O que vamos aprender

Hoje você aprenderá o que é e como funciona uma operação assíncrona e entender também qual a sua importância no Node.js. Para isso, você irá rever duas formas de realizar operações assíncronas em JavaScript: callbacks e Promises, e quais as diferenças entre elas.Além disso, você verá como é feito input (leitura) e output (escrita) em arquivos localmente com Node.js!
Por último, você aprenderá como reescrever código que utiliza callbacks de forma que ele passe a utilizar Promises.

Você será capaz de:

- Realizar operações assíncronas utilizando callbacks;
- Realizar operações assíncronas utilizando Promises;
- Ler e escrever arquivos localmente com Node.js;
- Escrever seus próprios scripts que criam e consomem Promises;
- Reescrever código que usa callbacks para que use Promises.

Por que isso é importante?

O JavaScript é uma linguagem *single-threaded* , ou seja, executa apenas uma operação de cada vez. Isso quer dizer que, quando temos uma operação demorada no código, toda vez que essa operação é executada, o JavaScript precisa esperar que ela termine antes de fazer qualquer outra coisa.
Para o navegador, isso significa travar até mesmo a renderização da tela e deixar o usuário sem ação nenhuma durante todo o tempo que essa operação demorar para ser completada. Para o servidor, isso quer dizer não conseguir processar nenhuma outra requisição até que determinada operação termine.
Sendo assim, para que possamos escrever aplicações com boa performance e um boa experiência para o usuário, é importante sabermos como realizar operações demoradas de forma assíncrona , ou seja, fora do contexto de execução do restante do JavaScript . Esse conhecimento pode ser, muitas vezes, a diferença entre escrever um código bom e performático e escrever um código que não funciona, ou é extremamente lento.

Callbacks

Primeiro, vamos começar com o conceito de callback. Conforme o próprio nome diz, callback tem a ver com "chamar de volta". Basicamente, toda vez que precisarmos que algo seja processado em segundo plano, devemos registrar uma callback. Ela será executada quando a operação que solicitamos for concluída. Podemos pensar

em callbacks como sendo uma forma de dizermos pro *runtime* JavaScript um "vê lá e me avisa". 😆
Vamos usar como exemplo a função `readFile` do módulo `fs` do Node.js. Ela realiza a leitura de um arquivo e, quando termina, chama uma função de callback, passando o resultado:

Copiar

```
const fs = require('fs');

fs.readFile('./arquivo.txt', (err, content) => {
  if (err) {
    console.error(`Erro ao ler o arquivo: ${err.message}`);
    return;
  }

  console.log(`Arquivo lido com sucesso. Conteúdo: ${content.toString('utf8')}`);
});
```

No exemplo acima, passamos uma função para `readFile` , à qual damos o nome de *callback* . Essa função de callback recebe dois parâmetros: o primeiro, que chamamos de `err` , é um erro que pode ter ocorrido durante a leitura do arquivo. Caso nenhum erro tenha ocorrido, esse parâmetro será `undefined` . O segundo parâmetro é, nesse caso, o conteúdo do arquivo, em forma de uma sequência de bytes, que chamamos de `content` . Caso ocorra um erro na leitura do arquivo, esse parâmetro será `undefined` .
Sabendo disso, veja abaixo o que esse código faz:

1. Primeiro, pedimos que o Node.js leia o arquivo, e passamos uma função de callback;
2. Quando a leitura do arquivo é concluída ou um erro acontece, nossa função é chamada;
3. Dentro dela, a primeira coisa que fazemos é verificar se existe um erro. Caso exista, escrevemos ele no console e encerramos a execução com o `return` ;
4. Caso nenhum erro tenha acontecido, sabemos que nosso arquivo foi lido com sucesso e, portanto, seu conteúdo está no segundo parâmetro, que chamamos de `content` .

Esse formato de callback que recebe dois parâmetros, erro e resultado, não foi utilizado por acaso. Callbacks desse tipo são chamadas de node-style callbacks e são, por convenção, a melhor maneira de se estruturar uma callback. Toda API de módulos nativos do Node.js utiliza esse mesmo formato de callbacks . Guarde essa informação, pois ela vai ser importante mais tarde. 😉

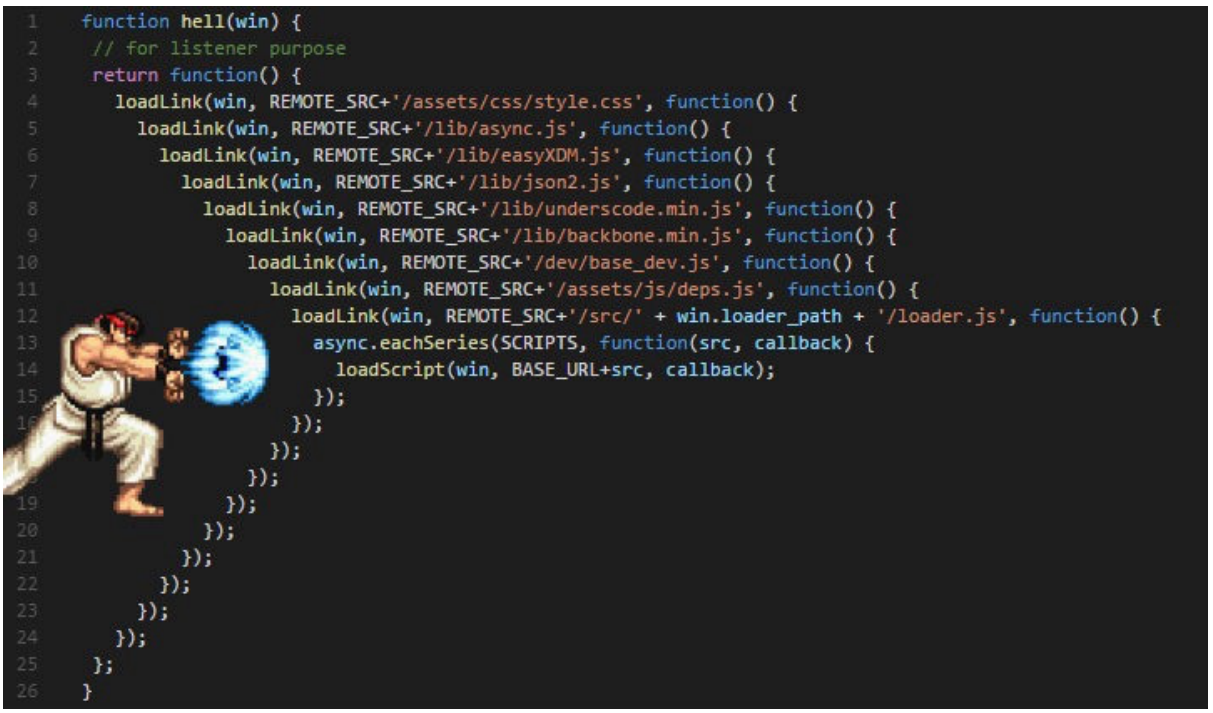
O lado ruim dos callbacks

A principal desvantagem das callbacks vem do fato de que o resultado de uma operação só existe dentro daquela callback; ou seja: se precisamos executar uma coisa depois da outra, precisamos colocar uma callback dentro da outra. À medida que vamos fazendo isso, vamos aumentando o nível de indentação necessária e, portanto, aumentamos a dificuldade de ler e até mesmo de dar manutenção no código. Vamos ver um exemplo:
Suponhamos que precisamos ler, sequencialmente, três arquivos, e que vamos fazê-lo de forma assíncrona, para não travar nosso servidor. O código para isso ficaria mais ou menos assim:

Copiar

```
const fs = require('fs');

fs.readFile('file1.txt', (err, file1Content) => {
  if (err) return console.log(`Erro ao ler arquivo 1: ${err.message}`);

  console.log(`Lido file1.txt com ${file1Content.byteLength} bytes`);

  fs.readFile('file2.txt', (err, file2Content) => {
    if (err) return console.log(`Erro ao ler o arquivo 2: ${err.message}`);

    console.log(`Lido file2.txt com ${file2Content.byteLength} bytes`);

    fs.readFile('file3.txt', (err, file3Content) => {
      if (err) return console.log(`Erro ao ler o arquivo 3: ${err.message}`);

      console.log(`Lido file3.txt com ${file3Content.byteLength} bytes`);
    });
  });
});
```

Com três níveis de indentação, já dá pra perceber que o código começa a ficar menos legível. Imagina como seria se tivéssemos ainda mais níveis de callbacks aninhadas?
A isso damos o nome de callback hell , que é quando temos uma callback dentro de outra, dentro de outra, dentro da outra etc., de forma que o código fica horrível de ler. Uma imagem que ilustra muito bem a callback hell é essa:



Uma forma de tentar resolver o problema é quebrar o código em infinitas funções menores, que não fazem nada além de chamar a próxima callback, mas isso também não é tão simples, organizado, ou mesmo bonito, e acaba por não funcionar. Veja um exemplo:

Copiar

```
const fs = require('fs');

const file3Callback = (err, file3Content) => {
  if (err) return console.log(`Erro ao ler o arquivo 3: ${err.message}`);

  console.log(`Lido file3.txt com ${file3Content.byteLength} bytes`);
};

const file2Callback = (err, file2Content) => {
  if (err) return console.log(`Erro ao ler o arquivo 2: ${err.message}`);

  console.log(`Lido file2.txt com ${file2Content.byteLength} bytes`);

  fs.readFile('file3.txt', file3Callback);
};

const file1Callback = (err, file1Content) => {
  if (err) return console.log(`Erro ao ler arquivo 1: ${err.message}`);

  console.log(`Lido file1.txt com ${file1Content.byteLength} bytes`);

  fs.readFile('file2.txt', file2Callback);
```

```
};

fs.readFile('file1.txt', file1Callback);
```

Depois de uma ou duas funções "aninhadas", fica fácil perder a linha de raciocínio, além de que é complicado entender logo de cara o fluxo em que o código acontece. Mas então como resolvemos isso?

Promises

Promises foram introduzidas à especificação do JavaScript em 2015 como uma forma de resolver a potencial *bagunça* trazida pelas callbacks. Sua ideia é um tanto quanto simples, mas faz uma grande diferença quando o assunto é melhorar a legibilidade do código. Na verdade, quando utilizamos Promises, ainda estamos utilizando um tipo de callback, mas que possui uma API mais legível e intuitiva. Bora entender melhor? Então segue a leitura!

O conceito de uma Promise, ou um objeto Promise, não é muito diferente da ideia de uma *promessa* na vida real: alguém se compromete com outra pessoa a fazer algo. Essa promessa pode ser cumprida e, portanto, resolvida , ou algo pode dar errado, fazendo com que não seja possível cumprir a promessa, que será então rejeitada . Promises no JavaScript funcionam do mesmo jeito: uma promessa é criada, e dentro dela existe código a ser executado. Se o código é executado sem nenhum problema, a Promise é resolvida através da função `resolve` , que veremos daqui a pouco. Se algo dá errado durante a execução do código, a Promise é rejeitada através da função `reject` .

OK, mas o que isso tem a ver com callbacks e com fluxo assíncrono?

A grande sacada das Promises está em como tratamos o sucesso ou o erro. Enquanto com callbacks temos apenas uma função que recebe tanto o sucesso quanto o erro, nas Promises temos uma forma de registrar uma callback para sucesso e outra forma de registrar uma callback para erros.

Além disso, outra grande vantagem das Promises está no fato de que podemos registrar *vários callbacks de sucesso* para serem executados um após o outro, sendo que o próximo callback recebe o resultado do callback anterior. Fazemos isso utilizando vários `.then` numa mesma Promise. As funções que passamos para cada `then` serão executadas em sequência, e o resultado de uma será passado para a próxima.

Antes de continuar assista o vídeo abaixo para entender como utilizar Promises.

Exemplo realizado no vídeo:

Exemplo 1: Tratando erros de forma síncrona.

Copiar

```
function dividirNumeros(num1, num2) {
```

```
  if (num2 == 0) throw new Error("Não pode ser feito uma divisão por zero");

  return num1 / num2;
}

try {
  const resultado = dividirNumeros(2, 1);
  console.log(`resultado: ${resultado}`);
} catch (e) {
  console.log(e.message);
}
```

Exemplo 2: Tratando erros de forma assíncrona.

Copiar

```
function dividirNumeros(num1, num2) {
  const promise = new Promise((resolve, reject) => {
    if (num2 == 0) reject(new Error("Não pode ser feito uma divisão por zero"));

    const resultado = num1 / num2;
    resolve(resultado)
  });

  return promise;
}

dividirNumeros(2, 1)
  .then(result => console.log(`sucesso: ${result}`))
  .catch(err => console.log(`erro: ${err.message}`));
```

No segundo exemplo, repare que a função `dividirNumeros` retorna uma Promise, ou seja: ela *promete* que vai dividir os números. Caso não consiga realizar a divisão, ela rejeita essa promessa, utilizando a função `reject` . Caso dê tudo certo, ela resolve a promessa, utilizando a função `resolve` . Tudo que será realizado de forma *assíncrona* , ou seja, em segundo plano, pode também ser encarado da mesma forma. Quando pedirmos, por exemplo, para o que o Node.js leia um arquivo do disco, ele nos retornará uma promessa de que vai ler esse arquivo. Se der tudo certo, essa promessa será resolvida. Caso contrário, ela será rejeitada.

Pra entender melhor, vamos usar um exemplo prático: vamos escrever uma função que *promete* ler arquivos do dia. Antes de começar, no entanto, vamos dar uma olhada na sintaxe da criação de uma Promise.

Sempre que precisarmos criar uma nova Promise, invocaremos o construtor através da palavra-chave `new` . Para esse construtor, devemos passar uma função, que é chamada de executor ; é ela quem vai, de fato, tentar cumprir a promessa que estamos fazendo. A função *executor* recebe outras duas funções como parâmetros: `resolve` e `reject` . Isso tudo fica assim:

Copiar

```
const p = new Promise((resolve, reject) => {
  // Aqui é onde vamos realizar a lógica que precisamos
  // para "tentar cumprir" a promessa
});
```

Feito isso, o próximo passo é escrever o código que, de fato, resolve a Promise. Já combinamos que nossa função promete ler um arquivo. Então, agora, vamos colocar dentro da função *executor* o código que busca resolver essa promessa:

Copiar

```
const fs = require('fs');

function readFilePromise (fileName) {
  return new Promise((resolve, reject) => {

    fs.readFile(fileName, (err, content) => {
      if (err) return reject(err);
      resolve(content);
    });

  });
}
```

Vamos entender o que estamos fazendo aqui:

- Recebemos, como parâmetro, o nome do arquivo que queremos ler, `fileName` na função `readFilePromise(fileName)` ;
- Criamos e retornamos uma nova Promise, `Promise((resolve, reject) => {}` ;
- Chamamos o módulo nativo do node, `fs` , para realizar a leitura desse arquivo, `fs.readFile(fileName, (err, content) => {})` ;
- Dentro da callback `fs.readFile(fileName, (err, content) => {})` que passamos para a função `readFile` , verificamos se ocorreu um erro (`if (err)`). Se sim, rejeitamos a Promise e encerramos a execução - `reject(err)` ;
- Caso não tenha acontecido nenhum erro, resolvemos a Promise com o resultado da leitura do arquivo - `resolve(content)` .

Dessa forma, quem chamar nossa função poderá consumir os resultados da leitura do arquivo ou tratar qualquer erro que ocorrer utilizando Promises.

Antes de prosseguir, para entender como podemos consumir uma Promise, vamos nos atentar a alguns detalhes:

- A função que passamos para a Promise só consegue retornar um resultado através da função `resolve` que ela recebe. Por isso, o fato de chamarmos `return reject(err)` não faz diferença, já que a Promise será rejeitada, e o retorno da callback passada para `readFile` é simplesmente ignorado. Na verdade, isso geralmente é válido para qualquer callback. Como callbacks geralmente são chamadas para lidar com resultados, seu retorno raramente importa para a função que a chamou ou que recebeu esses resultados.
- `resolve` e `reject` são os nomes das funções que a Promise passa para a função *executor* . No entanto, para nós, elas são apenas parâmetros que são passados pra nossa função. Logo, não importa muito o nome que damos a elas, pois para o JavaScript isso é indiferente. De qualquer forma, chamar essas funções de qualquer outra coisa não é considerado uma boa prática, pois pode dificultar a legibilidade do código.

Dito isso, vamos agora entender como podemos consumir essa Promise. Vimos antes que Promises permitem que a callback de erro seja registrada de determinada forma e que callbacks de sucesso sejam registradas de outra forma. Note o uso do plural aqui: utilizando Promises, podemos definir mais de uma callback de sucesso.

Vamos a um exemplo de como podemos consumir a Promise que estamos retornando da nossa função logo acima:

Copiar

```
// ...

readFilePromise('./file.txt') // A função me promete que vai ler o arquivo
  .then((content) => { // Caso ela cumpra o que prometeu
    console.log(`Lido arquivo com ${content.byteLength} bytes`); // Escrevo o resultado no console
  })
  .catch((err) => { // Caso ela não cumpra o que prometeu
    console.error(`Erro ao ler arquivo: ${err.message}`); // Escrevo o erro no console
  });
```

Por que isso é importante? Essa funcionalidade nos permite criar estruturas de *pipeline* , em que uma operação recebe como entrada o resultado da operação anterior, e esses resultados todos podem ser compostos e formar um único resultado de forma extremamente fácil!

Para demonstrar isso, e como Promises tornam o código mais legível, vamos reescrever o código que nos levou ao callback hell mas, dessa vez, utilizando Promises:

Copiar

```
const fs = require('fs');

function readFilePromise (fileName) {
  return new Promise((resolve, reject) => {
    fs.readFile(fileName, (err, content) => {
      if (err) return reject(err);
      resolve(content);
    });
  });
}

readFilePromise('file1.txt') // A função me promete que vai ler o arquivo
  .then((content) => { // Caso arquivo 1 seja lido,
    console.log(`Lido file1.txt com ${content.byteLength} bytes`); // Escrevo o resultado no console
    return readFilePromise('file2.txt'); // Chamo novamente a função, que me retorna uma nova Promise
  })
  .then(content => { // Caso a Promise retornada pelo `then` anterior seja resolvida,
    console.log(`Lido file2.txt com ${content.byteLength} bytes`); // Escrevemos o resultado no console
    return readFilePromise('file3.txt'); // E chamamos a função novamente, recebendo uma nova promessa
  })
  .then((content) => { // Caso a promessa de leitura do `file3.txt` seja resolvida,
    console.log(`Lido file3.txt com ${content.byteLength} bytes`); // Logamos o resultado no console
  })
  .catch((err) => { // Caso qualquer uma das promessas ao longo do caminho seja rejeitada
    console.log(`Erro ao ler arquivos: ${err.message}`); // Escrevemos o resultado no console
  });
```

E nada mais de callback hell! Agora temos um código muito mais simples de interpretar e que não vai nos dar dor de cabeça quando precisarmos modificar. 😁

Lendo arquivos com métodos síncronos

Agora que entendemos como funcionam callbacks e promises, vamos nos aprofundar um pouco mais no módulo `fs` do node e na leitura e escrita de arquivos.
Primeiro, é importante saber que não precisamos ler arquivos "em segundo plano". Podemos fazer isso de forma síncrona, ou seja: parar a execução de todo o programa até que um arquivo seja lido.
Os métodos assíncronos não esperam o comando atual terminar para iniciar o próximo. Se quisermos ler um arquivo de maneira assíncrona, o Javascript não vai esperar o arquivo inteiro ser lido para só então dar continuidade ao script. Se quisermos esse comportamento, precisamos de um método síncrono . O método disponibilizado pelo módulo `fs` para leitura síncrona de arquivos é o `fs.readFileSync` .
Vamos utilizá-lo no exemplo a seguir.
Para começar, vamos criar uma pasta para nosso projeto, chamada `io-local` .
Iniciaremos nosso projeto Node.js usando o comando `npm init` . Feito isso, vamos criar um arquivo chamado `readFileSync.js` e colocar nele o seguinte código:
io-local/readFileSync.js

Copiar

```
const fs = require('fs');

const nomeDoArquivo = 'meu-arquivo.txt';

try {
  const data = fs.readFileSync(nomeDoArquivo, 'utf8');
  console.log(data);
} catch (err) {
  console.error(`Erro ao ler o arquivo: ${err.path}`);
  console.log(err);
}
```

Logo após importarmos o módulo `fs` , criamos uma variável chamada `nomeDoArquivo` , contendo o nome (fixo) do arquivo que vamos ler e, em seguida, chamamos o método `fs.readFileSync` .
Rode o script com `node readFileSync.js` . Gerou um erro, certo? Isso aconteceu porque estamos tentando ler um arquivo que não existe! Vamos resolver esse probleminha daqui a pouco!

Método `fs.readFileSync`

Esse método é responsável por ler arquivos e trazer seu conteúdo para dentro do Node.js. Por ser síncrono , ele espera a leitura do arquivo terminar para, só então, atribuir o resultado à constante `data` .
O método `readFileSync` recebe dois parâmetros:

- O nome do arquivo;
- Um parâmetro opcional que, quando é uma string, define o *encoding* que será utilizado durante a leitura do arquivo.

Mas e se ocorrer um erro na leitura do arquivo?
Com funções síncronas, como `readFileSync` , você deve tratar explicitamente os erros que puderem acontecer. Nesse exemplo, usamos um bloco `try...catch` para capturar quaisquer erros que possam acontecer durante a leitura do arquivo e imprimimos uma mensagem para o usuário no terminal.
Agora vamos resolver o probleminha que estamos tendo ao tentar ler o arquivo!
Nota : Antes de continuar, não se esqueça de criar um arquivo `meu-arquivo.txt` com algum conteúdo dentro!
Ao rodar o script `readFileSync.js` com o comando `node readFileSync.js` , você deverá ver o conteúdo do seu arquivo impresso no terminal.
Mas e se tivéssemos outras partes do script que não deveriam esperar a leitura do arquivo ser feita? Por exemplo, e se tivéssemos que ler vários arquivos ao mesmo tempo? Para isso, utilizamos um método assíncrono , que veremos a seguir.

Lendo arquivos com métodos assíncronos

O método fornecido pelo módulo `fs` para leitura assíncrona de arquivos é o `fs.readFile` . Na versão padrão do `fs` , a função `readFile` aceita um callback, que é chamado quando a leitura do arquivo termina.
Continue lendo para ver o método `fs.readFile` em ação.
Vamos criar um arquivo chamado `readFile.js` dentro da nossa pasta `io-local` e colocar nele o seguinte código:
io-local/readFile.js

Copiar

```
const fs = require('fs');

const nomeDoArquivo = 'meu-arquivo.txt';

fs.readFile(nomeDoArquivo, 'utf8', (err, data) => {
  if (err) {
    console.error(`Não foi possível ler o arquivo ${nomeDoArquivo}\n Erro: ${err}`);
    process.exit(1);
  }
  console.log(`Conteúdo do arquivo: ${data}`);
```

```
});
```

Método fs.readFile

Esse método também é responsável por ler arquivos e trazer seu conteúdo para dentro do Node.js.
Ele recebe três parâmetros:

- O nome do arquivo;
- Um parâmetro opcional que, quando é uma string, define o *encoding* que será utilizado durante a leitura do arquivo;
- Uma callback que permite receber e manipular os dados lidos do arquivo.

A callback é uma função que recebe dois parâmetros: `err` e `data` . Caso ocorra um erro durante a leitura do arquivo, o parâmetro `err` virá preenchido com as informações referentes ao erro. Quando esse parâmetro vier vazio, significa que a leitura do conteúdo do arquivo ocorreu sem problemas. Nesse caso, o segundo parâmetro, `data` , virá preenchido com o conteúdo do arquivo.
Rode o comando `node readFile.js` . Você obterá uma saída semelhante a esta: `Conteúdo do arquivo: Meu texto! Meu texto! Meu texto! Meu texto!` .
O tipo de encoding que escolhemos é muito importante. Se não for especificado, por padrão, o arquivo será lido como *raw buffer* , que é um formato muito útil quando estamos enviando dados através de requisições HTTP. No nosso caso, como queremos manipular o conteúdo do arquivo como uma string, então o certo é especificar o encoding.
Nota : É importante lembrar que esses dados ficam armazenados em memória. Ou seja, caso você tenha um arquivo de 1GB de texto, você trará 1GB de dados para a memória RAM.
No entanto, essa não é a única forma do método `readFile` . O módulo `fs` possui um segundo modelo de API que, em vez de trabalhar com callbacks, retorna Promises, o que torna seu uso muito mais recomendável.
Para utilizar a interface de Promises do `fs` , precisamos alterar a importação do módulo `fs` , importando, agora `('fs').promises` . Vamos ver como ficaria o código acima se utilizarmos Promises:
io-local/readFile.js

Copiar

```
const fs = require('fs').promises;

const nomeDoArquivo = 'meu-arquivo.txt';

fs.readFile(nomeDoArquivo, 'utf8')
  .then((data) => {
    console.log(`Conteúdo do arquivo: ${data}`);
  })
```

```
  .catch((err) => {
    console.error(`Não foi possível ler o arquivo ${nomeDoArquivo}\n Erro: ${err}`);
    process.exit(1); // Encerra a execução do script e informa ao sistema operacional que houve um erro com código
  });
```

Dessa forma, sempre que precisarmos ler arquivos de forma assíncrona, podemos utilizar o método readFile do módulo ('fs').promises .

Escrevendo dados em arquivos

Escrever dados em arquivos é um processo bem parecido com a leitura de dados! Assim como o módulo ('fs').promises disponibiliza o método readFile , há também o método writeFile .
Atenção: O módulo fs também disponibiliza um método writeFile , que funciona utilizando callbacks. Vamos utilizar diretamente o módulo ('fs').promises , já que o uso de Promises é bem mais encorajado que o uso de callbacks 👍
io-local/writeFile.js

Copiar

```
const fs = require('fs').promises;

fs.writeFile('./meu-arquivo.txt', 'Meu textão')
  .then(() => {
    console.log('Arquivo escrito com sucesso!');
  })
  .catch((err) => {
    console.error(`Erro ao escrever o arquivo: ${err.message}`);
  });
```

Rode o script com node writeFile.js . Repare que o conteúdo do meu-arquivo.txt foi alterado para "Meu textão".

Utilizando async/await

Acontece que nem sempre queremos utilizar essa API das Promises. Muitas vezes, queremos simplesmente buscar o resultado e pronto. E é aí que entra o async/await . Essas duas palavras-chave foram criadas para trabalhar com Promises como se estivéssemos trabalhando com código síncrono.
A questão é que toda função na qual utilizamos async , automaticamente passa a retornar uma Promise, que será rejeitada em caso de erro, e resolvida em caso de sucesso.

O resultado de usarmos async / await é que o código fica com uma sintaxe quase idêntica à sintaxe utilizada para código síncrono. Veja como fica o exemplo anterior utilizando async/await :

Copiar

```
const fs = require('fs').promises;

async function main() {
  try {
    await fs.writeFile('./meu-arquivo.txt', 'Meu textão');
    console.log('Arquivo escrito com sucesso!');
  } catch (err) {
    console.error(`Erro ao escrever o arquivo: ${err.message}`);
  }
}

main()
```

Perceba que, para podermos utilizar o async/await , precisamos criar uma função main e colocar nossa lógica dentro dela. Isso acontece porque, por enquanto, o await só pode ser utilizado dentro de funções async .
Repare também que não temos mais nenhum .then , e que todo o tratamento de erro e sucesso foi feito com um try ... catch , da mesma forma que fizemos quando estávamos utilizando o fs.readFileSync .
Ainda sobre o writeFile , você pode especificar algumas opções na escrita de arquivos passando um terceiro parâmetro opcional para os métodos writeFile e writeFileSync . A opção flag especifica como o arquivo deve ser aberto e manipulado. O padrão é 'w' , que especifica que o arquivo deve ser aberto para escrita. Se o arquivo não existir, ele é criado. Caso contrário, ele é reescrito, ou seja, tem seu conteúdo apagado antes de o novo conteúdo ser escrito. A flag 'wx' , por exemplo, funciona como 'w' , mas lança um erro caso o arquivo já exista:

Copiar

```
const fs = require('fs').promises;

// A flag wx abre o arquivo para escrita **apenas** caso ele não exista. Caso o contrário, um erro será lançado
fs.writeFile('./meu-arquivo.txt', 'Eu estive aqui :eyes:', { flag: 'wx' })
  .then(() => {
    console.log('Arquivo salvo');
  })
  .catch((err) => {
    // Se o arquivo existir, um erro é retornado
    console.error('err');
  });
```

Note que, quando rodamos o código com a flag `wx` , tentando escrever no arquivo `meu-arquivo.txt` , é gerado o seguinte erro:

Copiar

```
[...]
[Error: EEXIST: file already exists, open './meu-arquivo.txt'] {
  errno: -17,
  code: 'EEXIST',
  syscall: 'open',
  path: './meu-arquivo.txt'
}
```

No código, mude o nome do arquivo para `meu-novo-arquivo.txt` e rode novamente o script com `node writeFileSync.js` . Repare que foi criado um novo arquivo com o nome que utilizamos e com o conteúdo `Eu estive aqui :eyes:` .
Você pode ler mais sobre as flags disponíveis aqui.

Rodando tudo junto

Promises são executadas assim que são criadas. Isso quer dizer que, no código abaixo, todos os arquivos serão lidos ao mesmo tempo e, portanto, não teremos uma forma de saber quando cada um vai terminar de ser lido. Mas e se precisarmos do resultado da leitura dos três arquivos?
Entra no palco: `Promise.all` !
O `Promise.all` é um método da Promise que nos permite passar um array de Promises e receber, de volta, uma única Promise. Ela será resolvida com os resultados de todas as Promises, assim que todas elas forem resolvidas. Esse método também nos permite utilizar um único `catch` , que será chamado caso qualquer uma das Promises seja rejeitada.
Vamos reescrever quase o mesmo código que fizemos lá em cima, que utilizamos para mostrar como Promises evitam o callback hell. Desta vez, vamos escrever, no final, a soma do tamanho de todos os arquivos. Além disso, vamos utilizar o módulo `('fs').promises` para não precisarmos trabalhar com callbacks manualmente.

Copiar

```
const fs = require('fs').promises;

Promise.all([
  fs.readFile('file1.txt'),
  fs.readFile('file2.txt'),
  fs.readFile('file3.txt'),
])
  .then(([file1, file2, file3]) => {
    const fileSizeSum = file1.byteLength + file2.byteLength + file3.byteLength;
    console.log(`Lidos 3 arquivos totalizando ${fileSizeSum} bytes`);
  })
```

```
  .catch((err) => {
    console.error(`Erro ao ler arquivos: ${err.message}`);
  });
```

Ótimo! Agora, estamos lendo os três arquivos ao mesmo tempo, e nosso .then será executado quando a leitura de todos eles terminar, recebendo como parâmetro um array com o resultado de cada uma das Promises.

Exercícios

back-end

Antes de começar: versionando seu código

Para versionar seu código, utilize o seu repositório de exercícios. 😉
Abaixo você vai ver exemplos de como organizar os exercícios do dia em uma branch, com arquivos e commits específicos para cada exercício. Você deve seguir este padrão para realizar os exercícios a seguir.

1. Abra a pasta de exercícios:
2. Copiar
3. `$ cd ~/trybe-exercicios`
4. Certifique-se de que está na branch main e ela está sincronizada com a remota. Caso você tenha arquivos modificados e não comitados, deverá fazer um commit ou checkout dos arquivos antes deste passo.
5. Copiar

`$ git checkout main`

6. `$ git pull`
7. A partir da main, crie uma branch com o nome exercicios/26.2 *(bloco 26, dia 2)*
8. Copiar
9. `$ git checkout -b exercicios/26.2`
10. Caso seja o primeiro dia deste módulo, crie um diretório para ele e o acesse na sequência:
11. Copiar

`$ mkdir back-end`

12. `$ cd back-end`
13. Caso seja o primeiro dia do bloco, crie um diretório para ele e o acesse na sequência:

14. Copiar

```
$ mkdir bloco-26-introducao-ao-desenvolvimento-web-com-nodejs
```

15. `$ cd bloco-26-introducao-ao-desenvolvimento-web-com-nodejs`
16. Crie um diretório para o dia e o acesse na sequência:
17. Copiar

```
$ mkdir dia-2-nodejs-fluxo-assincrono
```

18. `$ cd dia-2-nodejs-fluxo-assincrono`
19. Os arquivos referentes aos exercícios deste dia deverão ficar dentro do diretório ~/trybe-exercicios/back-end/block-26-introducao-ao-desenvolvimento-web-com-nodejs/dia-2-nodejs-fluxo-assincrono. Lembre-se de fazer commits pequenos e com mensagens bem descritivas, preferencialmente a cada exercício resolvido.

    Verifique os arquivos alterados/adicionados:
20. Copiar

```
$ git status
On branch exercicios/26.2
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

21. `  modified:   exercicio-1`

    Adicione os arquivos que farão parte daquele commit:
22. Copiar

```
# Se quiser adicionar os arquivos individualmente
$ git add caminhoParaArquivo

# Se quiser adicionar todos os arquivos de uma vez, porém, atente-se
para não adicionar arquivos indesejados acidentalmente
```

23. `$ git add --all`

    Faça o commit com uma mensagem descritiva das alterações:
24. Copiar
25. `$ git commit -m "Mensagem descrevendo alterações"`

26. Você pode visualizar o log de todos os commits já feitos naquela branch com git log.
27. Copiar

```
$ git log
commit 100c5ca0d64e2b8649f48edf3be13588a77b8fa4 (HEAD -> exercicios/26.2)
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 17:48:01 2019 -0300

    Exercicio 2 - mudando o evento de click para mouseover, tirei o alert e coloquei pra quando clicar aparecer uma imagem do lado direito da tela

commit c0701d91274c2ac8a29b9a7fbe4302accacf3c78
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 16:47:21 2019 -0300

    Exercicio 2 - adicionando um alert, usando função e o evento click

commit 6835287c44e9ac9cdd459003a7a6b1b1a7700157
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 15:46:32 2019 -0300

    Resolvendo o exercício 1 usando eventListener
```

28. Resolvendo o exercício 1 usando eventListener
29. Agora que temos as alterações salvas no repositório local precisamos enviá-las para o repositório remoto. No primeiro envio, a branch exercicios/26.2 não vai existir no repositório remoto, então precisamos configurar o remote utilizando a opção --set-upstream (ou -u, que é a forma abreviada).
30. Copiar
31. `$ git push -u origin exercicios/26.2`
32. Após realizar o passo 9, podemos abrir a Pull Request a partir do link que aparecerá na mensagem do push no terminal, ou na página do seu repositório de exercícios no GitHub através de um botão que aparecerá na interface. Escolha a forma que preferir e abra a Pull Request. De agora em diante, você repetirá o fluxo a partir do passo 7 para cada exercício adicionado, porém como já definimos a branch remota com -u anteriormente, agora podemos simplificar os comandos para:
33. Copiar

```
# Quando quiser enviar para o repositório remoto
$ git push

# Caso você queria sincronizar com o remoto, poderá utilizar apenas
```

```
$ git pull
```

34.
35. Quando terminar os exercícios, seus códigos devem estar todos commitados na branch exercicios/26.2, e disponíveis no repositório remoto do GitHub. Pra finalizar, compartilhe o link da Pull Request no canal de Code Review para a monitoria e/ou colegas revisarem. Faça review você também, lembre-se que é muito importante para o seu desenvolvimento ler o código de outras pessoas. 👊

Agora, a prática

1. Crie uma função que recebe três parâmetros retorna uma Promise .
    1. Caso algum dos parâmetros recebidos não seja um número, rejeite a Promise com o motivo "Informe apenas números" .
    2. Caso todos os parâmetros sejam numéricos, some os dois primeiros e multiplique o resultado pelo terceiro ((a + b) * c).
    3. Caso o resultado seja menor que 50, rejeite a Promise com o motivo "Valor muito baixo"
    4. Caso o resultado seja maior que 50, resolva a Promise com o valor obtido.
2. Escreva um código para consumir a função construída no exercício anterior.
    1. Gere um número aleatório de 1 a 100 para cada parâmetro que a função recebe. Para gerar um número aleatório, utilize o seguinte trecho de código: Math.floor(Math.random() * 100 + 1) .
    2. Chame a função do exercício anterior, passando os três números aleatórios como parâmetros.
    3. Utilize then e catch para manipular a Promise retornada pela função:
        1. Caso a Promise seja rejeitada, escreva na tela o motivo da rejeição.
        2. Caso a Promise seja resolvida, escreva na tela o resultado do cálculo.
3. Reescreva o código do exercício anterior para que utilize async/await .
- Lembre-se: a palavra chave await só pode ser utilizada dentro de funções async .

4. Realize o download deste arquivo e salve-o como simpsons.json . Utilize o arquivo baixado para realizar os requisitos abaixo.
- Você pode utilizar then e catch , async/await ou uma mistura dos dois para escrever seu código. Procure não utilizar callbacks.
    1. Crie uma função que leia todos os dados do arquivo e imprima cada personagem no formato id - Nome . Por exemplo: 1 - Homer Simpson .

    2. Crie uma função que receba o `id` de uma personagem como parâmetro e retorne uma `Promise` que é resolvida com os dados da personagem que possui o `id` informado. Caso não haja uma personagem com o `id` informado, rejeite a Promise com o motivo "id não encontrado".
    3. Crie uma função que altere o arquivo `simpsons.json` retirando os personagens com `id` 10 e 6.
    4. Crie uma função que leia o arquivo `simpsons.json` e crie um novo arquivo, chamado `simpsonFamily.json` , contendo as personagens com `id` de 1 a 4.
    5. Crie uma função que adicione ao arquivo `simpsonFamily.json` o personagem `Nelson Muntz` .
    6. Crie uma função que substitua o personagem `Nelson Muntz` pela personagem `Maggie Simpson` no arquivo `simpsonFamily.json` .
5. Crie uma função que lê e escreve vários arquivos ao mesmo tempo.
    1. Utilize o `Promise.all` para manipular vários arquivos ao mesmo tempo.
    2. Dado o seguinte array de strings: `['Finalmente', 'estou', 'usando', 'Promise.all', '!!!']` Faça com que sua função crie um arquivo contendo cada string, sendo o nome de cada arquivo igual a `file<index + 1>.txt` . Por exemplo, para a string "Finalmente", o nome do arquivo é `file1.txt` .
    3. Programe sua função para que ela faça a leitura de todos os arquivos criados no item anterior, armazene essa informação e escreva em um arquivo chamado `fileAll.txt` .

O conteúdo do arquivo `fileAll.txt` deverá ser `Finalmente estou usando Promise.all !!!` .

Bônus

1. Crie um script que mostre na tela o conteúdo de um arquivo escolhido pela pessoa usuária:
    1. Pergunte à pessoa usuária qual arquivo ela deseja ler.
    2. Leia o arquivo indicado.
    3. Caso o arquivo não exista, exiba na tela "Arquivo inexistente" e encerre a execução do script.
    4. Caso o arquivo exista, escreva seu conteúdo na tela.
2. Crie um script que substitua uma palavra por outra em um arquivo escolhido pela pessoa usuária:
    1. Pergunte à pessoa usuária qual arquivo ela deseja utilizar.
    2. Leia o arquivo.
    3. Caso o arquivo não exista, exiba um erro na tela e encerre a execução do script.
    4. Caso o arquivo exista, solicite a palavra a ser substituída.
    5. Solicite a nova palavra, que substituirá a palavra anterior.

6. Imprima na tela o conteúdo do arquivo com as palavras já substituídas.
7. Pergunte o nome do arquivo de destino.
8. Salve o novo arquivo no caminho de destino.

Dica: Utilize a classe RegExp do JS para substituir todas as ocorrências da palavra com `replace(new RegExp(palavra, 'g'), novaPalavra)` .

Recursos Adicionais

- Asynchrony: Under the Hood - Shelley Vohr - JSConf EU
- Entendendo Promises de uma vez por todas
- Using Promises | MDN
- Promises | MDN
- Entenda tudo sobre Async/Await
- Entendendo funções callback em JavaScript
- ECMAScript proposal: Top-level await
