

# Boas-vindas!

Esteja confortável, pegue uma água e se acomode em um local tranquilo que já começamos.

Como você **chega?**

1



2



3



Esta aula será

- gravada

# Resumo

## da última aula

- ✓ Revisar funções em Javascript
- ✓ Entenda um callback e como as funções se relacionam com ele.
- ✓ Usando promessas em Javascript
- ✓ Entenda a diferença entre programação síncrona e assíncrona.

Perguntas?

AULA 4 – BACKEND

# Introdução ao Nodejs

# Objetivos da aula



Usar programação síncrona e assíncrona e aplicá-la no uso de arquivos



Conhecer o módulo nativo do Nodejs para interagir com arquivos.



Conhecer o uso de arquivos com callbacks e promessas



Conhecer as vantagens e desvantagens do FileSystem, bem como exemplos práticos.

# Glossário - Aula 3

**Função definida:** Função que é declarada com um nome desde o início. Eles geralmente são usados para tarefas específicas e geralmente não são reatribuídos.

**Função Anônima:** Uma função que é declarada sem nome, mas é atribuída a uma variável desde o início. Geralmente é usado para reatribuições ou para retornos de chamada.

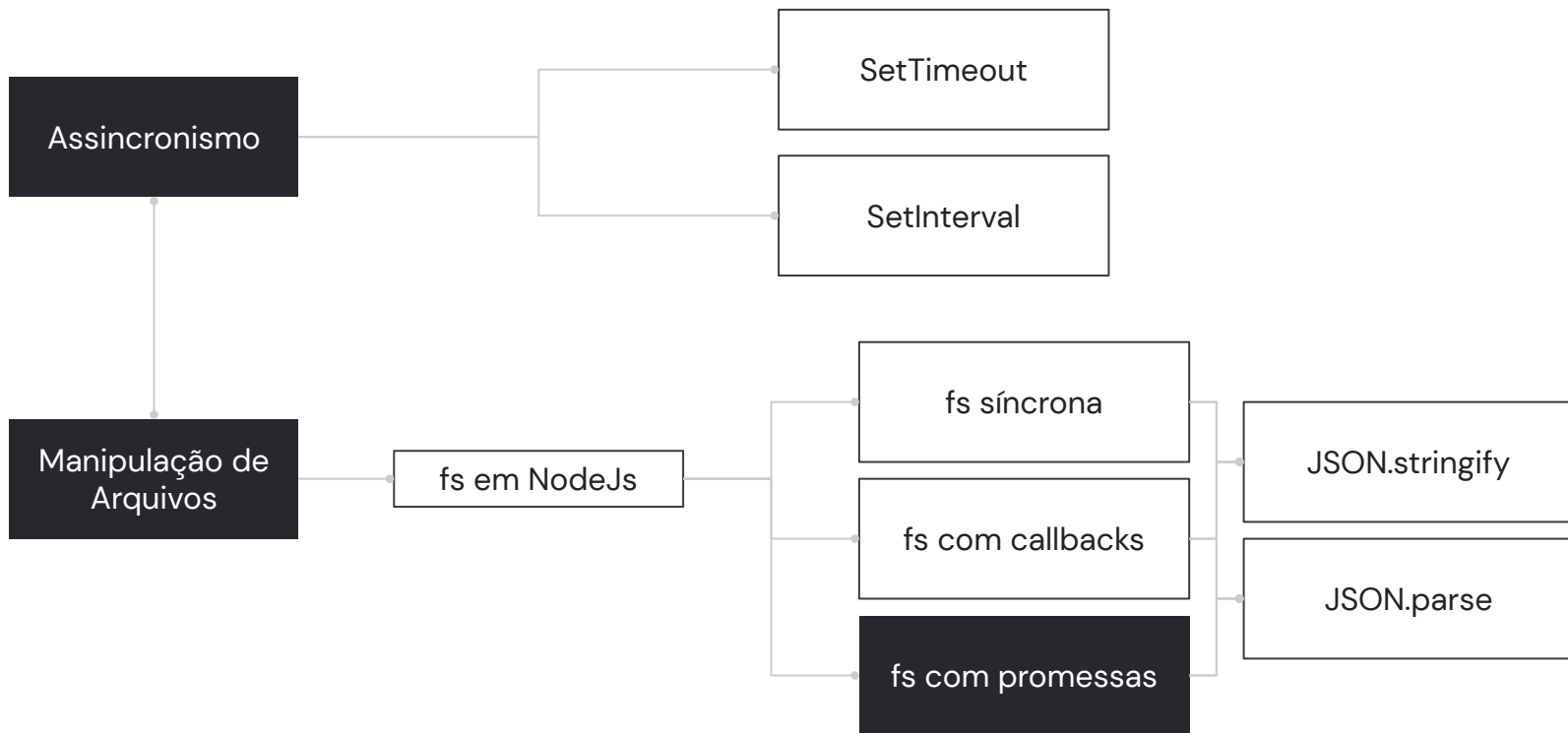
**Callback:** Bloco de código que encapsula uma ou mais instruções, para ser utilizado em qualquer outro momento do programa.

**Promessa:** Uma operação assíncrona que possui dois canais de saída: Resolver ou Rejeitar. Eles permitem um melhor controle do que callbacks

**Operação síncrona:** Uma operação de bloqueio na qual uma tarefa não pode iniciar até que a outra tarefa seja concluída.

**Operação assíncrona:** uma operação sem bloqueio na qual várias tarefas podem ser iniciadas, independentemente de as tarefas anteriores não terem sido concluídas.

# Mapa de conceitos - Javascript





## Para recordar...

**Sincronismo:** As operações síncronas ou de bloqueio são usadas quando precisamos que as operações sejam executadas uma após a outra, ou seja, é usada quando queremos que as tarefas sejam sequenciais, independentemente do tempo que cada operação leva.

**Assincronismo:** As operações assíncronas ou sem bloqueio são úteis quando precisamos de várias tarefas em execução, sem ter que esperar por tarefas que já estão em execução. Use-os quando precisar fazer alguma operação, sem afetar o fluxo principal.

# SetTimeout e setInterval

# setTimeout

**setTimeout** é usado para definir um cronômetro que executa uma tarefa após um determinado período de tempo. Isso nos permite entender em algumas linhas a ideia de assincronismo.

Ao contrário de uma operação síncrona, poderemos perceber como o `setTimeout` inicia sua execução, e uma vez decorrido o tempo, veremos o resultado, mesmo quando o resto das operações terminar.

[Documentação sobre sincronismo e assincronismo](#)

[Documentação sobre setTimeout](#)

# Exemplo de uso do setTimeout

JS sincrono.js

```
1 //Exemplo de operação síncrona
2 console.log("Iniciando tarefa!");
3 console.log("Executando operação");
4 console.log("Continuando operação");
5 console.log("Tarefa finalizada!");
6 /**
7  * Ordem de exibição:
8  * Iniciando a tarefa!
9  * Executando a operação
10  * Continuando operação
11  * Tarefa finalizada!
12 */
13
14 //Até agora tudo em ordem, um vai atrás do outro.
15 //E quanto a uma operação assíncrona?
16
```

JS setTimeout.js > ...

```
1 const temporizador = (callback) => {
2   setTimeout(() =>{
3     callback();
4   },5000);
5 }
6
7 let operacao = () => console.log("Executando operação");
8
9 console.log("Iniciando tarefa!");
10 temporizador(operacao); //Colocamos a "operação" no temporizador
11 console.log("Tarefa finalizada!");
12 /**
13  * Ordem de exibição:
14  * Iniciando tarefa!
15  * Tarefa finalizada!
16  * Executando operação
17  *
18  *
19  * Tarefa "operação" teve que esperar 5000 milissegundos (5 segundos)
20  * para poder rodar, mas sendo assíncrono, o programa pôde continuar
21  * e foi capaz de terminar sem esperar pela referida operação
22 */
```

# setInterval

**setInterval** funciona como `setTimeout`, a diferença é que ele irá zerar a contagem e executar a tarefa novamente toda vez que o intervalo de tempo for cumprido.

Um temporizador retorna um interruptor que permite parar o intervalo quando uma determinada operação é cumprida.

Muitas vezes é muito usado colocar limites de tempo em uma página para preencher formulários (Existem certas páginas que te dão um limite de tempo para fazer a operação, OU VOCÊ SAI).

# Exemplo do uso de setInterval

```
JS sincrono2.js > ...
1 //Exemplo de operação síncrona
2 console.log("Iniciando tarefa!");
3 0
4 console.log("Executando operação")
5 for(let contador = 1; contador<=5; contador++) {
6   console.log(contador);
7 }
8 console.log("Tarefa finalizada!")
9 /**
10  *Ordem de exibição:
11  *Iniciando tarefa!
12  * "Executando operação"
13  * 1
14  * 2
15  * 3
16  * 4
17  * 5
18  * "Tarefa finalizada!"
19  */
20
21 /**
22  * Novamente, tudo parece normal, a tarefa termina até
23  * que o loop terminou de contar de 1 a 5
24  * Como o assíncrono funcionará com intervalos?
25  */
26
```

```
JS setInterval.js > [0] contador > [0] timer > [0] setInterval() callback
1 //Exemplo com setInterval
2 let contador = () => {
3   let counter=1;
4   console.log("Executando operação");
5   let timer = setInterval(()=>{
6     console.log(counter++);
7     if(counter>5) {
8       clearInterval(timer); // após de contar 5
9     }
10  }, 1000)
11  /**
12   * Sendo um intervalo, o console.log(counter++) será executado
13   * a cada 1000 milissegundos (1 segundo)
14   */
15 }
16
17 console.log("Iniciando tarefa!");
18 contador();
19 console.log("Tarefa finalizada!")
20 /**
21  * Ordem de exibição:
22  * "Iniciando tarefa!"
23  * "Executando operação"
24  * "Tarefa finalizada!"
25  * 1 (aquí pasa 1 segundo)
26  * 2 (aquí pasa 1 segundo)
27  * 3 (aquí pasa 1 segundo)
28  * 4 (aquí pasa 1 segundo)
29  * 5
30  */
```

Além da memória... Gerenciamento de  
arquivos



## O problema: persistência de memória.

Quando começarmos a lidar com mais informações, nos depararemos com um dos **grandes incômodos do programador**: **ter que começar do 0 toda vez que o programa terminar sua execução.**

Todas as coisas que criamos, movemos ou trabalhamos, desaparecem, pois apenas persistem na memória, e esta é automaticamente apagada quando o programa termina.



## 👁️👁️ O mundo real não permite isso.

Se eu estivesse tentando cadastrar usuários no site da minha empresa, imagine se eu tivesse que atualizar a página!

Ao atualizar a página (**o servidor**), ela precisaria ser reiniciada e todos os usuários teriam desaparecido. Teríamos que pedir a cada usuário que se registrasse novamente e reenviasse suas informações.



## A solução!

A **primeira solução** para o problema de persistência de memória foi **salvar as informações em arquivos**. Estes são um conjunto de informações que podemos armazenar.

Assim, quando a informação for solicitada novamente, podemos ler o arquivo que salvamos e recuperar a informação, mesmo que o programa tenha finalizado.

Seu sistema operacional funciona assim!

# Implantando arquivos em nodejs - FS

# fs em nodejs

**fs** é a abreviação usada para **FileSystem**, que, como o nome indica, é um **sistema gerenciador de arquivos que nos fornecerá um nó para poder criar, ler, atualizar ou excluir um arquivo, sem precisar fazer isso do zero.**

Portanto, criar um arquivo com conteúdo será tão fácil quanto escrever algumas linhas de código, em vez de ter que lidar com dados binários e transformações complexas de nível inferior no computador.



## Importante

Considere que o curso utilizará muitas **soluções “já desenvolvidas anteriormente”**. A programação é baseada em um princípio fundamental: não reinvente a roda. Os módulos a utilizar são soluções já disponibilizadas por outros programadores, de forma a concentrar-se na resolução de problemas mais específicos.

[Documentação oficial fs](#)

# Como usamos o File System do Nodejs em nosso próprio código?



# Usando fs

fs existe desde que instalamos o Nodejs em nosso computador, portanto, para usá-lo, podemos chamá-lo de qualquer arquivo que tenhamos em nosso código com a seguinte linha:

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

A partir de então, todo o módulo FileSystem estará contido na variável fs. Devemos usá-lo apenas chamando seus métodos como uma classe. Podemos fazer isso de 3 maneiras: **síncronas**, com **callbacks** ou com **promessas**.

**1**

**Usando fs de forma síncrona**



# 1 fs síncrono

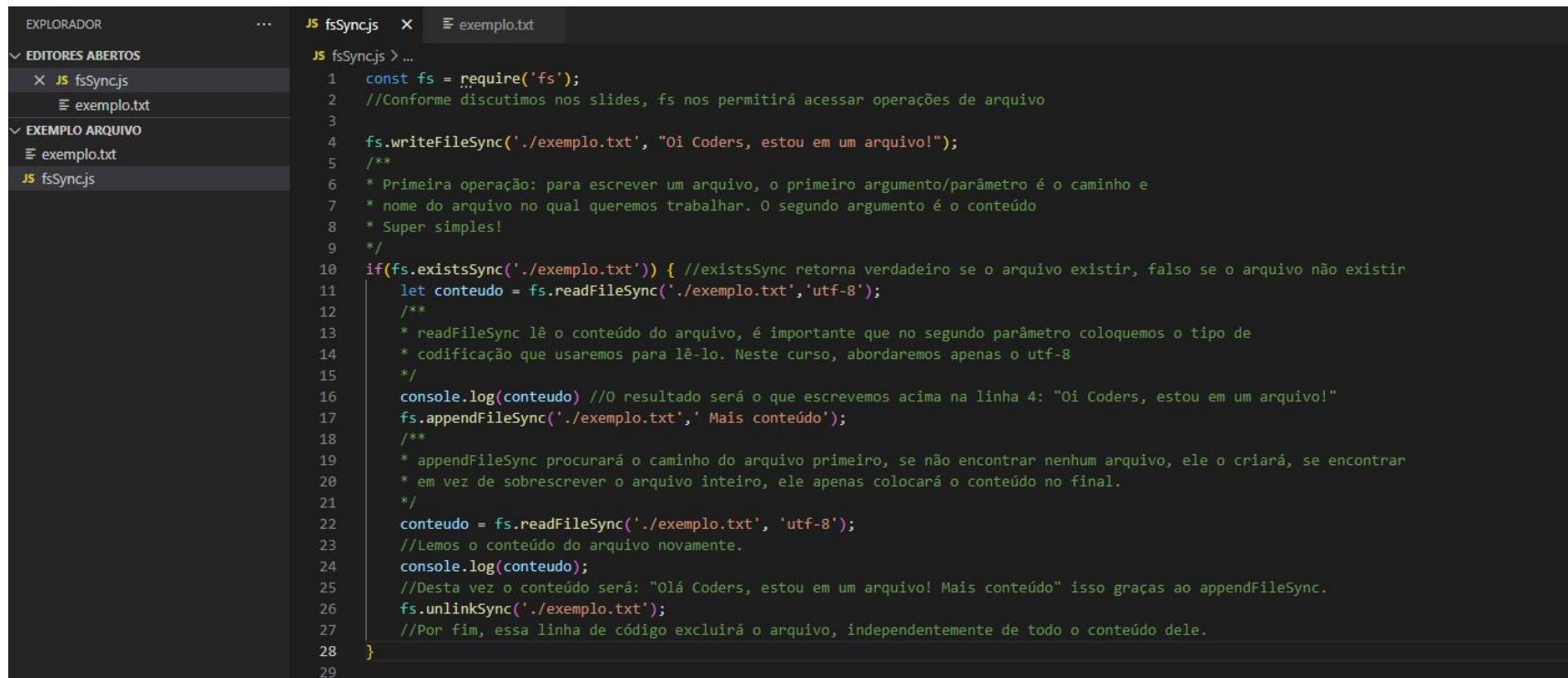
Usar fs de forma síncrona é bastante simples! Para isso, usaremos apenas a palavra Sync após cada operação que desejamos realizar. Existem muitas operações para trabalhar com arquivos, mas abordaremos apenas as principais.

As principais operações que podemos fazer com fs síncronos são:

- `writeFileSync` = Para gravar conteúdo em um arquivo. Se o arquivo não existir, ele o cria. Se existir, ele o sobrescreve.
- `readFileSync` = Para obter o conteúdo de um arquivo.
- `appendFileSync` = Para anexar conteúdo a um arquivo. Não é sobrescrito!
- `unlinkSync` = É o “delete” dos arquivos. excluirá o arquivo inteiro, não apenas o conteúdo.
- `existsSync` = Verifique se existe um arquivo!

# Exemplo de uso síncrono de fs

Vamos nos aprofundar na sintaxe das operações de arquivo síncronas com fs.



```
1  const fs = require('fs');
2  //Conforme discutimos nos slides, fs nos permitirá acessar operações de arquivo
3
4  fs.writeFileSync('./exemplo.txt', "Oi Coders, estou em um arquivo!");
5  /**
6   * Primeira operação: para escrever um arquivo, o primeiro argumento/parâmetro é o caminho e
7   * nome do arquivo no qual queremos trabalhar. O segundo argumento é o conteúdo
8   * Super simples!
9   */
10 if(fs.existsSync('./exemplo.txt')) { //existsSync retorna verdadeiro se o arquivo existir, falso se o arquivo não existir
11     let conteudo = fs.readFileSync('./exemplo.txt', 'utf-8');
12     /**
13      * readFileSync lê o conteúdo do arquivo, é importante que no segundo parâmetro coloquemos o tipo de
14      * codificação que usaremos para lê-lo. Neste curso, abordaremos apenas o utf-8
15      */
16     console.log(conteudo) //O resultado será o que escrevemos acima na linha 4: "Oi Coders, estou em um arquivo!"
17     fs.appendFileSync('./exemplo.txt', ' Mais conteúdo');
18     /**
19      * appendFileSync procurará o caminho do arquivo primeiro, se não encontrar nenhum arquivo, ele o criará, se encontrar
20      * em vez de sobrescrever o arquivo inteiro, ele apenas colocará o conteúdo no final.
21      */
22     conteudo = fs.readFileSync('./exemplo.txt', 'utf-8');
23     //Lemos o conteúdo do arquivo novamente.
24     console.log(conteudo);
25     //Desta vez o conteúdo será: "Olá Coders, estou em um arquivo! Mais conteúdo" isso graças ao appendFileSync.
26     fs.unlinkSync('./exemplo.txt');
27     //Por fim, essa linha de código excluirá o arquivo, independentemente de todo o conteúdo dele.
28 }
29
```

## **2** fs com callbacks

## 2 fs com callbacks

Funciona de maneira muito semelhante às operações síncronas. Somente no final eles receberão um último argumento, que, como podemos imaginar, deve ser um callback. Como vimos nas convenções de callback da última classe, o primeiro argumento geralmente é um erro.

Isso permite que você saiba se a operação correu bem ou se deu errado. Somente `readFile` lida com um segundo argumento, com o resultado da leitura do arquivo.

Finalmente: o tratamento de retorno de chamada é totalmente assíncrono, portanto, tenha cuidado ao usá-lo.

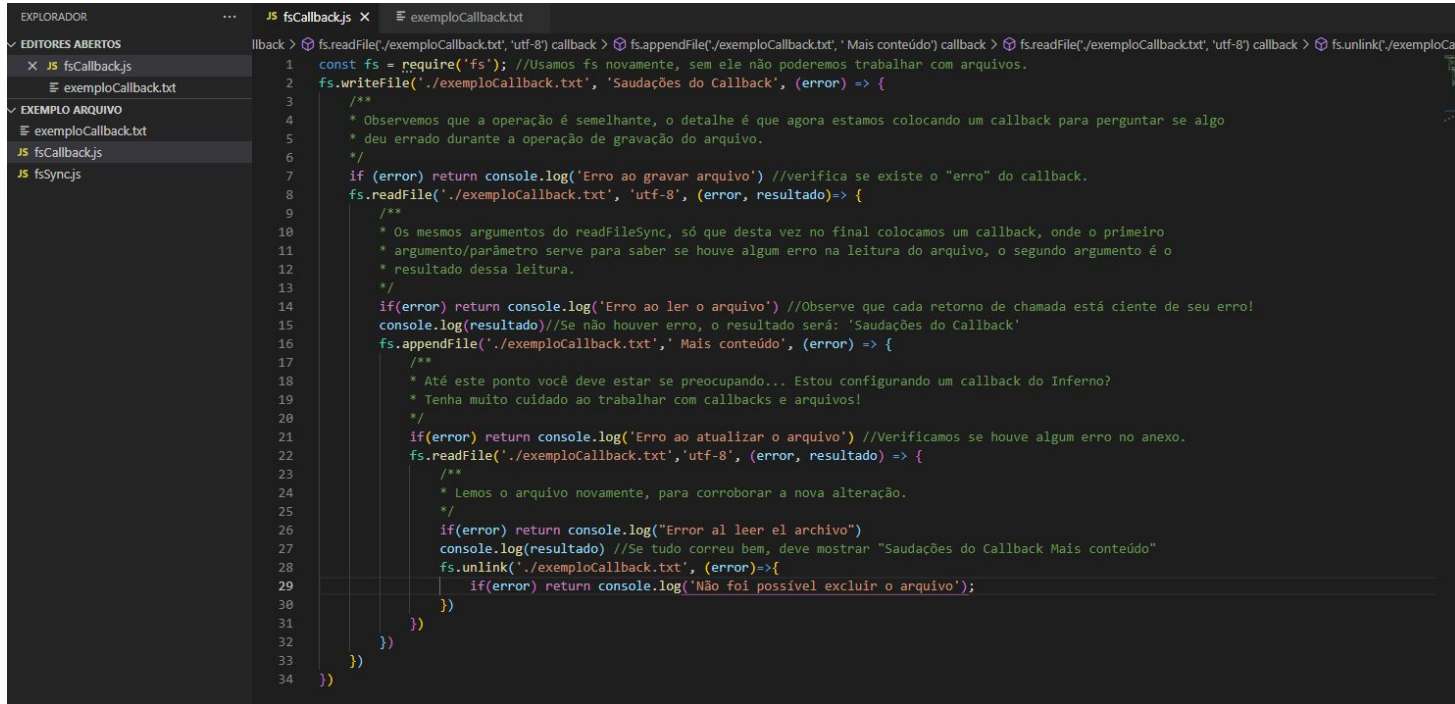
## 2 fs com callbacks

As principais operações que podemos fazer com fs com callbacks são:

- **writeFile** = Para gravar conteúdo em um arquivo. Se o arquivo não existir, ele o cria. Se existir, ele o sobrescreve. Apenas digitando, seu retorno de chamada lida apenas com: (erro)=>
- **readFile** = Para obter o conteúdo de um arquivo. Como ele solicita informações, seu retorno de chamada é da forma: (erro, resultado)=>
- **appendFile** = Para anexar conteúdo a um arquivo. Não é sobrescrito!, pois está apenas escrevendo, seu callback trata apenas: (erro)=>
- **unlink** = É o "excluir" dos arquivos. excluirá o arquivo inteiro, não apenas o conteúdo.
- Ao não retornar conteúdo, seu callback é apenas (erro) =>

# Exemplo de fs com callback

O mesmo procedimento do exemplo 1 será realizado, enfatizando os callbacks e como eles são tratados.



```
1 const fs = require('fs'); //Usamos fs novamente, sem ele não poderemos trabalhar com arquivos.
2 fs.writeFile('./exemploCallback.txt', 'Saudações do Callback', (error) => {
3     /**
4      * Observemos que a operação é semelhante, o detalhe é que agora estamos colocando um callback para perguntar se algo
5      * deu errado durante a operação de gravação do arquivo.
6      */
7     if (error) return console.log('Erro ao gravar arquivo') //verifica se existe o "erro" do callback.
8     fs.readFile('./exemploCallback.txt', 'utf-8', (error, resultado)=> {
9         /**
10          * Os mesmos argumentos do readFileSync, só que desta vez no final colocamos um callback, onde o primeiro
11          * argumento/parâmetro serve para saber se houve algum erro na leitura do arquivo, o segundo argumento é o
12          * resultado dessa leitura.
13          */
14         if(error) return console.log('Erro ao ler o arquivo') //Observe que cada retorno de chamada está ciente de seu erro!
15         console.log(resultado)//Se não houver erro, o resultado será: 'Saudações do Callback'
16         fs.appendFile('./exemploCallback.txt', ' Mais conteúdo', (error) => {
17             /**
18              * Até este ponto você deve estar se preocupando... Estou configurando um callback do Inferno?
19              * Tenha muito cuidado ao trabalhar com callbacks e arquivos!
20              */
21             if(error) return console.log('Erro ao atualizar o arquivo') //Verificamos se houve algum erro no anexo.
22             fs.readFile('./exemploCallback.txt','utf-8', (error, resultado) => {
23                 /**
24                  * Lemos o arquivo novamente, para corroborar a nova alteração.
25                  */
26                 if(error) return console.log("Error al leer el archivo")
27                 console.log(resultado) //Se tudo correu bem, deve mostrar "Saudações do Callback Mais conteúdo"
28                 fs.unlink('./exemploCallback.txt', (error)=>{
29                     if(error) return console.log('Não foi possível excluir o arquivo');
30                 })
31             })
32         })
33     })
34 })
```

## Importante

Lembre-se de que os **retornos de chamada são perigosos se você precisar encadear várias tarefas**. Se você precisar fazer operações de arquivo muito complexas, não use callbacks ou você acabará em um **CALLBACK HELL**👁️.

## Armazenar data e hora

Duração: 20 minutos

Pratique para revisar os conceitos de arquivos com callbacks

- ✓ Escreva um programa que crie um arquivo no qual você escreve a data e a hora atuais. Posteriormente, leia o arquivo e exiba o conteúdo pelo console.
- ✓ Use o módulo fs e suas operações de tipo callback.





# Break

5 minutos e voltamos!





# Break

10 minutos e voltamos!



**3**

**fs usando promessas**

## 3 fs com promessas

Já sabemos trabalhar com arquivos, já vimos como trabalhar com eles de forma assíncrona, agora vem o ponto mais valioso: **trabalhar com arquivos de forma assíncrona, com promessas**. Faremos isso com sua propriedade **fs.promises**.

Colocando nosso módulo **fs** com **.promises** estamos indicando que a **operação a ser realizada deve ser executada de forma assíncrona**, mas ao invés de manipulá-la com um callback, podemos fazê-lo com **.then** e **.catch**, ou com **async/await**.

## 3 fs com promessas

Os argumentos e a estrutura são quase idênticos ao síncrono, portanto suas principais operações serão:

- `fs.promises.writeFile` = Para gravar conteúdo em um arquivo. Se o arquivo não existir, ele o cria. Se existir, ele o sobrescreve.
- `fs.promises.readFile` = Para obter o conteúdo de um arquivo.
- `fs.promises.appendFile` = Para anexar conteúdo a um arquivo. Não é sobrescrito!
- `fs.promises.unlink` = É o "excluir" dos arquivos. Excluirá o arquivo inteiro, não apenas o conteúdo.

# Exemplo de fs com promises usando async/await

Será feito o mesmo procedimento dos exemplos 1 e 2, mas trabalhando com fs com seu submódulo de promessas. A implementação será com async/await

```
JS fsPromises.js > ...
1  const fs = require('fs') //Neste ponto, deve estar perfeitamente claro para nós como importar o módulo fileSystem
2  const operacoesAssincronas = async() => { //Observe que a função deve ser assíncrona pois trabalhamos com promessas.
3      //escrevemos um arquivo
4      await fs.promises.writeFile('./exemploPromessa.txt', 'Olá da promessa!') // (caminho e nome do arquivo, conteúdo)
5      //A utilização do módulo de promessas facilita a operação de forma a não exigir estar dentro de um callback.
6
7      let resultado = await fs.promises.readFile('./exemploPromessa.txt', 'utf-8') // (caminho e nome do arquivo, codificação)
8      console.log(resultado) // Veremos: "Olá da promessa!";
9
10     //Modificamos o arquivo
11     await fs.promises.appendFile('./exemploPromessa.txt', ' Conteúdo adicional') // (caminho e nome do arquivo, conteúdo)
12
13     //relemos o arquivo
14     resultado = await fs.promises.readFile('./exemploPromessa.txt', 'utf-8');
15     console.log(resultado); // Veremos: "Olá da promessa! Conteúdo adicional"
16
17     //finalmente, excluimos o arquivo
18     await fs.promises.unlink('./exemploPromessa.txt');
19 }
20
21 operacoesAssincronas();
22 //Temos um código muito mais limpo, muito mais simples e muito mais compreensível.
```

# Manipulando dados complexos com fs.promises

# Tratamento de dados complexos

Como você já pode imaginar, nem tudo é um arquivo .txt e, claro, nem tudo é uma simples string de texto. O que acontecerá quando quisermos salvar o conteúdo de uma variável, mesmo que seja um objeto? E se for uma correção? Normalmente, os arquivos com os quais costumamos trabalhar para armazenamento são arquivos **json**.

Para armazenar elementos mais complexos, contaremos com o elemento **JSON.stringify()** e **JSON. parser()**

[Documentação complementar sobre JSON](#)



# JSON.stringify

Uma vez que tenhamos o objeto que queremos salvar no arquivo, devemos lembrar que o arquivo não pode ser salvo apenas incorporando-o. **Precisamos convertê-lo para o formato json**, que é um formato padrão para salvar e enviar arquivos.

A sintaxe para fazer a conversão é:

```
JSON.stringify(objetoAConverter, replacer, '\t')
```

```
{
  "id": 1001,
  "type": "donut",
  "name": "Cake",
  "description": "https://www.jqueryscript.net",
  "price": 2.55,
  "available": { 2 items },
  "topping": [
    {
      "id": 5001,
      "type": "None"
    },
    {
      "id": 5002,
      "type": "Glazed"
    }
  ]
}
```

É assim que um JSON se parece. Muito semelhante a um objeto, **mas não é!**

# JSON.parse

Agora que entendemos como um objeto é convertido em JSON, fica claro mencionar que **JSON.parse** representa a operação oposta. Quando lemos um arquivo, o conteúdo não é manipulável, então **para recuperar o objeto que foi salvo e não apenas uma string que o representa, então temos que transformá-lo de volta**, isso é feito com JSON.parse

Sua sintaxe é:

```
JSON.parse(json_que_quero_transformar_em_objeto)
```

# Leitura e gravação de arquivos

Duração: 15 minutos

Escreva um programa executável em node.js que faça o seguinte:

- ✓ Abra um terminal no diretório de arquivos e execute o comando: `npm init -y`.
  - *Isso criará um arquivo especial (mais sobre isso mais tarde) chamado `package.json`*
- ✓ Leia o arquivo `package.json` e declare um objeto com o seguinte formato e dados:

```
const info = {  
  conteudoStr: (conteúdo do arquivo lido em formato string),  
  conteudoObj: (conteúdo do arquivo lido em formato de objeto),  
  size: (tamanho em bytes do arquivo)  
}
```

# Vantagens e desvantagens de usar arquivos





## Por que usá-los?

- Eles são **ótimos para começar a aprender persistência**, pois são muito fáceis de usar.
- Sendo nativo do node js, **não precisamos fazer instalações externas.**
- É muito **fácil de manipular dentro ou fora do nosso programa**, além de ser transferível.



## Desvantagens

- À medida que as informações forem crescendo, perceberemos que, **para modificar uma única coisa, precisamos ler o arquivo inteiro**, o que consome recursos importantes.
- Semelhante ao ponto anterior, **uma vez que um ponto de dados no arquivo foi modificado, tenho que reescrever completamente o arquivo**, o que é um processo desnecessário e complicado quando as informações são grandes.
- No final, **pode ser perigoso ter todas as informações em um arquivo facilmente removível** com um arrastar e soltar para outra pasta.

# Para pensar

Depois de ver as vantagens e desvantagens de seu uso, é hora de nos perguntarmos:

Ele é usado em desenvolvimento web ou é algo mais típico apenas de sistemas operacionais?

Compartilhe no chat suas opiniões

## Hands on Lab

Duração: 15 minutos

**Nesta instância da aula, vamos nos aprofundar na criação de promessas e no uso de `async await` com um exercício prático.**

**De que maneira?**

- ✓ O professor demonstrará como fazer e você poderá replicar em seu computador. Se surgirem dúvidas, você pode compartilhá-las para resolvê-las junto com a ajuda dos tutores.



## Hands on Lab

Duração: 15 minutos

### Como fazemos?

- ✓ Será criada uma classe que permite gerenciar usuários usando fs.promises, ela deve ter apenas dois métodos: Criar um usuário e consultar os usuários salvos.
- ✓ O gerente deve estar em uma classe em um arquivo externo chamado ManagerUsers.js
- ✓ O método "Criar usuário" deve receber um objeto com os campos:
  - Nome
  - Sobrenome
  - Idade
  - Curso

## Hands on Lab

Duração: 15 minutos

### Como fazemos?

- ✓ O método deve salvar um usuário em um arquivo "Usuarios.json", eles devem ser salvos dentro de um array, pois irão trabalhar com vários usuários
- ✓ O método "ConsultarUsuarios" deve ser capaz de ler um arquivo Users.json e retornar o array correspondente a esses usuários

Perguntas?

# Como foi a aula?

1

**Que bom**

O que foi super legal na aula e podemos sempre trazer para as próximas?

2

**Que pena**

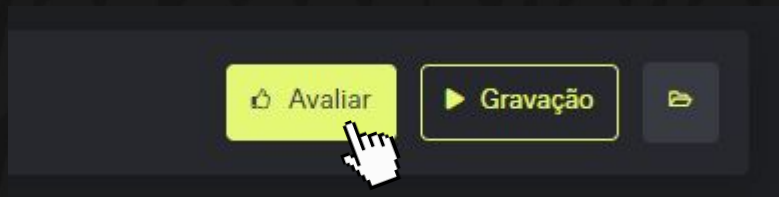
O que você acha que não funcionou bem e precisamos melhorar?

3

**Que tal**

Qual sugestão deveríamos tentar em próximas aulas?

# O que você achou da aula?



Seu feedback vale pontos para o Top 10!! 🕶️



## Deixe sua opinião!

1. Acesse a plataforma
2. Vá na aula do dia
3. Clique em **Avaliar**

DESAFIO OBRIGATÓRIO

# Gestão de arquivos em Javascript

Adicionamos `fileSystem` para alterar o modelo de persistência atual

# Vamos lembrar...

Com base no desafio entregável da aula 2:

- Estruturamos nossa primeira classe
- Adicionamos os métodos necessários à nossa classe para trabalhar com uma matriz de produtos

Agora, adicionamos `fileSystem` para alterar o modelo de persistência atual.

# Gestão de arquivos em Javascript

## Objetivo:

Crie uma classe chamada “ProductManager”, que permitirá trabalhar com vários produtos. Ele deve ser capaz de adicionar, consultar, modificar e excluir um produto e gerenciá-lo em persistência de arquivo (com base na entrega 1)

## Aspectos a incluir:

- A classe deve possuir uma variável `this.path`, que será inicializada a partir do construtor e deverá receber o caminho para trabalhar quando sua instância for gerada.

Você deve salvar os objetos no seguinte formato:

- `id` (deve ser incrementado automaticamente, não enviado no corpo)
- `title` (nome do produto)
- `description` (descrição do produto)
- `price` (preço)
- `thumbnail` (caminho da imagem)
- `code` (código identificador)
- `stock` (número de peças disponíveis)



## DESAFIO OBRIGATÓRIO

# Gestão de arquivos em Javascript

### Aspectos a incluir:

- Ele deve ter um método `addProduct` que deve receber um objeto com o formato especificado anteriormente, atribuir a ele um id auto-incrementado e salvá-lo no array (lembre-se de sempre salvá-lo como um array no arquivo).
- Você deve ter um método `getProducts`, que deve ler o arquivo de produtos e retornar todos os produtos em formato de array.
- Deve possuir um método `getProductById`, que deve receber um id, e após a leitura do arquivo, deve buscar o produto com o id especificado e retorná-lo em formato de objeto.
- Deve ter um método `updateProduct`, que deve receber o id do produto a ser atualizado, assim como o campo a ser atualizado (pode ser o objeto inteiro, como em um BD), e deve atualizar o produto que possui esse id em o arquivo. SEU ID NÃO DEVE SER EXCLUÍDO
- Você deve ter um método `deleteProduct`, que deve receber um id e deve deletar o produto com esse id no arquivo.

### Formato de entrega:

- Arquivo Javascript chamado `ProductManager.js`

# Resumo

## da aula de hoje

- ✓ Usar programação síncrona e assíncrona e aplicá-la no uso de arquivos
- ✓ Conhecer o módulo nativo do Nodejs para interagir com arquivos
- ✓ Conhecer o uso de arquivos com callbacks e promessas
- ✓ Conhecer as vantagens e desvantagens do FileSystem, bem como exemplos práticos.



**Você quer saber mais?**  
**Deixamos material**  
**extra da aula**

[CLIQUE AQUI](#)



**Obrigado por estudar  
conosco!**