**Browser JS versus node JS**

Neste primeiro tópico iremos abordar algumas das diferenças entre o javascript usado no navegador e o javascript usado no node js. Quando estamos trabalhando em um aplicativo node js não temos acesso a nenhum tipo de browser API, então não existe o DOM, e outros recursos do browser. Em node js a gente só trabalha com a parte de servidor sem a parte gráfica da interface que o browser proporciona. Essas são umas das desvantagens do node, mas vamos mudar um pouco e vamos falar um pouco das qualidades, em node js podemos acessar arquivos, informações sobre a conexão e também o requerimento de dados, também diferentes de aplicativos para navegador podemos acessar os módulos pelo padrão e também utiliza de simples javascript para se executar.

**BROWSER x NODE.JS**

✦DOM ✦No DOM

✦Window ✦No Window

✦Interactive Apps ✦Server-Side Apps

✦No Filesystem ✦Filesystem

✦Fragmentation ✦Versions

✦ES6 Modules ✦Common Js

**Instalações necessárias**

Primeiro cheque se em sua máquina você tem o node instalado se não o instale:

<https://nodejs.org/>

**CLI**

Agora vamos aprender como utilizar o node js criando um projeto primeiro crie uma pasta, abra ela no seu editor no meu caso VSCode e um crie um arquivo que pode ter qualquer nome .js. Agora crie uma variável const amount = 12 e dentro crie uma condição if(amount<=10) {console.log (‘small number’)} else {console.log (‘large number’)} e de fora crie um console.log (`hey its my first node js app!!`), agora você deve estar se perguntando como eu executo esse arquivo, vá ao terminal acesse a pasta utilizando o comando cd e drope a pasta, agora digite node (nome do arquivo.js), no meu caso app.js.

**GLOBALS**

Agora vamos cobrir os conceitos de globals ou variáveis globais em node js, ao invés de cobrir todas as variáveis globais iremos lhe dar uma ideia geral e ao progredirmos no curso iremos mostra-las em pratica. Quando trabalhamos em vanilla JS temos acesso as janelas que lhe proporciona bastante legais, porem em node não trabalhamos com janelas se por exemplo você usar um querySelector no seu projeto em node ele vai dar erro, mas em node também temos o conceito de globais o que significa em qualquer lugar do projeto se você quiser acessa-las será possível.

//GLOBALS - NO WINDOW!!!

// \_dirname - path to the current directory

// \_filename - file name

// require - function to use modules (common JS)

// module - info about the current module (file)

// process - info about env where the program is being executed

Se você pode usar o console.log() para testar as variáveis e observar o que elas fazem, dentro do node também temos acesso ao setInterval e o setTimeout crie um setInterval e de um log dentro dele chamando ‘hello world’ e coloque o intervalo em 1 segundo. setInterval (() => {console.log(‘hello world!’)},1000), para parar o terminal use o comando ctrl + c.

**MODULES**

Vamos falar sobre módulos primeiro você deve estar se perguntando eu tenho que fazer meu código todo nesse arquivo e a resposta é sim e não pois você irá separar seu código por módulos, porque seria impossível colocar todo o código em um único arquivo. Agora vamos criar uma simples função para mostrar o nome das pessoas.

// CommonJS, every file is module (by default)

//Modules - Encapsulated Code (only share minimum)

const secret = 'SUPER SECRET'

const john = 'john'

const peter = 'peter'

const sayHi = (*name*)=>{

    console.log(`Hello there ${*name*}`)

}

sayHi('susan')

sayHi(john)

sayHi(peter)

Observe não faria mais sentido se os nomes fossem separados ou as funções em diferentes arquivos e então podemos acessá-los ao longo da aplicação, então dessa maneira teremos arquivos menores e mais estruturados para a nossa aplicação isso é exatamente o que os módulos nos permitem fazer.

Crie dois arquivos separados um com qualquer nome guardando as variáveis e outro com qualquer nome guardando a função nos meus arquivos eu coloquei 4-nome.js e 5-utils.js. Olhando o código agora você digitando node app.js você acha que irá funcionar, e a resposta e não. Agora no arquivo 4-name.js dê um console.log() em module, e depois de rodar você nota algo interessante você tem um objeto com algumas propriedades e a que eu estou mais interessado é a exports que mostra que ela também é um objeto, então qualquer coisa que eu colocar dentro do objeto eu irei conseguir acessá-lo em qualquer lugar da aplicação, agora observe que antes as variáveis eram compartilhadas em qualquer lugar da aplicação. Para tornar os nomes e exporta-los precisamos usar a seguinte sintaxe, e possível fazer isso de várias maneiras a usada foi essa module.exports = {john, peter}. Agora podemos voltar para app.js e poderemos acessá-las usando uma variável global o require, podemos atribuir essas variáveis do arquivo 4-name.js a uma variável ou só usar o require, neste exemplo iremos atribui-las para isso digite const names = require(‘./4-names.js’), para chamar o require sempre precisamos usar ‘. /’ as vezes seus módulos serão 2 pastas ou 3 pastas acima, mas adicionando um ponto na frente você consegue acessá-los e também teremos módulos de terceiros e módulos construídos. Comente as funções chamando as variáveis e em seguida de um console.log () em names observe que ela retorna agora um objeto referenciando as variáveis, e também dessa maneira não temos acesso a secret porque ela só pertence ao arquivo e também temos que exportar a função em 5-utils.js, digite no final module.exports = sayHi e você estará exportando a função, vá no arquivo app.js e importe usando o require ficará assim const sayHi = require(‘./5-utils’), agora retire os comentários colocados na função. Opa ocorreu um erro em que o nome john não foi definido, mas importamos os nomes lembra, então temos duas opções a primeira desestruturar as variáveis e a segunda que são os nomes.propriedades, desestruturando ficaria assim const {john, peter} = require(‘./4-names’) e com nome propriedades só alteraríamos para names.john na função.

4-names.js

//local

const secret = 'SUPER SECRET'

// share

const john = 'john'

const peter = 'peter'

//console.log(module)

*module*.*exports* = {john,peter}

5-utils.js

const sayHi = (*name*)=>{

    console.log(`Hello there ${*name*}`)

}

*module*.*exports* = sayHi;

app.js

// CommonJS, every file is module (by default)

//Modules - Encapsulated Code (only share minimum)

const names = require('./4-names')

//const {john, peter} = require('./4-names')

const sayHi  = require('./5-utils')

console.log(names)

sayHi('susan')

sayHi(names.john)

sayHi(names.peter)

//sayHi(john)

//sayHi(peter)

**ALTERNATIVE SYNTAX**

Aprendemos duas formas como exportar os valores na primeira quando temos uma ou mais coisas exportamos em forma de objeto e a segunda exportamos a variável sozinha, também temos uma terceira forma vamos observar como ele é feita, primeiro crie um novo arquivo 6-alternative.js, agora crie um array const items = ['item1','item2'] e um objeto const person = {name: 'Bob',}, a primeira forma é exportando direto, no array retire o const e adicione um module.exports.items, assim você já estará exportando em forma de uma propriedade. E se você quiser ser mais rebelde você pode criar a propriedade no module.exports e salvar o valor nela, por exemplo module.exports.SinglePerson = person.

6-Alternative.js

//const items = ['item1','item2']

*module*.*exports*.items = ['item1','item2']

const person = {

    name: 'Bob',

}

*module*.*exports*.SinglePerson = person;

Vá ao app.js e crie uma variável para guardar os dados, const data = require (‘./6-alternative’) e depois de um console.log e observe que os dados foram retornados em forma de objeto. Estas são formas alternativas de se usar o module.exports.

app.js

// CommonJS, every file is module (by default)

//Modules - Encapsulated Code (only share minimum)

const names = require('./4-names')

//const {john, peter} = require('./4-names')

const sayHi  = require('./5-utils')

const data = require('./6-alternative')

console.log(names)

console.log(data)

sayHi('susan')

sayHi(names.john)

sayHi(names.peter)

sayHi(data.items[0])

sayHi(data.SinglePerson.name)

//sayHi(john)

//sayHi(peter)

**MIND GRENADE**

Crie um novo arquivo chamado 7-mind-grenade.js e dentro desse arquivo crie duas variáveis guardando dois números const numb1 = 5, const numb2 = 10, em seguida crie uma função normal para adicionar os valores function addvalues () {console.log (`the sum of the numbers is: ${numb1+numb2} `)}, agora vá ao app.js e insira só um require sem salvar em uma variável ou algo do tipo require (‘./7-mind-grenade.js’), agora rode o código e observe que ele mostrou a função e seu resultado, notamos que quando um module com uma função dentro que está sendo invocada no arquivo ela consegue rodar só utilizando do require sem assimilar ela a uma variável.

7-mind-grenade.js

const numb1 = 5

const numb2 = 10

function addvalues () {

    console.log(`the sum of the number is ${numb1+numb2}`)

}

addvalues()

app.js

// CommonJS, every file is module (by default)

//Modules - Encapsulated Code (only share minimum)

const names = require('./4-names')

//const {john, peter} = require('./4-names')

const sayHi  = require('./5-utils')

const data = require('./6-alternative')

console.log(names)

//console.log(data)

sayHi('susan')

sayHi(names.john)

sayHi(names.peter)

sayHi(data.items[0])

sayHi(data.SinglePerson.name)

//sayHi(john)

//sayHi(peter)

require('./7-mind-grenade')

**BUILT-IN MODULES**

O node tem muitos módulos construídos por eles mesmos, que podemos usar nos nossos aplicativos iremos abortar quatro módulos vendo somente o básico pois senão iremos demorar muito tempo cobrindo-os dentre eles são:

✦OS

✦PATH

✦FS

✦HTTP

Agora se você quer saber mais sobre as propriedades acesse: [https://nodejs.org/dist/latest-v14.x/docs/api/](https://nodejs.org/dist/latest-v14.x/docs/api/%20)

**OS MODULE**

O OS modulo permite que tenhamos várias propriedades e métodos uteis para interagir com o sistema operacional como servidor. A configuração padrão para os built-in módulos e alguns externos que iremos cobrir posteriormente são um pouco similares, que iremos criar uma variável e guardar no require o pacote, neste caso ‘os’. E uma vez que criamos isso temos múltiplas opções, você pode acessar esses métodos rodando a variável ou se quisermos um método ou propriedade especifico e só desestruturar no começo. Agora podemos fazer varias coisas como por exemplo obter a informação do usuário que está usando nesse momento. Primeiro crie uma variável const user, e depois chame o método os.userInfo(), dê um console.log no usuário e observe as informações que ele retornou. Outra forma de chamar esse métodos pode ser chamando direto no console.log usando o template strings, para testar essa forma iremos usar um método para olhar a usagem do sistema em segundos, logo abaixo crie um console.log e depois digite o texto {`The System Uptime is ${os.uptime()}seconds`}, também podemos criar um objeto que irá receber os dados, primeiro crie uma variável objeto const currentOS = {}, agora dentro das chaves crie as propriedades para chamar os métodos de os, name: os.type(),release: os.release(), totalMem:os.totalmem(), fremem: os.freemem(), e no final de um console.log na variável para ver o as propriedades.

9-OS-module.js

//const {arch} = require('os')

const *os* = require('os')

//info about current user

const user = *os*.userInfo()

console.log(user)

// return the system uptime in seconds

console.log(`The System Uptime is ${*os*.uptime()} seconds`)

//info about the computer

const currentOS = {

    name: *os*.type(),

    release: *os*.release(),

    totalMem:*os*.totalmem(),

    freeMem: *os*.freemem(),

}

console.log(currentOS)

**PATH MODULE**

Este modulo nos permite interagir com os locais de arquivo mais facilmente, a configuração do modulo vai ser como a do modulo passado vamos criar uma variável e utilizar o require para obtê-lo, const path = require(‘path’), agora iremos começar com um método simples o sep que retorna uma plataform specific separator, para isso de um console.log e chame path.sep, o próxima método que iremos falar é o join que permite que junte os seguimentos do path usando uma plataform specific separator resultado num retorno de um path normalizado. Para isso crie uma pasta com o nome conteúdo e dentro dela uma subpasta com nome subconteúdo e dentro dela adicione um test.txt e adicione um texto qualquer dentro dele. Agora no app.js crie uma variável const filePath = path.join(), dentro do join adicione o caminho das pastas até o arquivo, ‘/conteúdo’,’subconteúdo’,’test.txt’, agora de um console.log na variável e observe o caminho normalizado. outro método que podemos utilizar é o basename que pega a ultima parte do path que no caso seria test.txt, para isso crie uma variável const base = path.basename(filePath), e de um console.log no final para observar os resultados. E o ultimo método que iremos abordar é o resolve que lhe mostra o caminho absoluto, para isso crie uma variável, const absolute = path.resolve(), agora dentro do resolve adicione a variável global \_\_dirname que onde esta localizado a pasta e em seguida o caminho para o arquivo que você quer neste caso aqui conteúdo, subconteúdo, test.txt , e no final de um console.log para ver o resultado.

9-path-module.js

const path = require('path')

//plataform specific separator

console.log(path.sep)

//join method

const filePath = path.join('/content','subfolder','test.txt')

console.log(filePath)

//basename method

const base = path.basename(filePath)

console.log(base)

//resolve method

const absolute = path.resolve(\_\_dirname,'content','subfolder','test.txt')

console.log(absolute)

**FS MODULE (SYNCHRONOUS)**

O próximo modulo é um modulo que interage o com o sistema de arquivos, essencialmente temos duas maneiras de fazer uma é assíncrona sem bloqueio e a outra e síncrona e bloqueada primeiro iremos cobrir as duas e depois iremos explicar síncrono e assíncrona. Agora no app.js vamos desestruturar os dois métodos que iremos utilizar o ReadFileSync e o writeFileSync e chamar o require de ‘fs’, agora crie dois arquivos na sua pasta conteúdo um first.txt e um seconds.txt e dentro deles escreva um texto. Agora crie uma variável, const first = e chame o ReadFileSync, const first = ReadFileSync (), agora dentro do parêntesis temos que mostrar o caminho no primeiro parâmetro e o tipo de unicode, ‘./conteúdo/first.txt’,’utf-8’, faça isso para o segundo arquivo e de um console.log e observe que o ele mostra o texto que você digitou no arquivo, o outro método writeFileSync permite que você crie um arquivo se ele não existe ou reescreva-o se existente, primeiro chame o método e dentro dele temos que passar dois parâmetros o primeiro caminho e nome do arquivo e o segundo o texto podemos usar o template strings se necessário, neste exemplo o nome do arquivo é third-sync.txt, e escrevemos no parêntesis com o caminho assim ‘./conteúdo/third-sync.txt’, `Here is the result: ${first}, ${second}`, agora rode o node app.js e veja se o arquivo foi criado, agora no terceiro arquivo se você tentar reescreve-lo e quando rodar o node app.js novamente ele será reescrito com o que foi passado nos parâmetros, agora caso você queira adicionar mais linhas no seu arquivo temos que passar mais argumento para o writeFileSync um objeto,{flag: ’a’} isso adicionará mais uma linha cada vez que você rodar o node app.js. esses são os métodos de forma síncrona.

10-fs-sync.js

//const {readFileSync,writeFileSync} = require('fs')

const *fs* = require('fs');

//const first = readFileSync('./content/first.txt','utf-8')

const first = *fs*.readFileSync('./content/first.txt','utf-8')

console.log(first)

//const second = readFileSync('./content/second.txt','utf-8')

const second = *fs*.readFileSync('./content/second.txt','utf-8')

console.log(second)

*fs*.writeFileSync('./content/third-sync.txt',`Here is what the method do: ${first}, ${second}`,{flag:'a'})

**FS MODULE (ASYNCHRONOUS)**

Agora pegue o exercício anterior copie cole-o em app.js, agora iremos usar métodos diferentes como são assíncronos iremos usar o readFile e o writeFile, e para fazê-la rodar de forma assíncrona precisamos prover uma callback function, agora crie uma variável para guardar o require, const fs = require(‘fs’), em seguida crie uma variável chamando o método readFile, fs.readFile (‘’), no primeiro parâmetro precisamos colocar o caminho e o arquivo que iremos ler então ‘./conteúdo/first.txt’, precisamos chamar a callback function agora com dois parâmetros nela o primeiro err e o segundo result , (err, result)=>{}, dentro das chaves precisamos chamar esses parâmetros, primeiro observar se tem um erro para isso crie uma condição, if(err){concole.log(err); return null;}, e embaixo chamar o resultado, aqui nesse simples exemplo podemos usar console.log mas iremos fazer de uma forma diferente crie uma variável para salvar o resultado, const first = result. Agora rode o node app.js observe que lhe mostrará um Buffer, isso acontece pois não adicionamos o tipo de dado que ele mostrará no console, para isso adicione um terceiro parâmetro entre a callback e o caminho,’utf-8’ agora note que ele mostrará certinho como no exemplo passado. Agora para chamar o segundo texto crie o mesmo esquema abaixo, fs.readFile(‘’,’’,()=>{}), agora no primeiro parâmetro passe o caminho no segundo o tipo e na callback os dois parâmetros de err e resultado, ‘./conteúdo/second.txt’, ’utf-8’, (err, result)=>{}, agora nas chaves da callback procure mostrar o erro e crie uma variável para guardar o segundo resultado, if(err){concole.log(err); return null;} , const second = result. Agora para escrever um arquivo vamos usar o writeFile para isso inicie abaixo do segundo readFile, fs.writeFile(), agora passaremos os três parâmetros, um para o caminho do arquivo e outro mostrando o texto/conteúdo do arquivo, e por último a callback, ‘./conteúdo/third-async.txt’,` Here is what the method do: ${first}, ${second}`, ()=>{}. Não se esqueça de criar o arquivo novo certo antes de rodar o código, passe os dois parâmetros para a callback e em seguida teste se há erro no final de um console.log no resultado, (err, result)=>{ if(err){concole.log(err); return null;} console.log(result)}, agora rode o node app.js e repare no console temos um undefined porém, o que temos que observar é se no arquivo novo que criamos está transmitindo a mensagem que escolhemos colocar, outra coisa para repara no código e que criamos uma grande estrutura diferentemente da síncrona.

11-fs-async.js

//const {readFile,writeFile} = require('fs')

const *fs* = require('fs')

/\*

readFile('./content/first.txt','utf-8',(err,result)=>{

    if(err){

        console.log(err)

        return null;

    }

    //console.log(result)

    const first = result;

    readFile('./content/second.txt','utf-8',(err,result)=>{

        if(err){

            console.log(err)

            return null;

        }

        const second = result;

        writeFile('./content/third-async.txt',`Here is what the method do: ${first}, ${second}`,(err,result)=>{

            if(err){

                console.log(err)

                return null;

            }

            console.log(result)

        })

    })

})

\*/

*fs*.readFile('./content/first.txt','utf-8',(*err*,*result*)=>{

    if(*err*){

        console.log(*err*)

        return null;

    }

    //console.log(result)

    const first = *result*;

*fs*.readFile('./content/second.txt','utf-8',(*err*,*result*)=>{

        if(*err*){

            console.log(*err*)

            return null;

        }

        const second = *result*;

*fs*.writeFile('./content/third-async.txt',`Here is what the method do: ${first}, ${second}`,(*err*,*result*)=>{

            if(*err*){

                console.log(*err*)

                return null;

            }

            console.log(*result*)

        })

    })

})

**SYNCHRONOUS VS ASYNCHRONOUS**

Agora vamos analisar um pouco da sincronicidade e da assíncronicidade dos códigos feitos anteriormente abra o arquivo 10-fs-sync.js e adicione um console.log abaixo do require com ‘start’, agora adicione outros 2 ao final do código com ‘done with this task’, ’starting the next one’, e rode node 10-fs.sync.js. Note algo interessante o código começa termina e depois começa um novo processo, isso pode durar bastante tempo, imagine se tivéssemos dez ou mais usuários na aplicação e um usuário só faz uma dessas tarefas ou as duas isso demora muito tempo isso significa que o node não irá conseguir mexer com outros usuários eles ficariam na espera, e note também que o código é lido de forma síncrona linha por linha. Agora como uma alternativa vamos observar a forma assíncrona, vá ao arquivo 11-fs-async.js e adicione um console.log abaixo do require com ‘start’ em seguida vá até writeFile e no seu console.log(result) ele não está fazendo nada na aplicação então mude para ‘done with this task’ e no final do código adicione mais um console.log com ‘starting next one’, agora rode node 11-fs-async e note que a tarefa do código começa e logo ela descarrega passando para a próxima funcionalidade fazendo com que outros usuários possam usufruir simultaneamente do processo sem fazer com que ele demore, agora note que nossas callbacks estão meio bagunçadas alternativas para isso são usar das Promises ou do async/await.

10-fs-sync.js

//const {readFileSync,writeFileSync} = require('fs')

const *fs* = require('fs');

console.log('start')

//const first = readFileSync('./content/first.txt','utf-8')

const first = *fs*.readFileSync('./content/first.txt','utf-8')

//console.log(first)

//const second = readFileSync('./content/second.txt','utf-8')

const second = *fs*.readFileSync('./content/second.txt','utf-8')

//console.log(second)

*fs*.writeFileSync('./content/third-sync.txt',`Here is what the method do: ${first}, ${second}`,{flag:'a'})

console.log('done with this task')

console.log('start the next one')

11-fs-async.js

//const {readFile,writeFile} = require('fs')

const fs = require('fs')

console.log('start')

/\*

readFile('./content/first.txt','utf-8',(err,result)=>{

    if(err){

        console.log(err)

        return null;

    }

    //console.log(result)

    const first = result;

    readFile('./content/second.txt','utf-8',(err,result)=>{

        if(err){

            console.log(err)

            return null;

        }

        const second = result;

        writeFile('./content/third-async.txt',`Here is what the method do: ${first}, ${second}`,(err,result)=>{

            if(err){

                console.log(err)

                return null;

            }

            console.log(result)

        })

    })

})

\*/

fs.readFile('./content/first.txt','utf-8',(*err*,*result*)=>{

    if(err){

        console.log(err)

        return null;

    }

    //console.log(result)

    const first = *result*;

*fs*.readFile('./content/second.txt','utf-8',(*err*,*result*)=>{

        if(*err*){

            console.log(*err*)

            return null;

        }

        const second = *result*;

*fs*.writeFile('./content/third-async.txt',`Here is what the method do: ${first}, ${second}`,(*err*,*result*)=>{

            if(*err*){

                console.log(*err*)

                return null;

            }

            console.log('done with this task')

        })

    })

})

console.log('starting next task')

**HTTP**

Neste tópico iremos ir cobrindo conforme o código e a cada termo novo iremos explicando pois se não teríamos que explicar muitos termos e iria ficar um tópico imenso, esse modulo permite a gente fazer um web-server. Agora vamos montar as configurações para o módulo http, crie uma variável para guardar o pacote do modulo, const http = require (‘http’), neste modulo iremos criar um servidor web usando o método createServer, agora crie uma variável para usar o método e dentro do método ele busca uma callback, const server = http.createServer(()=>{}), e dentro da callback temos que passar dois parâmetros que são objetos o primeiro req que faz a requisição dos dados do cliente/pagina e o res que irá emitir a resposta de volta, também precisamos informar a porta que o servidor será colocado para isso usamos o método listen coloque-o abaixo do createServer, server.listen(5500), para mandar alguma coisa para o cliente/usuário precisamos usar o parâmetro res com o método write para escrever uma mensagem, res.write(‘Welcome to our home page!’), e para finalizar utilizamos o método end, res.end(), agora como temos um webserver o que eles fazem realmente é ouvir requerimentos, e você sempre quer que o seu server esteja online, então vá ao navegador digite localhost: número da porta, localhost:5500, e ai esta o que fizemos criamos o nosso primeiro webserver. Continuando no tópico por enquanto cada vez que alterarmos o código precisamos sair do node console, CTRL+C, vamos agora mostrar um pouco do parâmetro req que é um grande objeto cheio de propriedades, para observá-lo dê um console.log atualize seu navegador e observe o console, e quando navegamos de volta observamos o tamanho de propriedades e estamos procurando pela url, como estamos procurando pela pagina principal na pagina uma boa pratica que utilizamos no front-end e colocar ‘/’ como sua página principal então faça uma condição, if(req.url === ‘/’){res.end(‘Welcome to our home page.’)}, agora se quisermos adicionar outra pagina por exemplo adicionamos outra condição, if(req.url === ‘/about’){res.end(‘Here is the history about the page.’)}, agora caso o cliente/usuário quiser acessar uma página que não existe por exemplo podemos configurar um res.end com um html direto por isso não é certo é só um exemplo, sete um template string para utilizar um código html direto no node, res.end(`<h1>Ops!</h1><p>We can’t seen to find the page you are looking for!</p> <a href=”/”>Back Home</a>`) utilizamos aqui um h1 para o titulo um paragrafo explicando o que estava acontecendo e um link para retornar para a página principal.

12-http.js

const *http* = require('http')

const server = *http*.createServer((*req*,*res*)=>{

    if(*req*.url === '/'){

*res*.end('Welcome to our home page')

    }

    if(*req*.url === '/about'){

*res*.end('Here is the history about the page.')

    }

*res*.end(`

        <h1>Ops!</h2>

        <p>We can't seen to find the page you are looking for!</p>

        <a href="/">Back Home</a>

    `

    )

    //console.log(req)

    //res.write('Welcome to our home page')

    //res.end()

})

server.listen(5500)

**NPM**

Até esse ponto somos familiarizados com dois tipos de módulos os que a gente cria e o que o node proporciona mas agora vamos supor que queiramos criar barra lateral que muda de cor por exemplo, temos duas opções a primeira é criar ela do zero e a segunda é pesquisar no navegador e copiar e colar, porém sempre lembre que alguém já teve algum problema e compartilhou o código/projeto de alguma forma e a nossa único trabalho e adicionar com um comando no projeto, o npm ou node package manager proporciona isso porque quando instalamos o node ele vem junto, o npm nos proporciona três coisas a primeira e a reutilização dos nossos próprios códigos em diferentes projetos, utilizar o projeto de outras pessoas no nosso projeto e por último compartilhar as nossas soluções com outras pessoas/desenvolvedores. Para acessar o npm vá até <https://www.npmjs.com/> nele podemos encontrar de funções simples até frameworks ou projetos grandes. O npm chama o código reutilizável de pacote e um pacote e uma pasta que contém códigos com javascript no npm não tem controle de código qualquer um pode compartilhar e cabe a você filtrar pelos pacotes que são inúteis ou fracos para o seu projeto. Para observar a versão do npm digite no seu console npm –version, no npm temos dois tipos de instalações uma que permite instalar os pacotes globalmente isso significa que podemos usa-lo em qualquer projeto ou localmente que nos permite usar o pacote somente neste projeto. Para instalar um pacote localmente utilizamos o comando npm i/install <nomedopacote> e para adiciona-lo globalmente npm install -g <nomedopacote>. Agora que aprendemos tudo isso sobre o npm o que você acha que ainda falta para instalar o código, falta ainda falar sobre o package.json esse arquivo guarda todos os pacotes e informações sobre o projeto podemos cria-lo de forma manual ou de duas formas automática assim por dizer, neste curso não iremos mostrar a forma manual de criar o package.json por gastar muito tempo iremos logo direto para as formas automáticas, agora vá para o terminal e digite npm init, observe que agora ele dá passo a passo cada campo que você deve preencher ou não(só apertar enter), o primeiro package name se você não colocar o nome ele colocara o nome da pasta automaticamente, agora aperte enter a até o final e observe o arquivo package.json criado. Agora exclua esse package.json e use o outro comando automático npm init -y agora todas as propriedades foram criadas automaticamente sem o poder de escolha. Vamos para instalação do primeiro pacote com o proposito de instalação e observação não temos nenhum motivo para usar no projeto em si que estamos estudando no momento, npm install lodash, agora note que temos a propriedade dependency onde ficam todos os pacotes que foram e serão instalados, note também que foi criado uma pasta chamada node\_modules nela e onde ficam guardados todas as dependências instaladas e se você não tiver essa pasta o npm cria para você. Mas se você quiser instalar um pacote que é maior e que usa outras dependências você vai notar algo legal agora instale npm install bootstrap, como temos um pacote instalado o npm instala as dependências junto no node\_modules separando-as pelas pastas e se pasta tem outras dependências ela será colocada no node\_modules, agora vamos testar o uso do pacote que instalamos primeiro crie uma variável para requirir o pacote usado, const a = require (‘lodash’) depois crie uma variável para guardar um array dentro de um array isso quatro vezes, const items = [1,[2,[3,[4]]]], agora crie uma nova variável e use a variável requerida do lodash chamando um método para colocar todos os números visíveis em um único array, const newItems = a.flattenDeep(items) e depois de um console.log na variável newItems, rode o código node app.js e observe o terminal.

13-npm.js

//npm - global command, comes with node

//npm -- version

//local dependency - use it only is this particular project

//npm i <packagename>

//global dependency - use it in any project

//npm install -g <packagename>

//sudo install -g <packagename> (mac)

//package.json - manifest file (stores important info about project/package)

//manual approach (create.package.json in the root, create properties etc)

//npm init (step by step, press enter to skip)

//npm init -y (everything default)

const a = require('lodash')

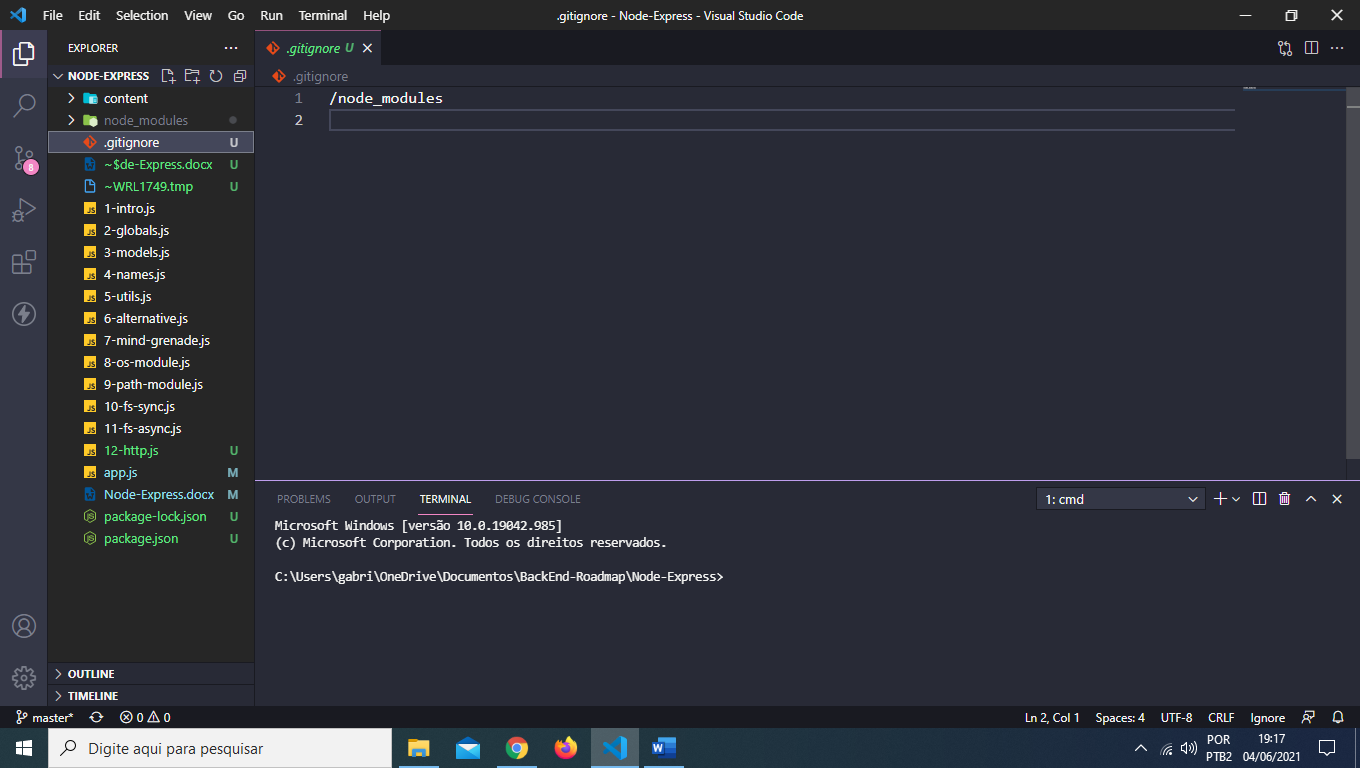
const items  = [1, [2,[3,[4]]]]

const newItems = a.flattenDeep(items)

console.log(newItems)

**SHARE CODE**

Utilizando o mesmo código passado agora vamos aprender um pouquinho como compartilhar o código usando o git/github, agora observe a pasta node\_modules você acha que quando você for compartilhar o código você também compartilhará essa pasta? Se a resposta for não está correto, pois esta pasta normalmente possui um tamanho muito grande então com o git temos uma forma para retirar essa pasta quando compartilharmos, crie um arquivo chamado .gitignore agora precisamos colocar o node\_modules dentro desse arquivo para fazer isso utilize o comando /node\_modules ou <qualqueroutroarquivo> para ignora-lo. Agora por exemplo como a pessoa irá usar o código compartilhado se não passamos o node\_modules pro repositório para isso se o projeto tem um arquivo package.json só precisamos usar o comando npm install que o node mesmo instalará as dependências necessárias faltando.



**NODEMON**

Agora que estamos familiarizados com o package.json e com o npm iremos instalar um pacote para a continuação do curso, o nodemon irá olhar os nossos arquivos e toda vez que atualizarmos ou mudarmos algo no código ele irá reiniciar automaticamente isso facilitará e muito a nossa vida pois não iremos ter que ficar fechando e abrindo o console novamente. Instale agora com o comando npm install nodemon -D isso fará que ele seja instalado de forma devDependency, isto é que isso só será usado na maquina local para testes e desenvolvimento. Agora temos que entender um pouquinho sobre o scripts em package.json nele podemos colocar comandos para testar/iniciar/terminar uma aplicação por exemplo, vá em scripts em package.json e adicione “start”: ”node app.js” e quando você usar o comando npm start no terminal ele já abrirá o código, em alguns você tem que provar o valor completo e para fazer isso adicione o comando “dev”: ”nodemon app.js” para executá-lo basta utilizar o comando npm run <nomedocomando> no caso dev isso iniciará o nodemon. Para desinstalar um pacote do npm usamos o comando npm uninstall <nomedopacote> tem outra forma de desinstalar um pacote de forma mais agressiva exclua seu package-lock.json, depois vá ao package.json e delete o pacote que deseja retirar depois e só ir no terminal e adicionar o comando npm install isso fará com que só os pacotes do package.json sejam instalados. Para instalar um pacote globalmente precisamos usar um comando npm install -g <nomedopacote> isso fará com que você possa usar o pacote em qualquer projeto. Sempre criando um projeto as vezes podemos ter problemas por isso sempre evite usar um pacote globalmente procure instalá-los de forma que sejam dependências só do projeto. Outra forma de instalar pacotes pelo npm foi o npx que permite instalar um pacote completo sem ele ser globalmente.

**PACKAGE-LOCK.JSON**

Agora o package-lock.json o porque dele e o por que precisamos dele, dependências possuem versões e algumas possuem dependências da dependências, quando você compartilhar o projeto você quer que as versões sejam as mesmas se não podem ocorrer alguns tipos de bugs ou a aplicação pode quebrar. Se você ficar interessado em mais informações sobre o package.json <https://nodejs.dev/learn/the-package-lock-json-file>.

**UPCOMING TOPICS**

Agora que aprendemos o npm no restante do curso iremos usar o nodemon e parar de entrar no terminal e digitar o comando node app.js os tópicos a seguir são importantes para a criação de um servidor web:

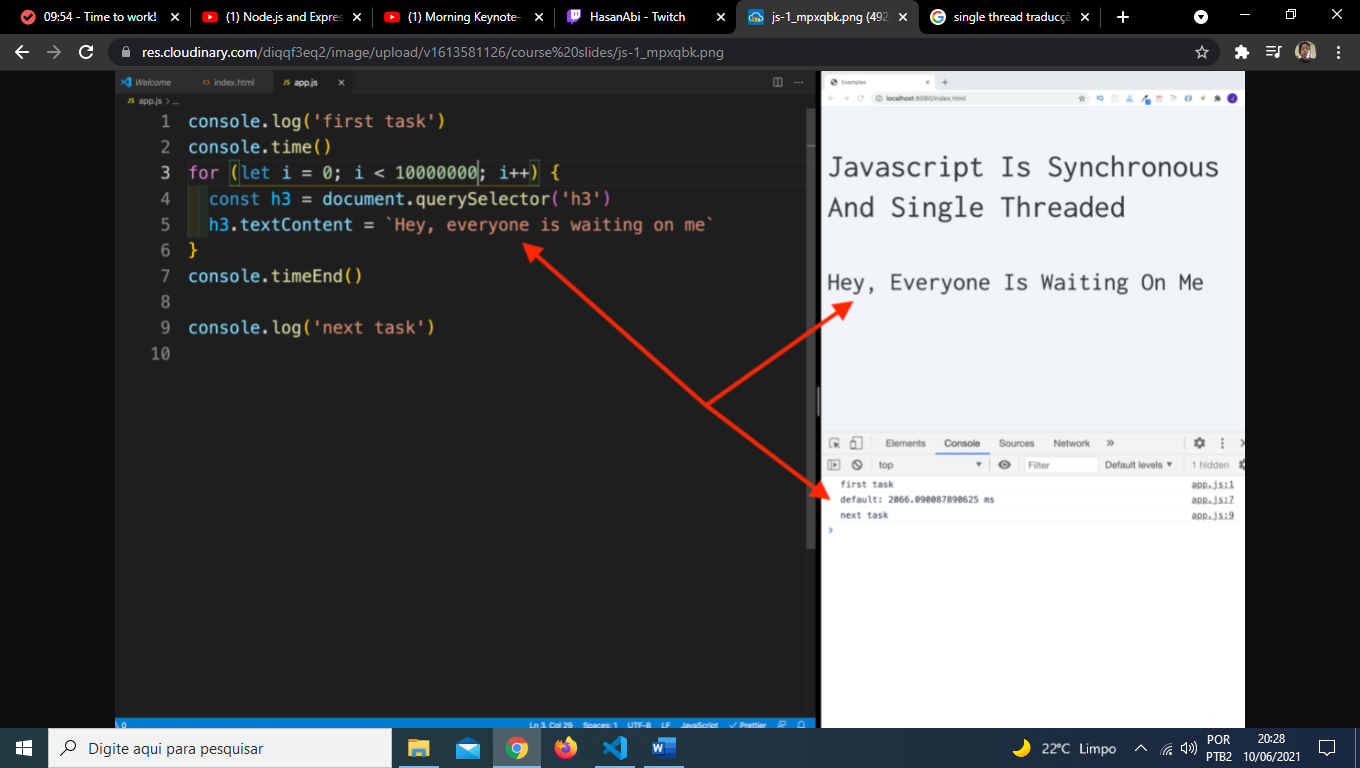
✦ Event Loop, Async Patterns, Events Emitter and Streams

✦ Main Concepts

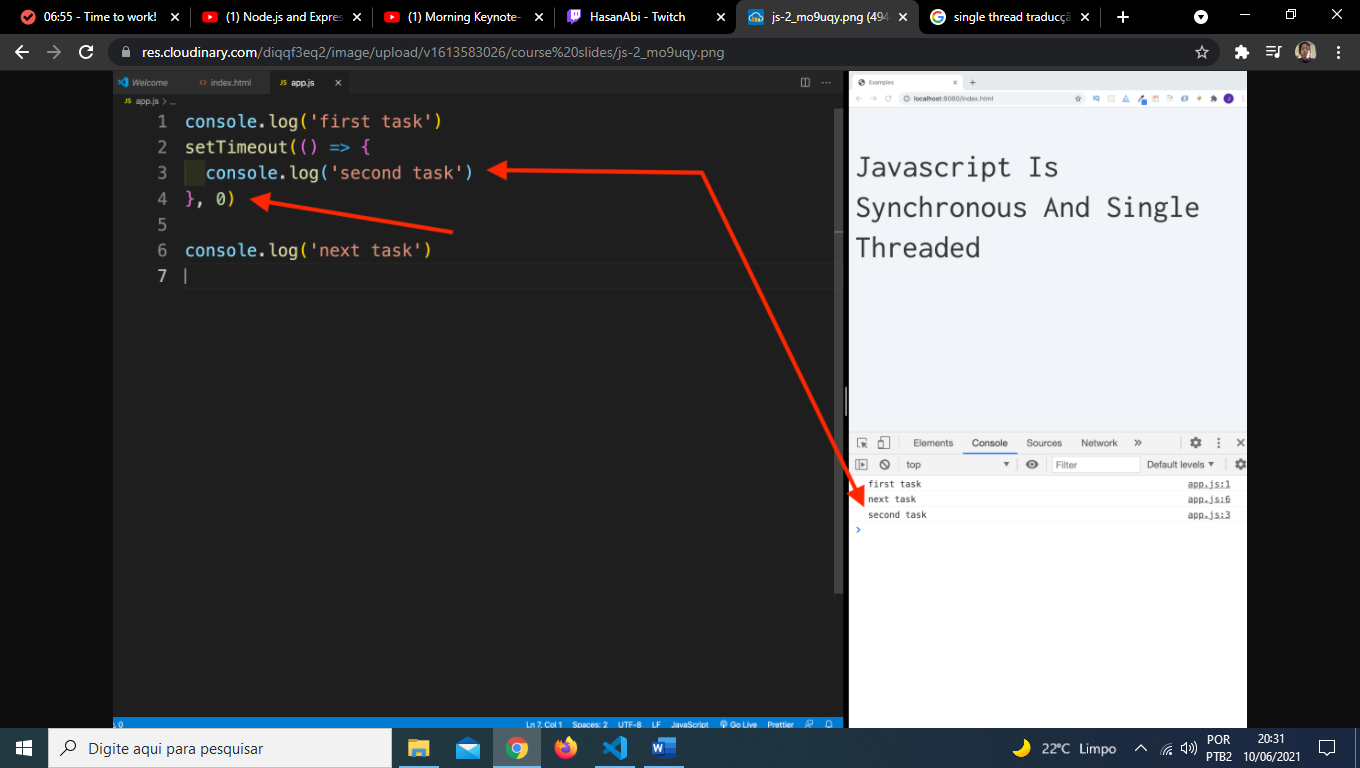
✦ Pre-Built Code

**EVENT LOOP**

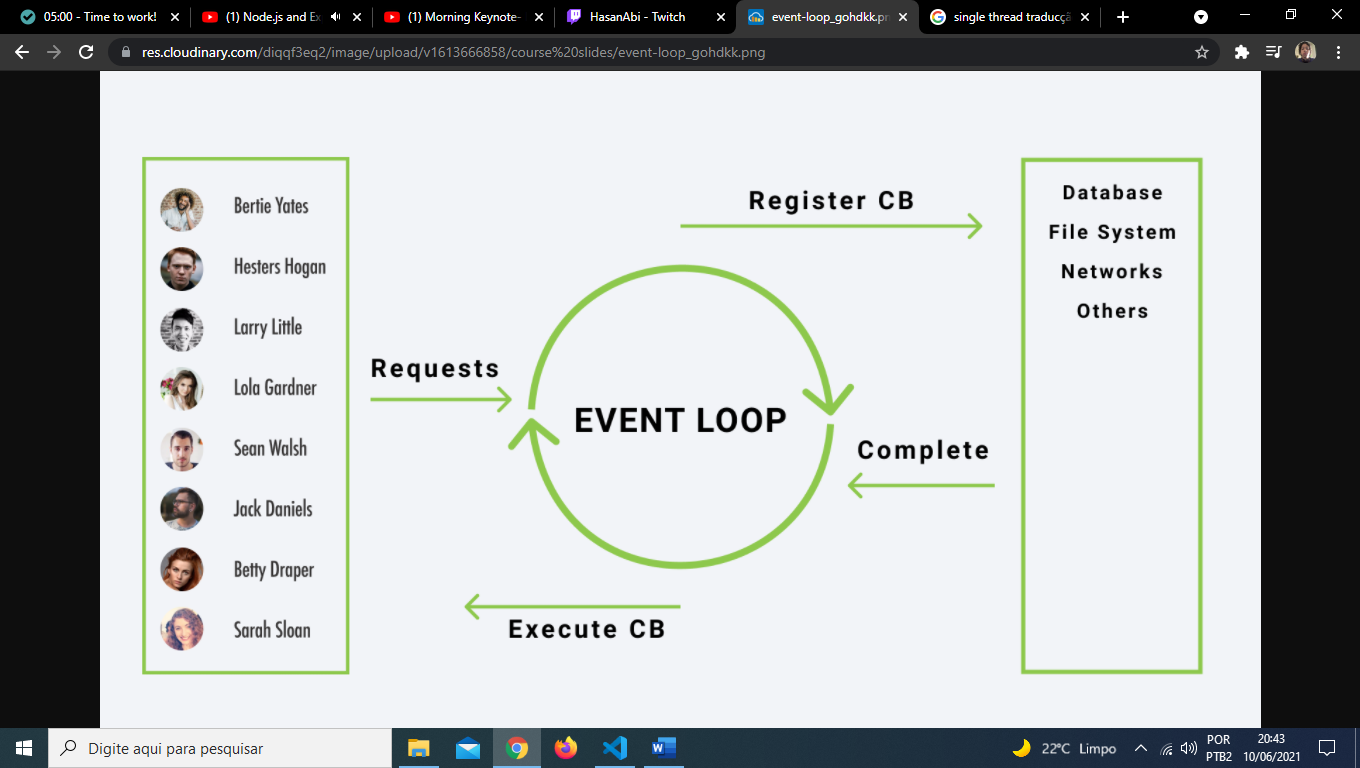
Event Loop em node.js é um tópico muito grande para ser abordado por completo então vamos ver só os conceitos básicos necessários. O event loop é o que permite o node.js produzir não bloqueáveis entradas/saídas de uma operação. Para procurar mais documentações vá ao seu navegador e digite node js event loop e o mais útil que eu acho seria esse [https://nodejs.dev/learn/the-nodejs-event-loop](https://nodejs.dev/learn/the-nodejs-event-loop%20) e quando se trata de vídeos digite no youtube event loop esse aqui é o mais completo [https://youtu.be/PNa9OMajw9w](https://youtu.be/PNa9OMajw9w%20) , agora acesse site para observar um slide sobre o loop-event em <course-api.com> em slides. Javascript é uma linguagem síncrona e de único fio quando afirmamos isso o javascript lê o código passo a passo e sempre quando uma tarefa é iniciada ela só pode começar outra quando finalizada.



Também quando adicionamos uma callback numa função mesmo por exemplo um setTimeout com 0 segundos de delay ela só irá rodar depois da próxima tarefa. E isso faz com que trazemos para o node.js o event loop.

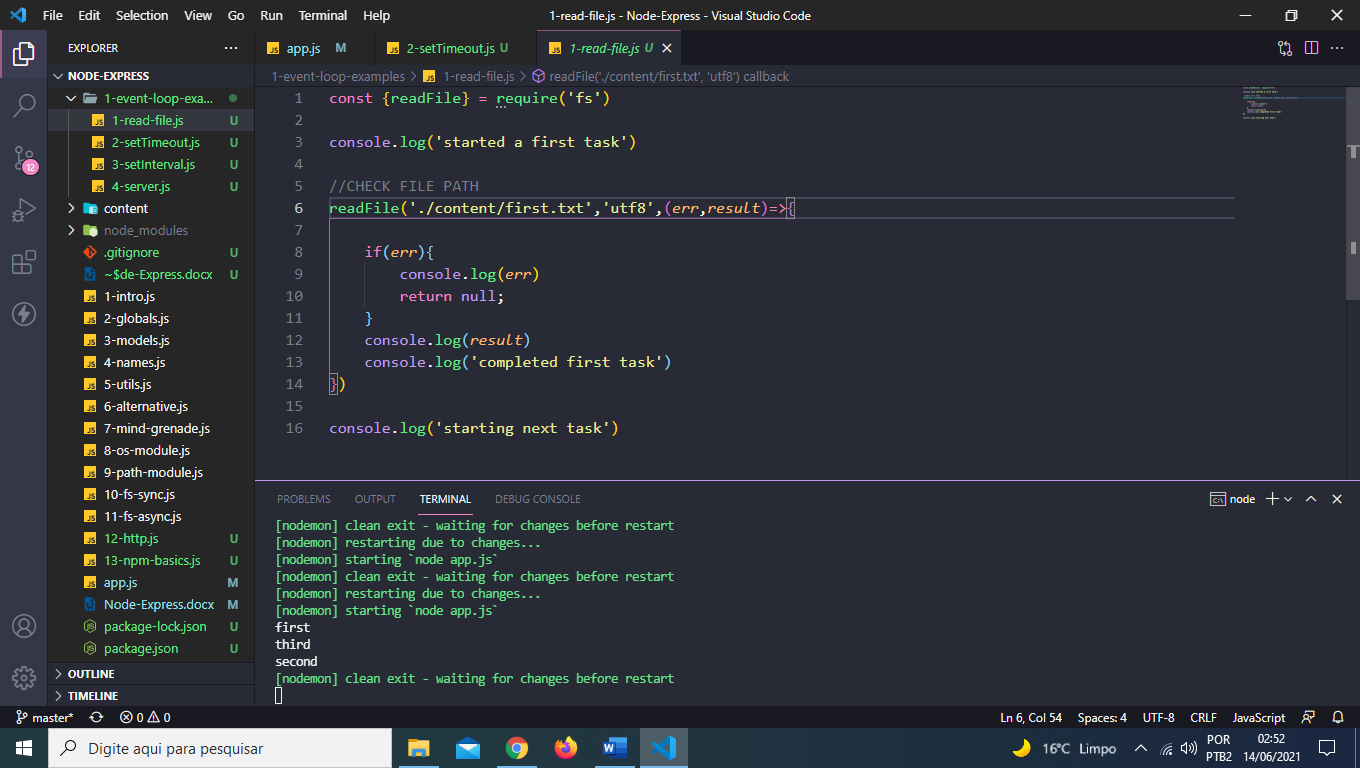


Vamos agora imaginar um cenário onde temos uma aplicação com usuários e eles estão entendidos então pedem requisições para a aplicação e cada vez que uma requisição chega o event loop é necessário para evitar esse tipo de cenário por exemplo vamos supor que um dos usuários tem uma tarefa demorada ou muito grande que requer tempo então nesse caso o event loop registra essa callback agora se o event loop não fizesse isso iriamos ter esse tipo de cenário como ele/usuário está requerendo uma tarefa todos os outros usuários teriam que esperar até que ele termine o processo para fazer sua requisição e o que event loop faz e registra a callback e só quando a tarefa é completada ela executa a callback, voltando um pouquinho no slide passado se adicionarmos um monte de console.log depois de ‘next task’ ele mostraria todos os console.log antes de ’second task’ independente do tempo no setTimeout. O event loop é o nosso amigo pois com a ajuda dele podemos descarregar essas tarefas de grande demora e deixar os nossos usuários felizes.

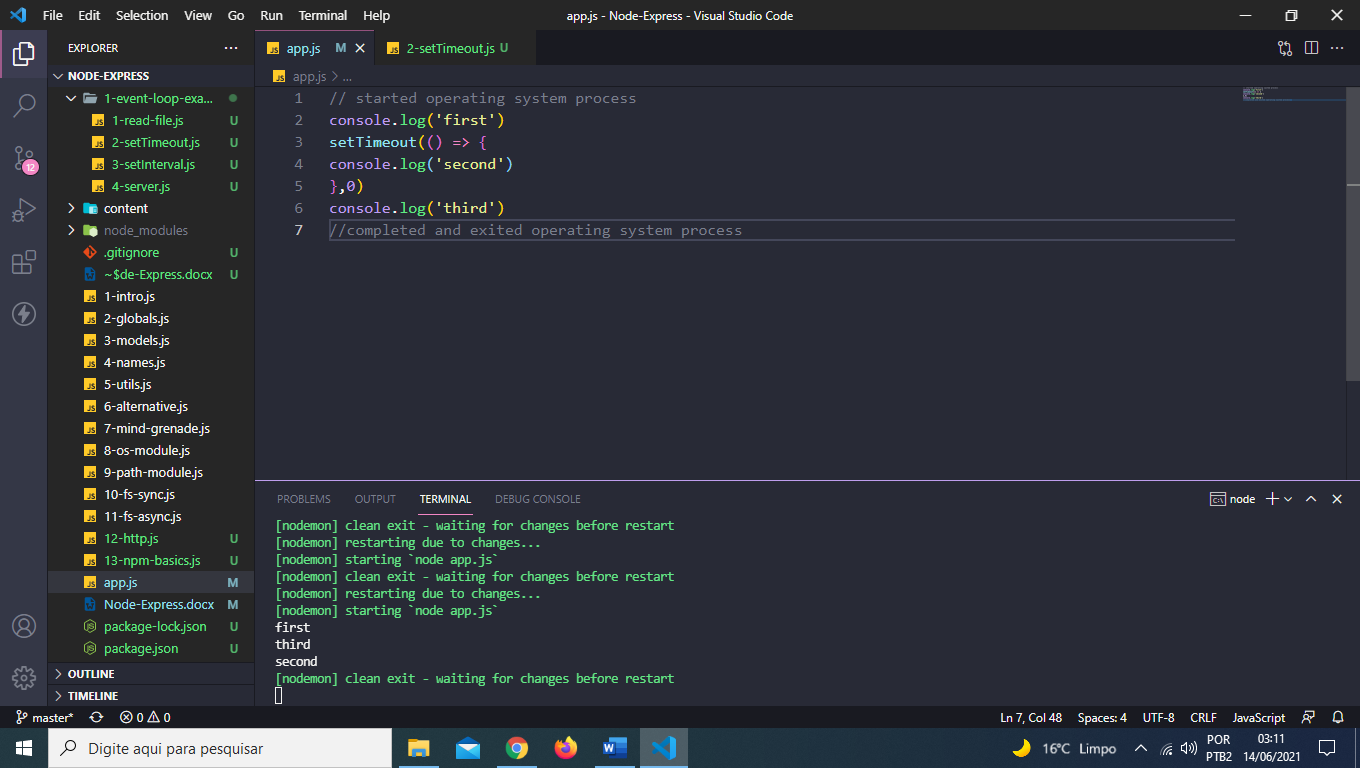


**EVENT LOOP CODE EXAMPLE**

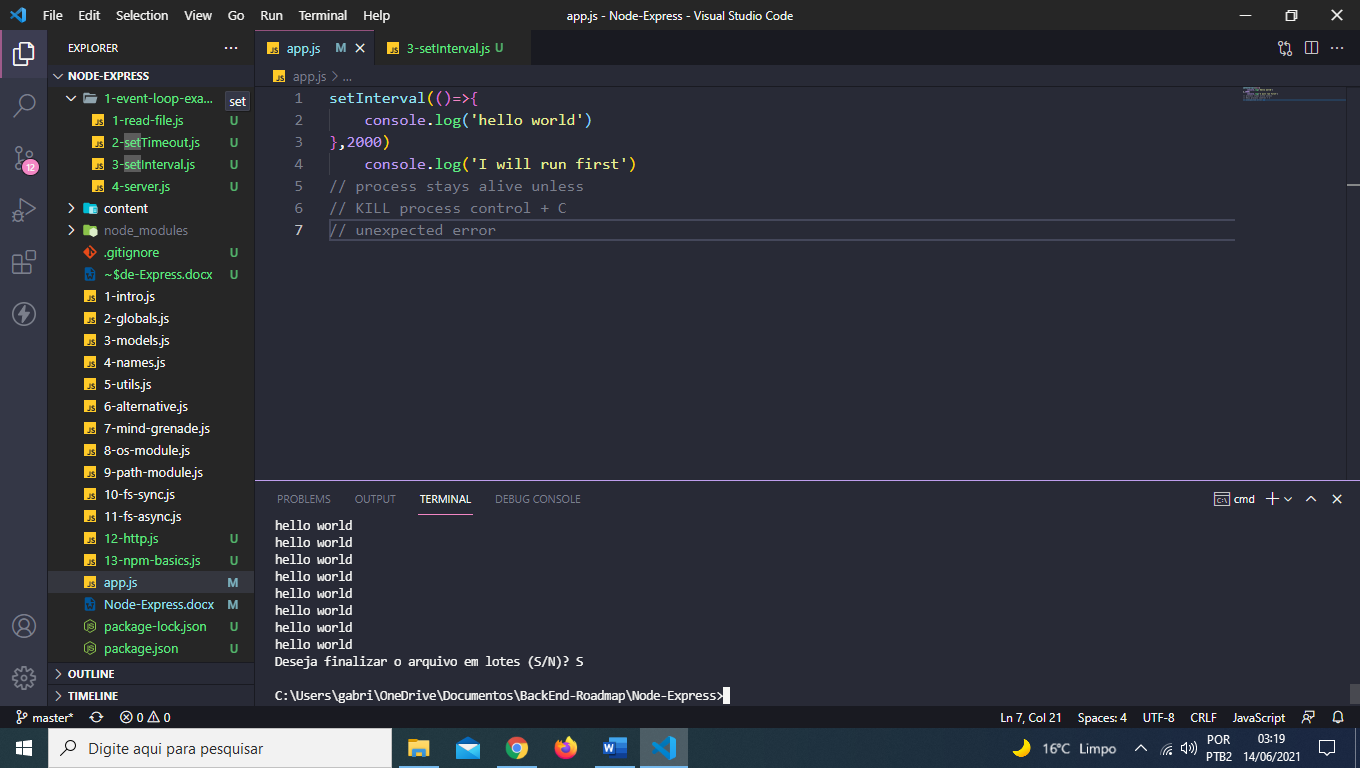
Tudo certo agora que aprendemos na teoria como funciona o event loop iremos fazer um na pratica para fixar mais na mente. O primeiro exemplo que vamos apresentar você deve ser familiar com ele pois é uma simples aplicação assíncrona do método readFile. Agora observe o código nota-se que primeiro criamos um console.log para iniciar a tarefa depois ela começa outra tarefa, mas antes dentro da callback ela mostra o resultado e completa, isso acontece, pois, a função é assíncrona e sabemos que o event loop irá descarregar.



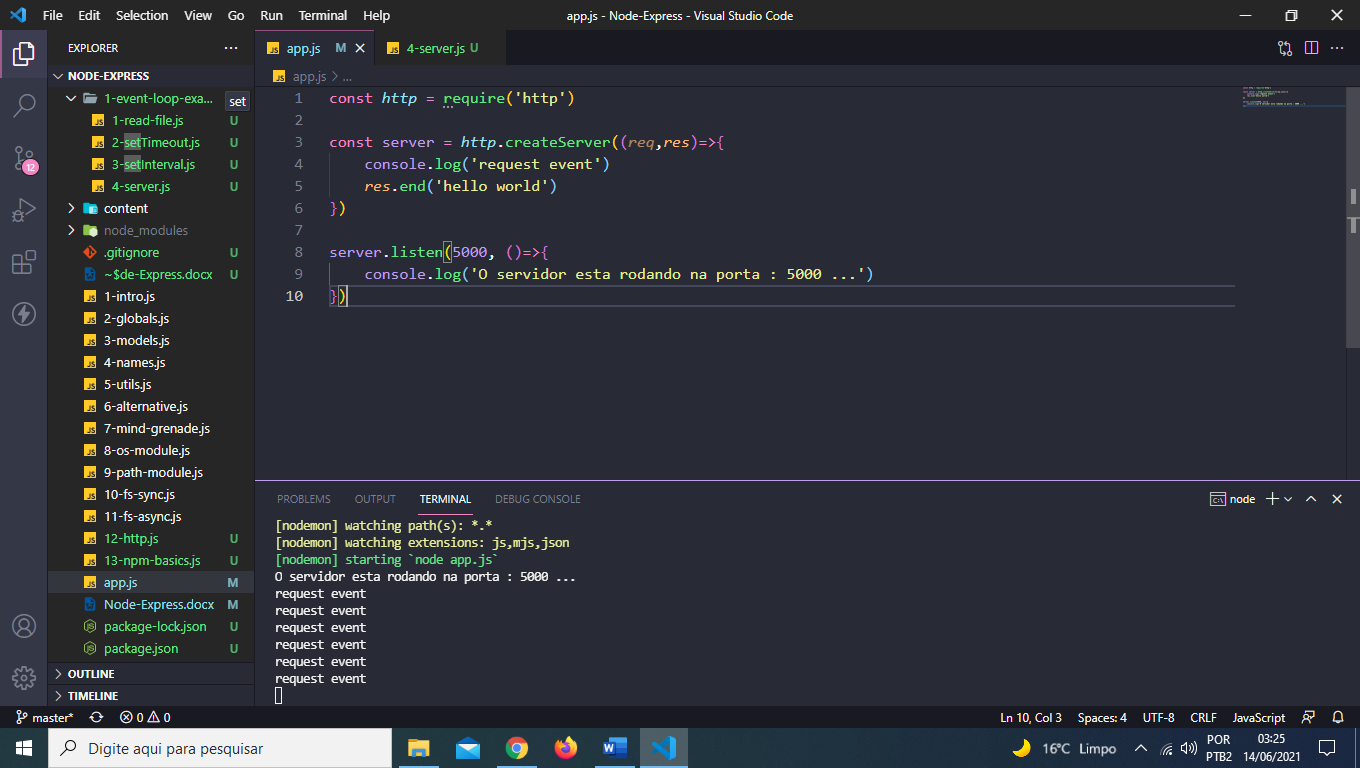
O próximo exemplo será um setTimeout primeiro crie um console.log informando a primeira tarefa depois crie um setTimeout com o tempo 0 e com um console.log dentro informando a segunda tarefa e de fora um terceiro console.log informando o inicio da terceira tarefa. Agora observe como colocamos o tempo do setTimeout para 0 você deve achar que ele irá de em ordem, porém está errado no resultado temos primeiro, terceiro, segundo isso porque a função setTimeout é assíncrona fazendo com que ela seja descarregada ativando o event loop, observe que coloquei comentários esses comentários são para mostrar o início da operação no sistema e a finalização e saída do processo de operação.



O próximo exemplo com o setInterval é um pouquinho diferente, o que eu vejo no terminal de código é o ‘I will run first’ e em seguida um hello world a cada 2s e note que não estamos terminando o processo como fizemos no exemplo passado e não estamos fazendo isso por setInterval é assíncrono o único jeito de terminar o processo e matando o terminal ou usando o comando ctrl + C ou ocorrerá um erro inesperado de qualquer outra forma irá sempre se manter vivo.

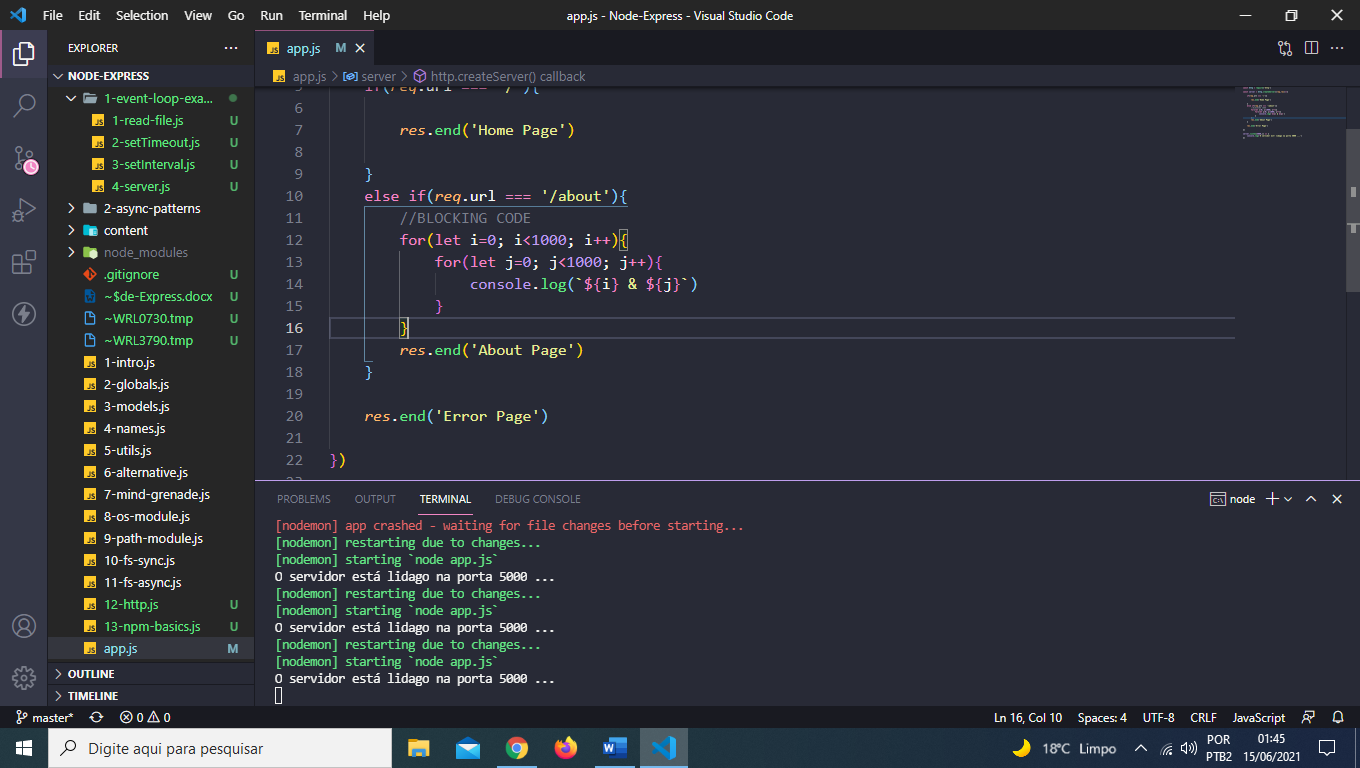


E o último exemplo é um servidor mostrando como ele se mantem vivo na hora de criar o servidor dentro da callback coloque um console.log para requisitar algo por exemplo e no res.end para a página digite um ‘hello word’ em seguida observe que quando entramos ele mostrará no terminal primeiro o caminho pela porta de entrada e toda vez que atualizarmos ou mudarmos o request event dentro da função assíncrona de criar o servidor será descarrega criando um event loop.



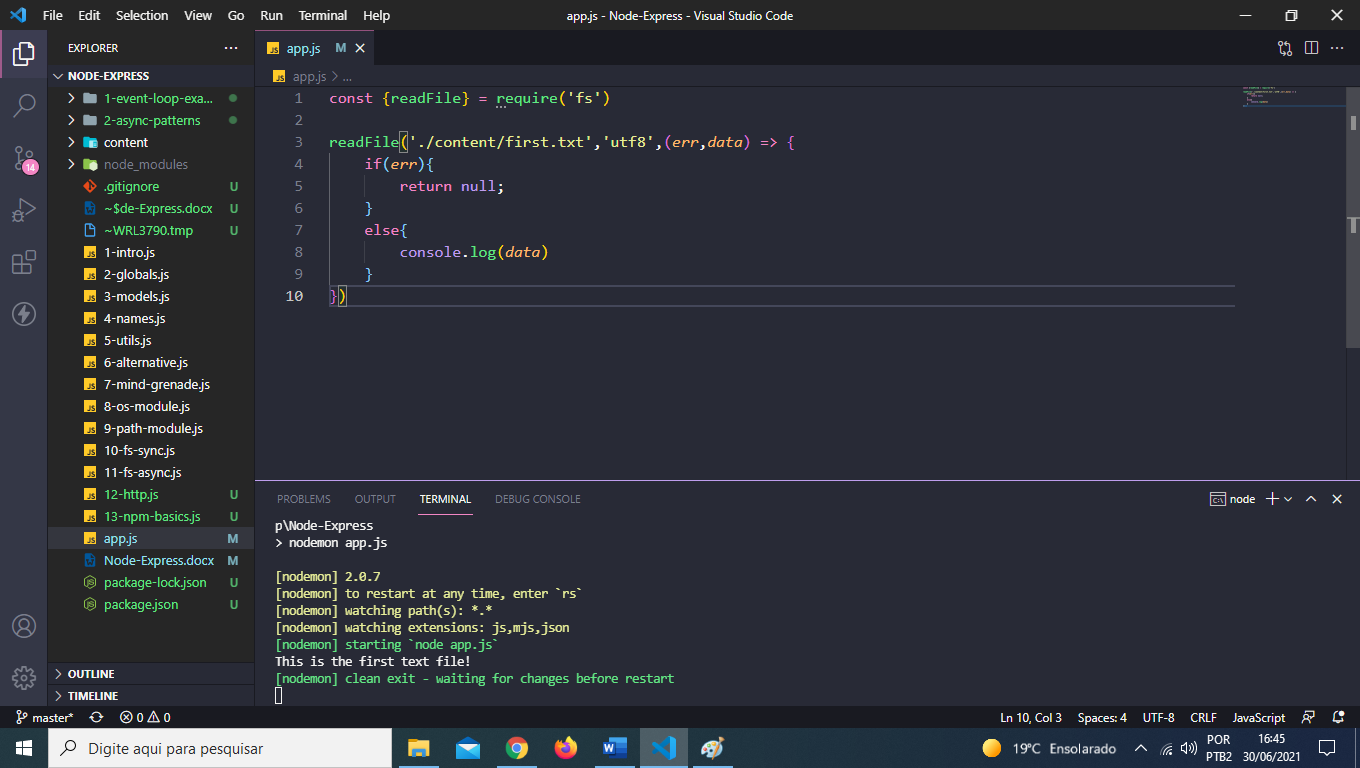
**ASYNC PATTERNS – BLOCKING CODE**

Então se relembrarmos os dois tipos de padrões que são síncronos e assíncronos, onde o assíncrono e bom porque não bloqueia o event loop, o problema é se usarmos essa abordagem com as callbacks o seu código ficará bagunçado bem rápido e você pode acabar se perdendo então nos próximos tópicos mostraremos alternativas para não bagunçar os eu código com uma callback dentro da outra. Primeiro pegue o servidor que criamos no arquivo 12-http.js coloque no app e em seguida inicie o nodemon com as duas abas para a home e a about page agora adicione um código de bloqueio na página about para fazer isso adicione um laço de repetição dentro de um laço de repetição for (let i=0; i<1000; i++) {for (let j=0; j<1000; j++) {console.log(`${i} & ${j}`)}} em seguida recarregue as duas páginas e observe no seu terminal que as duas páginas estão recarregando e elas só irão renderizar quando um serviço for terminado observe que com isso temos um código síncrono que demora um longo tempo para terminar e assim tornando uma boa pratica sempre deixar seu código de forma assíncrona.

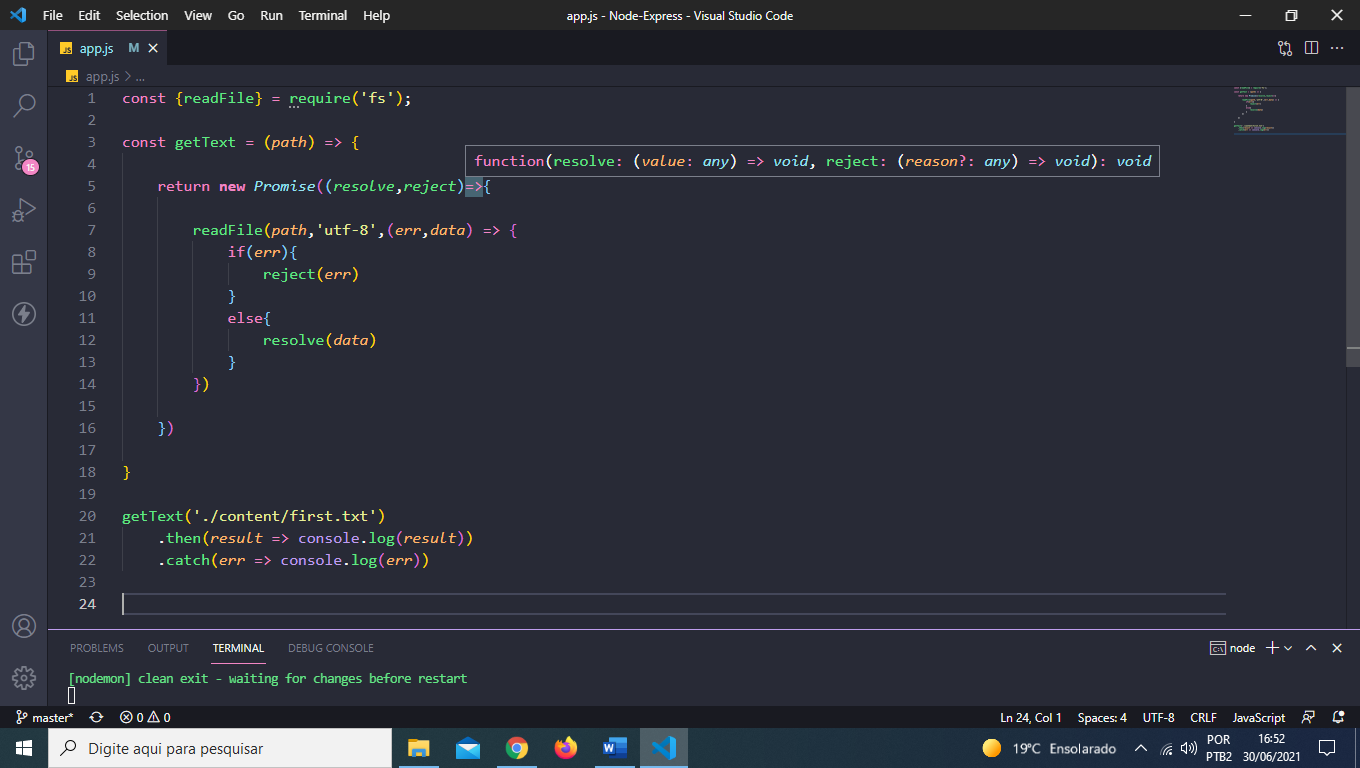


**ASYNC PATTERNS – SETUP PROMISES**

Para entender iremos refazer o setup do readFile que fizemos anteriormente, usando o nodemon veremos o que está dentro do conteúdo first.txt agora teremos um problema se performarmos múltiplas ações ao mesmo tempo, por exemplo ler dois arquivos a melhor solução no momento seria transformar isso em uma Promise, e eventualmente iremos adicionar o async/await isso só servirá para mostrar o que está acontecendo por trás das cenas.



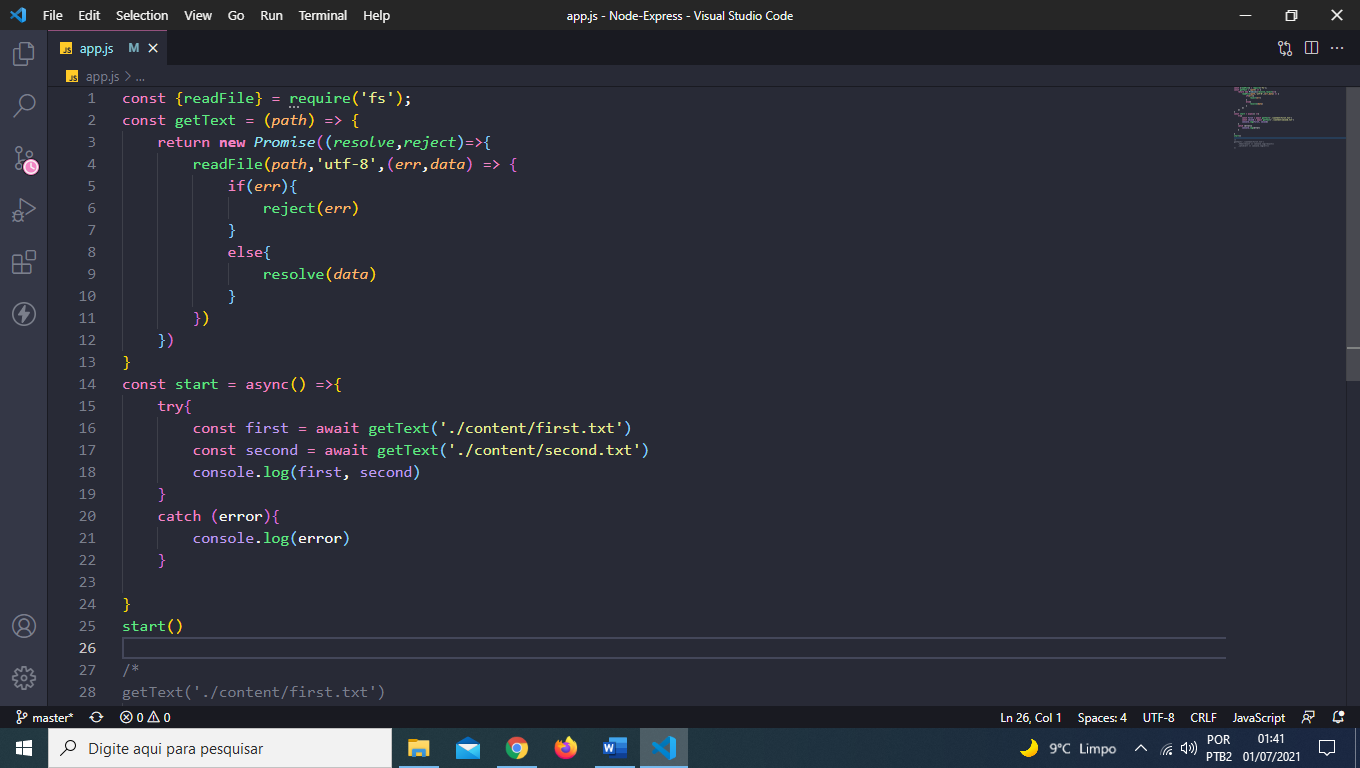
Mudando agora para uma forma de Promise ficará assim:



Se mudarmos a rota na função getText por exemplo adicionando mais um ponto ele irá mostrar o erro no terminal agora no próximo tópico iremos refatorar esse exemplo para async.

**ASYNC PATTERNS – REFACTOR TO ASYNC**

Depois da Promise estar feita temos uma maneira de deixá-la assíncrona para isso exatamente nesse exemplo precisamos criar uma função do zero, const start = async () => {}, quando fazemos isso para boas praticas sempre procure colocar em um try/catch, agora coloque as variáveis passadas a first e adicione uma segunda sempre utilizando o artificio do await para deixar essa função assíncrona e primeiro procuramos pelo caminho chamando a função getText, const first = getText(‘./contente/first.txt’) e depois adicione o await antes da função const first = await getText(‘./contente/first.txt’), agora comente a invocação do getText passado e invoque a função start, start() , agora dê um console.log() em first e note que antes tínhamos um amontoado de código agora ele ficou bem mais simples. Adicione um segundo valor para ler como fizemos no primeiro const second = await getText(‘./contente/second.txt’), no primeiro exercício do modulo fs queríamos ler os valores e escreve-los, o problema e que não passamos uma função de escrever e se fizermos isso agora ela terá que ser adicionada na Promise e embrulha-la. No próximo modulo iremos usar o modulo para desembrulhar uma Promise e deixar ela a mais simples possível.



**ASYNC PATTERNS – NODE’S NATIVE OPTION**

Tudo certo a gente tem a nossa função start ela é assíncrona estamos esperando as Promises se resolverem e podemos ver que ela já está bem mais limpa, agora como podemos colocar esse código sem o código se estender tanto, em node temos um modulo chamado útil, importe o modulo, const util = require(‘util’), esse modulo tem um método dentro dele chamado promisify e com uso dessa função podemos pegar esse readFile que está buscando uma callback e tornar em função que retorna a Promise. Agora como colocaremos isso crie uma variável com o mesmo nome de readFile e adicione Promise na frente, const readFilePromise = util.promisify(readFile), observe que temos que passar a função readFile para a promisify. Agora temos um erro pois getText não foi definido para isso e como também queremos escrever um arquivo passamos também para o modulo fs o método writeFile e também criamos uma variável para guardar o util, const {readFile, writeFile} = require('fs'); e const writeFilePromise = util.promisify(writeFile); em seguida passe a variável readFilePromise que criamos no lugar de getText e também adicione o encode logo após o caminho, ‘utf-8’ e como passamos na linha 5 writeFile no promisify o que nos podemos fazer é escrever o arquivo logo após ler agora escreva await e não associe com uma variável estamos obtendo um dado indefinido nós passamos agora a variável para escrever do promisify com o caminho e logo após uma template string para escrever no documento, await writeFilePromise (‘./contente/result-mind-grenade.txt’, `THIS IS AWESOME: ${first}, ${second}`) observe que criamos um arquivo e também conseguimos escrever dentro dele outra coisa que também e legal e que podemos pular também a parte do util modulo adicionando .promises no final do modulo fs e depois disso mudar os nomes das variável de novo adicionando um terceiro argumento no writeFile podemos duplicar o que foi escrito.

const {readFile, writeFile} = require('fs').*promises*;

/\*

const util = require('util')

const readFilePromise = util.promisify(readFile);

const writeFilePromise = util.promisify(writeFile);

\*/

const start = async() =>{

    try{

        const first = await readFile('./content/first.txt','utf-8')

        const second = await readFile('./content/second.txt','utf-8')

        await writeFile (

            './content/result-mind-grenade.txt',

            `THIS IS AWESOME: ${first},${second}`,

            {flag:'a'}

        )

        console.log(first, second)

    }

    catch (error){

        console.log(error)

    }

}

start()

/\*

const getText = (path) => {

    return new Promise((resolve,reject)=>{

        readFile(path,'utf-8',(err,data) => {

            if(err){

                reject(err)

            }

            else{

                resolve(data)

            }

        })

    })

}

\*/

/\*

getText('./content/first.txt')

    .then(result => console.log(result))

    .catch(err => console.log(err))

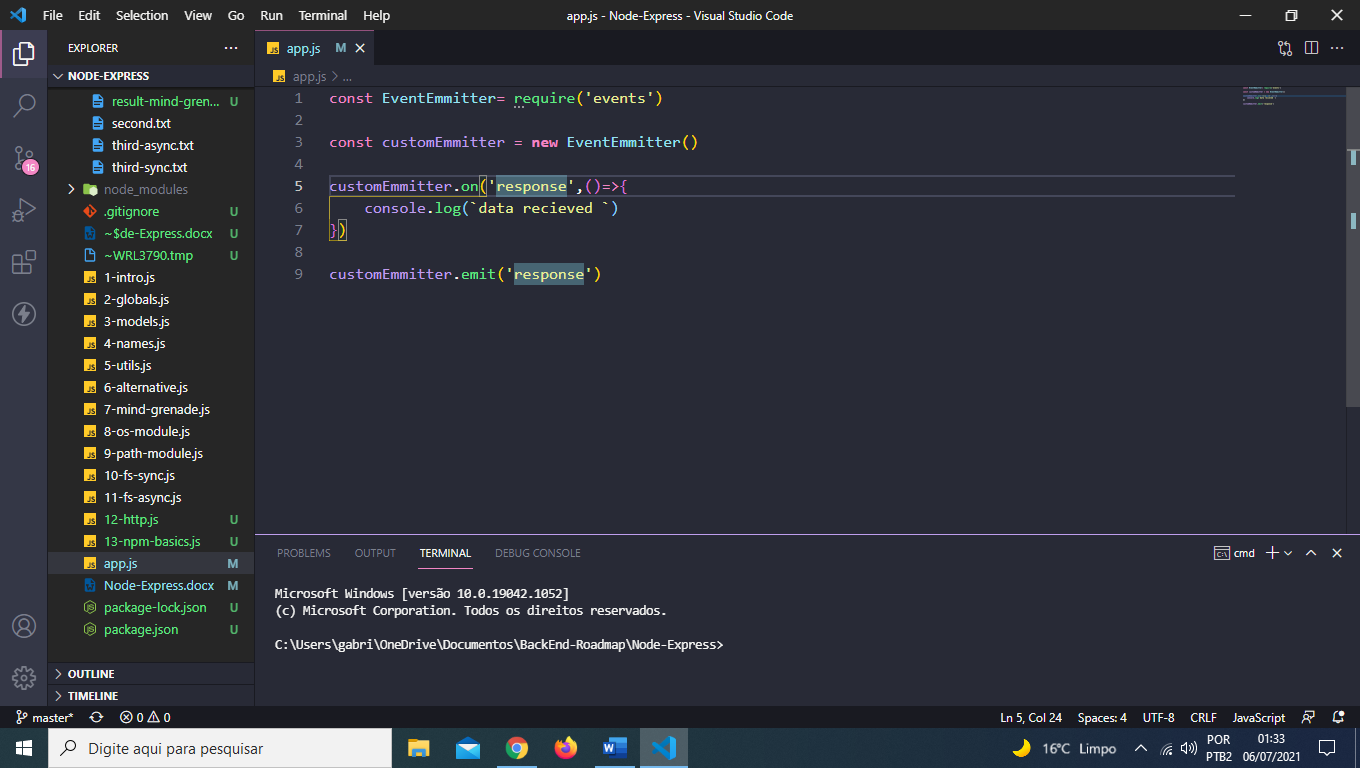
\*/

**EVENT EMMITERS**

Quando trabalhamos em projetos web browser em javascript uma das grandes coisas que lidamos são eventos por exemplo se um usuário clica em um botão e no programa nos lidamos com isso espero que tenha entendido o ponto principal que em parte do programa ele é controlado por eventos. Os exemplos iniciais são bem simples porem durante o curso ou exemplos que você encontrar esses eventos serão mais complicados. Agora vamos começar a fazer o primeiro exemplo.

**EVENT EMMITTER – CODE EXAMPLE**

Primeiro crie uma variável com o nome EventEmmitter requerendo o módulo events tornando a variável em uma classe e neste ponto temos duas opções, se você quer criar um evento customizado você precisa entender a classe ou você só quer transmitir um evento e ouvir ele então você simplesmente só cria uma instancia iremos com a segunda rota então crie uma variável customEmmiter ou qualquer nome que você quiser, e adicione um new atribuindo um novo nome a variável/função e a invoque-a. Esse objeto tem muitos métodos porem iremos abordar só dois eles são on e emit, on vai ouvir um evento especifico e emit irá transmitir o evento. E o jeito que configuramos é chamando a variável customEmmiter e depois. .on () agora como estamos fazendo uma configuração básica passe um argumento string ‘response’ e depois passe uma callback function dentro da callback dê um console.log() só para a observação no jogo `data recieved` e uma vez como o cenário foi criado precisamos passar o emissor então customEmmiter.emit() passando o mesmo argumento que esta no primeiro no caso response.



**EVENT EMMITTER – ADDITIONAL INFORMATION**

Primeiro podemos ter quantos métodos forem necessários dentro de um evento como por exemplo temos o mesmo evento porem dentro da callback passamos qualquer outra logica diferente da primeira, outro ponto é que ordem do on e emit interessa e muda pode ocorrer erros se não estiver nesta ordem por exemplo colocar o emit acima do on no caso ele não irá mostrar nada no console pois o evento veio depois da resposta, e o terceiro ponto e que podemos passar argumentos quando estamos omitindo o evento, por exemplo no emit adicionamos um nome no caso ‘john’ e também um número, então uma string e um número na sua callback você pode acessar esses argumentos como parâmetros por exemplo adicione nome e id e passe eles para console.log dentro da template string observe que os valores são emitidos normalmente.

const EventEmmitter= require('events')

const customEmmitter = **new** EventEmmitter()

customEmmitter.on('response',(*name*,*id*)=>{

    console.log(`data recieved`)

    console.log(`id: ${*id*}, nome: ${*name*};`)

})

customEmmitter.on('response',()=>{

    console.log(`some other logic here`)

})

customEmmitter.emit('response', 'john', 34)

**EVENT EMMITTER – HTTP MODULE EXAMPLE**

Primeiro pegue o setup do arquivo 14-request-event.js no endereço <https://github.com/john-smilga/node-express-course/tree/main/01-node-tutorial> temos outro modo de fazer um server em node usando a requisição de evento que aprendemos, da forma antiga usamos só os métodos requisitados pelo módulo http e passamos uma callback para fazer a requisição e a resposta, para criar usando a forma de eventos precisamos usar do método on e passar uma callback com uma classe fornecida pelo node para checar se é uma classe vá para [https://nodejs.org/api/http.html#http\_event\_request](https://nodejs.org/api/http.html%23http_event_request%20) e uma callback com os parâmetros req e res igual ao que usamos na outra forma e dentro da callback um res.end e pronto está feito o seu server com eventos para checar procure em localhost o número que você utilizou em server.listen().

const *http* = require('http')

// const server = http.createServer((req, res) => {

//   res.end('Welcome')

// })

// Using Event Emitter API

const server = *http*.createServer()

// emits request event

// subcribe to it / listen for it / respond to it

server.on('request', (*req*, *res*) => {

*res*.end('Welcome')

})

server.listen(5000)

**STREAMS**

Em node.js streams são usadas para ler ou escrever sequencialmente basicamente quando temos que manipular dados contínuos por exemplo continuous source ou big file, streams vem pra facilitar e existem até o momento quatro tipo de streams:

✦ Writable: usado para escrever dados sequencialmente;

✦Readable: usado para ler dados sequencialmente;

✦Duplex: usado para ler e escrever dados sequencialmente;

✦Transform: quando os dados podem ser modificados tanto lendo ou escrevendo;

Como em eventos o node fornece módulos que implementam a interface de streaming e uma coisa interessante é que streams extend event emitters class o que significa que podemos usar os eventos em streams.

**STREAMS - READ FILE**

Agora vamos aprender como usar o streams para ler arquivos, mas primeiro precisamos entender uma coisa quando usamos o método síncrono ou assíncrono lemos todo o arquivo mas caso temos um grande arquivo guardar ele em uma variável pode causar um erro então usando o método streams podemos ler esses grandes arquivos agora para o primeiro passo crie um arquivo js usando o modulo fs com um loop:

const {writeFileSync} = require('fs')

for(let i = 0; i < 10000; i++){

    writeFileSync('./content/big.txt',`hello word ${i} \n`,{ flag:'a' })

}

Agora execute o node, pronto o arquivo está na pasta crie um arquivo agora para ler a streams crie o modulo fs e depois procure pelo método createReadStream em seguida crie uma variável e dentro dela coloque o caminho do grande arquivo, em seguida vamos usar o event on procurando pelo argumento data e colocando uma callback logo após dentro dessa callback passe um parâmetro e mostre o resultado em um console.log. Por enquanto estamos mostrando só o tamanho dos arquivos porem podemos fazer mais coisas com as streams agora vamos observar umas informações adicionais os arquivos terão um tamanho de 64kb padrão, porém podemos controla-lo para isso passamos um objeto no createReadStream com nome highWaterMark coloque ele para 90000 e observe no console que teremos somente dois console.log informando o tamanho também podemos colocar o encode para mostrar os arquivos, e também por ultimo quero mencionar que também temos o error event que podemos acessar.

const {createReadStream} = require('fs')

const stream = createReadStream('./content/big.txt', {highWaterMark: 90000 , /\*encoding: 'utf-8' \*/})

// default 64kb

// last buffer - remainder

// highWaterMark - control size

// const stream  = createReadStream('./content/big.txt', { highWaterMark:9000 })

// const stream  = createReadStream('./content/big.txt', {encoding: 'utf-8 })

stream.on('data',(*result*)=>{

  console.log(*result*)

})

stream.on('error',(*error*)=>{

  console.log(*error*)

})

**STREAMS – HTTP EXAMPLE**

Agora vá no arquivo de criação de um grande arquivo e adicione mais uns zeros no loop e depois crie um arquivo para fazer o exemplo em http primeiro crie duas variáveis para guardar os módulos http e fs e depois crie um servidor com os métodos ReadFileSync e na porta 5000 depois ligue o nodemon e abra o browser e depois entre no modo desenvolvedor (ctrl + i) depois vá em network e observe o tamanho do arquivo em headers ele mostrará ele completo observe também que provavelmente o browser estará um pouco pesado pois isso influencia também o usuário em headers content-lenght, agora usaremos com o createReadStream primeiro crie uma variável para o createReadStream passando o caminho e o tipo do unicode depois use o event on com a classe open e uma callback e um event on para um erro com um res.end para o erro, dentro do evento on para abrir porém dentro do open normalmente não usamos o res.end, agora chame a variável a variável criada para o createReadStream e use o método pipe passando o res isso fará com que o tamanho do arquivo seja separado em pedaços deixando o browser mais leve e diminuindo o tamanho do arquivo melhorando para o usuário.

let *http* = require('http')

let *fs* = require('fs')

*http*.createServer( (*req*,*res*)=>{

  //const text = fs.readFileSync('./content/big.txt','utf-8')

  //res.end(text)

  const filesStream = *fs*.createReadStream('./content/big.txt','utf-8')

  filesStream.on('open',()=>{

    filesStream.pipe(*res*)

  })

  filesStream.on('error',(*err*)=>{

*res*.end(*err*)

  })

})

.listen(5000)

**HTTP REQUEST/RESPONSE CYCLE**

Vamos aprender agora como funciona a transmissão de dados na web vamos fazer isso com ajuda de alguns slides em <https://res.cloudinary.com/diqqf3eq2/image/upload/v1613596625/course%20slides/http-messages_lugv8b.png>, toda vez que usamos o browser fazemos uma requisição para um servidor que é responsável para nos mostrar os recursos disponíveis e depois o servidor manda uma resposta para você podendo ter ou não este recurso. Esta forma não está usando protocolos http isso é chamado de mensagens http, onde o usuário manda uma requisição http e o servidor devolve uma resposta http e é assim que trocamos dados pela web.

**HTTP MESSAGES**

A estrutura genérica para ambas as mensagens é similar ambas possuem uma única linha de partida, ambas possuem headers opcionais, uma linha que indica todos os dados fora mandada e o headers e todo o meta dado quanto o corpo opcional. Requisições de mensagens são o que o usuário manda para o servidor e a response ou resposta será o que o servidor irá mandar de volta para o usuário e geralmente quando falamos de requisições de mensagem na primeira linha terá um método depois uma URL vamos ficar interessados em duas coisas normal elas serão o método e o URL a gente também possui um slide separado para os métodos http iremos cobri-los quando formos trabalhar no código. Quando falamos de métodos efetivamente estamos nos comunicando o que iremos fazer como por exemplo se eu quero obter os dados de um recurso utilizamos o método GET, e se quiser adicionar algo isso será um método POST, agora observe no slide <https://res.cloudinary.com/diqqf3eq2/image/upload/v1614137856/course%20slides/http-req-res_bslzni.png> porque ele está mostrando o método GET, porque ele é a requisição padrão do navegador, a URL seria o endereço da página, o Headers é essencialmente opcional onde fica a meta informação sobre a requisição, e também temos o corpo opcional que quando um usuário esta requisitando alguma coisa, se você quiser só os recursos não haverá corpo, entretanto se quisermos adicionar um recurso no servidor você terá que prover algo para mostrar como dado e também e chamado de payload. Essa é a estrutura de uma Mensagem de Requisição. Agora vamos falar um pouco sobre as Mensagens de Resposta isto é o que vamos estar criando ele irá ter uma versão http e na maioria das vezes será 1.1, e terá um código status e um código texto, o código de status significa o qual é o resultado da requisição por exemplo o número no slide é 200 significa que a requisição foi um sucesso isso é o que geralmente mandamos de volta e de novo temos um Header que proporciona informações sobre a requisição de reposta, agora entre em <https://course-api.com/> ative o modo desenvolvedor e vá em network depois atualize a página e depois clique no primeiro nome e observe que a página tem um código status e abaixo tem mais informações ao lado temos preview e response, em preview você basicamente verá o site em sua forma html e em response você verá o código.

**HTTP BASICS**

Agora vamos dar inicio ao aprendizado utilizando o Express e usando os pequenos conceitos de http que aprendemos para trás vá nesse link e clone o repositório [https://github.com/john-smilga/node-express-course](https://github.com/john-smilga/node-express-course%20) agora vá na pasta 02-express-tutorial entre nela e utilize o comando npm install && npm start, se estiver aparecendo Express Tutorial no terminal é porque está tudo certo até o momento.

Primeiro crie uma variável para guardar o modulo http, depois crie uma variável servidor para criar o servidor usando o método createServer depois crie uma callback com os parâmetros de requisição (req) e de resposta (res), dentro da callback crie um console.log para informar que o usuário chegou ao servidor, abaixo use a variável servidor e em seguida chame o método listen informando a porta que os dados serão utilizados, utilize portas que não estão sendo usadas no momento. Entre em localhost: (número da porta) e observe que o ícone de carregamento acima do browser irá ficar carregando e isso significa que a parte do servidor precisa mandar uma resposta de volta para o navegador para isso usamos o método end na resposta ele permite que mandemos de volta ao navegador dois tipos de dados html ou escrita mesmo por enquanto mande um ‘home page’ de volta e observe no navegador em localhost que ele irá mostrar a mensagem, sempre que uma resposta for criada precisamos encerrá-la usando o método end.

//console.log('Express Tutorial')

const *http* = require('http');

const server = *http*.createServer((*req*,*res*)=>{

    console.log('user hit the server')

*res*.end('home page');

})

server.listen(5000);

**HTTP – HEADERS**

Para informar o headers do servidor precisamos usar um método de reposta ele se chama SetHeader informando o nome que no caso sempre será Content-Type e o valor que pode ser variado por exemplo text/html, text/json, e em seguida escrevemos abaixo usando o método writeHead informando o código status e depois um objeto que informa o tipo de conteúdo. Após informar isso escreva um res.write um html com um h1 ou h2 e em seguida vá no localhost observe que o que você escreveu mudou de tamanho por causa do html agora se você mudar o tipo de dado do Content-Type para qualquer outra coisa ele irá voltar para o texto normal pois ele não é mais html. O Express que iremos ver mais pra frente cuida disso e não precisa configurar. Para saber mais sobre código status entre no link [https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status%20%20) . E sempre importante informar o código status certinho pois nada impede que você faça alterações do código por exemplo informar um 404 mesmo o código estando correto.

//console.log('Express Tutorial')

const *http* = require('http');

const server = *http*.createServer((*req*,*res*)=>{

*res*.setHeader('Content-Type', 'text/html');

*res*.writeHead(200, { 'Content-Type': 'text/html' });

*res*.write('<h1>home page</h1>')

*res*.end();

    console.log('user hit')

})

server.listen(5000);

**HTTP-REQUEST OBJECT**

Agora que mechemos na resposta do código vamos começar a usar um pouco da requisição que mandamos de volta de um console.log em req e observe o bloco de código grande que aprece no terminal. Outra coisa que temos que ficar atento é a ao método http que está sendo utilizado para saber o tipo use req.method, outra coisa que iremos observar é o url utilizando o método req.url isso informará o url utilizado por exemplo coloque depois do localhost numero um /contact isso informará a url utilizada do caminho e se não tiver nada por exemplo só o localhost número informará a barra que é a home page e caso o caminho não exista a gente manda um 404 para o caminho informando que ela não existe porem se existir o terminal irá buscar.

//console.log('Express Tutorial')

const *http* = require('http');

const server = *http*.createServer((*req*,*res*)=>{

    //console.log(req.method)

    //console.log(req.url)

    const url = *req*.url;

    //home page

    if(url==='/'){

*res*.setHeader('Content-Type', 'text/html');

*res*.writeHead(200, { 'Content-Type': 'text/html' });

*res*.write('<h1>home page</h1>')

*res*.end();

    }

    //about page

    else if(url === '/about'){

*res*.setHeader('Content-Type', 'text/html');

*res*.writeHead(200, { 'Content-Type': 'text/html' });

*res*.write('<h1>about page</h1>')

*res*.end();

    }

    // 404

    else{

*res*.setHeader('Content-Type', 'text/html');

*res*.writeHead(404, { 'Content-Type': 'text/html' });

*res*.write('<h1>page not found</h1>')

*res*.end();

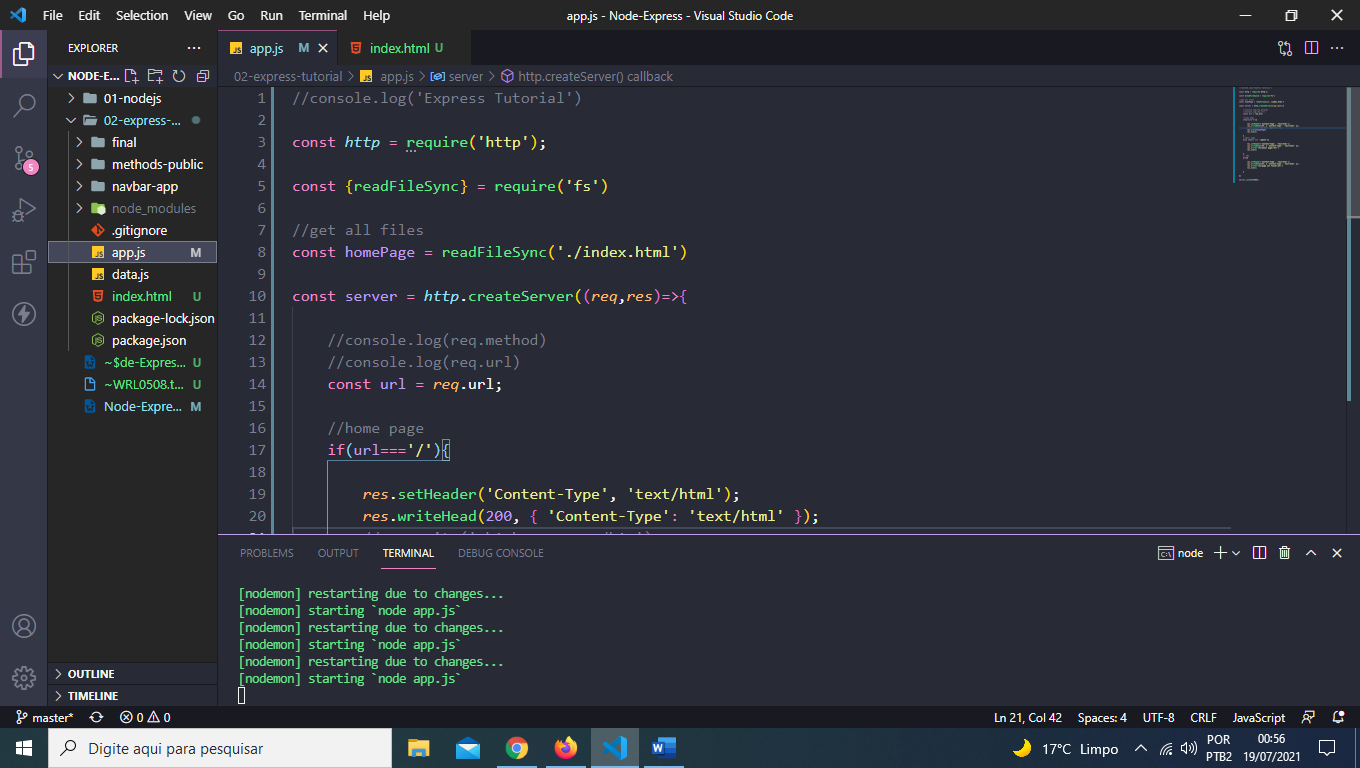
    }

})

server.listen(5000);

**HTTP-HTML FILE**

Nós não somos limitados a passar o html dentro do .write ou .end, imagine se você tiver que fazer o html nos métodos diretamente todas as vezes, para diminuir isso e melhor fazer um arquivo e passar esse arquivo para o sistema, iremos passar o conteúdo do arquivo não o arquivo diretamente isso significa que precisamos usar o content-type, esse exemplo agora é um bem simples pois quando começarmos o express o exemplo será um pouco maior, crie um arquivo index.html com um <h1> home page </h1> e um <h4> hello world </h4> e depois chame-o no res.write. Primeiro chame o modulo fs com chaves em readFileSync, const {readFileSync} = require(‘fs’), depois crie uma variável para guardar o caminho do index.html usando o readFileSync, const homepage = readFileSync (‘./index.html’), temos dois motivos para usar o readFileSync o primeiro é porque não estamos invocando toda vez que alguém vem ao servidor e também estamos apenas requisitando isso uma vez, agora que criamos a variável homepage troque o res.write para homepage, agora observe se mudarmos o content-type por exemplo para text/plain ele irá mostrar o código fonte do html ao invés do conteúdo do arquivo html.



//console.log('Express Tutorial')

const *http* = require('http');

const {readFileSync} = require('fs')

//get all files

const homePage = readFileSync('./index.html')

const server = *http*.createServer((*req*,*res*)=>{

    //console.log(req.method)

    //console.log(req.url)

    const url = *req*.url;

    //home page

    if(url==='/'){

*res*.setHeader('Content-Type', 'text/html');

*res*.writeHead(200, { 'Content-Type': 'text/html' });

        //res.write('<h1>home page</h1>')

*res*.write(homePage)

*res*.end();

    }

    //about page

    else if(url === '/about'){

*res*.setHeader('Content-Type', 'text/html');

*res*.writeHead(200, { 'Content-Type': 'text/html' });

*res*.write('<h1>about page</h1>')

*res*.end();

    }

    // 404

    else{

*res*.setHeader('Content-Type', 'text/html');

*res*.writeHead(404, { 'Content-Type': 'text/html' });

*res*.write('<h1>page not found</h1>')

*res*.end();

    }

})

server.listen(5000);

**HTTP – APP EXAMPLE**

Para utilizar no próximo exemplo olhe no arquivo acima na pasta navbar-app e em seguida olhe no navegador o index.html se ele estiver funcionando e mostrando e responsivo no navegador, está tudo correto em seguida mude a rota do arquivo adicionando em homepage, ‘./navbar-app/index.html’, o que acontece agora depois que você atualiza e o seguinte você s ove a estrutura html e não consegue ver o logo, ou o css, agora de um console.log em url e observe o que será retornado para o terminal, o servidor está pedindo mais três arquivos que foram colocados junto do html pois esses arquivos possuem caminhos no server, então temos que fazer para cada um dos caminhos um if else igual fizemos para about e o 404. O express veio para melhorar essa dinâmica e deixar mais simples o progresso do projeto.

//console.log('Express Tutorial')

const *http* = require('http');

const {readFileSync} = require('fs')

//get all files

const homePage = readFileSync('./navbar-app/index.html')

const homeStyle = readFileSync('./navbar-app/styles.css')

const homeLogo = readFileSync('./navbar-app/logo.svg')

const homeLogic = readFileSync('./navbar-app/browser-app.js')

const server = *http*.createServer((*req*,*res*)=>{

    //console.log(req.method)

    //console.log(req.url)

    const url = *req*.url;

    //home page

    if(url==='/'){

*res*.setHeader('Content-Type', 'text/html');

*res*.writeHead(200, { 'Content-Type': 'text/html' });

        //res.write('<h1>home page</h1>')

*res*.write(homePage)

*res*.end();

    }

    //styles for home page

    else if(url === '/styles.css'){

*res*.setHeader('Content-Type', 'text/css');

*res*.writeHead(200, { 'Content-Type': 'text/css' });

        //res.write('<h1>about page</h1>')

*res*.write(homeStyle)

*res*.end();

    }

    //logo for home page

    else if(url === '/logo.svg'){

*res*.setHeader('Content-Type', 'image/svg+xml');

*res*.writeHead(200, { 'Content-Type': 'image/svg+xml' });

        //res.write('<h1>about page</h1>')

*res*.write(homeLogo)

*res*.end();

    }

    //logic for home page

    else if(url === '/browser-app.js'){

*res*.setHeader('Content-Type', 'text/javascript');

*res*.writeHead(200, { 'Content-Type': 'application/javascript' });

        //res.write('<h1>about page</h1>')

*res*.write(homeLogic)

*res*.end();

    }

    //about page

    else if(url === '/about'){

*res*.setHeader('Content-Type', 'text/html');

*res*.writeHead(200, { 'Content-Type': 'text/html' });

*res*.write('<h1>about page</h1>')

*res*.end();

    }

    // 404

    else{

*res*.setHeader('Content-Type', 'text/html');

*res*.writeHead(404, { 'Content-Type': 'text/html' });

*res*.write('<h1>page not found</h1>')

*res*.end();

    }

})

server.listen(5000);

**EXPRESS BASICS**

O Express e um node.js web app framework desenvolvido para a criação de websites, api’s mais rapidamente e facilmente, o express não e um built-in modulo como o http porém ele o mais padrão em desenvolvimento de web apps com node.js, a documentação do express e simples e fácil para acessá-las entre em [https://expressjs.com/](https://expressjs.com/%20)  para instalar simplesmente use o comando npm install express –save.

Agora vamos aprender como configurar um servidor com o express, primeiro crie uma variável para guardar o modulo express, const express = require (‘express’), depois crie uma variável app (pode ser outro nome mais é o padrão) e invoque a função express, const app = express (), se você está um pouco preocupado com a sintaxe do express não se preocupe ela é bem similar ao do http, agora precisamos informar o local da porta do servidor para isso use app.listen(número da porta, callback), e informe dentro da callback um console.log para saber no terminal se está tudo certo, app.listen(5000, () => {console.log(‘server is listening on port 5000...’)}), como sabemos que todos os navegadores aceitam uma requisição get nós vamos com app.get() e passamos duas coisas dentro a primeira a rota por exemplo /soccer e depois uma callback com os parâmetros de requisição e reposta, e dentro da callback que colocamos o conteúdo como no http usamos o res.send () nele cabe dados de texto/html entre outros, app.get(‘/’,(req, res)=>{res.send(‘Home Page’)}) agora faça o mesmo para a página /about, app.get(‘/about’,(req, res)=>{res.send(‘About Page’)}), e também como no exemplo passado temos que fazer com o erro 404 vamos usar o método all para pegar todos os tipos de requisição get, put, update, etc passamos os mesmas coisas dentro a rota, e depois a callback, dentro da callback a uma forma de definir o código status usando o .status() , app.all(‘\*’,(req, res)=>{res.status(404).send(<h1> resource not found </h1>)}) o asterisco significa que engloba tudo.

const express = require('express')

const app = express()

//Metodos mais utilizados

//

//app.get

//app.post

//app.put

//app.delete

//app.all

//app.use

//app.listen

app.get('/', (*req*,*res*)=>{

    console.log('user hit the resource')

*res*.status(200).send('Home Page')

})

app.get('/about', (*req*,*res*)=>{

*res*.status(200).send('About Page')

})

app.all('\*', (*req*,*res*)=>{

*res*.status(404).send('<h1>resource not found</h1>')

})

app.listen(5000,() => {

    console.log('server is listening on port 5000...' )

})

**EXPRESS -NAVBAR-APP**

Para gravar melhor o aprendizado sobre o express comece do inicio de novo criando primeiro o listen e o require ‘express’, const express = require(‘express’); const app = express(); app.listen(5000, ()=>{console.log(‘Server is running on port 5000...’)}), depois crie as rotas uma para a pagina home e outra para caso ocorra erros, app.get(‘/’,(req, res)=>{}); app.all(‘\*’,(req, res)=>{}), diferente da forma de http quando fazemos uma rota para cada qualquer coisinha o express diminui esse trabalho, agora precisamos passar um caminho ou ‘path’ para mandar os arquivos tipo o index.html ou outros para o navegador para isso primeiro crie uma variável para guardar o módulo do node.js path, const path = require(‘path’), agora para mandar os arquivos para o navegador usamos um método chamado sendFile que vai permitir mandar os arquivos e dentro desse método precisamos passar a variável path com um método resolve ou join tanto faz e dentro desse método passar \_\_dirname e depois o caminho para o arquivo './navbar-app/index.html', res.sendFile(path.resolve(\_\_dirname, ‘./navbar-app/index.html’)) e na rota do erro passe res.status(404).send(‘resource not found’), agora observe do navegador que ele não passou os itens como css e o javascript igual o do http que fazemos anteriormente para passa-los é bem mais simples que no modulo http usamos o método do express use passando o middleware static do express passando os arquivos estáticos para uma pasta public arquivos estáticos são css, javascript e imagens aqui no nosso exemplo no caso styless.css, browser-app.js, logo.svg mova para a pasta public e depois passe no middleware static, app.use( express.static(‘./public’)).

const express = require('express')

const path = require('path')

const app = express()

//setup static and middleware

app.use(express.static('./public'))

app.get('/', (*req*,*res*) => {

*res*.sendFile(path.resolve(\_\_dirname,'./navbar-app/index.html'))

})

app.all('\*', (*req*,*res*) => {

*res*.status(404).send('resource not found')

})

app.listen(5000, () => {

    console.log('server is running on port 5000...')

})

**API VS SERVER-SIDE RENDERING**

Quando pensamos em Express na maioria dos casos você irá usar um dos dois tipos uma você vai usar para configurar/criar uma API ou usar modelos que renderizam do lado do servidor. Em express ou no caso http quando falamos de api falamos de configurar uma interface http para interagir com os nossos dados, agora os dados são mandados em JSON e para mandar a nossa resposta usaremos o método res.json() o que fará todo o trabalho pesado por exemplo configurar o tipo de conteúdo e stringficar todo dado, agora o caso de server-side rendering iremos criar modelos que mandarão de volta todo o html, css e javascript usando o método res.render(). O conceito que iremos usar no curso será de api pois focará mais no express, porém se você conseguir aprender bem os conceitos usando a forma de api você também conseguirá fazer com template engine.

✦API – JSON ✦SSR - Template

✦Send Data ✦Send Template

✦Res.Json() ✦Res.Render()

Para acessar os dados de uma api no express é bem simples primeiro monte o setup do express, const express = require('express'); const app = express (), depois crie o listen para mostrar o número da porta e transferir de lá, app.listen(5000, ()=>{ console.log('Server is listening on port 5000...')}), agora crie uma rota get para acessar esses dados que são JSON por exemplo /api ou só /, app.get(‘/’,(req, res)=>{}), para mostrar esses dados vamos fazer um pequeno exemplo crie um res.json() e em seguida passe dentro dele um array com dois objetos mostrando dois nomes, res.json([{name: ’john’}, {name: ‘susan’}]), agora acesse o localhost e observe o arquivo ele mostra como ele separado, agora dentro da nossa pasta tem um arquivo data.js ele tem um conglomerado de dados, esses dados são um array de produtos e um array de pessoas no final observe que ele tem um model.exports exportando os array para acessá-los basta no arquivo app.js desconstrui-lo, const {products} = require(‘./data’), e no res.json() simplesmente chamar os produtos, res.json(products).

const express = require('express')

const app = express()

const {products} = require('./data')

app.get('/',(*req*,*res*)=>{

    //res.json([{name: 'john'},{name:'susan'}])

*res*.json(products)

})

app.listen(5000, ()=>{

    console.log('Server is listening on port 5000...')

})

**PARAMS, QUERY STRING – SETUP**

Agora que vimos como fazer a configuração inicial para um arquivo json, vamos criar agora um para a api, vamos começar chamando a rota home page e redirecionar um link com h1 que irá direcionar o usuário para /api e depois produtos e mais pra frente cobrimos params e query string, agora no exemplo anterior troque o res.json por um res.send com os seguintes comandos, res.send('<h1>Home Page</h1> <a href="/api/products">products</a>') , isso redireciona o cliente para uma rota /api/products porém ainda não fizemos essa rota então agora iremos cria-la como nos exemplos anteriores faça, app.get(‘/api/products’, (req, res)=>{}), pronto criamos a rota agora para acessar esses dados basta dar um res.json(products) que acessaria todos os dados porem nem sempre precisaremos de todos os dados, agora fazendo um pouco mais realisticamente caso não quisermos acessar todos os dados faça um .map para ler todos os dados e escolher somente o necessário, const newProducts = products.map((product)=>{ const {id, name, image} = product; return {id, name, image}}) e no final e só chamar o newProducts dentro do res.json, res.json(newProducts) agora entre no seu localhost e pronto podemos acessar os dados escolhidos.

const express = require('express')

const app = express()

const {products} = require('./data')

app.get('/',(*req*,*res*)=>{

    //res.json([{name: 'john'},{name:'susan'}])

*res*.send('<h1>Home Page</h1> <a href="/api/products">products</a>')

})

app.get('/api/products', (*req*,*res*)=>{

    const newProducts = products.map((*product*)=>{

        const {id,name,image,} = *product*;

        return {id,name,image}

    })

*res*.json(newProducts)

})

app.listen(5000, ()=>{

    console.log('Server is listening on port 5000...')

})

**ROUTE PARAMS**

Agora vamos observar como garimpar a informação de um único especifico produto, por exemplo se navegarmos para /api/products/1 nos iremos conseguir acessar as três propriedades que passamos e também a descrição que não passamos, primeiro configure uma rota nova, app.get(‘/api/products/1’,(req, res)=>{}), para encontrar um único produto usamos o método find que irá buscar apenas o produto com o id 1, app.get(‘/api/products/1’,(req, res)=>{const singleProduct = products.find((product) => product === 1)}), agora quando acessamos essa rota ela irá acessar todos os dados inclusive a descrição que não foi colocado anteriormente porém tem um adendo isso só serve para esta rota e se tivéssemos por exemplo 100 ou 200 produtos isso não seria viável uma solução melhor que o express nos proporciona são as rotas parâmetro.

const express = require('express')

const app = express()

const {products} = require('./data')

app.get('/',(*req*,*res*)=>{

    //res.json([{name: 'john'},{name:'susan'}])

*res*.send('<h1>Home Page</h1> <a href="/api/products">products</a>')

})

app.get('/api/products', (*req*,*res*)=>{

    const newProducts = products.map((*product*)=>{

        const {id,name,image,} = *product*;

        return {id,name,image}

    })

*res*.json(newProducts)

})

app.get('/api/products/1', (*req*,*res*)=>{

    const singleProduct = products.find((*product*)=>*product*.id === 1)

*res*.json(singleProduct)

})

app.listen(5000, ()=>{

    console.log('Server is listening on port 5000...')

})

Para começarmos a mexer com os parâmetros adicione no lugar de 1, o sinal de dois pontos, : isso fará com que seja adicionado um parâmetro a rota, no meu caso adicione :productID, uma coisa que precisamos checar sempre e se na rota for um número essa número tem que voltar como string, isso é importante no nosso exemplo pois o nosso id é um número, agora de um console.log() em req.params esse e o método que usamos para configurar o parâmetro de uma rota para acessar diversos valores, agora desconstrua productID, const {productID} = req.params; pronto assim podemos acessar esse parâmetro, agora na função de um único produto como nosso id é um número temos duas opções transformar os dados de productID para string ou transformar productID em numero para isso no depois de === adicione Number(productID) isso fará com que os dados em string se transforme em números tornando-se capaz de acessar os dados agora no localhost observe que podemos acessar os diferentes números do dado e ele irá nos mostrar o produto singularmente. Caso o usuário digite um número ou qualquer outra coisa que não necessariamente seja o id do produto, a página não irá retornar nada porém para que possamos mostrar que essa rota/id não existe precisamos de uma condição mostrando um 404, para isso crie um if se o único produto não existir retornar um 404 e um texto, if(!singleProduct){return res.status(404).send(‘Product does not exist’)}e no final adicione um return na frente de res.json().

const express = require('express')

const app = express()

const {products} = require('./data')

app.get('/',(*req*,*res*)=>{

    //res.json([{name: 'john'},{name:'susan'}])

*res*.send('<h1>Home Page</h1> <a href="/api/products">products</a>')

})

app.get('/api/products', (*req*,*res*)=>{

    const newProducts = products.map((*product*)=>{

        const {id,name,image,} = *product*;

        return {id,name,image}

    })

*res*.json(newProducts)

})

app.get('/api/products/:productID', (*req*,*res*)=>{

    //console.log(req)

    //console.log(req.params)

    const {producID} = *req*.params;

    const singleProduct = products.find((*product*)=>*product*.id === *Number*(producID))

    if(!singleProduct){

        return *res*.status(404).send('Product does not exist')

    }

    return *res*.json(singleProduct)

})

app.listen(5000, ()=>{

    console.log('Server is listening on port 5000...')

})

Esses exemplos podem ser bem mais complexos como por exemplo eu quero acessar o produto pelo id também acessando as revisões pelo id, de um console.log em req.params e repare que os dados também foram acessados no terminal mande uma mensagem com res.send para ver se funcionou no localhost.

app.get('/api/products/:productID/reviews/:reviewID', (*req*,*res*)=>{

    console.log(*req*.params)

*res*.send('hello world')

})

**QUERY STRING**

Essencialmente query string é uma forma de mandar informação para o servidor usando a url, agora essas informações são usadas como parâmetros como por exemplo query database ou filtrar resultados isso e geralmente da pessoa que está configurando o servidor, agora vá ao seu navegador e procure hacker news algolia api o link é este <https://hn.algolia.com/api> esta api mostra um bom exemplo como devemos configurar o servidor agora note na url eles começam com um domínio que no caso é localhost 5000,depois /api que é uma boa pratica, depois uma versão e depois dela com uma lista qualquer no exemplo é items e depois disso uma rota parâmetro, continue rolando para baixo e chegamos na parte de pesquisa no caso sort por exemplo conseguir uma informação especifica ou informação designada por um termo, geralmente um parâmetro que temos que adicionar são páginas por exemplo quando um usuário clica num botão para avançar ou retroceder, e como adicionamos isso é geralmente usando key values por exemplo <http://hn.algolia.com/api/v1/search?query=foo&tags=story> nesse link a key é query. Agora para começar vamos criar uma rota com um parâmetro mais complexo, app.get(‘/api/v1/query’, (req, res) =>{}), agora dentro da callback function de um console.log() em req.query para acessar e depois de um res.send() para testar. No navegador digite o caminho até query e adicione quantos parâmetros a mais você quiser por exemplo localhost:5000/api/v1/query?name=lucas&id=1 agora como padrão no navegador temos a mensagem do res.send() porém no terminal obtemos os valores de query podendo acessá-los e dar uma funcionalidade, agora na url vamos adicionar uma query para Search e determinar um limite para os produtos, agora vamos fazer esta funcionalidade. Primeiro crie um objeto guardando search e limite em req.query, const {search, limit}= req.query ,depois crie uma variável let para guardar os produtos que serão ordenados, let sortedProduct = [...products], depois crie uma condição para achar esses produtos dentro do array para isso utilizamos o método filter() , if(search) { sortedProduct = sortedProduct.filter( (product) => { return product.name.startsWith(search) })}, agora uma condição para o limite se o produto for menor ou igual a zero ou maior que o que tem na lista mostra-los usaremos a função slice(), if(limit){sortedProduct= sortedProduct.slice(0, Number(limit))}, e depois uma condição para se não existir um valor na lista de produtos para isso você pode mandar uma mensagem com nenhum produto encontrado porém uma boa pratica é criar um objeto com sucesso e mandar um array data vazio, if(sortedProduct.length<1){ //res.status(200).send('no products match your search...') return res.status(200).json({success: true, data:[]})}, e no final retorne o produto ordenado ou sorted.

app.get('/api/v1/query',(*req*,*res*)=>{

    //console.log(req.query)

    const {search, limit} = *req*.query;

    let sortedProduct = [...products];

    if(search){

        sortedProduct = sortedProduct.filter((*product*)=>{

            return *product*.name.startsWith(search)

        })

    }

    if(limit){

        sortedProduct= sortedProduct.slice(0,*Number*(limit))

    }

    if(sortedProduct.length<1){

        //res.status(200).send('no products match your search...')

        return *res*.status(200).json({success:true, data:[]})

    }

*res*.status(200).json(sortedProduct)

    //res.send('hello world')

})

**MIDDLEWARE-SETUP**

Depois de aprendermos sobre route params e query string vamos falar um pouco sobre middleware, elas são funções que se executam durante a requisição do servidor, cada middleware tem um acesso a requisição e resposta de objetos e tratando de funcionalidade ela pode ter qualquer uma, agora vamos fazer alguns exemplos de código pois é mais fácil aprender praticando do que lendo textos. Antes de começar deixa eu te explicar uma coisa middleware está em todo o lugar no express e isso significa que este tópico não é um que você pode pular ele praticamente é a base do express, se você sentir um pouco de dificuldade no inicio não entre em pânico a cada exemplo que iremos passando melhor será seu entendimento. Começando agora limpe o seu app.js e deixe somente app e express e também o listen, adicione um comentário para sua referência explicando que tem uma requisição vinda e depois vem o middleware e depois a resposta, req => middleware => res, agora vamos começar por dois cenários em que eu tenho duas rotas a home e a about, agora por exemplo se colocarmos uma funcionalidade logger por exemplo nós temos que colocar ela nas duas rotas, agora crie três variáveis na home para guardar o método a url e o tempo e depois de um console.log(), const method = req.method; const url = req.url; const time = new Date().getFullYear(), agora toda vez que o usuário acessa a home aprece no terminal o método usado, a url que ele está e o ano. Já pensou fazer isso pra quinze ou mais rotas uma solução pra isso e fazer uma função ela terá toda essa logica e será anexada a rota, sendo assim você pode anexar em algumas rotas e outras não, agora crie uma função logger e depois pegue essas quatro linhas de código e passe para ela, const logger = ()=>{const method = req.method; const url = req.url; const time = new Date().getFullYear();console.log(method, url, time)}, agora onde anexamos essa função para que não dupliquemos nosso código, e a resposta e entre o caminho e a callback, agora no logger nos estamos acessando a requisição do objeto porque no momento não estamos passando, a boa noticia é que o express passa para os nossos middlewares por trás das cenas com o req, res e o next, agora indo no navegador e acessando temos um problema no navegador nós temos o spinner girando e girando sem resposta, quando trabalhamos com middleware você sempre tem que passar para o próximo middleware a não ser que o ciclo esteja terminando mandando um res.send(), se você quiser passar para a próxima função simplesmente adicione next e invoque, next().

const express = require('express')

const app = express()

// req => middleware => res

const logger = (*req*, *res*, *next*) => {

    const method = *req*.method;

    const url = *req*.url;

    const time = **new** *Date*().getFullYear()

    console.log(method, url, time)

    next()

}

app.get('/', logger, (*req*, *res*)=>{

*res*.send('Home')

})

app.get('/about', logger,(*req*, *res*)=>{

*res*.send('About')

})

app.listen(5000, ()=>{

    console.log('Server is listening on port 5000...')

})

**APP.USE**

Pronto estamos agora mais familiar com middleware function mas tem dois problemas com esta configuração no momento, primeiro o nosso app.js está um pouco desajeitado pois nós temos a função logger depois os métodos, e uma boa pratica ter esta função logger ou outro middleware criado em um arquivo separado isso irá deixar nosso app.js limpo e vai deixar mais fácil de navegar no arquivo, e o segundo problema é e se nós tivéssemos cinquenta rotas eu não quero ter que adicionar essa função manualmente a cada uma delas seria uma boa se tivesse uma função que adiciona-se pra qualquer rota, e sim ela existe utilizamos ela alguns exemplos atrás é o .use(), agora crie um arquivo para guardar o middleware no caso o nome do arquivo logger.js e passe a função para lá e adicione um mdule.exports = {logger}, agora passe no app.js o logger usando o require, const logger = require(‘./logger’), agora se você for em localhost 5000 você ainda consegue ver o console.log se você ver está tudo correto, agora a segunda pare é como eu posso aplicar esse logger para todas as minhas rotas, agora crie mais duas rotas e remova as funções logger e em seguida observe que a função manda para todas as rotas, um fator importante a ordem dos fatores aqui altera o programa por exemplo se uma rota está acima do .use() você não verá nada para o que foi passado para aquela rota, outra coisa podemos adicionar um caminho em app.use() e tudo o que for passado será adicionado depois do caminho adicionado na rota

const express = require('express')

const app = express()

const logger = require('./logger')

// req => middleware => res

app.use(logger)

app.get('/', (*req*, *res*)=>{

*res*.send('Home')

})

app.get('/about', (*req*, *res*)=>{

*res*.send('About')

})

app.get('/api/products', (*req*, *res*)=>{

*res*.send('Products')

})

app.get('/api/items', (*req*, *res*)=>{

*res*.send('Items')

})

app.listen(5000, ()=>{

    console.log('Server is listening on port 5000...')

})

**MULTIPLE MIDDLEWARE FUNCTIONS**

Agora nos iremos fazer o nosso exemplo ficar mais interessante adicionando outra função middleware e no processo iremos olhar para como executamos eles, qual a sintaxe para adicionar e qual a ordem de execução, vamos começar simples criando um novo arquivo com o nome authorize.js, agora use a mesma configuração que usamos no exemplo anterior usando req, res, next passando um console.log(‘authorize’) e função next() em seguida de um module.exports e chame em app.js a variável. E agora com a variável criada para executar ambos os middlewares e só os adicionar em um array em app.use(). Observe que quando acessamos localhost 5000 ele mostra primeiro o logger e depois o authorize mas se você mudar a ordem em app.use ele irá mostrar o authorize primeiro depois o logger, isso é só um exemplo simples de como autorizar o usuário usando uma query string, e não essa não é a forma como autorizamos usuários no express. Agora volte para authorize.js e configure uma query string para isso primeiro crie um objeto para guardar o usuário chamando o método req.query e depois crie uma condição se o usuário, tem algum nome mostre req.user = {name: ’gabriel’, id:2} e depois passe um next() dentro. Essa condição também incita que temos que criar o um else que seria caso não encontrasse um usuário mande um res.status(401).send(‘Unauthorized’), agora observe no navegador quando atualizamos a pagina mostra que não somos autorizados o mesmo no terminal, isso significa que eu não sou autorizado ao usuário, e não foi proporcionado um parâmetro query streams com uma key de user em especifico um valor de gabriel e como você não fez isso você não pode acessar o recurso. Agora se mudarmos algumas coisas por exemplo meu usuário é igual a gabriel a condição é adquirida e vamos para o próximo o porque disso é poderoso é porque podemos adicionar o usuário e não somente podemos checar isso com a query string e acessar essa informação, isso é só um exemplo normalmente o que nos temos que fazer é checar pelo jsonWebToken e se o token existir nos comunicamos com o banco de dados e conseguimos o usuário.

authorize.js

const authorize = (*req*, *res*, *next*) =>{

    const {user} = *req*.query;

    if(user === 'gabriel'){

*req*.user = {name: 'gabriel', id:2}

        next()

    }

    else{

*res*.status(401).send('Unauthorized')

    }

}

*module*.*exports* = authorize;

app.js

const express = require('express')

const app = express()

const logger = require('./logger')

const authorize = require('./authorize')

// req => middleware => res

app.use([logger, authorize])

//api/home/about/products

app.get('/', (*req*, *res*)=>{

*res*.send('Home')

})

app.get('/about', (*req*, *res*)=>{

*res*.send('About')

})

app.get('/api/products', (*req*, *res*)=>{

*res*.send('Products')

})

app.get('/api/items', (*req*, *res*)=>{

    console.log(*req*.user)

*res*.send('Items')

})

app.listen(5000, ()=>{

    console.log('Server is listening on port 5000...')

})

**ADDITIONAL MIDDLEWARE INFO**

Agora eu quero responder duas questões comuns primeiro e se tivermos acesso a app.use() desde que nós sabemos como usar nós vamos adicionar o middleware nas rotas e a reposta é sim pois imagine o cenário que eu não quero usar o app.use() em todas as rotas por exemplo eu só vou checar se o usuário é autorizado na rota /api/items adicionando o middleware só nessa rota eu vou conseguir acessar todas as outras sem problemas, e se você está pensando como podemos passar duas middlewares numa rota basta passar num array da mesma maneira que colocamos no app.use anteriormente, agora teste no navegador e no terminal se conseguir acessar essa rota e voltar está tudo ok agora volte para a home page e pronto você n]ao consegue acessar o middleware porque ela não foi passada para essa rota. A segunda questão seria quais as nossas opções perante middlewares? Até aqui nos cobrimos a opção dos nossos próprios middlewares criados sim nós podemos crias os nossos próprios middlewares as outras duas opções são pelo express sim o express nos provém middlewares deles nesse caso a gente não precisa se preocupar em configurar a funcionalidade nos só precisamos referenciar os documentos para ver quais opções são providas e fornecer esses dados, e se você lembra o exemplo do express nós usamos o app.use(express.static(‘./public’)) aqui no caso static e um middleware do express agora o ultimo deixa eu te mostrar um exemplo de middleware de terceiros, agora para middleware de terceiros nós temos que instala-los usando o grande npm, agora um dos mais populares para o login é este o morgan [https://www.npmjs.com/package/morgan](https://www.npmjs.com/package/morgan%20) agora instale-o com o npm i morgan depois crie uma variável para guardar const morgan = require(‘morgan’) no app.use() chame morgan(‘tiny’) e pronto isso irá mostrar no terminal as rotas métodos e tempo pra abrir no navegador.

const express = require('express')

const app = express()

const morgan  = require('morgan')

const logger = require('./logger')

const authorize = require('./authorize')

// req => middleware => res

//1. use vs route

//2. options -our own/ express/ third party

//app.use([logger, authorize])

//app.use(express.static('./public'))

app.use(morgan('tiny'))

//api/home/about/products

app.get('/', (*req*, *res*)=>{

*res*.send('Home')

})

app.get('/about', (*req*, *res*)=>{

*res*.send('About')

})

app.get('/api/products', (*req*, *res*)=>{

*res*.send('Products')

})

app.get('/api/items',/\*[logger, authorize],\*/(*req*, *res*)=>{

    console.log(*req*.user)

*res*.send('Items')

})

app.listen(5000, ()=>{

    console.log('Server is listening on port 5000...')

})

**METHODS – GET**

Lembrando do exemplo passado estudamos os ciclos http e nele tem os métodos http e em requisição get é o padrão performado pelo navegador mas claro temos outros métodos no slide em <https://course-api.com/slides/> podemos observar todos os métodos e como eles funcionam no caso de post por exemplo podemos mandar ou executar uma ordem, depois temos put caso você queira atualizar um dado e delete para excluir os dados. Como não aprendemos como conectar a um banco de dados não iremos usar os dados do exemplo anterior, agora indo ao app.js remova basicamente quase tudo deixando só o app.listen e o const app. Agora nós iremos usar api people do arquivo data.js e com esses dados eu vou mostrar com usar os quatro métodos http principais. Importe people de data.js, let {people} = require (‘./data’).

Agora vamos começar com a básica que usamos anteriormente usando o método GET.

const express = require('express')

const app = express()

let  {people} = require('./data')

app.get('/api/people', (*req*, *res*)=>{

*res*.status(200).json({success: true, data: people})

})

app.listen(5000, ()=>{

    console.log('Server is listening on port 5000...')

})

**METHODS – POST**

Para inserir dados no servidor usamos o método POST mas não é algo que se configura no navegador e começar a fazer requisições post, infelizmente não é assim que funciona, é por isso que será provido um pouco de ajuda agora guarde na mente antes de estabilizar um método post corretamente nós temos dois tipos de configuração, nós iremos instalar mais uma ferramenta que vai ser crucial no desenvolvimento do express server que vai permitir que a gente teste tudo mais rápido mas como é um método post eu quero mostrar os dois métodos utilizados para fazer agora em app.js o método app.use o que ele aplica os middlewares para todas as rotas e eu também menciono que temos a opção de criar um diretório publico agora para esse exemplo temos uma pasta chamada method-public é como você podem ver é só umas pasta com html, javascript e css esses arquivos foram criados simplesmente porque não podemos fazer requisições post do navegador. Primeiro vamos usar o app.use e chamar esses arquivos estáticos da pasta method-public, app.use (express.static (‘./methods-public’)).

Agora vá em methods-public e observe os dois tipos de forma que podemos fazer, navegue para index.html e nele iremos encontrar um típico arquivo com um nav, alguns estilos e um formulário, num formulário temos dois atributos a action e o method e vemos claramente que o valor do método e POST e o atributo de action informa para onde iremos mandar esse formulário e note que temos /login isso significa que esse caminho/url está no nosso servidor porém em app.js nos não a fizemos ainda, agora no navegador iremos inserir um dado e manda-lo para /login observe o que acontece entre no modo desenvolvedor e em network obtemos o tipo de método que está sendo utilizado. Quando rebemos uma requisição GET não mandamos de volta um corpo, e para adicionar alguma coisa ao servidor precisamos adicionar algo ao corpo e pegar aquele dado, no localhost no input que você digita um nome será fornecida uma chave por exemplo se alterarmos a tag name="name" no html para name="testing", você vê que irá mudar o valor da chave agora quando acessamos a rota que fomos redirecionados não conseguimos acessar o dado passado para o formulário e também não criamos o middleware para acessar a rota, para criar uma middleware com acesso ao método POST é só mudar o método na rota na hora da criação app.post('/login', (req, res)=>{ res.send('POST ')}) agora quando adicionarmos um dado no formulário pelo menos temos acesso a rota e conseguimos ver res.send(), infelizmente ainda não conseguimos acessar os dados que colocamos dentro formulário para isso precisamos do body parser você pode utilizar ele de dentro do express usando esse método ou você pode adicioná-lo por fora pelo npm, para entender melhor agora vamos fazer o seguinte se quisermos mandar um nome e ele tiver vazio você irá retornar uma mensagem falando que está vazio ou algo do gênero e se mandar um dado ele irá retornar um bem vindo com o dado. Caso a parte do seu front-end seja separada do seu servidor/ back-end você deve proporcionar o caminho todo o caminho onde o server é hospedado basicamente o domínio completo tecnicamente essa e uma parte do front-end mas é crucial que possamos entender.

const express = require('express')

const app = express()

let  {people} = require('./data')

//static assests

app.use(express.static('./methods-public'))

//parse form data

app.use(express.urlencoded({extended: false}))

app.get('/api/people', (*req*, *res*)=>{

*res*.status(200).json({success: true, data: people})

})

app.post('/login', (*req*, *res*)=>{

    console.log(*req*.body)

    const {name} = *req*.body;

    if(name){

        return *res*.status(200).send(`Welcome ${name}`)

    }

*res*.status(401).send('Please provide credentials...')

})

app.listen(5000, ()=>{

    console.log('Server is listening on port 5000...')

})

**MEHTOD POST – JAVASCRIPT**

Aqui nós usaremos javascript para mandar a requisição note algo interessante onde temos o content-type para a requisição do header e isso será uma applicattion/form-urlencoded e a razão de eu te mostrar isso é porque em javascript vai ser um pouco diferente só mantém esse primeiro como referência. Agora vamos navegar de volta para os arquivos estáticos basicamente para o nosso front-end e vamos observar a forma como foi feito pela forma javascript e o interessante aqui é que eu tenho o formulário, mas nesse caso será usado estritamente usando javascript e nos iremos usar o javascript para mandar nossa requisição http e isso é crucial, esta será a diferença em vez do formulário e claro o nosso content-type será diferente é por isso que lhe mostrei o exemplo passado mas você também irá notar logo abaixo nos temos nossa lista se você não acredita você pode observar em data.js os nomes então o que está acontecendo? Agora vá para o javascript html aqui é onde fica a logica do nosso front-end e observe index.html um pouco de css, nav e depois nos temos o formulário nesse caso nos não temos a ação e também não temos o método, o atributo nome ainda tem o valoro mesmo valore então eu posso presumir que este é o valor que está chegando com a minha requisição, também tenho um pouquinho do formulário de alerta no caso o axios aqui é onde iremos mostrar as funcionalidades e o resultado se você não é familiar tem um pacote chamado axios esse pacote deixa mais fácil configurar as requisições http em vez de usar o pacote do node o axios permite que uma api mais limpa e mensagens de erros mais simples no arquivo javascript.html passamos o script pelo cdn mas se quiséssemos poderíamos chama-lo como pacote deixando claro que este é um projeto front-end e para não te confundir foram feitos script tags ao em vez de uma fonte javascript separada como vocês podem ver eu estou selecionando o resultado isso é a minha div e eu tenho esse fetchPeople essa e a função que pega os dados das pessoas do servidor, agora note o caminho api/people é porque no servidor nos também temos essa rota então isso é para uma requisição get este é o padrão porém nesse caso nós não estamos fazendo com o navegador nós estamos fazendo com javascript então o axios tem o método get isso significa que no front-end nós estamos performando uma requisição get para o mesmo servidor, tenha certeza que os caminhos sempre sejam idênticos pois se não ocorrerá um erro e você não conseguirá dar um fetch nos dados. Agora vá em app.js e passe a rota post para /api/people preste atenção app.get /api/people e app.post /api/people não é a mesma coisa mesmo que tenha a mesma url de rota os métodos são diferentes em get estamos lendo os dados e em post estamos tentando adicionar dados no método post o status de sucesso é 201, primeiro mande um .send() simples com sucesso app.post('/api/people' ,(req, res)=>{res.status(201).send('Success')})

Agora digite um nome e confirme no botão e no modo desenvolvedor em network observe o arquivo criado people ele vai ter o status 201 e o Headers em Content-Type temos applicattion json no nosso front-end nós precisamos adicionar esse content-type quando estamos mandando os dados o que realmente interessante sobre o axios é que ele adiciona para a gente, em baixo perto do final nós temos esse request Payload e o nome é igual ao valor que você digitou, toda vez que você passar um dado ele estará vazio porém temos um truque aqui nesse caso como estamos manuseando casos formulário não estamos manuseando os dados em json, e sim nós sabemos como mandar os dados de volta em json mas aqui não estamos mexendo com os dados que virão em json e aqui é onde outro middleware de grande ajuda vem para nos ajudar app.use( express.json()) e abaixo nos temos /api/people e agora similarmente ao nosso login nos temos acesso aos dados do nosso formulário que estão sendo mandados com javascript no nosso req.body antes disso não tínhamos acesso e sim quando se trata de formulário aquela é uma maneira agora quando se trata de requisições http nesse caso mandada com javascript não nós não temos nenhum acesso e nos estamos recebendo um json e agora nós adicionamos um middleware especifico que cuida disso então agora nesse caso nós podemos adicionar mais logica, por exemplo caso estamos procurando por um nome e o nome é emitido então nós iremos mandar de volta aquele nome que foi criado para que podemos adicionar esse nome que foi criado para o front-end e de novo esse é um caso em que não estamos salvando esses dados em nenhum banco de dados mais pra frente iremos aprender como fazer isso mas por agora se um usuário tenta colocar ou passar nada terá uma mensagem de erro.

app.js

const express = require('express')

const app = express()

let  {people} = require('./data')

//static assests

app.use(express.static('./methods-public'))

//parse form data

app.use(express.urlencoded({extended: false}))

//parse json

app.use(express.json())

app.get('/api/people', (*req*, *res*)=>{

*res*.status(200).json({success: true, data: people})

})

app.post('/api/people' , (*req*, *res*)=>{

    const {name} = *req*.body;

    if(!name){

        return *res*.status(400).json({success: false, msg: 'please insert a data'})

    }

*res*.status(201).json({success: true, person: name})

})

app.post('/login', (*req*, *res*)=>{

    console.log(*req*.body)

    const {name} = *req*.body;

    if(name){

        return *res*.status(200).send(`Welcome ${name}`)

    }

*res*.status(401).send('Please provide credentials...')

})

app.listen(5000, ()=>{

    console.log('Server is listening on port 5000...')

})

**FERRAMENTAS ECONOMIZAR TEMPO REQUISIÇÕES HTTP**

No exemplo passado aprendemos as duas maneiras de passar uma requisição post, uma pelo formulário e a outra por javascript porém tem um grande problema nessa configuração e o problema aqui é se precisarmos criar uma interface front-end para testar alguma coisa ou qualquer rota a nossa produção vai ser bastante demorada porque vai levar mais tempo para criar uma interface do que criar uma rota, isso iria demorar de cinco a seis horas somente para testar essa rota, e pra isso temos umas ferramentas adicionais elas são o insomnia, o postman e o thunderman client esses programas são ótimos para testar Api’s , agora dentre qualquer um desses softwares faça uma nova requisição e digite a rota e o tipo de método do lado primeiro vamos testar o GET adicione a rota com o localhost aqui no exemplo localhost:5000/api/people agora observe que abaixo ele consegue mostrar o conteúdo da rota se estiver correto podemos testar isso imediatamente em poucos segundos ao invés de construir todo o front-end, agora para testar a rota POST vá em body e em seguida olhe o tipo de dado que é passado no caso pode ser xml, text, form, form-encode, binary, graphql ou json aqui no nosso exemplo estamos usando Json agora digite um dado aqui no exemplo passamos somente um nome {“name”: “luis”} para checar o tipo de dados que estamos passando vá em headers e olhe content-type e no momento em que clicamos em send nós temos a resposta.

**METHOD PUT**

Uma vez que estamos familiarizados com as ferramentas de teste para rotas vamos trabalhar agora nos outros dois métodos o PUT e o DELETE em baixo da rota POST adicione uma rota para PUT esse método serve para a edição dos dados ou atualização e se quisermos editar ou excluir precisamos passar o parâmetro da rota, onde efetivamente configuramos a rota por exemplo passamos a rota api/orders e simplesmente identificamos um item especifico no exemplo aqui o id, api/orders/:id, agora vá em app.js e crie a rota para acessar a lista em item especifico porque se esse item existir no corpo haverá dados para atualizar ou no caso de excluir não terá nada no corpo nós só iremos remover o item. Lembre-se que eu quero acessar os valores pelos parâmetros nós acessamos eles usando o req.params e a segunda coisa que vem ao falar de atualizar um dado, e que temos que passar algo no corpo porque por exemplo se eu mudar o nome de peter para gabriel ou outra coisa eu também preciso fornecer um valor, então você precisa entender que tem dois lados dessa requisição não somente estamos procurando um item especifico e para esses itens específicos nos usamos o params, é porque caso a gente quer o item numero um ou dois eu também preciso fornecer esse valor, porque de outra forma qual é ponto de se atualizar alguma coisa.

Vamos dizer que eu vou pegar o id pelo req.params, isso vai me dizer qual o item e a segunda coisa é se eu quero atualizar esse item e preciso do valor também usando o req.body os dois valores que iremos obter para nosso exemplo, só para fazer eles interessante de um console.log() neles.

 const {id} = *req*.params;

 const {name} = *req*.body;

 console.log(id, name)

Vá na ferramenta de teste que você está usando e mude a url selecionando qualquer uma pelo id por exemplo localhost:5000/api/people/1 agora se você mudar o numero do id por exemplo para 3 o numero do id irá mudar junto agora a partir desse ponto nós precisamos fazer a lógica para que ele mude corretamente, primeiro vamos configurar a lógica para salvar os dados locais com as pessoas do array eventualmente iremos trabalhar com banco de dados mas a ideia principal é exatamente a mesma, na ferramenta nos iremos mandar o id com a url e no corpo nós iremos mandar o valor então aquela parte não muda somente a funcionalidade aqui como temos aqui um simples javascript array eu só quero pegar aquela pessoa com aquele id se a pessoa não existir eu mando de volta uma mensagem de erro e se o id existir eu somente mudo nome/valor. A gente não irá estar provendo uma mensagem de erro se um valor não é inserido a gente irá cobrir isso quando mexermos com banco de dados no momento se fizéssemos isso seria uma grande perca de tempo primeiro vamos criar uma variável person e depois chamamos do array com o método find(), dentro dele crie uma arrow function.

app.put('/api/people/:id', (*req*, *res*)=>{

    const {id} = *req*.params;

    const {name} = *req*.body;

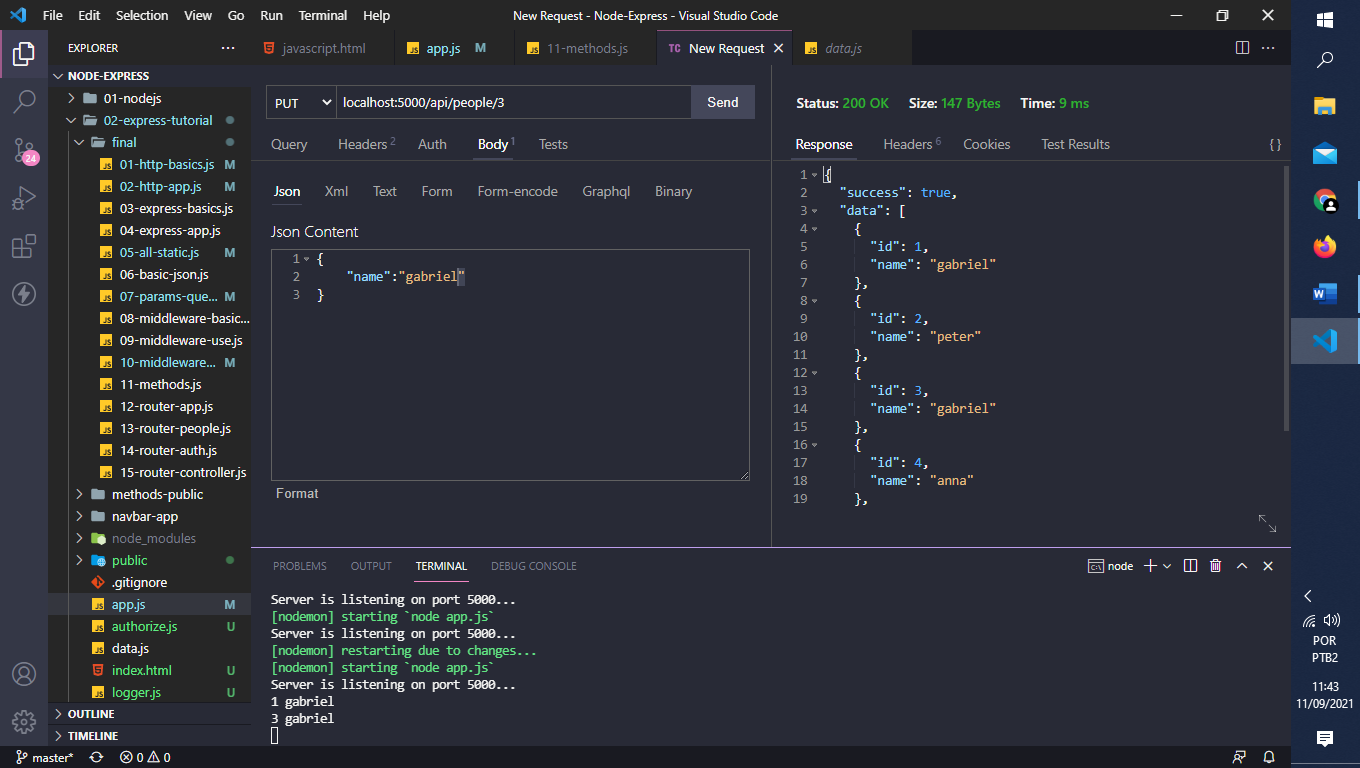
    const person = people.find((*person*)=>{

*person*.id === *Number*(id)

    })

})

Para não gastar tempo criando uma mensagem de erro nova olha um dos exemplos acima de mensagem de erro copie e mude os valores se necessário, se a pessoa não existir nos mandaremos de volta um 404, caso a pessoa exista nos vamos iterar sobre o array e para essa pessoa especifica que id combina eu irei mudar o nome desde que eu esteja esperando essa estar no corpo agora teste assim, se você quiser mudar o nome altere no body e o id da pessoa, porem observe que o dado não muda no arquivo data.js.



Put Method

app.put('/api/people/:id', (*req*, *res*)=>{

    const {id} = *req*.params;

    const {name} = *req*.body;

    const person = people.find((*person*)=> *person*.id === *Number*(id))

    if(!person){

        return *res*.status(404).json({success: false, msg: `no person with ${id} found`})

    }

    const newPeople = people.map((*person*)=>{

        if(*person*.id === *Number*(id)){

*person*.name = name

        }

        return *person*

    })

    console.log(id, name)

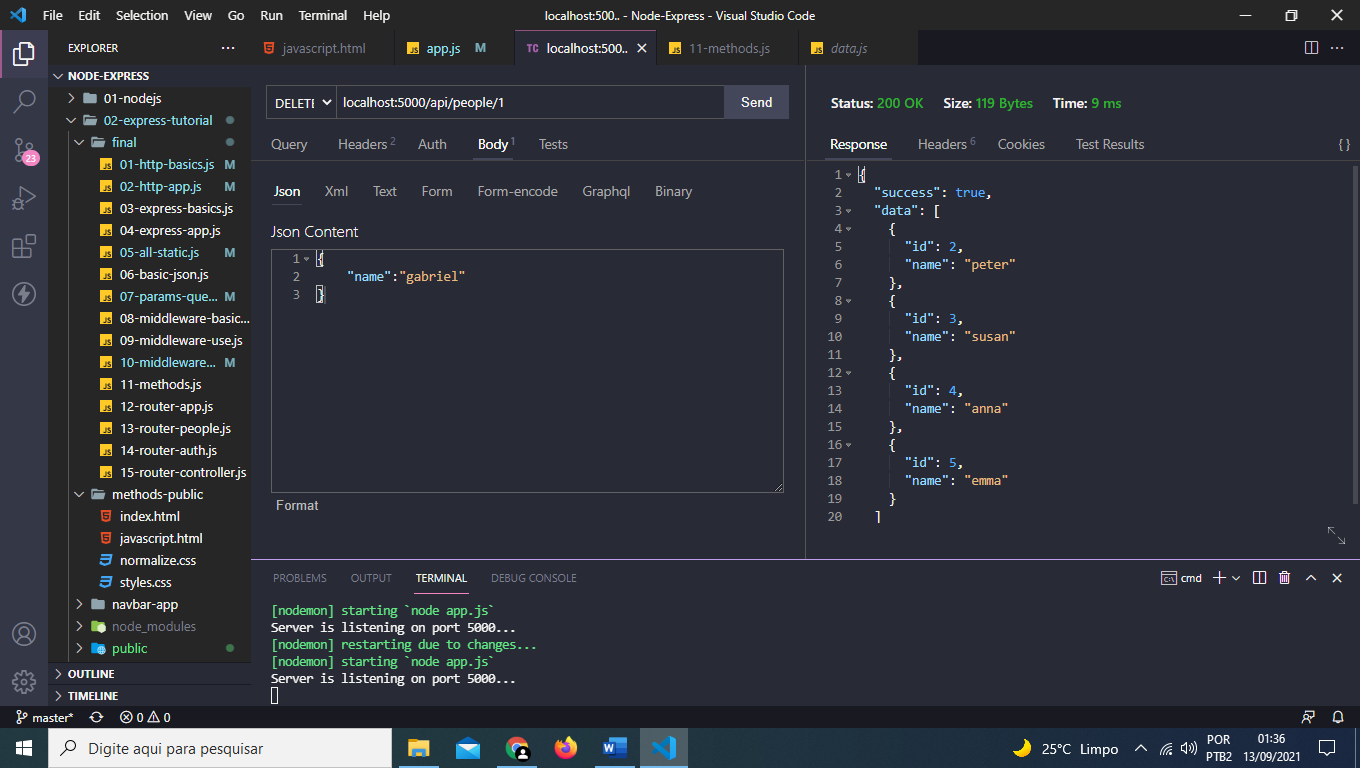
*res*.status(200).json({success: true, data: newPeople})

})

**DELETE METHOD**

Agora que aprendemos o método put vamos aprender o ultimo que é o delete se você tem uma lista você passa a url pra ela igual fizemos no exemplo usando o method put, a única diferença desse exemplo e do put é que quando deletamos não esperamos nada no corpo/body se o usuário chegar na url nós remos o usuário daquela lista e isso que iremos fazer. Uma coisa para fixar na cabeça é que mesmo que as rotas sejam quase a mesma mudando somente o método da rota elas sempre serão diferentes isso é crucial e tem que ser entendido, o método delete vai ser bem similar ao put na maioria das vezes agora pegue desde de const person até o if. Agora observe o que nos estamos tentando fazer estamos tentando achar uma pessoa que combina com id no parâmetro então para fazer isso primeiro mude em Number() para req.params.id e nesse caso a gente não desestruturou o id por isso precisamos inserir o req.params.id também insira no template string de res.json. Agora teste no software com uma url não existente por exemplo localhost:5000/api/people/abc se no body você a mensagem que você escreveu e o erro 404 e porque esta funcionando, e também a gente estava procurando por uma pessoa que não estava no array então agora eu volto para app.delete() e crio uma novas pessoa para filtrar o antigo array.

Ferramenta Thunder Client



Method Delete app.js

app.delete('/api/people/:id', (*req*, *res*)=>{

    const person = people.find((*person*)=> *person*.id === *Number*(*req*.params.id))

    if(!person){

        return *res*

            .status(404)

            .json({success: false, msg: `no person with id ${*req*.params.id} found`})

    }

    const newPeople = people.filter((*person*)=> *person*.id !== *Number*(*req*.params.id))

    return *res*

            .status(200)

            .json({success: true, data: newPeople})

})

**EXPRESS ROUTER**

Agora que aprendemos os métodos e como fazermos as rotas percebemos que está começando a ter um pequeno problema o nosso app.js está começando a ficar um pouco sobrecarregado com muitas funcionalidades e rotas, a solução para esse pequeno problema é usar o express router onde podemos juntar essas rotas e configurar as funcionalidade separadamente como controladores eventualmente quando trabalharmos com banco de dados irei mostrar o modelo comum o MVC basicamente é um padrão não uma regra mas é um padrão comum usado, nós não iremos usar nesse exemplo porque está faltando a parte do modelo porque não conectamos a um banco de dados, agora na requisição post /login eu quero que você primeiro mova ela para cima para entender o que você está fazendo, agora observe que nas rotas primeiro vem a /login depois vem a sequencias de /api/people não seria legal uma forma de agrupar todas essas rotas /api/people e também adicionar uma para o login e um arquivo separado, e a maneira que podemos fazer isso é configurando o router. A forma mais comum de se fazer é criar uma pasta com nome de routes e depois crie dois arquivos um para o login e outro para as pessoas agora vamos começar trabalhando em pessoa ou people.js, comece simples requerendo o express e ao invés de configurar com o app igual fizemos nos exemplos anteriores nós iremos usar a palavra router e nos deliberadamente pegamos a rota do express, essa vai ser uma rota instancia que vai cuidar das rotas agora nós iremos fazer parte da rota no app.js porém a parte das urls especificas iremos lidar com elas nesse arquivo people.js. agora pegue todas as rotas menos a /login copie e cole em people.js e depois troque o nome app por router e no final exporte esse arquivo usando o module.exports. Agora no app.js precisamos configurar um app.use para configurar o caminho das rotas que começam com /api/people usando meu arquivo people.js então primeiro pegue o let {people} copie e cole em people.js e em controllers nos vamos ter que fazer isso mais uma vez e exclua de app.js em seguida vá abaixo de app.use(express.json()) e passe a rota base que é /api/people chamando let people = require('./routes/people') e passe um app.use(‘/api/people’, people) esse requerimento para as rotas só funcionarão para os que se iniciam com /api/people a gente salva porém temos um problema, uma vez que configuramos esse router com um caminho base em people nós temos uma grande bagunça porque nós já temos a base para a nossa rota em app.js e agora em people.js para resolver isso simplesmente em people.js troque as rotas base por / trocando todas as rotas e deixando tudo o que fica depois de /api/people/ .

People.js

const express = require('express')

const router = express.Router();

let {people} = require('../data')

router.get('/', (*req*, *res*)=>{

*res*.status(200).json({success: true, data: people})

})

router.post('/' , (*req*, *res*)=>{

    const {name} = *req*.body;

    if(!name){

        return *res*.status(400).json({success: false, msg: 'please insert a data'})

    }

*res*.status(201).json({success: true, person: name})

})

router.put('/:id', (*req*, *res*)=>{

    const {id} = *req*.params;

    const {name} = *req*.body;

    const person = people.find((*person*)=> *person*.id === *Number*(id))

    if(!person){

        return *res*.status(404).json({success: false, msg: `no person with ${id} found`})

    }

    const newPeople = people.map((*person*)=>{

        if(*person*.id === *Number*(id)){

*person*.name = name

        }

        return *person*

    })

    console.log(id, name)

*res*.status(200).json({success: true, data: newPeople})

})

router.delete('/:id', (*req*, *res*)=>{

    const person = people.find((*person*)=> *person*.id === *Number*(*req*.params.id))

    if(!person){

        return *res*

            .status(404)

            .json({success: false, msg: `no person with id ${*req*.params.id} found`})

    }

    const newPeople = people.filter((*person*)=> *person*.id !== *Number*(*req*.params.id))

    return *res*

            .status(200)

            .json({success: true, data: newPeople})

})

*module*.*exports* = router

**CONTROLLERS – EXPRESS ROUTER**

Agora iremos separar as funcionalidades das nossas rotas em arquivos diferentes pois ele ainda está bagunçado olhe em auth.js nós só temos uma requisição por isso ele parece limpo agora em people.js vai estar uma bagunça pois temos muitas funcionalidades e rotas uma melhor configuração para resolver isso é separar as funcionalidades em arquivos diferentes, crie uma pasta chamada controllers para guardaras funcionalidades e dentro dessa pasta crie um arquivo pode ser com o mesmo nome para você saber a funcionalidade, vamos fazer só pra people.js pois auth.js só tem uma requisição e não é necessário agora vá no arquivo people.js em controllers e crie uma função para guardar as funções de people.js em routes assim const getPerson = (req, res)=>{ res.status(200).json({success: true, data: people})} repita o processo com todas as outras rotas pegando somente tudo o que vem depois de (req, res) e no final exporte guardando as funções criadas em um objeto. Pegue também o dado de let {people} = require(‘../data’) e bote em controllers no final o arquivo ficará assim.

controllers/people.js

let {people} = require('../data')

const getPerson = (*req*, *res*)=>{

*res*.status(200).json({success: true, data: people})

}

const postPerson =  (*req*, *res*)=>{

    const {name} = *req*.body;

    if(!name){

        return *res*.status(400).json({success: false, msg: 'please insert a data'})

    }

*res*.status(201).json({success: true, person: name})

}

const putPerson =  (*req*, *res*)=>{

    const {id} = *req*.params;

    const {name} = *req*.body;

    const person = people.find((*person*)=> *person*.id === *Number*(id))

    if(!person){

        return *res*.status(404).json({success: false, msg: `no person with ${id} found`})

    }

    const newPeople = people.map((*person*)=>{

        if(*person*.id === *Number*(id)){

*person*.name = name

        }

        return *person*

    })

    console.log(id, name)

*res*.status(200).json({success: true, data: newPeople})

}

const deletePerson = (*req*, *res*)=>{

    const person = people.find((*person*)=> *person*.id === *Number*(*req*.params.id))

    if(!person){

        return *res*

            .status(404)

            .json({success: false, msg: `no person with id ${*req*.params.id} found`})

    }

    const newPeople = people.filter((*person*)=> *person*.id !== *Number*(*req*.params.id))

    return *res*

            .status(200)

            .json({success: true, data: newPeople})

}

*module*.*exports* = {

    getPerson,

    postPerson,

    putPerson,

    deletePerson

}

Agora em people.js em routes faça a requisição das funções que você passou para controllers, dessa forma seu condigo ficará mais limpo e também mais fácil de entender temos duas formas de passar as rotas a primeira que usa mais código de linha mas e bem mais explicativa.

router.get('/', getPerson)

router.post('/', postPerson)

router.put('/:id', putPerson)

router.delete('/:id', deletePerson)

E a segunda que é com dois códigos de linha e usa uma função route do express que nela você só passa o caminho e na frente coloca os métodos e as funções das mesmas.

const express = require('express')

const router = express.Router();

const {

    getPerson,

    postPerson,

    putPerson,

    deletePerson

} =  require('../controllers/people')

//router.get('/', getPerson)

//router.post('/', postPerson)

//router.put('/:id', putPerson)

//router.delete('/:id', deletePerson)

router.route('/').get(getPerson).post(postPerson)

router.route('/:id').put(putPerson).delete(deletePerson)

*module*.*exports* = router