Chat app tutorial

Using HTML, CSS, JS and Node.js

PRECISIT

Introduction

The aim of this exercise is to give an understanding of the basic concepts of web programming and an introduction to some slightly more advanced concepts such as asynchronous programming and web sockets. We will build a simple chat application using modern web technologies that lets users communicate in real-time. When a user sends a message this should be sent to a server that distributes the message to all other connected users.

The chat client will be built using HTML, CSS and JavaScript. The chat server will be built using Node.js and some common Node.js frameworks such as Express.js and Socket.io.

Some useful links:

- List of HTML elements: https://developer.mozilla.org/en-US/docs/Web/HTML/Element
- List of CSS properties: https://www.w3schools.com/cssref/
- How to use Chrome DevTools: https://developer.chrome.com/devtools
- Socket.io cheat sheet: https://socket.io/docs/emit-cheatsheet/

Part 1: Content and styling using HTML and CSS (frontend)

First of all, let's create a web page that resembles a simple chat application. For now we won't worry about functionality, just the look and feel of the application.

1. Create a project directory on your computer with an HTML file called *index.html* and two sub-folders called *css* and *js*.

2. Open the HTML file and create a HTML shell.

    a. If using Atom, type "html" and press the tab key to generate a shell.

    b. Otherwise, get the template from here: http://htmlshell.com/

    c. You can view your webpage by opening the .html file in your browser.

3. Download normalize.css from https://necolas.github.io/normalize.css/, add it to your *css* folder and include it in the `<head>` section of your *index.html* using a `<link>` element.

4. Add the necessary HTML code for your chat application in the `<body>` section:

    a. Create a heading containing the name of your application.

    b. Create an unordered list with a few list items containing hard-coded chat messages. Each message in the message list should contain the time it was created and a message text. For example:

    ```
    19:48:00 - This is my first chat message
    19:48:05 - This is my second chat message
    ```

    Later we will remove these and replace them with actual chat messages.

    c. Create a simple form containing a text input and a button.

5. Add a personal touch to your application using some CSS styles.

    a. Create a stylesheet (.css file) in your *css* folder and include it in the `<head>` section, after normalize.css.

    b. Add styling to your web page using CSS classes and the HTML `class` attribute.

c. Feel free to make it look however you like using colors, fonts, borders and so on! There exist way too many CSS properties to name them all here. Have a look at the CSS reference linked above to get started.

6. Add some wrapper elements to your HTML code to be able to center the content of your web page. For example you could use the `<header>` and `<main>` elements along with some CSS styling.

7. Make the page responsive (mobile friendly):

    a. Add `<meta name="viewport" content="width=device-width, initial-scale=1.0">` to the `<head>` section.

    b. Play around with CSS properties such as `width` and `max-width` to make the content scale appropriately on different screen sizes.

    c. Test it on different screen sizes by using Chrome DevTools or simply resizing your browser window.

Summary: We now have a simple HTML web page with CSS styling. The page contains a list of chat messages and a form for sending new messages. The page is responsive and works well in desktop as well as mobile browsers.

Part 2: Functionality using JavaScript (frontend)

Our web page is not very dynamic so far. When clicking the send button for example, nothing happens. To add some interactive functionality to the web page, e.g. adding a new message to the chat when the send button is clicked, we will be using JavaScript.

1. Create a JavaScript (.js) file in your *js* folder and include it in the `<head>` section using a `<script>` element. Add the attribute `defer` to your `<script>` element to make sure the script is not executed until the whole page has finished loading.

2. Test that the script is working by adding a `console.log` statement and checking the console in your browser (e.g. using Chrome DevTools).

3. Create a `function` called `sendMessage` that takes the value of the text input and logs it in the console using `console.log`.

    a. Create the function `sendMessage`.

    b. Add a statement to the function that gets the current value of the text input element `document.getElementById()` and stores it in a variable.

    c. Add a `console.log` statement that outputs the value of the variable.

4. Attach/bind the function `sendMessage` to the `click` event of the button using the method `addEventListener()`.

5. Empty the value of the text input after outputting it to the console.

6. Add the current time to the beginning of each sent message.

    a. Download moment.js from https://momentjs.com/, add it to your *js* folder and include it in the `<head>` section, before your custom .js file. This is a library that makes it easy to work with dates and times in JavaScript. It will allow you to output the current time for each chat message.

    b. In the `sendMessage` function, add a variable containing the current time (when the button was clicked) using `moment().format()`.

    c. Combine the string variable containing the current time with the string variable containing the text message and output this using `console.log`.

7. Instead of outputting the message to the console, let's add it to our message list (`<ul>` element).

    a. In the `sendMessage` function, create a `<li>` element using `document.createElement()`. This will create a `<li>` element node that we later can insert into the message list.

    b. Also create a text node using `document.createTextNode()`. This node should contain the message (current time + input text).

    c. Append the text node to the `<li>` element node using `appendChild()`.

    d. Append the `<li>` element node to the message list element by using `appendChild()` again.

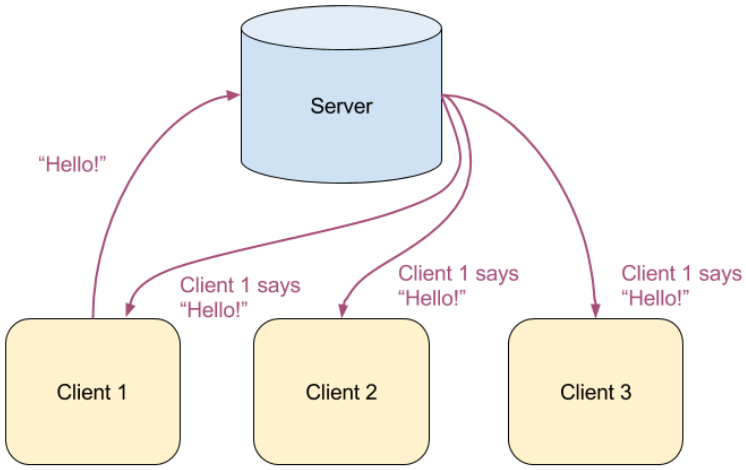
8. Add an if statement to beginning of `sendMessage` to only add the new message to the message list if the message is not empty/has a value.

    a. Use the length property of the text input value.

Summary: When clicking the send button a click event handler will trigger the function sendMessage. This function takes the value from the text input along with the current time (using moment.js) and stores it in a variable. The function then creates a list element node and a text node containing the message (current time + input text) and appends it to the message list. When the message has been added to the message list the value of the text input is cleared.

Part 3: Setting up the chat server using Node.js (backend)

We now have a chat application that can send new messages, but these messages are only visible to the person sending the message. Why? Because we don't distribute the message to other clients connected to the chat. To do that we need to create a server that can receive new messages and broadcast them to all connected clients. Our aim is to have a client-server architecture like this:



To implement this architecture we will create a web server using Node.js. The server will serve all the required frontend files (the HTML, CSS and JavaScript we created in part 1 and 2) to people connecting to the server. We will also use Socket.io to allow the server to communicate with connected clients in real-time. Before proceeding with the following steps you need to make sure you have Node.js installed on your computer.

1. Create a new Node.js application.

    a. Open a new terminal window (Mac OSX) or command prompt (Windows) and browse to the directory where you want your server to live.

    b. Run the command **`npm init`** and follow the instructions (the default options will work fine for us). This will create a *package.json* file containing information about our application.

    c. Run the command **`npm install --save express socket.io moment path`** This will install the third party libraries we need to build our chat server, which is Express.js, Socket.io and Moment.js. By using the **`--save`** flag these libraries will be saved as dependencies in the *package.json* file.

d. Create a file called *index.js* in the server directory.

2. Create a web server by adding the following boilerplate code to *index.js*:

```
var moment = require('moment');
var path = require('path');
var express = require('express');
var app = express();
var http = require('http').Server(app);
var io = require('socket.io')(http);
var port = 3001;

// the rest of the code goes here

http.listen(port, function(){
  console.log('Server started on port: ' + port);
});
```

   This code will import our installed libraries and start a HTTP web server on port 3001. This, or something similar, is used in almost every Node.js application.

3. Run the command **`node index.js`** to start the server. Check that it works by going to the address http://localhost:3001 in your browser. You should see the message *`Cannot GET /`* in your browser.

   The reason we get this error is because we haven't configured our server to serve any content to the browser yet. In step 4 we will fix this.

   Note: When you change the code in *index.js* you have to shutdown the server (ctrl + C) and run **`node index.js`** again for the changes to take effect.

4. Make it so all your previously created frontend files are served by the backend when accessing http://localhost:3001 in your browser.

   a. Move or copy your whole frontend project directory into your Node.js server directory and rename it *public*.

   b. Add the following line of code to *index.js:*

```
app.use(express.static(path.join(__dirname, 'public')));
```

      This will make the *public* folder accessible from your browser. You can read more

about serving static files with Express.js here.

c. Restart your server and try accessing http://localhost:3001 again. You should now see the frontend working as usual! The difference from before is that we can access it from our browser by going to the server address *localhost* instead of using the whole .html file path like we did before.

5. Add functionality for receiving and forwarding messages from/to clients in real-time using Socket.io.

   a. Create a handler for new socket connections using `io.on('connection', fn)` where `fn` is a function that will contain all the remaining code for the backend. `fn` will receive a `socket` instance as an input parameter which can be used to send and receive messages to/from the connected client.

      Use `console.log` to output a message in the terminal window/command prompt when a new client connects. It should look something like this:

      ```
      io.on('connection', function(socket) {
        console.log('New client connected');
        // all remaining code goes here...
      });
      ```

   b. Create a handler to output a "Client disconnected" message when a client disconnects (closes the web page). Use `socket.on('disconnect', fn)` where `fn` is a function that outputs a text using `console.log`.

   c. Send a welcome message to each new client when they connect to the server. Use `socket.emit('newMessage', message)` where `'newMessage'` is what we choose the name the event for sending messages and `message` is an object containing the current time and a text message. For example:

      ```
      socket.emit('newMessage', {
        time: moment().format('HH:mm:ss'),
        text: 'Welcome to the chat!'
      });
      ```

   d. Create a handler for receiving new messages from the client and forwarding it to all connected clients. Use `socket.on('newMessage', fn)` to receive new messages from the client.

      `fn` will receive a string as an input parameter containing the message text. Output this using `console.log`.

```
socket.on('newMessage', function(messageText) {
  console.log(messageText);
});
```

e. Use the string `messageText` to create a new message object containing this text and the current time, just like we did for the welcome message. For example we could change the code above to this:

```
socket.on('newMessage', function(messageText) {
  var message = {
    time: moment().format('HH:mm:ss'),
    text: messageText
  };
  console.log(message);
});
```

f. Forward the `message` object to all connected clients using `io.sockets.emit('newMessage', message)`. For example:

```
socket.on('newMessage', function(messageText) {
  var message = {
    time: moment().format('HH:mm:ss'),
    text: messageText
  };
  console.log(message);
  io.sockets.emit('newMessage', message);
});
```

Our chat server is now complete!

Summary: We have created a chat server using Node.js and Socket.io. When new clients connect to the server we serve them all the HTML, CSS and JavaScript content for the app and then send them a welcome message. When the server receives a new message from a client it adds the current time to the message and forwards it to all connected clients.

Part 4: Connecting client to server (frontend)

We now have a functioning chat server that can receive messages from chat clients and forward it to all other clients. The only thing still remaining is to connect the chat server (backend) to our frontend. We will do this by using Socket.io again, but in the frontend this time.

1. Add a `<script>` element to the `<head>` section of your *index.html* (before *main.js*) with the source *"socket.io/socket.io.js"*. Note that we don't actually have this *socket.io* directory anywhere in our application, it's served automatically by the Node.js server when we're using Socket.io.

   Note: If you have skipped part 3 in this tutorial you can download *socket.io.js* from here (ctrl/cmd + S). Add it to your *js* folder and include it in the `<head>` section, before your *main.js* file.

2. Connect to the chat server by adding

   ```
   var socket = io.connect('http://localhost:3001');
   ```

   to the top of your *main.js* file. The address and port number may vary depending on how you configured your chat server in part 3.

3. We need to break down our `sendMessage` function into two separate functions; one for actually **sending** the message to the server and one for **receiving** new messages from the server.

   a. Create a function called `receiveMessage` which takes an input parameter called `message`. `message` will be an object containing the properties `time` and `text` that is sent from the server.

      i. Move the code used for creating/appending the message element to the `receiveMessage` function.

      ii. Remove the code for getting current time, this is now handled by the server. We get the current time for each message by accessing the property `message.time`.

      iii. Use the property `message.text` instead of the text input value to get the text for the new message.

iv. Attach/bind the function `receiveMessage` to the socket event `newMessage` using `socket.on('newMessage', fn)`.

b. Remove *moment.js* from your *index.html* file and also from the *js* directory, since we're not using it anymore.

c. Add code to the `sendMessage` function for sending the current text input value to the server using `socket.emit('newMessage', valueFromTextInput)`.

Done! Try connecting some clients to the chat using multiple browser windows.

Summary: By using Socket.io in the frontend and the backend we have are now able to send messages to and receive messages from all connected clients in our chat app.

When the user enters the web page he/she is connected to the chat server using Socket.io.

When clicking the send button a click event handler will trigger the function sendMessage. This function sends the value of the text input to the chat server. When the message has been sent the value of the text input is cleared.

When a new chat message is received from the server the newMessage event handler will trigger the function receiveMessage. This function creates a list element node and a text node containing the message (current time + message text) and appends it to the message list.

Further improvements

- Improved styling:
  - CSS transitions or animations.
  - Fixed height of the message list so that the message form stays in a fixed position at the bottom.
- Add functionality for selecting name when connecting to the chat.
  - Show the name of the sender when receiving a new message.
  - Show a list of connected users.
- Anything else you can think of to make your chat app more awesome!