Return appropriate error messages (e.g., "Scheduled status not found", 500 for server errors).

### 3. ****Chat Session Management****

* **Models**:
  + ChatSessionManager: Stores session metadata, including sessionUUID, user, sessionStatus, etc.
  + SessionChatInfo: Stores chat logs, including user queries and AI responses.
  + SessionDocumentInfo: Maps documents to chat sessions, specifying whether they can participate in the chat.
* **API Logic**:
  + **Create Session**: POST /qna/api/chat/session
    - Validates token, creates a new session, and stores it in ChatSessionManager.
  + **Add Documents to Session**: PUT /qna/api/chat/session
    - Validate session and add documents to the session via SessionDocumentInfo.
* **Error Handling**:
  + If sessionUUID or documentUUIDs are missing or invalid, return 400 (Bad Request).
  + Return 500 for server errors.

### 4. ****WebSocket for Real-time QnA****

**WebSocket API Endpoint**: /qna/api/chat/docChat/

**Connection Setup**:

* + Authenticate the user and validate sessionUUID in the query string.
  + Fetch the associated documents for the session from SessionDocumentInfo.
  + Send confirmation message upon successful connection.

**Message Flow**:

* + **Incoming Messages**:
    - Expect a user query ({"question": "user's question"}).
    - Validate and process the query.
  + **Outgoing Messages**:
    - **Connection Success**: {"message": "Connected to WebSocket!"}.
    - **Streaming Status**: { "type": "status", "response": "start" } when response generation starts.
    - **AI Response**: { "type": "answer", "response": "AI-generated answer chunk" } streamed chunk by chunk.
    - **Streaming End**: { "type": "status", "response": "end" }.

**Vector Search**:

* + Use the Weaviate vector database to find relevant document chunks.
  + Perform the query against stored vectors (using the documentUUIDs of the current session).

**AI Response**:

* + Combine user query and relevant document context.
  + Call the AI model to generate responses asynchronously and stream them back to the client in real-time.

### 5. ****Database Relationships****

* **DocumentInfo**:
  + Many-to-one relationship with User (the uploader).
  + Many-to-one relationship with ProcessStatus (the current status of the document).
* **ChatSessionManager**:
  + One-to-many relationship with SessionChatInfo (chat logs).
  + One-to-many relationship with SessionDocumentInfo (documents related to the session).
* **SessionDocumentInfo**:
  + Many-to-one relationships with ChatSessionManager and DocumentInfo.

### 6. ****Security Considerations****

**Authentication**:

* + Token authentication ensures that only valid users can access the API and WebSocket connection.

**Authorization**:

* + Ensure that only the user who created the session can interact with their session and documents.

**Rate Limiting**:

* + Implement rate limiting for the WebSocket connections to prevent abuse.
  + Limit the number of concurrent WebSocket connections per user.

### 7. ****Error Handling & Responses****

* For the WebSocket API, handle errors like invalid sessionUUID, parsing issues, and database failures.
* Return clear error messages for API calls with missing parameters or invalid states.

### 8. ****Performance Considerations****

* **Vector Database (Weaviate)**: Ensure that queries to the Weaviate database are efficient and indexed properly.
* **Real-Time Processing**: The WebSocket API must be optimized for real-time message handling, chunked streaming, and maintaining connections.
* **Chunking & Ingestion**: Efficient chunking of documents before sending to the Weaviate database for faster search and retrieval during the QnA session.

## UML Diagram Representation

Here’s how the system can be broken down in terms of entities:

* **DocumentInfo** (Many-to-one with ProcessStatus)
* **ChatSessionManager** (One-to-many with SessionChatInfo, One-to-many with SessionDocumentInfo)
* **SessionDocumentInfo** (Many-to-one with DocumentInfo and ChatSessionManager)
* **SessionChatInfo** (Many-to-one with ChatSessionManager)

Each of these entities communicates based on the WebSocket and API interactions described above.