The Requirements' Implementation Details

1. Group Chat, Private Chat & Recent Messages:

1.1. Database Connection (database.py):

- **Library Used**: *motor* (asynchronous MongoDB driver) and *pymongo*.
- **Why**: *Motor* provides non-blocking access to MongoDB, which is essential for handling high concurrent requests in real-time systems like group chat or private chats.
- **How**:
    - *AsyncIOMotorClient* connects the application to the MongoDB database using environment variables (like *MONGO_USER*, *MONGO_PASSWORD*, etc.).
    - The database class is a singleton that ensures a single connection instance is shared across the application.
    - The *get_db()* method provides access to the database, raising an error if the database connection has not been established.

1.2. Schema Definition (schemas.py):

- **Library Used**: *Pydantic* (for data validation) and *bson* (to handle ObjectId in MongoDB).
- **Why**: *Pydantic* ensures that data structures are validated, which is crucial when dealing with real-time data exchanges in group chat and private chat scenarios. Using *ObjectId* from *bson* ensures compatibility with MongoDB IDs.
- **How**:
    - **UserModel** and **MessageModel** define the structure of user and message documents.
    - *MessageModel* includes attributes like *user_id*, *chat_room_id*, *content*, and *timestamp* to represent each message. Additionally, it supports different message types (text, image, file) through the *MessageType* Enum.

1.3. Real-Time Messaging with Socket.io (sockets.py):

- **Library Used**: *Socket.IO* (for Socket.io communication) and *JWT* (for authentication).
- **Why**: *Socket.IO* enables bi-directional communication between users in real-time, making it ideal for implementing group chat and private chat. *JWT* (JSON Web Tokens) ensures secure authentication during Socket.io connections.
- **How**:
    - **Connect Event**: Authenticates the user by validating the token and assigns them to a chat room using *sio_server.enter_room()*. It also updates the

user's online status and retrieves recent messages and online users to display to the client.
    - **Chat Event**: Handles the transmission of messages from one user to the entire chat room. It calls the *create_message()* function (in *operations.py*) to save the message to the database and emits the new message to all connected users in the room.
    - **Disconnect Event**: Updates the user's online status when they leave a room and notifies other participants.

1.4. Message and User Handling (operations.py):

- **Library Used**: *FastAPI* (for API implementation) and MongoDB operations (through *motor*).
- **Why**: The real-time messaging requires both saving message data to the database and ensuring that user statuses (e.g., online, offline) are updated instantly.
- **How**:
    - **Message Creation**: The *create_message()* function inserts a new message document into the database, including details like *user_id*, *chat_room_id*, and *message_type* (text, image, file). The message is then broadcast to all users in the room.
    - **User Status Updates**: *update_user_online_status_db()* modifies the user's online status in the database, and *rooms_online_users()* retrieves the list of currently online users in a chat room.

1.5. FastAPI Configuration (main.py):

- **Library Used**: *FastAPI* (to set up the REST APIs) and *Socket.IO* (for Socket.io integration).
- **Why**: *FastAPI* is used for its speed and ability to handle asynchronous operations, which are crucial for real-time systems.
- **How**:
    - **Application Setup**: The *FastAPI* app is configured with Socket.io endpoints (using *Socket.IO*) and serves as the core of the chat service.
    - **Middleware and CORS**: The app uses *CORS* middleware to handle cross-origin requests, ensuring that the frontend can communicate with the backend even when hosted on different domains or ports.
    - **Lifespan Management**: The *lifespan* function manages the startup and shutdown processes, ensuring the database connection is established and terminated properly.

1.6. Retrieving Older Messages:

- **Library Used**: *MongoDB* (for storing and querying message history).
- **Why**: Users should be able to retrieve older messages when they scroll up in the chat room, enabling them to view the chat history.
- **How**:

- **Socket.io Event** (*sockets.py*):
  - The *get_more_messages* event is triggered when a user scrolls up to load older messages. This event calls the *get_recent_messages()* function to retrieve older messages from the database.
- *get_recent_messages()* (*operations.py*):
  - This function queries the *messages* collection for a given *chat_room_id* and returns the last 50 messages (or fewer if there are not enough). If the user scrolls up further, it can retrieve messages before a given *before_id* (a reference to the last message shown).

2. Authentication:

2.1. JWT-Based Authentication:

- **Library Used**: *jose* for JWT handling.
- **Why**: JWT provides a stateless and secure way to verify user identity. It encodes user information in a token, which can be decoded and verified by the server on each request without needing to store session information in the database.
- **How**:
  - When a user logs in successfully, a JWT token is generated with the user's email and username, along with an expiration time.
  - This token is then provided to the user, who includes it in the *Authorization* header (*Bearer* token) for accessing protected APIs.
  - The token is validated in subsequent requests using the *get_current_user()* function to ensure the user has the necessary credentials to access the system.
- **Token Creation** (utils.py):
  - *create_access_token():* Generates a JWT token containing user information (email, username) and an expiration date. The token is encoded using a secret key (stored securely via Docker Secrets) and the HS256 algorithm.
- **Token Validation** (inits_apis.py):
  - *get_current_user():* This function is used as a dependency in FastAPI routes to authenticate users before granting them access to the API. It decodes the token, verifies the user's identity, and ensures the token is valid.
- **How it works**:
  - The function decodes the JWT using the *SECRET_KEY* and validates the user's identity by checking their email and username against the database.
  - If the token is valid, it retrieves the corresponding user from the database and grants them access to the API.

2.2. Secure Token Storage using Docker Secrets:

- **Technology Used**: Docker Secrets.

- **Why**: Handling sensitive data such as the *SECRET_KEY* used to encode and decode JWT tokens must be done securely. Docker Secrets allow us to store this key securely and make it accessible only to the FastAPI service.
- **How**:
    - The **Docker Compose** file defines the *SECRET_KEY* as a secret stored in */run/secrets/secret_key*.
    - Inside the FastAPI application, the key is loaded using the path specified in the secrets file. If the key is not found in the environment variables, it is fetched securely from the Docker Secrets file.
    - **Code Integration**: The FastAPI service uses the environment variable *SECRET_KEY_FILE* to load the secret key in a secure manner.
    - The key is used in both the *create_access_token()* and *get_current_user()* functions for encoding and decoding JWT tokens.

2.3. User Password Management:

- **Library Used**: *passlib* (for password hashing and verification).
- **Why**: Storing raw passwords is insecure. Instead, passwords are hashed using *bcrypt*, ensuring that even if the database is compromised, the passwords remain protected.
- **How**:
    - **Password Hashing** (utils.py):
        - *get_password_hash():* Hashes the plain password using the *bcrypt* algorithm before storing it in the database.
    - **Password Verification**:
        - *verify_password():* Compares the plain password provided during login with the hashed password stored in the database.

2.4. APIs for Authentication:

- **Login API** (*inits_apis.py*):
    - **Endpoint**: */token*
    - **How it Works**:
        - The user submits their credentials (email/username and password) through the *OAuth2PasswordRequestForm*.
        - The backend verifies the credentials and returns an access token if they are valid. The access token is used for authenticating subsequent requests.
        - If the credentials are incorrect, an *HTTP 401 Unauthorized* error is returned.
- **Refresh Token API**:
    - **Endpoint**: */refresh-token*
    - **How it Works**:
        - The refresh token API allows users to get a new access token without having to log in again, as long as the user is already authenticated.

        - The API returns a new token with an updated expiration time for continued access.

2.5. Admin Authentication:

- **How**: The *get_current_admin_user()* function (in *admin_apis.py*) extends the functionality of *get_current_user()* to ensure that only users with *is_admin = True* can access the admin routes.
- **Why**: This ensures that only authorized users with admin privileges can perform administrative tasks like managing users or chat rooms.
- **Admin Check**:
    - If the user does not have admin privileges, an *HTTP 403 Forbidden* error is raised, preventing unauthorized access to admin-only resources.

2.6. Authentication Middleware in FastAPI:

- **Library Used**: *OAuth2PasswordBearer* (from *fastapi.security*).
- **Why**: This built-in FastAPI security mechanism provides an easy way to implement token-based authentication using OAuth2 with password flow.
- **How**:
    - *oauth2_scheme:* Defines the token URL (*/token*) for the OAuth2PasswordBearer flow.
    - This scheme is used in the *get_current_user()* function to retrieve and validate the token provided in the *Authorization* header of incoming requests.

3. Online Users & Multimedia:

3.1. Online Users in Private Chats:

- **Library Used**: *MongoDB* (for storing user status) and *FastAPI* (for API implementation).
- **Why**: The system must efficiently track users' online statuses and retrieve this information from the database to display it to other users.
- **How**:
    - **API Endpoint** (*inits_apis.py*):
        - */pv-online-users:* This API retrieves a list of users that the current user has private chats with and are currently online. It utilizes the *get_online_users_pv()* function to query the database for private chat sessions and filter out users who are online.
    - *get_online_users_pv()* (*operations.py*):
        - This function queries the *chat_room_sessions* collection to find all private chat sessions associated with the current user. For each session, it checks if the other user in the session is online and adds their username to the list of online users.

3.2. Online Users in Group Chats:

- **Library Used**: *Socket.IO* (for real-time updates) and *MongoDB* (for user status management).
- **Why**: Real-time display of online users in a group chat is crucial for a dynamic and interactive chat experience.
- **How**:
    - **WebSocket Event** (*sockets.py*):
        - When a user connects to a group chat room, the system checks the online status of other users in the room using the *rooms_online_users()* function. The list of online users is emitted to the client in real-time via Socket.io.
    - *rooms_online_users()* (*operations.py*):
        - This function retrieves all active chat room sessions for a given room, checks which users are online, and returns their usernames. The system avoids displaying the current user's own username.

3.3. User Online Status Update:

- **Library Used**: *FastAPI* (for API implementation) and *MongoDB* (for updating online status).
- **Why**: The system needs to update each user's online status when they log in, log out, or switch between online and offline modes. This ensures accurate real-time status for all users.
- **How**:
    - **API Endpoint** (*inits_apis.py*):
        - */user/online-status:* This endpoint allows a user to update their online status. When a user changes their status, the *update_user_online_status_db()* function is called to update the *is_online* field in the MongoDB database.
    - *update_user_online_status_db()* (*operations.py*):
        - This function updates the user's online status in the *users* collection by setting the *is_online* field based on the provided boolean value.

3.4. Sending Multimedia Messages (Images & Files):

- **Library Used**: *aiofiles* (for asynchronous file handling) and *MongoDB* (for storing message metadata).
- **Why**: Users should be able to send not only text messages but also multimedia content (images and files). This enhances the chat experience by allowing rich media sharing.
- **How**:
    - **Socket.io Event** (*sockets.py*):

        - When a user sends a message, the *chat* event handles the message, including any multimedia content. If the message contains an image or file, the file is saved to the server using *aiofiles*.
    - *create_message()* (*operations.py*):
        - This function is responsible for creating a message entry in the database. If the message contains multimedia content, the *file_name* and *file_path* fields are populated and stored alongside the text content.
        - The uploaded file is saved in the *uploads* directory, and its path is stored in the message document.