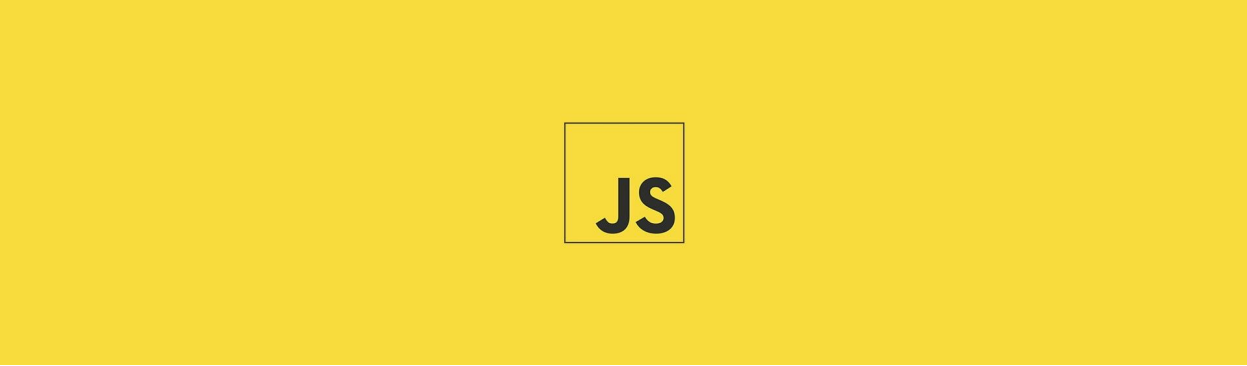


Javascript

> com Jonathan Comin Ribeiro

Sumário

O que vamos aprender no curso:

O que é Javascript?

É uma linguagem de programação interpretada e orientada a objetos. Amplamente utilizada em navegadores (Frontend, Client side), contudo, nos últimos anos, também vem ganhando espaço em aplicações Backend (Server side), como por exemplo o NodeJs.

Atualmente a 3º linguagem mais usada do mundo (segundo o site PYPL)

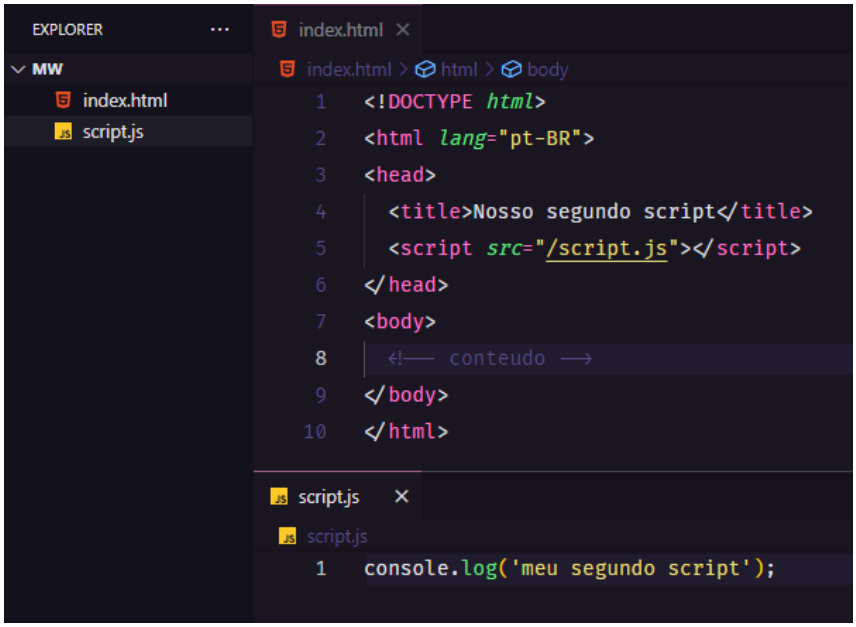
▼ Como e onde criar um script?

Para essa simples pergunta, teremos algumas variações de respostas válidas.

A primeira resposta que poderemos obter é criar o nosso script dentro da `HEAD` do nosso `HTML`, como o exemplo abaixo:

```
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <title>Nosso primeiro script incorpado</title>
    <script>
        console.log("Meu primeiro script");
    </script>
  </head>
  <body>
    <!-- conteudo -->
  </body>
</hmtl>
```

A segunda resposta que poderemos obter é criar o nosso `script` em um arquivo separado, com a extensão `.js`, e logo após, na `HEAD` do nosso `HTML`, incluimos ele via atributo `src` da tag `script`, exemplo:



▼ código do exemplo

`index.html`

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <title>Nosso segundo script</title>
  <script src="/script.js"></script>
</head>
<body>
  <!-- conteudo -->
</body>
</html>
```

`script.js`

```
console.log('meu segundo script');
```

E por fim, também podemos declara-los diretamente em eventos de elementos `HTML`, como o exemplo abaixo:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <title>Nosso terceiro script</title>
</head>
<body>
  <!-- conteudo -->
  <button onclick="console.log('clicou')">Click</button>
</body>
</html>
```

▼ Declarando variáveis

No JS, atualmente existem 3 tipos de declarações de variáveis, são elas:

`var`, `let` e `const`

```
var numero-1; //invalid
var numero_1 //valid
var 1_numero // invalid


let name_2;
const name_3;

//atribuições de valores
name = 'Jonathan';
name_2 = 'javascript';

alert(name_2);

var idade = 1;
var altura = 1.68;
var validacao = false;
var validacao_2 = true;

// Array -> Vetor
var notasDoSemestre = [4.38 , 5.0, 10];
var coisasNaCarteira = ['dinheiro', 'cartao', 10, 5.6];

var oCachorro = {
  olhos: 2,
  orelhas: 2,
  barulho: 'au au',
  caminha: function(){
    console.log('caminhando');
  }
};

// ARRAY
console.log(notasDoSemestre[0]); // 4.38
console.log(coisasNaCarteira[9999]); // undefined

//OBJETO
console.log(oCachorro.barulho); // au au
console.log(oCachorro['olhos']); // au au
```

Há algumas pequenas diferenças entre essas variáveis, por exemplo, o `var` sofre o que chamamos de ‘Hoisting’, enquanto `let` não.

Para lhe exemplificar de uma forma mais fácil o que seria esse tal ‘Hoisting’ vamos criar um exemplo:

```
var name = 'valor inicial';
if(true){
  var name = 'valor alterado';
}
console.log(name); //valor alterado
```

A variável name do tipo `var` , mesmo tendo seu valor iniciado como “valor inicial”, teve seu valor alterado dentro do escopo ({}) do if. Ou seja, é como se a declaração que aconteceu dentro do escopo (name) fosse feita globalmente, e não dentro do escopo.

Já com `let` esse tipo de problema não ocorre, pois o `let` respeita o escopo no qual foi declarado, como no exemplo:

```
let name = 'valor inicial';
if(true){
  let name = 'valor alterado';
}
console.log(name); //valor inicial
```

A variável `const` como o próprio nome já nos indica, se refere a um valor imutável, como por exemplo o valor de PI, onde nunca é alterado.

```
const PI = 3.14;
```

▼ Criando funções

As funções no javascript podem ser declaradas da seguinte forma:

```
//funções vazias (void)
function minhaFuncao(){
  console.log('minha primeira função');
}
//funções com retornos
function maiorDeIdade(){
  return 'SIM';
}
//funções que recebem parametros
function soma(a,b){
  return a+b;
}

//funções anônimas / closure
(function(){
  console.log('executo, porem logo não existo mais')
})();

//Arrow Functions
//() => {
//  console.log('arrow');
//}

var fulano = () => {
  console.log('de tal');
}
var siclano = (sobrenome) => {
  console.log(sobrenome);
```

```
}

// funções alocadas em variáveis
var salvandoFuncao = function (){
  console.log('minha funcao salva');
}

//chamando funcao alocada em variável
salvandoFuncao();
```

▼ Operadores de Comparação

```
// igual (==)
console.log( '1' == 1 ); //true

// diferente (!=)
console.log( '1' != 2 ); //true

// estritamente igual (===)
console.log( '1' === 1 ); //false

// estritamente diferente (!==)
console.log( '1' !== 1 ); //true

// maior que (>)
console.log( 2 > 1 ); //true

// maior ou igual que (>=)
console.log( 2 >= 1 ); //true

// menor que (<)
console.log( 2 < 1 ); //false

// menor ou igual que (<=)
console.log( 2 <= 1 ); //false

// operador OU ( || ), basta com que qualquer um dos lados seja true
console.log( true || false); // true
F ou F = F
F ou V = V
V ou V = V
V ou F = V

//operador AND (&&)
console.log( false && true ); //false
F e F = F
F e V = F
V e F = F
V e V = V

//operador NOT (!)
console.log( !false ); //true
```

▼ Estruturas de controle de fluxo | Desvios condicionais

```
var idade = 18;

if(idade < 17 ) {
  console.log('Menor de idade');
}
else if (idade === 18){
  console.log('Tem exatamente 18 anos');
}
else{
  console.log('Tem mais que 18 anos');
```

```
}

//////////////////////////////////////
var nome = 'Siclano';

switch(nome){
  case 'Siclano':
    console.log('Nome é Siclano');
    break;
  case 'Maria':
    console.log('Nome é Maria');
    break;
  default:
    console.log('Não localizamos o nome: ', nome);
    break;
}
```

▼ Estruturas de repetição

```
for (let index = 0, index < 10; index++){
  console.log(index);
}

var alunos = ['Jonathan', 'Bruna', 'Murilo', 'Gabriel', 'Gustavo', 'João', 'Lucas', 'Léo', 'Yan'];

for ( let i = 0 ; i < alunos.length ; i++){
  if(alunos[i] === 'Jonathan'){

    alunos[i] = alunos[i] + ' | Professor';

  }else{

    alunos[i] = alunos[i] + ' | Aluno';

  }
}

for ( let i = 0 ; i < alunos.length ; i++){
  console.log(alunos[i])
}

//////////////////////////////////////

var contador = 1;
while(contador < 10){
  console.log(contador);
  contador++;
}

//////////////////////////////////////

var cronometro = 10;
do{
  console.log(cronometro);
  cronometro--;
}while(cronometro > 0);

//////////////////////////////////////

var notas = [1,5,10];
for (let nota of notas){
  console.log(nota);
}

//////////////////////////////////////

var usuario = {
  nome: 'Jonathan',
  idade: 26
};
for (let chave in usuario){
  console.log('chave: ',chave,' | valor: ',usuario[chave]);
```

```
}

//////////////////////////////////////
var notas_2 = [1,5,10];
notas_2.forEach(function(nota){
  console.log(nota);
})
```

▼ Manipulando a DOM

Com JS temos o poder de manipular e alterar completamente a DOM do HTML.

Podemos por exemplo alterar, criar e remover elementos, adicionar ou remover estilização, adicionar eventos programaticamente, validar formulários, aplicar máscaras em inputs e etc. De modo geral, o JS é o responsavel por transformar o conteúdo estático de uma página em dinâmico.

Como dito acima, vamos por em pratica como obter nosso primeiro elemento via JS:

```
//buscando diretamente uma tag button
var button = document.querySelector('button');
//buscando por um ID
var input = document.querySelector('#inputComIDNome');
//buscando por um class
var div = document.querySelector('.container');
```

Como vimos acima, com a `document.querySelector` podemos buscar qualquer elemento na DOM utilizando dos mesmos conhecimentos que temos de seletores CSS.
A `document.querySelector` irá nos trazer a primeira ocorrencia da seleção na DOM, ou seja, um único elemento.

Para buscar todas as ocorrencias da seleção devemos utilizar a `document.querySelectorAll`

```
var anchors = document.querySelectorAll('a');
```

Com o elemento selecionado salvo em uma variável, podemos alterar basicamente todos os atributos que ele possuir, seja estilização, conteúdo, valor e etc…

```
var inputElement = document.querySelector('input');
inputElement.value = '123';

inputElement.name = 'novoName';
inputElement.id = 'novoId';

inputElement.classList.add('uma_nova_classe');
inputElement.classList.remove('uma_outra_classe');
inputElement.style.display = 'block';
inputElement.style.fontSize = 12;

//////////////////////////////////////

var h1Element= document.querySelector('h1');
h1Element.innerHTML = 'um novo título';
```

Assim como podemos manipular o elemento via JS, também podemos criar novos elementos a partir do JS:

```
var divElement = document.createElement('div');
divElement.innerHTML = 'conteudo';

var body = document.querySelector('body');
body.appendChild(divElement);
```

▼ Criando Eventos na DOM

Eventos são ações que foram ou serão executadas, sejam disparados por clicks feitos pelo usuário, quanto disparados programaticamente.

Para isso podemos utilizar eventos padrões de elementos HTML, ou até mesmo criar eventos personalizados.

Exemplo de evento padrão:

```
<button onclick="alert('usuario clicou!')">Click</button>
```

No exemplo acima, estamos utilizando do evento nativo do elemento “button”, no qual, ao ser clicado dispara a função vinculada a ele. Também podemos obter esse mesmo comportamento adicionando o evento totalmente via JS.

```
<button>Click</button>

<script>
  var buttonElement = document.querySelector('button');
  buttonElement.addEventListener('click', function(){
    alert('clicou');
  })
</script>
```

Os eventos mais comumente usados são

```
var form = document.querySelector('form');
form.addEventListener('submit', function(){
    alert('enviou o formulário');
})

var link = document.querySelector('a');
link.addEventListener('mouseenter', function(){
    alert('mouse em cima do link');
});
link.addEventListener('mouseleave', function(){
    alert('mouse saiu de cima do link');
})

var button = document.querySelector('button');
button.addEventListener('click', function(){
    alert('clicou no botão');
});

var input = document.querySelector('input');
input.addEventListener('change', function(event){
    console.log(event.target.value);
});
input.addEventListener('input', function(event){
    console.log(event.target.value);
});
input.addEventListener('focus', function(event){
    console.log('input com foco');
});
input.addEventListener('blur', function(event){
    console.log('input perdeu o foco');
});
```

Também é possível criar eventos personalizados:

```
document.addEventListener('meuEventoPersonalizado', ()=>{
  alert('Disparou meu evento personalizado');
})


document.dispatchEvent(new Event('meuEventoPersonalizado'));
```

▼ Manipulando Strings e Arrays

Strings e Arrays são valores compostos, e nos oferecem métodos nativos para buscar, alterar, incrementar e afins. Veremos alguns exemplos comumente usados:

String

```
var nome = 'Jonathan Comin Ribeiro';

var segundaLetra = vetor.at(1); // o

var primeiroSobrenome = nome.substr(9,5); // Comin

var primeiraLetraSobrenome = nome.slice(9,10); // C

var stringSeparadaPeloEspaco = nome.split(' '); // ['Jonathan', 'Comin','Ribeiro']

var contemNumero = nome.includes('123'); // false

var indexDaOcorrencia = nome.indexOf('on'); // 1

var indexDaOcorrencia_2 = nome.indexOf('3'); // -1
```

Array

```
var vetor = ['Jonathan', 26, 'Javascript'];

var idade = vetor.at(1); // 26

var tecnologia = vetor.at(-1); // 'Javascript'

var temMeuNome = vetor.includes('Jonathan'); // true

var indexMinhaIdade = vetor.indexOf(26); // 1

vetor.shift(); // [26,'Javascript'] -> remove o primeiro valor do array

vetor.unshift('typescript', 2); // ['typescript', 2, 'Jonathan', 26, 'Javascript'] -> adicionar novos valores no início do array

vetor.pop(); // remove o ultimo valor do array

vetor.push('Comin'); // adiciona o valor no final do array

// percorre cada item do array
vetor.forEach((valor) => {
  console.log(valor);
})

// procutar um valor ou objeto
var usuarios = [
  {id: 1, name: 'Jonathan'},
```

```
  {id: 2, name: 'Aluno'},
]
// função find utilizando function (nivel 1)
var instrutor = usuarios.find(function(usuario){  return usuario.name === 'Jonathan'   }); // {id: 1, name: 'Jonathan'}
// função find utilizando arrow function especificando o retorno dentro da função (nivel 2)
var instrutor = usuarios.find(usuario => {  return usuario.name === 'Jonathan'   }); // {id: 1, name: 'Jonathan'}
//função find utilizando arrow function com retorno implícito (nivel 3)
var instrutor = usuarios.find(usuario => usuario.name === 'Jonathan'   ); // {id: 1, name: 'Jonathan'}
```

▼ Usando Module Sintax

Os novos navegadores já possuem suporte ao Module Sintax, o que nos permite organizar melhor nossos códigos, fazendo a importação somente quando necessário.
Para isso nosso arquivo `.js` deve conter a palavra reservada `export` nos elementos que quer deixar “público” para que outro arquivo javascript possa “importar”, e nesse momento usamos a palavra reservada `import`

```
//arquivo chamado meuScript.js

export var nome = 'Jonathan';

export function soma(a,b){
  console.log(a+b);
}

export default exportacaoPadrao = 'meu modulo';
```

```
//arquivo chamado index.js

import exportacaoPadrao, {nome, soma} from '/meuScript.js';

console.log(exportacaoPadrao); // 'meu modulo'

console.log(nome); // 'Jonathan'

soma(1,2); // 3
```

▼ Template String

Os Template String são literais delimitados com caracteres de aspas simples `` ` ``, permitindo criar strings de várias linhas, e interpolação de valores de variáveis em seu corpo.

Exemplo:

```
var nome = 'Jonathan';
var idade = 26;
var linguagem = 'javascript';

var um_texto = `Olá meu nome é ${nome},
  tenho ${idade} e vou ser seu professor de ${linguagem}
` ;
```

▼ Tratamentos de erros com Try, Catch e Finally

Quando falamos em erros estamos falando de algo como:

- Tentar converter um dado de um tipo para outro tipo;
- Tentar buscar a dados do usuário do backend via API;

Como conseguimos perceber, sempre que “tentarmos” alguma coisa, provavelmente precisaremos de um try…catch para fazer com que nossa aplicação não gere problemas em sua execução.

E para previnir esses possiveis erros podemos:

```
try{
  //tente usar uma função que não existe
  getCPF();
}catch(err){
  //caso algo falhe, venha aqui
  console.log(err); //ReferenceError: getNumer is not defined
}

try{
  //tente ler uma informação a partir de uma variavel undefined
  var objeto = undefined;
  objeto.idade;
}catch(err){
  console.log(err)
}
```

O termo `finally` no final do conjunto `try catch` é opcional, e é utilizado quando queremos que algo aconteça indendente do resultado do `try catch`, e ele é chamado sempre no fim da execução do bloco `try catch` independente de haver um comando `return` . Podemos ver melhor seu funcionamento no exemplo abaixo:

```
function getUsuario(){
  try{

    var usuario = { nome: 'jonathan' };
    return usuario.nome;

  }catch(err){

      console.log('algum erro');

  }finally{

      console.log('acabou de executar a função getUsuario()');

  }
}

console.log('Retorno da função getUsuario: ', getUsuario());
// acabou de executar a função getUsuario()
// Retorno da função getUsuario: jonathan
```

▼ Persistindo dados no navegador com localStorage e sessionStorage

Como sabemos, quando o usuário abre nosso index.html, todo nosso conteúdo é carregado do zero, e com isso, trazemos alguns problemas para o usuário, como por exemplo:

- O usuário adiciona um produto no carrinho de compras, aperta f5 e o carrinho de compras zera.

- O usuário faz login em um sistema, na tela logada ele aperta f5 e é surpreendido quando o sitema diz que ele esta deslogado.

Para contornar esses problemas, podemos persistir esses dados na localStorage ou na sessionStorage.

Por mais que os dois métodos se pareçam muito, há diferença entre eles:

- localStorage: os dados não expiram, e se mantém mesmo ao abrir novas abas, ou fechar o navegador.
- sessionStorage: os dados expiram quando o navegador é fechado. Ao abrir uma nova aba é criada uma nova sessão. Ideal para persistir dados quando a página é recarregada.

Como utilizar:

```
var nome = 'Jonathan';
localStorage.setItem('local_save', nome);
var nome_restored = localStorage.getItem('local_save');
localStorage.removeItem('local_save');
```

```
var nome = 'Jonathan';
sessionStorage.setItem('session_save', nome);
var nome_restored = sessionStorage.getItem('session_save');
sessionStorage.removeItem('session_save');
```

Exemplo real de uso:

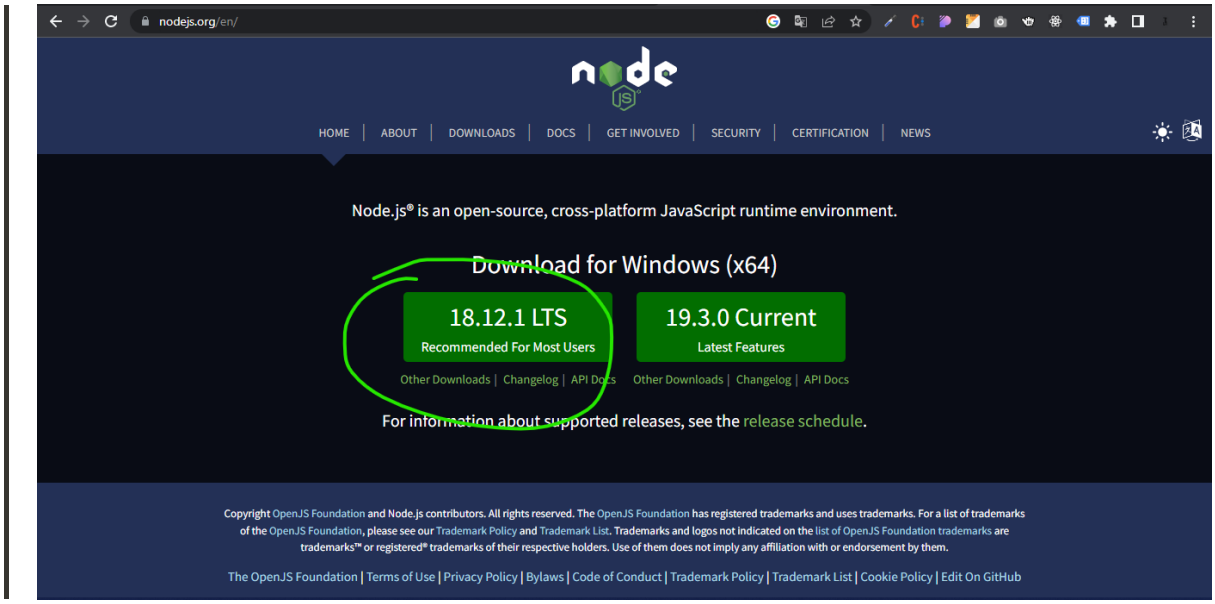
```
var user;
var saved_user = localStorage.getItem('user_saved');
if(saved_user){
  user = saved_user;
}

console.log(user);
```

Projeto Frontend e Backend

▼ Rodando nosso projeto localmente

Backend Nodejs

> É necessário ter o nodejs instalado em sua maquina. Site oficia aqui.

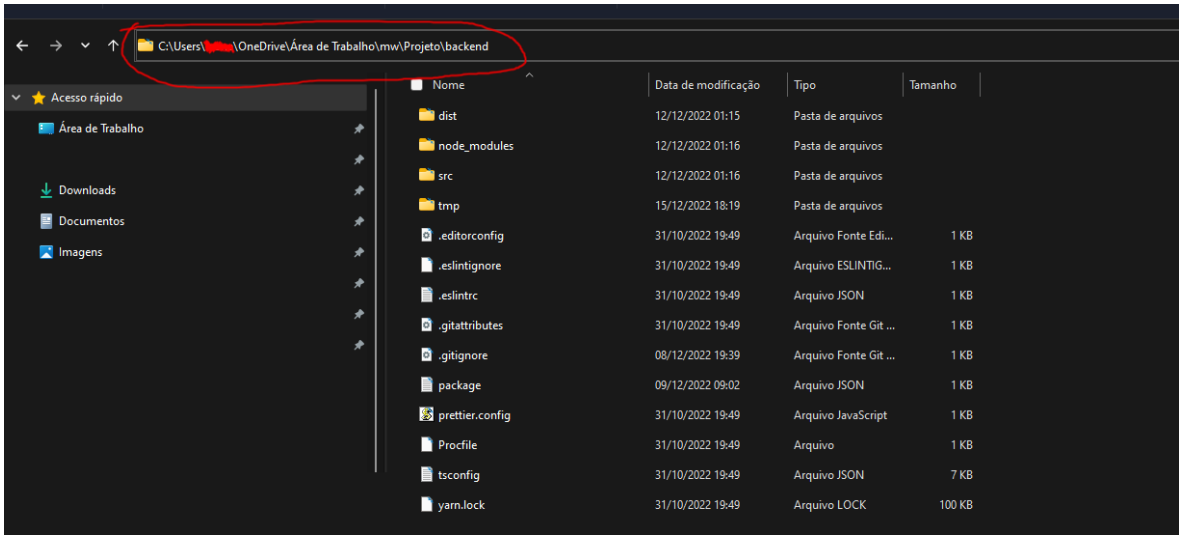


Para rodar nosso backend precisamos de duas coisas:

- Saber o caminho completo até nosso projeto backend.
- Iniciar o servidor nodejs nesse caminho.

▼ Descobrindo o caminho completo até o projeto backend

Basta ir até a pasta e clicar na área destacada em vermelho, irá nos mostrar o caminho completo até essa pasta. Basta copiar esse endereço inteiro.

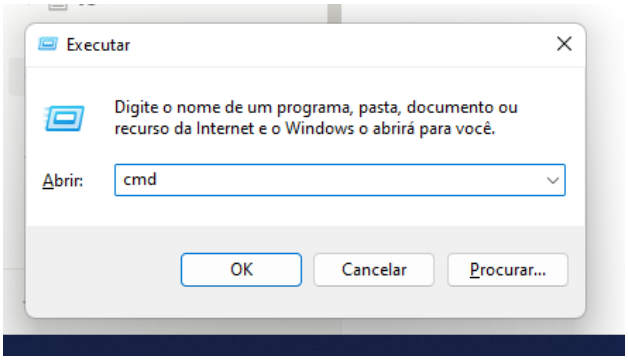


▼ Iniciando servidor nodejs

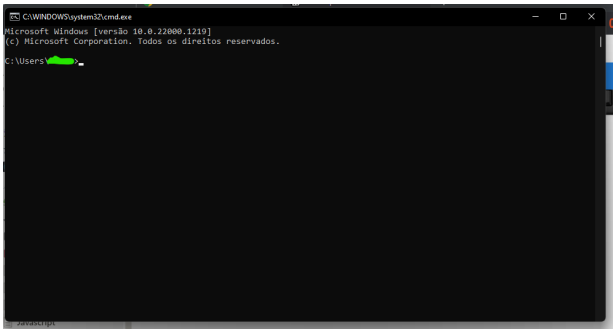
▼ Abra o `cmd`

pressione o botão windows + r

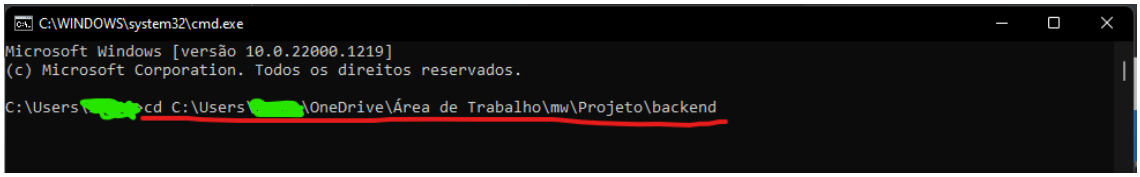
digite cmd e pressione OK



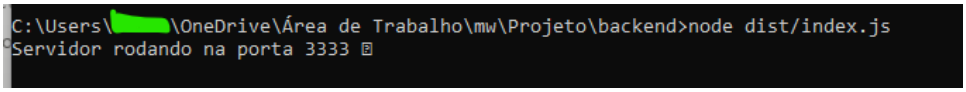
Deve abrir uma tela como essa



▼ No cmd digite o comando “cd” e informe o caminho dos arquivos backend e de um “enter”: Ex:
`cd C:\Users\#seuusuario#\OneDrive\Área de Trabalho\mw\Projeto\backend`



▼ No cmd digite “node dist/index.js”

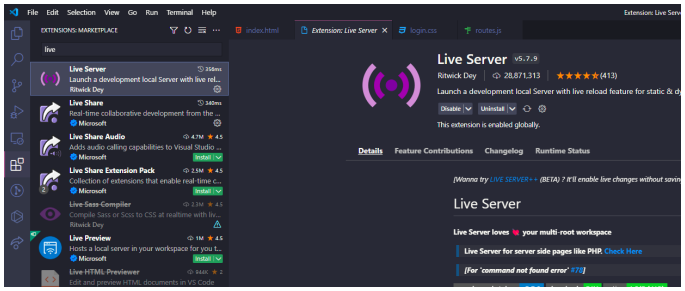


No final ele deve apresentar essa mensagem de “Servidor rodando na porta 3333”

▼ Pronto. Deixe o cmd aberto enquanto precisar do backend rodando.

Frontend Vanilla JS

> Necessário ter o Live Server instalado como extensão no seu VSCode

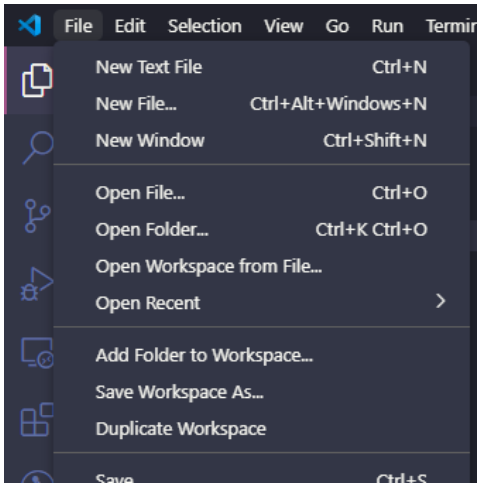


Para iniciar nosso Frontend precisamos de 2 passos:

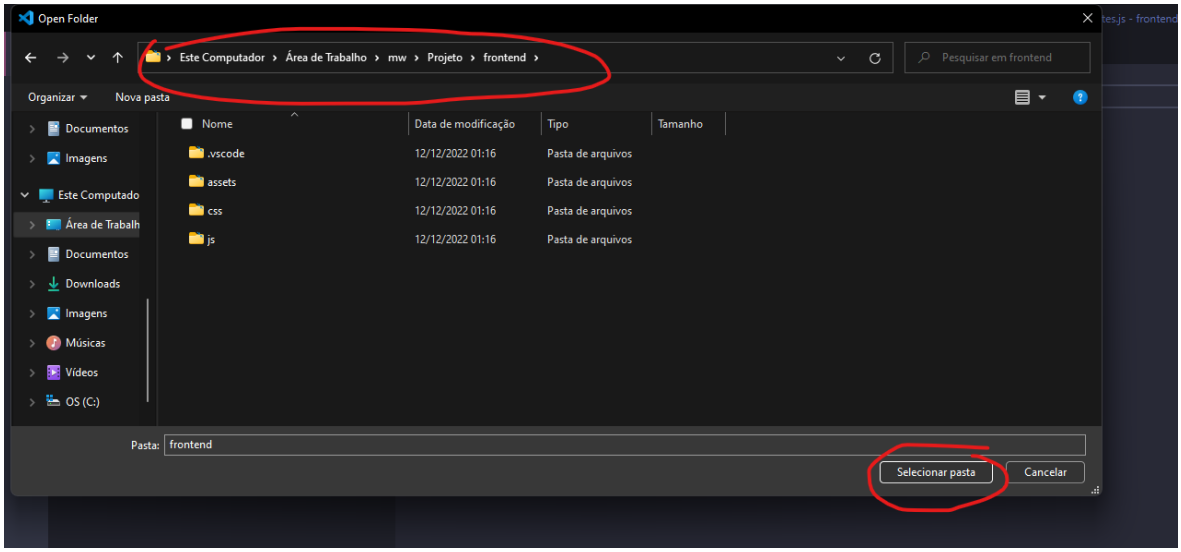
- Abrir o VSCode na nossa pasta frontend
- Iniciar o Live Server

▼ Abrindo a nossa pasta frontend no vscode

Abra o VSCode e vá para “File” ou “Arquivo”



Clique em “Open Folder” ou “Abrir pasta” e selecionar a nossa pasta frontend



▼ Iniciar o Live Server

Basta clicar com o botão direito do mouse sobre qualquer arquivo `.html` e selecionar a opção “Open with Live Server” ou “Abrir com Live Server”