04

Template Strings

Transcrição

Vamos voltar ao arquivo `DateHelper.js` e analisar o método `dataParaTexto`. Neste, faremos a concatenação de *strings*:

```
class DateHelper {

    dataParaTexto(data) {

        return data.getDate()
            + '/' + (data.getMonth() + 1)
            + '/' + data.getFullYear();
      }
...
```

Adicionamos os parênteses para trabalharmos corretamente com o elemento `mês`. Nós temos a opção de utilizar um recurso das versões posteriores ao ES2015: ***template string***. Vamos ver como ela funciona.

Primeiramente, digitaremos no Console:

```
let nome = 'Flávio'
undefined
let idade = 18
undefined
console.log('A idade de' + nome + ' é ' + idade + '.')
A idade de Flávio é 18.
undefined
```

Para usarmos o template string, adicionaremos a seguinte linha:

```
console.log(`A idade de nome é idade.`)
```

Observe que adicionamos o ` (backtick), mas se executarmos o código desta forma, será exibida a frase: `A idade de nome é idade.`. Ele não entendeu que o `nome` deve ser substituído pelo valor da variável. Mas se colocarmos `nome` dentro de uma expressão, conseguiremos o resultado esperado.

```
console.log(`A idade de ${nome} é ${idade}.`);
A idade de Flávio é 18.
```

Com o uso de `${}` dentro da string, ele fará o mecanismo de **interpolação**. A expressão irá interpolar o conteúdo das variáveis `nome` e `idade` na string. Se entendemos corretamente esta estrutura, ela é menos sujeita a erro do que a anterior que continha várias concatenações. Se transpormos isto no `DateHelper`, podemos melhorar o código:

```
    return `${data.getDate()}/${data.getMonth()+1}/${data.getFullYear()}`;
```

Observe que somamos o mês com `1`. O trecho do código ficou assim:

```
class DateHelper {

    dataParaTexto(data) {

        return `${data.getDate()}/${data.getMonth()+1}/${data.getFullYear()}`;
    }

    static textoParaData(texto) {

        return new Date(...texto.split('-').map((item,indice) => item - indice % 2));
      }
  }
```

Desta vez, não foi necessário adicionar parênteses, porque cada expressão será avaliada individualmente para fazer a interpolação com a string. Outra vantagem do template string é que podemos apagar os sinais `+` entre as expressões e apenas, separar cada uma em linhas diferentes:

```
static dataParaTexto(data) {

    return `${data.getDate()}
        /${data.getMonth()+1}
        /${data.getFullYear()}`;
}
```

Mas manteremos o código como estava, apenas em uma linha. Feito isso, vamos testar o código no navegador.

A data aparecerá corretamente. Este foi o nosso primeiro contato com o template string, veremos que este é um recurso poderoso do JavaScript. Você verá o que faremos com ele. Por enquanto, já não precisamos ficar concatenando um monte de coisa, porque ele faz a interpolação automaticamente.

Outro ação que realizaremos: pediremos para o `DateHelper` converter o texto com `/`. No console, digitaremos:

```
DateHelper.textoParaData(`11/12/2017`)
```

Ele retornará que a data é inválida, porque o texto para data deve receber o `ano-mês-dia`. Nós já vamos validar na variável `textoParaData` se passamos uma string no padrão estabelecido, exibindo uma mensagem caso o padrão não seja exibido. Faremos algo denominado *fail-fast*, assim que passar algo errado pelo método, falharemos rápido.

Vamos adicionar um **expressão regular**, que será sinalizada por **barras** (`//`):

```
static textoParaData(texto) {

    /\d{4}-\d{2}-\d{2}/.test(texto)
    return new Date(...texto.split('-').map((item, indice) => item - indice % 2));
}
```

Os valores `4` , `2` e `2` sinalizam que os números terão tais quantidade de dígitos, respectivamente. Com `test` , pedimos que a expressão teste se o texto segue o padrão.

> Você pode se aprofundar no assunto com o curso de Expressão Regular (https://www.alura.com.br/curso-online-expressoes-regulares) da Alura.

Queremos lançar um erro caso o texto não siga o padrão, por isso, adicionaremos um `if` . Caso siga, o retorno será verdadeiro. Em seguida, adicionaremos o `throw new Error` >

```
if(!/\d{4}-\d{2}-\d{2}/.test(texto))
    throw new Error('Deve estar no formato aaaa-mm-dd');
```

A linha com o `throw new` só será executada se o `if` for **falso**, por isso, usamos o sinal de `!` .. Se quisemos colocar mais uma instrução abaixo, teremos que lembrar de colocá-las em um bloco usando `{}` e assim, evitar problemas. Será que funcionará? Vamos fazer um teste no Console:

```
DateHelper.textoParaData('2017-11-12')
Sun Nov 12 2017 00:00:00 GMT-0200 (BRST)
```

A data exibida está correta. Depois, forçaremos o erro no Console para ver o que acontece.

```
DateHelper.textoParaData('2017/11/12')
Uncaught Error: Deve estar no formato aaaa-mm-dd(..)
```

Ele nos retornará uma mensagem de erro. O mesmo ocorrerá se digitarmos no campo ano, por exemplo, um número com a quantidade de dígitos maior ou menor que `4` . Vemos que a expressão regular é usada justamente para encontrar padrões, podendo ser usadas no nosso código JavaScript. É a oportunidade de utilizarmos diferentes conhecimentos que vimos nos cursos da Alura.

Observe ainda que se inserirmos uma data com caracteres adicionais à esquerda ou à direita, o erro não será lançado. Por exemplo, se definirmos a data `30/09/20199` , o ano será truncado para os primeiros dígitos. Se informarmos ao método a data `130/09/2019` , o dia será truncado para os 2 últimos dígitos.

Para que a validação funcione corretamente, acrescente marcadores na expressão regular para indicar que queremos validar o texto exato, e nenhum caracter a mais:

```
if (!/^\d{4}-\d{2}-\d{2}$/.test(texto))
    throw new Error('Deve estar no formato aaaa-mm-dd');
```