API Documentation

---

1. POST /login

DESC: Get username and password for existing user login
BACKEND : Check if username and password is correct in UserDB and if username present in Banned List.

REQUEST
{
	”username”: <string>,
	“password”: <string>
}

RESPONSE
{
	"errCode" : 0,
	"data" : <object> [User Object],
},
{
	"errCode" : 1,
	"data" : <string> [a string of fail reason], (Banned, act and pwd do not match, etc)
}

---

2. POST /register

DESC: Register a new user

BACKEND : Add user to UserDB

REQUEST
{
	”username”: <String>,
	“password”: <String>,
	“age” : <Number>,
	“imageUrl”:<String>,
	“interests” :<String>
	“school”: <String>
}

RESPONSE:
{
	"errCode" : 0,
	"data" : <object> [User Object],
},
{
	"errCode" : 1,
	"data" : <string> [Username existed],
}

---

3. GET /getPublicChatrooms

DESC: get public chat rooms list

BACKEND : Get Public chat room names and chatroomID from ChatroomDB

REQUEST: {}

RESPONSE:
{
	"errCode" : 0,
	"data" : List<{chatRoomId: <long>, chatRoomName: <string>, isPrivate: <boolean>}>,
},
{
	"errCode" : 1,
	"data" : <string> [Fail reason],
}

---

4. GET /getMyChatrooms/

DESC: get my chat rooms

BACKEND: From User Object get all chat rooms names and roomID's with current user participated.

RESPONSE:
{
"errCode" : 0,
"data" : List<{chatRoomId: <long>, chatRoomName: <string>, isPrivate: <boolean>}>,
},
{
"errCode" : 1,
"data" : <string> [Fail reason],
}

---

5. <Poll every 5s> GET /getUserNotification

DESC: Get User Notifications (Private Room invitations etc.)

BACKEND: Get all Notifications from the MessageDB for that particular userID.

RESPONSE:
{
"errCode" : 0,
"data" : List<{NotificationId: <long>, NotificationContent: <string>, SenderId:<long>}>,
“data”: List<{
id: <long>,
content: <string>,
type: “invite” | “message”,
status: Enum{ ACCEPTED, DECLINED, NO_STATUS(PENDING) }
},
{
"errCode" : 1,
"data" : <string> [Fail reason],
}

---

6. POST /createRoom

DESC: Create a new Chat room.
BACKEND: Create Chatroom object from the request using ChatRoomDB, check if Chatroom name is unique.

REQUEST: {
“roomName” : <String>,
“roomSize” : <Integer>,
“isPrivate” : <Boolean>,
}

RESPONSE:
{
"errCode" : 0,
“data” : “success”,
},
{
"errCode" : 1,
"data" : <string> [Fail reason],
}

---

7. GET /getListofUser/:chatroomId

DESC: Get a user list from the indicated chatroom.

BACKEND : getList of all the users from a particular Chat room using the ChatroomDB.

RESPONSE:
{
"errCode" : 0,
"data" : List<{userId: <long>, userName: <string>, userAvatar: <string>}>,
},
{
"errCode" : 1,
"data" : <string> [Fail reason],
}

---

8. GET /getListofBlockedUser/:chatroomId

DESC:Returns list of Blocked users for a particular chatroom.

BACKEND : getListofBlocked users for a particular chatroom using ChatroomDB.

RESPONSE:
{
"errCode" : 0,
"data" : List<{UserId: <long>, UserName: <string>, UserAvatar: <string>}>,
},
{
"errCode" : 1,
"data" : <string> [Fail reason],
}

---

9. GET /getChatRoom/:chatroomId

DESC: get the chatroom user clicked.

BACKEND:gets the chatRoom name and chatRoomId from the ChatroomDB using chatroomID Also check if it is admin who is viewing the chatRoom.

RESPONSE:
{
"errCode" : 0,
"data" :{ChatRoomId: <long>, ChatRoomName: <string>, isPrivate: <boolean>
messages(filtered): List<Msg object>,isAdmin : Boolean, isBlocked: <boolean>},
},
{
"errCode" : 1,
"data" : <string> [Fail reason],
}

---

10. POST /leaveChatRoom

DESC: delete the room from the user's list.

BACKEND: Gets the user from UserDB and removes the Chatroom from the users list of Chat rooms.

REQUEST:
{
	~~“user”: <UserId>~~
	“chatRoomId”: <ChatRoomId>
}

RESPONSE:
{
	"errCode" : 0,
	“errMsg” : “success”,
},
{
	"errCode" : 1,
	"errMsg" : <string> [Fail reason],
}

11A. POST /joinRoom

DESC: user joins an existing chat room.

BACKEND : gets the user from userDB and add the chatroom to users list of chatroom with current timeStamp.

REQUEST:
{
	~~“user”: <UserId>~~
	“chatroomId”: <ChatroomId>
}

RESPONSE:
{
	"errCode" : 0,
	"errMsg" : “success”,
},
{
	"errCode" : 1,
	"errMsg" : <string> [Fail reason],
}

11B. POST /opInvitation/:notificationIdIc

DESC: user joins an existing chat room.

BACKEND : gets the user from userDB and add the chatroom to users list of chatroom with current timeStamp.

REQUEST:
{
“accept”: <boolean> // false for decline
}

RESPONSE:
{
"errCode" : 0,
"errMsg" : “success”,

},
{
"errCode" : 1,
"errMsg" : <string> [Fail reason],
}

12.<ADMIN> POST /removeUser

DESC: Removes the user from the Chatroom.

BACKEND : Remove a user from the chat room user list.

REQUEST:
{
“user”: <UserId>
“chatRoomId”: <ChatRoomId>
}

RESPONSE:
{
"errCode" : 0,
"errMsg" : “success”,
},
{

"errCode" : 1,
"errMsg" : <string> [Fail reason],
}

13.<ADMIN> POST /blockUser

DESC: Blocks the selected user from the chat room.

BACKEND : Add the user to the blockList in a chatroom object.

REQUEST:
{
"user": <UserId>
"chatRoomId": <ChatRoomId>
}

RESPONSE:
{
"errCode" : 0,
"data" : "success",
},
{
"errCode" : 1,
"data" : <string> [Fail reason],
}

14.<ADMIN> GET /getListofUserToInvite /:chatroomId

DESC: Returns the list of users which are not part of this chat room so that Admin can send invites.

BACKEND : Get a filtered list of users who are not part of this chat room from UserDB.

RESPONSE:
{
"errCode" : 0,

"data" : List<{UserId: <long>, UserName: <string>, UserAvator: <string>}>,
},
{
"errCode" : 1,
"data" : <string> [Fail reason],
}

15.<ADMIN> POST /sendInvite

DESC: Admin clicks on the Send Invite button.

BACKEND : Generate a notification, send it to the user, Make Notification using MessageDB and send it to userId along with Chatroom name as part of the Notification text.

REQUEST:
{
“user” : <UserId>
“chatRoomId” : <ChatRoomId>
}

RESPONSE:
{
"errCode" : 0,
"data" : “success”,
},
{
"errCode" : 1,
"data" : <string> [Fail reason],
}

16.<ADMIN> POST /unblockUser

DESC: Unblock a user in a chat room.

BACKEND : Get the chatroom from chatRoom DB and remove the userID from the blocked List of that chatRoom.

REQUEST:
{

“user” : <UserId>
“chatRoomId” : <ChatRoomId>
}

RESPONSE:
{
"errCode" : 0,
"data" : “success”,
},
{
"errCode" : 1,
"data" : <string> [Fail reason],
}

---

17. POST /editMessage

DESC: edit a previous message in the chat room.

BACKEND: Get the chat room from ChatroomDB, update the message in the message list with the new content. Then send the message through webSocket.

REQUEST:
{
“messageId”: <MessageId>
“content”: <string>
}

RESPONSE:
{
"errCode" : 0,
"data" : “success”,
// the edited message will be sent by socket
},
{
"errCode" : 1,
"data" : <string> [Fail reason],
}

---

18. POST /removeMessage

DESC: Delete(Recall) a previous message.

BACKEND : Check if the user is the message owner(or Admin) and then get the Chatroom from charRoomDB and delete the specified message from the message list.

REQUEST:
{
	“messageId”: <MessageId>
	“chatRoomId”: <ChatroomId>
}

RESPONSE:
{
	"errCode" : 0,
	"data" : “success”,
	// The removed message id will send by webSocket
},
{
	"errCode" : 1,
	"data" : <string> [Fail reason],
}

---

19. POST /LeaveAllChatroom

DESC: User Leaves all chat rooms.

BACKEND : get the user from userDB and empty the Chatroom map for that user.

RESPONSE:
{
	"errCode" : 0,
	"data" : “success”,
},
{
	"errCode" : 1,
	"data" : <string> [Fail reason],
}

---

20. <Poll every 5s> GET /getUserInfo

DESC: Get user information.

BACKEND : Will check the message from each user and update the hateSpeech count and return the User Object.

RESPONSE:

```
{
        "errCode" : 0,
        "data" : <object> [User Object],
},
{
        "errCode" : 1,
        "data" : <string> [Fail reason],
}
```

---

21. GET /logout ??

Web Socket

ws://locahost:4567/chatapp?userId=1

FRONTEND -> BACKEND

Public Messages

```
{
        “type”: “public”,
        “content”: <String>,
        “senderId”: <UserId>,
        “receiverId”: -1,
        “roomId”: <RoomId>
}
```

Private Messages

```
{
        “type”: “private”,
        “content”: <String>,
        “senderId”: <UserId>,
        “receiverId”: <UserId>,
        “roomId”: <RoomId>
}
```

BACKEND -> FRONTEND

Broadcast new message to the user (when server received a new message)

```
{
        “messageId” : <MessageId>,
        “type”: “new” | “edit” | “remove”, // If type is empty, this message is a system notification
        “isPrivate”: <boolean>,
        “content” : <String>,
        “sender”: {
                        “userId”: <UserId>,
                        “userName”: <String>,
                        “userAvatar”: <String>,
        },
        “receiver”: {
                        “userId”: <UserId>,
                        “userName”: <String>,
                        “userAvatar”: <String>,

        }
}
```

Broadcast edited message to the user

```
{
        “messageId” : <MessageId>,
        “type”: “edit”,
        “content” : <String>
}
```

Broadcast removed message to the user

```
{
        “messageId” : <MessageId>,
        “type”: “remove”,
}
```

Use Cases and Our API

	Use Cases		API Usage	Remarks
2	A user must initiate the creation of a chat room with a room size		6. POST /createRoom	
3	A user can only send a message to someone else in the same chat room as them		7. GET /getListofUser/:chatroomId	
4	A unblocked user should be notified that their message has been received			To be handled via Websocket, if user clicks on send and no error back from websocket that means message has been received
5	A user may be in multiple chat rooms (public and private)		11. POST /joinRoom	
6	A user can choose to exit one chat room or all chat rooms		19. POST /LeaveAllChatroom	
7			10. POST /leaveChatRoom	
8	At least one of the users in the chat must be the admin		6. POST /createRoom	User who creates the group is the Admin
9	Admin broadcast approved requests of qualified users. Admin monitor users and messages. Admin can ban users and delete messages.		18. POST /removeMessage	
10			13.<ADMIN> POST /blockUser	
11	A user creates an account with a profile that includes their age, school, and interests		2. POST /register	
12	A user can join a public chat room (no special interests) if they are able to join. Admin invite users to join private rooms. The user does not see the chat history of messages sent before they entered the room.		11. POST /joinRoom	
13			15.<ADMIN> POST /sendInvite	
14			14.<ADMIN> GET /getListofUserToInvite /:chatroomId	

15	A user will be warned and eventually forcibly banned from all chat rooms if the user uses the phrase “hate speech” in a message.		20. GET /getUserInfo	
16	An unblocked user can send direct or broadcast messages to all users in the chat room		7. GET /getListofUser/:chatroomId	User either send a broadcast(public) message in the group or send a private to a particular receiver from the chatRoom user list
17	A user can determine who is in the chat room		7. GET /getListofUser/:chatroomId	
18	A user can determine what chat rooms they have joined and what public rooms they can join		4. GET /getMyChatRooms/:userID	
19			3. GET /getPublicChatroom	
20	When user1 sends a message to user2, user1's message should only appear on user1 and user2's screen. It should not appear on user3's screen.		To be handled via Websocket;	We are mainting the receiver in the Private messages that should help void this issue
21	Messages can contain text, images (emojis), and/or links. Messages can be edited, recalled, deleted. Sending files is not supported.		17. POST /editMessage	Only Recall and Edit functionality (Design Decision). User can only recall their messages but Admin can recall anyones message
22			18. DELETE /removeMessage	
23	If message msg2 is sent after message msg1, msg2 should appear on the screen below msg1			Filtering of messages to be done before sending to frontend
24	If a user leaves a chat room, if should be clear to the other users why the user left (voluntarily left, connection closed, forced to leave). Users can report other users.		10. POST /leaveChatRoom	Need to return Notification to all users that "xyz user has left the Chat Room voluntarily" in Response to a users /leaveChatRoom
25			12.<ADMIN> POST /removeUser	Need to return Notification to all users that "xyz user has been removed from the ChatRoom" in Response to a users /removeUser
26	No JavaScript alerts should be used in the ChatApp since they will pause the app.			We won't user window.alert

---

Interfaces and Abstract Classes

We building our app back-end based on ex7, so we have controller, adapter, and models
In Controller:
We have all the get and post requests listed above to satisfy all the use cases.

In Adapter:

In Models:

We have three models: chatroom, message, and user

Abstract Class Chatroom:

Summary: Chatroom Class will be used as the abstract class for composing Private and Public ChatRoom classes. It will hold the functions which will be called on the chatApp.

IChatRoomFac will be used to make chat rooms. ChatroomDB will implement the IChatRoomFac interface and will be a Singleton instance.

1. Chatroom:
    - ChatRoom class
        - ➢ Long: id
        - ➢ String: roomName
        - ➢ Boolean: isPrivate
        - ➢ User: admin
        - ➢ Int: roomSize
        - ➢ MessageDB: messages
        - ➢ arraylist<User>: blockedUsers
        - ➢ arraylist<User>: users
        - ➢ Method:
            - ➔ Remove user: public void removeUser(userId)
            - ➔ Block user from chat room: public void addUserToBlockList(userId)
            - ➔ Unblock user from chat room: public void removeUserFromBlockList(userId)
            - ➔ Add a message in a chat room: public void addNewMessage(message)
            - ➔ Recall a message in a chat room: public void deleteMessage(messageId)
            - ➔ Check if the user join in this chatroom: public boolean userIsJoined(userId)
            - ➔ Check if the user is the admin in this chatroom: public boolean userIsAdmin(userId)
            - ➔ Add user to the chatroom: public void addUser(user)
            - ➔ getUserMessaheList(userId, timestamp)
            - ➔ getUserInfo(userId)
            - ➔ broadcastAll
            - ➔ broadcastPublic
            - ➔ broadcastSystem
            - ➔ broadcastEdit
            - ➔ broadcastDelete
            - ➔ broadcastPrivate
    - ChatRoomDB class implements IChatRoomDB interface
        - ➢ list<chatRoom>: allChatRooms
        - ➢ Method:

        - ➔ Get all the public chat room that the user do not join: public ArayList<ChatRoom> getPublicChatroom
        - ➔ Get chat room according to the chatroomId: public ChatRoom getChatRoom
        - ➔ Add a chat room to ChatRoomDB: public void addChatRoom
        - ➔ Remove a chat room from CharRoomDB: public void removeChatRoom
        - ➔ Check if the room exist: public boolean roomNameExist
    - PrivateChatRoom class extends ChatRoom class
    - PublicChatRoom class extends ChatRoom class
    - ChatRoomFac class extends IChatRoomFac interface
        - ➢ ChatRoomFac chatRoomFacIns
        - ➢ Long nextChatroomId
        - ➢ method:
            - ➔ Makes a chat room, adds it to the allChatRooms list: ChatRoom makeChatRoom

Message related classes and Interfaces:
Summary:
The abstract class AMessage will be used to create concrete classes PublicMessage, PrivateMessage, Notification, and SystemMessage. The abstract class contains general attributes and methods that all concrete classes need.
The interface IMessageFac will be used to create the concrete class MessageDB.
The MessageDB class implements the IMessageFac interface and it stores and generates different types of messages upon request.

Message:

- AMessage class
    - ➢ String: type
    - ➢ Long: id
    - ➢ String: text
    - ➢ String: content (emoji)
    - ➢ Long: senderId
    - ➢ Timestamp: sentAt
    - ➢ Boolean: isSeen
    - ➢ Method:
- MessageDB class implements IMessageDB interface
    - ➢ list<AMessage>: allMessages
    - ➢ Method:
        - ➔ Add message: public void addMessage

        - ➔ Get message: public AMessage getMessage
        - ➔ Get all messages: public ArrayList<AMessage> getAllMessages
        - ➔ Remove message: public void removeMessage

    - MessageFac class implements IMessageFac interface
        - ➢ MessageFac messageFacIns
        - ➢ Long: nextMessageId
        - ➢ method:
            - ➔ Make a message: public AMessage makeMessage
    - PrivateMessage class extends AMessage class
        - ➢ Long: receiverId
    - PublicMessage class extends AMessage class
    - SystemMessage class extends AMessage class

Notification:

- ANotification class
    - ➢ Enum Status
    - ➢ String: type
    - ➢ Long: id
    - ➢ String: text
    - ➢ String: content
    - ➢ Long: senderId
    - ➢ Status: status
    - ➢ methods:
- NotificationDB class implements INotificationDB interface
    - ➢ ArrayList<ANotification>: allNotifications
    - ➢ methods:
        - ➔ Public void addNotification
- NotificationFac class implements INotificationFac interface
    - ➢ NotificationFac: notificationFacIns
    - ➢ Long: nextNotificationId
    - ➢ methods:
        - ➔ Public status NotificationFac getInstance()
        - ➔ Public ANotification makeNotification
- InvNotification class extends ANotification class
    - ➢ Long: chatroomId
    - ➢ methods:
- SysNotification class extends ANotification class

User Related Classes and Interfaces:
Summary:
The interface IUserFac is used to create new users. The concrete class UserDB implements the interface IUserFac, and it is used to create and store users.

2. User:
    - User class
        - ➢ Long: id
        - ➢ String: name
        - ➢ String: schoolName
        - ➢ String: interests
        - ➢ String: userName
        - ➢ HashMap<ChatRoom, Timestamp> chatrooms
        - ➢ Boolean: isOnline
        - ➢ Timestamp: lastSeen
        - ➢ String: imageUrl
        - ➢ Int: age
        - ➢ Int: hateSpeechCount
        - ➢ String: password
        - ➢ NotificationDB: notificationDB
        - ➢ Methods:
            - ➔ Public void leaveChatRoom
            - ➔ Public void joinRoom
            - ➔ Public HashMap<ChatRoom, Timestamp> getChatRooms
            - ➔ Public Timestamp getJoin Time
            - ➔ Public boolean isBanned
            - ➔ Public void addNotification
            - ➔ Public boolean canBeInvited
            - ➔ Public void addHateSpeechCount
    - UserDB class implements IUserDB interface
        - ➢ ArrayList<User>: allUsers
        - ➢ UserDB: userDBIns
        - ➢ method:
            - ➔ Public boolean loginCheck
            - ➔ Public void addUser
            - ➔ Public boolean usernameExist
            - ➔ Public User getUser

        - ➔ Public long getuserIdByName
- UserFac class implements IUserFac interface
    - ➢ UserFac: userFacIns
    - ➢ UserDB: userDB
    - ➢ Int: nextUserId
    - ➢ method:
        - ➔ Public static UserFac getInstance()
        - ➔ Public User makeUser

Design Decisions which differentiate us from Others :

1. User who creates the room is the Admin.
2. Either User can delete its messages in the chatroom or Admin can delete any delete message in the chat room. Definition of Delete →”It won’t be shown the chatroom”
3. Users can “Edit” ONLY their messages. Definition of Edit →”Edit the content of existing message”
4. Hate Speech is checked by the virtual Admin who check every message for “hate speech” string and converts it into “******” and sends it across to the frontend.
5. Hate Speech count for each user if it crosses 5, user will get warning but after 10 hate speeches user will get Banned from the application.User cannot login to the application.(Even admin of a group can get banned i.e. group can become immortal).
6. Users can select their avatars from the existing list.