03

Limitando operações - debounceTime

Transcrição

Desta vez precisaremos filtrar a quantidade de cliques do usuário dentro de uma janela de tempo. Por exemplo, se ele clicar diversas vezes no botão dentro da janela de tempo de meio segundo nós ignoraremos todos os cliques. Apenas o clique que ultrapassar meio segundo sem que um novo clique seja realizado disparará a ação do botão. Em suma, queremos aplicar o *pattern* **Debounce**. Nossa solução também deve ser reutilizável.

Dentro de `app/utils/operators./js` vamos criar a função `debounceTime`:

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

export const debounceTime = (milliseconds, fn) => {

    return () => {

    };
};
```

Nossa função terá uma estrutura semelhante à função `takeUntil` que criamos no vídeo anterior. o primeiro parâmetro é o tempo em milissegundos no qual só pode haver uma chamada de função. Já o segundo parâmetro é a função que respeitará a janela de tempo definida.

Sabemos que podemos postergar a execução de uma função através da função `setTimeout`. Vamos utilizá-la:

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

export const debounceTime = (milliseconds, fn) => {
    return () => {
        setTimeout(fn, milliseconds);
    };
};
```

Excelente, a função retornada por `debounceTime` executará nossa função postergando sua execução na quantidade de milissegundos que definimos como parâmetro. Nossa função ainda não esta pronta, porém vamos testá-lo em um exemplo de escopo menor.

Vamos alterar o módulo `app/app.js`. Nele, importaremos `debounceTime` e logo em seguida faremos um teste:

```
// 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, debounceTime } from './utils/operators.js';
```

```
const showMessage = () => console.log('Opa!');
const operation2 = debounceTime(500, showMessage);
operation2();
operation2();
operation2();
// código posterior omitido
```

No exemplo acima, como estamos executando na sequência três chamadas da função `operation` em milésimos de segundo, sendo assim, pelo o que nos foi exigido, apenas a última operação deverá ser processada, porém todas são executadas.

Nosso código posterga em meio segundo a execução de cada função. Precisamos que assim que a função for chamada novamente, ela cancele o timer anterior agendando um novo. Dessa forma, podemos executar 1000 vezes a mesma função que apenas a última será executada quando o tempo de meio segundo expirar.

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

export const debounceTime = (milliseconds, fn) => {
    let timer = 0;
    return () => {
        clearTimeout(timer);
        timer = setTimeout(fn, milliseconds);
    };
};
```

Sempre que a função retornada por `debounceTime` for chamada, ela cancelará o agendamento da chamada da função `fn` através de `clearTimeout(timer)`. A variável timer começa de 0, isto é, um timer que não existe, mas que não dará erro algum ao ser chamado por `clearTimeout`. Depois de cancelarmos, atribuímos um novo valor à `timer` através do retorno de `setTimeout`. Mais uma vez a closure vem nos ajudar, porque a função retornada por `debounceTime` lembrará do estado desta variável, atualizando-a sempre que necessário.

Agora nosso teste funciona como esperado, exibindo apenas o resultado da chamada da função. Agora, vamos remover nosso teste e utilizar nossa recém criada função:

```
// 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, debounceTime } from './utils/operators.js';

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

const operation2 = debounceTime(500, operation1);

document
.querySelector('#myButton')
.onclick = operation2;
```

Se clicarmos freneticamente no botão e pararmos durante meio segundo, a função `takeUntil` será chamada e executará nossa operação.

Podemos ainda organizar nosso código desta maneira:

```
// 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, debounceTime } from './utils/operators.js';

const action = debounceTime(500, takeUntil(3, () =>
    service
    .sumItems('2143')
    .then(console.log)
    .catch(console.log)
));

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

Excelente!