How to RUN Frontend:

Npm start

How to RUN Backend:

Node app.js

View 1

View 2

View 3

User Registration

API call to store username, password and email during initial registration. This API requires a JSON input as given below

API URL: ***(*POST*)*** ***http://localhost:8085/api/users/***

Request format:

req:{ user: { username : String, password: String, email: String } }

Response format:

res:{ userId: String }

User Login

API call to post username and password during login. This API requires a JSON input as given below

API URL: ***(POST) http://localhost:8085/api/users/login{Enter the username}***

Request format:

req:{ user: { username : String, password: String } }

Response format:

res:{ userId: String }

Create Room

API call to create a room to initiate chat between set of individuals. This API requires a JSON input as given below

API URL: ***(POST) http://localhost:8085/api/rooms/Rooms***

Request format:

req:{ room: { displayName: String, type: String, createdBy: String, createdAt: String }

Response format:

Res: { _Id: String }

Add Member

API call to add a member to a chat room. This API requires a JSON input as given below

API URL: ***(Post) http://localhost:8085/api/rooms/Rooms/{enter the room ID}/users***

Request format:

req:{ userId: <String> }

Response format:

Status: 200 OK

To Retrieve List of Chat Rooms

API call retrieve the list of existing public, private and unapproved rooms. This API requires a JSON input as given below

API URL: ***(Get) http://localhost:8085/api/userrooms/users/{Enter user ID}/Rooms/{Enter chat room type}***

Request format:

None

Response format:

res:{ _Id: <String>, displayName: <String>, createdBy: <String>, createdAt: <String> }

Retrieve All Messages

API call to retrieve all the messages of a particular chat room. This API requires a JSON input as given below

API URL: ***(Get) http://localhost:8085/api/rooms/Rooms/{Enter room ID}/Messages***

Request format:

None

Response format:

res:{ messages: [Array of message objects] }

Each message object:{ userId: <String>, roomId: <String>, message: <String>, sentAt: <String> }

Send Messages

API call to send messages for storing in the database and to send to other users in the chat room. This API requires a JSON input as given below

API URL: ***(Post) http://localhost:8085/api/rooms/Rooms/{Enter room ID}/Messages***

Request format:

req:{ messages:{ userId: <String>, message: <String>, sentAt: <String> } }

Response format:

res: { messages:{ userId: <String>, roomId: <String>, message: <String>, sentAt: <String> } }

Confirm Membership

API call when a member confirms the membership to enter a chat room. This API requires a JSON input as given below

API URL: ***(Put) http://localhost:8085/api/rooms/Rooms/{Enter room ID}/Users/{Enter user ID}***

Request format:

req:{ joinStatus: int }

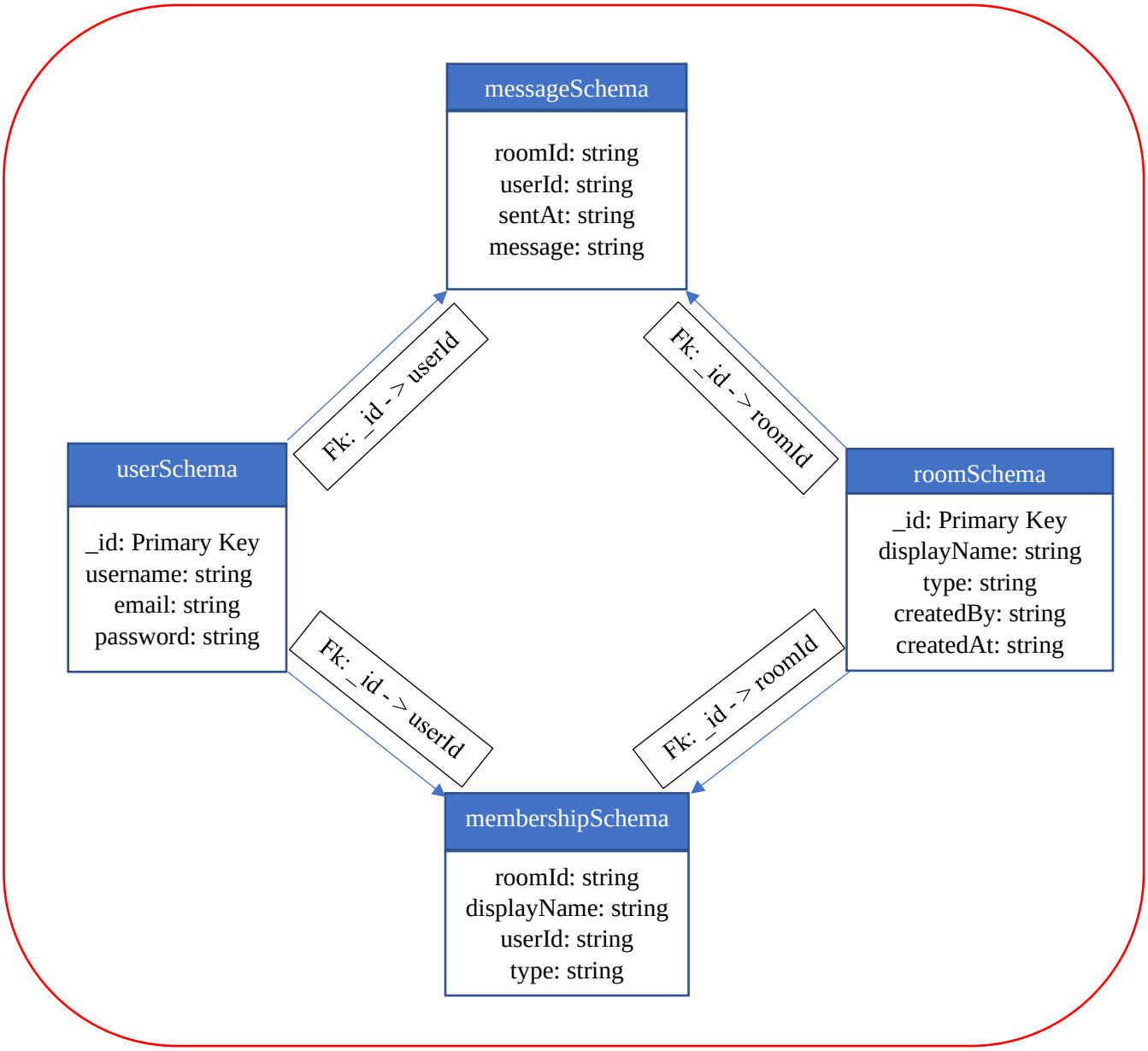
Response format:

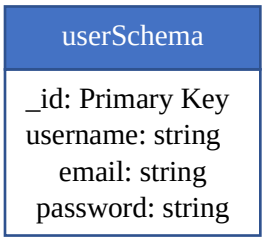
Status: 200 OK

The application requires collection of a lot of data for its effective functioning. Categorically, this required data has been segregated as explained below and have been stored in the database to be effectively queried, managed and manipulated. We need the below data to run the application successfully:

- User Info: Firstly, we need the user info who is using the application and logging into it to send/receive chats.
- Chat Room meta Info: A metadata of all the chats between different individuals needs to be stored along with when and by whom the chats were initiated.
- The members of different chat rooms that exist, the contents of chat inside these chat rooms and who all invited/accepted the room membership requests.
- The message data in particular rooms that was exchanged between individuals.

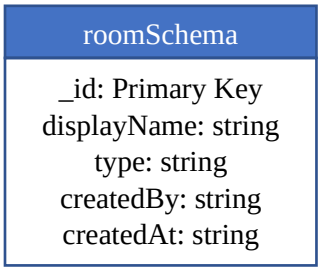
Based on the need of how the data needs to be collected and stored, a database schema has been designed and drawn below to effectively convey the schema:





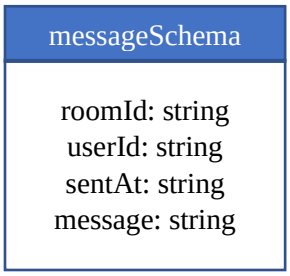
The *userSchema* collection contains the above shown document objects. Each of its significance is explained below:

_id: Primary key of the table that stores a unique identity of a user to be referenced across the database
userName: Display name of the registered user
email: Email address of the user
password: Password of the user



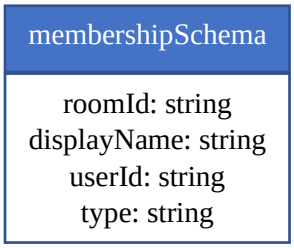
The *roomSchema* collection contains the above shown document objects. Each of its significance is explained below:

_id: Primary key of the table that stores a unique identity of the room to be referenced across the database
displayName: Unique name of the chat room that has been created for a set of users
type: This is to signify if the specific room is either private or public
createdBy: This fields stores the name of the user who created the chat
createdAt: It stores the timestamp at which a particular chat is exchanged



The *messageSchema* collection contains the above shown document objects. Each of its significance is explained below:

roomId: Unique ID (foreign key) of the chat room that a set of individuals are a part of
userId: This is to signify if the specific room is either private or public
sentAt: This field stores the timestamp at which the message has been sent
message: This field stores the message content that has been sent



The *membershipSchema* collection contains the above shown document objects. Each of its significance is explained below:

roomId: Unique ID (foreign key) of the chat room that a set of individuals are a part of
displayName: This field is to store the display name of the chat room that a user would be in, if approved
userId: This field stores the ID of the user who is pending approval to be a part of the chat room
type: This field stores the status of the membership – Unapproved or Private (Public isn't taken into account as public room needs no approvals)

WORKING:

Firstly, a user can register into the application by furnishing his details in the login page, which is posted to the database where the users schema is updated. These details are used to validate a user's credentials while logging in later. Kindly note that the emphasis was to implement and execute the functionality, therefore, the username-password validation and authentication is not executed at this point of time.

Successful login redirects the application to a view which contains two options – Create Room & Dashboard. Create Room functionality lets us create a new room with individuals. Dashboard consists of three modules – Private Rooms, Public Rooms and Unapproved Rooms. The Private Rooms shows the list of private chat rooms that the user has with others. The Public Rooms shows the list of public chat rooms that others have created and sent the messages for the user to read. The Unapproved Rooms shows the list of pending chat room requests that has been requested by other users. Accepting these requests creates new chat rooms with requested users.

When we attempt to create a new room, we would select the type of room-Private or Public and select required individuals whom you wanted to be part of the room. Upon clicking submit button, a create room API gets called at the backend and the room will be created. If the creation was successful, a series of add member API requests are sent to the server for adding desired members. Once the members are added, the create room function ends.

When the dashboard functionality is rendered, the dashboard hits 3 different API calls for getting the user's chat room metadata. Each of the modules as mentioned above contains several chat rooms as components which can be selected and navigating to corresponding chat rooms. However, unapproved chat rooms have a different functionality when compared to the other two modules. Upon selecting a chat room under the unapproved list, the approval confirmation dialog box pops-up for confirmation to join the specific chat room. On clicking the pop-up, user will be entered into the chat room as the confirm member API call gets hit.

The views of Create Chat Room and Dashboard are hosted on REST APIs while the Chat Room totally works on Web Socket protocol. Inside a chat room, a sent message is broadcasted to all the room members and a post message API call is hit to store the messages in the database. Later, these messages can be retrieved through the get messages API from backend.