# Chat Room Server

Tyson Kamp

## Summary

This is a sample code submission, designed to implement the criteria contained in the Appendix. The code ended up much more monolithic and suboptimal (in terms of structure) than I would like but given time and unfamiliarity with js I stopped at this point. The total time spent on this was about 15 hours.

Text written in Courier New is intended to be text used interacting with the server.

## Usage

### Installed packages

npm install es-module-loader --save-dev

### How to Start

In the application folder there is a TDKEnv.bsh file. That file needs to be sourced from that folder (as it sets the application folder environment variable that the tests use). Each window / prompt will need those env vars set. Knowing this, start the TDKChatServer.js in one window, and use the commands below to add users and then run the TDKClient.js or telnet to use the chat room.

### Login

Use login uname,pwd to authenticate and issue other command.

### Users

Shows a list of users currently authenticated and connected.

### AddUser

Use adduser uname,pwd to add a user. Note that only the admin account can add or remove users.

### Me

Use me This is my message to send "This is my message" to the chat room.

### Quit

Use this to leave the chatroom.

## Architecture Considerations

### Scalability

This implementation should scale to about 1k concurrent users with one server process running based on what is available online regarding the number of simultaneous connections to a socket server in this context. If we wanted more than 1k users (in one chat room) then we would have to start using more than one process and persistence for the list of users and the chat history, and load balance incoming requests across chat room server processes.

### Durability/Persistence

Due to time constraints and lack of familiarity with the MEAN stack I was unable to implement persistence. The requirements for this app are really basic: a brief conversation history and username/pwds is all that's needed. A variety of data stores (sql or nosql) could be used. Practically speaking any of them could handle millions of users, once considering billions then I'd consider something else but again it depends on the specific numbers. I wouldn't rule out any of the db vendors or technologies without knowing more about the requirements.

### Security

#### Passwords (Data at rest)

Care was taken to not store **any** data that could be read plain text (in the application). No plain text passwords are stored or ever should be, only hashed strings that were once passwords. Passwords are hashed with MD5, and attempted login credentials are hashed and compared with sored hashed values. No salt is added, but ought to be in a more serious implementation. In this sample the admin password and hashed password are store in config files. In prod you would just use a different password and take a lot of care to make sure the code that computes the hashed value of the password is secured so an adversary doesn't know the scheme employed.

#### Configuration (Data at rest)

The app was coded preferring to load information from the environment. The goal is to use environments that are created and configured dynamically, and destroyed on a regular basis. Config information ought to be in a config server (rather than in config files, or even worse in code). The idea is to have config information stored securely in one location, and not in files spread around the network. For this demo, the admin account and hashed password are in scripts that set environment variables. With more time this demo should be altered so that

#### Socket communication (Data in transit)

The requirements were for a socket connection and a telnet client. However SSL would be required from a security perspective as passwords are sent plain text over sockets (and secure sockets for that matter but the channel is encrypted).

#### Logging (Data at rest)

Although it's best practice to never log any user ids (and for sure passwords), there may be some console logging of messages coming in for debug purposes. For proper security and sustainment there must be some logging/tracing but must omit user ids, passwords, IP addresses (to the extent possible) and things that might indicate which accounts are performing activities.

#### Miscellaneous

It would be a good idea to enforce a limit on message length to avoid buffer overflow exploits.

## Testing

#### JSLint

Running JSLint, only complaints were for whitespace. Then put NPM coding standards into the code and had many complaints, mostly about missing semicolons, which were removed due to coding standards. Those two can fight it out!

#### Mocha

Given more time I'd like to have refactored the code into smaller pieces, testable without the socket connection (just calling module exported functions directly and testing).

#### Command Line Tests

There is a js client that I used called TDKClient.js that I find easier to use than the telnet client, although telnet can also be used. There are a few scripts in the test folder to test logging in with the admin account, creating accounts, and sending a portion of "The Stranger" (Camus) as "me" messages. These are the beginning of a test battery that should be developed.

Steps to do basic testing:

* Open a shell window, navigate to the folder where the code is downloaded to (something/TDKChatRoom)
* Run “. ./TDKEnv.bsh”
* Cd tests
* Run “./addUserFentinman.bsh”, and repeat for “Sarah” and “Tyson”. Each will check that the expected output came back from the server.
* Run “./runAll.bsh” to run all of the test scripts.

Appendix

1. Implement a chat system where someone can telnet into the server to a port you've defined.
2. The node net library can only be used to implement sockets. Do not use [engine.io/socket.io/etc](http://engine.io/socket.io/etc).
3. Any amount of users can connect to the server (assume at most 5 users for this project).
4. The server only hosts a single, global chat room.
5. When the user first connects, they should be prompted for a username with some simple validation rules that should re-prompt the user if the validation fails.
6. The chat room should retain at least the last 10 messages sent to it. Newly connected users should be able to see the last 10 messages as well.
7. The chat room should at least show the user and the message that was sent (eg <User> This is my message).
8. Implement three commands
   1. /users (The server privately lists to the requester the users in the chat)
   2. /me <action> (eg /me kicks the can) would cause the server to broadcast to the chat something like \* User kicks the can
   3. /quit - disconnects the user
9. Include some unit tests to test the functionality
10. Include a readme file on how to use / setup and any other notes
11. The test system will be on the latest version of OS X’s terminal using the telnet command