Entenda tudo sobre Async/Await

imasters.com.br/front-end/entenda-tudo-sobre-asyncawait

27 de junho de 2017

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/. A API tem *endpoints* para filmes, personagens, planetas, espécies, veículos e naves. Todos os exemplos abaixo estão disponíveis no meu GitHub.

Escrevendo uma função assíncrona com Promises

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

```
const fetch = require('node-fetch');

function getPerson(id) {
  fetch(`http://swapi.co/api/people/${id}`)
    .then(response => response.json())
    .then(person => console.log(person.name));
}

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:

```
$ node sample.js
Luke Skywalker
```

Escrevendo uma função assíncrona com async/await

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

```
const fetch = require('node-fetch');

async function getPerson(id) {
  const response = await fetch(`http://swapi.co/api/people/${id}`);
  const person = await response.json();
  console.log(person.name);
}

getPerson(1);
```

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

```
// Promise
function getPerson(id) {...}

// async/await
async function getPerson(id) {...}
```

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

```
// Promise
fetch(`http://swapi.co/api/people/${id}`)
  .then(response => response.json())
  .then(person => console.log(person.name));

// async/await
const response = await fetch(`http://swapi.co/api/people/${id}`);
const person = await response.json();
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.

```
$ node sample.js
Luke Skywalker
```

Utilizando async/await com Promises

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

```
const fetch = require('node-fetch');

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

getPerson(1)
  .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:

```
$ node sample.js
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().

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

Executando o código, ainda temos o mesmo resultado.

```
$ node sample.js
Luke Skywalker
```

Resolvendo ou rejeitando uma Promise com async/await

Para resolver uma Promise com async/await:

```
async function getPerson(id) {
  return id;
}

getPerson(1)
  .then(id => console.log(id)); // 1
```

Para rejeitar uma Promise com async/await:

```
async function getPerson(id) {
  throw Error('Not found');
}

getPerson(0)
  .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.

```
const fetch = require('node-fetch');

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

// id = 0
getPerson(0)
  .then(person => console.log(person.name));
```

Executando o código, temos o seguinte resultado.

```
$ node sample.js
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.

```
...

// id = 0
getPerson(0)
  .then(person => console.log(person));
```

Executando o código, temos o seguinte resultado:

```
$ node sample.js
{ 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.

```
const fetch = require('node-fetch');

async function getPerson(id) {
  const response = await fetch(`http://swapi.co/api/people/${id}`);
  const body = await response.json();

  if (response.status !== 200) {
    throw Error(body.detail);
  }

  return body;
}
```

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.

```
getPerson(1)
  .then(person => console.log(person.name)) //Luke Skywalker
  .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.

```
getPerson(0)
  .then(person => console.log(person.name))
  .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:

```
const fetch = require('node-fetch');

async function getPerson(id) {...}

async function loadPerson(id) {
  try {
    const person = await getPerson(id);
    console.log(person.name);

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

loadPerson(0);
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:

```
$ node sample.js
Not found
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.

```
const fetch = require('node-fetch');

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

getPerson(1)
  .then(person => console.log(person.name));
```

Executando o código, temos o seguinte resultado:

```
$ node sample.js
Luke Skywalker
```

Podemos utilizar arrow functions também.

```
const fetch = require('node-fetch');

// arrow function
const getPerson = async (id) => {
  const response = await fetch(`http://swapi.co/api/people/${id}`);
  return await response.json();
};

getPerson(1)
  .then(person => console.log(person.name));
```

Executando o código, ainda temos o mesmo resultado:

```
$ node sample.js
Luke Skywalker
```

Entendendo o await

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

```
const fetch = require('node-fetch');

const getPerson = async (id) => {
  const response = await fetch(`http://swapi.co/api/people/${id}`);
  return await response.json();
};

const person = await getPerson(1);
console.log(person.name);
```

Executando o código, temos o seguinte resultado:

```
$ node sample.js

const person = await getPerson(1);
               ^^^^^^^^

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:

```
$("document").ready(function () {
  // code
});
```

Utilizando JavaScript

```
(function () {
  // code
})();
```

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

```
(async function () {
  // code
})();
```

Podemos utilizar arrow functions também:

```
(async () => {
  // code
})();
```

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

```
const fetch = require('node-fetch');

const getPerson = async (id) => {
  const response = await fetch(`http://swapi.co/api/people/${id}`);
  return await response.json();
};

// IIFE - Immediately-Invoked Function Expression
(async function () {
  const person = await getPerson(1);
  console.log(person.name);
})();
```

Executando o código, ainda temos o mesmo resultado:

```
$ node sample.js
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.

```
const fetch = require('node-fetch');

// Class
class StarWars {
  async getPerson(id) {
    const response = await fetch(`http://swapi.co/api/people/${id}`);
    return await response.json();
  }
}

// IIFE - Immediately-Invoked Function Expression
(async () => {
  const sw = new StarWars();
  const person = await sw.getPerson(1);
  console.log(person.name);
})();
```

Executando o código, ainda temos o mesmo resultado.

```
$ node sample.js
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.

```
// star-wars.js

const fetch = require('node-fetch');

class StarWars {
  async getPerson(id) {
    const response = await fetch(`http://swapi.co/api/people/${id}`);
    return await response.json();
  }

  async getFilm(id) {
    const response = await fetch(`http://swapi.co/api/films/${id}`);
    return await response.json();
  }
}

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:

```
const StarWars = require('./star-wars');

async function loadData() {
  const sw = new StarWars();

  const person = await sw.getPerson(1);
  const film = await sw.getFilm(1);

  console.log(person.name);
  console.log(film.title);
}

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:

```
$ node sample.js
Luke Skywalker
A New Hope
```

Aguardando múltiplas requisições simultâneas

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

```
const StarWars = require('./star-wars');

async function loadData() {
  const sw = new StarWars();

  const personPromise = sw.getPerson(1);
  const filmPromise = sw.getFilm(1);

  const person = await personPromise;
  const film = await filmPromise;

  console.log(person.name);
  console.log(film.title);
}

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:

```
$ node sample.js
Luke Skywalker
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.

```
const StarWars = require('./star-wars');

async function loadData() {
  const sw = new StarWars();

  const [person, film] = await Promise.all([sw.getPerson(1), sw.getFilm(1)]);

  console.log(person.name);
  console.log(film.title);
}

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.

```
$ node sample.js
Luke Skywalker
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*