Artigos > **Front-end**

Async/await no JavaScript: o que é e quando usar a programação assíncrona?

Juliana Amoasei
Atualizado em 25/07/2022

O que é o Async/await?

No dia-a-dia do desenvolvimento web, utilizamos muito (e cada vez mais) dados externos - por exemplo, recebidos através de um *endpoint* de uma API REST (um microserviço) ou resultados de algum outro processamento. Ou seja, quando isso ocorre o sistema tem que esperar os dados "chegarem" antes de utilizar esse resultado.

Costumamos chamar de **programação assíncrona** o ato de executar uma tarefa em "segundo plano", sem nosso controle direto disso. Sem explicitamente trabalhar com threads e coordená-las. Escrevendo basicamente da forma tradicional que temos. Porém, é importante frisar o comportamento do JavaScript de "executar uma coisa por vez". Com isso em mente o assíncrono no JavaScript vai separar seu código em duas partes: coisas que rodam agora, coisas que vão rodar depois de algo acontecer... Calma. Vai ficar mais fácil de entender.

Trabalhando com front-end, vemos que uma boa parte do que ocorre no âmbito do navegador é *event-driven*. Ou seja, o código aguarda algum evento acontecer (por exemplo, o usuário clicar em um botão) antes de executar qualquer código. Outros exemplos de eventos, além de clique do mouse, são toque na tela, determinada tecla ser pressionada, o cursor do mouse passar em cima de algum elemento, etc). Mas, para além destas interações do usuário com a interface, há muitas outras situações que podem ser síncronas ou assíncronas.

Para exemplificar, podemos pensar em comunicação. Uma ligação telefônica é um exemplo de comunicação síncrona: quando falamos ao telefone, as informações chegam e saem em sequência, uma após a outra; fazemos uma pergunta, recebemos logo em seguida a resposta, com os dados dessa resposta fazemos outro comentário, etc.

Por outro lado, uma conversa online via algum mensageiro, como o WhatsApp ou ou Telegram, é um exemplo de comunicação assíncrona: enviamos uma mensagem e não ficamos olhando para a tela, esperando, até a outra pessoa responder (ou pelo menos não deveríamos!). Afinal de contas, não temos como saber quando, e se, essa resposta vai chegar. Mandamos a mensagem e vamos fazer outras coisas enquanto a resposta não chega, ao contrário do telefone.

Com o código, o processo é parecido: um código síncrono é aquele de ocorre em sequência, uma instrução após a outra.

```
function soma(num1, num2) {
 return num1 + num2;
}

console.log(soma(2, 2)) // 4
```

Até aí, tudo normal. O JavaScript executou uma linha após a outra.

Mas o que acontece quando, por exemplo, nosso código precisa receber alguns dados de uma API? Ao mesmo tempo que é preciso aguardar a requisição e resposta da API, não podemos bloquear o funcionamento de todo o nosso programa; seria a mesma coisa que

enviar uma mensagem pelo WhatsApp e ficar esperando a resposta sem fazer mais nada nesse meio tempo.

É para esse tipo de situação, que requer **processamento assíncrono** que existem as *Promises*, ou, literalmente, promessas. O sentido de *Promise* em JavaScript é similar ao literal: Uma pessoa te passa o contato do Telegram e pede para que você mande uma mensagem pra ela, *prometendo que vai responder*... O que não temos como saber se vai acontecer.

Quando enviamos uma requisição de dados a uma API, temos uma promessa de que estes dados irão chegar, mas enquanto isso não acontece, o sistema deve continuar rodando. Se, por exemplo, o servidor estiver caído, essa promessa de dados pode não se cumprir, e temos que lidar com isso. As *Promises* trabalham neste contexto - elas são a ferramenta que o JavaScript utiliza para lidar com código assíncrono.

Existem algumas formas de se trabalhar com processamento assíncrono (ou seja, *Promises*) em JavaScript: utilizando o método `.then()`, as palavras-chave `async` e `await` ou o objeto `Promise` e seus métodos. Aqui, vamos focar no uso de `.then()`, `async/await` e no uso do método `Promise.all`.

O que é o Promises com `.then()` e como utilizar?

Já que estávamos falando sobre APIs REST, vamos ver um exemplo usando a Fetch API do JavaScript para buscar dados e convertê-los para o formato JSON. Esta API (que funciona nativamente nos navegadores atuais e acabou de ser integrada ao Node.js na versão 18) tem alguns métodos internos e já retorna por padrão uma *Promise* que vai resolver a requisição, tendo ou não sucesso.

```
function getUser(userId) {
 const userData = fetch(`https://api.com/api/user/${userId}`)
   .then(response => response.json())
   .then(data => console.log(data.name))
}

getUser(1); // "Nome Sobrenome"
```

Destrinchando o código acima: a função `getUser()` recebe um id de usuário como parâmetro, para que seja passado para o *endpoint* REST fictício. A função `fetch()` recebe como parâmetro o *endpoint* e retorna uma *Promise*.

E como funcionam as Promises?

Promises têm um método chamado `.then()`, que recebe uma função callback e retorna um "objeto-promessa". **Não é um retorno dos dados, é a *promessa* do retorno destes dados**.

Assim, podemos escrever o código do que irá acontecer em seguida, com os dados recebidos pela Promise, e o JavaScript vai aguardar a *resolução* da Promise sem pausar o fluxo do programa.

O resultado pode ou não estar pronto ainda, e não há forma de pegar o valor de uma Promise de modo síncrono; Só é possível requisitar à Promise que execute uma função quando o resultado estiver disponível - seja ele o que foi solicitado (os dados da API, por exemplo), ou uma mensagem de erro caso algo tenha dado errado com a requisição (o servidor pode estar fora do ar, por exemplo).

No exemplo acima: ao iniciarmos uma cadeia de promessas - no caso, para fazer uma requisição HTTP - enquanto a resposta está pendente ela retorna um `Promise object`. O objeto, por sua vez, define uma instância do método `.then()`. Ao invés de passar o retorno da função callback diretamente para a função inicial, ela é passada para `.then()`. Quando o resultado da requisição HTTP chega, o corpo da requisição é convertido para JSON e este valor convertido é passado para o próximo método `.then()`.

A cadeia de funções `fetch().then().then()` não significa que há múltiplas funções callbacks sendo usadas com o mesmo objeto de resposta, e sim que cada instância de `.then()` retorna, por sua vez, um `new Promise()`. Toda a cadeia é lida de forma síncrona na primeira execução, e em seguida executada de forma assíncrona.

Capturando erros com Promises

O manejo de erros é importante em operações assíncronas. Quando o código é síncrono, ele pode lançar pelo menos uma exceção mesmo sem nenhum tipo de tratamento de erros. Porém, no assíncrono, exceções não tratadas muitas vezes passam sem aviso e fica muito mais difícil de debugar.

Não há como utilizar o `try/catch` quando usamos o .then(), pois a computação só será efetuada após o retorno do objeto-Promise. Então devemos passar funções que executem as alternativas, para o caso de sucesso ou falha da operação. Por exemplo:

```
function getUser(userId) {
 const userData = fetch(`https://api.com/api/user/${userId}`)
   .then(response => response.json())
   .then(data => console.log(data.name))
   .catch(error => console.log(error))
```

```
    .finally(() => /*{ aviso de fim de carregamento }*/)
}

getUser(1); // "Nome Sobrenome"
```

Além do método `.then()` que recebe o objeto-Promise para ser resolvido, o método `.catch()` retorna no caso de rejeição da Promise. Além disso, o último método, `.finally()` , é chamado independente de sucesso ou falha da promessa e a função callback deste método é sempre executada por último. Esta função pode ser usada, por exemplo, para fechar uma conexão ou dar algum aviso de fim de carregamento.

Resolvendo várias promessas

No caso de várias promessas que devem ser resolvidas pelo programa (por exemplo, alguns dados em *endpoints* REST diferentes), pode-se utilizar `Promise.all` :

```
const endpoints = [
 "https://api.com/api/user/1",
 "https://api.com/api/user/2",
 "https://api.com/api/user/3",
 "https://api.com/api/user/4"
]

const promises = endpoints.map(url => fetch(url).then(res => res.json()))

Promise.all(promises)
 .then(body => console.log(body.name))
```

No exemplo acima, um array de *endpoints* (`endpoints`) é percorrido com `.map` e as promessas resultantes do `fetch` passadas para a variável `promises` em um novo array. Todo este array de promessas será percorrido e resolvido por `Promise.all` - no exemplo acima a função callback apenas passa para `console.log` a propriedade fictícia `name` .

> *Uma Promise podem estar "pendente" (pending) ou "resolvida" (settled). Os estados possíveis, uma vez que uma Promise está settled, são "concluída"*

(fulfilled) ou "rejeitada" (rejected). Após passar de pending para settled e se definir como fulfilled ou rejected, a Promise não muda mais de estado.

Como usar o `async/await`?

As palavras-chave `async` e `await`, implementadas a partir do ES2017, são uma sintaxe que simplifica a programação assíncrona, facilitando o fluxo de escrita e leitura do código; assim é possível escrever código que funciona de forma assíncrona, porém é lido e estruturado de forma síncrona. O `async/await` também trabalha com o código assíncrono baseado em Promises, porém esconde as promessas para que a leitura e a escrita seja mais fluídas.

Definindo uma função como `async`, podemos utilizar a palavra-chave `await` antes de qualquer expressão que retorne uma promessa. Dessa forma, a execução da função externa (a função `async`) será pausada até que a Promise seja resolvida.

A palavra-chave `await` recebe uma Promise e a transforma em um valor de retorno (ou lança uma exceção em caso de erro). Quando utilizamos `await`, o JavaScript vai aguardar até que a Promise finalize. Se for finalizada com sucesso (o termo utilizado é *fulfilled*), o valor obtido é retornado. Se a Promise for rejeitada (o termo utilizado é *rejected*), é retornado o erro lançado pela exceção.

Um exemplo:

```
let response = await fetch(`https://api.com/api/user/${userId}`);
let userData = await response.json();
```

Só é possível usar `await` em funções declaradas com a palavra-chave `async`, então vamos adicioná-la:

```
async function getUser(userId) {
 let response = await fetch(`https://api.com/api/user/${userId}`);
 let userData = await response.json();
 return userData.name; // nas linhas de return não é necessário usar await
}
```

Uma função declarada como `async` significa que o valor de retorno da função será, "por baixo dos panos", uma Promise. Se a Promise se resolver normalmente, o objeto-Promise retornará o valor. Caso lance uma exceção, podemos usar o try/catch como estamos acostumados em programas síncronos.

Para executar a função `getUser()`, já que ela retorna uma Promise, pode-se usar `await`:

```
exibeDadosUser(await getUser(1))
```

Lembrando que `await` só funciona se estiver dentro de outra função `async`. Caso não esteja, você ainda pode usar `.then()` normalmente:

```
getUser(1).then(exibeDadosUser).catch(reject)
```

Resolvendo várias promessas

No caso de várias promessas que devem ser resolvidas para a execução do programa (por exemplo, alguns dados em *endpoints* REST diferentes), pode-se utilizar `async/await` em conjunto com `Promise.all`:

```
async function getUser(userId) {
 let response = await fetch(`https://api.com/api/user/${userId}`);
 let userData = await response.json();
 return userData;
}

let [user1, user2] = await Promise.all([getUser(1), getUser(2)])
```

Há diferenças entre `.then()` e `async/await`?

Em termos de sintaxe, o método `.then()` traz uma sintaxe que faz mais sentido quando utilizamos o JavaScript de forma funcional, enquanto o fluxo das declarações com `async/await` fazem sentido ao serem utilizadas em métodos de classes.

O `async/await` surgiu como uma opção mais "legível" ao `.then()`, e como vimos acima é possível usar os dois métodos em um mesmo código e, a partir da versão 10 do Node.js, ambas as formas são equivalentes em termos de performance. O `async/await` simplifica a escrita e a interpretação do código, mas não é tão flexível e só funciona com uma Promise por vez.

Como resolver esse tipo de caso, por exemplo, requisitando uma array de id de pedidos de determinado cliente de um comércio:

```
async function getCustomerOrders(customerId) {
 const response = await fetch(`https://api.com/api/customer/${customerId}`)
 const customer = await response.json()

 return await Promise.all(customer.orders.map(async (orderId) => {
   const response = await fetch(`https://api.com/api/order/${orderId}`)
   const orderData = await response.json()
   return orderData.amount
 }))
}
```

No caso acima, usamos `Promise.all` em conjunto com `.map` para fazer todas as requisições, resolver todas as promessas e processar todos os resultados em um único array.

Iterações com `async/await`

Mas e se precisarmos manejar várias Promises, mas não quisermos fazer isso de uma vez? Um exemplo clássico desta situação é acessar um banco de dados com milhares de registros. Neste caso, não queremos que todas as requisições sejam feitas dessa forma, pois o excesso de requisições simultâneas pode causar problemas de performance e até derrubar o serviço.

Neste caso o `async/await` é mais indicado, pois vai resolver uma Promise por vez.

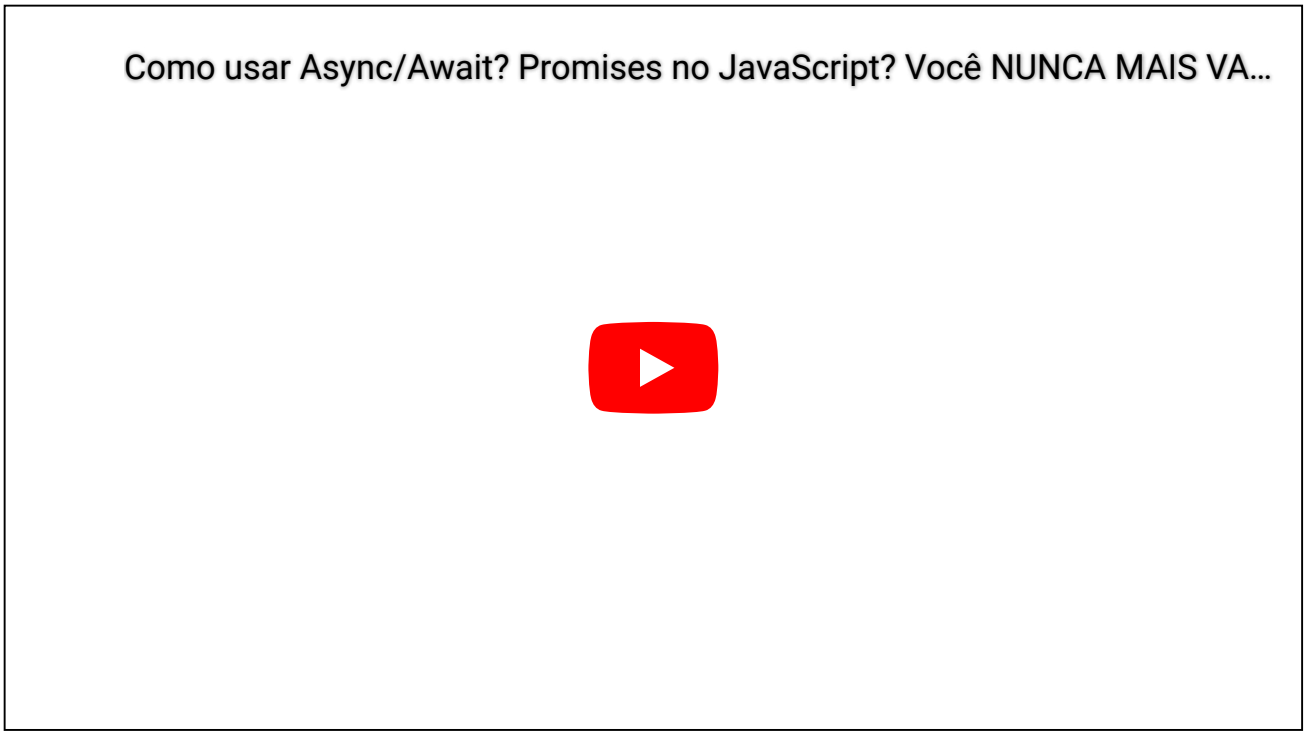
```
async function printCustomer(customerId){
 //lógica fictícia da função
}

async function getAndPrintAllCustomers() {
 const sql = 'SELECT id FROM customers'
 const customers = await db.query(sql, [])
 for (const customer of customers) {
   await printCustomer(customer.id)
 }
}
```

No caso acima, não queremos fazer todas as requisições ao banco de uma vez, e sim de forma sequencial - em cada iteração do `for`, o `await` vai resolver uma promessa por vez.

Confira um video explicando tudo isso e muito mais

Aqui o DevSoutinho explica pra gente em um video sensacional. É mais um exemplo para você praticar e testar seus conhecimentos:



E um outro vídeo com bastante callbacks: