01

Limitando operações - takeUntil

Transcrição

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/772-js-padrao-funcional/stages/04-project.zip) completo do projeto anterior e continuar seus estudos a partir daqui.

Nossa aplicação tem mais um requisito. O usuário só poderá executar a operação de busca de notas ficais clicando no botão três vezes! Em suma, aprenderemos como definir um limite máximo de execução para uma função. Porém, como todo bom programador, vamos isolar o código que limita a execução de funções para que possamos reutilizá-lo em todas as nossas aplicação.

Vamos declarar a função `takeUntil` em `app/utils/operators.js`, nome inspirado no *operator* `takeUntil` da biblioteca `RxJS`. Aliás, este curso é um curso cheio de inspirações, não?

```
// app/utils/operators.js
// código anterior omitido

export const takeUntil = (times, fn) => {

};
```

Nossa função recebe dois parâmetros. O primeiro indica o número máximo de vezes que executaremos uma função, já o segundo é a própria função que desejamos executar.

Nossa função precisará retornar uma nova função configurada para chamar a função `fn` um número máximo de vezes:

```
// app/utils/operators.js
// código anterior omitido
export const takeUntil = (times, fn) => {
    return () => {
        fn();
    }
};
```

Porém, do jeito que está, ela simplesmente chama a função `fn`. Que tal fazermos um teste para termos um *big picture* de como usaremos a função `takeUntil`? Vamos lá!

Vamos alterar `app/app.js` e importar a função `takeUntil` para em seguida criar uma simples função que será chamada várias vezes:

```
// app/app.js
import { log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil } from './utils/operators.js';
```

```
// código temporário de teste
const showMessage = () => console.log('Opa!');

// recebe a função que encapsula showMessage
const operation = takeUntil(3, showMessage);

// exibirá 10 mensagem, não é o que queremos!
let counter = 10;
while(counter--) operation();

// código posterior omitido
```

São exibidas 10 mensagens, com certeza não é o que queremos. Vamos concluir a implementação da função `takeUntil`:

```
// app/utils/operators.js

export const takeUntil = (times, fn) => {
    return () => {
        if(times-- > 0) fn();
    };
};
```

A função retornada lembrará dos parâmetros recebidos pela função `takeUntil` que a retornou. Essa memória ocorre devido ao suporte de closure na linguagem JavaScript. Em outras palavras, a função retornada lembra do contexto no qual foi declarada. Toda vez que ela for executada, verificaremos se `times--`. Como usamos um pós-decremento, primeiro verificamos seu valor para depois decrementá-la. Vale lembrar que qualquer número exceto zero é avaliado `true`. Enquanto `times` for maior que zero, chamaremos a função. Todavia, podemos simplificar ainda mais nosso código, removendo o bloco mais externo, pois só temos uma única instrução no bloco de `takeUntil`.

```
export const takeUntil = (times, fn) =>
    () => {
        if(times-- > 0) fn();
    };
```

Podemos otimizar ainda mais, removendo o bloco da função retornada:

```
// ERRO!
export const takeUntil = (times, fn) =>
    () => if(times-- > 0) fn();
```

Só podemos remover o bloco se a *arrow function* tiver uma instrução apenas e no caso ela possui duas, a condição `if` e a chamada `fn`. Mas um pequeno ajuste resolverá essa questão:

```
export const takeUntil = (times, fn) => () => times-- > 0 && fn();
```

Como estamos usando uma condição `&&`, a segunda condição, a chamada de `fn` só será executada de `times` for `true`. Mais enxuto, não?

Agora, ao recarregarmos nossa página veremos que nosso teste exibirá apenas três mensagens no console. Agora, podemos remover o código de teste que adicionamos para utilizar a função `takeUntil` no problema ele ela deve resolver.

```
import { log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil } from './utils/operators.js';

const operation1 = takeUntil(3, () =>
    service
    .sumItems('2143')
    .then(console.log)
    .catch(console.log)
);

document
.querySelector('#myButton')
.onclick = () => operation1();
```

Perfeito! Podemos clicar quantas vezes desejarmos que a busca e processamento das notas fiscais só será realizado no máximo três vezes.

Todavia, podemos simplificar ligeiramente nosso código. Vejamos a linha:

```
document
.querySelector('#myButton')
.onclick = () => operation1();
```

Nela, associamos à propriedade `onclick` uma função anônima declarada com arrow function que ao ser chamada pelo clique do botão chamará `operation1`. Ela poderia ser escrita assim:

```
document
.querySelector('#myButton')
.onclick = event => operation1();
```

Todavia, não temos interesse em trabalhar com o parâmetro event, aquele que é passado automaticamente pelo navegador para a função de callback passado para o evento quando ele é disparado. Nesse sentido, podemos atribuir diretamente `operation1`:

```
// modificando o código!
document
.querySelector('#myButton')
.onclick = operation1;
```

Não se preocupe com o nome `operation1`. Eu deixei esse nome para que possamos ver com clareza cada operação que criarmos inclusive a combinação que faremos entre elas. Aliás, chegou a hora de implementarmos mais uma operação.