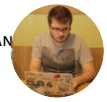
DESENVOLVIMENTO

4 MAI, 2018

# Funções assíncronas e retornos: como o Async/Await tornaram o código mais legível



LUCAS SANTOS (HTTPS://IMASTERS.COM.BR/PERFIL/LUCASSAN
Tem 26 artigos publicados com 29053 visualizações desde 2017





LUCAS SANTOS (HTTPS://IMASTERS.COM.BR/PERFIL/LUCASSANTOS)

26

Estudante de Ciências da Computação pela UFABC. Desenvolvedor desde 2011, trabalha com infraestrutura e arquitetura de aplicações de alta disponibilidade. Participante ativo de comunidades, organizador do ABCDev, membro do Core Team do Training Center e moderador no fórum iMasters.

LEIA MAIS (HTTPS://IMASTERS.COM.BR/PERFIL/LUCASSANTOS)

2 JUL, 2020
Design Patterns com JavaScript & TypeScript: Padrões Estruturais (https://imasters.com.br/javascript/design-patterns-com-javascript-typescript-padroes-estruturais)

8 ABR, 2020
O Lado Escuro do JavaScript: O retorno (https://imasters.com.br/javascript/o-lado-escuro-do-javascript-o-retorno)

9 MAR, 2020
Design Patterns com JavaScript & TypeScript (https://imasters.com.br/devsecops/design-patterns-com-javascript-typescript)

Durante os anos de atualização do ECMAScript, o JavaScript veio recebendo uma série de funcionalidades novas que facilitaram muito a vida do desenvolvedor. O suporte a Promises do ES6 (https://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects) criou um mundo completamente novo, onde poderíamos tirar proveito do assincronismo da linguagem e de todo o seu ferramental performático.

O problema com as Promises era o chamado “promise hell”, assim como o tão conhecido callback hell das versões anteriores do JavaScript, esse novo problema surge quando começamos a encadear muitas chamadas then nas nossas resoluções de Promises, por exemplo:

```
usuario.obterPerfil()
    .then((perfil) => {
        usuario.obterMidias(perfil)
            .then((midias) => {
                midias.obterDados()
                    .then((dados) => {...})
        })
    })
```

Percebe como a legibilidade fica complicada? Para resolver esse problema, o ES2017 implementou o que hoje chamamos de Async/Await, que deixou o código muito mais legível. Por exemplo, o código anterior ficaria desta forma:

```
const perfil = await usuario.obterPerfil()
const midiasPerfil = await usuario.obterMidias(perfil)
const midiasDados = await midias.obterDados()
```

Porém, agora ganhamos um novo tipo de função, as **funções assíncronas.** Em suma, uma função assíncrona contém a palavra-chave async em sua assinatura e, por padrão, retorna uma Promise. Esses retornos ficam um pouco confusos quando temos que retornar exatamente o resultado de uma Promise para a função anterior, então vamos entender um pouco como podemos utilizar o async/await junto com o return, e veja que escolher a forma certa é muito importante.

Vamos começar com uma função assíncrona simples:

```
async function espereEDecida() {
  // Vamos esperar um segundo primeiro
  await new Promise((resolve) => setTimeout(resolve, 1000))

  // Vamos fazer uma jogada aleatória, com 50% de chance para true ou false
  const sim = Boolean(Math.round(Math.random()))

  if (sim) return 'yeah!'
  throw Error('Ops!')
}
```

Basicamente, essa função dá uma resposta que pode ser uma resolução ou um erro dentro de um segundo depois de sua chamada.

A partir daí, temos quatro formas de poder chamar essa função, e as quatro se comportam de formas completamente diferentes:

### 1. Uma chamada simples

Podemos realizar somente uma chamada desse tipo:

```
async function f() {
  try {
    espereEDecida()
  } catch (err) { return 'peguei' }
}
```

Neste exemplo, se chamarmos a função f, a Promise que for retornada **sempre vai ser resolvida para undefined**, sem nenhum tipo de espera. Isso acontece porque não estamos sequer tratando o resultado retornado por espereEDecida, estamos apenas invocando a função e colocando sua chamada na fila de execuções. Uma vez que ela é executada, nós não tratamos o código de nenhuma maneira, então não fazemos nada com ela.

Esse tipo de código é muito raro de ser utilizado, e geralmente é considerado um erro, uma vez que não estamos fazendo nada com o código, mas, se o retorno desta função fosse uma Promise pura ao invés de uma string, poderíamos utilizar esse tipo de chamada para executar uma Promise interna sem nos importar com o resultado.

### 2. Somente Await

Agora podemos fazer desta forma:

```
async function f() {
  try {
    await espereEDecida()
  } catch (err) { return 'peguei' }
}
```

Se chamarmos nossa função agora, a Promise interna vai **sempre esperar um segundo** e depois vai seguir para a nossa jogada de sorte que **pode ser resolvida como undefined ou então vai ser resolvida com “peguei”**.

Isso vai acontecer porque estamos esperando o resultado de espereEDecida(). Se ela for rejeitada, temos um throw que será pego pelo nosso bloco catch na função f. No entanto, se ela for resolvida com sucesso, teremos somente undefined como retorno porque não estamos armazenando o valor da Promise em lugar nenhum.

### 3. Retornando uma Promise

A terceira forma seria algo assim:

```
async function f() {
  try {
    return espereEDecida()
  } catch (err) { return 'peguei' }
}
```

Agora, se chamarmos a função f neste caso, teremos um resultado interessante, pois nossa Promise de espereEDecida() vai **sempre esperar um segundo e ser resolvida ou rejeitada com ‘ops!’**, como seria o esperado dela, mas o interessante é que nosso bloco catch não vai nunca ser executado, pois estamos delegando o retorno dessa Promise para quem está chamando a função f, e não para ela própria.

### 4. Retornando com Await

Este é o caso em que juntamos todos os anteriores em um, ficando alguma coisa assim:

```
async function f() {
  try {
    return await espereEDecida()
  } catch (err) { return 'peguei' }
}
```

Aqui teremos algo muito parecido com o caso anterior. Se chamarmos a função f, a Promise de dentro **vai esperar um segundo e ser resolvida ou rejeitada com ‘peguei’**, veja que agora a rejeição da Promise não está em quem está invocando a mesma, mas sim dentro da própria função.

Isso acontece porque estamos esperando o resultado de espereEDecida(). Se ela for rejeitada, então essa rejeição se transformará em um throw que será pego pelo nosso catch, dando o retorno de ‘peguei’. Agora, se ela for resolvida, o retorno da Promise será ‘yeah!’, que será retornado pela função f para seu invocador.

Ficou um pouco confuso, não é? Vamos simplificar. Este passo pode ser pensado como sendo uma sequência de duas execuções:

```
async function f() {
  try {
    // Aqui esperamos o resultado da função, ela pode retornar 'Yeah!' ou 'Ops!'
    const resultado = await espereEDecida()

    // Se a função espereEDecida() rejeitar, teremos um throw
    // Ele será pego pelo nosso catch abaixo
    // Caso contrário, vamos retornar o resultado propriamente dito, que será 'Yeah!'
    return resultado
  } catch (err) { return 'peguei' }
}
```

O caso 4 tem uma peculiaridade, pois sem os blocos try/catch, um return seguido de await é completamente redundante (já até criaram uma regra do ESLint (https://github.com/eslint/eslint/blob/master/docs/rules/no-return-await.md) para estes casos). Isso porque o retorno de uma função async é automaticamente envolto em um Promise.resolve, como nesta página da MDN (https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/funcoes_assincronas). Ou seja, se fizermos algo assim:

```
async function pegarDadosProcessados(url) {
  let v;
  try {
    v = await baixarDados(url);
  } catch(e) {
    v = await baixarDadosReservas(url);
  }
  return processarDadosNoWorker(v);
}
```

E retornarmos a Promise diretamente no final da função, teremos implicitamente um Promise.resolve(processarDadosNoWorker(v)).

Funções assíncronas são bastante comuns atualmente, mas algumas dicas como estas não são muito explícitas e exigem um pouco de testes e erros para se descobrir. Portanto, acredito que saímos daqui com um pouco mais de conhecimento sobre o que as funções assíncronas fazem e como elas funcionam no JavaScript.

O Lado Escuro do JavaScript: O retorno (https://imasters.com.br/javascript/o-lado-escuro-do-javascript-o-retorno)

LUCAS SANTOS (HTTPS://IMASTERS.COM.BR/PERFIL/LUCASSANTOS)
9 MAR, 2020

Design Patterns com JavaScript & TypeScript (https://imasters.com.br/devsecops/design-patterns-com-javascript-typescript)

LUCAS SANTOS (HTTPS://IMASTERS.COM.BR/PERFIL/LUCASSANTOS)
12 FEV, 2020

Entendendo o ciclo de publicação do Node.js (https://imasters.com.br/javascript/entendendo-o-ciclo-de-publicacao-do-node-js)

LUCAS SANTOS (HTTPS://IMASTERS.COM.BR/PERFIL/LUCASSANTOS)
14 JAN, 2020

JavaScript em 2020: O que esperar (https://imasters.com.br/javascript/javascript-em-2020-o-que-esperar)

LUCAS SANTOS (HTTPS://IMASTERS.COM.BR/PERFIL/LUCASSANTOS)
13 DEZ, 2019

Optional Chaining e null coalescing operator com TypeScript (https://imasters.com.br/javascript/optional-chaining-e-null-coalescing-operator-com-typescript)



SAIBA MAIS
(HTTPS://IMASTERS.COM.BR/PERFIL/LUCASSANTOS)

Lucas Santos
(https://twitter.com/_StaticVoid) (mailto:lhs.santoss@gmail.com)
26 Artigo(s)

Estudante de Ciências da Computação pela UFABC. Desenvolvedor desde 2011, trabalha com infraestrutura e arquitetura de aplicações de alta disponibilidade. Participante ativo de comunidades, organizador do ABCDev, membro do Core Team do Training Center e moderador no fórum iMasters.