Assignment Four: Real-time Chatroom

In the previous assignment, you built a database-backed chatroom with Node. In this assignment, you'll take it to a whole new level by using socket.io to add real-time features.

The Problem

You've attained critical acclaim for your work on the original chatroom server, but your users are starting to get antsy. They've noticed that their messages take a long time to show up, and there a few features that they feel like they're missing. It's time to take the chatroom into the 21st century with some real-time goodness.

Requirements

Starting with your existing chatroom as a template, build a real-time chatroom that does not have to poll the server for new messages. Remove any existing refresh/Ajax update logic that you have in your chatroom - it will no longer be necessary. Messages will now be delivered to your client via events fired by socket.io.

Your new chatroom product should also show, for each room, a list of the users currently in that room. The users should be removed from the list when they close their browser window.

Any other real-time features you can add would be welcome!

Getting Started

Open your `package.json` file, and add the following to your `dependencies` object:

```
"socket.io": "0.9.x"
```

Then update your dependencies with:

```
npm install
```

Now that all your dependencies are installed, open your `server.js` file, import socket.io, and set it up with your Express app. This will require a few modifications to your existing setup code, most notably you need to manually create an HTTP server, and listen on that instead of the app:

```
var http = require('http'); // this is new
var express = require('express');
var app = express();
var server = http.createServer(app); // this is new

// add socket.io
var io = require('socket.io').listen(server);

// your server code here

// changed from *app*.listen(8080);
server.listen(8080);
```

Then, on each page of your site that needs to communicate with the server in real-time (probably just `room.html`, if that's what you called it), you need to add the socket.io client-side script to the `head`:

```
<script src="/socket.io/socket.io.js"></script>
<script type="text/javascript">
var socket = io.connect();

// the rest of your client-side scripting here...
</script>
```

Note that this is automatically served by socket.io, you don't need to copy this script file anywhere.

— Using socket.io —

Socket.io works like a normal `EventEmitter` in Node, except that it can emit events back and forth between a client and a server. Come up with a reasonable set of events, and then emit them from either the client or the server to indicate state changes.

One possible set of events could be:

- `join(roomName, nickname)`: sent from the client when joining a new room
- `nickname(nickname)`: sent from the client when the nickname changes after initial join
- `message(message)`: sent from the client when a new message is sent

- `message(nickname, message, time)`: sent from the server when a new message for the room is received
- `membershipChanged(members)`: sent from the server when a user joins, leaves, or changes their nickname (the `members` parameter should be a complete list of users)

Note that we don't include a `leave` event because the user closing their browser window (the `disconnect` event) works as an implicit leave.

To send an event from server to client, you first obtain an individual socket object, or a collection of socket objects (like `io.sockets` for all connected users), and then call the `.emit(...)` method. The first parameter is the name of the message, and any subsequent parameters are sent as function arguments:

```
// send to one user
var socket = ...; // obtained through some trickery
socket.emit('myCoolEvent', param1, param2);

// send to all users
io.sockets.emit('myOtherEvent', paramA, paramB, paramC);
```

On the client-side, you obtain your socket by calling `io.connect()` (generally with no parameters) **once** at the top of your script. You can then assign event handlers in the normal Node-y way:

```
var socket = io.connect();

// lots of app code

socket.on('myCoolEvent', function(param1, param2){
    // this code is run when the server emits an event to us
});
```

Sending events in reverse works the exact same way: setup event handlers on the server on individual sockets with `socket.on(...)`, and then call `socket.emit(...)` from the client side.

Generally speaking, if the client will be sending events to the server, you register your handlers for those events inside the `io.sockets.on('connection', ...)` handler, since it's the first time you get to interact with the new socket object.

More detailed examples can be found on the socket.io "How to Use" page.

— Wiring Things Up —

The first order of business is to setup a server-side handler for what happens when a new user connects. One strategy is to keep a global table of users somewhere along with their nicknames.

```
var users = [];
io.sockets.on('connection', function(socket){
    // handle a newly-connected socket
    users.push(socket);

    // add event handlers for this user
    socket.on('disconnect', function(){
        var idx = users.indexOf(socket);
        users.splice(idx, 1);
    });
});
```

This will be insufficient, however, because we actually want to keep track of users in different *rooms*, not users in general. It turns out that socket.io has a feature called "rooms" which is perfect for this. It also does most of the bookkeeping on our behalf, so we don't need to maintain our own users table. Awesome.

What we'll do is create an event called "join" that a user will call immediately after joining a chatroom. We'll use this as an indication that the user should be joined into the room. Our new connection handler looks like this:

```
io.sockets.on('connection', function(socket){
    // clients emit this when they join new rooms
    socket.on('join', function(roomName, nickname){
        socket.join(roomName); // this is a socket.io method
        socket.nickname = nickname; // yay JavaScript! see below
    });

    // this gets emitted if a user changes their nickname
    socket.on('nickname', function(nickname){
        socket.nickname = nickname;
    });

    // the client emits this when they want to send a message
    socket.on('message', function(message){
        // process an incoming message
        // note that you somehow need to determine what room this is in
        // io.sockets.manager.roomClients[socket.id] may be of some help, or you
        // could consider adding another custom property to the socket object
    });

    // the client disconnected/closed their browser window
    socket.on('disconnect', function(){
        // what might we need to do here? hmmmm
    });
});
```

IMPORTANT NOTE: this code exploits a very unique feature of JavaScript. Note that we just assign nicknames to socket objects, even though socket.io does not have a nickname property for sockets. This works, though, because objects in JavaScript are really just hash tables under-the-hood. The moral of the story is, you can assign any property to any object, anywhere, in JavaScript, and you can then go back and get that property later.

On the client side, we might have something like this:

```
// inside <script type="text/javascript">
var socket = io.connect();

// fired when the page has loaded
window.addEventListener('load', function(){
    // handle incoming messages
    socket.on('message', function(nickname, message, time){
        // display a newly-arrived message
    });

    // handle room membership changes
    socket.on('membershipChanged', function(members){
        // display the new member list
    });

    // get the nickname
    var nickname = prompt('Enter a nickname:');

    // join the room
    socket.emit('join', meta('roomName'), nickname);
}, false);
```

Whenever a new message comes in on the server side, after determining the room name, you can send it out to all of the users in the room like so:

```
io.sockets.in(roomName).emit('message', nickname, message, time);
```

Finally, whenever someone leaves a room, joins a room, or changes their nickname, you'll want to send a membership update to everyone in the room. This can be accomplished with something like this.

```
function broadcastMembership(roomName) {
    // fetch all sockets in a room
    var sockets = io.sockets.clients(roomName);

    // pull the nicknames out of the socket objects using array.map(...)
    var nicknames = sockets.map(function(socket){
        return socket.nickname;
    });

    // send them out
    io.sockets.in(roomName).emit('membershipChanged', nicknames);
}

// e.g.
broadcastMembership('ABC123');
```

There will undoubtedly be some remaining rough edges (when a new user joins, do they immediately see a list of other users in the room? do they see past messages?), but there isn't much more to it than that. **Make sure that you preserve your existing database code**, we still want messages to be logged as they're sent across the wire.

— Other Niceties —

Any value you can add to your user experience is always good for brownie points. Think about what real-time features you can add with socket.io that would enhance your users' experience when chatting, and feel free to add them.

A few ideas that could be worth some extra credit:

- Live typing notifications ("Justin is typing...") or potentially even live character sending. The latter would be tricky, both from an implementation standpoint, and from a user experience standpoint. If you do it, find a way to do it that isn't annoying!
- Idle time detection and notification ("Justin is away"). You could track when the user switches windows/tabs with the `blur` event, and possibly track mouse events to see if they're on the window but not touching anything for awhile.
- Direct person-to-person messaging. This can be within the context of a room (so two users in a room can carry on a private side conversation) - you **don't** have to support communication between users in different rooms. The latter is not technically more difficult, but would require some thought in terms of an appropriate UI for finding other users.
- A user can already be in multiple chat rooms if they open multiple tabs/browser windows (they're treated as distinct sockets), but what if I wanted to be able to have multiple chat rooms in a single window? This is a large undertaking, both from a UI perspective and an engineering perspective.

We'll also probably be willing to give extra credit for any novel features you come up with on your own, particularly if they take advantage of the real-time power of socket.io. Be creative!

— Handing In —

Before handing in your project, make sure you've checked to make sure you've listed any additional dependencies in your `package.json` file, and you've filled in your `README.md` file (you can use any format you like for your README, but Markdown is highly recommended).

To hand in your project, from your project directory, run:

```
cs132_handin realtime
```

That's it!