



TABLELESS

[HTML](#) [JAVASCRIPT](#) [CSS](#) [BACK-END](#) [SOFT SKILL](#) [EXTRAS](#) [MAIS CATEGORIAS](#)

## ENTENDENDO PROGRAMAÇÃO FUNCIONAL EM JAVASCRIPT DE UMA VEZ

Você já percebeu que cada vez mais o termo Programação Funcional vem sendo usado pela comunidade?

por [Matheus Lima](#) 14/09/2017 **2 comentários**

~ 6 min. / 1154 palavras

No meu último *post*, por exemplo: [O que TODO desenvolvedor JavaScript precisa saber](#), um dos pontos que gerou mais dúvida foi justamente o da Programação Funcional.

Continue lendo esse artigo para aprender:

1. Quais as vantagens de usar a Programação Funcional;
2. Como usá-la tanto em ES5 quanto em ES6;

3. Como usar Pure Functions e Higher Order Functions

99

Shares

4. Qual a diferença entre *Map*, *Filter* e *Reduce*;

5. O que é *Currying*;

6. Como compor funções de maneira eficaz;

Para entender as verdadeiras motivações, temos obrigatoriamente que voltar aos conceitos básicos.

A função abaixo possui *inputs* e *outputs* bem definidos:

1	<code>function square(x) {</code>
2	<code>    return x * x;</code>
3	<code>}</code>
4	
5	<code>square(2); // 4</code>

square.js hosted with ❤ by GitHub [view raw](#)

Ela recebe como parâmetro uma variável *x* e retorna um *int* que é a multiplicação de *x* com ele mesmo.

A função abaixo porém, não possui *inputs* e *outputs* tão bem definidos:

1	<code>function generateDate() {</code>
2	<code>    var date = new Date();</code>
3	<code>    generate(date);</code>
4	<code>}</code>
5	
6	<code>generateDate(); // ???</code>

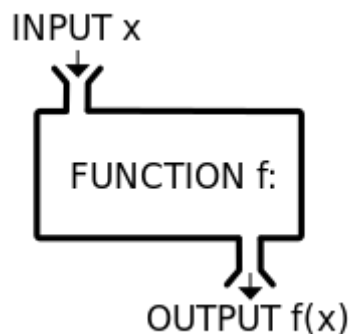
99  
Shares

Ela não recebe nada como parâmetro e retorna o que parece ser uma data processada mas não temos como ter certeza.

Um ponto que vale ser reforçado: só porque não declaramos explicitamente os *inputs* e *outputs* dessa função não quer dizer que a mesma não os tenha. Eles apenas estão ocultos. E isso pode gerar um dos piores problemas nas aplicações modernas: os Efeitos Colaterais (*side-effects*).

Funções como as de cima que possuem *inputs* e *outputs* ocultos e podem gerar *side-effects* são chamadas de Funções Impuras (*impure functions*). Outra característica importante delas é que se invocarmos uma função impura diversas vezes, o retorno dela nem sempre será o mesmo. O que dificulta a manutenção e os testes na sua aplicação.

Funções Puras (*pure functions*) por outro lado, como o primeiro exemplo desse *post*, tem *inputs* e *outputs* declarados e não geram *side-effects*. Além disso, o retorno de uma função pura dado um parâmetro será sempre o mesmo. Obviamente os seus testes serão mais fáceis de desenvolver, assim como a manutenção da sua aplicação.



Porque escrever funções puras e remover *side-effects* é a base da Programação Funcional.

---

Agora que temos uma ideia melhor da motivação de se usar Programação Funcional, podemos começar a ver os casos reais e aprender na prática como usá-la.

## 1) Higher-Order Functions

Matematicamente falando, funções que operam sobre outras funções ou as recebendo como parâmetro ou as retornando são chamadas de [Higher-Order Functions](#).

Essas funções nos permitem fazer abstrações não apenas de valores mas também de ações, como no exemplo abaixo:

1	<code>var calculate = function(fn, x, y) {</code>
2	<code>  return fn(x, y);</code>
3	<code>};</code>

calculate.js hosted with ❤ by GitHub [view raw](#)

A função **calculate** recebe três parâmetros. O primeiro é uma função qualquer que será invocada passando como parâmetro **x** e **y**.

Pensando em um cenário em que precisamos tanto de uma soma quanto de uma multiplicação, podemos pensar na solução dessa forma em FS5:

99  
Shares

```
1  var sum = function(x, y) {
2      return x + y;
3  };
4
5  var mult = function(x, y) {
6      return x * y;
7  };
8
9  calculate(sum, 2, 5); // 7
10 calculate(mult, 2, 5); // 10
```

calculateES5.js hosted with ❤ by GitHub

[view raw](#)

Ou dessa, bem mais curta, em ES6:

```
1  const sum = (x, y) => x + y;
2  const mult = (x, y) => x * y;
3
4  calculate(sum, 2, 5); // 7
5  calculate(mult, 2, 5); // 10
```

calculateES6.js hosted with ❤ by GitHub

[view raw](#)

*Higher-order functions* estão em todos os lugares no ecossistema do JavaScript. Se você já usou testes unitários com [Jasmine](#) ou [Mocha](#), então o trecho abaixo deve ser familiar:

```
1  describe("A suite", function() {
2      it("contains spec with an expectation", function() {
3          expect(true).toBe(true);
4      });
5  });
```

99

Shares

```
5 });
```

jasmine.js hosted with ❤ by GitHub

[view raw](#)

Percebemos que o segundo parâmetro tanto de **describe** quanto de **it** são funções. Por definição ambas são *higher-order functions*.

Outro exemplo também é o bom e velho **jQuery**. Podemos perceber que praticamente todo o código gerado por ele era composto de *higher-order functions*, como o exemplo abaixo:

```
1 $(btn1).click(function() {
2     doSomething();
3 });
4
5 $(btn2).bind("click", function() {
6     doSomethingElse();
7 });
8
9 $(document).ajaxStart(function() {
10     $(log).text("Triggered ajaxStart handler.");
11 });
```

jquery.js hosted with ❤ by GitHub

[view raw](#)

Em aplicações desenvolvidas com **AngularJS** também não é diferente, observe atentamente a definição de um *controller*.

```
1 var app = angular.module('app');
```

99

Shares

```
4  });
```

angular.js hosted with ❤ by GitHub

[view raw](#)

Agora que já temos conhecimento dos fundamentos: *pure functions* e *higher-order functions* podemos nos aprofundar um pouco mais...

## 2) Map

A função **map** invoca um *callback* e retorna um novo *array* com o resultado desse *callback* aplicado em cada item do *array* inicial.

Imaginando um cenário em que temos um *array* de inteiros e precisamos do quadrado de cada valor desse *array*, podemos fazer dessa forma bem simples usando a função **map** em ES5:

```
1  var numbers = [1, 2, 3];
2
3  var square = function(x) {
4      return x * x;
5  };
6
7  var squaredNumbers = numbers.map(square); // [1, 4, 9]
```

mapES5.js hosted with ❤ by GitHub

[view raw](#)

Ou assim em ES6:

99  
Shares

```
2  const square = x => x * x;  
3  const squaredNumbers = numbers.map(square); // [1, 4, 9]
```

mapES6.js hosted with ❤ by GitHub

[view raw](#)

Nesse outro cenário abaixo, percebemos o reaproveitamento de código que podemos conseguir ao usar o **map**.

Possuímos dois *arrays* de objetos diferentes, porém ambos tem o campo *name*, e precisamos de uma função que retorne um novo *array* apenas com os *names* dos objetos:

```
1  var students = [  
2    { name: 'Anna', grade: 6 },  
3    { name: 'John', grade: 4 },  
4    { name: 'Maria', grade: 9 }  
5  ];  
6  
7  var teachers = [  
8    { name: 'Mark', salary: 2500 },  
9    { name: 'Todd', salary: 3700 },  
10   { name: 'Angela', salary: 1900 }  
11  ];  
12  
13  var byName = function(object) {  
14    return object.name;  
15  };  
16  
17  var byNames = function(list) {  
18    return list.map(byName);  
19  };
```

99

Shares



```
21 byNames(students); // ["Anna", "John", "Maria"]
22 byNames(teachers); // ["Mark", "Todd", "Angela"]
```

map2ES5.js hosted with ❤ by GitHub

[view raw](#)

Em ES6:

```
1  const students = [
2    { name: 'Anna', grade: 6 },
3    { name: 'John', grade: 4 },
4    { name: 'Maria', grade: 9 }
5  ];
6
7  const teachers = [
8    { name: 'Mark', salary: 2500 },
9    { name: 'Todd', salary: 3700 },
10   { name: 'Angela', salary: 1900 }
11 ];
12
13 const byName = object => object.name;
14 const byNames = list => list.map(byName);
15
16 byNames(students); // ["Anna", "John", "Maria"]
17 byNames(teachers); // ["Mark", "Todd", "Angela"]
```

map2ES6.js hosted with ❤ by GitHub

[view raw](#)

Podemos melhorar ainda mais esse trecho de código alterando as funções *byName* e *byNames* para que o atributo *name* não esteja mais tão acoplado. Podemos simplesmente receber como parâmetro qualquer atributo

99

Shares

### 3) Filter

A função **filter** é bem semelhante ao *map*: ela também recebe um *callback* como parâmetro e também retorna um novo *array*, a única diferença é que **filter**, como o próprio nome diz, retorna um filtro dos elementos do *array* inicial baseado na função de *callback*.

Imaginando que temos um *array* de inteiros e desejamos retornar apenas aqueles que são maiores do que 4. Podemos resolver assim usando o **filter** com ES5:

```
1 var numbers = [1, 4, 7, 10];
2
3 var isBiggerThanFour = function(value) {
4     return value > 4;
5 };
6
7 var numbersBiggerThanFour = numbers.filter(isBiggerThanFour); // [7, 10]
```

filterES5.js hosted with ❤ by GitHub [view raw](#)

Ou com ES6:

```
1 const numbers = [1, 4, 7, 10];
2 const isBiggerThanFour = value => value > 4;
3
4 const numbersBiggerThanFour = numbers.filter(isBiggerThanFour); // [7, 10]
```

filterES6.js hosted with ❤ by GitHub [view raw](#)

99  
Shares

Outro exercício é uma melhoria na função **isBiggerThanFour**, deveríamos alterá-la para receber como parâmetro qualquer inteiro que desejamos fazer a comparação.

## 4) Reduce

Uma das funções que mais gera dúvidas é o **reduce**. Ele recebe como parâmetro um *callback* e um valor inicial, com o objetivo de reduzir o array a um único valor. O cenário mais comum para explicar o **reduce** é uma soma:

```
1 var numbers = [1, 2, 3];
2
3 var sum = function(x, y) {
4     return x + y;
5 };
6
7 var numbersSum = numbers.reduce(sum, 0); // 6
```

reduceES5.js hosted with ❤ by GitHub [view raw](#)

Com ES6 seria assim:

```
1 const numbers = [1, 2, 3];
2 const sum = (x, y) => x + y;
3 const numbersSum = numbers.reduce(sum, 0); // 6
```

reduceES6.js hosted with ❤ by GitHub [view raw](#)


```
1 const numbers = [1, 2, 3];
2 const sum = (x, y) => x + y;
3 const numbersSum = numbers.reduce(sum, 10); // 16
```

reduce2ES6.js hosted with  by GitHub [view raw](#)

Mas o **reduce** não serve apenas para somas, podemos também trabalhar com *strings*. Imaginando que nós temos um *array* de meses e precisamos retornar o meses dessa forma: **JAN/FEV/MAR ... / DEZ**.

Podemos fazer assim:

```
1 var months = ['JAN', 'FEV', 'MAR', /*...*/ , 'DEZ'];
2
3 var monthsShortener = function(previous, current) {
4     return previous + '/' + current;
5 };
6
7 var shortenedMonths = months.reduce(monthsShortener, '');
8 // /JAN/FEV/MAR ... /DEZ
```

reduceMonths.js hosted with  by GitHub [view raw](#)

Não era bem o que a gente queria inicialmente.

Nós queríamos isso: **JAN/FEV/MAR ... / DEZ** Mas obtivemos isso: **\*\* /JAN/FEV/MAR ... / DEZ\*\***

Devemos alterar nossa função *monthsShortener* para adicionar uma condição que faça a prevenção desse erro:

```
1 var months = ['JAN', 'FEV', 'MAR', /*...*/ , 'DEZ'];
2
```

99  
Shares

```
5     return current;
6   } else {
7     return previous + '/' + current;
8   }
9 };
10
11 var shortenedMonths = months.reduce(monthsShortener, '');
12 // JAN/FEV/MAR ... /DEZ
```

reduceMonths2.js hosted with ❤ by GitHub

[view raw](#)

Feito! E também na versão em ES6:

```
1  const months = ['JAN', 'FEV', 'MAR', /*...*/ , 'DEZ'];
2
3  const monthsShortener = (previous, current) => {
4    if (previous === '') {
5      return current;
6    } else {
7      return previous + '/' + current;
8    }
9  };
10
11 const shortenedMonths = months.reduce(monthsShortener, '');
12 // JAN/FEV/MAR ... /DEZ
```

reduceMonthsES6.js hosted with ❤ by GitHub

[view raw](#)

99  
Shares

A técnica de transformar uma função com múltiplos parâmetros em uma sequência de funções que aceitam apenas um parâmetro é chamada de Currying.

Se na teoria ficou confuso, na prática seria transformar isso:

1	<code>var add = function(x, y) {</code>
2	<code>    return x + y;</code>
3	<code>};</code>
4	
5	<code>add(1, 2) // 3</code>

noCurrying.js hosted with ❤ by GitHub [view raw](#)

Nisso:

1	<code>var add = function(x) {</code>
2	<code>    return function(y) {</code>
3	<code>        return x + y;</code>
4	<code>    };</code>
5	<code>};</code>
6	
7	<code>add(1)(2); // 3</code>

currying.js hosted with ❤ by GitHub [view raw](#)

A princípio parece que estamos apenas adicionando mais dificuldade sem nenhum ganho. Porém temos uma grande vantagem: transformar 0 código em pequenos pedaços mais expressivos e com maior reuso.

99  
Shares

Pensando em uma aplicação que possui diversos trechos do código uma soma com 5 e outra com 10, podemos usar a segunda versão da função *add* dessa forma:

```
1 var addFive = add(5);
2 var addTen = add(10);
3
4 addFive(3); // 8
5 addFive(1); // 6
6
7 addTen(1); // 11
8 addTen(10); //20
```

curryingES5.js hosted with ❤ by GitHub [view raw](#)

Mais um exemplo seria um *Hello World* simples com uma *curried function*. Podemos implementá-lo desse jeito com ES5:

```
1 var greeting = function(greet) {
2     return function(name) {
3         return greet + ' ' + name;
4     };
5 };
6
7 var hello = greeting('Hello');
8 hello('World'); // Hello World
9 hello('Matheus'); // Hello Matheus
```

currying2ES5.js hosted with ❤ by GitHub [view raw](#)

99

Shares

```
1  const greeting = greet => name => greet + ' ' + name;
2  const hello = greeting('Hello');
3
4  hello('World'); // Hello World
5  hello('Matheus'); // Hello Matheus
```

currying2ES6.js hosted with  by GitHub[view raw](#)

## 6) Compose

Podemos compor funções pequenas para gerar outras mais complexas de forma bem fácil em JavaScript. A vantagem é o poder de usar essas funções mais complexas, de forma simples, em toda aplicação. Ou seja, aumentamos o reuso.

Por exemplo, em uma aplicação em que necessitamos de uma função para transformar uma *string* passada pelo usuário em um grito: mudar para caracteres maiúsculos e adicionar uma exclamação no final. Podemos fazer assim em ES5:

```
1  var compose = function(f, g) {
2      return function(x) {
3          return f(g(x));
4      };
5  };
6
7  var toUpperCase = function(x) {
8      return x.toUpperCase();
9  };
```

99

Shares



```
11 var exclaim = function(x) {  
12     return x + '!';  
13 };  
14  
15 var angry = compose(toUpperCase, exclaim);  
16  
17 angry('ahhh'); // AHHH!
```

composeES5.js hosted with ❤ by GitHub

[view raw](#)

Ou em ES6:

```
1 const compose = (f, g) => x => f(g(x));  
2  
3 const toUpperCase = x => x.toUpperCase();  
4 const exclaim = x => x + '!';  
5  
6 const angry = compose(toUpperCase, exclaim);  
7  
8 angry('ahhh'); // AHHH!
```

composeES6.js hosted with ❤ by GitHub

[view raw](#)

Esse artigo foi escrito originalmente no [canal do Tableless no Medium!](#)

## Leia mais aqui no Tableless:

99  
Shares

- [O porquê do Bitcoin – Parte 2: A mitologia grega, o Napster e os Refugiados da Síria](#)
- [Template Strings](#)
- [O porquê do Bitcoin – Parte 1: La casa de papel, a Roma Antiga e a Década Perdida](#)
- [Algoritmos de ordenação e o seu JavaScript](#)

2 Comentários

Tableless

 Entrar ▾ Recomendar 5 Tweet Compartilhar

Ordenar por Mais recentes ▾




Participe da discussão...

FAZER LOGIN COM

OU REGISTRE-SE NO DISQUS **Rafael Gaspar** • 2 anos atrás

Sensacional o conteúdo, muito obrigado.

  • Responder • Compartilhar ›**André Nunes** • 2 anos atrásÓtimo post, simples e objetivo. Consegui esclarecer algumas dúvidas que tinha sobre estes conceitos.  
Parabéns!!  • Responder • Compartilhar ›

99

Shares

## Desenvolvimento web em camadas com API - Artigos sobre HTML, JavaScript, CSS e ...

2 comentários • 10 meses atrás



**Luciano Braga** — Olha, vou fazer uma critica construtiva. Seu artigo é mto superficial. Falando sobre o desenvolvimento web em camadas eu

## React Hooks - Artigos sobre HTML, JavaScript, CSS e desenvolvimento web

1 comentário • 6 meses atrás



**Paulo Griettner** — Infelizmente não consegui acompanhar o artigo, pois nenhum dos exemplos mencionados, apareceu para mim.

## Guia fácil para usar localStorage com Javascript - Tableless - Website com ...

1 comentário • 3 meses atrás



**Guilherme Costa** — Muito Bom

## Um pensamento sobre trabalho remoto - Artigos sobre HTML, JavaScript, CSS e ...

5 comentários • 6 meses atrás



**Mestre dos Magos** — "Produzir melhor e com qualidade é preferível do que tempo de trabalho.", obviamente que melhor qualidade é preferível a

Inscreva-se Adicione o Disqus no seu site [Adicionar Disqus](#) [Adicionar](#)

## Você vai gostar de ler:

O que TODO desenvolvedor JavaScript precisa saber

O porquê do Bitcoin — Parte 2: A mitologia grega, o Napster e os Refugiados da Síria

Template Strings

O porquê do Bitcoin — Parte 1: La casa de papel, a Roma Antiga e a Década Perdida

Algoritmos de ordenação e o seu

99

Shares

## Categorias

tecnologia-e-tendências (314)

geral (297)

artigos (290)

técnicas-e-práticas (269)

javascript (263)

código (212)

browsers (206)

html (190)

css (183)

mercado (121)



*Este site é apoiado por:*

alura

 organizze

 School of Net

## SOBRE

SOBRE O TABLELESS

CONTATO

ANUNCIE

SEJA UM AUTOR

FAZEMOS CÓDIGO FRONT-END

## ACOMPANHE

WEBINARS

FÓRUM

99  
Shares

CANAL NO TELEGRAM

## COMUNIDADE

UDACITY | CURSO DE REACT

FEMUG

MEETUPCSS

PODCAST ZOFE

BRAZILJS

DEVNAESTRADA

FRONT-END BRASIL

Escrito pela e para a comunidade web brasileira. [Ajude.](#)

[Change privacy settings](#)