Project specification document by Team NullSleep

Overview

This document describes the design of Chat App, including components, explanation for interfaces, use cases, design decisions and the API spec details for Chat App. We use MVC model to implement Chat App in this assignment.

1. User Cases

1.1. Chat App Page

- The main view of the chat app webpage is divided into three columns.
- The left column is the room lists displaying all the available rooms and their status / valid operations, such as “Join”, “Owned” and “Joined”.
- The middle column is the rooms that user owned and joined. In each room. All the users in that room are displayed and the user can choose to chat with any users in the same room. Group messages and notifications are shown in each room for 5 seconds. The user can leave each room or leave all joined rooms all together.
- The right column is the user’s chats, which contain all private chats. The user can end chat or send private messages. The user can leave each chat or leave all chats together.

1.2. User Log in

- When user clicks “Login” button, a pop-up form will show and the user needs to input profile info “”User Name”, “Age”, ”Location” and “School”.
- User can click “Clear” button to clear all the inputs and type again.
- User can click “Cancel” button to cancel log in.
- After clicking “Submit” button, the pop-up form disappears. User profile is sent to backend server through Web Socket connection.

1.3. Creating a Chat Room

- Users need to create their profiles (age, location, and school) before they can join or create chat rooms.
- When user creates a chat room, this user will be the owner of that chat room. The owner can set up some restrictions about age, location, and school to allow only the users who meet the restrictions able to join that chat room.
- Each chat room must have an owner.

1.4. Joining a Chat Room

- A user can only see a chat if his profile meet the chat room requirements.
- Users can join any chat rooms listed on the left column chat room list.
- A user can join multiple chat rooms.

- A user can see the lists of all the users in a chat room after joining the room.
- If a user is already in a chat room, he/she cannot join the chat room again.

1.5. Send a Message in Chat Room

- Only the owner of chat room is able to broadcast group message to all users in that chat room.
- The group message will be shown in for 5 seconds.
- A user can send private messages to another user in the same chat room. The private message can be only seen by those two users.
- A user will be notified when the message he/she sent out has been received.
- Each message will be displayed as a separate line. The order of the messages displayed on the screen is based on the time they have been sent out.

1.6. Leaving a Chat Room

- If a user used the word “hate” in any message, he/she will be removed from all the chat rooms that he/she has joined.
- If a user leaves a chat room, a group message will be sent to notify all other users the reason why that user left (voluntarily left, connection closed, forced to leave).
- A user can leave all chat rooms all together.
- If all users left a chat room, the room will be removed and cannot be accessed any more.
- If the owner of a chat room left for any reasons, the user who joined the chat room as the second user will become the owner of that room. If the second left, the third one becomes the owner.

2. Design Decisions

- We use pop-up dialog in browser for user to Login, Create a Room, Update Profile Each chat room and each user has a unique ID to specifically identify each other.
- We will use only one page, which is the index page, together with pop-up forms (login and update profile) for ChatApp.
- We don’t use any endpoint on backend . All communications are done through WebSocket. All user interactions, like sending message, joining room, login are transferred from frontend to backend via WebSocket, data about interaction are formatted in JSON format.
- We will not support register since we don’t have a database as storage layer. So that no password needed.
- Each tab in browser is a user, so that a single user could talk to himself using different tabs in the same browser window. Closing a tab means

user stops using an account. His Web Socket connection terminates, and he is considered logged out.

- Server sends two types of information to client: 1. notify frontend view to update the room information (left and middle columns); 2. notify frontend view to update the chat information (right column)
- We use special character ‘|’ as delimiter, so that inputs can contain space.
- To view the private message from user B, user A needs to click “chat” button first. A cannot see private message from B if A does not plan to chat with B.
- Broadcast messages and notifications are shown in “My room” column only for 5 seconds. They are not shown in private chat box.

3. View

3.1. Technics to use

- We will use HTML, CSS, JavaScript and Bootstrap.

3.2. Communication with Server

Each tab represents a user. The frontend communicates with server through web sockets, using JSON format data.

3.3. Elements and User interaction

- All room would be displayed in a room list, showing information about each chat room.
- There is button under each room info for user to join.
- There is a button to create a room. A pop-up dialog will show up with forms to fill when creating room.
- User will see a list of rooms he owns.
- User will see a list of rooms he has joined.
- User will see a list of active chatting.
- There is a button in each joined room to leave.
- There is a button to leave all chat rooms.
- There is a button in each chat to leave.
- There is a button to leave all chats.

4. Controller

The controller module communicates with all the clients on the web socket.

4.1. ChatAppController

4.2. This class is responsible for handling the Heroku port, setting static file location, and initialize WebSocket on backend. **WebSocketController**

- This class handles the WebSocket communication from users. It manages a web socket on server side. It utilizes DispatcherAdapter (refer to discussion below) to handle user interactions.

5. Model

5.1. DispatcherAdapter

This class is responsible for handling web socket. When the client sends a message, WebSocketController determines the user interaction and calls methods in DispactherAdapter. This object is an observable, and its observers are users.

5.1.1. newSession:

Creates an id for the received session and adds it to the map as a key-value pair; the session is the key, and the id is the value.

5.1.2. getUserIdFromSession:

Returns the session id, a value, from the key given, a session.

5.1.3. containsSession:

Boolean return for whether a session is in our map.

5.1.4. loadUser:

Constructs a user instance; puts user instance into map with session id as key-value pair; iterates through chatrooms, applying each chatroom filter to the user, and adding each that passes; sends chatroom and chatbox responses back to client; returns this user instance.

5.1.5. loadRoom:

Construct a new room; create an id and adds id-room to map; adds room to the owner's joined room list; sends a command to all observers (all users). AddRoomCmd, which will handle adding the room and sending a chatroom response for each user; sends chatroom response back to owner client.

5.1.6. unloadUser:

Iterates through user's joined rooms; adds notification of log out to each; calls leaveRoom for each; removes this user from the map.

5.1.7. unloadRoom:

Calls removeAllUsers, a method of this room, which will clear this room's users; sends a command to all observers (all users), RemoveRoomCmd, which will handle removing the room in each user's case; removes room from the map.

5.1.8. joinRoom:

Apply room filter to user; if user passes, updates room lists for this user, and adds user as an observer to this room; sends chatroom response to client.

5.1.9. leaveRoom:

Updates joined/available rooms for this user; removes user as an observer from this room; adds notification of the departure to the room object, with a reason; sends chatroom and chatbox responses to client; if this user is the owner, calls unloadRoom.

5.1.10. voluntaryLeaveRoom:

Calls leaveRoom, but adds reason for leaving to the body.

5.1.11. ejectFromRoom:

Calls leaveRoom, but adds reason for leaving to the body.

5.1.12. sendMessage:

Check if message contains words not allowed; if so, call ejectFromRoom and return to halt method; construct new message from body info; store message in chat history of room; add message with new id to map; send chatbox response to client.

5.1.13. broadcastMessage:
Check if message has illegal words; if so, call unloadRoom and return to halt method; add notification to room object; send chatroom response to all users in this room.

5.1.14. constructAndSendResponseForUser:
Sends the chatroom response to the user.

5.1.15. ackMessage:
Sets message to “received”; sends chatbox response to the user.

5.1.16. getUsers:
Return all users of a room.

5.1.17. getNotifications:
Return all notifications of a room.

5.1.18. getChatHistory:
Return all chat history between users A and B within a room.

5.1.19. getRoomsForUser:
Constructs chatroom response for a particular client. The chatroom response is of type UserRoomsResponse.

5.1.20. getChatBoxForUser:
Constructs chatbox response for a particular client. The chatbox response is of type UserChatHistoryResponse.

5.1.21. isRelevant:
Boolean return for whether a userId is found in a particular key that contains two userIds.

5.1.22. getAnotherUserName:
Returns a name. A helper method.

5.1.23. getAnotherUserId:
Returns an id. A helper method.

5.2. Model - Package cmd

Encapsulates command classes sent to users.

5.2.1. CmdFactory:
Contains static methods for creating concrete commands.

5.2.2. AddRoomCmd:
Updates the context (user) with a newly created room, if this room is not in the user’s available list; sends chatroom response to session of context.

5.2.3. EnforceFilterCmd:
Checks if context (user) passes room filter; adds to user’s available room list, or removes room from user’s lists, based on the outcome.

5.2.4. JoinRoomCmd:
Currently does nothing (previously sent a message to the context (user) that another user had just joined a room).

5.2.5. LeaveRoomCmd:

Removes a room if the user leaving is the owner.

5.2.6. RemoveRoomCmd:

Updates the context (user) that a room has been deleted; sends chatroom response and chatbox response to session of context.

5.2.7. IUserCmd.java

The Interface IUserCmd provides an interface whenever Observable like Dispatcher or ChatRoom want to notify observers (users)
Execute is the function such that all command will execute once the command is passed to observer's update

5.3. Model - Package obj

5.3.1. ChatBox

Represents a chat between two users. It encapsulates the two users in the chat and chat message history.

5.3.2. ChatRoom

Represents a chat room. It encapsulates the owner of chat room, the restrictions of chat room (age range, school, location), all users joined this room, notifications in this room, and message history.

5.3.3. Message

Represents a single message between two users. It encapsulates the sender and receiver IDs, the room it belongs to, and a flag indicating whether the message is received.

5.3.4. User

This class represents a user, including user profile like user session, name, location, school, age. There are two lists tracking all rooms available to this user and all rooms this user has joined. We don't use another list to track all rooms owned by this user, since this can be determined by ChatRoom's owner field.

5.4. Model - Package res

Encapsulate response classes, which are sent to the client via web socket.

5.4.1. AResponse

Abstract class of response representing a message sendt from server to client.

5.4.2. RoomNotficationsResponse

Represents a notification in a room as response.

5.4.3. UserChatHistoryResponse

Represents a list of chats for a user. Frontend uses this response to update all chats for user.

5.4.4. UserRoomsResponse

Represents a list of rooms for a user. Frontend uses this response to update different type of rooms for a user (owned/joined/available)