**Project Documentation**

1. **Git Repository Organization**

* **Repository Structure**: The repository follows a well-structured format where the client-side (Angular) and server-side (Node.js) code are kept in their respective folders. The Angular app is organized into components and services, while the Node.js server manages the backend logic.
* **Branching**: The development process followed a simple branching strategy. A single main branch was used for development and was updated frequently. Bug fixes and enhancements were committed regularly.
* **Update Frequency**: The repository was frequently updated to track the progress of the project, with each key feature (such as user management, group management, and messaging) being committed as independent tasks.
* **Server/Frontend**: The server-side files are located under /Chat-server, while the frontend Angular app is located under /src with all related components, services, and modules.

1. **Data Structures Used**

**Client-Side (Angular):**

* **Users**

{

id: number;

username: string;

password: string;

roles: string[]; // ['Super Admin', 'Group Admin', 'User']

groups: Group[];

}

* **Groups:**

{

id: number;

name: string;

userIds: number[]; // List of user IDs

channels: Channel[];

}

* **Channels:**

{

id: number;

name: string;

}

* **Notifications**

{

message: string;

groupId: number;

userId: number;

}

**Server-Side (Node.js):**

* **Users**: Stored in a JSON file (users.json). Each user has a unique id, username, password, roles, and associated groups.
* **Groups**: Managed in memory. Each group has a unique groupId, groupName, adminIds (for Group Admins), and userIds.
* **Channels**: Managed in memory, where each channel is associated with a group and contains the list of user IDs that belong to it.

1. **Angular Architecture**

• **Components**:

* **login.component.ts**: Handles user authentication (login and registration).
* **dashboard.component.ts**: Displays the groups and channels a user is part of, allows sending/receiving messages.
* **group.component.ts:** Manages group creation, channel creation, and user approvals within groups.
* **superadmin.component.ts**: Allows Super Admin to manage users, create groups, promote users to Group Admin, and manage channels.

• **Services:**

* **auth.service.ts**: Manages user login, registration, and user state.
* **group.service.ts**: Handles group and channel-related operations such as creating groups/channels, sending join requests, and loading notifications.
* **chat.service.ts**: Manages WebSocket communication for sending/receiving messages in real-time.

• **Models:**

* **User model**: Represented as objects containing id, username, password, roles, and groups.
* **Group model**: Represented as objects containing id, name, userIds, and channels.
* **Channel model**: Represented as objects containing id and name.
* **Notification model**: Represented as objects containing message, groupId, and userId.

• **Routes:**

* + **/login**: Displays the login and registration page.
  + **/dashboard**: Displays user-specific group and channel data.
  + /**group**: Displays the group management interface for Group Admins.
  + **/super-admin**: Displays the user management interface for Super Admins.

1. **Node Server Architecture (REST API)**

* **Modules**:
  + **express**: Manages HTTP requests and routes.
  + **socket.io**: Manages real-time messaging functionality between clients and the server.
  + **cors**: Allows cross-origin requests between the server and frontend.
  + **body-parser**: Parses incoming request bodies in a middleware.
* **Functions**:
  + User login and registration (/login, /user).
  + User promotion and role updates (/user/promote, /user/upgrade).
  + Group and channel creation (/group, /channel).
  + Real-time messaging with WebSocket (Socket.io).
* **Files**:
  + server.js: Main entry point for the backend, handles routing and real-time communication.
  + users.json: Contains all the user data, simulating a database for simplicity.
* **Global Variables**:
  + users: Contains the list of all users, fetched from the JSON file.
  + groups: Contains the list of all groups, managed in memory.
  + channels: Contains the list of all channels, managed in memory.

1. **List of Server-Side Routes (REST API)**

* **POST** /login:
  + Parameters: username, password.
  + Returns: Authenticated user information.
  + Purpose: To authenticate users and redirect them based on their roles.
* **POST** /user:
  + Parameters: username, password, email.
  + Returns: Confirmation message for successful registration.
  + Purpose: To register new users.
* **POST** /user/promote:
  + Parameters: userId.
  + Returns: Confirmation message for promotion to Group Admin.
  + Purpose: To promote a user to Group Admin.
* **POST** /user/upgrade:
  + Parameters: userId.
  + Returns: Confirmation message for promotion to Super Admin.
  + Purpose: To upgrade a user to Super Admin.
* **POST** /group:
  + Parameters: groupName, adminId.
  + Returns: Confirmation message and newly created group details.
  + Purpose: To create a new group.
* **DELETE** /user/:userId:
  + Parameters: userId.
  + Returns: Confirmation message for successful user removal.
  + Purpose: To remove a user from the system.

1. **Client-Server Interaction**

* **Login and Registration**:
  + Upon login, the client sends the username and password to the /login route. If successful, the authenticated user is stored locally in the browser.
  + New users register via /user, which stores their information in the users.json file.
* **Group and Channel Management**:
  + When a Group Admin creates a new group or channel, a request is sent to /group or /channel, and the server updates its in-memory list of groups or channels.
  + These updates are reflected on the client side through Angular's GroupService, which re-fetches the latest data and updates the displayed lists in real-time.
* **Messaging**:

The client subscribes to WebSocket events using Socket.io. When a message is sent, the server broadcasts the message to all clients connected to the same channel, and the message list on the client side is updated dynamically