**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')

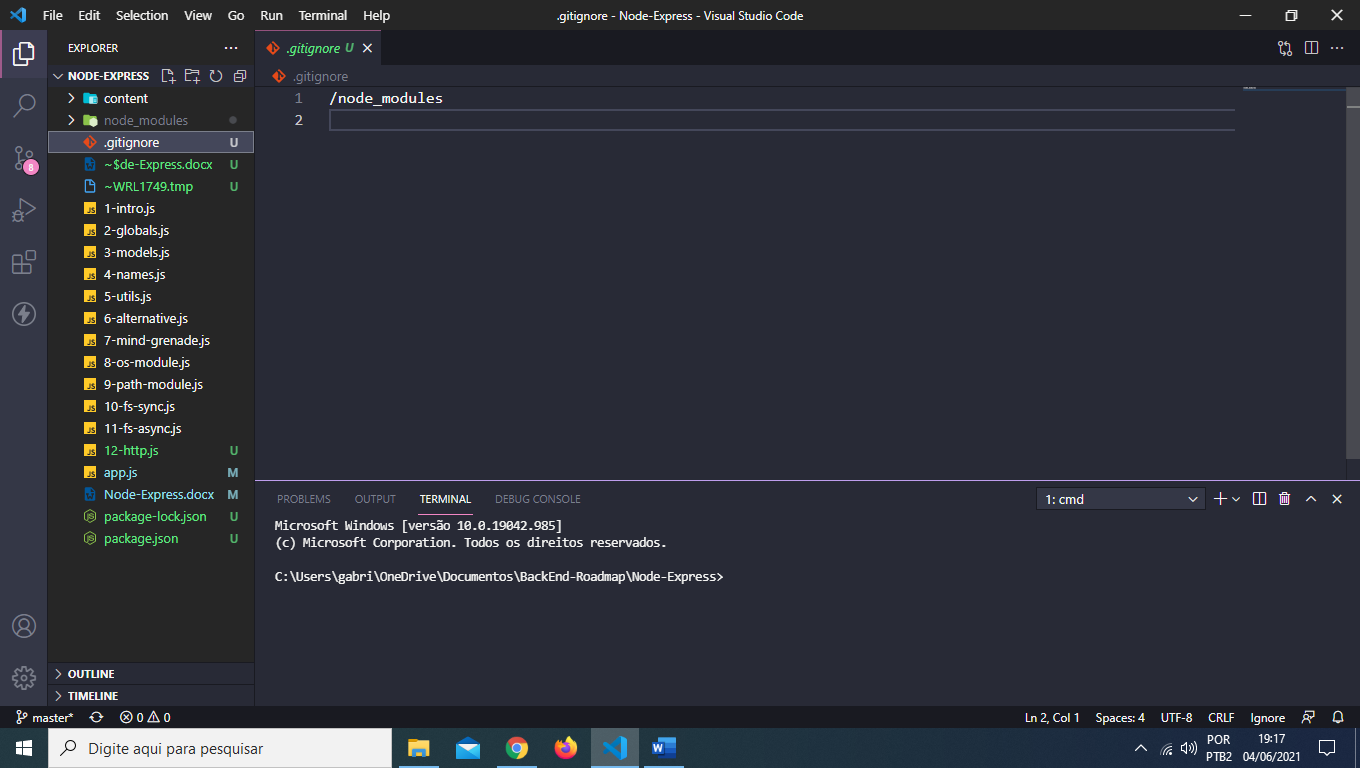
const items  = [1, [2,[3,[4]]]]

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:

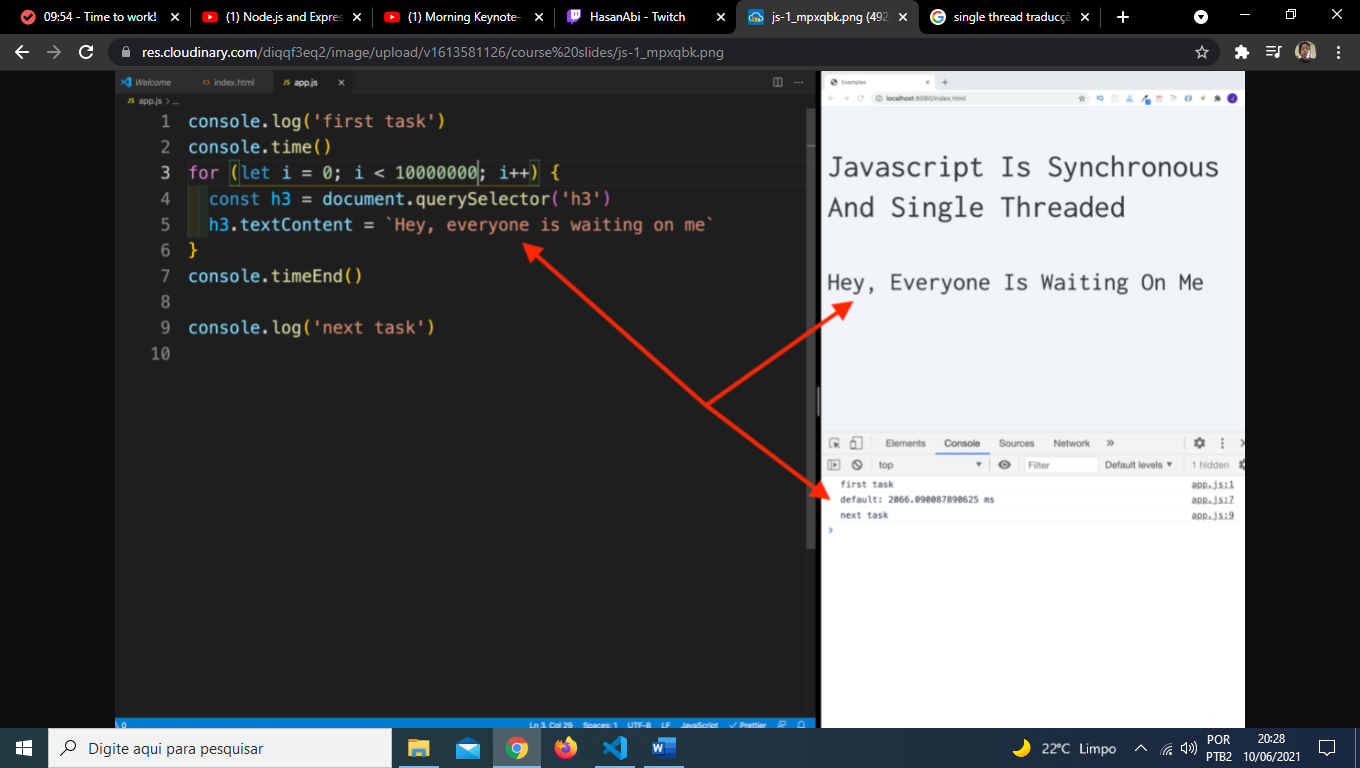
✦ Event Loop, Async Patterns, Events Emitter and Streams

✦ Main Concepts

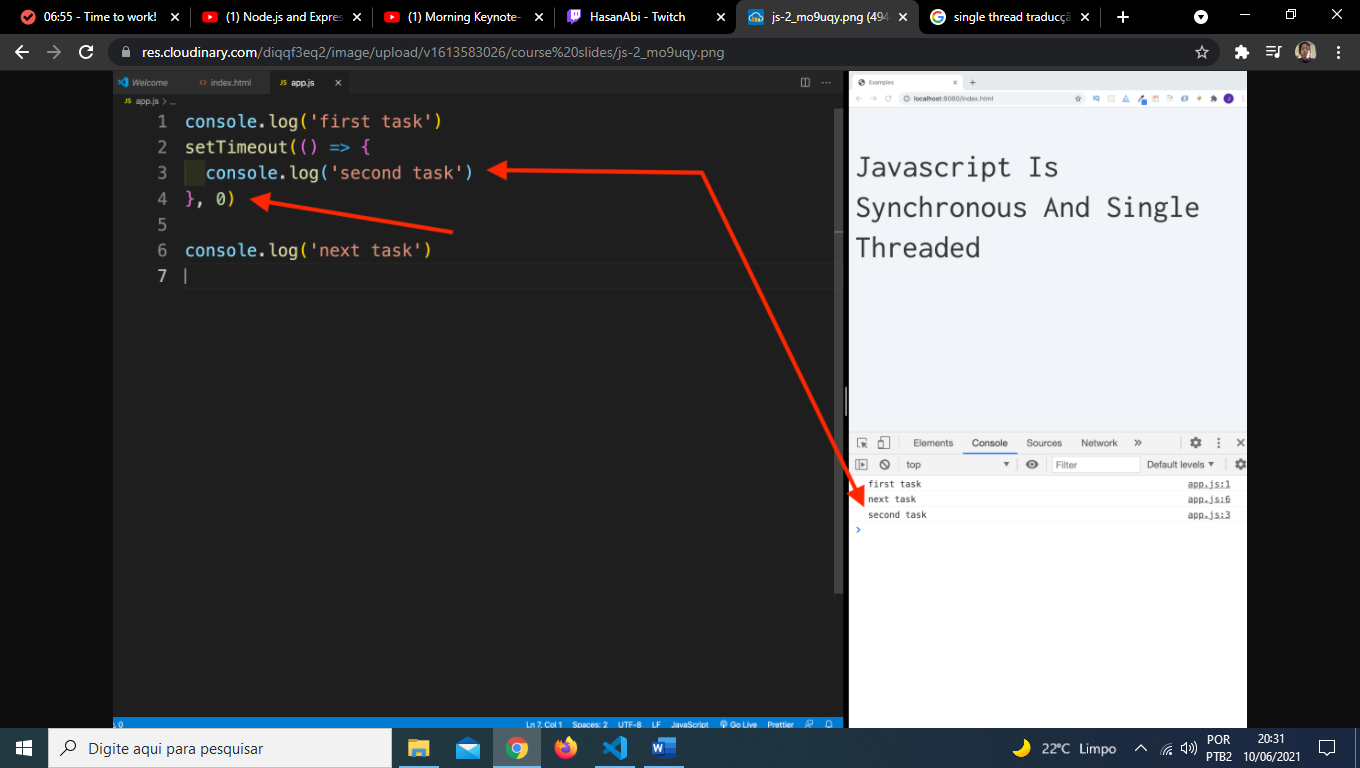
✦ Pre-Built Code

**EVENT LOOP**

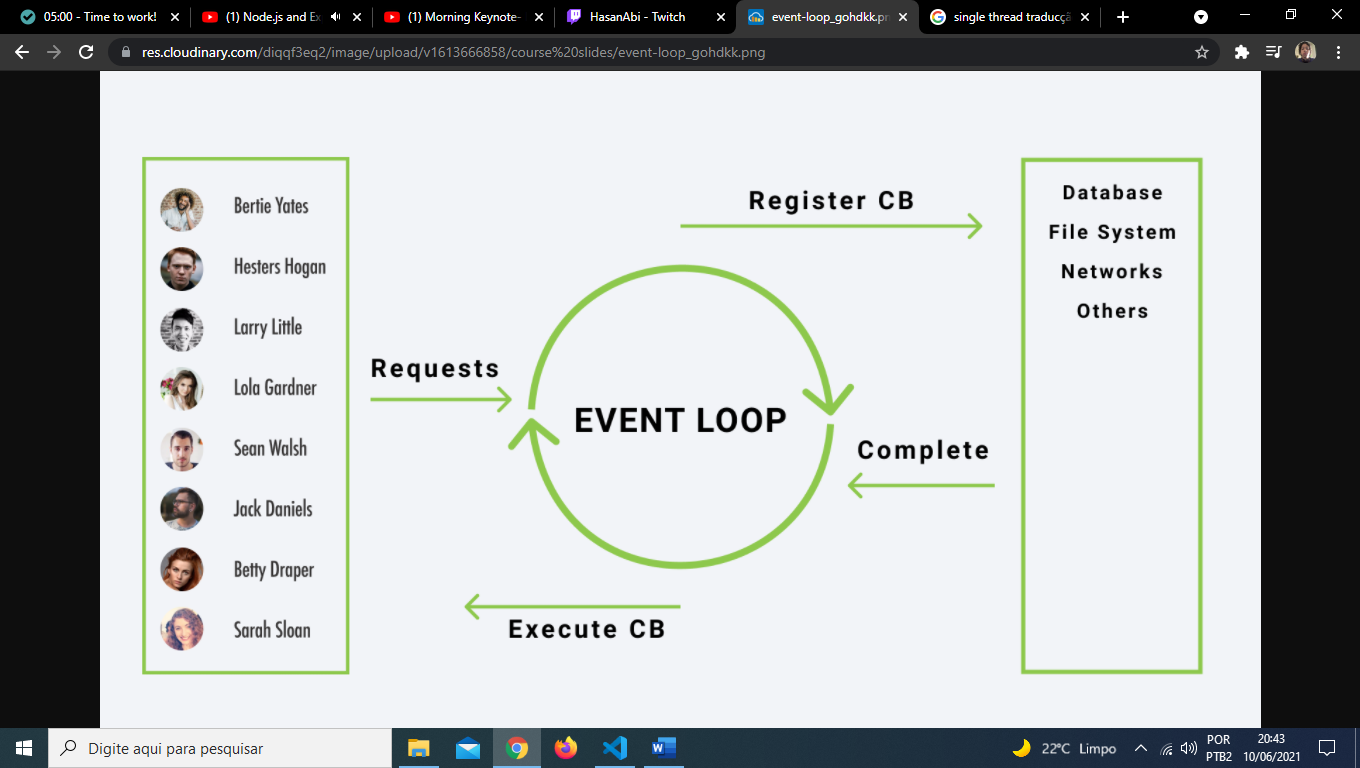
Event Loop em node.js é um tópico muito grande para ser abordado por completo então vamos ver só os conceitos básicos necessários. O event loop é o que permite o node.js produzir não bloqueáveis entradas/saídas de uma operação. Para procurar mais documentações vá ao seu navegador e digite node js event 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.

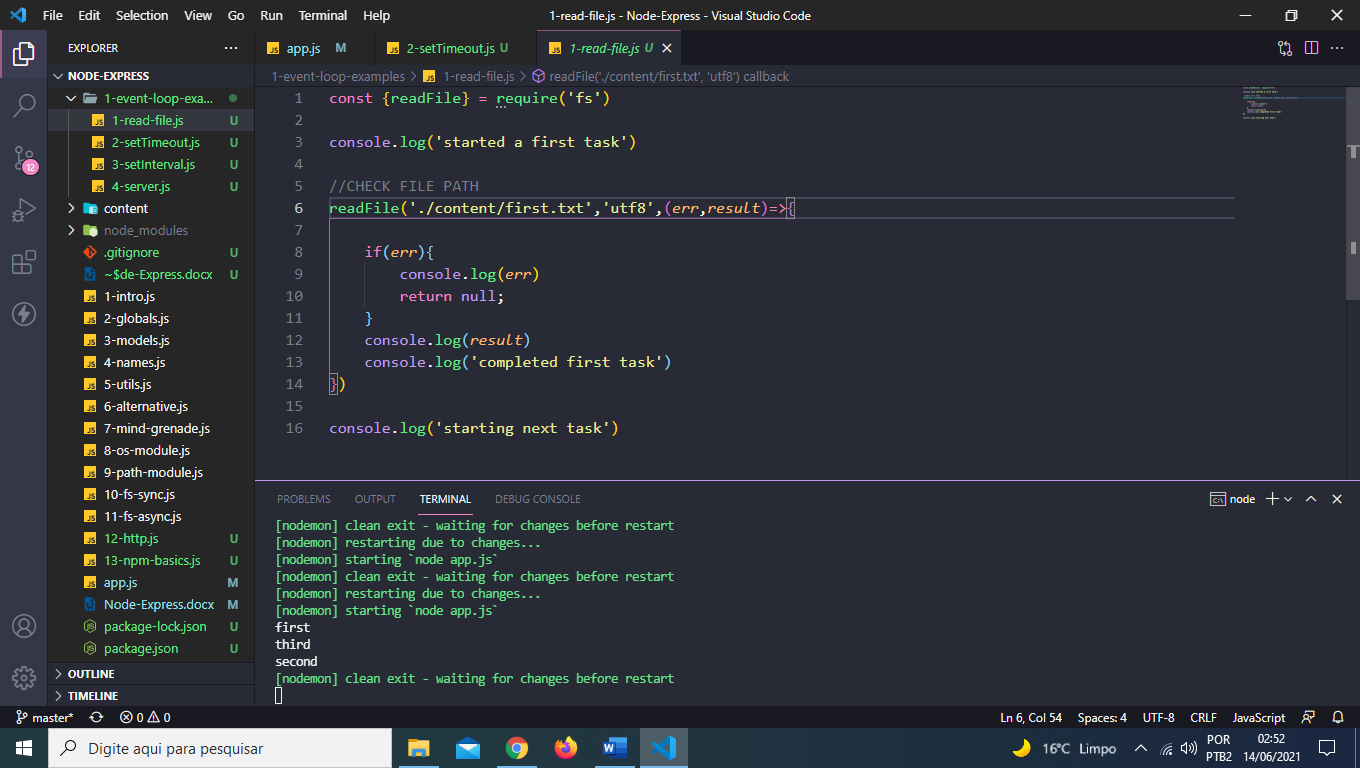


Vamos agora imaginar um cenário onde temos uma aplicação com usuários e eles estão entendidos então pedem requisições para a aplicação e cada vez que uma requisição chega o event loop é necessário para evitar esse tipo de cenário por exemplo vamos supor que um dos usuários tem uma tarefa demorada ou 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.

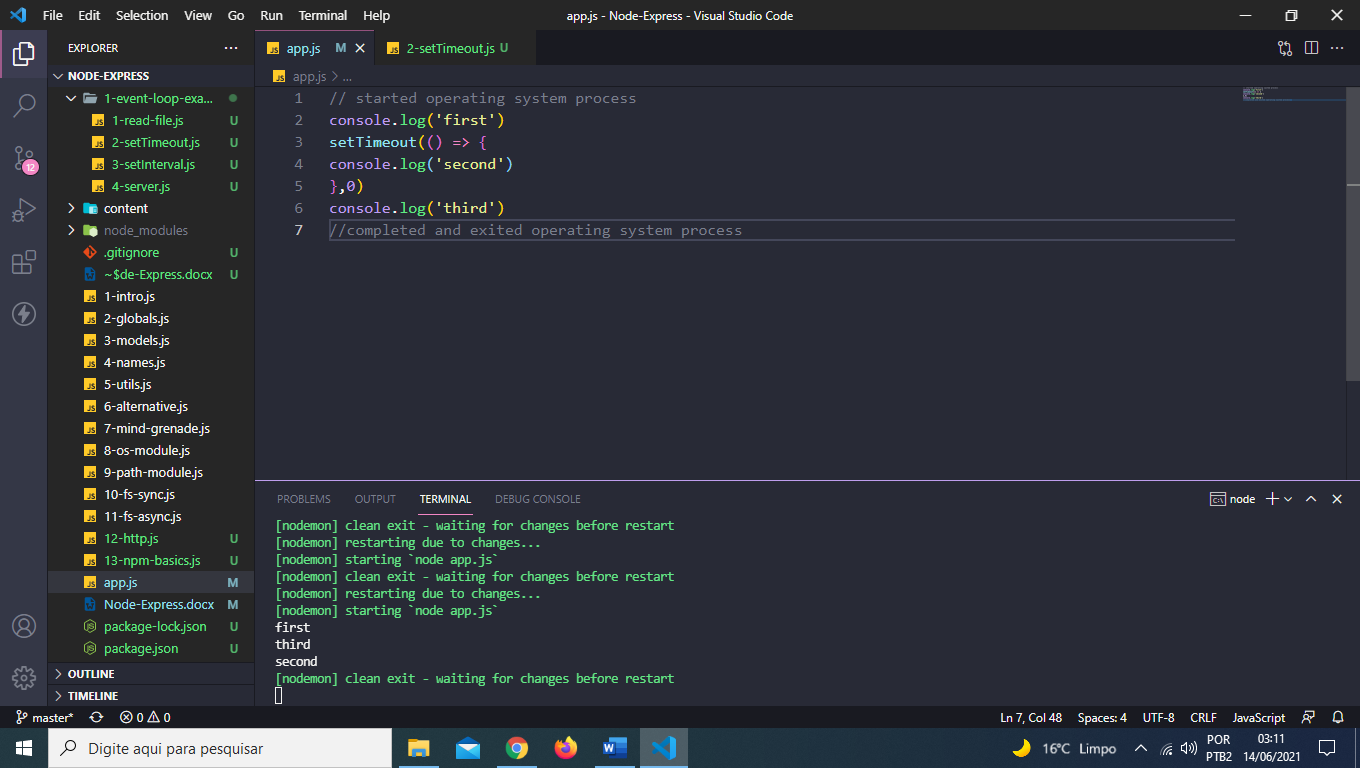


**EVENT LOOP CODE EXAMPLE**

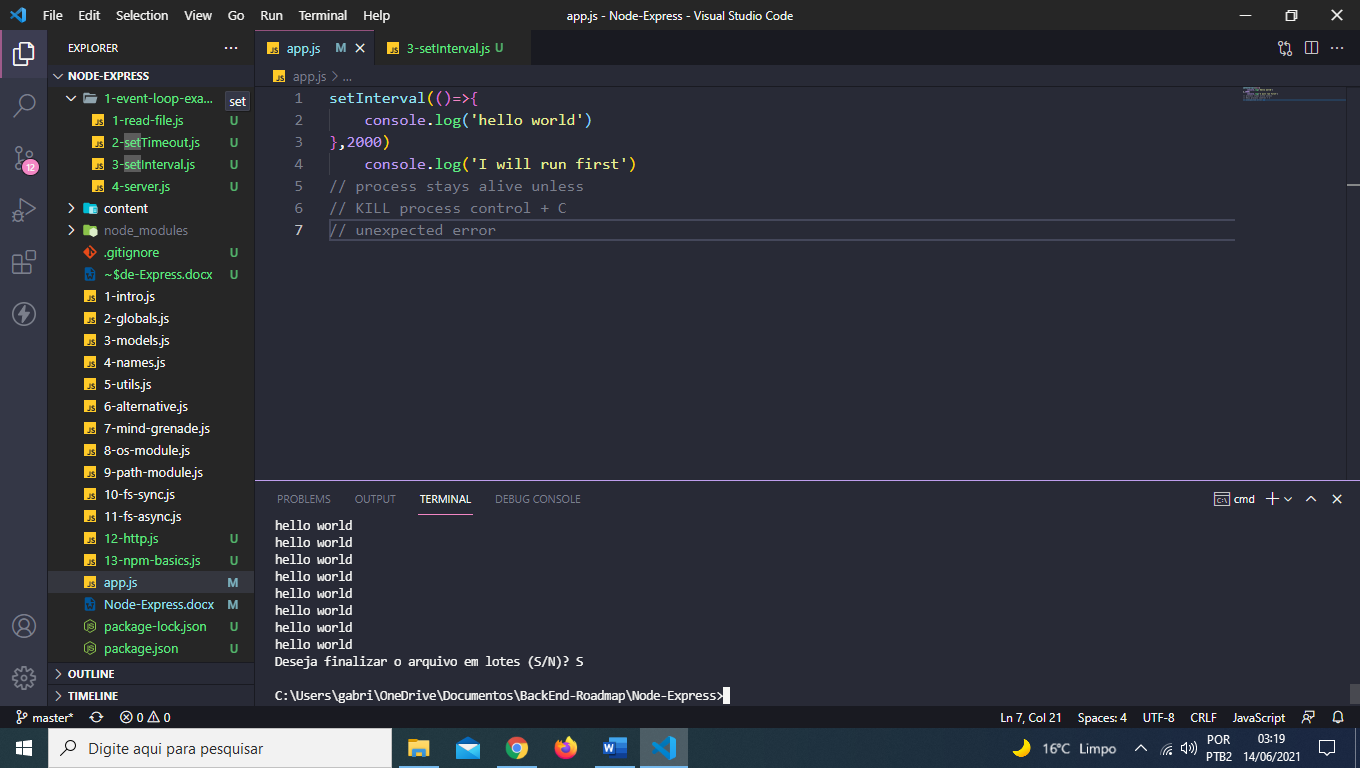
Tudo certo agora que aprendemos na teoria como funciona o event loop iremos fazer um na pratica para fixar mais na mente. O primeiro exemplo que vamos apresentar você deve ser familiar com ele pois é uma simples aplicação assíncrona do método readFile. Agora observe o código nota-se que primeiro criamos 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.



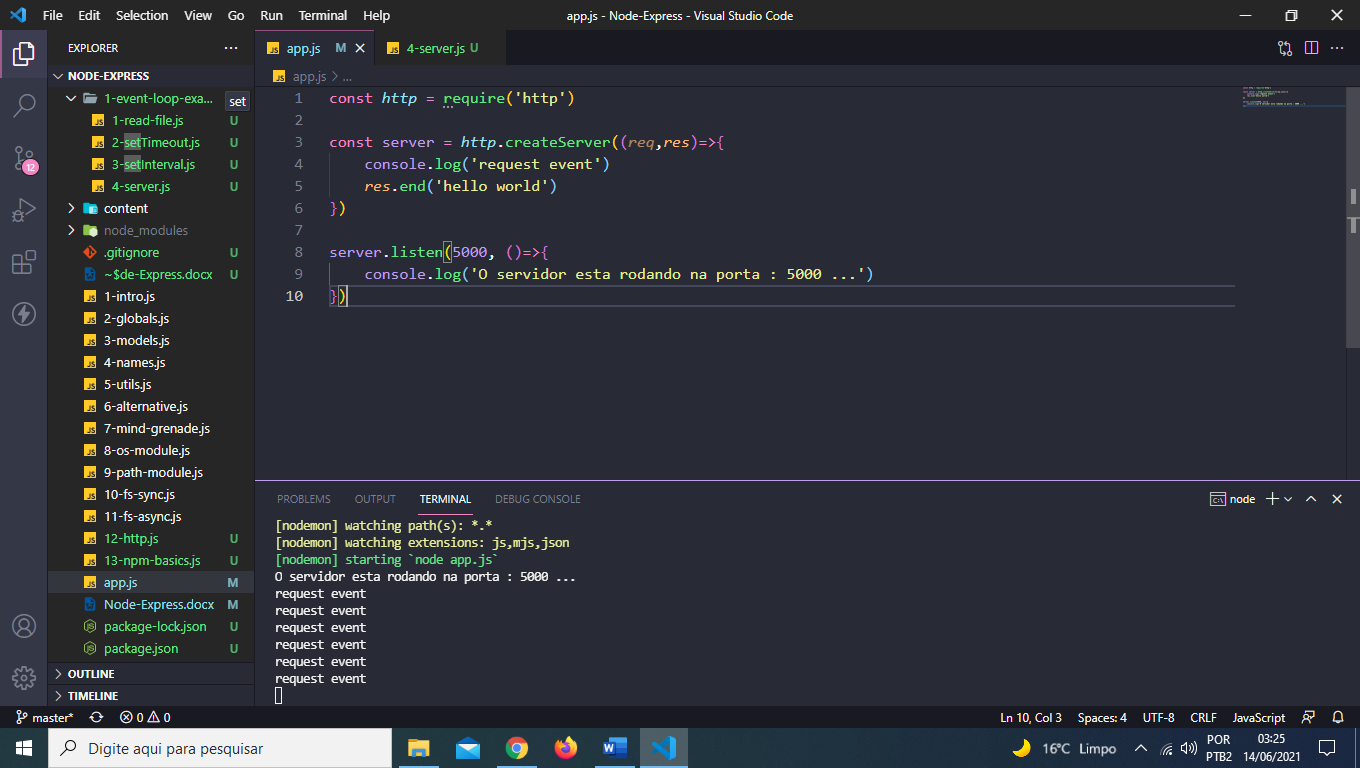
O próximo exemplo será um setTimeout primeiro crie um console.log informando a primeira tarefa depois crie um setTimeout com o tempo 0 e com um console.log dentro informando a segunda tarefa e de fora um terceiro console.log informando o inicio da terceira tarefa. Agora observe como colocamos o tempo 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.

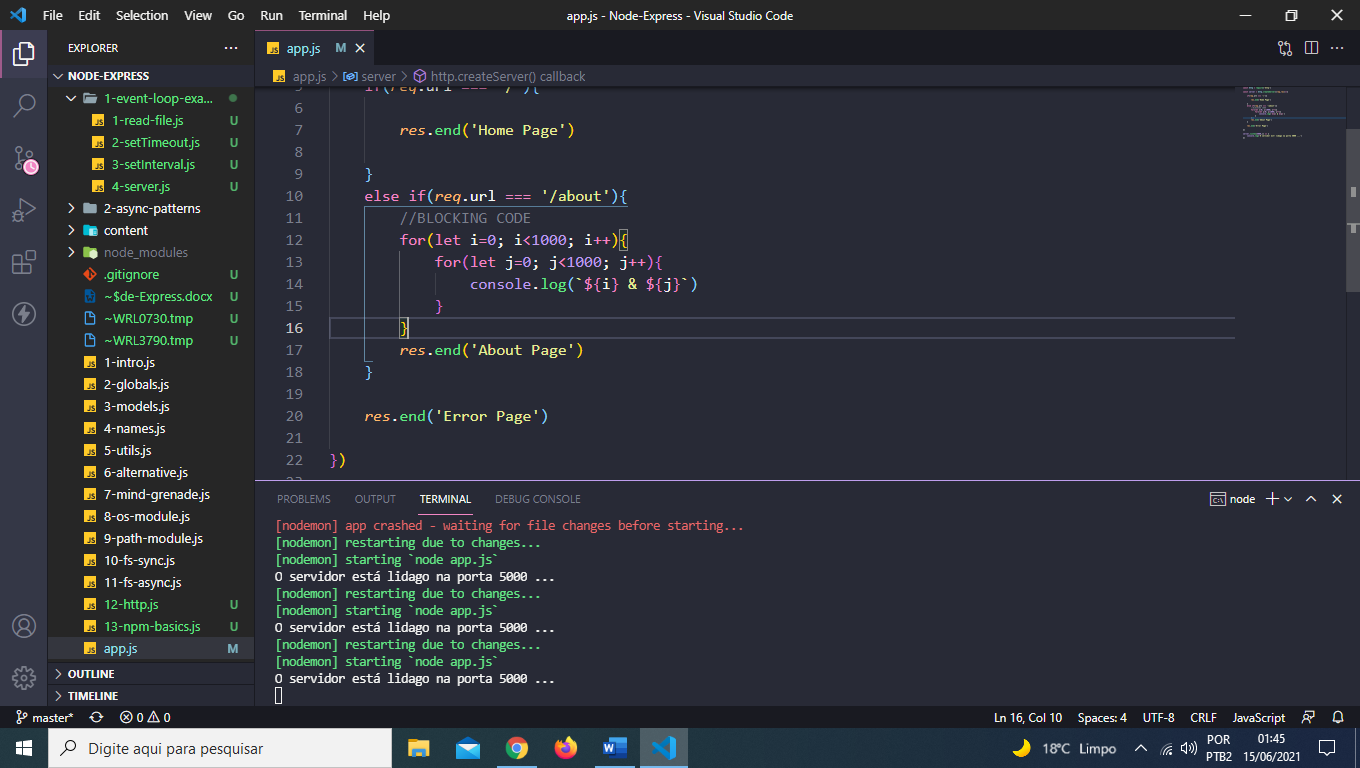


E o último exemplo é um servidor mostrando como ele se mantem vivo na hora de criar o servidor dentro da callback coloque um console.log para requisitar algo por exemplo e no res.end para a página digite um ‘hello word’ em seguida observe que quando entramos ele mostrará no terminal primeiro o caminho 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.



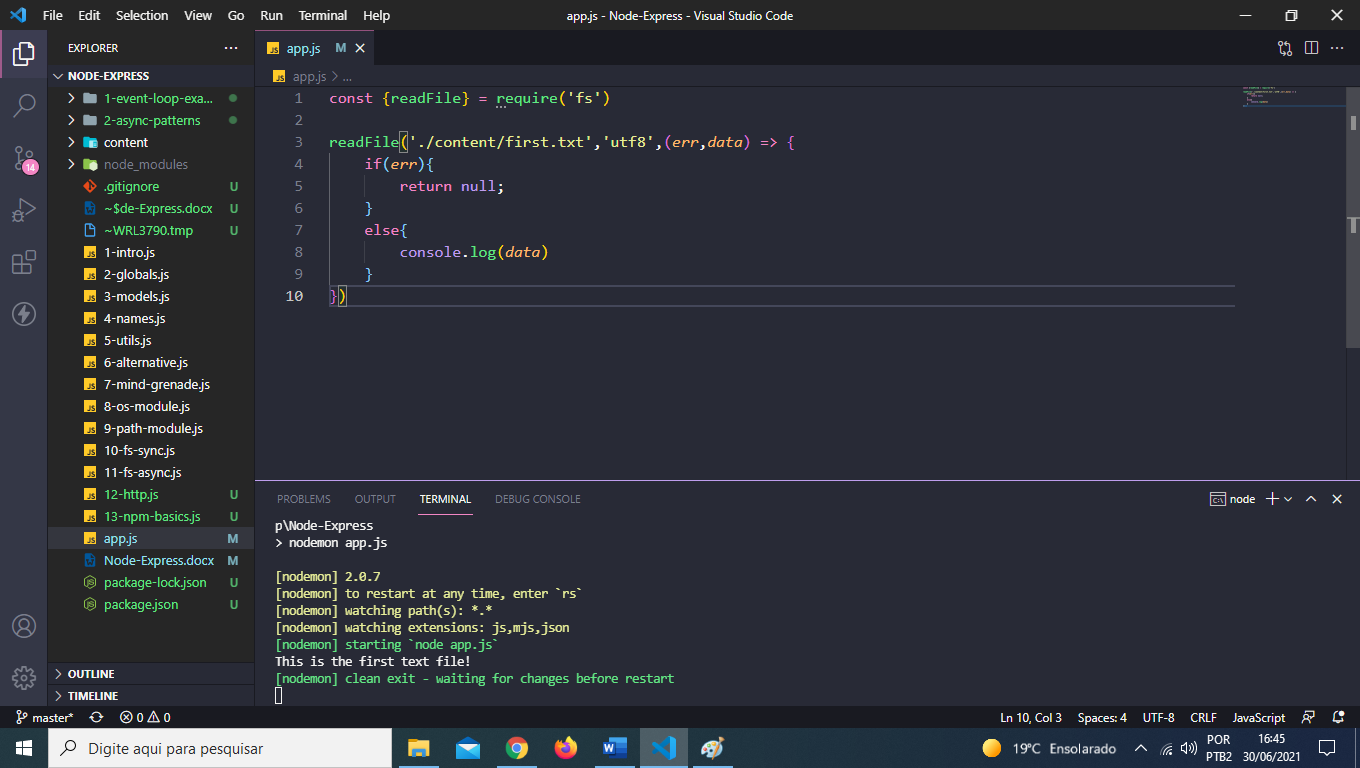
**ASYNC PATTERNS – BLOCKING CODE**

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.

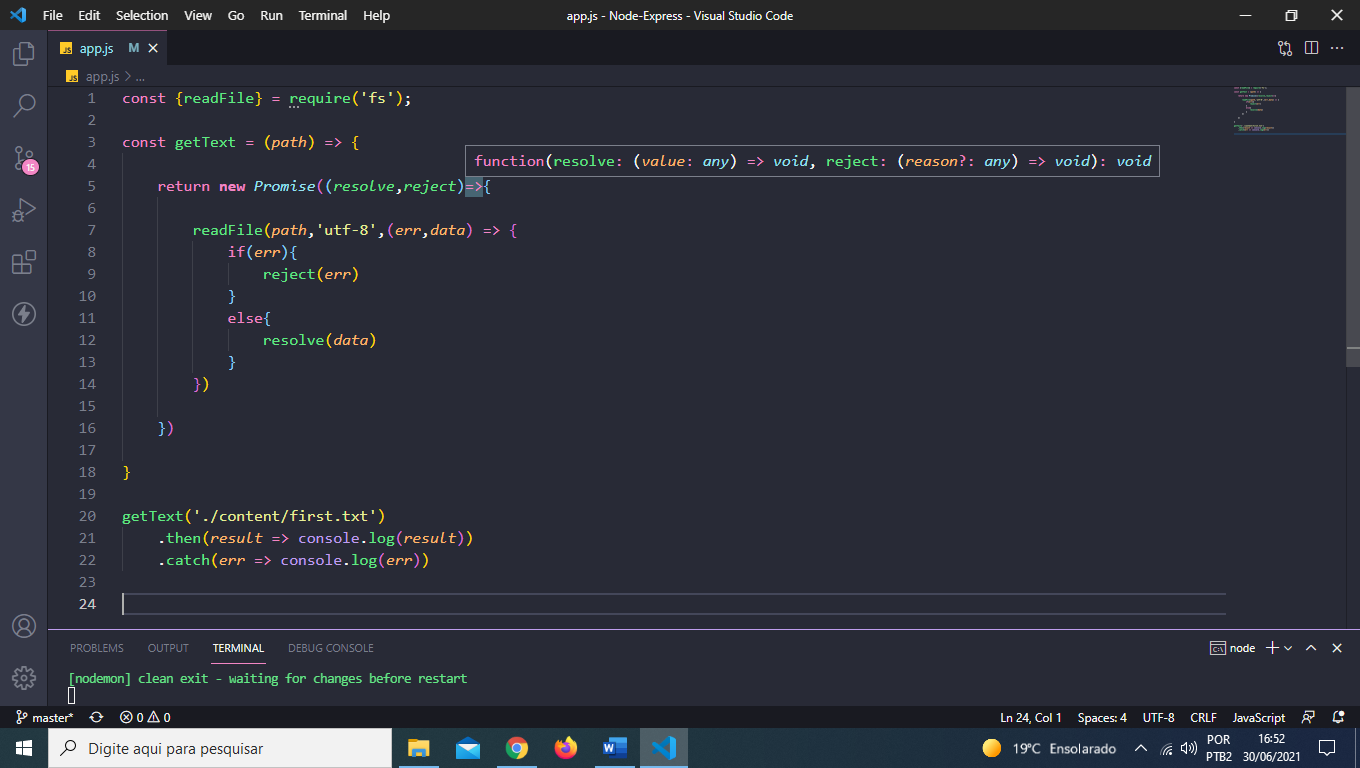


**ASYNC PATTERNS – SETUP PROMISES**

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.



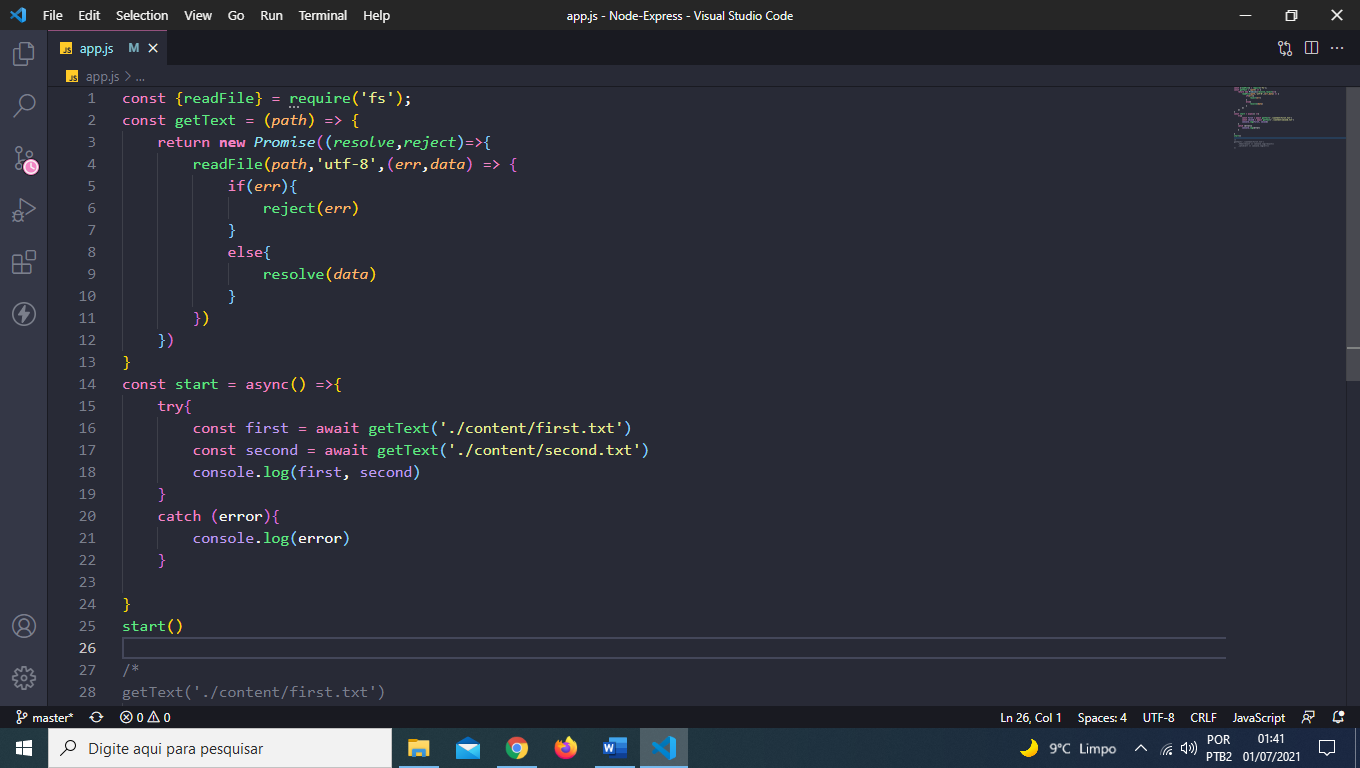
Mudando agora para uma forma de Promise ficará assim:



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))

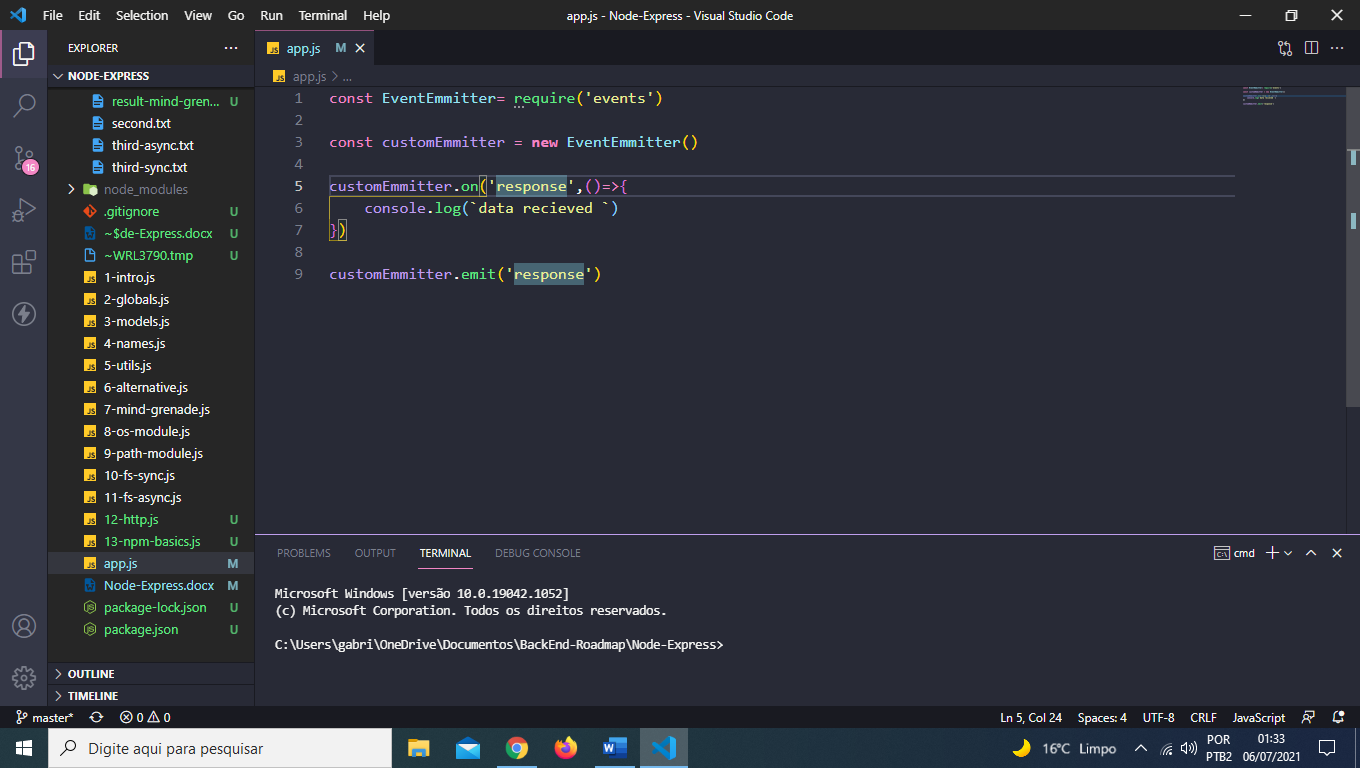
\*/

**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**