



Decolando no JavaScript



Prepare-se para o React.
Aprenda **ES6, ES7 & ES8+**.

Introdução

Aprenda tudo o que você precisa saber sobre JavaScript pra começar os seus estudos em um framework front-end, mais especificamente, no React. Aprenderemos funcionalidades introduzidas nas suas especificações mais modernas (ES6, ES7, ES8+), portanto, o conteúdo que você lerá nas próximas páginas está 100% atualizado e coerente com os dias atuais.

Começamos por aqui...

Antes de qualquer coisa, eu quero agradecer você, leitor, por você ter depositado a sua confiança em mim e por ter feito o download deste material. Pode ter certeza de que você não vai se arrepender, pois eu dediquei muitas horas do meu tempo pra preparar o melhor conteúdo possível pra você.



Sobre o Autor

Me chamo Felipe Rocha, tenho 18 anos e programo desde os meus 11. Sou apaixonado por tecnologia e sou simplesmente fascinado pelo poder que a programação me dá. Com ela eu consigo colocar as minhas ideias em prática – convertê-las em um produto real e eventualmente, em um negócio – e criar produtos que podem impactar a vida das pessoas.

Acredito que a propagação de conhecimento é um dos mecanismos mais transformadores da nossa sociedade, e é por isso que amo fazer o que estou fazendo hoje, neste material, e o que faço diariamente nas redes sociais – compartilhar o meu conhecimento e ajudar as pessoas com ele. Hoje, eu vou ajudar você.

Eu (também) passei por dificuldades...

Antes de qualquer coisa, eu quero dizer para você que no início eu passei por muitas dificuldades, principalmente para aprender o tema que vou ensinar para você neste material. Portanto, não se desespere caso você fique com dificuldades para entender algum assunto – isso é completamente normal. Apenas foque em não desistir, e continuar estudando mesmo quando a desmotivação bater na porta. Para ajudar você nessa, vou deixar o meu Instagram e o meu canal no YouTube aqui embaixo, para caso você precise de ajuda.

Bora lá?

**Agora que você já me conhece melhor,
vamos começar a nossa jornada.
Vamos para o conteúdo!**

O que vamos abordar

- **Constantes e Variáveis de Escopo (const & let)**
- **Const vs. let**
- **Template Strings**
- **Arrow Functions**
- **Destructuring**
- **Promises**

Constantes e Variáveis de Escopo (const & let)

Antes do **ES6**, só havia uma maneira de declarar uma variável no JavaScript: utilizando a keyword **var**.

E o grande calcanhar de aquiles dessas variáveis declaradas com o **var** era que elas **não possuíam block scope**. Para entender o que isso quer dizer, dê uma olhada no exemplo abaixo:



```
if (5 > 4) {  
    var message = "hello world!";  
}  
  
console.log(message); // -> hello world!
```

Esse comportamento é no mínimo estranho, concorda? Declaramos a variável **message** dentro do **bloco de código do if** e, mesmo assim, conseguimos acessá-la **fora dele**. Como você pode imaginar, esse não é o ideal na maioria das situações, afinal, se declaramos uma variável dentro de um **if**, provavelmente não queremos que ela fique acessível fora dele. Esse tipo de situação acontece por conta da ausência do **block scoping**.

E as keywords **const** e **let** vieram justamente para resolver esse problema. Elas possuem **block scoping** e, portanto, só são acessíveis dentro do bloco onde foram declaradas.

Replicando o mesmo exemplo anterior, mas agora usando **const**, teremos o seguinte resultado:

```
● ● ●  
if (5 > 4) {  
    const message = "hello world!";  
}  
  
console.log(message); // -> ReferenceError: message is not defined
```

Agora faz sentido! Declaramos a variável **message** dentro de um bloco **if**, portanto, ela só é acessível dentro dele. Sensacional!

const vs. let

Tanto `const` quanto `let` possuem **block scoping**. A diferença entre as duas é que as variáveis declaradas com `const` não podem ser reatribuídas, já as com `let`, podem.

```
● ● ●  
const message = "hello world!";  
message = "hello!";  
// -> TypeError: Assignment to constant variable.  
  
let name = 'John Doe';  
name = 'Doe John';  
console.log(name) // -> Doe John
```



Beleza, mas qual devo usar?

Opte sempre por utilizar `const` e/ou `let`. As variáveis declaradas com `var`, por não possuírem **block** scoping, podem causar muita confusão no código, principalmente em aplicações maiores.

Template Strings



No **ES6**, foram introduzidas as **template strings**. Elas proporcionam uma forma extremamente mais limpa de adicionar expressões em strings. Dê uma olhada neste exemplo:

```
● ● ●  
const apple = "apple";  
  
// Sem Template Literals  
console.log("i love ` , apple, ` !");  
  
// Com Template Literals  
console.log(`i love ${apple}!`);
```

Viu como é mais fácil? E para usá-las, basta criar a string com **acento grave (`)** e colocar as expressões em volta de um **cifrão** e **duas chaves (\${})**.

Arrow Functions



As **arrow functions** foram introduzidas no **ES6**. Elas possuem uma sintaxe mais curta e simples quando comparadas às funções tradicionais:

```
// Funções convencionais
function printName(name) {
    console.log(`Hello, ${name}!`)
}

// Arrow Functions
const printName = (name) => {
    console.log(`Hello, ${name}!`)
}
```

Também conseguimos dar um **return implícito** quando a escrevemos da seguinte forma:



```
function sum(a, b) {  
    return a + b;  
}  
  
const sum = (a, b) => a + b;
```

Ambas as funções acima geram o mesmo resultado: elas retornam a **soma** da variável **a** com a **b**. Sensacional!

Beleza... a sintaxe é excelente, mas o que realmente torna uma **arrow function** tão vantajosa é o seu **this**. Antes de eu explicá-lo para você, vamos entender como ele funciona em uma função convencional.

Entendendo o `this` em uma função convencional

O `this` em uma função é definido no lugar em que ela é chamada. Por exemplo:

```
var name = "Doe John"

class Person {
    constructor(name) {
        this.name = name;
    }

    printNameFunction() {
        setTimeout(function() {
            console.log(`Function: ${this.name}`)
        }, 100)
    }
}

const person = new Person('John Doe');
person.printNameFunction() // -> Function: Doe John
```

Podemos ver que, ao invés de considerar o `this` do lugar onde foi criada — dentro do método `printNameFunction`, cujo o `this` é o da classe `Person` — ela considerou o do lugar onde foi chamada. Portanto, o `this.name` é igual a `Doe John`, como definimos na primeira linha do código, e não `John Doe`.

Esse comportamento costuma causar muitos problemas e pode ser, muitas vezes, confuso para o desenvolvedor. Em outras linguagens de programação, como Java e C#, o `this.name` da função `printNameFunction` seria igual a `John Doe` — o que é o esperado.

Entendendo o `this` em uma Arrow Function

O `this` em uma **arrow function**, diferentemente do `this` em uma função convencional, é definido no **lugar onde ela foi criada**, e não onde foi executada. Por exemplo:

```
var name = "Doe John";

class Person {
    constructor(name) {
        this.name = name;
    }

    printNameArrow() {
        setTimeout(() => {
            console.log(`Arrow: ${this.name}`);
        }, 100);
    }
}

const person = new Person("John Doe");
person.printNameArrow(); // -> Arrow: John Doe
```

Isso é, simplesmente, sensacional. Conseguimos ter um controle muito maior do valor do `this` da nossa função. Nas **arrow functions**, ele se **comporta como deveria** — é herdado do contexto onde a função foi criada.

Beleza, mas qual devo usar?

Para responder essa pergunta, eu vou mostrar para você um caso onde as **arrow functions** podem não ser a melhor opção: **na criação de métodos de objetos**.

Dê uma olhada nesse exemplo:

```
const person = {  
  name: "John Doe",  
  sings: () => {  
    console.log(`#${this.name} is singing!`);  
  },  
};  
  
person.sings(); // -> undefined
```

A propriedade `sings` executa uma **arrow function**. Nela, o `this.name` é **`undefined`**, porque, como já aprendemos, as **arrow functions** herdam o `this` do **contexto onde foram criadas**. Neste exemplo, o contexto em que foram criadas é o mesmo ao qual a variável `person` pertence — o contexto global. No navegador, por exemplo, ele seria o `window`. Se definíssemos o `name` fora do objeto — no contexto global —, por exemplo, teríamos este resultado:

```
● ● ●

var name = "Doe John";

const person = {
    name: "John Doe",
    sings: () => {
        console.log(` ${this.name} is singing!`);
    },
};

person.sings(); // -> Doe John is singing!
```

Agora, o `this.name` é igual a **Doe John** porque o definimos no contexto onde a **arrow function** foi criada — neste caso, no global.

Portanto, em situações como essa, pode não ser uma boa ideia utilizar as **arrow functions**. Mas, nas demais, ela provavelmente será a melhor opção.

The background of the image is a dark, slightly blurred photograph of a laptop's keyboard and trackpad area. The lighting is low, creating a moody atmosphere.

Destru- turing

Destructuring, também conhecido como **desestruturação**, é uma forma extremamente simples de acessar as propriedades de um objeto e/ou valores de uma lista, que foi introduzida no **ES6**.

Utilizando o destructuring em listas (arrays)

Imagine que temos uma **lista de números**, e que queremos criar três variáveis para cada um dos três primeiros elementos dessa lista. Como faríamos isso?



```
const numbers = [1, 2, 3];

const first = numbers[0];
const second = numbers[1];
const third = numbers[2];

console.log(first, second, third) // -> 1, 2, 3
```

Não parece um código muito legal, concorda? Imagina se adicionássemos mais cinco números na lista e que quiséssemos também guardar cada um deles em uma variável... teríamos que criar mais cinco linhas de código, o que não é o ideal.

Mas, uma boa notícia! O **destructuring** consegue resolver essa situação de uma forma extremamente mais limpa:

```
● ● ●  
const [first, second, third] = numbers;  
console.log(first, second, third) // -> 1, 2, 3
```

Sensacional, concorda? Nosso código ficou muito mais enxuto e, na minha opinião, mais legível também.

Além disso, conseguimos interagir com o **resto de listas**, e usá-los até mesmo na atribuição de variáveis. Se isso soou confuso, fica tranquilo(a) hahaha! Dê uma olhada nos exemplos:

```
● ● ●  
const [a, b, ...rest] = [10, 20, 30, 40, 50];  
console.log(a); // -> 10  
console.log(b); // -> 20  
console.log(rest); // -> [30, 40, 50]
```

Pegamos o **10** e o **20** e os colocamos nas variáveis **a** e **b**, respectivamente. Depois, atribuímos o **resto** da lista a variável **rest**. Quando quisermos pegar o restante de uma lista, utilizamos os **três pontos (...)**, como fizemos acima.

Podemos fazer mais algumas mágicas com esses **três pontinhos** hahaha! Dá uma olhada:

```
const numbers = [10, 20, 30];
const moreNumbers = [...numbers, 40, 50];

console.log(numbers); // -> [10, 20, 30]
console.log(moreNumbers); // -> [10, 20, 30, 40, 50]
```

Sensa- cional!

E pra fechar, também conseguimos inverter os valores de variáveis:



```
let a = 1;
let b = 3;

[a, b] = [b, a];

console.log(a); // -> 3
console.log(b); // -> 1
```

Utilizando o destructuring em objetos

Imagine que temos um objeto `user`, com algumas propriedades, como `nome`, `idade` e `email`, e que queremos acessar todas essas propriedades. Como faríamos isso?

```
const user = {  
    name: "John Doe",  
    age: 21,  
    email: "john@doe.com",  
};  
  
console.log(user.name); // -> John Doe  
console.log(user.age); // -> 21  
console.log(user.email); // -> john@doe.com
```

Mais uma vez, isso não parece ser uma forma muito ideal de acessá-las, concorda?

Imagine que tivéssemos mais dez propriedades dentro do `user` e que quiséssemos acessar todas elas. Escrever um "`user.`" pra cada uma delas deixará o nosso código bem repetitivo.

Vejamos como podemos fazer isso — de uma forma extremamente melhor — utilizando **destructuring**:

```
const user = {  
    name: "John Doe",  
    age: 21,  
    email: "john@doe.com",  
};  
  
const { name, age, email } = user;  
console.log(name); // -> John Doe  
console.log(age); // -> 21  
console.log(email); // -> john@doe.com
```

Simplesmente sensacional! Retiramos aqueles redundantes "**user.**" do nosso código, e agora conseguimos acessar as propriedades do **user** mais facilmente.

Mas calma, não é só isso! Vamos supor que eu queira, por algum motivo, mudar o nome da variável que guarda o **name** do **user**, mas não alterar o nome da propriedade em si. Posso fazer isso da seguinte forma:



```
const user = {  
    name: "John Doe",  
    age: 21,  
    email: "john@doe.com",  
};  
  
const { name: fullName } = user;  
console.log(fullName); // -> John Doe
```

Agora, posso me referir ao `user.name` por meio da variável `fullName`. Simplesmente incrível!

E, claro, também conseguimos, assim como nas listas, interagir com o `resto` de um objeto:



```
const { a, b, ...rest } = { a: 10, b: 20, c: 30, d: 40 };  
  
console.log(a); // -> 10  
console.log(b); // -> 20  
console.log(rest); // -> { c: 30, d: 40 }
```

Agora, posso me referir ao `user.name` por meio da variável `fullName`. Simplesmente incrível!

E, claro, também conseguimos, assim como nas listas, interagir com o resto de um objeto:

```
const { a = 10, b = 5 } = { a: 3 };

console.log(a); // -> 3
console.log(b); // -> 5
```

Fazendo isso, conseguimos nos prevenir de acessarmos uma propriedade inexistente em um objeto. No exemplo acima, o `b` não existe no objeto no qual estamos utilizando **destructuring**, mas, mesmo assim, conseguimos atribuir um valor para ele. Mas é importante ressaltar que o valor original não será alterado; ele continuará sendo `{ a: 3 }`.

E por último, mas não menos importante, podemos também usar o **destructuring** em **propriedades nestadas**:

```
● ● ●  
  
const user = {  
  name: {  
    firstName: "John",  
    lastName: "Doe",  
  },  
};  
  
const { name: firstName, lastName } = user;  
  
console.log(firstName); // -> John  
console.log(lastName); // -> Doe
```

Simplesmente Sensacional!!!

Promises

Antes de entender o que e para que servem as **Promises**, precisamos compreender o funcionamento do JavaScript como um todo.

O JavaScript só executa uma coisa de cada vez

Quando executamos uma função, ela é enviada para um lugar chamado **call stack**, o qual consegue executar apenas uma por vez. Vamos exemplificar:



```
1 const printName = (name) => {
2     console.log(name);
3 }
4
5 const printAge = (age) => {
6     console.log(age);
7 }
8
9 printName("John Doe");
10 printAge("21");
```

Vamos por partes:

Quando executamos uma função, ela é enviada para um lugar chamado call stack, o qual consegue executar apenas uma por vez. Vamos exemplificar:

1. Quando rodarmos esse código, a primeira coisa que o JavaScript vai executar, obviamente, é a **linha 1**. Portanto, ele assinalará uma função à variável **printName**.
2. Depois, na **linha 5**, ele vai assinalar uma função à variável **printAge**.
3. E, finalmente, na linha 9, ele vai **executar** a função **printName**. Essa função será enviada para o **call stack**, que executará ela normalmente e logará "John Doe" no console. Após isso, o **call stack estará vazio**.
4. Depois, na linha **10**, vamos **executar** a função **printAge**, que será enviada para o **call stack** — assim como foi a **printName** — e será executada. "21" será logado no console.

No fluxo acima podemos facilmente visualizar que o JavaScript **não executa mais de uma função por vez**. O **call stack** recebe uma, a executa e, só após executá-la, ele recebe a próxima.

Mas e quando uma função demora muito para ser executada?

Vamos supor que enviamos para o **call stack** uma função que **faz uma requisição para uma API**. Imagine que essa API esteja muito lenta, e que leve **10 segundos** para retornar algo para nós. Consequentemente, essa função **levará 10 segundos para completar sua execução**. O que acontecerá com o **call stack**? Ele simplesmente ficará **congelado** até que essa função termine de ser executada, ou seja, ele ficará parado por **10 segundos**. Relembrando: ele só executa **uma coisa de cada vez**.

Mas, uma boa notícia: as **Promises** vieram resolver justamente este tipo de problema! Imagine o seguinte código:

```
● ● ●

const printName = ( name ) => {
  setTimeout( () => {
    console.log( name );
  }, 5000 );
};

const printAge = ( age ) => {
  console.log( age );
};

printName("John Doe");
printAge( "21" );
```

Ele é bem similar ao que vimos anteriormente, mas com uma diferença: a função `printName` vai levar **5 segundos para ser executada**, porque colocamos um `setTimeout` dentro dela.

O que vai acontecer, neste caso? Bom, o **call stack** executará a função `printName` e, consequentemente, ficará **parado** por **5 segundos**. A nossa aplicação ficará totalmente congelada e a `printAge` só será executada após esse período. Porque, mais uma vez, o **call stack só executa uma coisa de cada vez**.



```
printName('John Doe');
printAge('21');

// -> John Doe
// 5 segundos depois...
// -> 21
```

Resolvendo isso com Promises

Com as **Promises**, podemos enviar uma função para o **call stack** que **não o congelará**, mesmo que ela leve, como no exemplo acima, 5 segundos para completar a sua execução.

Vamos reproduzir o mesmo exemplo anterior, mas agora utilizando-as:

```
● ● ●  
1 const returnName = (name) => {  
2     return new Promise((resolve, reject) => {  
3         try {  
4             setTimeout(() => {  
5                 resolve(name);  
6             }, 5000);  
7         } catch (error) {  
8             reject(error);  
9         }  
10    });  
11};  
12  
13 const printAge = (age) => {  
14     console.log(age);  
15};  
16  
17 returnName("John Doe").then((name) => console.log(name));  
18 printAge("21");
```

Não se desespere! Vamos juntos entender como funciona a **sintaxe** de uma **Promise**:

A primeira coisa que fazemos é **retornar uma nova Promise**, como fizemos na **linha 2**. Ela recebe dois parâmetros: **resolve** e **reject**. O **resolve** é chamado quando tudo ocorre bem, e nenhum erro é apresentado. Já o **reject** é executado quando algo de errado acontece.

Agora, quando executamos a função **returnName**, na **linha 17**, ela "fala" o seguinte para o **call stack**: olha, me deixe de lado por enquanto, e siga executando as próximas funções que estão na fila (neste caso, a **printAge**) que eu **prometo** que, daqui a pouco, eu vou te retornar alguma coisa. E a mágica acontece aqui, pois, mesmo que a **returnName** demore **5 segundos** para completar a sua execução, o **call stack** pegará a próxima função (**printName**) e a executará normalmente, **sem esperar por estes 5 segundos**.

E **então**, por meio do **.then**, podemos passar uma função que será executada **após a Promise ter sido resolvida (por meio do resolve)**. Ela recebe como parâmetro o valor que foi passado no **resolve**, na **linha 5**. Neste caso, "John Doe" será logado no console.

```
● ● ●

returnName("John Doe").then((name) => console.log(name));
printAge("21");

// -> 21
// 5 segundos depois...
// -> John Doe
```

Perceba que o "21" foi logado antes do "John Doe", mesmo com a função `returnName` tendo sido chamada **antes** da `printAge`. Isso é uma demonstração perfeita das **Promises** em ação!

Obs.: podemos passar uma função para o `.catch` da mesma maneira que passamos para o `.then`. Ela será executada quando a **Promise** for **rejeitada (por meio do reject)**, e receberá como parâmetro o valor que passarmos para o `reject`.

Promises com `async/await`

Ao invés de usarmos a sintaxe que vimos acima para lidar com as **Promises** (`.then` e `.catch`) podemos utilizar uma muito melhor e mais limpa, que foi introduzida no **ES2017**: o **async await**. Vamos vê-la na prática em um exemplo real: vamos chamar uma API utilizando-a!

Mas antes, vamos fazer este processo de chamada de um API com a sintaxe que já conhecemos, `.then` e `.catch`.

```
1 function fetchUsers() {
2     fetch("https://jsonplaceholder.typicode.com/users")
3         .then((response) => response.json())
4         .then((jsonResponse) => console.log(jsonResponse));
5 }
6
7 fetchUsers();
```

Neste exemplo, estamos utilizando uma função nativa do JavaScript, chamada **fetch**. Com ela, podemos fazer requisições para APIs. Neste caso, fizemos para uma que retornará uma lista de usuários.

O **fetch** retorna uma **Promise**, com a **response (resposta)** da requisição em seu **resolve** e o **erro** (caso ocorra) em seu **reject**. Na linha 3, lidamos justamente com ela, convertendo-a para JSON utilizando o método **.json**, que também é nativo do JavaScript. Este método retorna uma **Promise**, com o valor no qual o invocamos convertido para JSON em seu **resolve**. Na linha 4, recebemos este valor convertido e o logamos no console.

O resultado será o seguinte:

Dê um ZOOM para enxergar melhor

```
▼ (10) [{} , {} , {} , {} , {} , {} , {} , {} , {} , {}] 1
▶ 0: {id: 1, name: "Leanne Graham", username: "Bret", email: "Sincere@april.biz", address: {}, ...}
▶ 1: {id: 2, name: "Ervin Howell", username: "Antonette", email: "Shanna@melissa.tv", address: {}, ...}
▶ 2: {id: 3, name: "Clementine Bauch", username: "Samantha", email: "Nathan@yesenia.net", address: {}, ...}
▶ 3: {id: 4, name: "Patricia Lebsack", username: "Karianne", email: "Julianne.OConner@kory.org", address: {}, ...}
▶ 4: {id: 5, name: "Chelsey Dietrich", username: "Kamren", email: "Lucio_Hettinger@annie.ca", address: {}, ...}
▶ 5: {id: 6, name: "Mrs. Dennis Schulist", username: "Leopoldo_Corkery", email: "Karley_Dach@jasper.info", address: {}, ...}
▶ 6: {id: 7, name: "Kurtis Weissnat", username: "Elwyn.Skiles", email: "Telly.Hoeger@billy.biz", address: {}, ...}
▶ 7: {id: 8, name: "Nicholas Runolfsdottir V", username: "Maxime_Nienow", email: "Sherwood@rosamond.me", address: {}, ...}
▶ 8: {id: 9, name: "Glenna Reichert", username: "Delphine", email: "Chaim_McDermott@dana.io", address: {}, ...}
▶ 9: {id: 10, name: "Clementina DuBuque", username: "Moriah.Stanton", email: "Rey.Padberg@karina.biz", address: {}, ...}
```

Funcionou que é uma beleza! Mas esse código pode melhorar bastante se usarmos o **async/await**. Portanto, vamos convertê-lo:

```
1 async function fetchUsers() {
2     const response = await fetch("https://jsonplaceholder.typicode.com/users");
3     const jsonResponse = await response.json();
4
5     console.log(jsonResponse);
6 }
7
8 fetchUsers();
```

A primeira coisa que fizemos, na linha 1, foi adicionar a keyword **async** ao início da nossa função, marcando-a, assim, como assíncrona. Precisamos fazer isso pois **só podemos usar o `async/await` em funções assíncronas.**

Agora, a mágica começa na linha 2. Veja que, ao invés de usar **.then** para acessar a **response** do **fetch (que é uma Promise)**, simplesmente adicionamos **await** antes dele. Agora, o valor do seu **resolve** será guardado na variável **response**.

Na linha 3, vamos converter a **response** para JSON — como fizemos anteriormente. Para isso, vamos utilizar, novamente, a keyword **await** antes de chamar o método **.json (que é uma Promise)** e, mais uma vez, o valor do seu **resolve (valor convertido para JSON)** será armazenado na variável **jsonResponse**, que será, na linha 5, logada no console:

Dê um ZOOM para enxergar melhor

```
▼ (10) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}] ⓘ
▶ 0: {id: 1, name: "Leanne Graham", username: "Bret", email: "Sincere@april.biz", address: {...}, ...}
▶ 1: {id: 2, name: "Ervin Howell", username: "Antonette", email: "Shanna@melissa.tv", address: {...}, ...}
▶ 2: {id: 3, name: "Clementine Bauch", username: "Samantha", email: "Nathan@yesenia.net", address: {...}, ...}
▶ 3: {id: 4, name: "Patricia Lebsack", username: "Karianne", email: "Julianne.OConner@kory.org", address: {...}, ...}
▶ 4: {id: 5, name: "Chelsey Dietrich", username: "Kamren", email: "Lucio_Hettinger@annie.ca", address: {...}, ...}
▶ 5: {id: 6, name: "Mrs. Dennis Schulist", username: "Leopoldo_Corkery", email: "Karley_Dach@jasper.info", address: {...}, ...}
▶ 6: {id: 7, name: "Kurtis Weissnat", username: "Elwyn.Skiles", email: "Telly.Hoeger@billy.biz", address: {...}, ...}
▶ 7: {id: 8, name: "Nicholas Runolfsdottir V", username: "Maxime_Nienow", email: "Sherwood@rosamond.me", address: {...}, ...}
▶ 8: {id: 9, name: "Glenna Reichert", username: "Delphine", email: "Chaim_McDermott@dana.io", address: {...}, ...}
▶ 9: {id: 10, name: "Clementina DuBuque", username: "Moriah.Stanton", email: "Rey.Padberg@karina.biz", address: {...}, ...}
```

Ou seja, com o **async/await** conseguimos armazenar o **resolve** de uma **Promise** dentro de uma variável, e eliminar a utilização do **.then!** Simplesmente sensacional, não acha?

Beleza, mas e o .catch?

Em caso de erro em uma **Promise**, podemos lidar com o seu **reject** por meio do **.catch**. Beleza! Disso você já sabe muito bem. Mas, no **async/await**, podemos lidar com ele de uma forma mais limpa. Simplesmente colocamos todas as nossas chamadas de **Promises (await)** dentro de um bloco **try, catch**:

```
● ● ●  
1 async function fetchUsers() {  
2   try {  
3     const response = await fetch("https://jsonplaceholder.typicode.com/users");  
4     const jsonResponse = await response.json();  
5  
6     console.log(jsonResponse);  
7   } catch (error) {  
8     console.log(`Error: ${error}`);  
9   }  
10 }  
11  
12 fetchUsers();
```

Agora, se alguma das **Promises (fetch ou .json)** falharem, o nosso bloco **catch** será executado. Portanto, o erro (**reject**) será logado no console.

Bônus: map & filter

Como uma forma de recompensar você, leitor(a), por ter ficado até aqui, vou lhe apresentar dois métodos importantíssimos, e que são muito utilizados em frameworks front-end, especialmente no React: o **map** e o **filter**.

Ambos são úteis na hora de trabalhar com **manipulação de listas (arrays)**. Vamos começar pelo **map**, e depois vamos para o **filter**.

Map

A função **map** executa uma função em cada elemento de um array e retorna um novo array com estes elementos, tenha sido eles modificados ou não por ela. Vamos vê-lo na prática:

```
● ● ●  
const numbers = [1, 2, 3];  
  
const numbersMultipliedBy2 = numbers.map((number) => number * 2);  
  
console.log(numbersMultipliedBy2); // -> [2, 4, 6]
```

Veja que executamos a função "**(number) number *2**" para cada elemento do array **numbers**, ou seja, **multiplicamos cada um por 2**. O resultado final foi "**[2, 4, 6]**". Vale lembrar que o que será adicionado no array gerado pelo map é o que é retornado na função que passamos para ele. No exemplo acima, retornamos "**number *2**".

Vamos para mais um exemplo, mas agora, utilizando o **map** em um **array de objetos**:

```
const users = [
  {
    name: "John",
    age: 20,
  },
  {
    name: "Doe",
    age: 40,
  },
];

const usersWithAgeMultipliedBy2 = users.map((user) => {
  return { ...user, age: user.age * 2 };
});

console.log(usersWithAgeMultipliedBy2);
// -> [{name: "John", age: 40}, {name: "Doe", age: 80}]
```

Veja que para cada elemento do array **users**, retornamos um objeto com o elemento (user) e com a sua idade (age) multiplicada por 2.

Filter

Assim como o **map**, o **filter** executa uma função para cada elemento de um array e retorna um novo, mas a principal diferença dele é a seguinte: no array gerado por ele, só são adicionados os itens no qual a função recebida por ele retornou **true**.

Isso pareceu bem confuso, não é? Mas vamos vê-lo na prática, que fica mais fácil:



```
const numbers = [1, 2, 3, 4];

const evenNumbers = numbers.filter((number) => number / 2 === 0);

console.log(evenNumbers); // -> [2, 4];
```

Para cada elemento do array **numbers**, verificamos se o resto da sua divisão por 2 é 0 (**number / 2 === 0**), ou seja, se ele é par. Se for, **true** será retornado e, portanto, ele será adicionado ao novo array gerado pelo filter. Se não, **false** será retornando, e ele não será adicionado.

No final, o array gerado pelo filter, o **evenNumbers**, terá apenas o 2 e 4, pois apenas eles, dentre todos os elementos do array **numbers**, são pares.

Vamos para mais um exemplo, mas agora, utilizando o **filter** em um **array de objetos**:

```
const users = [
  {
    name: "John",
    age: 20,
  },
  {
    name: "Doe",
    age: 40,
  },
];

const usersWithAge20 = users.filter((user) => user.age === 20);

console.log(usersWithAgeMultipliedBy2);
// -> [{name: 'John', age: 20}]
```

Para cada elemento do array `users`, verificamos se ele tem a idade (`age`) igual a 20. Se tiver, `true` será retornando, e ele será adicionado ao novo array gerado pelo filter, `usersWithAge20`. Se não, `false` será retornando, e ele não será adicionado.

Minha dica de ouro para você fixar este conteúdo

Sei que o que você viu aqui pode parecer assustador – embora eu tenha dado o meu melhor para tornar o mais simples para você – mas fique tranquilo, isso é totalmente normal. Como falei para você no início, eu já me senti assim várias vezes.

A dica de ouro que eu dou para você aprender de verdade tudo que você viu aqui é: **foque em entender um tema de cada vez, e pratique-o no processo.** Não tente focar em todos de uma só vez. Dê **pequenos passos**.

Se você precisar de ajuda, estou aqui para você. Basta me mandar uma mensagem no nosso Instagram, @dicasparadevs, beleza? :)



Fim da Linha

Chegamos ao final, pelo menos por aqui, porque agora é com você. Gostaria de parabenizá-lo novamente por ter escolhido este material e por ter lido-o até aqui, você deu um grande passo e estou orgulhoso de você.

Escrevi cada linha deste e-book com muito carinho e, especialmente, priorizando você e sua aprendizagem. Espero que tudo tenha ficado claro e que você tenha gostado do que leu.

E por último, mas não menos importante, quero dizer para você que você não está sozinho nessa. **Temos uma comunidade incrível em nosso Instagram, @dicasparadevs, onde há muitos(as) programadores(as) incríveis que estão buscando conhecimento, assim como você.** Se você tiver alguma dúvida, como já citei acima, você pode ficar à vontade para me mandar uma mensagem lá!

Obrigado!

JS

•••
**Loops de
repetição do
JavaScript**

