Extra Notes - NodeJS - Node.js & Web Sockets

- Dr Nick Hayward

A collection of notes on Node.js and Socket.io, including a basic example app with iterative development.

Contents

Intro This app uses Node.js, Express, Socket.io &c. to create an example messaging app, which showcases usage of Socket.io.

Heroku app URL,

- https://spire-sockets.herokuapp.com/

v0.1 - project setup Initial project setup is as follows,

- create project directory & cd

```
mkdir node-socket-io
cd node-socket-io
```

- initialise a new Node.js based project

```
npm init
```

and answer the basic questions in the terminal.

- initialise a new Git repository for the project

```
git init
```

- install Express using NPM

```
npm i express --save
```

v0.1 - initial structure We can start our app with the following structure,

```
|-- node-express-starter
    |__ .git
    |__ node_modules
    |__ package.json
    |__ public
        |__ index.html
    |__ server
        |__ server.js
```

`server` directory for the Express based server logic, and `public` for the initial static files.

v0.1 - server setup Add initial `server.js` file and logic, e.g.

```
// require node module 'path' - built-in module
const path = require('path');
// require express module
const express = require('express');

// define path to static dir public
const publicDir = path.join(__dirname, '../public');
// define variable to call express methods
var app = express();

// configure express static middleware
app.use(express.static(publicDir));

// start server on port 3030 - add callback function
app.listen(3030, () => {
  console.log('server running on port 3030');
});
```

v0.1 - static files This basic server will allow us to server static files from the `public` directory, e.g. a starter `index.html` file

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>websockets starter</title>
  </head>
  <body>
    <h3>WebSockets Starter</h3>
  </body>
</html>
```

Then, we can start and test the new server

```
node server/server.js
```

which will be available at the specified port, e.g. https://localhost:3030

v0.1 - add Heroku config Update `server.js` to define port for local and remote

```
const port = process.env.PORT || 3030;
```

and modify server to use this path

```
app.listen(port, () => {
  console.log(`server running on port ${port}`);
});
```

v0.1 - modify package.json Add a script call for startup and minimum Node.js version for Heroku, e.g.

```
...
"scripts": {
  "start": "node server/server.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
"engines": {
  "node": "9.2.1"
},
...
```

v0.1 - add .gitignore Add a `.gitignore` file to the project root, e.g.

```
node_modules/
```

Then, push project to GitHub if applicable.

v0.1 - add Heroku hosting Create new Heroku project in root of app

```
heroku create
```

and then push the local Git repo to Heroku, e.g.

```
git push heroku master
```

Then open hosted app for testing,

```
heroku open
```

Rename the project, if necessary, as follows

```
heroku apps:rename new_name --app current_name
```

v0.2 - add socket.io Update server to work with web sockets. The server needs to accept these socket connections, whilst the client makes them.

Install Socket.io,

```
npm i socket.io --save
```

and then add require to `server.js`

```
const socketIO = require('socket.io');
```

Then, integrate sockets with the server.

modify server to work with sockets Express relies on the HTTP module. However, by default this usage is implicit as part of the basic usage of Express.

For socket.io, we need to make this HTTP module usage explicit, which then allows us to integrate sockets in to the server. This module is available by default (i.e. no extra NPM install), but it still needs to be explicitly required

e.g.

```
const http = require('http');
```

Then, we create an explicit server using this module, e.g.

```
var server = http.createServer
```

n.b. when we call `app.listen()` , this implicitly calls the `createServer` method passing `app` as the argument for server.

However, when we explicitly call `createServer` we need to pass a function with the standard `req` and `res` callbacks, e.g.

```
var server = http.createServer((req, res) => {
  ...
});
```

However, due to the nature of the integration of HTTP and Express, we can simply create the server as follows, e.g.

```
var server = http.createServer(app);
```

in effect passing Express as the argument for the explicit HTTP server. We can also update the call to the listen method as well, so we are now using the explicit HTTP server, e.g.

```
server.listen(port, () => {
  console.log(`server is running on ${port}`);
});
```

Then, we can pass the server to socket.io, e.g.

```
var io = socketIO(server);
```

This will now return the web socket server in this variable `io` . This is how we can then communicate back and forth between the server and client using web sockets.

v2.0 - server and socket.io.js library We're now ready to send and receive web socket connections between the server and client.

By loading this server with socketIO, we now have access to the `socket.io.js` library on the client-side, e.g.

- localhost:3030/socket.io/socket.io.js

In effect, the server provides a route to the *socket.io* library. We can now call this library from our HTML files, e.g. in the `index.html` file for our app

```
<script src="/socket.io/socket.io.js"></script>
```

This gives us access to the *socket.io* library's methods &c. on the client-side.

We can start by creating a variable to hold the socket connection in our app, which we need for communications in the app. From the client to the server, and vice-versa. e.g.

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

So, we now have a live communication with the server.

v2.0 - socket communication Communication is now possible in the form of an *event*. These *events* may be emitted by either the client or the server, and either may then listen as well.

Sockets are a persistent technology, keeping a line of communication open as long as both parites, client and server, require. So, updates are executed immediately.

test events - users A couple of events are built-in by default, which help demonstrate the concept of *events* with socket.io. We can keep track of new users, and disconnected users. e.g. we might create a new user as they join the app.

v2.0 - listen for events - default server event In `server.js` , we need to add a listener for the sockets. We can then listen for an event, e.g.

```
io.on('connection', (socket) => {
  console.log('new user connection...');
});
```

This listens for a connection to the app, and then executes the callback. This method is called with a `socket` , which is the individual, specific socket for the captured event in the listener.

Likewise, we can also listen for the default `disconnect` event. However, as a socket is already connected, we have to listen for that specific socket disconnection, e.g.

```
io.on('connection', (socket) => {
  console.log('new user connection...');

  socket.on('disconnect', () => {
    console.log('user was disconnected...');
  });
});
```

v2.0 - listen for events - default client event Likewise, we can listen and respond to events in the client.

In `index.html` , we can add the following to listen for a socket event

```
socket.on('connect', () => {
  console.log('connected to server...');
});
```

The client is listening for a successful connection to the server, and then executing the defined callback function.

As with the server, we can also listen for a disconnect event,

```
socket.on('disconnect', () => {
  console.log('disconnected from server...');
});
```

v3.0 - listening and emitting custom events Quick refactor of code,

- move sockets logic into separate JS file, `/public/js/index.js` , and then load script file in `index.html` .

We can now add custom events for a new note, e.g.

- emit from the server for newly created note - listen on the client for a new note
-

v3.0 - listening and emitting custom events - new note from server to client For a custom event, we define an event name for use with a socket, e.g.

```
socket.on('newNote', () => {
  console.log('new note');
});
```

which we can add to the `index.js` file for the client logic in the app. This is the event listener on the client-side for a new note created on the server.

So, we now need to create an event on the server to emit a new note, e.g.

```
socket.emit('newNote', {
  text: '...',
  createdAt: '...'
});
```

The data emitted as part of this custom server event is provided as the first part of the argument to the callback function on the client for this event.

So, we can update the client to be able to use this data, e.g.

```
socket.on('newNote', (note) => {
  console.log(`new note - ${note}`);
});
```

v3.0 - listening and emitting custom events - new note from client to server For this custom event, we need to create a listener on the server, and an emitter on the client.

In `server.js` , we can add a listener in the callback for `io.on()` , e.g.

```
socket.on('createNote', (newNote) => {
  console.log(`createNote from client - ${newNote}`);
});
```

Then, we create an emitter for a new note on the client side, e.g.

```
// emit custom event for new created note - listen on the server
socket.emit('createNote', {
  text: 'a new note has been created by the client...'
});
```

To ensure this emit event is only executed once the socket is connected, we might initially add it in the `connect` event for the client.

v0.4 - event broadcasting We now need to modify server and client to enable app-wide broadcasts, e.g. from multiple clients, tabs, browsers &c.

In `server.js` , we add a new emit event for a newly created note, e.g.

```
socket.on('createNote', (newNote) => {
  console.log(`createNote from client - ${newNote}`);
  io.emit('newNote', {
    text: newNote.text,
    createdAt: currentDate
  })
});
```

By adding this app-wide emit event, we can then remove the socket specific emit events for a new note from both server.js and index.js. These were initially added simply for testing events with sockets. For testing, we can either use a browsers JS console or add a UI form &c. to the app.

e.g. in a browser's JS console, we can issue the following test command

```
socket.emit('createNote', {text: 'a new test note...'});
```

v0.4 - customise event emits With the current server emit for createNote, the event message is returned to all users, including the original user who created the new note.

However, it may only be necessary to broadcast an event to specific users. e.g. a message sent to all to inform them a new note has been created, but a custom message to the original user to inform them the note has been created &c.

Another example might include when a new user has joined the app. All users might receive a message about the new user, and the new user receives a custom welcome message.

So, we need to use a different option for emitting events from the server.

modify emit from server - broadcasting In `server.js` , we can add a specific broadcast call from the current socket,

```
socket.broadcast.emit('newNote', {
  text: newNote.text,
  createdAt: currentDate
});
```

This will broadcast the new note &c. to all connected users, except for the user that connected to this specific socket. i.e. the user that sent the createNote event from the client.

custom events for users - join and connect to app When a user joins the app, two messages will now be sent by `server.js` , e.g.

```
// emit message to specific user of the socket
socket.emit('userMessage', {
  from: 'admin',
  text: 'welcome to the notes app...',
  createdAt: currentDate
});
// broadcast message to all remaining users
socket.broadcast.emit('groupMessage', {
  from: 'admin',
  text: 'a new user has joined the notes app...',
  createdAt: currentDate
});
```

A user message is broadcast to a specific user each time they join the app. Then, a group message is broadcast to all connected users for each new user joining.

We need to update `index.js` to listen for a user and group message, e.g.

```
// listen for a message to a single user - current socket connection
socket.on('userMessage', (message) => {
  console.log(message);
});
// listen for a broadcast group message
// - all connected users except originating user (i.e. new user joined...)
socket.on('groupMessage', (message) => {
  console.log(message);
});
```

v0.5 - abstract utilities - socket message creation Define a function to abstract creation of the object we send in a message for `socket.emit` &c.

This will form part of a group of utilities we can add to our app in `/server/utils/messaging.js` .

e.g.

```
var currentDate = new Date().getTime();
var messageGenerator = (from, text) => {
  return {
    from,
    text,
    createdAt: currentDate
  }
};
```

Then, we can export this method e.g.

```
module.exports = {
  messageGenerator
};
```

for use in `server.js` &c., with a standard `require` call to the file.

use message generator in `server.js` We can use the abstracted message generator with emit calls, e.g.

```
socket.emit('userMessage', messageGenerator('admin', 'welcome User to the notes app...'));
```

v0.6 - events & acknowledgements Event acknowledgements are a built-in feature of Socket.io,

For new notes, we have the following event flows

- emit event on server -> listen on client - io.emit newNote back to client
- emit event on client -> listen on server - socket.emit newNote to server

So, we might use an acknowledgement for the `createNote` event from the server back to the client. We emit an acknowledgement from the server, and listen for it on the client. The acknowledgement needs to be handled at both ends, server and client.

e.g. on client, we add a callback to the socket.emit event

```
// emit event for create note from client
socket.emit('createNote', {
  author: 'spire',
  text: 'test note from the client...'
}, () => {
  console.log('acknowledgement received...');
});
```

and then on the server

```
socket.on('createNote', (note, clientCallback) => {
  console.log('createNote from client', note);
  // app wide event broadcast
  io.emit('newNote', noteGenerator(note.author, note.text));
  // send call to function on client side - callback setup in socket.emit for createNote
  clientCallback();
});
```

The acknowledgement call is now sent from the server to the client to confirm that the original emit event has been received successfully on the server from the client.

v0.6 - acknowledgements - return data in acknowledgement We can also return data in the acknowledgement call from the server to the client.

We can return a single item, or multiple in an object. e.g.

```
// send call to function on client side - callback setup in socket.emit for createNote
clientCallback({
  text: 'acknowledging new note...',
  createdAt: currentDate
});
```

v0.7 - client forms We need to add some client-side forms to allow a user to create a new note &c.

The first example will use HTML5 with jQuery.

jQuery and HTML5 We can download and add jQuery to the app's `index.html` page, e.g.

```
<script src="/utils/jquery-3.2.1.min.js"></script>
```

Our HTML5 form in `index.html`

```
<form id="note-form">
  <input name="note" type="text" placeholder="add some text..." />
  <button>create note</button>
</form>
```

If we submit this form, it will simply cause a page refresh and add a parameter to the URL, e.g.

```
localhost:3000/?note=a+new+note
```

and so on. Therefore, to avoid this page refresh, we need to add a listener for the form's click event on submit in JavaScript.

We'll add client-side UI logic to a file, `ui.js` , which uses jQuery, e.g.

```
$('#note-form').on('submit', (e) => {
  // stop default behaviour for event - i.e. page refresh for form submit
  e.preventDefault();
  console.log('create note form submitted...');
});
```

Initially, we can use the `e` argument (for event) in the callback to prevent the form's default behaviour. This will stop the default page refresh for each form submit.

Then, we can call `socket.emit()` to respond to the form submit for the newly created note, e.g.

```
socket.emit('createNote', {
  author: 'amelie',
  text: $('[name=note]').val()
}, () => {

});
```

render messages to UI - note creation Add element placeholder in `index.html` DOM to store return messages &c. from server.

```
<ol id="messages"></ol>
```

In `index.js` , we can create an element to add to the DOM in the listener for a new note, e.g.

```
// listen for new note created on the server
socket.on('newNote', (note) => {
  console.log(note);
  // create element with note content
  var li = $('<li>');
```

```
  li.text(`${note.author}: ${note.text}`);
  // append new element to DOM placeholder
  $('#messages').append(li);
});
```

render messages to UI - users We can also render messages for user events, including *welcome* and *new* users.

```
// listen for a message to a single user - current socket connection
socket.on('userMessage', (message) => {
  console.log(message);
  // create element with message content
  var li = $('<li>');
  li.text(`${message.from}: ${message.text}`);
  // append new element to DOM placeholder
  $('#messages').append(li);
});
```

So, we've updated the listener for a `userMessage` sent from the server and received by the client. Likewise, we can update the listener for a `groupMessage` , e.g.

```
// listen for a broadcast group message
// - all connected users except originating user (i.e. new user joined...)
socket.on('groupMessage', (message) => {
  console.log(message);
  // create element with message content
  var li = $('<li>');
  li.text(`${message.from}: ${message.text}`);
  // append new element to DOM placeholder
  $('#messages').append(li);
});
```

v0.8 - update UX for client We can now modify the user experience (UX) for the app.

For example, we might clear the entered text once the form data has been sent.

clear input field So, we can modify the listener for the form's input text field, e.g.

```
// create note form submit - event listener
$('#note-form').on('submit', (e) => {
  // stop default behaviour for event - i.e. page refresh for form submit
  e.preventDefault();
  console.log('create note form submitted...');

  var inputText = $('[name=note]');

  // emit data for new note to server...
  socket.emit('createNote', {
    author: 'amelie',
    text: inputText.val()
  }, () => {
    // clear value for message input field in form
    inputText.val('');
  });

});
```

We've abstracted the selector for the attribute on the input field, and then clear it in the callback for the `emit()`

method.

add focus to the input field We can also ensure that the note input text field is set as focus as the app loads. In `index.html` , we can update the attributes for the input text field, e.g.

```
<input name="note" type="text" placeholder="new note text..." autofocus />
```

We also have the option to remove the autocomplete feature for this input field, e.g.

```
<input name="note" type="text" placeholder="new note text..." autofocus automcomplete="off"/>
```

v0.8.1 - add timestamps for notes &c. We need to add formatted timestamps for the creation of a note, updates, users &c.

We can use the Moment.js library to help format these timestamps,

- Moment.js - https://momentjs.com/

We can install it using various tools, including NPM, Yarn, Bower &c.

```
npm i moment --save
```

or we can simply download a copy of the JS file for local reference in the app.

v0.8.1 - Moment.js usage - server-side We can `require` Moment in our app, for Node apps, e.g.

```
var moment = require('moment');
```

Also use Moment to get current timestamp, instead of plain JS

```
var currentDate = new Date().getTime();
```

we can now use,

```
moment().valueOf();
```

For the generation of notes and messages we can update the functions in `messaging.js` and `notes.js` , e.g.

```
// abstract generation of message - use Moment.js for timestamp
var noteGenerator = (author, text) => {
  return {
    author,
    text,
    createdAt: moment().valueOf()
  }
};
```

The call to create the timestamp is in the function itself to ensure it is called each time a new note, or message &c., is created.

v0.8.1 - Moment.js usage - client-side rendering We need to add the Moment.js library to the app to be able use it for client-side rendering, e.g. in `index.html`

We can add a copy of `moment.js` to the `public/utils` directory for the project,

```
<script src="/utils/moment.js"></script>
```

format time for new note In the current listener for a new note,

```
// listen for new note created on the server
socket.on('newNote', (note) => {
  console.log(note);
  // create element with note content
  var li = $('<li>');
  li.text(`${note.author}: ${note.text}`);
  // append new element to DOM placeholder
  $('#messages').append(li);
});
```

we can now add a call to Moment to format the timestamp for each new note, e.g.

```
var formatTime = moment(note.createdAt).format('h:mm a');
```

We can then output this formatted time for rendering to the user, e.g.

```
li.text(`${note.author} @ ${formatTime}: ${note.text}`);
```

format new time for users and messages We can also update messaging in the app, e.g.

```
// listen for a message to a single user - current socket connection
socket.on('userMessage', (message) => {
  console.log(message);
  // use Moment to format createdAt timestamp
  var formatTime = moment(message.createdAt).format('h:mm:ss a');
  // create element with message content
  var li = $('<li>');
  li.text(`${message.from} @ ${formatTime}: ${message.text}`);
  // append new element to DOM placeholder
  $('#messages').append(li);
});
```

Extras - add some style Create a new folder for the CSS stylesheets, `/public/css` , and then add a stylesheet file for the app styling, `style.css` .

Then, we can reference this file in the head metadata of the `index.html` file, e.g.

```
<link rel="stylesheet" href="/css/style.css">
```