



UNIVERSIDADE DE
VASSOURAS

**LABORATÓRIO DE PROGRAMAÇÃO DE
WEB SITES**

JavaScript

Prof. Msc. Caio Jannuzzi

Sumário

JavaScript

O que é	02
Iniciando	02
Variáveis	03
Tipos de variáveis	06
Operadores	09
Declarações condicionais	13
Function	15
Object	16
Array	17
Métodos de array	19
Loops	22
DOM	27
API com Axios	36
JavaScript Assíncrono: Async/Await	39

Módulo: JavaScript

O que é

JavaScript, ou JS é uma linguagem de programação, sendo a principal linguagem do frontend. Ela é responsável pelo dinamismo de uma página, ou seja, promove a interação com o usuário. Um exemplo clássico de JavaScript são os menus mobile, quando clicamos nele é feito uma interação com o usuário (no caso abrindo os itens do menu).

Iniciando

JavaScript é escrito em arquivos .js. Para criar um documento JavaScript é fácil, basta salvar o arquivo como script.js (lembro que você pode nomeá-lo como quiser, mas não esqueça do .js) dessa forma teremos nosso documento semipronto para o uso.

Agora iremos criar a conexão do JavaScript com nosso arquivo HTML. Antes do fechamento da tag <body> de seu documento HTML, vamos adicionar uma tag <script> para conectá-los. Isso é, na tag <script> atribuímos o atributo src="" (que vem do inglês source), esse atributo é responsável em fazer a conexão com outros arquivos.

Variáveis

Variáveis são lugares na memória onde colocamos valores para podermos trabalhar com eles posteriormente. As variáveis são um dos fatores para mantermos o código dinâmico, fácil de ser lido e compreendido Isso é, uma vez armazenado um valor em uma memória podemos utilizar seu valor ao longo do nosso código.

No JavaScript não há necessidade de declarar o tipo da variável, mas isso não significa que ela não tem tipo. Isso torna o JavaScript uma linguagem de tipagem dinâmica, o tipo é inferido pelo valor do dado e a checagem (type checking) é feita em tempo de execução (runtime). Então se você tiver uma variável chamado “nome” com o valor de “iuri” e ao longo do nosso código mudar seu valor para 10, para o JavaScript isso está certo. Isso tem seus lados bons e ruins.

Por exemplo, essa é a declaração de uma variável:

```
1 let nome = "Caio" // Está correto
2 nome = 10; // Está correto também
3 // Resultado final de nome é 10
```

Observação: o // são comentários. Comentários são linhas de códigos não executáveis.

A barra dupla // serve para comentar uma linha de código.

Começa com /* e termina com */ podemos comentar várias linhas de código em JavaScript.

"Ok! , mas o que é esse const em nosso código?"

Bem, para declarar uma variável temos três opções: var, let e const.

- Uso do var: Uma variável declarada com var possui o que chamamos de escopo de função. Isso significa que se criarmos uma variável deste tipo dentro de uma função, você pode modificar em qualquer parte desta função, mesmo se criar outra função dentro dela.
- Uso do let: Diferente de var, declarar como let leva em conta o bloco de código onde foi declarada. Isso significa que se a declararmos dentro de um uma função, ela será "enxergada" apenas dentro deste escopo.
- Uso do const: Uma variável const é usada para definir uma constante. Diferente de var e let, as variáveis de const não podem ser atualizadas nem declaradas novamente.

Var



```
1 function exibirNome() {  
2     var nome = "Caio" ;  
3 }  
4  
5 console.log(nome); // Erro: nome não está definido
```

Let



```
1 let nome = "Caio" ;  
2  
3 if (true) {  
4     let nome = "Kenay";  
5     console.log(nome); // Retornará "Kenay"  
6 }  
7  
8 console.log(nome); // Retornará "Caio"
```

Por que isso não retorna um erro? Porque as duas instâncias são tratadas como variáveis diferentes, já que são de escopos diferentes.

Const

```
● ● ●  
1 const nome = "Caio" ;  
2 nome = "Kenay";  
3 // Erro: atribuição a uma variável constante.
```

Assim como as declarações de let, os consts sofrem o hoisting para o topo do escopo, mas não são inicializadas.

Tipos de variáveis

String

Uma string é uma sequência de caracteres usados para representar texto. São declaradas usando aspas simples ou aspas duplas.

```
● ● ●  
1 var nome = "Caio" ;  
2 // Ou  
3 var nome = 'Caio' ;
```

Number

Number é um tipo de dado numérico. O JavaScript não diferencia números inteiros, em ponto flutuante, double...



```
1 var numero = 10;  
2 var pi = 3.14;
```

Boolean

Um booleano é um tipo de dado lógico que pode ter apenas um de dois valores possíveis: true (verdadeiro) ou false (falso).



```
1 var javascript = true;  
2 var java = false;
```

Object

Objetos são tipos de dados especiais que podem armazenar vários valores ao mesmo tempo.

```
● ● ●  
1 var pessoa = {  
2   nome: "Caio" ,  
3   idade: 22,  
4   frontend: true,  
5 };
```

Function

Uma função é um conjunto de instruções que executa uma tarefa ou calcula um valor.

```
● ● ●  
1 function nome() {  
2   console.log("Caio");  
3 }
```

Undefined

Uma variável que não foi atribuída a um valor específico, assume o valor `undefined` (indefinido).



```
1 var texto
```

Null

O valor null é um valor nulo ou "vazio" (exemplo: que aponta para um objeto inexistente).



```
1 var texto = null;
```

Operadores

Operadores em JavaScript são símbolos especiais que envolvem um ou mais operandos com a finalidade de produzir um determinado resultado. Não irei citar todos os operadores, somente os mais utilizados.

Operadores de atribuição

"Um operador de atribuição atribui um valor ao operando à sua esquerda baseado no valor do operando à direita. O operador de atribuição básica é o igual (=), que atribui o valor do operando à direita ao operando à esquerda. Isto é, $x = y$ atribui o valor de y a x ." - MDN Web Docs

Operador	Operador encurtador
Atribuição	$x = y$
Atribuição de adição	$x += y$
Atribuição de subtração	$x -= y$
Atribuição de multiplicação	$x *= y$
Atribuição de divisão	$x /= y$
Atribuição de resto	$x %= y$

Operadores aritméticos

"Operadores aritméticos tomam valores numéricos como seus operandos e retornam um único valor numérico. Os operadores aritméticos padrão são os de soma (+), subtração (-), multiplicação (*) e divisão (/). Estes operadores trabalham da mesma forma como na maioria das linguagens de programação quando utilizados com números" - MDN Web Docs

Operador	Descrição
Módulo (%)	Retorna o inteiro restante da divisão dos dois operadores.
Incremento (++)	Adiciona um ao seu operando.
Decremento (--)	Subtrai um de seu operando.
Negação (-)	Retorna a negação de seu operando.
Adição (+)	Tenta converter o operando em um número.

Operadores de comparação

"Um operador de comparação compara seus operandos e retorna um valor lógico baseado em se a comparação é verdadeira. Na maioria dos casos, se dois operandos não são do mesmo tipo, o JavaScript tenta convertê-los para um tipo apropriado. Isto geralmente resulta na realização de uma comparação numérica." - MDN Web Docs

Operador	Descrição
Igual (==)	Retorna true caso os operandos sejam iguais.
Não igual (!=)	Retorna true caso os operandos não sejam iguais.
Estritamente igual (===)	Retorna true caso os operandos sejam iguais e do mesmo tipo.
Estritamente não igual (!==)	Retorna true caso os operandos não sejam iguais e do mesmo tipo.
Maior que (>)	Retorna true caso o operando da esquerda seja maior que o da direita.
Maior que ou igual (>=)	Retorna true caso o operando da esquerda seja maior ou igual que o da direita.
Menor que (<)	Retorna true caso o operando da esquerda seja menor que o da direita.
Menor que ou igual (<=)	Retorna true caso o operando da esquerda seja menor ou igual que o da direita.

Declarações condicionais

As estruturas condicionais são utilizadas para verificar uma condição e definir se algo deve ou não acontecer.

Podemos representar condições em JavaScript utilizando o comando de estrutura condicional if (se) da seguinte maneira:

```
1 if (condicao) {  
2   // O código será executado caso a condição  
3   // seja verdadeira  
4 }
```

Agora você deve estar se perguntando “Mas Iuri, e se essa condição for falsa?”. Nesse nosso exemplo, se a condição for falsa o fluxo irá continuar sem executar nosso código dentro do if.

Declarações condicionais

Agora, caso você queira executar algo quando a condição for falsa, utilizamos o else (se não). O else sempre será executado caso o if seja falso.

```
● ● ●  
1 if (condicao) {  
2 // Será executado caso a condição seja verdadeiro  
3 } else {  
4 // Será executado caso a condição seja falso  
5 }
```

No contexto de estruturas condicionais... vamos imaginar que temos um sistema de verificação de nome, e você queria verificar apenas se dois nomes existem em nosso sistema. Você não concorda comigo que fazer dois if e else para verificar os dois nomes ficaria algo repetitivo? Ainda mais que nosso else iria retornar a mesma coisa nas duas condições.

O else if (se não se) nos ajuda a fazer mais do que uma condição, ou seja se a condição anterior não for verdadeira verifique uma nova condição.

```
● ● ●  
1 if (condicao) {  
2 // Será executado caso a condição seja verdadeiro  
3 } else if (condicaoDois) {  
4 // Será executado caso a primeira condição seja falsa  
5 } else {  
6 // Será executado caso todas as condições sejam falsas  
7 }
```

Function

Uma função, ou function é um conjunto de instruções que executa uma tarefa.

Imagine que você tenha duas tarefas, uma é executar a soma de dois números e a outra é mostrar o nome do usuário no sistema. Percebeu que essas tarefas são duas funções diferentes? Imagine essas duas tarefas espalhadas em nosso código, iria atrapalhar muito, mas com uma função teremos um agrupamento de código que faz uma determinada tarefa em nossa aplicação.

A definição da função (também chamada de declaração de função) consiste no uso da palavra-chave `function`, seguida por:

- Nome da função.
- Lista de argumentos para a função, entre parênteses e separados por vírgulas.
- Declarações JavaScript que definem a função, entre chaves `{ }`.

Por exemplo, o código a seguir define uma função simples chamada `somar`:



```
1 function somar(numero) {  
2   return numero * numero;  
3 }  
4  
5 console.log(somar(10));
```

Veja que a função de somar recebe um argumento chamado numero. Isso significa que a nossa função está esperando o valor de numero (que no exemplo é 10) para multiplicar por si mesmo (numero * numero). A declaração return especifica o valor retornado pela função.

Object

Imagine que você tenha uma variável chamada pessoa e queira ter uma coleção de propriedades, como nome, idade, cidade... Isso é possível? Claro que é, isso é chamado de objeto. Objeto é uma coleção de propriedades, sendo cada propriedade definida como uma sintaxe de chave: valor. A chave pode ser uma string e o valor pode ser qualquer informação.

```
1 const pessoa = {  
2   nome: "Iuri",  
3   idade: 22,  
4 };
```

Para acessar uma propriedade de um objeto, use uma das duas notações (syntaxes):

- Notação de ponto (objeto.propriedade).
- Notação de array (objeto["propriedade"]).

```
1 pessoa.nome; // Resultado: Caio  
2 pessoa["nome"]; // Resultado: Caio
```

Array

Arrays, ou matrizes são lista de objetos. Elas são objetos que contêm múltiplos valores armazenados em uma lista.

Se nós não tivéssemos arrays, teríamos que armazenar cada item em uma variável separada. Isso é, uma variável chamada, frutas1, outras chamadas fruta2, fruta3, fruta4.... imagine se fosse 100 frutas. Isto seria muito mais longo de escrever e acessar.

Arrays são construídas de colchetes, os quais contém uma lista de itens separada por vírgulas.

Vamos supor que queremos armazenar a nossa lista de frutas, nós temos o seguinte código.

```
● ● ●  
1 var frutas = ["banana", "maçã", "uva", "kiwi"];
```

Você pode acessar itens individuais em um array usando a notação de colchetes, da mesma forma que você acessa as letras em uma string.

```
● ● ●  
1 frutas[2];
```

Observação: o índice do array sempre começa na posição 0.

Métodos de array

Arrays nos fornece vários métodos para facilitar nosso trabalho em certas situações. Com cada método que vamos abordar não estão disponíveis em objetos ou qualquer outra coisa além de Arrays.

Veremos os seguintes métodos de Array (é bom lembrar que existem outros métodos de array):

- concat()
- join()
- push()
- pop()
- shift()
- slice()
- splice()
- reverse()

Concat

O método “concat()”, que serve para concatenar dois arrays, no exemplo vamos concatenar o array front com o array back.

```
1 var front = ["html", "css", "javascript"];
2 var back = ["java", "php", "node"];
3
4 front = front.concat(back);
5 console.log(front);
6 // Resultado será um array com os elementos dois arrays
```

Join

O método “join()” puxa elementos de um array e lista no formato de string.

```
1 var front = ["html", "css", "javascript"];
2 front = front.join("-");
3
4 console.log(front);
5 // Resultado será as propriedades do array
6 // separar com um traço
```

Push

O método “push()” serve para adicionarmos elementos no final do array.

```
1 var front = ["html", "css", "javascript"];
2 front.push("react");
3
4 console.log(front);
5 // Resultado será um novo elemento no final do array
```

Pop

O método “pop()” remove o último elemento de um array.

```
1 var front = ["html", "css", "javascript"];
2 front.pop();
3
4 console.log(front);
5 // Resultado será a remoção do último elemento do array
```

Shift

O método “shift()” remove o primeiro elemento do array.



```
1 var front = ["html", "css", "javascript"];
2 front.shift();
3
4 console.log(front);
5 // Resultado será a remoção do primeiro elemento do array
```

Reverse

O método “reverse()” inverte a ordem dos elementos do array.



```
1 var front = ["html", "css", "javascript"];
2 front.reverse();
3
4 console.log(front);
5 // Resultado será reversão dos elementos do array
```

Loops

Laços de repetição, ou loops como o próprio nome já diz, ele faz com que blocos de códigos sejam repetidos diversas vezes até que certa condição seja cumprida.

Um loop geralmente possui um ou mais dos seguintes itens:

- O contador, que é inicializado com um certo valor - este é o ponto inicial do loop.
- A condição de saída, que é o critério no qual o loop para - geralmente o contador atinge um certo valor.
- A expressão, que geralmente incrementa o contador em uma pequena quantidade a cada loop, sucessivamente, até atingir a condição de saída.

For

O primeiro que você usará na maior parte do tempo, é o loop for, ele tem a seguinte sintaxe:

```
● ● ●  
1 for (inicializador; condição; expressão) {  
2   // Código que será executado  
3 }
```

No exemplo do For temos:

- A palavra-chave for, seguido por parênteses.
- Dentro do parênteses temos três itens, separados por ponto e vírgula:

- O inicializador— geralmente é uma variável configurada para um número, que é incrementado para contar o número de vezes que o loop foi executado. É também por vezes referido como uma variável de contador.
- A condição — como mencionado anteriormente, aqui é definido quando o loop deve parar de executar. Geralmente, essa é uma expressão que apresenta um operador de comparação, um teste para verificar se a condição de saída foi atendida.
- A expressão — isso sempre é avaliado (ou executado) cada vez que o loop passou por uma iteração completa. Geralmente serve para incrementar (ou, em alguns casos, decrementar) a variável do contador, aproximando-a do valor da condição de saída.

For na prática

```
● ● ●  
1 for (let i = 1; i ≤ 5; i++) {  
2   console.log(i); // Saída: 1, 2, 3, 4, 5  
3 }
```

Explicando: dentro do nosso laço for temos o “let i=1;” ele é o nosso inicializador, ele basicamente está falando que nossa variável de inicialização é o “i” que tem o valor de 1.

Logo em seguida temos o “i<=5”, ele é o condição, como o próprio nome já diz, ele é a condição que irá executar em nosso loop até que seja cumprida. Como o valor inicial de i é 1 ele será executado até seu valor ser igual a 5.

E por último temos o “i++”, ele é o iterador, ou seja, é o responsável pelo incremento da nossa variável i até chegar em 5.

While

“While funciona de maneira muito semelhante ao loop for, exceto que a variável inicializadora é definida antes do loop, e a expressão final é incluída dentro do loop após o código a ser executado - em vez de esses dois itens serem incluídos dentro dos parênteses. A condição de saída está incluída dentro dos parênteses, que são precedidos pela palavra-chave while e não por for.” - MDN Web Docs

```
● ● ●  
1 inicializador;  
2 while (condição) {  
3 // Código  
4 expressão;  
5 }
```

Do while

```
● ● ●  
1 inicializador;  
2 do {  
3 // Código  
4 expressão;  
5 } while (condição);
```

Nesse caso, a condição é avaliada depois que o bloco de código é executado, ou seja, o código dentro do "do" será executado pelo menos uma vez antes de ter certeza que a condição é verdadeira.

DOM

"O Document Object Model (DOM) é uma interface de programação para os documentos HTML e XML. Representa a página de forma que os programas possam alterar a estrutura do documento, alterar o estilo e conteúdo. O DOM representa o documento com nós e objetos, dessa forma, as linguagens de programação podem se conectar à página." - MDN Web Docs

O que são eventos no JavaScript?

Os eventos são ações que são realizadas em um determinado elemento da página, seja ele um texto ou uma imagem, por exemplo. Muitas das interações do usuário que está visitando sua página com o conteúdo do seu site podem ser consideradas eventos.

Principais eventos:

- onload: Disparado quando documento é carregado.
- onunload: Disparado quando documento é descarregado de janela ou de frame.
- onsubmit: Disparado quando formulário é submetido.
- onreset: Disparado quando formulário é "limpado" via botão de reset.
- onselect: Disparado quando texto é selecionado numa área de entrada de texto.
- onchange: Disparado quando elemento perde o foco e foi modificado.
- onclick: Disparado quando botão de formulário é selecionado via click do mouse.
- onfocus: Disparado quando o elemento recebe foco: clicando o mouse dentro do elemento ou entrando no mesmo via Tab.
- ondblclick: Disparado quando ocorre um click duplo do mouse.
- onmousedown: Disparado quando mouse é pressionado enquanto está sobre um elemento.
- onmouseup: Disparado quando mouse é despressionado.
- onmouseover: Disparado quando cursor do mouse é movido sobre elemento
- onmouseout: Disparado quando mouse é movido fora do elemento onde estava
- onkeydown: Disparado quando tecla é pressionada.

Existem diversas maneiras de se aplicar esses eventos aos elementos HTML, são elas:

- Inline.
- Em um arquivo externo, usando um manipulador de eventos.

Qual é melhor? Bem, usar a maneira de arquivo externo deixa nosso documento HTML limpo de código JavaScript. Já com inline, aplicamos diretamente na tag HTML o evento JavaScript.

```
1 <button onclick="alert('Oi')">
2   Sou um evento inline
3 </button>
```

O exemplo anterior está funcionando corretamente, aos clicarmos no botão irá disparar um evento chamado “onclick” que irá abrir uma simples caixas de diálogo (alert) porém, estamos adicionando JavaScript junto com código HTML. Irei reproduzir o mesmo evento em um arquivo JavaScript.



```
1 document.querySelector("button").addEventListener(  
2 "click", () => {  
3   alert("oi");  
4 });
```

Sei que olhando esse código parece que utilizar eventos inline é mais fácil e rápido, mas estamos utilizando só um exemplo, imagina uma aplicação cheia de animações e interações do usuário, você mesmo ficaria perdido e terá um código JavaScript maior.

Tenha calma que irei explicar cada parte desse código futuramente.

Selecionando elementos HTML

Seletores são métodos que selecionam tags HTML (e seu conteúdo), seja pelo id, class ou pela própria tag. Isso é muito útil quando queremos alterar a DOM, seja adicionando novos elementos ou alterando o que já existe nela. Veremos os seguintes métodos:

- getElementById()
- getElementsByClassName()
- querySelector()
- querySelectorAll()

Observação: todas as propriedades, métodos e eventos disponíveis para manipular e criar páginas são organizados em objetos, por exemplo, o objeto document representa o próprio documento.

getElementById

Este método retorna um elemento correspondente ao id passado como parâmetro. Veja o exemplo abaixo.



```
1 let eSW = document.getElementById("ebook");
```

Este método retorna um elemento correspondente ao id passado como parâmetro. Veja o exemplo abaixo.

getElementsByClassName

Como o próprio nome já diz, essa função vai retornar os elementos que possuem uma mesma classe passada.



```
1 let eSW = document.getElementsByClassName("ebook");
```

Enquanto a função anterior retorna um único elemento, esta retorna uma `HTMLCollection` (uma coleção) de elementos.

querySelector

Diferente dos outros métodos, este método utiliza “.” para indicar a seleção de uma classes, “#” para indicar seleção de ids ou a própria tag.

```
1 let teste1 = document.querySelector("#souId");
2 let teste2 = document.querySelector(".souClass");
3 let teste3 = document.querySelector("div");
```

Um ponto interessante do `querySelector()` só retorna o primeiro elemento encontrado que tem a mesma seleção.

querySelectorAll

Este método é similar ao método `querySelector` que também utiliza os seletores do CSS, porém retorna uma `NodeList` com todos os elementos que correspondem ao seletor criado.



```
1 let teste = document.querySelectorAll("div");
2 // Irá retornar todas as divs da página
```

Interações do usuário

Existem duas formas comuns (existem outras formas) de adicionar estilos CSS em nossa página com JavaScript:

- Estilos Inline.
- Classes CSS a elementos.

A propriedade style em JavaScript retorna o estilo de um elemento na forma de um objeto CSSStyleDeclaration que contém uma lista de todas as propriedades de estilos. Dessa forma conseguimos adicionar estilos CSS em nossa aplicação através do JavaScript.



```
1 var button = document.querySelector("button");
2
3 button.addEventListener("click", () => {
4   button.style.backgroundColor = "red";
5 });
```

Perceba que diferente das propriedades do CSS, no JavaScript não temos o “-” (hífen) e substituímos ele pela convenção de nomenclatura lowerCamelCase.

Dessa forma adicionamos background-color do CSS em nosso botão de forma de estilos inline. Mas veja que isso leva a um problema que tivemos com o JavaScript dentro do HTML.

Adicionando classes CSS nos elementos

Utilizando a propriedade classList você pode obter, definir ou remover classes CSS facilmente de um elemento. O exemplo a seguir mostrará como adicionar uma classe chamada “outraCor” em um elemento <div> com id=’ “UNIVASS”’.

```
1 var elementoDiv = document.getElementById ("UNIVASS");
2
3 elementoDiv.addEventListener("click", () => {
4   elementoDiv.classList.toggle("outraCor");
5 });
```

Uma observação: eu criei uma classe no CSS que tem a propriedade background-color, assim quando ela for chamada através do click, irá mudar a cor do elemento selecionado (que no caso é a div com o id="iuricode").

addEventListener

O método addEventListener (ou conhecido como evento de escuta) permite configurar funções a serem chamadas quando um evento acontece, que em nosso exemplo, quando um usuário clica em um botão. No exemplo a seguir mostrar que o elemento nomeElemento está sendo escutado caso o usuário faça um click.

Sintaxe do addEventListener:



```
1 alvo.addEventListener(evento, função);
```

- alvo: é o elemento HTML ao qual você deseja adicionar os eventos.
- evento: é onde você especifica qual é o tipo de evento que irá disparar a ação/função. Uma coisa interessante no evento é que diferente do JavaScript no HTML, ele não tem o “on” dos eventos como onclick, onchange, onblur...

- função: especifica a função a ser executada quando o evento é detectado. Em nossos exemplos foram utilizados a sintaxe arrow function que deixa a expressão da nossa função mais curta e anônima. Mas você pode simplesmente aplicar uma outra função código no lugar.

```
1 var nomeElemento = document.getElementById("div");
2
3 nomeElemento.addEventListener("click", () => {
4   ...
5});
```

O arrow function não terá uma explicação mais afundo pois ele é um recursos introduzido no ES6 e estamos apenas abordando os fundamentos. Mas caso queria saber sobre ele clique [aqui](#).

API com Axios

API, ou Application Programming Interface, é um conjunto de definições e protocolos para integrar softwares de aplicações. Um exemplo clássico de API é a do Google Maps. Por meio de seu código, conseguimos ter acesso a localização, muitas outras aplicações utilizam os dados do Google Maps adaptando-o da melhor forma.

O que é Axios?

Axios é uma biblioteca JavaScript baseada em promessas (promises) para interceptar requisições (requests) HTTP. Esses interceptadores são úteis quando queremos pegar ou alterar requisições HTTP. Cada requisição HTTP pode usar um dos métodos de requisição existente, temos sete tipos de requisição, mas nesse ebook iremos abordar somente o método GET para consultar os dados em uma API.

A primeira coisa a se fazer quando estamos trabalhando com uma biblioteca ou framework, é procurar o CDN da biblioteca para importá-la em nossa aplicação.

```
1 <script
2   src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js">
3 </script>
4 <script src="main.js"></script>
```

Dessa forma temos acesso aos recursos do Axios.

Uma observação importante: a importação do CDN tem que vir antes do nosso arquivo script que iremos utilizar para consumir a API (isso se aplica a qualquer biblioteca).

Adicionando a API

Em nosso exemplo iremos utilizar a API de usuário do GitHub.

```
1 axios.get("https://api.github.com/users/ xxxx");
```

Primeiro, você importa o Axios para que possa ser usado na aplicação. Em seguida, executa uma requisição utilizando o GET através do método get() para obter uma promessa que retorna um objeto de resposta que pode dar sucesso ou erro, dependendo da resposta da API.

```
1 axios
2 .get("https://api.github.com/users/ xxxx");
3 .then(function (response) {
4   console.log(response.data);
5   console.log(response.data.login);
6 })
7 .catch(function (error) {
8   console.log(error);
9 });
```

"Ok | ... mas o que é esse then?"

Lembra que eu falei que o Axios trabalha com promessas? Exatamente isso que está acontecendo! Caso a promessa for cumprida, o argumento `then()` será chamado; se a promessa for rejeitada, o argumento `catch()` será chamado.

Dentro do argumento `then()` estamos imprimindo duas coisas no console do navegador: `response.data` e `response.data.login`. O `response.data` está imprimindo todos os dados da nossa API no console, já o `response.data.login` está imprimindo o nosso username do GitHub.

Mas Axios não é a única solução. Temos nativamente no JavaScript o `fetch()`. Ele é perfeitamente capaz de reproduzir as principais funcionalidades do Axios! Mas o Axios é uma biblioteca que traz bastante benefícios quando estamos trabalhando com APIs.

JavaScript Assíncrono: Async/Await

Agora que você já sabe trabalhar com Promises no Javascript, iremos aprender sobre `async/await`.

O `async/await` é uma nova forma de tratar Promises dentro do nosso código, evitando a criação de cascatas de `.then` como vimos no exemplo passado.

Primeiramente gostaria de explicar como o JavaScript funciona por debaixo dos panos, tornando mais fácil o entendimento do assíncrono.

O comportamento do JavaScript de "executar uma coisa por vez". Com o JavaScript assíncrono, o comportamento vai separar seu código em duas partes: coisas que rodam AGORA, coisas que vão rodar DEPOIS de algo acontecer. Calma que já vai ficar mais fácil de entender.

Para exemplificar, podemos pensar em comunicação:

Uma ligação/chamada é um exemplo de comunicação síncrona: quando falamos ao telefone, as informações chegam e saem em sequência, uma após a outra.

Por outro lado, uma conversa online via a mensagem, como o WhatsApp, é um exemplo de comunicação assíncrona: quando enviamos uma mensagem e as informações não chegam logo em sequência, a gente espera até a outra pessoa responder.

Sintaxe

Utilizamos o prefixo `async` antes de definir uma função para indicar que estamos tratando de um código assíncrono, e com o prefixo adicionado, podemos utilizar o `await` antes das Promises indicando um ponto a ser aguardado pelo código. Já irei explicar melhor sobre `Async` e `Await`.

```
● ● ●  
1 // Sem o Async/Await  
2 function fetchUser(user) {  
3   api.get().then(response => {  
4     console.log(response); });  
5 }  
6  
7 // Com o Async/Await  
8 async function fetchUser(user) {  
9   const response = await api.get();  
10  console.log(response);  
11 }
```

Veja como o código ficou mais limpo, não precisamos mais declarar os `.then` ou `.catch` e ter medo de alguma Promise não executar antes de utilizarmos seu resultado pois o `await` faz todo papel de aguardar com que a requisição retorne seu resultado.

- Async. Essa palavra pode ser usada ao criar uma função. Quando adicionamos esse identificador na criação da função, nós definimos que ela será uma função assíncrona, e o melhor, retornará uma promessa. Quando usarmos a expressão return estaremos, na realidade, resolvendo uma promessa.
- Await. Essa palavra será usada com o objetivo de esperar a resolução de uma função assíncrona. Se houver uma série de funções assíncronas, a expressão await definirá que o código só terá sequência quando a função anterior for resolvida. Um detalhe muito importante: a expressão await só será aceita em uma função que já for assíncrona, ou seja, que possuir o identificador async.

Vamos substituir o then?

O `async/await` surgiu como uma opção mais "clean" ao `.then()`, mas é possível usar os dois métodos em um mesmo código. O `async/await` simplifica a escrita e a interpretação do código, mas não é possível comparar os dois, pois possuem funcionamentos lógicos diferentes. Ao usarmos o `then`, as Promises são executadas em paralelo, enquanto o `async/await` trata as Promises de forma sequencial, como se estivesse realmente executando um código síncrono, que imprime os resultados um após o outro.

Confuso ainda? Vamos esclarecer agora!

Um exemplo: uma inserção de dados no banco pode demorar um pouco, então para receber uma resposta positiva antes de tomar outra ação, condicionamos a inserção em uma função assíncrona.

Assim é possível esperar todo o processo do banco de dados finalizar para darmos andamento ao nosso sistema.

```
● ● ●  
1 function primeiraFuncao() {  
2   console.log("Segundo");  
3 }  
4  
5 async function segundaFuncao() {  
6   await primeiraFuncao()  
7  
8   console.log("Primeiro");  
9 }  
10  
11 segundaFuncao()
```

Imagine que dentro da função “primeiraFunção” tenha as informações do bando de dados, é que o console “Primeiro” dependa das informações da função “primeiraFuncao” para realizar a ação. Para que a função “segundaFuncao” seja executada com as informações necessária, precisamos transforma-la em uma função assíncrona.

Sei que esse conceito é um pouco complexo de entender, mas com muitas práticas você irá entender melhor o funcionamento de funções assíncronas no JavaScript.