GroupChat

Download Starter Code <../websocket-groupchat.zip>

GroupChat is an Express-based app that uses websocks for a live, multi-room chat system.

In this exercise, you'll explore good OO design for more sophisticated applications, as well as get some hands-on practice with websockets. The goal of this exercise is to read and understand the code, if you'd like, there are some additional features you can implement.

We provide working code to you; the exercise is both to explore this code and add some new features, listed at the end.

Trying Out The App

```
$ node server.js
```

Then go to http://localhost:3000/room-name <http://localhost:3000/room-name>. You can use any room name you want — so you can go to "/random" or "/humor" or "/library" or such — each of these will be a different room with different possible users.

Open a second tab on the same computer and visit the same room, and you should be able to chat with each other.

Client-Side Code

The HTML is straightforward; it shows a list of messages and has a form for a new message:

chat.html

```
<!doctype html>
<html>
<head>
  <title>GroupChat</title>
  <link rel="stylesheet" href="/css/styles.css">
</head>
<body>

<ul id="messages"></ul>

<form id="msg-form">
  <input id="m" autocomplete="off"/>
  <button>Send</button>
</form>

<script src="https://unpkg.com/jquery"></script>
<script src="/js/chat.js"></script>
```

```
</body>
</html>
```

The client-side JS makes a websocket connection to the server (injecting the roomName into the ws URL), and handles websocket events:

static/js/chat.js

```
/** Client-side of groupchat. */

const urlParts = document.URL.split("/");
const roomName = urlParts[urlParts.length - 1];
const ws = new WebSocket(`ws://localhost:3000/chat/${roomName}`);


const name = prompt("Username?");


/** called when connection opens, sends join info to server. */

ws.onopen = function(evt) {
  console.log("open", evt);

  let data = {type: "join", name: name};
  ws.send(JSON.stringify(data));
};


/** called when msg received from server; displays it. */

ws.onmessage = function(evt) {
  console.log("message", evt);

  let msg = JSON.parse(evt.data);
  let item;

  if (msg.type === "note") {
    item = $(`<li><i>${msg.text}</i></li>`);
  }

  else if (msg.type === "chat") {
    item = $(`<li><b>${msg.name}: </b>${msg.text}</li>`);
  }

  else {
    return console.error(`bad message: ${msg}`);
  }

  $('#messages').append(item);
};


/** called on error; logs it. */
```

```
ws.onerror = function (evt) {
  console.error(`err ${evt}`);
};


/** called on connection-closed; logs it. */

ws.onclose = function (evt) {
  console.log("close", evt);
};


/** send message when button pushed. */

$('form').submit(function (evt) {
  evt.preventDefault();

  let data = {type: "chat", text: $("#m").val()};
  ws.send(JSON.stringify(data));

  $('#m').val('');
});
```

Server-Side Code

> **Warning: STOP and try to not read all the code!**
>
> This app is designed with two classes that encapsulate functionality around chat rooms and chat users. Read through the express ***app.js*** and see if you can get a sense of *what* the ***ChatUser*** class does before you look at the code for it.
>
> Similarly, don't read the ***Room*** class until you've gotten a sense of what is should do.
>
> In real life, you'll often read code like this, "top-down" — looking at the overall application, and only digging into particular parts when you need to really understand how they work.

Express App

We use an npm package, ***ws-express***, which makes it easy to have Express routes which use websockets. We define one route, which handles connections to ***/chat/[roomName]***.

Otherwise, it's a fairly straightforward Express app:

- it serves static CSS/JS
- it serves a static HTML page

app.js

```
/** app for groupchat */

const express = require('express');
const app = express();

// serve stuff in static/ folder

app.use(express.static('static/'));

/** Handle websocket chat */

// allow for app.ws routes for websocket routes
const wsExpress = require('express-ws')(app);

const ChatUser = require('./ChatUser');

/** Handle a persistent connection to /chat/[roomName]
 *
 * Note that this is only called *once* per client --- not every time
 * a particular websocket chat is sent.
 *
 * `ws` becomes the socket for the client; it is specific to that visitor.
 * The `ws.send` method is how we'll send messages back to that socket.
 */

app.ws('/chat/:roomName', function(ws, req, next) {
  try {
    const user = new ChatUser(
      ws.send.bind(ws), // fn to call to message this user
      req.params.roomName // name of room for user
    );

    // register handlers for message-received, connection-closed

    ws.on('message', function(data) {
      try {
        user.handleMessage(data);
      } catch (err) {
        console.error(err);
      }
    });

    ws.on('close', function() {
      try {
        user.handleClose();
      } catch (err) {
        console.error(err);
      }
    });
  } catch (err) {
    console.error(err);
  }
});
```

```
/** serve homepage --- just static HTML
 *
 * Allow any roomName to come after homepage --- client JS will find the
 * roomname in the URL.
 *
 * */

app.get('/:roomName', function(req, res, next) {
  res.sendFile(`${__dirname}/chat.html`);
});

module.exports = app;
```

As always, there's a small ***server.js*** to start up the server.

Chat User and Chat Room Classes

We've code to encapsulate functionality of a chat user:

chatuser.js

```
/** Functionality related to chatting. */

// Room is an abstraction of a chat channel
const Room = require('./Room');

/** ChatUser is a individual connection from client -> server to chat. */

class ChatUser {
  /** make chat: store connection-device, rooom */

  constructor(send, roomName) {
    this._send = send; // "send" function for this user
    this.room = Room.get(roomName); // room user will be in
    this.name = null; // becomes the username of the visitor

    console.log(`created chat in ${this.room.name}`);
  }

  /** send msgs to this client using underlying connection-send-function */

  send(data) {
    try {
      this._send(data);
    } catch {
      // If trying to send to a user fails, ignore it
    }
  }

  /** handle joining: add to room members, announce join */

  handleJoin(name) {
```

```
    this.name = name;
    this.room.join(this);
    this.room.broadcast({
      type: 'note',
      text: `${this.name} joined "${this.room.name}".`
    });
  }

  /** handle a chat: broadcast to room. */

  handleChat(text) {
    this.room.broadcast({
      name: this.name,
      type: 'chat',
      text: text
    });
  }

  /** Handle messages from client:
   *
   * - {type: "join", name: username} : join
   * - {type: "chat", text: msg }     : chat
   */

  handleMessage(jsonData) {
    let msg = JSON.parse(jsonData);

    if (msg.type === 'join') this.handleJoin(msg.name);
    else if (msg.type === 'chat') this.handleChat(msg.text);
    else throw new Error(`bad message: ${msg.type}`);
  }

  /** Connection was closed: leave room, announce exit to others */

  handleClose() {
    this.room.leave(this);
    this.room.broadcast({
      type: 'note',
      text: `${this.name} left ${this.room.name}.`
    });
  }
}

module.exports = ChatUser;
```

We have code to encapsulate functionality of a chat room:

room.js

```
/** Chat rooms that can be joined/left/broadcast to. */

// in-memory storage of roomNames -> room

const ROOMS = new Map();
```

```
/** Room is a collection of listening members; this becomes a "chat room"
 *   where individual users can join/leave/broadcast to.
 */

class Room {
  /** get room by that name, creating if nonexistent
   *
   * This uses a programming pattern often called a "registry" ---
   * users of this class only need to .get to find a room; they don't
   * need to know about the ROOMS variable that holds the rooms. To
   * them, the Room class manages all of this stuff for them.
   **/

  static get(roomName) {
    if (!ROOMS.has(roomName)) {
      ROOMS.set(roomName, new Room(roomName));
    }

    return ROOMS.get(roomName);
  }

  /** make a new room, starting with empty set of listeners */

  constructor(roomName) {
    this.name = roomName;
    this.members = new Set();
  }

  /** member joining a room. */

  join(member) {
    this.members.add(member);
  }

  /** member leaving a room. */

  leave(member) {
    this.members.delete(member);
  }

  /** send message to all members in a room. */

  broadcast(data) {
    for (let member of this.members) {
      member.send(JSON.stringify(data));
    }
  }
}

module.exports = Room;
```

> **Note: Design Note**

> (It may be worthwhile to notice the design here: neither of these is particularly hard-coded to websockets. As long as ***ChatUser*** is given a ***send*** function it can call to send a response, they're pretty agnostic. You could re-use these in a plain-HTTP chat system or even a very different domain, like a chat-by-email system.)

Further Study

There are different features you can add to this, depending on how much time you have. They're ordered by estimated difficulty.

Get a Joke!

Add a feature where users can get a joke from the server.

If the user types ***/joke*** into the new message field, this *should not* be broadcast to all users — instead, it should return a joke to *just that user*. (You can always return the same joke, pick a random one from a list, or, if you're feeling ambitious, use the ***icanhazdadjokes.com*** API).

Want a hint?

Approach

You could do this by:

- change client-side ***chat.js*** to recognize the ***/joke*** command and send a different type of message to server, like `{type: "get-joke"}`.
- Have the ***ChatUser*** method that handles messages call out to a new method that gets a joke. It could return a response to the client, like `{type: "chat", text: "What do you call eight hobbits? A hob-byte!", name: "Server"}`

Listing Members

Add a feature where users can see a list of all the members in a room.

If the user types ***/members*** into the new message field, this *should not* be broadcast to all users — instead, it should return a list the usernames of the users in the current room, like: "In room: juanita, jenny, jeff".

Once you've thought about how to approach this, hover for a thing to think about.

Should you change ***Room***?

You could get a list of users by changing only the ***ChatUser*** class; it can get the users of a room by looking at ***this.room.members***.

Or, you could add a new method to ***Room*** that lists the members, and call that from ***ChatUser***.

Either could work, and either could be good design. Do you have a preference?

Send Private Message

Add a command like ***/priv user message*** that sends a private message that only that other user sees.

Change Your Username

Add a command like ***/name myNewName*** that change your username. You'll need to figure out how to communicate this to server. It should announce this change to the room you're in.

View The Solution <solution/>