Simple WebChat

This project will teach you how to build and deploy a simple webchatting program using NodeJs

Getting Started

1. Required Software
2. Clone The Repository
3. Starting The Project
4. Required Modules
5. Test Code

Requrired Software

- Node 12.16
- Visual Studio Code (Recommended, but any text editor will work)
- Chrome (Recommended Web Browser)
- Git (Optional, NOT RECOMENDED FOR NOVICE USERS)

For this guide I will exclusively be using Visual Studio Code. If you would like to use something else you will have to translate my steps into your editor.

Please download and install all required software before continuing, all install options can be left to default. After this process is finished you can move onto the next steps.

Clone The Repository (OPTIONAL)

If you would like to compare your code with mine follow this process. If not you are welcome to this section. While I recommend writing all the code yourself, downloading my source code may help you solve issues in your code and may help you understand the code better.

To download my code simply click the `Clone Or Download` Button on the Github Page then select the `Download Zip` option. This will download the zip file to your computer. To use the code simply unzip the file to the desired location.

(**ADVANCED**) Alternatively if you have installed the Git software, you can use that to download the repository. Do this by opening up a terminal window and navigating to a folder where you would like to store the code in. Type `git clone #`. This will download the files to your computer and you can start editing the code.

Starting The Project

Now that you have all the required software it is time to start the project.

If this is your first time using a Code Editor I strongly recommend reading the basic documentation of VSCode. You can find that here. I will do my best to keep this a beginner friendly as possible but I do expect you to know some basic file termonology and be able to navigate around the workspace.

Start by opening Visual Studio Code. From the Menu select `File --> Open Folder`. Then navigate to where you would like to store the project, create a new folder, and open that new folder.

Now you will have a new workspace and we can begin writing code!

Lets start by Opening a terminal by going to the Menu and selecting `Terminal --> New Terminal`. A terminal window should appear at the bottom of Visual Studio.

Inside this terminal type `npm init`. Follow the prompts and fill it out as you see fit. Npm is a tool provided with NodeJS and allows you to create node projects, add modules to it, and manage your project. We will use a few of these features later on.

You should see a file called `package.json` appear in the file explorer on the left side of the screen. If you do not see the `package.json` file make sure you have the file explorer visible (see below). If you have the file explorer open and do not see a `package.json` file, then an error has occurred or Node was not installed correctly. Make sure NodeJS is properly installed and try again. **NOTE**: you may need to restart Visual Studio or your computer after installing Node, sometimes the terminal interface does not work properly until the system is restarted, while rare it does happen.

If you do not have the file browser visible, select the icon that looks like 2 pieces of paper on the top left of the window. This in turn will open the file explorer, which shows all the files in the project.

Test Code

Ok, now that we have started the project its time to write some code!

Lets begin by writing a simple test to make sure everything is working.

Start by creating a new file called `index.js`. You can do this by right clicking on the file explorer and selecting `New File`.

After doing this the file will open and you can start writing code in it. For simplicity sake we will write a simple *Hello World* program to make sure Node is working and get you comfortable writing javascript.

In the `index.js` file type

```
console.log("Hello World!");
```

This code will output the text "Hello World" to the terminal when run.

Now lets run this code to make sure it runs. In the terminal type `node index.js`, you should see `Hello World!` appear in the console.

If you do not see an output there was a problem. Double check the code spelling and make sure node is working correctly.

Required Packages

Now that we know Node is working, we can begin the project. First we must download the packages needed to run the webchat. These packages are libraries of pre-written code that will allow us to complete some of

our tasks.

In this project we will be using Express and Websockets. To install these two packages type the following lines in the terminal.

- `npm install express`
- `npm install ws`
- `npm install express-session`
- `npm install uuid`

Now the packages should be installed and ready for use.

Begining the Project

Now that all the prerequisite are complete we can finally begin writing the code.

This project will focus on both the front end and backend. The Frontend code is code that encompasses the webpage and is what the end user will see. The backend code will be the code that allows the chat to work. The frontend will be done in HTML and the backend will be done in Javascript. We will get more into frontend and backend later, for now all you need to understand is the frontend is everything the user will see and the backend is the code that makes everything work.

The Backend

All our backend code will be done in the `index.js`, so lets begin by opening up the file.

We can remove the `console.log("Hello World")` test code since we do not need that.

Ok now we can finally begin the project. We will start by initializing the required modules. The following code below will go over initilizing the modules and setting the variables

```
// initiate the modules we downloaded
const session = require('express-session'); // module for session parser
const express = require('express'); // module for express router
const http = require('http'); // module for http server
const uuid = require('uuid'); // module for uuid (we will use this to create
unique user ids)
const WebSocket = require('ws'); // module for websocket

// Session stores user information, so that.
const sessionParser = session({
    saveUninitialized: false,
    secret: 'superSecretString',
    resave: false
});

// creates the backend for the app and tells it to use certian modules
const app = express(); // init express router and assign it to variable app
app.use(express.static('public')); // tell app to use index.html in the public
folder (we will go over this in the frontend section)
app.use(sessionParser); // tells app to use the session parser
```

```
// Creates a list of users. A Map is a data type that stores data as key, pair values. If you are interested in learning more about this data type use the link below.
// https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map
const userList = new Map();

// creates the server and sets it to use the express router we set up earlier.
const server = http.createServer(app);

// starts the websocket listener. it will recieve connections on the http server address
const wss = new WebSocket.Server({ clientTracking: false, noServer: true });
```

This code will be begin initializing all the variables that we will be using later.

Now we will tell the http server how to handle requests from the webbrowser. These next steps will go over adding users to the userList and assigning them a websocket so the server can directly communicate with the client.

An important things to note. Javascript strings can be created in 3 different ways. You can use double quotes `""`, double apostrophes `''`, or double accents ``` `` ```. When you use double accents you are allowed to add varables directly into the string. When run the variable will be automatically added to the string.

Example

```
var x = 12;
console.log(`X is equal to ${x}`)
```

```
X is equal to 12
```

Keep this in mind when following along with the code below, because some of the console logs use the double accents to easily format the output to use variables.

```
// This code handles update requests from the client to server.
// It assigns a user to a unique id and begins the websocket connection
server.on('upgrade', function (request, socket, head) {
    console.log('Parsing session from request...');

    // if the request is from a new user (ie new person connects to the website)
    // assign a user to a unique id and add the user to the userlist
    sessionParser(request, {}, () => {
        if (!request.session.userId) {
            // assigns a unique user id for a new user
            const id = uuid.v4();

            // log user connected to server and assign the userid to the session
```

```
created
            console.log(`Updating session for user ${id}`);
            request.session.userId = id;
        }

        // log that the parsing is done and the user is valid
        console.log('Session is parsed!');

        // send a message back to the client that tells the client it is now ok to
talk to the server
        wss.handleUpgrade(request, socket, head, function (ws) {
            wss.emit('connection', ws, request);
        });
    });
});
```

The next section of code will handle messages from the user to the server.

```
// This code handles the websocket connection between the user and server
wss.on('connection', function (ws, request) {
    // get the user ID from the current user session and assign it to a variable
    const userId = request.session.userId;

    // assign the websocket associated with the user to the userid. This is
important to tracking and sending messages to the correct user.
    userList.set(userId, ws);

    // code to run when the websocket receives a message from the user
    ws.on('message', function (message) {
        // log message to console. Not necessary but useful for debugging
        console.log(`Received message ${message} from user ${userId}`);

        // send the message to all users connected to the app.
        // if you would like more information how the forEach function works use
this link.
        // https://developer.mozilla.org/en-
US/docs/Web/JavaScript/Reference/Global_Objects/Map/forEach
        userList.forEach(function (client) {
            // send message to the client
            client.send(message);
        });
    });

    // on socket close. delete user from map.
    ws.on('close', function () {
        userList.delete(userId);
    });
});
```

Now that all the initialization and logic is done. We can begin the server and move onto the front end.

```
// Start the server.
server.listen(8080, function () {
    console.log('Listening on http://localhost:8080');
});
```

Useful Links

Here are some useful links to better understand some of the tools we are using in this project.

Programing Documentation

- Node.js Documentation (good luck...)
- JSON Files
- Package.json
- Routing in Express

Tool Documentation

- VSCode