Before starting, you must:

* Install Latest version of NodeJS (nodejs.org)
* Install Visual Studio Code.

Then, you can just jump to section 2, part 5.

There is no window/document object. There is global/process.

One of the most fundamental features of Node: Module System

To see the convention name of a module, and it’s name that we can use in require, check Docs.

Git repo for the course: <https://links.mead.io/nodecourse>

Module.exports = name; *(in a file.js we gonna require in some other file.js for the name variable)*

To install a module globally, we must add the -g

To install it with admin rights(?), use “sudo” prefix. (sudo npm I [nodemon@1.18.5](mailto:nodemon@1.18.5) -g)

Nodemon app.js => will update the file at any save done to app.js

To terminate the Nodemon, press CTRL+C

Process.argv => argv = arguments variable

Package: yargs => will parse the arguments

A screenshot of a computer

Description automatically generated with medium confidence

^ console.log( yargs.argv );

YARGS implements a –help on the file too. (which will show up the commands tied to the file)

The builder property will contain only the options we want as a given command to support.

(options as arguments)

Title can be an option to our function. By default, we are not demanded to mention it.

To enable that, we must use the demandOption: true inside the command builder option.

Text

Description automatically generated

To enforce a type for the demanded option, we can use property type: ‘string’ like.

yargs.parse() => goes through the process of parsing the arguments with all the config details provided

fs.readFileSync() => returns a data buffer (binary code). We can .toString() to get the string.

For debugging, we can just console.log() everything and everywhere.

There is also the Node Debugger that comes with the V8 JS Engine.

* Just add “debugger” somewhere in the script, and run cmd

**node inspect** app.js add --title="Shopping" --body="sst"

* Go to chrome://inspect and under the Target, click on Inspect

Graphical user interface, text

Description automatically generated

When we finished all steps (by pressing the PLAY blue button), we can restart it by using “restart” in the <debug> command.

**Call Stack** => track the execution of our program, by keeping track of all **functions** that are currently running. (like when we get an error, where every function called is listed until the error spotted)

* **Data structure:** add on the top of the list / remove the top item on the list

SetTimeout is not a JS function, and V8 has no implementation for it. Is NodeJS which creates an implementation of setTimeout using c++ and provides it to your NodeJS scripts to use. => it is an async way to wait a specified amount of time, and then have a function run.

So when we call setTimeout, is registering an event with NodeJS APIs, and that is an event callback pair, where the event in this case is simply to wait 2 seconds, and the callback is the function to run.

Another event callback pair might be to wait for a database request to complete, then run the callback that does something with the data.

**Node APIs** => ^

**JS is a single threaded programming language**. You can do one thing at a time.

But that doesn’t mean Node.js is completely single threaded. The code you run is indeed still single threaded, **but Node uses other threads in C++ behind the scenes to manage your events.**

**Callback Queue** => maintain a list of all the callback functions that are ready to get executed.

This is the moment the **Event Loop** comes into play. It looks at the Call Stack, and at the Callback Queue.

If the call stack is empty, it’s going to run items from the callback queue.

npm init -y => answers “yes” to all questions

Use postman-request package instead of request package, as it is now deprecated.

The “postman-request” will always return an object: body {} so we go in response.body.

encodeURIComponent(string input) - to convert the input to a safe string. (will convert ? to %3F for e.g.)

**QUESTION: In our weather API, in the accessed object, there is array weather\_descriptions.**

**When we are destructuring the object, how can we specify an index of that array?**

**If we try { weather\_descriptions[0]:description } it fails: SyntaxError: Unexpected token '['**

HTTP ‘n’ HTTPS modules of NodeJS

const request = http.request( url, (response) => {

let data = “” ;

response.on( ‘data’, (chunk) => {

// allows us to register a handler – different events we can call it for: data.

// this event will fire when data comes in

// it can be called once, or more times, so we must save the chunk in a variable until all finished

})

Response.on(‘end’,() => {

// the end event -> when we are done

})

})

request.on( ‘error’, (error) => { // some error handling });

request.end(); // we are just now sending the request.

//-------------------------------------------------------------------------------------------------------------------

**ExpressJS** jumps in.

Fast, unopinionated, minimalist web framework for NodeJS

npm i express@4.16.4

const express = require(‘express’)

const app = express();

app.get(route, ( info about incoming request aka req, res ) => { // let’s us configure what the server should do when someone tries to get the resource of the specific url (maybe we should back HTML, or maybe JSON)

res.send( “text” ); // sending something back to the requester. (can be html or object/array)

})

app.listen(3000, () => { console.log(‘Server is up on port 3000”) }); // starts the server – development port = 3000 (it’s not a standard)

// port 80 => for HTTP based websites

When doing changes to the code, server won’t be updated. We would be needed to close it and restart.

Instead, we can use nodemon which will reload the code at every change!

Node provides an absolute path to the public directory.

\_\_dirname

\_\_filename

NodeJS Path helps.

app.use( express.static(path.join(\_\_dirname,"../public")) )

express.static() => takes the path to the folder we want to serve up

**Relative path:** ./css/style.css

**Absolute path:** /css/style.css

**Template engine:** Handlebars => allows us to do 2 things:

1. Render dynamic documents as opposed to static ones
2. Create code that we can reuse across

But for using with express, we use: HBS package

Dynamic pages will now have the extension .hbs and will be placed in the project directory, under /views

To load the page ^ with express, we must add a route, and use res.render(index) (instead of res.send)

App.set( ‘view engine’, ‘hbs’ )

App.set( ‘views’, viewsPath )

https://expressjs.com/en/4x/api.html#app.set

hbs.registerPartials( partialsPath ); // sets the directory path for partials

To add a partial in another .hbs file, we must use {{>partial file name}}

To make the nodemon listen to other file changes too (instead of just .js files), we must use:

Nodemon src/app.js -e js,hbs

For a route that was not registered, we can display a 404 error: Not found.

To do that, expressJS gives the route \* .

App.get( ‘\*’, (req, res) => {} );

And can be mixed with another path, like: ‘/help/\*’

But those must be added as latest routes.

**req.query** => provides the key/value pairs from the url

**fetch() is not available in NodeJS, as it is part of the browser API.**

**Heroku** is going to give us everything we need to deploy our Node.js application to a production server.

**SSH** => Secure Shell = secure communication between 2 machines (through ssh key)

ls -a -l ~/.ssh => checking if SSH exists on computer. (~ symbol is a shortcut for user directory)

ssh-keygen -t rsa -b 4096 -C “[myemail@address.com](mailto:myemail@address.com)” => generating ssh key of type RSA (very secure and popular protocol), with 4096 bits, with a comment for the key (as a label) ((common to just use your email address for it))



First is a secret file, the key we won’t ever share, and we will just keep it on our machine.

Second is the public file key which we will share with GitHub and Heroku, for a secure communication between our machine and their server.

eval $(ssh-agent -s) => try to start up SSH agent, or simply print the process if it is already running.

ssh-add ~/.ssh/id\_rsa => add the identity

cat ~/.ssh/id\_rsa.pub => concatenates the content of the file out to the terminal

ssh -T [git@github.com](mailto:git@github.com) => testing our connection to github

heroku keys:add => heroku looks for available public keys in ~/.ssh/, and asks which one to use (if more)

heroku create app-name-unique-wide => create a new application

const port = process.env.PORT || 3000;

When running locally, script will use port 3000.

When running on Heroku, we get the port from process.env(ironment).PORT

3 changes needed:

1. Tell Heroku how to start the application (“scripts” { “start”: “node src/app.js” })
2. Make sure that all applications listen the PORT Heroku is providing
3. Fetching the data from the right path.

(move from localhost:3000/weather?search to /weather?search)

git remote => view all the remotes that are configured

git push heroku master => push to heroku git remote

When heroku sees that new commits has been pushed, is going to deploy our app again.

"scripts": {

    "start": "node src/app.js",

    "dev": "nodemon src/app.js -e js,hbs"

  },

npm uninstall -g nodemon => uninstalling global package, as “dev” cmd won’t work for somebody who doesn’t have nodemon installed globally.

npm i nodemon@1.2.0 --save-dev => installing nodemon as a devDependency only, which is used only when developing the app. When deploying it, they are not installed in the production server.

The command nodemon src/app.js won’t work anymore too. Only the ‘dev’ command will be able to access it.

**MangoDB is NoSQL**

SQL = Structured Query Language =uses=> tables

((in a table, classic rows/records and columns)

NoSQL = Not only SQL =uses=> collection (like JSON)

((in an array, one object = document, one property = field))

**ERROR: Section 10 is deprecated. (There is no such thing as Robo 3T anymore)**

**Instead, use Atlas.**

**There is a file in this directory with more details; check that out.**

**Text

Description automatically generated**

result.ops => an array of documents (in our case, one document) ((document as in NoSQL!))

^ from the documentation

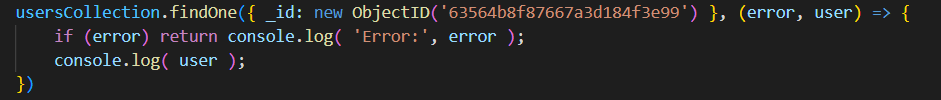
In MongoDB, the \_id s are known for GUIds (?) => Globally Unique Identifiers

\_id.id => raw binary code of the id.

ObjectId(‘o42ij3fi’) => the ID is not just a string. It is actually what is returned by the function, which is binary data.

We can fetch data from collection (db) by using collection.findOne / .find for e.g., for which it receives 2 arguments: an object, with properties used for finding/filtering documents (rows), and the callback function with (error, document);

When searching by ID, we must give \_id a new ObjectId(‘string id’)

Like: 

.find => doesn’t take a callback as a second argument, because .find will return a Cursor. (a pointer)

MongoDB doesn’t assume you need the documents in an array, so let’s you do whatever you want.

The Cursor has a .toArray(( error, documents ) => { console.log( documents ) } ) , .count( same ), etc.

When updating documents, we can use .updateOne / .updateMany.

For the update parameter, we cannot use name: ‘Andrew”, but we need to use update operators to define the behavior we want to perform.

https://www.mongodb.com/docs/manual/reference/operator/update/

Text

Description automatically generated

.updateOne / .updateMany (probabil .find / .findOne too) returns a promise, if no callback is specified.

**Mongoose library.**

Text

Description automatically generated

https://mongoosejs.com/

Graphical user interface, text, application

Description automatically generated

**Validation and Sanitization are possible.**

Text

Description automatically generated

REST API

Representational State Transfer – Application Programming Interface (REST API or RESTful API)

express.use( express.json() ) => will automatically parse incoming JSON to an object, so we can access it in our request handlers.

A screenshot of a computer

Description automatically generated with medium confidence

<https://www.webfx.com/web-development/glossary/http-status-codes/>

Text

Description automatically generated

**Mongoose provides query methods on the model class, like this:**

https://mongoosejs.com/docs/queries.html Text

Description automatically generated

const updates = Object.keys( req.body )

Object.keys( obj ) => puts all properties of the object in an array, and returns it.

When about routing, here we go:

const router = new express.Router()

there are router.get .post .patch .delete

But, in order for the route to work, we must register it with our existing app.

const router = new express.Router()

router.get( '/test', (req, res) => {

    res.send('This is my other router')

})

app.use(router)

All routers must be placed in app/src/routers

Then, we can import it app/src/index.js by

const userRouter = require('./routers/user')

app.use(userRouter)

Hashing algorithm: BCrypt JS npm i bcryptjs

To hash a password, we can use:

const bcrypt = require('bcryptjs')

const hashedPassword = bcrypt.hash(password, 8)

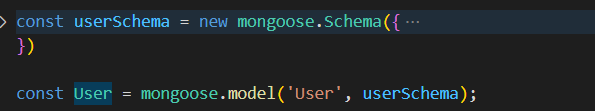
8 stands for rounds of hashing. 8 is the perfect balance between a secure pass and a good hashing speed

**Encryption != Hashing**

Encryption gives the opportunity to decode the password too, while Hashing can only encode it.

Middleware (also called pre and post *hooks*) are functions which are passed control during execution of asynchronous functions.

Middleware is specified on the schema level and is useful for writing [plugins](https://mongoosejs.com/docs/plugins.html).



For a schema, we have .pre and .post methods which can take place before/after an event.

Accepts 2 arguments: event name, and callback function (if anonymous, cannot be arrow func!!!)

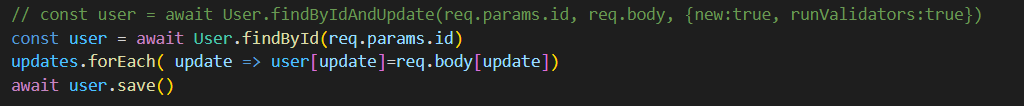
Also, need to call next() at the end of the callback.

A screenshot of a computer

Description automatically generated with medium confidence

The problem is when trying to update an item with .patch, because .findByIdAndUpdate **makes a direct call** to mongodb and .save of mangoose never gets called!

Instead, we must use .findById and we must do the update manually! (for when need to hash a pass)



Mangoose also provides a way to confirm/not if a property was changed. (like our pass)

Text

Description automatically generated

To add our own methods to model, we need to use a mongoose Schema just like before. That’s the only way we can add it.

So, after defining the Schema, we can just create the method under e.g. userSchema.statics.ourMethodName = () => {}

To access the method, we must use e.g. User.ourMethodName()

As another important property of a document field, we have ‘unique: true’ which will make sure the input value is unique in the document. (will never repeat. Good for Emails!)

Text

Description automatically generated

If done later, database must be wiped in order for that ^ to work.

JWT = JSON Web Token

npm i jsonwebtoken

const token = jwt.sign({ unique id }, ‘secret-phrase’, { expiresIn: “7 days” } ) // 3rd argument is optional

console.log( token )

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiJhYmMxMjMiLCJpYXQiOjE2NjY3Njc0Mjl9.HLYc2VXLHUSFxTG2j\_XPxCqph3EMBa2Fdh\_GoAj8\_N8

header => base 64 encoded JSON string => metainformation (type of token, jwt, the algorithm used)

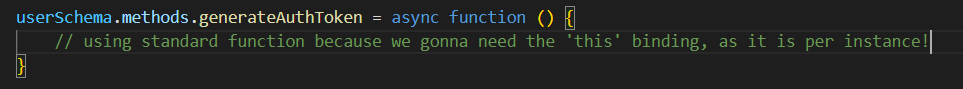
payload/body => base 64 encoded JSON string => contains data that we provided (the id)

signature => used to verify the token

The Goal of the JSON Web Token is not to hide content.

userSchema.methods.ourNewMethodName

^ to register new methods to the user instance (not User model, like earlier, when used .statics)



**Express Middleware**

// without middleware: new request -> run route handler

// with middleware: new request -> do something -> run route handler

Text

Description automatically generated

^ and that’s how we can stop certain requests to happen.

To add middleware to an individual route, we gonna put the middleware as the second parameter, and move the callback function to be the third function.

Text

Description automatically generated

In our case, if ‘auth’ calls the next() function, the async callback function will be called. Otherwise, not.

For when authenticating, we must add ‘Authorization’ key to **Header**, with value ‘Bearer <token>’

Then, in the middleware, we read the header info, remove the ‘Bearer ‘ and verify the token.

Then, look for the user with that token.

const auth = async (req, res, next) => {

    try {

        const token = req.header('Authorization').replace('Bearer ','')

        const decoded = jwt.verify(token, 'thisismynewcourse')

        const user = await User.findOne({\_id: decoded.\_id, 'tokens.token': token})

        if (!user) throw new Error()

        req.user = user

        next()

    } catch(e) {

        res.status(401).send({error: 'Please authenticate first.'})

    }

}

Advanced Postman -> pm stands for postman.

Graphical user interface, application

Description automatically generated

Hiding private data

Text

Description automatically generated

We can do this also by not calling user.getPublicProfile! How? We simply change the name of the method to toJSON and things will go by themselves!

**.toJSON is a middleware (?!) which is called before an object is JSON.stringified (?!)**

To create a connection between Tasks and Users, we have 2 options:

1. Create an array in User where we store the IDs of tasks owned by that user
2. Create a new property to Task, where we store the ID of owner

Text

Description automatically generated

Mongoose also helps with ability to add an option “ref” which accepts the name of another Model.

This way you can tie 2 models between.

Owner: {

Type: mongoose.Schema.Types.ObjectId,

Ref: ‘User’

}

User used for ref is the name used in mongoose.model( name, schema ) inside model.js

await task.populate(‘owner’) => allows us to populate data from a relationship such as the data we have for owner.

* Is gonna find the user who’s associated with this task, and task.owner will now be their profile (the entire document instead of being just the id)

Virtual property => is not data from database, is actually a relationship between 2 entities. (user / task)

userSchema.virtual('tasks', {

    ref: 'Task'

})

Examples of a route with auth required:

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

We can set the route (/tasks) to have filters like ?completed=true / false

When we read the req.query.completed, we get ‘true’ / ‘false’ as strings, and is our job to convert them to Boolean.

Actually, every .query we get from the URL is a string.

Here is a nice trick:

Graphical user interface

Description automatically generated

Text

Description automatically generated

Mongoose lets us add options when fetching data, like limiting the number of items we receive.

Graphical user interface, application

Description automatically generated

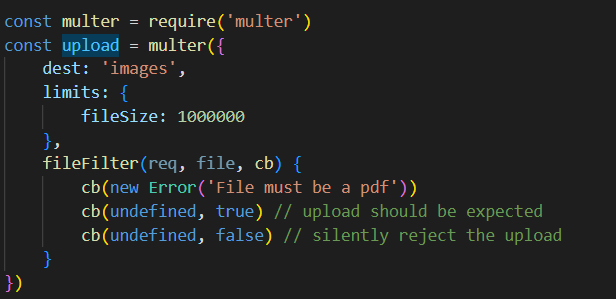
**If there is no limit specified in the URL, mongoose will just ignore the option.limit for good!**

**Ternary operators => named so because there are 3 pieces: condition, value for when condition is true, and value for when condition is false.**

Express by default doesn’t actually support file uploads.

But there is a npm library also released and maintained by the same people who does Express.

npm i [multer@1.4.1](mailto:multer@1.4.1) (Faced 3 high errors)



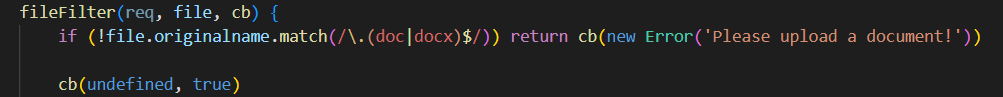
dest : ‘images’ (app/images)

limits : { fileSize : 1 000 000 } (max file size is 1mb)

fileFilter(request, file, cb)

Regex stands for Regular Expresions

[www.regex101.com](http://www.regex101.com)



**Handling Express Errors** – by using a callback function as the 4th argument, with (err, req, res, next) parameters, which lets express know this is the function to handle any uncaught errors.

Text

Description automatically generated

**Serving up files** - ending back the avatar to the browser, from text to image:

Text

Description automatically generated

**Auto-cropping and Image formatting (before saving into the database)**

npm i sharp

(if you using latest version of nodeJS, use latest version of sharp. Don’t @0.21.1)

Await sharp(req.file.buffer).png.toBuffer()

Text

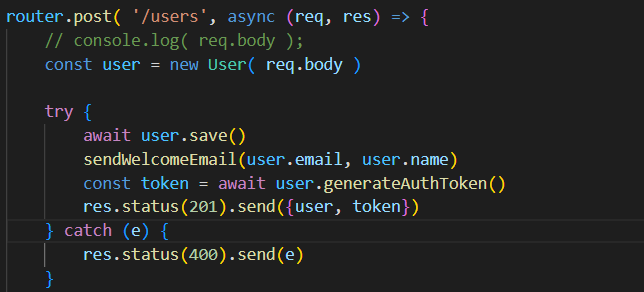
Description automatically generated

**Section 15**

Read the related file in this repo.

Even if sending a mail is an async method, we don’t need to wait for it to continue our script.

So there is no need to use await sendWelcomeEmail. We just send it, and that’s it.

****

**Html** property is also available for sgMail.send({}), to display html code inside.

To load Environment Variables, we can use env-cmd which is cross OS compatible.

npm i env-cmd

app directory/config/dev.env

Text

Description automatically generatedA screenshot of a computer

Description automatically generated with medium confidence

process.env.KEY

Text

Description automatically generated

Heroku deployment



Config env of heroku



If get –app error, make sure you set it through



**Testing NodeJS**

With JestJS ( there is also MochaJS)

npm i jest



When we want to register a test case, we use the test( ‘test name’, () => {} )

Graphical user interface, text

Description automatically generated

Why to test?

* Saves time
* Creates reliable software
* Gives flexibility to developers
  + Refactoring
  + Collaborating
  + Profiling
* Peace of mind

JestJS has expect() which can save us some time. Instead of doing everything (like the commented part),

we can use expect() with method .toBe. There are dozens of methods for it.

Graphical user interface, text

Description automatically generated

When providing an argument to the callback function, which is actually also a callback function, JEST will know that the test case is async!

Text

Description automatically generated

JEST should use a testing environment (test.env) because we don’t want to run tests on our real database. So, as a difference, the app/config/test.env will include a link to a database meant for testing!

Must also config some related to JEST

Text

Description automatically generated

By default, Jest is supposing you are using testEnvironment ‘jsdom’ (javascript env similar to what you see in browser)

Gr8 for testing Express applications is: supertest

npm i supertest –save-dev

In order to load Express into test cases, we had to do some refactoring.

So, almost everything from index.js gone to a new file: app.js

Out of app.js, we exported the app as a module, to be able to call it from index.js (app.listen())

So, when doing tests, we need just app.js, and when doing development, we load index.js

Text

Description automatically generated

JestJS comes with other global functions like: beforeEach( () => {} ) and afterEach( () => {} ) which are called before/after each test case apart.

Setting authorization in a test case:

Text

Description automatically generated

By .set, after .get, we are customizing the header of the request, so we can include our key=value pair regarding the authentication.

In the \_\_mocks\_\_ folder, inside tests, we can mock libraries. (like the SendGrid -> we don’t need to send mails with every test case we run)

This



Will lead to a folder named @sendgrid inside \_\_mocks\_\_, which will have a mail.js file inside!

Inside it, we will create our own functions, empty, but named by the original library function names.

Text

Description automatically generated

This way, during our tests, no mails will be sent. (especially when we using a free provider, which limits our number of mails that we can send per day c: )

fixtures => things that allow you to set up the environment your tests are going to run in.

Graphical user interface, application

Description automatically generated

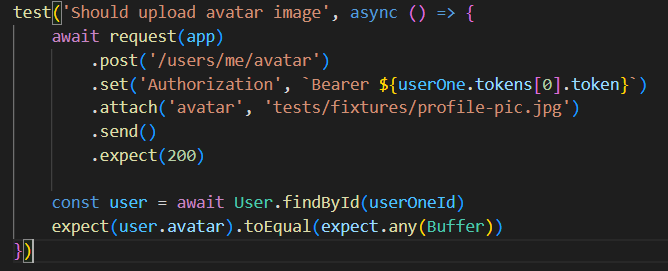
For e.g., we gonna put an image inside this directory that our test can use when trying to upload to the server to test that endpoint

{} === {} Is false. When comparing anonymous objects, it looks for the memory data of them which is not equal.

So, expect({}).toBe({}) will lead to a negative response.

That’s why expect({}).toEqual({}) was created!

Here’s how to create a test case with uploading a file (from fixtures directory)



Before creating the Task Test Suite, we must share the database between user.test.js and task.test.js.

What database? Well, the userOne like, which are used in the test cases.

Text

Description automatically generated

So user.test.js si task.tes.js just required the exported data from fixtures/db.js

JestJS runs our test suites in separate

Our test suites (being done in the same time) can interfere. So they must be realistical, but we have to take that into account too!

To avoid it, we can use:



This will make sure JestJS run tests one by one, so they cannot have conflicts.

**Web Sockets (last part of course)**

* Allow for full-duplex (bidirectional communication

This means that client can initiate communication with the server, BUT also server can initiate communication with the client!

On HTTP requests, it was the client’s job to initiate the request asking for data from the server, and just then the server could send a response.

* WebSocket is a separate protocol from HTTP
* Persistent connection between client and server

**socket.io library**

* The way node can use Web Sockets

npm i socket.io

Though, to use express with web sockets, we need to refactor it a bit.

We need http library, saved in a constant. (http)

After we create the app, we must create a server with the app as argument.

Then, instead of calling the app.listen(), we will do server.listen() !

const http = require('http')

const server = http.createServer(express\_app)

server.listen(port, () => {})

At this point, our server supports Web Sockets

Text

Description automatically generated

We just required the socket.io library, and served the server as an argument.

const socketio = require('socket.io')

const io = socketio(server)

Then, for the client to connect to the server through socket.io library, we must use it’s client-side version:

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

Then we can load our client-side script which will handle the connection

<script src="/js/chat.js"></script>

So inside chat.js, we must just add call io() and now the server will be aware of the client’s existence!

Adding an argument (socket) to the io.on() callback will provide you an object which contains information about the new connection. So we can use methods on that socket object

socket.emit => will send an event to client, from server



Any extra argument we add to the socket.emit() will be passed to the client receiving the event!

Also, any method used on the ‘socket’ from the server, is targeting that only client.

To send an event to everybody, we must use ‘io’ object!

Client side down below, receiving the ‘count’ from server.

Text

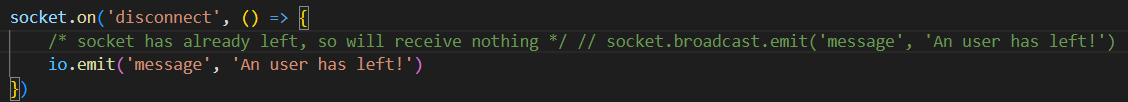
Description automatically generated

**Broadcasting Events** will be sent to everybody except one specified client. (like the one texting!)



**There is no ‘disconnect’ event on io, but on socket!**

So, to know when a client disconnects, we do just this:



^ inside io.on(‘connection’)

We don’t need to broadcast an event to all the other clients, when somebody leaves. We just send an event to everybody, because the client we want to omit has already left.

Share location

Not all browsers support navigator.geolocation, so better check for it first before using.

Graphical user interface

Description automatically generated with medium confidence

Accepts a callback function as argument. Browser don’t have promises built in, so responses comes so.

In a basic communication through Web Sockets, client would never know if the event he sent to the server was actually received.

For this, we have **Event Acknowledgements** !

Server 🡪 client 🡪 acknowledge server

Client 🡪 server 🡪 acknowledge client

This acknowledgement happens when, in the socket.emit from client is added a final argument as a callback function. Then, when server receives the event, passes a last argument ‘callback’ and calls it after the entire script has ran.

e.g.

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

Also, of course callbacks can get as many arguments as wanted.

bad-words npm package

^ we can use this filter library and call the callback before sending message to everybody else, if it has bad words inside!

const Filter = require('bad-words')

Text

Description automatically generated

Text

Description automatically generated

Timestamp => nr of milliseconds since midnight, 1 jan, 1970. (known as the Unix epoch)

Moment library => gives us all of the tools necessary to format time.