Alex Ferguson
CS494 Portland State University
November 24, 2018

Internet Draft
Intended Status: Internet Relay Chat Application Specification
Expires: 05/24/2019

Internet Relay Chat Application

Status of this Memo

Copyright Notice

Abstract

This memo describes the communication protocol for an Internet Relay Chat application designed following a client/server architecture, implemented over web sockets.

Table of Contents

1. Introduction

This specification describes an Internet Relay Chat application that can be used to connect clients together in order to chat amongst each other. This is accomplished through the use of a client/server architecture, in which the client contains a basic GUI and business logic, and the server provides a RESTful web API with various endpoints and a server side socket implementation for bidirectional communication.

2. Basic information

Communication in the IRC application take place over the TCP/IP protocol via calls to the REST API and via Socket.io. The front end client runs via an Angular development server running on port 4200. The back end server runs on port 8080. When the client logs in to the application, the socket connection is opened and stays open until the user logs out. Via asynchronous calls to the server, implemented on the frontend via the use of

Angular Observables, users can create rooms and send messages to one another in those rooms.

All REST API post requests sent to the server take the collected information from the client and store it in the request body in JSON format. Responses to API requests from the server to the client return the requested information in JSON format.

3. Users

Upon navigating to the application, a user is presented with a login page. In order for an individual to use the application and its functionality the must have an account registered with the application. For namesake, an account is called a User.

On the server side a User contains: a unique numerical id, a username, a password, and an isActive flag. On the client side a User contains: the unique numerical id generated by Postres, the username, and the isActive flag.

   a. **Create an account**

   To create an account, the user must enter a valid username, a password, and then reenter that password to confirm it. Once these have been entered, the password is not stored as part of the model on the front end for security reasons.

   A valid username is at least 4 characters in length.
   A valid password is at least 8 characters in length.

   b. **Login**

   In order to login to the application a user must enter a username and password combination.

   - If the user doesn't exist, the server returns an error alerting the individual that a matching user could not be found.
   - If the user does exist, the password is then checked against the hashed password in the database.
     - If the password matches, the server creates a UserResource containing the id, username, and isActive flag. This is then sent back to the client.
     - If the password does not match, an error is sent to the client alerting the individual that the password "does not match our records".

   As long as the entered username and password combination is valid, the user is then routed to the lobby, in which they can use the application functionality.

c. **Logout**
   When the user is done using the application they can logout of the application by clicking their name in the upper left corner and then clicking logout. When the user clicks this button, an asynchronous call to the server sets their isActive status to false and they are routed to the login page.

d. **List active users**
   When a user navigates to the lobby, an asynchronous call is sent to the server to retrieve a list of all users who have an isActive flag set to true.

e. **List users in a room**
   When a user joins a room to begin chatting, a list of users who are currently in the room is provided on the left hand side. This list is maintained on the server

4. Rooms
   One of the core features of the application is the ability to create rooms in which users can send messages to one another. As such, the application supports basic CRUD functionality relating to rooms.
   On both the client side and the server side a room consists of: a unique numerical room id, a room name, a room description, and the id of the user who created the room to represent the room owner.

   a. **Creating a room**
      A user can create a room by filling in the necessary information listed above. An asynchronous call is made to the server, the room is created, and then sent back to the client to be listed in the view.

   b. **Listing all rooms**
      After successfully logging in, the server requests all of the rooms from the database and responds to the client, by providing that list. The client then organizes these rooms into a list that is presented to the user.

   c. **Joining a room**
      When a user wishes to join a room in which they can chat with other users in that room, they click on one of the rooms presented to them in the list of available rooms. When the user clicks on one of these rooms, they are presented with a basic chat GUI and subscribed to messages in that room.

   d. **Leaving a room**
      When a user wishes to leave a room, they can simply navigate back to the lobby from the room they are currently in by clicking Go to Lobby. When the user navigates back to the lobby from the room they were in, they are unsubscribed from messages in that room.

e. Removing a room

When a user wishes to remove a room from the database they must be the person who initially created that room. If they are the owner of that room they are allowed to remove the room by clicking on the X near the room name in the lobby component.

When a room is removed, the client sends a request to remove the room to the server. The server then attempts to remove the room.

- If the room is successfully deleted, the server sends a response to the client indicating success and the client removes the room from the list of available rooms and updates the list of rooms for all other clients.
- If there is an issue during removal of the room the server sends a response to the client indicating there was an issue during room removal.

5. Messaging

Messaging is accomplished via the use of Socket.io. This is accomplished by using the Angular socket.io-client library on the front end and socket.io on the back end. This enables full, bidirectional communication between the server and all connected clients.

On both the client and the server, messages contain the text of the message as a string, a from field as a string, and a to field as a list of numbers. The list of numbers corresponds to the ids of the rooms the message is to be sent to.

a. **Subscribing to messages**

When a user enters a room, the SocketService uses an Observable to subscribe the user to receiving messages in that room. While they remain in that room the user will continue to receive any messages that other clients send.
The Observable listens for message events that are sent from the server and responds to that event by adding the message to the list of messages for the room.

b. **Sending messages**

To send a message, the user simply types their message into an input form that is included in the room. A valid message is at least 2 characters in length. When the user types a valid message into the input field, the send button then allows them to send that message to the server.
The message is then emitted to the server by the socket and emitted by to the client by the server side socket.

c. **Broadcasting a message**

A user can choose to broadcast a message to multiple rooms of their choice. From the lobby, the user can click on broadcast, type in their message, choose the rooms that it should be sent to from the list of available rooms, and broadcast the message. Users in any of the rooms chosen will receive the message.

6. Error Handling

   Error handling is handled somewhat differently on the client and the server. However, both are fully implemented and working.

   On the server side there are a few different types of error handling. In the REST API, promises are used to asynchronously fetch information from the database. All API route handlers have a corresponding catch() block that handles any errors that may occur when fetching information. Along all routes that can be requested, useful error messages and result status codes are sent back to the client in case of any error.

   The server itself also implements two asynchronous event listeners that listen for two types of prominent errors that could cause the server to crash. These events are: uncaughtException for any exceptions that might cause the server to crash and unhandledRejection for any asynchronous promise related errors that could cause the server to crash. In these events the server logs the error to the console and continues to run without crashing.

   On the client side there are also a few different types of error handlers implemented. Since the Angular framework makes heavy use of Observables when making asynchronous requests to the server, in order to get the information out of the request you must subscribe to the observable. The Observable's .subscribe() call takes two callback functions, one for success and one for any errors that might have occurred. Each observable in the application contains an error callback that logs the error to the console so that you can see what might have gone wrong, whether on the server or the client.

   The client side socket also listens for two types of issues that might occur relating to the socket. These two events are: connect_error for listening to issues that revolve around connecting to the server side socket, and error that listens for general errors that might occur. In both of these instances, the errors are logged to the console.

7. Extra features

   Due to the scope of the project and all of the other work that I have had to work on throughout the term I wasn't able to implement any extra features on top of what I got done. What I would have liked to work on is private messaging between individual users.

8. Future feature enhancements

   Going forward with this application, there are a few different things that I would like to change in order to make the application more pleasant.

   - The GUI isn’t the cleanest in terms of usability. I would like to go through and find a really nice template to follow in order to make the user experience nicer.
   - Currently, the application keeps a list of messages on the server side and emits those messages back to the client. I think it would have been much nicer to also store messages in the database, at least for a certain period of time. This could make fetching conversations while the user is offline a useful feature.
   - I would like to rethink a bit of my overall architecture in order to make the frontend a lot cleaner. I think, given a bit more thought, the frontend architecture could also strengthen the overall user interface.
   - I would like to create profile pages for each user. On their profile page they could add information about themselves, add a profile picture, and also have notifications from other users that they can read and respond to. Not quite a social media style of notifications; rather a private messaging kind of feature.