Manipulação Avançada do DOM com `querySelector` e `querySelectorAll`

1. A EVOLUÇÃO DA SELEÇÃO DE ELEMENTOS NO DOM

Em JavaScript, a capacidade de selecionar e interagir com elementos da página é a base da manipulação do DOM (*Document Object Model*). Tradicionalmente, isso era feito com uma variedade de métodos, cada um com uma finalidade específica. No entanto, o desenvolvimento moderno da linguagem nos trouxe os métodos **`querySelector`** e **`querySelectorAll`**, uma abordagem mais poderosa e flexível que centraliza a lógica de seleção de elementos em uma sintaxe única e familiar.

1.1. Por que `querySelector` é Mais Dinâmico?

Os métodos tradicionais, como `getElementById`, `getElementsByTagName` e `getElementsByClassName`, são eficazes, mas limitados em seu escopo. Cada um foi projetado para um único tipo de seletor:

- `getElementById` busca um elemento por seu atributo `id`.
- `getElementsByTagName` busca uma coleção de elementos pelo nome da sua tag (ex: `div`, `p`).
- `getElementsByClassName` busca uma coleção de elementos que compartilham uma classe CSS.

Essa especialização obriga o desenvolvedor a alternar entre diferentes métodos dependendo do que precisa selecionar. Em contraste, `querySelector` e `querySelectorAll` surgem como uma solução genérica e poderosa. Eles utilizam a mesma sintaxe dos seletores CSS, permitindo selecionar qualquer elemento — seja por ID, classe, tag, atributo ou uma combinação complexa deles — através de um único padrão de chamada.

Essa abordagem unificada não apenas simplifica o código, mas também abre um leque de possibilidades para seleções mais precisas e contextuais. A seguir, vamos explorar a diferença fundamental que define quando usar um ou outro.

1.2. Um Elemento vs. Uma Coleção

O propósito desta seção é clarificar a distinção crucial no comportamento e, principalmente, no valor de retorno entre `querySelector` e `querySelectorAll`. Entender essa diferença é o primeiro passo para utilizá-los de forma correta e eficiente.

1.2.1. querySelector: Retornando o Primeiro Elemento

O método `querySelector` varre o documento a partir do topo e retorna **apenas o primeiro elemento** que corresponde ao seletor CSS especificado. Mesmo que existam dezenas de elementos que atendam ao critério de busca, ele para e retorna assim que encontra o primeiro.

```
// Seleciona a PRIMEIRA tag 'div' que encontrar no documento.
const primeiraDiv = document.querySelector('div');

// Ao imprimir, veremos apenas um único elemento HTML.
console.log(primeiraDiv);
```

Este comportamento o torna ideal para buscar elementos que você sabe que são únicos (como um elemento com um ID específico) ou quando você só precisa interagir com a primeira ocorrência de um tipo de elemento.

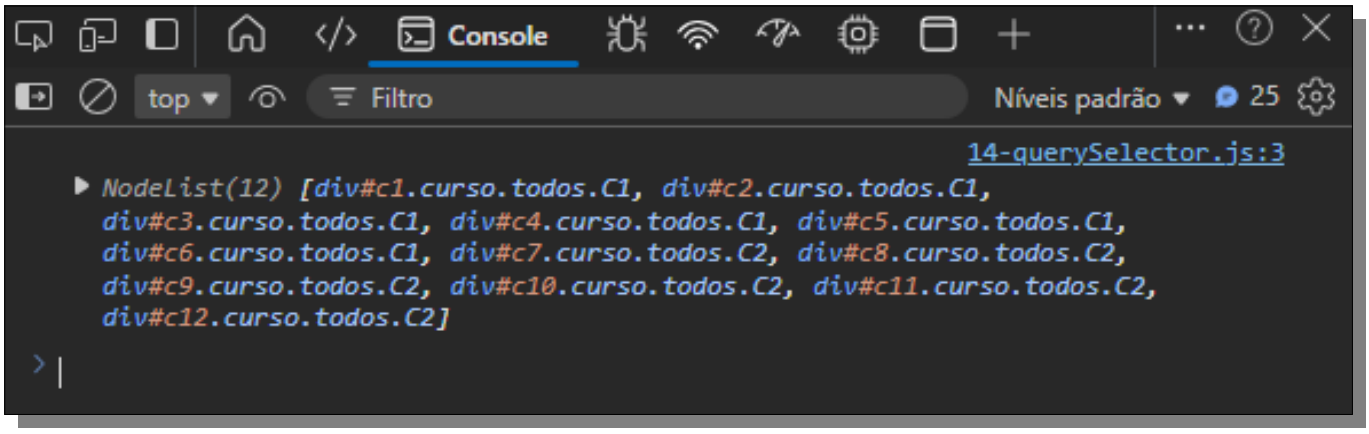
1.2.2. querySelectorAll: Retornando Todos os Elementos

Em contrapartida, o `querySelectorAll` não para no primeiro resultado. Ele continua varrendo todo o documento e retorna **todos os elementos** que correspondem ao seletor. O resultado é entregue em uma coleção estática chamada `NodeList`, que se assemelha a um array.

```
// Seleciona TODAS as tags 'div' do documento.
const todasAsDivs = document.querySelectorAll('div');

// Ao imprimir, veremos uma NodeList (uma coleção de nós) com todos os elementos
'div'.
console.log(todasAsDivs);
```

```
<div id="c1" class="curso todos C1">HTML</div>
<div id="c2" class="curso todos C1">CSS</div>
<div id="c3" class="curso todos C1">Javascript</div>
<div id="c4" class="curso todos C1">PHP</div>
<div id="c5" class="curso todos C1">React</div>
<div id="c6" class="curso todos C1">MySQL</div>
<div id="c7" class="curso todos C2">C++</div>
<div id="c8" class="curso todos C2">C#</div>
<div id="c9" class="curso todos C2">Arduino</div>
<div id="c10" class="curso todos C2">React Native</div>
<div id="c11" class="curso todos C2">Python</div>
<div id="c12" class="curso todos C2">Unity</div>
```



Este método é a escolha certa quando você precisa operar em um grupo de elementos, como aplicar um estilo a todos os itens de uma lista ou adicionar um evento a vários botões. Agora que a diferença de retorno está clara, vamos ver como a sintaxe unificada funciona na prática.

1.3. Sintaxe Unificada: Selecionando por Tag, Classe e ID

A principal vantagem dos *query selectors* é a sua sintaxe familiar, emprestada diretamente do CSS. Isso centraliza a lógica de seleção em um padrão consistente, independentemente do tipo de seletor que você está utilizando.

1.3.1. Selecionando por Classe

Para selecionar elementos por sua classe, utilizamos o mesmo prefixo de ponto (`.`) usado no CSS. O `querySelectorAll` retornará uma `NodeList` com todos os elementos que possuem a classe especificada.

```
// O ponto (.) antes de "curso" indica que estamos selecionando por classe.
const todosOsCursos = document.querySelectorAll('.curso');

console.log(todosOsCursos);
```

1.3.2. Selecionando por ID

Para selecionar um elemento pelo seu ID único, utilizamos o prefixo de cerquilha (#). Como os IDs devem ser únicos em um documento, existem duas abordagens possíveis, cada uma com um resultado ligeiramente diferente.

- **Exemplo 1: Usando `querySelector` para obter o elemento diretamente.** Esta é a forma mais comum e direta, pois retorna o elemento HTML diretamente.
- **Exemplo 2: Usando `querySelectorAll` e acessando o primeiro índice.** Mesmo para um ID único, `querySelectorAll` retornará uma `NodeList` contendo um único item. Para acessar o elemento, é preciso indicar o índice `[0]`.

Nota: *Como IDs devem ser únicos por definição, o uso de `document.querySelector('#meuID')` é geralmente mais prático e recomendado do que `document.querySelectorAll('#meuID')[0]`.*

Com a capacidade de selecionar múltiplos elementos, o próximo passo é aprender a manipular a coleção retornada pelo `querySelectorAll`.

1.4. Manipulando a `NodeList`

Quando utilizamos `querySelectorAll`, o resultado é uma `NodeList`. É crucial entender a diferença entre ela e a `HTMLCollection` retornada por métodos mais antigos como `getElementsByTagName`. Uma `HTMLCollection` é **viva** (*live*), ou seja, ela é atualizada automaticamente se novos elementos que correspondem ao seletor forem adicionados ao DOM. Em contraste, a `NodeList` retornada por `querySelectorAll` é **estática** — ela é um "instantâneo"

(*snapshot*) dos elementos no momento da chamada e não mudará, mesmo que o DOM seja alterado. Essa distinção é fundamental para evitar *bugs* inesperados.

1.4.1. Convertendo `NodeList` em Array

Em muitos cenários, é vantajoso converter a `NodeList` estática para um *Array* JavaScript padrão para ter acesso a um conjunto mais rico de métodos de manipulação. A conversão desbloqueia todo o poder dos métodos de *array* do JavaScript, como `map`, `filter` e `reduce`, que não estão disponíveis nativamente na `NodeList`. A maneira mais simples e moderna de realizar essa conversão é utilizando o operador *spread* (`...`).

```
// Primeiro, obtemos a NodeList com todos os elementos da classe '.curso'.
const cursosNodeList = document.querySelectorAll('.curso');

// Em seguida, usamos o operador spread (...) para criar um novo Array a partir
da NodeList.
const cursosArray = [...cursosNodeList];

// Agora 'cursosArray' é um Array de verdade e podemos usar métodos como map,
forEach, etc.
console.log('NodeList Original:', cursosNodeList);
console.log('Array Convertido:', cursosArray);
```

Por exemplo, após converter para `cursosArray`, você poderia facilmente obter um *array* de apenas os IDs dos cursos usando `cursosArray.map(curso => curso.id)`, uma tarefa que não é diretamente possível na `NodeList` original. Com os elementos devidamente organizados, podemos explorar as técnicas mais avançadas que os *query selectors* oferecem.

1.5. Técnicas de Seleção Avançadas

A verdadeira flexibilidade dos *query selectors* se revela em cenários de seleção mais complexos, que vão muito além do básico de tags, classes e IDs. A sintaxe CSS permite criar seletores compostos e contextuais de alta precisão.

1.5.1. Seleção Múltipla de Elementos

É possível selecionar múltiplos tipos de elementos em uma única chamada, simplesmente separando os diferentes seletores por uma vírgula (`,`). Isso funciona como um operador "OU", instruindo o método a buscar todos os elementos que correspondam a qualquer um dos seletores fornecidos.

```
// A vírgula funciona como um "OU" lógico: selecione todos os elementos 'div' E
TAMBÉM todos os elementos 'p'.
const divs_e_paragrafos = document.querySelectorAll('div, p');

// Retorna uma NodeList contendo ambos os tipos de elementos.
console.log(divs_e_paragrafos);
```

1.5.2. Seleção por Atributos

Podemos selecionar elementos com base na presença de um atributo específico, independentemente do seu valor. A sintaxe `[atributo]` permite filtrar elementos que contêm um determinado atributo em sua declaração HTML. Neste exemplo, selecionamos apenas os elementos `div` que possuem o atributo `class` definido.

```
// Seleciona todas as 'divs' que contêm o atributo 'class'.
// Divs sem esse atributo serão ignoradas.
const divsComClasse = document.querySelectorAll('div[class]');

console.log(divsComClasse);
```

1.5.3. Seleção Hierárquica (Descendentes)

Uma das capacidades mais poderosas é a seleção contextual. Ao colocar um espaço entre dois seletores, você especifica uma relação de descendência: "encontre o segundo tipo de elemento que esteja em qualquer lugar dentro do primeiro". Isso é distinto do seletor de filho direto (`div > p`), que selecionaria apenas elementos `<p>` que são filhos imediatos de uma `<div>`, não netos ou descendentes mais distantes. O seletor de descendente (espaço) é mais geral e, frequentemente, mais útil.

No exemplo abaixo, o seletor `div p` instrui o método a encontrar todos os elementos `<p>` que são descendentes (filhos, netos, etc.) de um elemento `<div>`.

```
// O espaço entre 'div' e 'p' significa: "selecione todo 'p' que esteja dentro
de uma 'div'".
// Isso é extremamente poderoso para filtrar seleções.
const paragrafosDentroDeDivs = document.querySelectorAll('div p');

console.log(paragrafosDentroDeDivs);
```

Nota: *No código que originou este exemplo, havia elementos `<p>` que não estavam aninhados em uma `<div>`. Esses elementos não foram incluídos no resultado final, pois o seletor especificava que o elemento `<p>` deveria ter uma `<div>` como ancestral (um elemento pai, avô, etc.). Isso demonstra a precisão e o poder de filtragem da seleção hierárquica.*

2. POR QUE ADOTAR `QUERYSELECTOR`?

Ao final desta exploração, fica claro que `querySelector` e `querySelectorAll` são muito mais do que simples substitutos para os métodos de seleção mais antigos. Eles representam uma mudança fundamental na forma como interagimos com o DOM, oferecendo dinamismo, uma sintaxe unificada e um poder de seleção avançado. As principais vantagens podem ser resumidas em:

- **Versatilidade:** Um único padrão de método para selecionar por tag, classe, ID, atributo ou qualquer combinação entre eles, simplificando o código e a lógica.
- **Poder:** Capacidade de realizar seleções complexas com base em hierarquia e atributos, permitindo um controle granular sobre quais elementos são retornados.
- **Modernidade:** Alinhado com as práticas modernas de desenvolvimento JavaScript e com o padrão de seletores CSS que os desenvolvedores já conhecem e utilizam.

Embora métodos como `getElementById` ainda sejam perfeitamente funcionais, eles oferecem uma vantagem de performance em cenários específicos. Essa ligeira vantagem de desempenho ocorre porque `getElementById` utiliza um mecanismo de busca direta e altamente otimizado no navegador, enquanto `querySelector` precisa primeiro analisar uma *string* de seletor CSS antes de iniciar a busca, introduzindo uma pequena sobrecarga. Para a grande maioria das aplicações, essa diferença é insignificante, e a flexibilidade do `querySelector` é uma troca que vale a pena. Sem dúvida, `querySelector` e `querySelectorAll` são as ferramentas preferenciais para a manipulação do DOM no desenvolvimento web moderno.