**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**