PROMISES E ASYNC AWAIT

(AULA 20)

CURSO BÁSICO DE PROGRAMAÇÃO COM JAVASCRIPT

MAYARA MARQUES

mmrosatab@gmail.com

JS

SUMÁRIO

- Operações assíncronas x operações síncronas
- O que são promises?
- Mão na massa

OPERAÇÕES SÍNCRONAS X OPERAÇÕES ASSÍNCRONAS

Ju é dona de uma pequena lanchonete de hambúrgueres no bairro. Ela é dedicada, simpática, e... faz tudo sozinha!

Sempre que um cliente chega, Ju:

1. Anota o pedido,
2. Vai para a chapa fazer o hambúrguer,
3. Enquanto o lanche assa, ela espera do lado, olhando o relógio,
4. Depois pega a bebida,
5. Recebe o pagamento,
6. E só então entrega tudo ao cliente.

Enquanto Ju cuida de um pedido, os outros clientes **vão chegando... e esperando... e esperando...**

OPERAÇÕES SÍNCRONAS X OPERAÇÕES ASSÍNCRONAS

Desse modo, a tarefa de atender um pedido de cliente é **bloqueante**, no sentido de que, até que Ju termine um atendimento de pedido, um cliente novo não poderá ser atendido.

Esse é o mundo **síncrono**: cada tarefa só começa depois que a anterior termina. Tudo é feito em **sequência**, e o tempo de espera pode ser grande.

OPERAÇÕES SÍNCRONAS X OPERAÇÕES ASSÍNCRONAS

Mas um dia, Ju decidiu contratar ajudantes!

Agora, enquanto ela anota o pedido,

Um ajudante já começa a fritar o hambúrguer,

Outro vai buscar a bebida,

E enquanto tudo acontece, Ju já está atendendo o próximo cliente.

Quando cada ajudante termina, entrega a parte dele para Ju, que junta tudo e entrega o pedido completo.

OPERAÇÕES SÍNCRONAS X OPERAÇÕES ASSÍNCRONAS

Esse é o mundo **assíncrono**: as tarefas acontecem ao mesmo tempo**, e Ju não precisa mais esperar uma tarefa terminar para começar a outra.**

OPERAÇÕES SÍNCRONAS X OPERAÇÕES ASSÍNCRONAS

Assim como vimos no contexto da Ju, operações semelhantes acontecem dentro da programação. Algumas operações podem ser bloqueantes, ou seja, uma operação seguinte só poderá ser executada depois que a atual for finalizada. Como também, podemos ter operações que são não bloqueantes, permitindo que outras operações sejam iniciadas enquanto a mesma não é finalizada.

Vejamos alguns exemplos:

OPERAÇÕES SÍNCRONAS

```
console.log("1. Início")
console.log("2. Meio")
console.log("3. Fim")
```

Veja que o código executa linha por linha, esperando cada uma terminar.

```
1. Início
2. Meio
3. Fim
```

OPERAÇÕES SÍNCRONAS

```
console.log("1. Início")

for(let i = 0; i < 100; i++){

    if(i === 50){
        console.log("2. Meio")
    }

}

console.log("3. Fim")
```

Aqui também temos um fluxo síncrono, onde o JavaScript executa cada linha somente após terminar a anterior.
Mesmo que o *for* execute rapidamente, ele bloqueia o restante do código até terminar.

```
1. Início
2. Meio
3. Fim
```

OPERAÇÕES ASSÍNCRONAS

A função **setTimeout** permite que uma ou mais instruções sejam executadas depois de um determinado tempo, em milissegundos.

```
console.log("1. Início")

setTimeout(() => {
  console.log("2. Meio")
}, 2000)

console.log("3. Fim")
```

O valor **2000** representa **2000 milissegundos**, ou seja, **2 segundos**.

```
1. Início
3. Fim
2. Meio
```

Perceba que o console.log("3. Fim") foi executado antes do console.log("2. Meio").
Isso aconteceu porque a função setTimeout não bloqueia o restante do código — ela agenda a execução para depois de 2 segundos e permite que o código continue rodando.

OPERAÇÕES ASSÍNCRONAS

Assim como acontece com o setTimeout, existem outras operações que também são assíncronas em JavaScript.

Exemplos:

1. **Requisições ao backend** (como chamadas de API)
2. **Leitura de arquivos**
3. **Acesso a banco de dados**

Essas operações podem levar tempo para serem concluídas, mas não precisamos que o restante do código espere por elas.

Em breve, veremos com mais detalhes como o JavaScript gerencia esse comportamento assíncrono.

PROMISES

As promises são objetos JavaScript que representam eventuais conclusões ou falhas de operações assíncronas. Elas funcionam com a mesma ideia de promessas da vida real.

Uma promise pode apresentar os seguintes estados:

- **pendente**: estado inicial, nem cumprido nem rejeitado.
- **cumprida**: significa que a operação foi concluída com sucesso.
- **rejeitada**: significa que a operação falhou.

PROMISES

Para criar uma promise, precisamos criar uma instância por meio da classe Promise.

```
const minhaPromise = new Promise((resolve, reject) => {
  const sucesso = true

  if (sucesso) {
    resolve("Deu certo!")
  } else {
    reject("Algo deu errado...")
  }
})
```

resolve → função chamada se deu tudo certo.

reject → função chamada se deu erro.

A lógica dentro da Promise pode ser um setTimeout, fetch, ou qualquer coisa assíncrona.

PROMISES

Consumindo uma Promise com .then()

```
minhaPromise
  .then((resultado) => {
    console.log("Sucesso:", resultado)
  })
  .catch((erro) => {
    console.error("Erro:", erro)
  })
```

.then() é executado se a Promise for resolvida com sucesso.

.catch() é executado se a Promise for rejeitada com erro.

PROMISES

Consumindo uma Promise com .then()

```
const esperar2Segundos = new Promise((resolve) => {
  setTimeout(() => {
    resolve("Passaram-se 2 segundos!")
  }, 2000)
})

esperar2Segundos.then((mensagem) => {
  console.log(mensagem) // "Passaram-se 2 segundos!"
})
```

A Promise espera 2 segundos e depois é resolvida.

PROMISES

Promise rejeitada

```
const promessaComErro = new Promise((_, reject) => {
  reject("Ocorreu um erro!")
})

promessaComErro
  .then((res) => {
    console.log("Sucesso:", res)
  })
  .catch((erro) => {
    console.error("Erro:", erro)
  })
```

PROMISES

Como mencionado, existem várias operações assíncronas que podem ser feitas em nosso código.

Vejamos, mais de perto, como poderíamos solicitar uma informação a um outro sistema por meio de uma requisição HTTP.

API

Uma **API (Application Programming Interface, ou Interface de Programação de Aplicações)** é um conjunto de regras e definições que permite que diferentes softwares “conversem” entre si.

Ela funciona como um cardápio de um restaurante: você não precisa saber como o prato é preparado na cozinha, apenas pede pelo nome e recebe o resultado. Na API, você faz uma requisição pedindo algo e recebe uma resposta com os dados ou ação solicitada.

API

Principais pontos sobre API

- **Interface**: Ela define como você pode interagir com um sistema — quais funções existem, quais dados precisa enviar e qual formato será recebido.
- **Abstração**: Você não precisa conhecer o funcionamento interno, apenas seguir as regras documentadas.
- **Comunicação**: Geralmente, as APIs permitem comunicação entre sistemas diferentes (ex.: um site e um servidor remoto).

Tipos de API mais comuns

- APIs Web: Usadas para comunicação via internet, normalmente utilizando HTTP/HTTPS (ex.: API do GitHub, API do Google Maps).
- APIs de bibliotecas: Funções e métodos prontos que você pode chamar dentro do seu código.
- APIs de sistema operacional: Funções para interagir com recursos do sistema, como arquivos e rede.

FETCH

O Fetch é uma **API nativa** para fazer requisições HTTP, de forma **assíncrona**, para servidores e buscar ou enviar dados pela internet — por exemplo, acessar uma API, enviar dados de um formulário, buscar informações de um banco de dados etc.

Um detalhe importante a destacar é que o fetch retorna uma ***promise***.

FETCH

O fetch utiliza a sintaxe a seguir para realizar chamadas:

FETCH

O fetch utiliza a sintaxe a seguir para realizar chamadas:

Como o fetch retorna uma promise, podemos utilizar o **.then** e **.catch** para lidar com seus resultados.

```
fetch('https://api.exemplo.com/dados')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Erro:', error))
```

```
minhaPromise
  .then((resultado) => {
    console.log("Sucesso:", resultado)
  })
  .catch((erro) => {
    console.error("Erro:", erro)
  })
```

SOLICITADO DADOS AO VIA CEP COM O FETCH

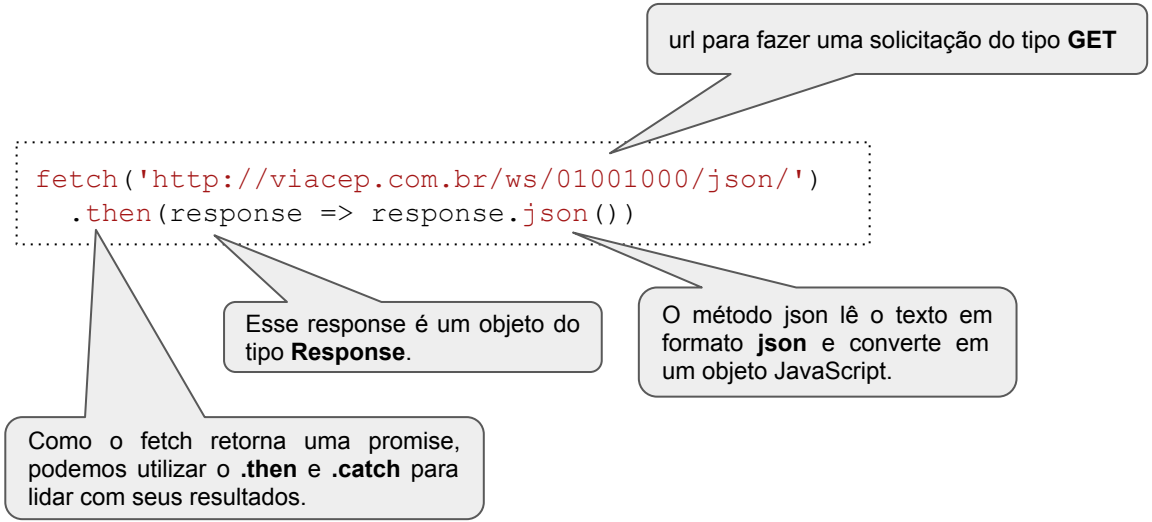
O viacep API é um serviço que permite buscar informações sobre um endereço postal.

https://viacep.com.br/

SOLICITADO DADOS AO VIA CEP COM O FETCH

Podemos utilizar as urls fornecidas pelo Viacep para obter informações sobre um determinado endereço.

SOLICITADO DADOS AO VIA CEP COM O FETCH



Quando você usa fetch no JavaScript, ele faz uma requisição HTTP e retorna uma Promise que resolve para um objeto **Response**. Esse objeto **Response** representa a resposta da requisição, **mas não é o dado em si** — é uma "**embalagem**" que contém várias informações, como os headers, status, e o corpo da resposta (que pode estar em vários formatos).

SOLICITADO DADOS AO VIA CEP COM O FETCH

O método .json() do objeto **Response** serve para:

- Ler o corpo da resposta (response body) que está em formato JSON.
- Converter esse corpo JSON em um objeto JavaScript que você pode manipular diretamente.

Ou seja, .json() é uma função assíncrona que lê o texto JSON da resposta e transforma esse texto em um objeto JavaScript, retornando outra Promise que resolve com esse objeto.

SOLICITADO DADOS AO VIA CEP COM O FETCH

Por que não podemos usar o dado diretamente?

Porque o fetch não sabe automaticamente como você quer tratar o corpo da resposta — ele pode ser JSON, texto, blob, etc. Você precisa especificar explicitamente o formato que quer extrair, por exemplo:

- response.json() para JSON
- response.text() para texto puro
- response.blob() para arquivos binários

SOLICITADO DADOS AO VIA CEP COM O FETCH

```
fetch('http://viacep.com.br/ws/01001000/json/')
      .then(res => console.log(res))
```

Se fizermos um console.log no lugar do .json, poderemos notar o formato objeto **Response**, bem como a propriedade **body** de onde o conteúdo principal é extraído.

O status 200 indica que a solicitação foi realizada com sucesso.

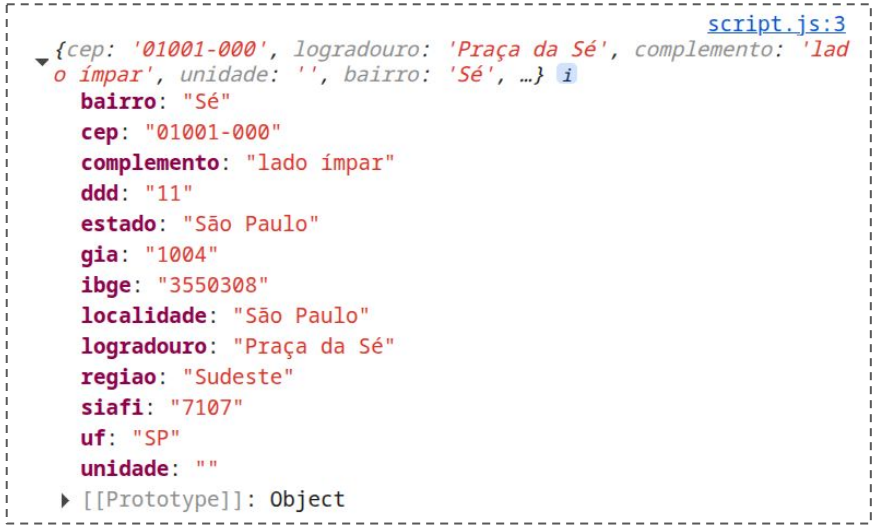
script.js:2

```
▼ Response i
    body: (...)
    bodyUsed: false
  ▶ headers: Headers {}
    ok: true
    redirected: true
    status: 200
    statusText: "OK"
    type: "cors"
    url: "https://viacep.com.br/ws/01001000/json/"
  ▶ [[Prototype]]: Response
```

SOLICITADO DADOS AO VIA CEP COM O FETCH

```
fetch('http://viacep.com.br/ws/01001000/json/')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Erro:', error))
```

Após o body da requisição estar convertido para o formato adequado, podemos acessar o dado propriamente dito no próximo encadeamento do .**then**.



SOLICITADO DADOS AO VIA CEP COM O FETCH

```
fetch('http://viacep.com.br/ws/01001000/json1/')
      .then(res => res.json())
      .then(data => console.log(data))
      .catch(err => console.error(err))
```

Simulando um erro passando uma url que não existe

```
Access to fetch at '                                          index.html:1
https://viacep.com.br/ws/01001000/json1/' (redirected from '
http://viacep.com.br/ws/01001000/json1/') from origin '
http://127.0.0.1:5500' has been blocked by CORS policy: No
'Access-Control-Allow-Origin' header is present on the requested
resource.

▶ GET https://viacep.com.br/ws/01001000/json1/      script.js:1
net::ERR_FAILED 400 (Bad Request)

▶ TypeError: Failed to fetch                          script.js:4
    at script.js:1:1

>
```

ASYNC/AWAIT

O ***async/await*** é uma forma mais limpa e legível de trabalhar com operações assíncronas.

O ***async*** é uma palavra-chave usada antes de uma função para indicar que ela vai trabalhar com código assíncrono e vai retornar uma Promise.

O await realiza pausa a execução da função até que a Promise seja resolvida (ou rejeitada). É como dizer: "Espere aqui até ter a resposta, depois continue".

ASYNC/AWAIT

```
async function buscarEndereco() {
  try {
    const response = await fetch('http://viacep.com.br/ws/01001000/json/')
    const data = await response.json()
    console.log(data)
  } catch (error) {
    console.error('Erro:', error)
  }
}

buscarEndereco()
```

O bloco ***try/catch*** é uma estrutura de tratamento de erros no JavaScript. Ela serve para testar um bloco de código (no try) e, se acontecer algum erro, capturar esse erro no catch sem que o programa quebre.

ASYNC/AWAIT

- O async/await é outra forma de escrever a mesma lógica que pode ser feita com o .then e .catch
- O async transforma a função em algo que retorna uma Promise.
- O await só funciona dentro de funções async.
- O código fica mais legível.
- O await não bloqueia o JavaScript inteiro, apenas a função onde ele está.

MÃO NA MASSA

MÃO NA MASSA

1. Crie uma função que retorne uma Promise que é resolvida com a mensagem "Processo concluído" após 2 segundos.

2. Crie uma função que retorne uma Promise que será rejeitada com a mensagem "Erro ao processar" após 3 segundos.

3. Crie uma função que retorna uma Promise com um número. No .then(), dobre o valor recebido, e no próximo .then() triplique o valor.

4. Crie uma função que retorna uma Promise que pode ser resolvida ou rejeitada aleatoriamente. Trate o erro com .catch()

5. Faça uma requisição do tipo GET para a API pública https://jsonplaceholder.typicode.com/posts e exiba os dados recebidos no console.

6. Faça uma requisição para a API pública https://jsonplaceholder.typicode.com/users e exiba apenas o nome (name) e o email (email) de cada usuário.

DESAFIO

🏆 Consulta de CEP

Crie uma aplicação que:

Possua um campo de texto para digitar um CEP.

Ao informar o CEP e acionar a busca (use um botão), faça uma requisição para a API do ViaCEP (https://viacep.com.br/ws/{CEP}/json/).

Exiba os dados retornados (logradouro, bairro, localidade, UF) na tela, logo abaixo do campo de input.

REFERÊNCIAS

- https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function