JavaScript Avançado

Carlos Dias da Silva Júnior

Formação Inicial e Continuada

IFMG

Carlos Dias da Silva Júnior

JavaScript Avançado

1ª Edição

Belo Horizonte

Instituto Federal de Minas Gerais

2022

Pró-reitor de Extensão	Carlos Bernardes Rosa Júnior
Diretor de Projetos de Extensão	Niltom Vieira Junior
Coordenação do curso	Carlos Dias da Silva Júnior
Arte gráfica	Ângela Bacon
Diagramação	Eduardo dos Santos Oliveira

FICHA CATALOGRÁFICA
Dados Internacionais de Catalogação na Publicação (CIP)

S586j Silva Júnior, Carlos Dias da.

JavaScript: avançado. / Carlos Dias da Silva Júnior. – Belo Horizonte: Instituto Federal de Minas Gerais, 2022.

49 p.: il. Color.

E-book, no formato PDF.
Formação Inicial e Continuada.
ISBN 978-65-5876-032-0

1. Arrays. 2. Web API. 3. Browser.4. JavaScript. I. Título.

CDD 005

Catalogação: Rejane Valéria Santos- CRB-6/2907

Índice para catálogo sistemático:
1. Programação - 005

2022
Direitos exclusivos cedidos ao
Instituto Federal de Minas Gerais
Avenida Mário Werneck, 2590,
CEP: 30575-180, Buritis, Belo Horizonte – MG,
Telefone: (31) 2513-5157

Sobre o material

Este curso é autoexplicativo e não possui tutoria. O material didático, incluindo suas videoaulas, foi projetado para que você consiga evoluir de forma autônoma e suficiente.

Caso opte por imprimir este *e-book*, você não perderá a possiblidade de acessar os materiais multimídia e complementares. Os *links* podem ser acessados usando o seu celular, por meio do glossário de Códigos QR disponível no fim deste livro.

Embora o material passe por revisão, somos gratos em receber suas sugestões para possíveis correções (erros ortográficos, conceituais, *links* inativos etc.). A sua participação é muito importante para a nossa constante melhoria. Acesse, a qualquer momento, o Formulário “Sugestões para Correção do Material Didático” clicando nesse *link* ou acessando o QR *Code* a seguir:

Para saber mais sobre a Plataforma +IFMG acesse

mais.ifmg.edu.br

Palavra do autor

A linguagem JavaScript é uma das mais utilizadas atualmente sendo, entre as linguagens disponíveis, a mais cogitada para o desenvolvimento web. Além disso, ela se faz presente em diversas aplicações industriais, no mercado de trabalho e até mesmo no ambiente acadêmico, sendo, portanto extremamente proveitosa sua compreensão para aperfeiçoamento dos conhecimentos de programação e dinamicidade dos códigos e interações cliente-servidor.

Sendo assim, a partir deste conteúdo buscou-se desenvolver os conhecimentos de JavaScript avançado com uma linguagem simples e dinâmica, com exemplos direcionados e de fácil entendimento. Na primeira parte da apostila procurou-se apresentar uma revisão robusta a respeito dos conteúdos básicos de JavaScript, para uma melhor compreensão das ferramentas apresentadas mais à frente.

No mais a proposta da segunda parte da apostila, é estabelecer o contato do aluno com as ferramentas avançadas de JavaScript a partir de conteúdos voltados para o desenvolvimento de aplicações, introdução de funções assíncronas que permitem modificações de uma página Web, e uso de funções, e manipulação do visual.

Ademais, com o conteúdo da apostila pretende-se aprimorar o contato do aluno com a linguagem JavaScript, possibilitando uma compreensão geral de programação e desenvolvimento de sistemas.

Desejo que vocês tenham sucesso em sua jornada, não somente no curso de JavaScript, mas como futuros desenvolvedores!

O autor.

Apresentação do curso

Este curso está dividido em quatro semanas, cujos objetivos de cada uma são apresentados, sucintamente, a seguir.

SEMANA 1	Tem como objetivo introduzir os estudos ao JavaScript avançado, revisando conteúdos que serão úteis ao longo do curso.
SEMANA 2	Nesta semana entenderemos como é estruturado o DOM, e estudar como modificar elementos visuais em uma página da web utilizando JavaScript.
SEMANA 3	Para agilizar o desenvolvimento de aplicações, é necessário conhecer várias ferramentas. Nesta semana aprenderemos sobre algumas ferramentas muito úteis e como obter dados de aplicações externas.
SEMANA 4	Com foco em muitas ações simultâneas, nesta semana final entenderemos como utilizar funções assíncronas de forma a evitar que a interface esteja travada para o usuário, enquanto várias ações são feitas pela aplicação.

Carga horária: 40 horas.

Estudo proposto: 2 horas por dia, durante 5 dias da semana

Apresentação dos Ícones

Os ícones são elementos gráficos para facilitar os estudos, fique atento quando eles aparecem no texto. Veja aqui o seu significado:

Atenção: indica pontos de maior importância no texto.

Dica do professor: novas informações ou curiosidades relacionadas ao tema em estudo.

Atividade: sugestão de tarefas e atividades para o desenvolvimento da aprendizagem.

Mídia digital: sugestão de recursos audiovisuais para enriquecer a aprendizagem.

Sumário

Objetivos
Tem como objetivo introduzir os estudos ao JavaScript avançado, revisando conteúdos que serão úteis ao longo do curso.

Mídia digital: Para ficar mais claro cada passo aqui nesta semana, veja a videoaula “Revisando JS”.

1.1 Arrays

Como nós sabemos, para uma representação eficaz e simples de um conjunto de dados como, por exemplo, listas de compras ou listas de carros, podemos utilizar o objeto ou variável especial do tipo array, que permite o armazenamento e estruturação dos dados em um único nome.

Além disso, o uso do array permite, também, a consulta desses valores a partir de um número de índice, sendo possível percorrer a lista, fazendo consultas específicas, como, por exemplo, caso tivéssemos uma lista de chamada do terceiro ano de uma escola, poderíamos, como visto abaixo, criar um array e atribuir valores a ele: os nomes dos alunos.

```
const name = ["Ana", "Alicia","Betina"];
```

Dessa maneira, nós conseguimos a partir do uso do array, percorrer os nomes presentes na lista para encontrar um aluno específico. Lembrando que as quebras de linhas e os espaços não possuem importância nessa aplicação e, assim, podemos realizar o exemplo acima também da maneira abaixo:

```
const name = [
    "Ana",
    "Alicia",
    "Betina"
];
```

Vamos agora conhecer a função Join(), que é uma das muitas funções utilizadas em JavaScript para manipulação de arrays. A Join() é usada convencionalmente como um método para junção de elementos de um dado array em uma string, retornando esta mesma string, a partir do uso de um separador.

Usamos este separador para especificar a string e separar cada elemento no array, sendo a vírgula (,) o separador padrão utilizado. Porém se o separador for uma string vazia, todos os elementos são mantidos sem nenhum caractere entre eles.

Observe o exemplo abaixo:

```
const name = ["Ana", "Alicia", "Betina"];
console.log(name.join(" * "));
```

Este irá produzir a saída:

```
Ana * Alicia * Betina
```

Observe que o que separa cada elemento da string criada é o separador especificado “*”, ao chamar o método join().

Quando utilizamos o Join() para a conversão de arrays em string todos os elementos de um dado array são unidos em apenas uma string, porém se este elemento for null ou undefined uma string vazia é retornada.

Veja agora um exemplo de diversas possibilidades do uso de elementos de junção de variáveis, a partir da aplicação de um array.join(). No exemplo interativo abaixo, nós podemos observar a criação de um array, com 3 nomes, e a junção desses 3 nomes por 2 vezes.

```
const name = ["Ana", "Alicia", "Betina"];

console.log(name.join());
// Saída no console: “Ana,Alicia,Betina”

console.log(name.join(''));
// Saída no console: "AnaAliciaBetina"
```

Na primeira junção foi utilizado a vírgula como elemento separador, sendo apresentado “Ana,Alicia,Betina”, porém na segunda junção não foi utilizado elemento separador sendo portanto apresentado como "AnaAliciaBetina".

Temos também o método pop() que é utilizado quando se quer remover o último elemento de uma matriz, retornando aquele mesmo elemento. A função Pop() é normalmente dada como genérica, o que significa que seu uso é amplo em objetos que complementam arrays. Vejamos melhor sua aplicação a partir do exemplo abaixo.

```
let nome = ["Ana", "Alicia", "Betina"];
console.log(nome); // ["Ana", "Alicia", "Betina"];

let UltimoNome = nome.pop();

console.log(nome); // ["Ana", "Alicia"];
console.log(UltimoNome); // “Betina”
```

Observe no exemplo acima que, quando aplicamos a função pop(), removemos o último dos 3 elementos apresentados, retornando portanto “Betina”.

Já o método dos arrays push(), é semelhante à função que aprendemos anteriormente, já que, assim como a função pop() esta função é genérica e pode ser utilizada em objetos que implementam arrays. Porém, diferentemente da função pop() que remove o último elemento ao final de um array e o retorna, a função push() é usada quando se quer realizar a adição de elementos ao final de um array e retornar este mesmo elemento.

É importante lembrarmos que a propriedade “length” é responsável por determinar o “comprimento”, ou seja, quantos elementos há no array. Sendo assim, um array vazio possui length igual a 0. À medida que adicionamos elementos, esse comprimento é aumentado e remover elementos o reduz. Ou seja, existe uma relação de dependência entre a função push() e a propriedade “length”.

Veja o exemplo abaixo em que criamos um array, adicionamos 2 elementos, e criamos uma variável total com o novo comprimento do array.

```
let nome = ["Ana", "Alicia", "Betina"];
let total = nome.push("Pablo",
"Gabriel");

console.log(nome); // ["Ana", "Alicia", "Betina","Pablo", "Gabriel"]
```

Outro método de arrays é o responsável pela remoção e retorno do primeiro elemento de um array. Este método é o shift() e, assim como os elementos push, pop, e join, este também é genérico, portanto, podemos utilizar o método shift em objetos que possuam semelhança com o array.

```
const nomes = ["Ana", "Alicia", "Betina"];
console.log("nomes antes:" + nomes);
// nomes  antes: ["Ana", "Alicia", "Betina"]
const shifted = nomes.shift();

console.log("nomes depois: " + nomes);
// nomes depois: ["Alicia", "Betina"]
console.log("Elemento removido: " +
shifted);
```

No código apresentado anteriormente temos a lista de nomes antes e depois de removermos seu primeiro elemento, além de retornar este primeiro elemento. Existem ainda diversas maneiras de utilizarmos shift(), como no exemplo a seguir em que, dentro de um laço de repetição while, removemos sucessivamente os elementos do array até que este esteja vazio.

```
const name = ["Ana", "Alicia", "Betina"];
while( (i = name.shift()) !== undefined ) {
   console.log(i);
}
// Ana Alicia Betina
```

Seguimos agora para o método splice(): este tem o objetivo de alterar os conteúdos presentes na lista, inserindo elementos novos ao mesmo tempo que remove os elementos antigos. Para utilizarmos essa função devemos respeitar algumas condições em relação ao índice a qual deve iniciar a alterar a lista, como por exemplo: se o índice for negativo será iniciado a partir daquele número de elementos a partir do fim. Além disso, caso o índice seja maior que o tamanho total da função, nenhum elemento será alterado.

É importante também que se tenha um inteiro assinalando o número de elementos que serão removidos, e para isso é importante especificar o parâmetro deleteCount, já que existem também condições em relação a ele, pois caso, maior que o número de elementos restantes na lista iniciando pelo índice, todos os elementos serão deletados desde o início até o final da lista, além disso, caso o deleteCount seja 0 nenhum elemento é removido. Ademais, quanto aos elementos que serão adicionados a lista, caso não se especifique o elemento a função Splice() removerá elementos da lista. Quanto ao retorno do elemento

removido da lista, é importante nos atentarmos a algumas condições, como, caso nenhum elemento seja removido é retornada uma lista vazia, porém, se removermos apenas um elemento, irá retornar uma lista contendo apenas um elemento.

Como já entendemos as condições associadas ao uso da função splice(), vejamos a aplicação a partir do exemplo abaixo.

```
let name = ["Ana", "Alicia", "Betina", "Carlos"];
//remove 0 elementos a partir do índice 2, e insere "Tiago"
let removed = name.splice(2, 0, "Tiago");
//name é ["Ana", "Alicia", "Tiago" , "Betina", "Carlos"]
//removed é [], nenhum elemento removido

name = ["Ana", "Alicia", "Betina", "Carlos"];
//remove 1 elemento do índice 3
removed = name.splice(3, 1);
//name é ["Ana", "Alicia" , "Tiago", "Carlos"]
//removed é ["Betina"]

name = ["Ana", "Alicia", "Betina", "Carlos"];
//remove 1 elemento a partir do índice 2, e insere "Luna"
removed = name.splice(2, 1, "Luna");
//name é ["Ana", "Alicia", "Luna" , "Carlos"]
//removed é ["Tiago"]

name = ["Ana", "Alicia", "Betina", "Carlos"];
removed = name.splice(0, 2, "Leia", "Obi", "Luck");
//name é ['Leia', 'Obi', 'Luck', 'Betina', 'Carlos']
//removed é ['Ana', 'Alicia']

//remove 2 elementos a partir do indice 3
name = ["Ana", "Alicia", "Betina",
"Carlos"]; removed = name.splice(3,
Number.MAX_VALUE);
//name é ["Leia", "Obi", "Luck"]
```

Dica do Professor: Há mais detalhes sobre como funciona todos os métodos de arrays neste link: https://www.w3schools.com/js/js_arrays.asp.

1.2 Objetos

Para darmos continuidade ao conteúdo do curso iremos agora entender o conceito de objetos em JavaScript, e seu uso. O objeto pode ser visto como um acervo de propriedades, e estas propriedades são definidas como uma sintaxe de par chave : valor, onde a chave pode ser uma string enquanto que o valor pode ser uma informação qualquer. Para criarmos um objeto é preciso utilizar a sintaxe literal do objeto, como podemos observar no exemplo a seguir.

```
let pessoa = {
   "primeiroNome": "Carlos",
   "ultimoNome":  "Dias"
}
```

Neste código, podemos observar o número de propriedades e seus respectivos valores: o número de propriedades foi 2, a primeira assinalada pela chave “primeiroNome” com valor de string “Carlos”, e a segunda assinalada pela chave “ultimoNome” com valor de string “Dias”. Considerando como identificamos as propriedades e valores em um código, iremos acessar a propriedade de um objeto, para isso, utiliza-se duas sintaxes, “Notação de array” e “Notação de Ponto”. Na notação do tipo array, acessamos as propriedades do objeto a partir de colchetes [ ], porém caso se tenha espaço no nome atribuído utiliza-se aspas “ “, como podemos observar no exemplo a seguir:

```
let pessoa = {
   "primeiroNome": "Carlos",
   "ultimoNome":  "Dias"
}
console.log(pessoa["primeiroNome"]);
console.log(pessoa["ultimoNome"]);
```

Outra maneira, mais usual quando se trata de manipulação de objetos em JavaScript, é utilizando ponto. Veja o exemplo a seguir:

```
let pessoa = {
   "primeiroNome": "Carlos",
   "ultimoNome":  "Dias"
}
console.log(pessoa.primeiroNome);
console.log(pessoa.ultimoNome);
```

Considerando que conseguimos acessar as propriedades do objeto, podemos também realizar a operação de adicionar, excluir ou modificar uma propriedade. No exemplo abaixo podemos observar a adição da propriedade idade ao objeto pessoa atribuindo o valor de 27 a ela.

```
let pessoa = {
   "primeiroNome": "Carlos",
   "ultimoNome":  "Dias"
}

pessoa.idade= 27;
```

Observe que a chave idade não existia anteriormente, e por isso ela é criada e adicionado o valor no objeto. Se desejarmos modificar uma propriedade, basta usar a mesma notação, especificando uma chave já existente no objeto. Enquanto que, no exemplo abaixo, iremos remover a propriedade idade do objeto pessoa

```
let pessoa = {
   "primeiroNome": "Carlos",
   "ultimoNome":  "Dias"
}

pessoa.idade= 27;
delete
```

É possível também criar funções integradas ao objeto, onde se adiciona métodos que serão utilizados como parte desse objeto. Para isso, atribui-se uma propriedade ao objeto, e chama-se a função a partir dessa propriedade. Assim, quando se tem uma função como propriedade de um objeto, chamamos essa condição de método. O código abaixo, exemplifica o uso da função através da propriedade “falar” com função dizerOi().

```
let pessoa = {
   "primeiroNome": "Carlos",
   "ultimoNome":  "Dias"
}

function dizerOi () {
   console.log ("Oi eu sou uma pessoa!");
}

pessoa.falar = dizerOi
pessoa.falar();
```

1.3 Funções

Agora nós iremos desbravar o universo das funções em JavaScript, lembrando que, em linguagens de programação, as funções podem ser definidas como subconjuntos de linhas de códigos, como se fosse um subprograma, que pode ser chamado por outro. Ou seja, assim como nos laços de repetição, que nos permite ter blocos de código que vão ser repetidos, nas funções também podemos fazer blocos, que serão utilizados em diferentes locais do código e até mesmo repetidas vezes. Além disso, as funções podem receber argumentos, ou seja, receber dados para usar dentro dos blocos de código.

É importante salientar que uma função pode ter várias linhas e não necessariamente precisa retornar algo ou receber algum argumento. Para retornar a função é necessário utilizar a palavra reservada function, e na sequência, o nome da função, a lista de argumentos para a função entre parênteses e separados por vírgulas, e por último as declarações JavaScript que definem a função entre chaves { }.

Agora que relembramos o conceito de função, vamos exemplificar a declaração de uma função, seu escopo e o uso de objetos de argumento. Para que possamos declarar uma função é necessário nomeá-la e chamá-la, sendo assim, teremos abaixo um exemplo de definição de uma função square.

```
function square(numero) {
    return numero * numero;
}
```

No exemplo pudemos perceber que a função declarada (square) recebe um argumento nomeado como "número", o que significa que será retornado o argumento da função multiplicado por si mesmo, "número"x"número", onde a declaração return (en-US) citada anteriormente irá especificar o valor retornado pela função.

Agora que relembramos o conceito de função, iremos exemplificar o escopo de uma função e o uso de objetos de argumento. O escopo pode ser entendido como a acessibilidade das funções, objetos e variáveis em diversas partes do código, ou seja, o escopo irá definir o que pode ser acessado em uma parte específica do código. Nesse sentido, em termos de acessibilidade podemos utilizar as funções, uma vez que, uma função possui capacidade para acessar todas variáveis, bem como, funções definidas fora do escopo onde ela está definida.

Nesse sentido, vamos agora a partir do exemplo abaixo verificar a possibilidade de acesso das variáveis definidas na função hospedeira e os demais acessos.

```
let num1 = 10,
   num2 = 3,
   nome = "Carlos";

// Essa função é definida no escopo global
function multiplica() {
 return num1 * num2;
}

multiplica(); // Retorna 30

//Função aninhada
function getScore () {
 let num1 = 1,
     num2 = 3;

 function add() {
   return nome + " fez " + (num1 + num2) + " pontos";
 }

 return add();
}

getScore(); // Retorna "Carlos fez 4 pontos"
```

Além do que vimos com o exemplo anterior, podemos utilizar as funções também para chamar a si mesma, o nome dado a esse tipo de função é Recursiva, e para que isso seja possível, podemos usar o nome da função, arguments.callee (en-US), ou ainda, uma variável no escopo que se refere a função.

Quando uma função é recursiva, existem situações em que similar a laços de repetição, já que na função recursiva também se tem a execução do código por diversas

vezes, além do que, ambas necessitam de uma condição, para que não se tenha um laço ou recursão infinita. No exemplo a seguir podemos observar o uso de uma função Recursiva.

```
function loop (x) {
  if (x >= 5) // "x >= 5" a condição de parada (equivalente a "!(x < 5)")
     return;
  // faça isso
  loop(x + 1); // chamada recursiva
}
loop(0);
```

Atividade: Para concluir, vá até a sala virtual e responda o questionário desta semana.

Objetivos
Nesta semana entenderemos como é estruturado o DOM, e estudar como modificar elementos visuais em uma página da web utilizando JavaScript.

2.1 O que é o "Document Object Mode" (DOM)

Com os conhecimentos básicos em linguagem de programação sabemos que uma página web é um documento, e que este documento pode ser manipulado e exibido como fonte HTML, ou ainda na janela de um navegador.

Seguiremos agora para a manipulação do visual, para isso é importante entendermos o conceito de "Document Object Mode" (DOM), e algumas de suas aplicações. Nesse sentido, temos o DOM como a representação dos dados que vão constituir a estrutura, bem como, o assunto que estará presente no documento da Web.

Podemos pensar no DOM como uma interpretação orientada aos objetos da página da web, onde se pode fazer alterações a partir de uma linguagem de programação como JavaScript.

Para que possamos manipular o visual é necessário que tenhamos acesso à árvore DOM do navegador. Neste caso, o grande responsável em JavaScript é o objeto document. Sendo assim, quando algo for criado pelo navegador Web, podemos acessá-lo a partir do objeto Javascript document. Existem diversos métodos que o objeto document permite o acesso, podemos ver alguns destes e suas execuções no quadro a seguir:

Método	Executa
createElement	Cria um nodo elemento na página.
documentElement	Captura o elemento raiz <html> de um documento HTML.
getElementsByTagName	Retorna um array dos elementos com o mesmo nome.
getElementById	Busca um elemento da página Web com o uso do atributo id do elemento.

Tabela 1 - Métodos básicos para manipulação do DOM.
Fonte: autoral

Para entendermos melhor a relação entre DOM e o JavaScript pense da seguinte maneira: **Página Web** = **DOM** + **JavaScript**. Podemos observar que o conteúdo das páginas Web são armazenados no DOM e são acessados via JavaScript, logo a junção de ambos constituem a representação estrutural de toda a aplicação.

Na figura abaixo, temos um exemplo de árvore DOM de uma página Web, e nele conseguimos observar que os elementos da página Web estão todos acessíveis para serem manipulados.

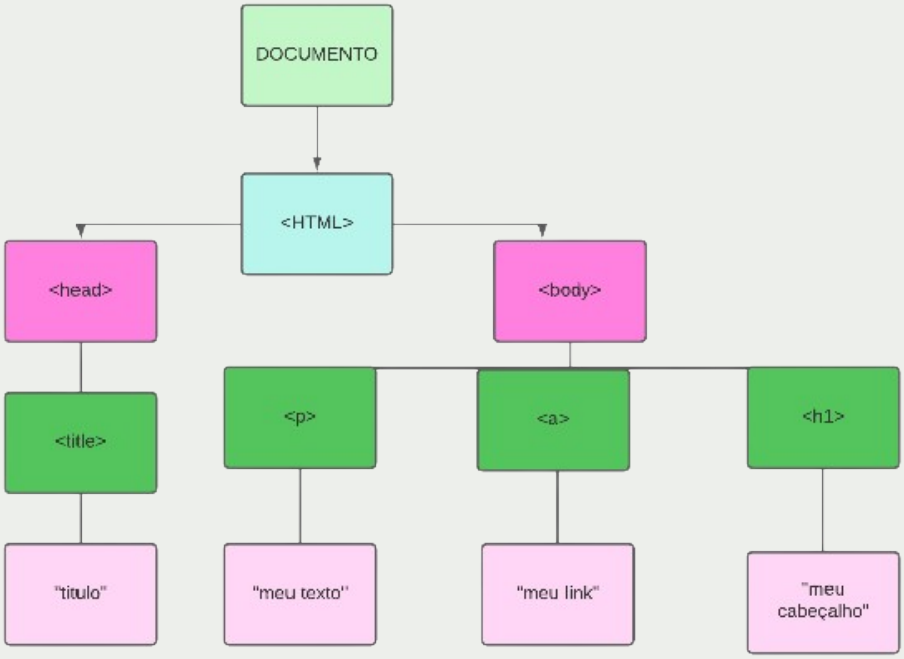

Figura 1- Árvore DOM de uma página Web.
Fonte: autoral

No geral, podemos pensar nas páginas Web como árvores, sendo assim, teríamos como raiz o elemento HTML e seus “brotos” como head e o body, que geram outros brotos sucessivamente. Aqueles elementos que não possuem “brotos”, são chamados de nós folhas, sendo entendidos como elementos de apresentação, que no caso da figura acima são os elementos: “titulo”; “meu texto”, “meu link”, “meu cabeçalho”.

Até então podemos percorrer uma árvore utilizando a variável document, e que existem conexões,“brotos” e “folhas”, que possibilitam percorrer essa árvores DOM subindo e descendo a partir dessas conexões. Lembrando que, os elementos “meu texto”, “meu link”, “meu cabeçalho”, e “titulo”, são elementos especiais de nós mas também são nós, sendo portanto, também agrupados pelo DOM.

2.1 Acessando e modificando o DOM

Mídia digital: Na videoaula “Manipulando o visual”, há um exemplo prático de como se pode modificar o que aparece ao usuário de sua aplicação. Veja na sala-virtual
este material!

A partir dos conhecimentos acumulados até agora vamos entender como modificar HTML a partir do JavaScript. Para que seja possível navegar através do código é importante que se tenha uma identificação única, e para isso se atribui um ID. Nesse sentido, cada elemento no documento terá uma identificação específica, ou seja, seu próprio ID. Utilizamos, o “getElementById” para optar por um elemento tomando como referência a identificação ID, sendo possível buscar e retornar o elemento a partir do seu identificador, como podemos observar no código abaixo:

```
<html>
<head>
   <title>Cadastro</title>
</head>
<body>
   <label>Digite o seu nome</label>
   <input id="nome">
   <script>
       let inputNome = document.getElementById("nome");
       console.log(inputNome);
   </script>
</body>
</html>
```

No código será exibido no console “<input id="nome">”, que é o elemento HTML que tem o id “nome”, retornando, portanto o elemento a partir do seu identificador. Devemos ficar atentos em relação ao uso do parâmetro ID, uma vez que este possui sensibilidade em relação a letras maiúsculas e minúsculas, sendo necessária atenção para que não seja retornado valor nulo.

Dando continuidade, para que possamos fazer alterações do conteúdo ao longo de uma página Web (DOM), utilizando o document.getElementById associado a innerHTML ou ao innerText. Apesar do innerHTML e innerText serem entendidos como elementos de

modificação, eles possuem notáveis diferenças, o innerHTML é responsável por alterar o conteúdo do elemento a partir do elemento HTML, enquanto que o innerText altera o conteúdo do elemento apenas como texto, iremos entender melhor essas diferenças a partir do exemplo abaixo,

```
<html>
<head>
   <title>Teste</title>
</head>
<body>
   <p id="texto">Texto aqui</p>
   <p id="conteudo">Nada aqui</p>
   <button onclick="mudar()">Mudar</button>
   <script>
       function mudar(){
           document.getElementById("texto").innerText = "Novo
texto";
           document.getElementById("conteudo").innerHTML = "Texto
com <b>HTML</b>";
       }
   </script>
</body>
</html>
```

2.1 Eventos do DOM

Como estamos trabalhando com manipulação do DOM, até agora foi possível alterar elementos visuais, textos e até mesmo adicionar HTML. Porém, uma parte importante no desenvolvimento de aplicações com interface com o usuário é a captura de interações com os elementos disponíveis. Por exemplo, se há um formulário a ser preenchido, e o usuário pode clicar em um botão que irá disparar o envio dos dados que preencheu, faz-se necessário uma maneira de se capturar o momento em que esta ação ocorreu, para então executar um código em JS.

Considerando os eventos de click com o mouse, veja o exemplo simples a seguir: nele temos um botão apenas na tela que ao ser clicado irá invocar a função “alertar”.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Document</title>
</head>
<body>
   <button onclick="alertar()">OK</button>
   <script>
       function alertar(){
           alert("Clicou no
           botão!");
       }
   </script>
</body>
```

Veja o destaque, em amarelo, na tag “button” em que é especificado a função alertar. Esta é uma maneira de se adicionar a execução de funções a partir de eventos diretamente no HTML. Porém, também é possível criar “ouvintes de eventos”.

Imagine o seguinte: precisamos que "alguém" fique observando se o botão foi clicado para chamar uma função quando isto ocorrer. Em JavaScript, isto é feito através de EventListeners.

Quando se cria um EventListener, não é necessário adicionar no HTML parametros nos elementos como “onclick”, pois os eventos serão adicionados diretamente via código JavaScript. Veja o código a seguir com exemplo de como criar um EventListener:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Document</title>
</head>
<body>
   <button id="ok">OK</button>
   <script>
```

```
        document.getElementById("ok")
        .addEventListener("click", alertar, false);

        function alertar(){
            alert("Clicou no
            botão!");
        }
    </script>
</body>
</html>
```

O funcionamento neste caso, é o mesmo do anterior: ao clicar no botão “OK”, será exibido um alerta ao usuário. Porém, este código é fácilmente modificavél para diferentes tipos de eventos disparados pelo usuário. Como exemplo, caso se deseje que dispare uma função quando o usuário passar o mouse sobre o botão, basta modificar o EventListener para a seguinte forma:

```
document.getElementById("ok")
.addEventListener("mouseover", alertar, false);
```

Atividade: Para concluir, vá até a sala virtual e responda o questionário desta semana.

Objetivos
Para agilizar o desenvolvimento de aplicações, é necessário conhecer várias ferramentas. Nesta semana aprenderemos sobre algumas ferramentas muito úteis e como obter dados de aplicações externas.

Quando trabalhamos com sistemas mais complexos, mas informações e mais recursos são necessários: colocar animações em certas partes do visual, mostrar ao usuário informações de sistemas externos, exibir gráficos de dados importantes ou até mesmo exibir gráficos em 3D. Programar todos os recursos necessários pode ser trabalhoso e demorado, o que complica a vida como desenvolvedor e pode comprometer a entrega do seu trabalho.

Por isso, é importante conhecer as chamadas APIs (Application Programming Interfaces) ou, em português, “Interfaces de programação de aplicativos”. Estas são construções que permitem ao desenvolvedor utilizar de funcionalidades complexas, de maneira mais simples, pois “empacotam” e abstraem um código complexo em uma interface amigável e acessível.

3.1 Browser APIs

Mídia digital: Na videoaula “Web API”, através de interface fornecida pelo navegador, criamos um sistema de cadastro com dados salvos localmente. Veja na sala- virtual este material!

Como estamos trabalhando com JavaScript essencialmente em um navegador, é fato que precisamos utilizar de ferramentas disponíveis nessa plataforma. Por exemplo, na seção anterior, utilizamos o Fetch, que é uma ferramenta que o próprio navegador faz requisições via HTTP para outros servidores.

Mas existem diversas dessas APIs disponíveis nos navegadores, e muitas delas não estão padronizadas (algumas estão disponíveis em apenas alguns navegadores). Porém, nesta seção vamos trabalhar com algumas APIs muito interessantes e úteis em aplicações.

A primeira delas é a Fullscreen API. Imagine a seguinte situação: você está fazendo uma loja online e gostaria que, quando o usuário clicar em um botão, todos os elementos da tela desaparecem, ficando apenas a imagem do produto em tela cheia.

Esta API pede ao navegador que seja exibida apenas uma parte da página em tela cheia. Veja o exemplo a seguir:

```
<html>
<head>
   <title>Cadastro</title>
</head>
<body>
   <label>Digite o CEP (apenas números)</label>
   <input id="cep" type="number">
   <button onclick="buscarCEP()">Buscar</button>
   <script>
       function buscarCEP(){
           document.getElementById("cep")
           .requestFullscreen();
       }
   </script>
</body>
</html>
```

Observe que ao clicar no botão de buscar, o navegador fica em tela cheia, com apenas a caixa de entrada de dados do CEP. É importante salientar que esta API pode ser chamada apenas se houve alguma interação com o usuário, ou seja, não é possível iniciar a página em tela cheia, sem que o mesmo clique em um botão, por exemplo.

Outra API interessante é a Geolocation API: com ela é possível saber a localização do usuário que está acessando a página, claro que com autorização do mesmo. Veja o código JS de exemplo a seguir:

```
navigator.geolocation.getCurrentPosition(success, error);

function success(position) {
   const latitude = position.coords.latitude;
   const longitude = position.coords.longitude;
   alert(latitude+","+longitude)
}

function error() {
   alert("Erro ao tentar obter localização");
}
```

Ao testar este código, em uma página HTML, você verá uma caixa do navegador perguntando se autoriza compartilhar a localização. Se autorizado, será exibida a latitude e longitude em um alerta, se não irá exibir que não conseguiu obter localização. No caso de

computadores, que não têm hardware interno para obtenção de dados de posicionamento global (latitude e longitude) essa posição é aproximada. Mas para celulares e dispositivos móveis em geral, a precisão é muito boa.

Por fim, podemos utilizar a API mais interessante: Canva. Com esta, é possível desenhar na tela formas geométricas, apenas passando uma "tela" para fazer os desenhos, ou seja, um elemento HTML em que serão adicionados os elementos. Veja o código abaixo:

```
<!DOCTYPE html>
<html>
 <head>Canvas API</head>
 <body>
   <canvas id="canvas"> </canvas>
   <script>
       const canvas =
       document.getElementById("canvas"); const ctx =
       canvas.getContext("2d"); ctx.fillStyle =
       "yellow";
       ctx.fillRect(10, 10, 150, 100);
   </script>
 </body>
```

Observe que estamos pegando um elemento com id "canvas", que utilizaremos como área para desenho. Depois, escolhemos um contexto, que no caso é "2d" indicando que todos os desenhos serão em duas dimensões. Depois utilizamos os métodos da Canvas API para exibir ao usuário um retângulo amarelo.

Há várias outras APIs disponíveis no navegador, e o que apresentamos aqui é apenas o básico para utilizar algumas delas. Se desejar se aprofundar mais, veja os links de referência.

3.2 Web History

Esta talvez seja a API mais simples que vamos ver esta semana: o objetivo é apenas controlar o histórico de páginas navegadas pelo usuário. Obviamente, por questões de segurança, o sistema consegue acessar apenas o que foi acessado na aba atual, controlando redirecionamentos.

Para começar, vejamos a função "back": ela funciona como o botão de voltar (aquela seta para a esquerda que geralmente os navegadores têm próximo à barra de endereços) fazendo com que o navegador redirecione o usuário para a página que ele estava anteriormente. A sintaxe é simples:

```
window.history.back();
```

É possível inserir esta linha em uma função de evento de click em um botão de sua aplicação, por exemplo, para fazer o usuário voltar à página anterior (como em aplicativos para celular, que têm o botão de voltar no topo).

E se quiser que o usuário queria avance no histórico (por exemplo, fez ele voltar uma página para preencher um formulário e agora quer que ele volte para a página que estava) basta usar o método “forward”, como a seguir:

```
window.history.forward();
```

Essa navegação é simple e pode ser usada também através da função “go”: por exemplo, se a aplicação precisa que o usuário volte 2 páginas no histórico, basta fazer

```
window.history.go(-2);
```

3.3 Web Workers API

Como estamos pensando em aplicações complexas nesse curso, cheias de recursos e com várias funções sendo executadas pode acontecer de haver necessidade de alguma parte do código ser executada em paralelo com outra. Imagine a seguinte situação: sua aplicação tem dados a serem tratados, enquanto o usuário interage com outras funcionalidades. Da maneira que estamos trabalhando atualmente o código é todo síncrono (veremos na semana que vem como contornar isso de outras maneiras) o que significa que as tarefas precisam ser completadas para outras serem executadas.

Porém, nos navegadores modernos, surge a possibilidade de criar tarefas em paralelo (“threads” como se diz no mundo da computação) que irão executar códigos de forma simultânea. Para isto, há a Web Worker API: um trecho de código que é executado em paralelo, mas que pode comunicar com o código principal, enviando informações e dados de variáveis calculadas.

Para se utilizar um Web Worker basta criá-lo da seguinte maneira:

```
const worker = new Worker("nome_do_arquivo.js");
```

Observe que o código do Worker precisa estar em um arquivo de forma separada, e essa variável é criada na página da aplicação. Depois de criado, o Worker precisa ser iniciado, isto é, executar a tarefa que irá rodar em paralelo, da seguinte maneira:

```
worker.postMessage();
```

Dica do Professor: Na sala virtual, há um arquivo html de exemplo de teste de Web Workers sem utilizar um arquivo externo. É uma maneira legal de testar essas funcionalidades, rodando localmente!

O interessante é que o Worker pode enviar mensagens com os dados para o programa principal. Por exemplo, no programa principal, é criado um EventListener para mensagens vindo do worker: sempre que uma nova mensagem é recebida, é exibida no "console.log". Veja o código principal abaixo:

```
worker.addEventListener('message', function(e) {
console.log('Worker enviou essa mensagem: ', e.data);
}, false);

worker.postMessage('Ola worker!');//Envia a mensagem ao worker
```

Enquanto isso, no arquivo do worker, é criado um EventListener para receber as mensagens do programa principal e responder mensagens de volta, como no código a seguir:

```
self.addEventListener('message', function(e) {
      self.postMessage(e.data);
      }, false);
```

Atividade: Para concluir, vá até a sala virtual e responda o questionário desta semana.

Objetivos
Com foco em muitas ações simultâneas, nesta semana final entenderemos como utilizar funções assíncronas de forma a evitar que a interface esteja travada para o usuário, enquanto várias ações são feitas pela aplicação.

Para falarmos do modelo de programação “Assíncrono” em Javascript é importante diferenciá-lo do modelo "Síncrono". Em uma programação Síncrona quando se chama uma função, essa concretiza uma ação durável e seu retorno é feito apenas quando a ação tiver sido completada, ou seja, o programa é detido pelo tempo que a ação leva para ser finalizada. No modelo assíncrono, por outro lado, quando se chama uma função o programa continua sendo executado, assim, quando essa ação é encerrada o programa é “informado” e tem acesso ao resultado, ou seja, nesse modelo várias coisas acontecem ao mesmo tempo.

Com a diferenciação do modelo Sincrono e Assincrono, conseguimos entender a importância de executar tarefas sem nosso controle direto, e de maneira independente da concretização da ação. Para entendermos como isso é possível em JavaScript, vamos entender como esta linguagem se comporta.

O que acontece, é que o JavaScript acaba separando seu código em 2 diferentes partes, a primeira parte fica com o que deve ser rodado imediatamente, enquanto que a segunda é composta por ações que são executadas posteriormente. Para ficar mais claro vamos observar o código abaixo, onde são executadas as ações linha após linha, configurando assim um modelo do tipo Síncrono.

```
function somar(num1, num2) {
   return num1 + num2;
}
console.log(somar(2, 4));
```

4.1 Promisses

Agora que entendemos a fundo o conceito de modelo Sincrono e Assincrono, pensemos no caso de precisarmos receber dados de uma API, neste caso, seria preciso aguardar o pedido e a resposta da API, já que não se pode simplesmente barrar o funcionamento do programa para que nossas solicitações sejam atendidas, nesse sentido, começaremos a entender o conceito de “Promises” ou no sentido literal Promessas que são usadas no processamento Assíncrono, para retornar um objeto que represente o evento futuro.

Podemos conceituar Promises em JavaScript como sendo uma ação assíncrona, que pode, em algum momento, ser completada e produzir valor, retornando seu valor quando pronta. Por exemplo, quando enviamos uma requisição de dados a uma API, utilizamos o Promises como uma garantia de que em algum momento essa requisição será comprida, porém, caso não seja, por algum motivo específico, o sistema continua rodando. Uma das maneiras de se utilizar o Promises é a partir do método .then(), iremos compreendê-lo melhor a partir do código abaixo. Neste exemplo utilizamos a API de testes Reqres, que produz dados fictícios para teste de sistemas web.

```
function getUserEmail(id) {
const userData = fetch("https://reqres.in/api/users/"+id)
  .then(resposta => resposta.json())
  .then(dados => console.log(dados.data.email))
}

getUserEmail(2);
```

No código acima podemos observar a função getUserEmail() recebe um id de usuário como parâmetro, que é utilizado para compor a URL da requisição utilizada para obter o dado desejado. Em seguida, o fetch retorna uma Promise*, que* irá buscar os dados nessa URL. Em seguida, é tratada a resposta obtida pelo fetch, com o then, em que a resposta será transformada em JSON. A função que faz esta última conversão também retorna uma Promisse, então precisamos novamente do then para indicar que, após finalizada a conversão, exiba no console o email do usuário.

4.2 Async/Await

Vamos agora entender o funcionamento do objeto Promise e seus métodos a partir da expressão “Async” e “Await”. O async/await é uma maneira de simplificar a programação do tipo Assíncrona, já que a partir dele podemos estruturar um código de maneira síncrona podendo porém fazê-lo funcionar de maneira Assíncrona. Para isso ser possível, o async/await trabalha o código baseado em Promises, tornando essas promessas de forma “restrita” para garantir a fluidez do código.

Agora iremos entender como utilizar a função async/await, para isso, é importante primeiro definirmos a função como async, para depois utilizarmos a await antes de que se tenha algum retorno de promessa, ou seja, a função async é pausada até que a Promise seja cumprida. Explicando de maneira mais simples: O uso da await permite que aguardemos a finalização da Promise, caso a finalização aconteça utilizamos o termo *“fulfilled”,* e caso não aconteça entendemos como erro e o termo utilizado é "rejected". Iremos agora a partir do código abaixo entendermos melhor o funcionamento.

```
let response = await fetch("https://reqres.in/api/users/"+id);
let userData = await response.json();
```

Para utilizarmos await devemos obrigatoriamente criar uma função assíncrona, especificada com a palavra reservada async, como podemos observar abaixo.

```
async function getUserData(userId) {
   let response = await fetch("https://reqres.in/api/users/"+id);
   let dados = await response.json();
   return dados.data;
}
```

Observe que os dados são retornados pela função apenas depois que todas as Promises são respondidas, no caso a Promise do fetch e a de conversão de JSON.

4.3 Aplicação: consumindo API externa

Mídia digital: Na videoaula "JS Assincrono", consumimos uma API externa para auto completar um formulário de cadastro. Veja na sala-virtual este material!

Uma maneira interessante que vimos até agora de utilizar sistemas assíncronos é consultar dados em servidores externos: utilizando fetch, com Promises e convertendo os dados para String, tudo fica muito mais rápido e dinâmico. Por exemplo, se desejamos completar o endereço para um usuário através do CEP que ele preencheu, podemos utilizar os dados disponibilizados pelo projeto Brasil API. Para isto, é necessário apenas acessar uma URL com o CEP e será retornado um JSON com as informações de endereço de acordo com o que foi fornecido.

A primeira etapa é criar um arquivo HTML, com os itens básicos para o sistema: um campo para o usuário digitar o CEP, um botão de busca e uma área para aparecer o resultado da busca. Veja o código a seguir:

```
<html>
<head>
```

```
    <title>Cadastro</title>
</head>
<body>
    <label>Digite o CEP (apenas números)</label>
    <input id="cep" type="number">
    <button onclick="buscarCEP()">Buscar</button>
    <p id="endereco"></p>
    <script>
        CODIGO JS AQUI
    </script>
</body>
</html>
```

Destacado em amarelo, temos os seguintes elementos: uma “label”, que exibe ao usuário o que ele deve digitar, uma “input” com um identificador (id) “cep” para o usuário digitar apenas os números do CEP (por isso está escrito “type=’number’”) , o botão que irá chamar a função “buscar CEP” e um campo de texto que iremos inserir o resultado da busca.

Vamos adicionar o código básico para a função “buscarCEP”. Primeiramente, vamos testar se conseguimos capturar o que o usuário digitou, e mostrar no campo de texto abaixo, como no exemplo a seguir:

```
function buscarCEP(){
    let cep = document.getElementById("cep").value;
    let endereco = "esse é o CEP:"+cep;
    document.getElementById("endereco").innerText = endereco;
}
```

Agora que já está ok, podemos usar a API Fetch do JS: com ela é possível realizar pedidos e obter respostas de páginas externas e internas ao sistema que estamos programando. Basicamente, a Fetch recebe um argumento que é a URL a ser acessada, e precisamos de dois métodos para tratar os dados recebidos: “then” e “catch”. Enquanto este primeiro é responsável por tratar os dados depois de serem obtidos, o segundo é chamado quando um erro é gerado. Então, o que estamos fazendo, “traduzindo” para português, é dizendo ao programa “Busque (fetch) neste endereço e depois (then) faça isso, se der erro capture(catch) e faça isso”. Veja o código a seguir com uma aplicação simples.

```
function buscarCEP(){
```

```
    let cep = document.getElementById("cep").value;

    fetch("https://brasilapi.com.br/api/cep/v2/"+cep)
    .then(function(response) {
        response.json().then(function(dados) {
            console.log(dados);
        });
    })
    .catch(function() {
        document.getElementById("endereco").innerText = "[ERRO]";
    });
}
```

Para testar o que o sistema do Brasil API retorna, faremos um teste com o CEP do IFMG campus Ibirité: 32407190. Ao clicar no botão de busca, será exibido no console do navegador os dados, como a seguir:

```
{
    cep: "32407190"
    city: "Ibirité"
    location:
        coordinates: {longitude: '-44.0370399', latitude: '-20.0363767'}
        type: "Point"
        [[Prototype]]:
        Object
    neighborhood: "Vista Alegre - 1ª Seção"
    service: "correios"
    state: "MG"
    street: "Rua Mato Grosso"
}
```

Observe que se trata de um JSON, com as informações estruturadas como a cidade, nome da rua e estado. Iremos exibir as informações organizadas da seguinte maneira: Rua *[street]* , Bairro *[neighborhood]* , *[city]* - *[state]*". Para isto, devemos fazer uma união dos dados recebidos através da consulta. Veja o código abaixo:

```
function buscarCEP(){
    let cep = document.getElementById("cep").value;
    document.getElementById("endereco").innerText = "Buscando...";
    fetch("https://brasilapi.com.br/api/cep/v2/"+cep)
    .then(function(response) {
```

```
        response.json().then(function(dados) {
            console.log(dados);
            document.getElementById("endereco").innerText = dados.street +
            ", Bairro "+dados.neighborhood +","+
            dados.city+"-"+dados.state;
        });
    })
    .catch(function() {
        document.getElementById("endereco").innerText = "[ERRO]";
    });
}
```

Além disso, é necessário destacar que a resposta do servidor externo pode demorar um pouco, então precisamos informar ao usuário que estamos buscando o dado que ele pediu. Para isto, veja a linha em destaque (de amarelo): antes de fazer a busca usando Fetch, é exibido ao usuário o texto “Buscando…” para que o mesmo saiba que uma operação está sendo realizada. Depois de buscar, essa caixa de texto é preenchida com o valor encontrado.

Atividade: Para concluir o curso e gerar o seu certificado, vá até a sala virtual e responda ao Questionário “Avaliação final”. Este teste é constituído por 10 perguntas de múltipla escolha, que se baseiam em todo o conteúdo estudado.

Eis que nossa jornada chega ao fim! Foi uma aventura longa, mas muitas habilidades foram adquiridas até aqui. Espero que seja apenas o primeiro de vários aprendizados que terá daqui para frente, e que tenha muito sucesso!

Referências

GRILLO, Filipe Del Nero; FORTES, RENATA PONTIN DE MATTOS. **Aprendendo JavaScript**. São Carlos: USP, 2008.

HAVERBEKE, Marijn. **Eloquent javascript: A modern introduction to programming**. No

MDN. **Primeiros passos com JavaScript**. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/First_steps>. Acesso em: 9 abr. 2022.

Microsoft. **JavaScript na Microsoft**. Disponível em: <https://docs.microsoft.com/pt-br/javascript/>. Acesso em: 9 abr. 2022.

ROSSETTO, A. G. DE M. **Apostila de Linguagem de Programação Web**. Rio Grande do Sul: Universidade Aberta do Brasil/Instituto Federal Sul-rio-grandense, 2012.

Starch Press, 2018.

w3Schools. **JavaScript tutorial**. Disponível em: <https://www.w3schools.com/js/>. Acesso em: 9 abr. 2022.

Currículo do autor

Técnico em Eletrônica(2011) e Engenheiro de Automação Industrial(2017) ambos pelo Centro Federal de Educação Tecnológica de Minas Gerais. Experiência em pesquisa e desenvolvimento na área de engenharia elétrica. Durante a graduação participou de diversas competições de robótica, além de programas de iniciação científica. Foi professor de disciplinas na área de controle e automação, como "Modelamento de sistemas de controle", "Sistemas Multivariável", "Sistemas microprocessados", "Programação de computadores" e "Segurança e confiabilidade na automação industrial". Além disso, orientou equipes do ensino médio para participar nas Olimpíadas Brasileira de Robótica, e do ensino superior em competição internacional do INPE, o Cubedesign. Atualmente é professor do ensino básico, técnico tecnológico no Instituto Federal de Minas Gerais, campus Ibirité e membro do grupo de pesquisa Robotics and Intelligent Systems - EPIIBOTS.

Currículo Lattes: http://lattes.cnpq.br/4034444219840272

Feito por professor	Data	Revisão de *layout*	Data	Versão
Carlos Dias da Silva Junior	18/08/2022	Viviane L Martins	21/08//2022	1.0

Glossário de códigos QR (*Quick Response*)

Mídia digital
Videoaula 1
“Revisando JS”.

Mídia Digital
Videoaula 2“Manipulando o visual”

Mídia digital
Videoaula 3
“Web API”

Mídia digital
Videoaula “JS Assincrono”

Dica do professor
Mais informações sobre métodos de arrays

Plataforma +IFMG

Formação Inicial e Continuada EaD

A Pró-Reitoria de Extensão (Proex), desde o ano de 2020, concentrou seus esforços na criação do Programa +IFMG. Esta iniciativa consiste em uma plataforma de cursos *online*, cujo objetivo, além de multiplicar o conhecimento institucional em Educação a Distância (EaD), é aumentar a abrangência social do IFMG, incentivando a qualificação profissional. Assim, o programa contribui para o IFMG cumprir seu papel na oferta de uma educação pública, de qualidade e cada vez mais acessível.

Para essa realização, a Proex constituiu uma equipe multidisciplinar, contando com especialistas em educação, *web design*, *design* instrucional, programação, revisão de texto, locução, produção e edição de vídeos e muito mais. Além disso, contamos com o apoio sinérgico de diversos setores institucionais e também com a imprescindível contribuição de muitos servidores (professores e técnico-administrativos) que trabalharam como autores dos materiais didáticos, compartilhando conhecimento em suas áreas de atuação.

A fim de assegurar a mais alta qualidade na produção destes cursos, a Proex adquiriu estúdios de EaD, equipados com câmeras de vídeo, microfones, sistemas de iluminação e isolação acústica, para todos os 18 *campi* do IFMG.

Somando à nossa plataforma de cursos *online*, o Programa +IFMG disponibilizará também, para toda a comunidade, uma Rádio *Web* Educativa, um aplicativo móvel para Android e IOS, um canal no Youtube com a finalidade de promover a divulgação cultural e científica e cursos preparatórios para nosso processo seletivo, bem como para o Enem, considerando os saberes contemplados por todos os nossos cursos.

Parafraseando Freire, acreditamos que a educação muda as pessoas e estas, por sua vez, transformam o mundo. Foi assim que o +IFMG foi criado.

O +IFMG significa um IFMG cada vez mais perto de você!

Professor Carlos Bernardes Rosa Jr.
Pró-Reitor de Extensão do IFMG

Características deste livro:
Formato: A4
Tipologia: Arial e Capriola.
E-book:
1ª. Edição
Formato digital

Instituto Federal de Minas Gerais
Pró-Reitoria de Extensão

Instituto Federal de Minas Gerais
Pró-Reitoria de Extensão