System Design Document

For the Shinkai Chat App

Functional Requirements:

- User Registration and Authentication
- One on One Chat
- Group Chat functionality
- Real-time message updates

API Endpoints:

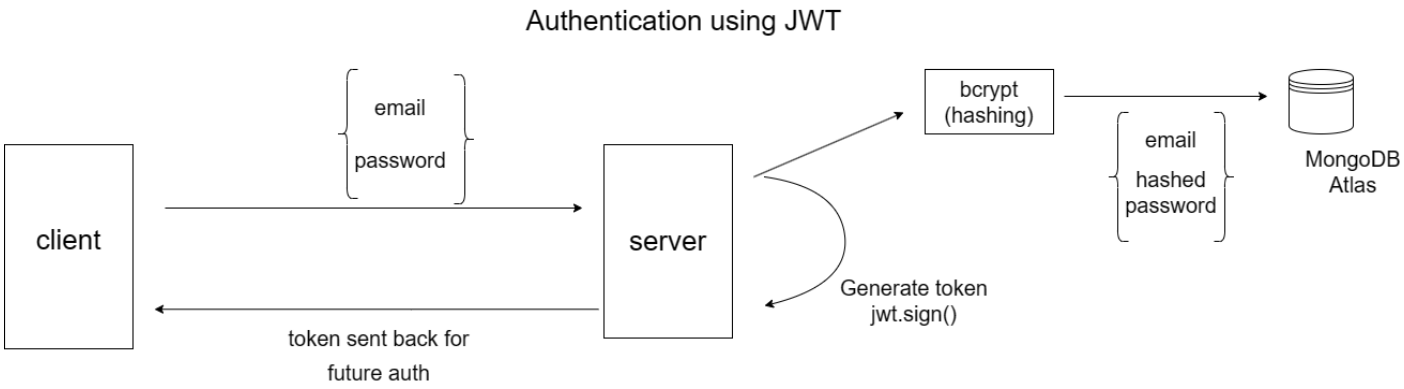
- First, we consider **user authentication**, **/api/user**:
    - **POST /api/user**: Register new user
    - **POST /api/user/login**: Authenticate the user/Login
    - **GET /api/user?search=**: Get or Search all users
- Now, we consider the API endpoints for **one on one and group chats**, **/api/chat**:
    - **POST /api/chat/:** Create or Fetch One to One chat
    - **GET /api/chat/:** Fetch all chats for a user
    - **POST /api/chat/group**: Create new group chat
    - **PUT /api/chat/rename:** Rename group
    - **PUT /api/chat/groupremove:** Remove user from group
    - **PUT /api/chat/groupadd:** Add user to group
- Finally, the API endpoints for the **messages** themselves, **/api/message:**
    - **GET /api/Message/:chatId**: Get all messages
    - **POST /api/Message/**: Create new message

User Registration and Authentication:

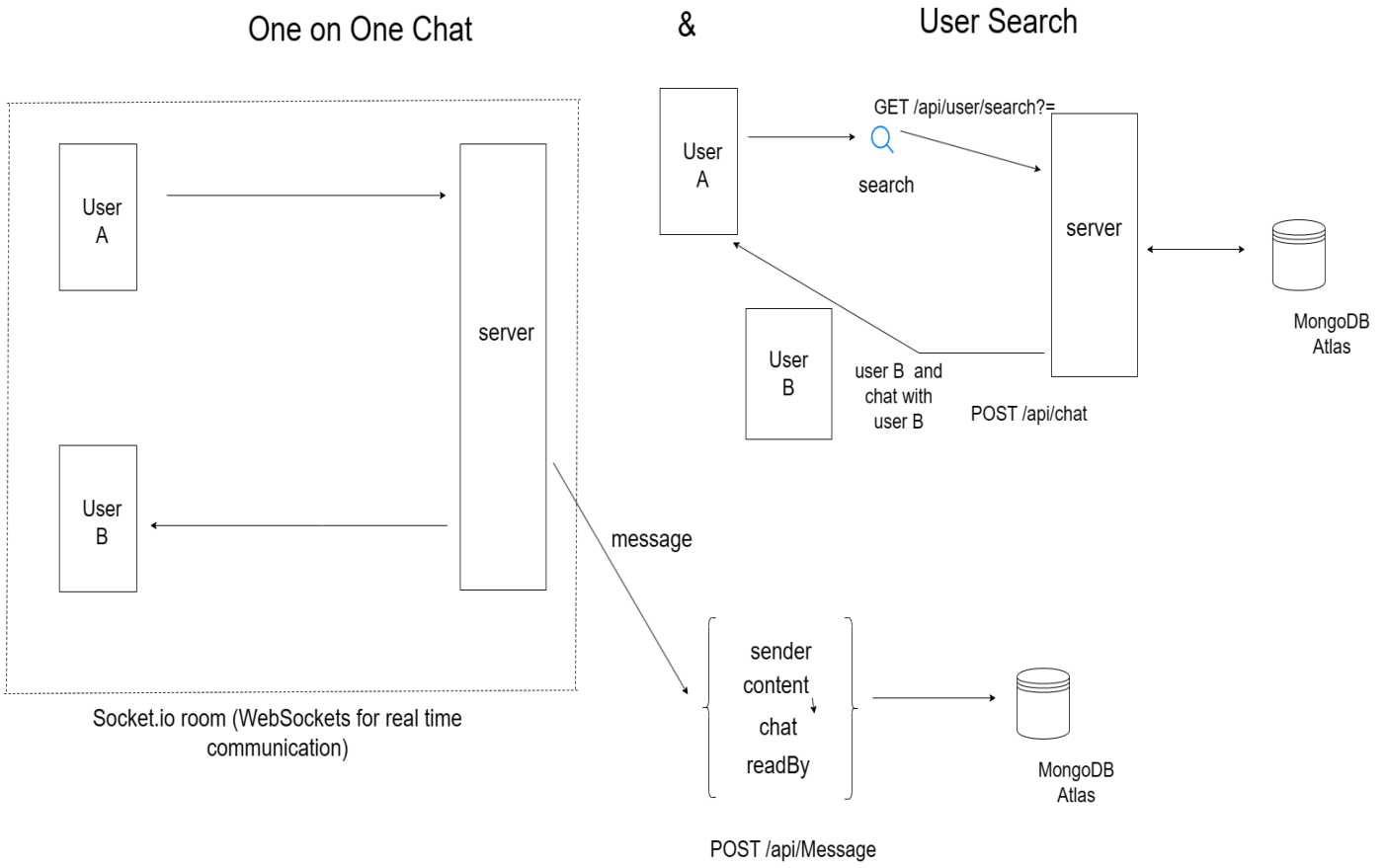
The user registration and authentication takes place using JSON Web Tokens (JWT). The user sends their name, email and password to the server at the time of registration. The password is hashed using **bcrypt**. To make the output unpredictable, *we salt it*. The name and email sent by the user, on the other hand, is used to **generate a JWT token**, which is **sent back to the client**, while the data sent by the client to the server is **stored in a MongoDB Atlas collection, albeit with the hashed passwords** so that **unauthorised or unwanted personne**l do not have access to user passwords, which

would *otherwise signify a breach of trust, privacy and security*.

This JWT token, which has been sent back to the client, can now be used to authenticate the client on any API request. In case the JWT token is tampered with, its signature changes, thus invalidating it and rendering it unfit for authentication.



One on One Chat (Messaging) & User Search:



This entire design consists of two sub-designs: One on one chat, ie the messaging system between users and User search.

Tackling **User Search** first, the client makes use of two API endpoints provided to it. The first is the **GET /api/user/search?=** endpoint to actually search for a user and get a result back from the server. Once there is a result, the client then fires a **POST** request to **/api/chat**. If there already exists a chat between the currently logged in user (client) and the searched user, the API endpoint simply returns that chat, otherwise it creates a new chat.

The second sub-design is that of the **messaging system: one on one chat**. As messaging needs to be real-time, **WebSockets** seemed to be the obvious and industry standard solution to pick here. **Socket.io**, a high level wrapper around websockets, came in clutch to maximise productivity, DX, as well as helping build the group chat feature as later discussed.

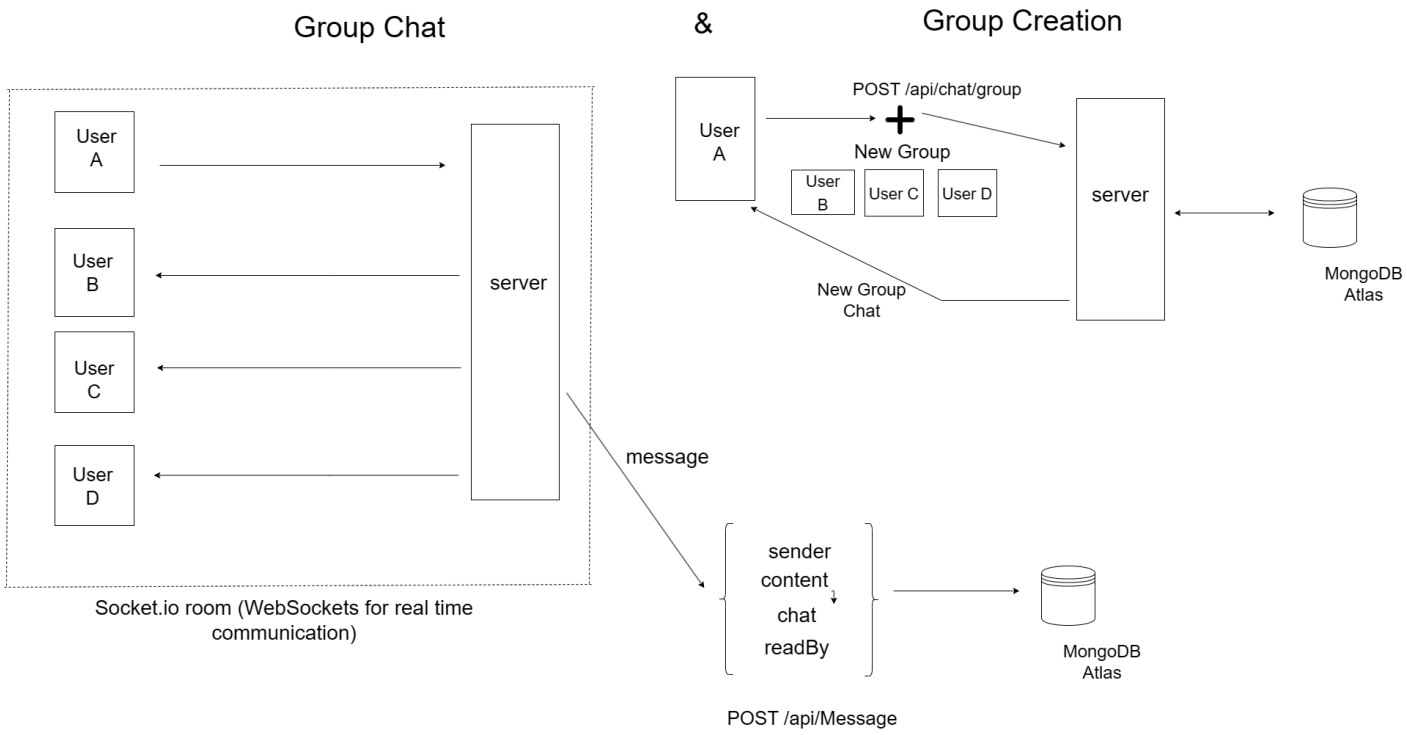
For two users User A and User B involved in one on one messaging/chatting, with a message going from A to B:

- User A, User B and the server all **join the same socket.io room**. This allows private messaging between User A and User B.
- User A **emits a new message**. The server receives this and **re-emits this back** to User B.
- Meanwhile the message is **stored in a MongoDB Atlas collection** through the **POST /api/message** request, and the **chats are fetched** whenever the two users chat again.

Group Chat (Messaging) & Group Creation:

Group creation involves searching, through the same design as outlined previously, for users, then firing a **POST /api/chat/group** request to the server for creating a group.

Group chat is not extremely different from one to one chat. If Users A, B, C and D are involved in a Group, said users can simply join a room (again, already outlined in the one-on-one chat design section) and then the process remains fairly similar, with messages being, emitted to the server, and then the server re-emitting the messages to the rest of the users in the room (instead of only one).



Notifications:

Shinkai handles notifications completely client side. If the user receives a messages from a chat that is not the active (currently being used) chat, the client records the sender's name, and displays that as a notification. The count is increased with each chat.