

- Back-End(<https://imasters.com.br/back-end>)
- Mobile(<https://imasters.com.br/mobile>)
- Front End(<https://imasters.com.br/front-end>)
- DevSecOps(<https://imasters.com.br/devsecops>)
- Design & UX(<https://imasters.com.br/design-ux>)
- Data(<https://imasters.com.br/data>)
- APIs e Microsserviços(<https://imasters.com.br/apis-microsservicos>)
- IoT e Makers(<https://imasters.com.br/iot-makers>)

PATROCINADORES:


FRONT END



27 JUN, 2017

Entenda tudo sobre Async/Await

100 visualizações


 (<https://www.facebook.com/sharer?u=https://imasters.com.br/front-end/entenda-tudo-sobre-asyncawait>) (<https://twitter.com/share?url=https://imasters.com.br/front-end/entenda-tudo-sobre-asyncawait>) (<https://www.linkedin.com/shareArticle?url=https://imasters.com.br/front-end/entenda-tudo-sobre-asyncawait>)

COMPARTILHE!

ROBERTO ACHAR
([HTTPS://IMASTERS.COM.BR/PERFIL/ROBERTOACHAR](https://imasters.com.br/perfil/robertoachar))
Tem 4 artigos publicados com 5618 visualizações desde 2017



PUBLICIDADE



Create Your Own Website

Start Now

ROBERTO ACHAR ([HTTPS://IMASTERS.COM.BR/PERFIL/ROBERTOACHAR](https://imasters.com.br/perfil/robertoachar))4 

É Full Stack Web Developer e fascinado pelo mundo Open Source. Gosta de escrever sobre Node.js, TypeScript, JavaScript e Angular. Nas horas vagas joga video-game, é marido e pai do Dudu.

LEIA MAIS ([HTTPS://IMASTERS.COM.BR/PERFIL/ROBERTOACHAR](https://imasters.com.br/perfil/robertoachar))

4 SET, 2017

TypeScript – Classes vs Interfaces (<https://imasters.com.br/desenvolvimento/typescript-classes-vs-interfaces>)

20 JUN, 2017

Conheça todo o poder do Console (<https://imasters.com.br/front-end/conheca-todo-o-poder-do-console>)

12 JUN, 2017

Guia para projetos Open Source (<https://imasters.com.br/desenvolvimento/guia-para-projetos-open-source>)

Async/Await é uma das novas funcionalidades do ES2017. Com ela, é possível escrever código assíncrono como se estivéssemos escrevendo código síncrono. Essa funcionalidade já está disponível a partir da versão 7.6 do Node.js.

Neste artigo, pretendo demonstrar todas as possibilidades que aprendi até agora para trabalhar com async/await. Para todos os exemplos abaixo, vou utilizar a API do Star Wars: <https://swapi.co/> (<https://swapi.co/>). A API tem endpoints para filmes, personagens, planetas, espécies, veículos e naves. Todos os exemplos abaixo estão disponíveis no meu GitHub (<https://github.com/robertoachar/async-await>).

Escrevendo uma função assíncrona com Promises

Antes de introduzir o assunto principal, vamos ver um exemplo de função assíncrona utilizando Promises.

```
1 | const fetch = require('node-fetch');
2 |
3 | function getPerson(id) {
4 |   fetch(`http://swapi.co/api/people/${id}`)
5 |     .then(response => response.json())
6 |     .then(person => console.log(person.name));
7 | }
8 |
9 | getPerson(1);
```

No código acima, a função `getPerson()` faz uma chamada na API, processa o resultado e exibe o nome do personagem. Esse é um cenário bem comum encontrado hoje em dia.

Executando o código, temos o seguinte resultado:

```
1 | $ node sample.js
2 | Luke Skywalker
```

Escrevendo uma função assíncrona com async/await

Agora iremos escrever a mesma função, mas utilizando `async/await`.

```
1 | const fetch = require('node-fetch');
2 |
3 | async function getPerson(id) {
4 |   const response = await fetch(`http://swapi.co/api/people/${id}`);
5 |   const person = await response.json();
6 |   console.log(person.name);
7 | }
8 |
9 | getPerson(1);
```

O primeiro passo é converter a declaração `function` para `async function`. Desta forma, estamos definindo que esta função será assíncrona.

```
1 | // Promise
2 | function getPerson(id) {...}
3 |
4 | // async/await
5 | async function getPerson(id) {...}
```

O próximo passo é utilizar `await` para cada processamento assíncrono dentro da função.

```
1 | // Promise
2 | fetch(`http://swapi.co/api/people/${id}`)
3 |   .then(response => response.json())
4 |   .then(person => console.log(person.name));
5 |
6 | // async/await
7 | const response = await fetch(`http://swapi.co/api/people/${id}`);
8 | const person = await response.json();
9 | console.log(person.name);
```

A própria leitura/interpretação do código fica mais fácil utilizando async/await. É como se estivéssemos programando de forma síncrona.

Executando o código, ainda temos o mesmo resultado.

```
1 | $ node sample.js
2 | Luke Skywalker
```

Utilizando async/await com Promises

Podemos combinar os dois mundos e utilizar async/await junto com Promises.

```
1 | const fetch = require('node-fetch');
2 |
3 | async function getPerson(id) {
4 |   const response = await fetch(`http://swapi.co/api/people/${id}`);
5 |   const person = await response.json();
6 |   return person;
7 | }
8 |
9 | getPerson(1)
10 |   .then(person => console.log(person.name));
```

Funções assíncronas sempre retornam Promises.

Nesse exemplo, estamos retornando o objeto person da função assíncrona e utilizando Promises para exibir o resultado no Console, pois o retorno da função é uma Promise.

Executando o código, ainda temos o mesmo resultado:

```
1 | $ node sample.js
2 | Luke Skywalker
```

No exemplo acima, estamos armazenando o retorno da chamada response.json() na variável person e retornando-a. Podemos simplificar e retornar diretamente o resultado da chamada response.json().

```
1 | async function getPerson(id) {
2 |   const response = await fetch(`http://swapi.co/api/people/${id}`);
3 |   return await response.json();
4 | }
```

Executando o código, ainda temos o mesmo resultado.

```
1 | $ node sample.js
2 | Luke Skywalker
```

Resolvendo ou rejeitando uma Promise com async/await

Para resolver uma Promise com async/await:

```
1 | async function getPerson(id) {  
2 |   return id;  
3 | }  
4 |  
5 | getPerson(1)  
6 |   .then(id => console.log(id)); // 1
```

Para rejeitar uma Promise com async/await:

```
1 | async function getPerson(id) {  
2 |   throw Error('Not found');  
3 | }  
4 |  
5 | getPerson(0)  
6 |   .catch(err => console.error(err.message)); // Not found
```

Tratamento de erros com Throw Error()

Uma maneira de tratar erros com funções assíncronas é utilizando `throw Error()`. O código abaixo irá tentar recuperar um personagem com `id = 0`.

```
1 | const fetch = require('node-fetch');  
2 |  
3 | async function getPerson(id) {  
4 |   const response = await fetch(`http://swapi.co/api/people/${id}`);  
5 |   return await response.json();  
6 | }  
7 |  
8 | // id = 0  
9 | getPerson(0)  
10 |   .then(person => console.log(person.name));
```

Executando o código, temos o seguinte resultado.

```
1 | $ node sample.js  
2 | undefined
```

Recebemos `undefined`, pois como não existe nenhum personagem com `id = 0`, a propriedade `name` não será preenchida.

Alterando a saída do console de `person.name` para `person` podemos entender o que está acontecendo.

```
1 | ...  
2 |  
3 | // id = 0  
4 | getPerson(0)  
5 |   .then(person => console.log(person));
```

Executando o código, temos o seguinte resultado:

```
1 | $ node sample.js  
2 | { detail: 'Not found' }
```

Agora ficou mais claro o motivo de termos recebido `undefined` anteriormente. O objeto `person` possui apenas a propriedade `detail` com o valor “Not found”.

Nesse caso, podemos utilizar o `throw Error()` para tratar esse erro.

```
1 | const fetch = require('node-fetch');
2 |
3 | async function getPerson(id) {
4 |   const response = await fetch(`http://swapi.co/api/people/${id}`);
5 |   const body = await response.json();
6 |
7 |   if (response.status !== 200) {
8 |     throw Error(body.detail);
9 |   }
10 |
11 |   return body;
12 | }
```

Armazenamos o retorno da chamada `response.json()` na variável `body` e testamos o status do `response`. Se o status for diferente de 200 (OK), disparamos um erro com o conteúdo de `body.detail`, caso contrário, retornamos o `body`.

Na primeira execução, passaremos o valor correto. O status do `response` será igual à 200 (OK) e o `body` será retornado, exibindo o nome do personagem.

```
1 | getPerson(1)
2 |   .then(person => console.log(person.name)) //Luke Skywalker
3 |   .catch(err => console.error(err.message));
```

Na segunda execução, passaremos o valor errado. O status do `response` será diferente de 200 (OK) e um erro será disparado com o conteúdo de `body.detail`. Esse erro será capturado no `catch` e a mensagem “Not found” será exibida.

```
1 | getPerson(0)
2 |   .then(person => console.log(person.name))
3 |   .catch(err => console.error(err.message)); // Not found
```

Tratamento de erros com try/catch

Outra alternativa para tratar erros com funções assíncronas é utilizando `try/catch`:

```
1 | const fetch = require('node-fetch');
2 |
3 | async function getPerson(id) {...}
4 |
5 | async function loadPerson(id) {
6 |   try {
7 |     const person = await getPerson(id);
8 |     console.log(person.name);
9 |
10 |   } catch (err) {
11 |     console.error(err.message);
12 |   }
13 | }
14 |
15 | loadPerson(0);
16 | loadPerson(1);
```

No exemplo acima, a função `getPerson()` permanece a mesma e introduzimos uma nova função assíncrona `loadPerson()` que fará o tratamento de erro.

Executando o código, temos o seguinte resultado:

```
1 | $ node sample.js
2 | Not found
3 | Luke Skywalker
```

Utilizando async/await com function expressions

Neste exemplo, estamos atribuindo a função assíncrona para a variável `getPerson`. A chamada para a função é a mesma e o resultado também.

```
1 | const fetch = require('node-fetch');
2 |
3 | // regular function
4 | const getPerson = async function (id) {
5 |   const response = await fetch(`http://swapi.co/api/people/${id}`);
6 |   return await response.json();
7 | };
8 |
9 | getPerson(1)
10 | .then(person => console.log(person.name));
```

Executando o código, temos o seguinte resultado:

```
1 | $ node sample.js
2 | Luke Skywalker
```

Podemos utilizar arrow functions também.

```
1 | const fetch = require('node-fetch');
2 |
3 | // arrow function
4 | const getPerson = async (id) => {
5 |   const response = await fetch(`http://swapi.co/api/people/${id}`);
6 |   return await response.json();
7 | };
8 |
9 | getPerson(1)
10 | .then(person => console.log(person.name));
```

Executando o código, ainda temos o mesmo resultado:

```
1 | $ node sample.js
2 | Luke Skywalker
```

Entendendo o await

Nesse exemplo, tentaremos utilizar o `await` fora do escopo de uma função com `async`.

```
1 | const fetch = require('node-fetch');
2 |
3 | const getPerson = async (id) => {
4 |   const response = await fetch(`http://swapi.co/api/people/${id}`);
5 |   return await response.json();
6 | };
7 |
8 | const person = await getPerson(1);
9 | console.log(person.name);
```

Executando o código, temos o seguinte resultado:

```
1 | $ node sample.js
2 |
3 | const person = await getPerson(1);
4 |           ^^^^^^^
5 |
6 | SyntaxError: Unexpected identifier
```

`Await` só pode ser utilizado dentro de funções com `Async`.

O resultado é o erro acima, pois só podemos utilizar o `await` dentro do escopo de funções com `async`. No caso acima poderíamos ter utilizado `Promise` no lugar do `await`.

IIFE – Immediately-Invoked Function Expression

IIFE são funções que são invocadas imediatamente após a execução do programa. Para refrescar sua memória, podemos utilizar os seguintes exemplos:

→ Utilizando jQuery:

```
1 | $("document").ready(function () {  
2 |     // code  
3 | });
```

→ Utilizando JavaScript

```
1 | (function () {  
2 |     // code  
3 | })();
```

A IIFE de funções assíncronas é idêntica ao JavaScript, adicionando `async` na frente de `function`.

```
1 | (async function () {  
2 |     // code  
3 | })();
```

Podemos utilizar arrow functions também:

```
1 | (async () => {  
2 |     // code  
3 | })();
```

Como só podemos utilizar `await` dentro do escopo de funções com `async`, para resolver o caso acima, precisamos utilizar IIFE – Immediately-Invoked Function Expression.

```
1 | const fetch = require('node-fetch');  
2 |  
3 | const getPerson = async (id) => {  
4 |     const response = await fetch(`http://swapi.co/api/people/${id}`);  
5 |     return await response.json();  
6 | };  
7 |  
8 | // IIFE - Immediately-Invoked Function Expression  
9 | (async function () {  
10 |     const person = await getPerson(1);  
11 |     console.log(person.name);  
12 | })();
```

Executando o código, ainda temos o mesmo resultado:

```
1 | $ node sample.js  
2 | Luke Skywalker
```

Utilizando `async/await` com classes

É muito simples utilizar `async/await` com classes, basta adicionar `async` no início das funções. Neste exemplo, criamos uma classe chamada `StarWars` e definimos a função assíncrona `getPerson()`. Dentro do IIFE, instanciamos a classe `StarWars` e utilizamos o `await`.

```
1 | const fetch = require('node-fetch');
2 |
3 | // Class
4 | class StarWars {
5 |   async getPerson(id) {
6 |     const response = await fetch(`http://swapi.co/api/people/${id}`);
7 |     return await response.json();
8 |   }
9 | }
10 |
11 | // IIFE - Immediately-Invoked Function Expression
12 | (async () => {
13 |   const sw = new StarWars();
14 |   const person = await sw.getPerson(1);
15 |   console.log(person.name);
16 | })();
```

Executando o código, ainda temos o mesmo resultado.

```
1 | $ node sample.js
2 | Luke Skywalker
```

Exportando uma classe com funções assíncronas

Para os 03 próximos exemplos, vamos escrever a classe StarWars em um arquivo separado e vamos adicionar uma nova função para entender como trabalhar com múltiplas requisições.

```
1 | // star-wars.js
2 |
3 | const fetch = require('node-fetch');
4 |
5 | class StarWars {
6 |   async getPerson(id) {
7 |     const response = await fetch(`http://swapi.co/api/people/${id}`);
8 |     return await response.json();
9 |   }
10 |
11 |   async getFilm(id) {
12 |     const response = await fetch(`http://swapi.co/api/films/${id}`);
13 |     return await response.json();
14 |   }
15 | }
16 |
17 | module.exports = StarWars;
```

No código acima, criamos e exportamos a classe StarWars com duas funções assíncronas: `getPerson()` e `getFilm()`.

Aguardando múltiplas requisições em sequência

Podemos utilizar `async/await` para trabalhar com múltiplas requisições em sequência:


```
1 | const StarWars = require('./star-wars');
2 |
3 | async function loadData() {
4 |   const sw = new StarWars();
5 |
6 |   const person = await sw.getPerson(1);
7 |   const film = await sw.getFilm(1);
8 |
9 |   console.log(person.name);
10 |  console.log(film.title);
11 | }
12 |
13 | loadData();
```

Neste exemplo, `loadData()` chama a função `getPerson()` e aguarda o resultado. Após o retorno do resultado, a função chama `getFilm()` e aguarda o resultado. Só após o segundo resultado é que os dados serão exibidos. Se cada uma das funções, `getPerson()` e `getFilm()`, demorassem 01 segundo para responder, os dados seriam exibidos após 02 segundos.

Executando o código, temos o seguinte resultado:

```
1 | $ node sample.js
2 | Luke Skywalker
3 | A New Hope
```

Aguardando múltiplas requisições simultâneas

Podemos utilizar `async/await` para trabalhar com múltiplas requisições ao mesmo tempo.

```
1 | const StarWars = require('./star-wars');
2 |
3 | async function loadData() {
4 |   const sw = new StarWars();
5 |
6 |   const personPromise = sw.getPerson(1);
7 |   const filmPromise = sw.getFilm(1);
8 |
9 |   const person = await personPromise;
10 |  const film = await filmPromise;
11 |
12 |  console.log(person.name);
13 |  console.log(film.title);
14 | }
15 |
16 | loadData();
```

Neste exemplo, `loadData()` chama simultaneamente as funções `getPerson()` e `getFilm()` e armazena as chamadas respectivamente em `personPromise` e `filmPromise`. Reparem que não foi utilizado `await` nas chamadas dessas funções, ele foi utilizado apenas no retorno dos resultados para as variáveis `person` e `film`. Os dados só serão exibidos quando as duas chamadas retornarem. Se cada uma das funções, `getPerson()` e `getFilm()`, demorassem 01 segundo para responder, os dados seriam exibidos após 01 segundo.

Executando o código, ainda temos o mesmo resultado:

```
1 | $ node sample.js
2 | Luke Skywalker
3 | A New Hope
```

Aguardando múltiplas requisições simultâneas usando `Promise.all()`

Neste último exemplo, veremos uma forma mais simples e mais elegante de aguardar múltiplas requisições simultâneas.

```
1 | const StarWars = require('./star-wars');
2 |
3 | async function loadData() {
4 |   const sw = new StarWars();
5 |
6 |   const [person, film] = await Promise.all([sw.getPerson(1), sw.getFilm(1)]);
7 |
8 |   console.log(person.name);
9 |   console.log(film.title);
10 | }
11 |
12 | loadData();
```

O código acima aguarda o retorno das duas funções simultaneamente utilizando `Promise.all()`. Os resultados das funções `getPerson()` e `getFilm()` serão armazenados respectivamente em `person` e `film` utilizando destructuring, uma nova funcionalidade do ES6. Os dados só serão exibidos após o retorno das duas funções.

Executando o código, ainda temos o mesmo resultado.

```
1 | $ node sample.js
2 | Luke Skywalker
3 | A New Hope
```

Conclusão

Essa nova funcionalidade traz inúmeros benefícios quando trabalhamos com funções assíncronas. Espero ter conseguido esclarecer todas as dúvidas com relação à `async/await`. Caso conheça algum outro cenário onde possamos utilizar `async/await`, deixe nos comentários.

Vamos em frente!

“Talk is cheap. Show me the code.” – Linus Torvalds



De 0 a 10, o quanto você recomendaria este artigo para um amigo?



ARTIGOS PUBLICADOS POR ESTE AUTOR



ROBERTO ACHAR ([HTTPS://IMASTERS.COM.BR/PERFIL/ROBERTOACHAR](https://imasters.com.br/perfil/robertoachar))
4 SET, 2017



TypeScript – Classes vs Interfaces (<https://imasters.com.br/desenvolvimento/typescript-classes-vs-interfaces>)



ROBERTO ACHAR ([HTTPS://IMASTERS.COM.BR/PERFIL/ROBERTOACHAR](https://imasters.com.br/perfil/robertoachar))
20 JUN, 2017



Conheça todo o poder do Console (<https://imasters.com.br/front-end/conheca-todo-o-poder-do-console>)



ROBERTO ACHAR ([HTTPS://IMASTERS.COM.BR/PERFIL/ROBERTOACHAR](https://imasters.com.br/perfil/robertoachar))
12 JUN, 2017



Guia para projetos Open Source (<https://imasters.com.br/desenvolvimento/guia-para-projetos-open-source>)

Este projeto é mantido e patrocinado pelas empresas



(<https://www.idexo.com.br/>)



(<https://www.userede.com.br/>)



(https://www.schoolofnet.com/cursos/gratuitos?utm_source=imasters&utm_medium=patrocinador&utm_content=institucional_patrocinador)



(<https://developers.totvs.com/>)

ASSINE NOSSA Newsletter

Fique em dia com as novidades do iMasters! Assine nossa newsletter e receba conteúdos especiais curados por nossa equipe



Qual é o seu e-mail?

ASSINAR



SOBRE O IMASTERS ([HTTPS://IMASTERS.COM.BR/P/SOBRE-O-IMASTERS](https://imasters.com.br/p/sobre-o-imasters))

POLÍTICA DE PRIVACIDADE ([HTTPS://IMASTERS.COM.BR/P/POLITICA-DE-PRIVACIDADE](https://imasters.com.br/p/politica-de-privacidade))

FALE CONOSCO ([HTTPS://IMASTERS.COM.BR/FALE-CONOSCO/](https://imasters.com.br/fale-conosco/))

QUERO SER AUTOR ([HTTPS://IMASTERS.COM.BR/P/QUERO-SER-AUTOR](https://imasters.com.br/p/quero-ser-autor))

FÓRUM ([HTTPS://FORUM.IMASTERS.COM.BR/](https://forum.imasters.com.br/))

7MASTERS ([HTTPS://SETEMASTERS.IMASTERS.COM.BR/](https://setemasters.imasters.com.br/))

AGENDA ([HTTPS://IMASTERS.COM.BR/AGENDA/](https://imasters.com.br/agenda/))

IMASTERS.COM ([HTTPS://IMASTERS.COM/](https://imasters.com/))

Transformando a Área da Saúde

Com as ferramentas certas, médicos e cientistas transformarão vidas e o futuro da pesquisa

NVIDIA

SABER MAIS



SAIBA MAIS
([HTTPS://IMASTERS.COM.BR/PERFIL/ROBERTOACHAR](https://imasters.com.br/perfil/robertoachar))

Roberto Achar



(<https://www.facebook.com/showmethecode>)



(<https://twitter.com/robertoachar>)

✍ 4 Artigo(s)

É Full Stack Web Developer e fascinado pelo mundo Open Source. Gosta de video-game, é marido e pai do Dudu.

Faça o seu site hoje mesmo

Conte com Nossos Especialistas a Qualquer
Hora para Garantir Seu Sucesso Online.

HostGator - Hospedagem

1 comentário

Classificar por **Mais antigos**



Adicione um comentário...



Antonio Bicunha

Boa noite Roberto Achar, o que eu venho batendo cabeça a muito tempo é:

Eu queria um simples método síncrono que me retornasse um valor e a partir desse valor eu chamaria outro método ou não.

Todos os exemplos que vejo são executados métodos encadeados, mais nenhum usa um resultado do primeiro para chamar a segunda.

Teria como fazer um exemplo?

Eu venho do JS puro, e vejo que com typescript eu preciso escrever o triplo de código pra fazer um simples

```
var usuario = getUsuario();
```

```
var categoria = getCategoria(usuario);
```

Curtir · Responder · 1 a



Marcos Full Stack

É só validar o retorno da função, daí se for o resultado que você que precisa é só chamar outra função

Curtir · Responder · 41 sem