

Trabalhando com Expressões Regulares (Regex)

Por João Siles

Imagine que você precisa encontrar todos os arquivos de imagem em uma pasta, ou verificar se um usuário digitou um e-mail válido. Fazer isso manualmente seria um pesadelo, certo? É aí que as **Expressões Regulares (Regex)** entram em cena como verdadeiros superpoderes da manipulação de texto!

Em resumo, uma Regex é uma sequência de caracteres especiais que define um padrão de busca. Com ela, podemos:

- **Buscar** padrões específicos em textos (e-mails, números de telefone, datas, etc.).
- **Substituir** trechos de texto que possuem um padrão.
- **Validar** se uma entrada de dados segue um formato esperado (como um CPF!).
- **Extrair** informações específicas de um texto.

Pode parecer um pouco intimidante no começo, mas com os exemplos certos, você vai pegar o jeito rapidinho!

Existem duas formas principais de criar uma expressão regular em JavaScript (e em muitas outras linguagens):

- **Literal:** A Regex fica entre barras /. É usada quando o padrão é conhecido e não vai mudar.

```
1 const regexLiteral = /abc/; // Procura exatamente a sequência "abc"
```

- **Construtor RegExp:** Usamos a palavra-chave new RegExp() e a Regex é passada como uma string. É útil quando o padrão é dinâmico ou construído a partir de variáveis.

```
1 const padrao = "def";  
2 const regexConstrutor = new RegExp(padrao); // Procura o valor da variável 'padrao' ("def")
```

Desvendando os Caracteres Especiais da Regex

A beleza (e a possível confusão inicial) da Regex reside nos seus caracteres especiais. Vamos conhecer alguns dos mais usados:

- **.** (**Ponto**): Corresponde a qualquer caractere único, exceto uma nova linha.
- **\d**: Corresponde a qualquer dígito (0-9).
- **\w**: Corresponde a qualquer caractere alfanumérico (letras, números e o underscore _).
- **\s**: Corresponde a qualquer caractere de espaço em branco (espaço, tabulação, nova linha, etc.).
- **[]** (**Conjunto de caracteres**): Define um conjunto de caracteres que podem corresponder.
 - [abc] corresponde a 'a', 'b' ou 'c'.
 - [0-9] corresponde a qualquer dígito de 0 a 9 (o mesmo que \d).
 - [a-z] corresponde a qualquer letra minúscula.
 - [A-Z] corresponde a qualquer letra maiúscula.
 - [^abc] corresponde a qualquer caractere *que não* seja 'a', 'b' ou 'c'.
- **()** (**Grupo de captura**): Agrupa parte da expressão regular e permite capturar o texto correspondente.
- **|** (**OU**): Permite especificar alternativas. (a|b) corresponde a 'a' ou 'b'.
- **Quantificadores**: Indicam quantas vezes um caractere ou grupo pode ocorrer.
 - *: Zero ou mais vezes.
 - +: Uma ou mais vezes.
 - ?: Zero ou uma vez (opcional).
 - {n}: Exatamente n vezes.
 - {n,}: n ou mais vezes.
 - {n,m}: Pelo menos n e no máximo m vezes.
- **Âncoras**: Indicam a posição da correspondência.
 - ^: Início da string.
 - \$: Fim da string.
- **Escape (\)**: Usado para "escapar" caracteres especiais se você quiser procurá-los literalmente. Por exemplo, para procurar um ponto literal, use \.

Vamos criar um exemplo para CPF? Um CPF tem o seguinte formato: 000.000.000-00. Vamos construir nossa **Regex** passo a passo:

1. **Os três primeiros dígitos**: Cada dígito pode ser de 0 a 9, e temos três deles. Podemos usar \d{3}.
2. **O primeiro ponto**: É um caractere literal, então precisamos escapá-lo: \.
3. **Os três segundos dígitos**: Novamente, três dígitos: \d{3}.
4. **O segundo ponto**: Outro ponto literal: \.

5. **Os três terceiros dígitos:** Mais três dígitos: `\d{3}`.
6. **O hífen:** Um caractere literal: `-`.
7. **Os dois últimos dígitos:** Dois dígitos finais: `\d{2}`.

Juntando tudo, nossa expressão regular para o formato básico do CPF fica assim:



```
1 const regexCPFBasico = /^\\d{3}\\.\\d{3}\\.\\d{3}-\\d{2}$/;
```

- `^` : Garante que a correspondência comece no início da string.
- `\\d{3}` : Busca exatamente três dígitos.
- `\\.` : Busca um ponto literal.
- `-` : Busca um hífen literal.
- `\\d{2}` : Busca exatamente dois dígitos.
- `$` : Garante que a correspondência termine no final da string.

Vamos criar um arquivo chamado **cpf.html** e digitar o seguinte código:

```

1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Validação de CPF com Regex</title>
7  </head>
8  <body>
9      <h1>Validação de CPF</h1>
10     <form id="meuFormulario">
11         <div>
12             <label for="cpf">CPF:</label>
13             <input type="text" id="cpf" name="cpf" placeholder="000.000.000-00">
14             <p id="mensagemErro" style="color: red;"></p>
15         </div>
16         <button type="button" onclick="validarCPF()">Validar</button>
17     </form>
18
19     <script>
20         function validarCPF() {
21             const cpfInput = document.getElementById('cpf').value;
22             const mensagemErro = document.getElementById('mensagemErro');
23
24             const regexCPFBasico = /^\d{3}\.\d{3}\.\d{3}-\d{2}$/;
25
26             if (regexCPFBasico.test(cpfInput)) {
27                 mensagemErro.textContent = "CPF válido!";
28                 mensagemErro.style.color = "green";
29             } else {
30                 mensagemErro.textContent = "CPF inválido!";
31                 mensagemErro.style.color = "red";
32             }
33         }
34     </script>
35 </body>
36 </html>

```

Após o teste você verá que o campo de digitação do documento vai apenas aceitar o formato correto. Vale também as seguintes considerações:

- **Conjuntos de Caracteres Mais Específicos:** Se quiséssemos aceitar apenas CPFs que não começassem com '0', poderíamos usar `^[1-9]\d{2}\....`
- **O Quantificador Opcional ?:** Se a máscara do CPF fosse opcional (por exemplo, o usuário pudesse digitar 000000000000 ou 000.000.000-00), a **Regex** ficaria mais complexa, envolvendo o uso de ? para tornar os pontos e o hífen opcionais em certas posições (mas isso exigiria mais lógica para garantir a validade real).
- **Flags (Modificadores):** As Regex podem ter flags que alteram seu comportamento. Alguns exemplos:
 - i: Ignora a diferença entre maiúsculas e minúsculas (/abc/i corresponderia a "abc", "ABC", "aBc", etc.).
 - g: Busca todas as ocorrências do padrão (sem o g, a busca para na primeira correspondência).

Para adicionar uma flag na forma literal, colocamos após a última barra: `/abc/i`. Para o construtor `RegExp`, passamos a flag como segundo argumento: `new RegExp("abc", "i")`.