⚡ PROMOÇÃO! Qualquer Curso Online por **R$19,99** cada, com Certificado Digital e acesso Vitalício! ⚡

×

JavaScript Assíncrono: O Guia Completo - Parte 5 - Entendendo Async/Await

JAVASCRIPT JS ASSÍNCRONO ASYNC

16 de Março de 2020

Anthony Rafael

Neste artigo iremos falar sobre async/await. Se desejar, você pode conferir as outras postagens dessa série nestes links:

1. JavaScript Assíncrono: O Guia Completo - Parte 1 - O que é um código assíncrono?
2. JavaScript Assíncrono: O Guia Completo - Parte 2 - Callbacks
3. JavaScript Assíncrono: O Guia Completo - Parte 3 - Treinando Callbacks
4. JavaScript Assíncrono: O Guia Completo - Parte 4 - Usando Promises

Aprendemos no último artigo sobre as promises, um recurso muito útil que o JavaScript nos proporciona para códigos assíncronos. Quando as promises foram lançadas, talvez pensássemos que já estava excelente: elas nos ajudam a criar um código organizado, legível, simples de entender e de desenvolver. Talvez pensássemos que não tinha como ficar melhor. Mas tinha sim! Em junho de 2017 foi lançada uma nova versão do EcmaScript, a versão ES2017 ou ES8. Essa versão introduziu um novo conceito na linguagem JavaScript: **funções assíncronas, ou async/await**. O objetivo dessas funções é facilitar ainda mais o uso de promises. Uma função assíncrona consiste em criarmos uma função "normal", mas que ganhará como que "superpoderes", tornando-se uma função capaz de

trabalhar com código assíncrono. Como isso é possível? Podemos começar analisando o que as palavras async e await representam.

Async/Await

1 - Async. Essa palavra pode ser usada ao criar uma função convencional. Quando adicionamos esse identificador na criação desta função, nós definimos que ela será uma função assíncrona, e o melhor, retornará uma **promessa**! Quando usarmos a expressão return estaremos, na realidade, resolvendo uma promessa.

2 - Await. Essa palavra será usada com o objetivo de esperar a resolução de uma função assíncrona. Se houver uma série de funções assíncronas, a expressão await definirá que o código só terá sequência quando a função anterior for resolvida. Um detalhe muito importante: a expressão await só será aceita em uma função que já for assíncrona, ou seja, que possuir o identificador async em seu início.

Vamos desenvolver um primeiro exemplo desse conceito. Iremos criar uma função que juntará duas strings. Essa função será assíncrona. Depois, iremos chamar essa função usando o identificador await. Veja o script abaixo:

```
async function joinStrings(string1, string2) {
    return string1 + ' ' + string2;
}
async function init() {
    await joinStrings('Hcode', 'Treinamentos').then(result => {;
        console.log(result);
    });
}
init();
```

Explicando: a função **joinStrings()** irá realizar a concatenação de duas strings; ela foi criada com o identificador async, deixando claro que ela é uma função assíncrona. A função **init()** irá realizar a chamada dela, mas com o identificador await. Além disso, perceba que nós usamos o método .then() após chamar a função, como fazemos em uma promessa. Isso é possível pois a função joinStrings() de fato é uma promise, mas não precisamos criar toda a sua estrutura. Precisamos apenas adicionar a palavra async em sua criação. Demais, não é mesmo?

Esse foi um primeiro exemplo e já foi possível perceber que funções assíncronas são muito úteis. Mas talvez consigamos perceber isso ainda melhor quando as usamos para substituir uma série de promessas. Vamos ver um exemplo disso.

Nós iremos criar três funções que irão manipular uma string: 1) uma que irá converter a primeira letra da string para maiúscula; 2) uma que irá inverter o conteúdo da string e 3) uma que irá adicionar um texto a essa string. Todas essas funções serão promessas. Veja o código de criação delas:

```
function firstLetterToUpper(string) {
    return new Promise((resolve, reject) => {
        let firstLetter = string.charAt(0);
        let newString = firstLetter.toUpperCase() + string.slice(1);
        resolve(newString);
    });
}

function reverseString(string) {
```

```
    return new Promise((resolve, reject) => {
        let stringReverse = string.split('').reverse().join('');
        resolve(stringReverse);
    });
}

function addHelloToString(string) {
    return new Promise((resolve, reject) => {
        let newString = string + ' - Hello World';
        resolve(newString);
    });

}
```

Perfeito. Nossas funções foram criadas e estão bem simples de entender. Vamos agora chamá-las. Mas, vamos fazer diferente: nós iremos chamar uma após a outra, informando como parâmetro para cada uma o valor retornado pela promessa anterior. Além disso, iremos realizar um **console.log()** em cada valor retornado. O código para a requisição dessas funções é o seguinte:

```
firstLetterToUpper('hcode')
    .then(newValue => {
        console.log(newValue);
        reverseString(newValue)
            .then(reverseValue => {
                console.log(reverseValue);
                addHelloToString(reverseValue)
                    .then(helloString => {
                        console.log(helloString);
                    })
            })
    });
```

Ao executar esse código, vemos o seguinte resultado:

```
Hcode
edocH
edocH - Hello World
▶ Promise {<resolved>: undefined}
```

Muito bem, funcionou sem problemas. Contudo, observe que a chamada de várias promessas em sequência ficou um pouco difícil de ler e entender. Está parecendo um pouco aquela pirâmide de callbacks que falamos no artigo anterior. Além disso, usamos o método **.then()** várias vezes, o que torna o código um tanto repetitivo. Vamos converter esse código para funções assíncronas e ver o mesmo resultado usando esse novo conceito.

Primeiramente, as funções não irão mais retornar promessas, mas apenas ter o identificador async em seu início. Além disso, iremos substituir o método **resolve()** pelo **return**. O código das funções ficará assim:

```
async function firstLetterToUpper(string) {
    let firstLetter = string.charAt(0);
    let newString = firstLetter.toUpperCase() + string.slice(1);
```

```
    return newString;
}

async function reverseString(string) {
    let stringReverse = string.split('').reverse().join('');
    return stringReverse;
}

async function addHelloToString(string) {
    let newString = string + ' - Hello World';
    return newString;
}
```

O código já ficou bem mais enxuto. Vamos agora realizar a chamada de cada função e executar o console.log(), mas usando o await. O código ficará assim:

```
let upperValue = await firstLetterToUpper('hcode');
console.log(upperValue);

let reverseValue = await reverseString(upperValue);
console.log(reverseValue);

let helloValue = await addHelloToString(reverseValue);
console.log(helloValue);
```

Note que nós adicionamos o valor retornado de cada função em uma variável, outro recurso interessante que podemos usar quando trabalhamos com async/await. Além disso, como uma função é chamada abaixo da outra, de maneira independente, parece que o nosso código é síncrono, mais intuitivo, simples de entender, mas “por baixo dos panos” está ocorrendo um código assíncrono.

Antes de executar esse código, precisamos fazer apenas mais uma coisa: lembra que dissemos que o await só funciona em uma função assíncrona? Então, para evitar erros, vamos adicionar a chamada das funções em uma função principal, que nós podemos chamar de **callFunctions()**. O código final ficará assim:

```
async function callFunctions() {

    let upperValue = await firstLetterToUpper('hcode');
    console.log(upperValue);

    let reverseValue = await reverseString(upperValue);
    console.log(reverseValue);

    let helloValue = await addHelloToString(reverseValue);
    console.log(helloValue);

}
callFunctions();
```

Se executarmos esse código, teremos o mesmo resultado:

Excelente! Funcionou da mesma maneira, mas o nosso código ficou mais enxuto, organizado, fácil de compreender. Essas são as vantagens de usar funções assíncronas.

Tratando Erros

Sabemos que as promessas recuperam os erros com o método **.catch()**. Mas, e as funções assíncronas, como elas fazem isso? Nós temos à disposição esse mesmo método, pois as funções assíncronas retornam promessas, então podemos acessar os mesmos recursos.

Ao invés de usar o método **reject()** em uma função assíncrona, podemos simplesmente usar o código **throw new Error()**. Depois, na resposta da função, usar o método **.catch()** ou usar um bloco de **try / catch**. Temos assim duas opções para usar.

Agora que entendemos o conceito de funções assíncronas, vamos implementá-las em nosso projeto da Hcode Games.

Praticando

Iremos alterar primeiro o arquivo requests.js. Atualmente, as funções desse arquivo estão retornando promessas, e trabalhando com a API de Fetch, que também retorna uma promessa. Vamos adaptar os dois para async/await. A função **getGameByName()** ficará assim:

```
async function getGameByName(name) {
    try {

        let result = await fetch(`https://api.rawg.io/api/games?search=${name}`);

        result = await result.json();

        return result;

    } catch (error) {
        console.error(err);
    }
}
```

Vamos entender as mudanças juntos. Nós retiramos a declaração do **return new Promise()** e estamos usando um bloco de try/catch para realizar as operações na função. Além disso, perceba que usamos o await na chamada da função **fetch()**, pois ela retorna uma promessa. Além disso, usamos o await na linha abaixo dela também, pois o método **.json()** é uma promessa da mesma forma. Por fim, veja que o código ficou bem simples com essas alterações. Vamos fazer o mesmo para a função **getRelatedGamesByName()**. Segue o código dela:

```
async function getRelatedGamesByName(name) {
    try {
```

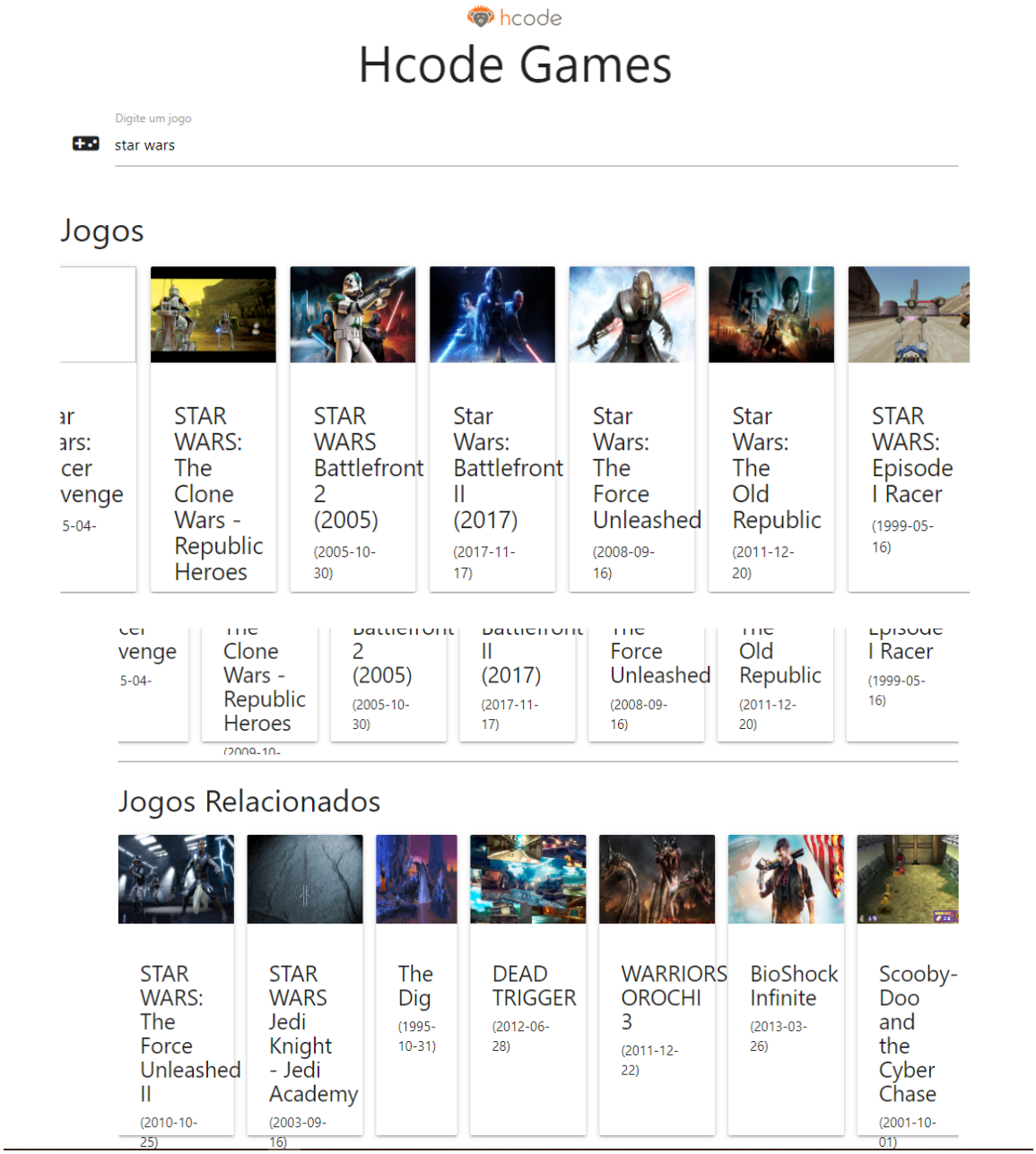
```
        let result = await fetch(`https://api.rawg.io/api/games/${name}/suggested`);

        result = await result.json();

        return result;

    } catch (err) {
        console.error(err);
    }
}
```

Ótimo. Podemos partir para o arquivo HTML. Agora, ao chamar as funções, não iremos mais usar o método **.then()**. Ao invés disso, iremos armazenar o valor retornado pela função em uma variável e continuar o código abaixo dela normalmente, como se fosse um código síncrono. Para isso, vamos precisar mudar a função **initGames()** para uma async function. Seu código ficará assim:

```
async function initGames(gamename) {
    setGameLoad(gamesListPrincipal);
    gamesListRelationed.innerHTML = '';
    let games = await getGameByName(gamename);
    gamesListPrincipal.innerHTML = '';
    games.results.forEach(game => {
        let divGame = setGameHTML(game);
        divGame.addEventListener('click', async e => {
            setGameLoad(gamesListRelationed);
            let gameTag = e.currentTarget;
            let gamename = gameTag.dataset.gamename;
            let gamesRelationed = await getRelatedGamesByName(gamename);
            gamesListRelationed.innerHTML = '';
            gamesRelationed.results.forEach(game => {

                let divGameRelationed = setGameHTML(game);

                gamesListRelationed.append(divGameRelationed);

            });

        });

        gamesListPrincipal.append(divGame);

    });

}
```

Nossa função agora é assíncrona e está esperando o valor retornado pelas promessas. Apenas um detalhe importante: perceba que foi necessário adicionar o identificador async no método **addEventListener()**, pois esse método cria uma nova função, e dentro dela usamos o await.

Após essas alterações, vamos testar o nosso projeto (guardamos o melhor jogo pro final):



E o nosso projeto está funcionando sem problemas! Muito bom, conseguimos adaptar nosso site para async/await e agora temos um projeto que foi moldado em vários conceitos de código assíncrono: callbacks, promises e por fim funções assíncronas. Você pode acessar todos os códigos desenvolvidos nessa série neste link: https://github.com/hcodebr/javascript-assincrono-blog.

Chegamos ao fim da nossa jornada pelo JavaScript assíncrono. Com o conhecimento que adquirimos juntos nesses artigos, estamos agora preparados para decidir quando usaremos cada um desses recursos nas situações que podemos nos deparar durante o nosso desenvolvimento. Esperamos que tenham gostado dessa série e que ela o ajude no seu aprendizado em JavaScript.

Teria algum outro tema que você gostaria de ver aqui em nosso Blog? Você pode enviar suas sugestões clicando aqui.

Muito obrigado por ter lido essa matéria! A gente se vê no próximo artigo :)