

# AULA 4/4

# JS: Requisições e Processamento Assíncrono

- Imagine que você está em um restaurante pedindo sua comida favorita. Você faz o pedido ao garçom, e ele vai até a cozinha para preparar seu prato.
- Enquanto isso, você não fica parado esperando, né? Você pode brincar no celular, conversar ou desenhar. Quando o prato fica pronto, o garçom traz para você. Isso é muito mais eficiente do que ficar parado só olhando para a cozinha.
- No mundo dos computadores e da programação, o código assíncrono funciona assim. Imagine que o "garçom" é o JavaScript e o "pedido de comida" é uma tarefa, como carregar uma foto de um site ou buscar informações de um servidor.

# JS: Requisições e Processamento Assíncrono

- Se o JavaScript ficasse parado esperando a comida (ou a tarefa) ficar pronta, o site inteiro ia travar e ninguém conseguiria fazer mais nada, como clicar em botões ou assistir a um vídeo.
- Com código assíncrono, o JavaScript pode continuar fazendo outras coisas enquanto espera a resposta do servidor ou que a imagem seja carregada. Assim que a resposta chega, ele pega e faz o que precisa, sem deixar tudo parado.
- É como ser um garçom inteligente que faz várias coisas ao mesmo tempo, ajudando todo mundo a receber o que precisa sem atrasos!

# JS: Requisições e Processamento Assíncrono

## HTML

```
<label for="quota">Number of primes:</label>
<input type="text" id="quota" name="quota" value="1000000" />

<button id="generate">Generate primes</button>
<button id="reload">Reload</button>

<div id="output"></div>
```

```
const quota = document.querySelector("#quota");
const output = document.querySelector("#output");

document.querySelector("#generate").addEventListener("click", () => {
  const primes = generatePrimes(quota.value);
  output.textContent = `Finished generating ${quota.value} primes!`;
});

document.querySelector("#reload").addEventListener("click", () => {
  document.location.reload();
});
```

```
const MAX_PRIME = 1000000;

function isPrime(n) {
  for (let i = 2; i <= Math.sqrt(n); i++) {
    if (n % i === 0) {
      return false;
    }
  }
  return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generatePrimes(quota) {
  const primes = [];
  while (primes.length < quota) {
    const candidate = random(MAX_PRIME);
    if (isPrime(candidate)) {
      primes.push(candidate);
    }
  }
  return primes;
}
```

# JS: Requisições e Processamento Assíncrono

- Qual o motivo do travamento no navegador ao tentar descobrir os números primos?
  - A razão para isso é que este programa em JavaScript é single-threaded (thread única).
  - Uma thread é uma sequência de instruções que um programa segue. Como o programa consiste em uma única thread, ele só pode fazer uma coisa de cada vez.
  - Então, se estiver esperando que uma chamada síncrona demorada retorne, ele não pode fazer mais nada.

# JS: Requisições e Processamento Assíncrono

- O que precisamos é de uma maneira para que nosso programa:
  - Inicie uma operação de longa duração chamando uma função.
  - Faça com que essa função inicie a operação e retorne imediatamente, para que o programa continue responsivo a outros eventos.
  - Execute a operação de forma que não bloqueie a thread principal, por exemplo, iniciando uma nova thread.
  - Nos notifique com o resultado da operação quando ela for concluída.
- É exatamente isso que as **funções assíncronas** nos permitem fazer.

# JS: Requisições e Processamento Assíncrono

HTML

```
<button id="xhr">Click to start request</button>  
<button id="reload">Reload</button>  
  
<pre readonly class="event-log"></pre>
```

JS

```
const log = document.querySelector(".event-log");  
  
document.querySelector("#xhr").addEventListener("click", () => {  
  log.textContent = "";  
  
  const xhr = new XMLHttpRequest();  
  
  xhr.addEventListener("loadend", () => {  
    log.textContent = `${log.textContent}Finished with status: ${xhr.status}`;  
  });  
  
  xhr.open(  
    "GET",  
    "https://raw.githubusercontent.com/mdn/content/main/files/en-us/_wikihistory.json",  
  );  
  xhr.send();  
  log.textContent = `${log.textContent}Started XHR request\n`;  
});  
  
document.querySelector("#reload").addEventListener("click", () => {  
  log.textContent = "";  
  document.location.reload();  
});
```

[https://developer.mozilla.org/en-US/docs/Learn\\_web\\_development/Extensions/Async\\_JS/Introducing](https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Async_JS/Introducing)

# JS: Requisições e Processamento Assíncrono

- Um event handler (manipulador de eventos) é um tipo específico de callback.
- Um callback é simplesmente uma função que é passada para outra função, com a expectativa de que o callback seja chamado no momento apropriado. Como acabamos de ver, os callbacks costumavam ser a principal maneira de implementar funções assíncronas em JavaScript.
- No entanto, o código baseado em callbacks pode se tornar difícil de entender quando o próprio callback precisa chamar funções que também aceitam callbacks. Essa é uma situação comum quando você precisa realizar uma operação que se divide em uma série de funções assíncronas.
- Por este motivo, APIs modernas não utilizam callbacks, ao invés disso, utilizam programação assíncrona com Promises. Tópico que iremos aprender a seguir :-)



# JS: Promises

- Promises são a base da programação assíncrona no JavaScript moderno.
- Uma promise é um objeto retornado por uma função assíncrona que representa o estado atual da operação.
- No momento em que a promise é retornada para quem chamou a função, a operação frequentemente ainda não está concluída, mas o objeto promise fornece métodos para lidar com o eventual sucesso ou falha da operação.
- Com uma API baseada em promises, a função assíncrona inicia a operação e retorna um objeto do tipo Promise.
- Você pode então anexar manipuladores a esse objeto promise, e esses manipuladores serão executados quando a operação for concluída com sucesso ou falhar.

# JS: Promises Exemplo de Uso

- Faremos uma solicitação HTTP para o servidor.
- Em uma solicitação HTTP, enviamos uma mensagem para um servidor remoto, e ele nos retorna uma resposta.
- Neste caso, enviaremos uma solicitação para obter um arquivo JSON do servidor.
- A API `fetch()` é a substituição moderna baseada em promises para o `XMLHttpRequest` que mostramos no exemplo anterior.

# JS: Promises Exemplo de Uso

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

console.log(fetchPromise);

fetchPromise.then((response) => {
  console.log(`Received response: ${response.status}`);
});

console.log("Started request...");
```

1. Chamando a API **fetch()** e atribuindo o valor retornado à variável **fetchPromise**.
2. Registrando no console o valor da variável **fetchPromise**. Isso deve exibir algo como: **Promise { <state>: "pending" }**, indicando que temos um objeto **Promise**, e ele possui um estado com valor "pending". O estado "pending" significa que a operação de fetch ainda está em andamento.
3. Passando uma função manipuladora para o método **then()** da **Promise**. Quando (e se) a operação de fetch for bem-sucedida, a promise chamará nosso manipulador, passando um objeto **Response**, que contém a resposta do servidor.
4. Registrando uma mensagem no console indicando que a solicitação foi iniciada.

# JS: Promises Exemplo de Uso

- Com a API `fetch()`, assim que você obtém um objeto `Response`, é necessário chamar outra função para obter os dados da resposta.
- Neste caso, queremos obter os dados da resposta como JSON, então chamamos o método `json()` do objeto `Response`.
- Acontece que o método `json()` também é assíncrono. Portanto, este é um caso em que precisamos chamar duas funções assíncronas sucessivas.

# JS: Promises Exemplo de Uso

```
const fetchPromise = fetch(  
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",  
);  
  
fetchPromise.then((response) => {  
  const jsonPromise = response.json();  
  jsonPromise.then((data) => {  
    console.log(data[0].name);  
  });  
});
```

- Neste exemplo, como antes, adicionamos um manipulador `then()` à promise retornada pelo `fetch()`. Mas, desta vez, nosso manipulador chama `response.json()` e, em seguida, passa um novo manipulador `then()` para a promise retornada por `response.json()`.
- Isso deve registrar no console "baked beans" (o nome do primeiro produto listado em "products.json").

# JS: Promises Exemplo de Uso

- Lembra quando dissemos que ao chamar um callback dentro de outro callback acabávamos criando níveis sucessivamente mais aninhados de código?
- E dissemos que esse "inferno dos callbacks" tornava nosso código difícil de entender? Isso não é exatamente a mesma coisa, só que com chamadas ao `then()`?
- De fato, é. Mas a característica elegante das promises é que o próprio `then()` retorna uma promise, que será resolvida com o resultado da função passada para ele.
- Isso significa que podemos (e certamente devemos) reescrever o código.

# JS: Promises Exemplo de Uso

```
const fetchPromise = fetch(  
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",  
);  
  
fetchPromise  
  .then((response) => response.json())  
  .then((data) => {  
    console.log(data[0].name);  
  }  
);
```

- Em vez de chamar o segundo `then()` dentro do manipulador do primeiro `then()`, podemos retornar a promise retornada por `json()` e chamar o segundo `then()` nesse valor de retorno.
- Isso é chamado de encadeamento de promises (promise chaining) e nos permite evitar níveis cada vez maiores de indentação quando precisamos fazer chamadas consecutivas a funções assíncronas.

# JS: Promises Exemplo de Uso

Por fim ...

```
const fetchPromise = fetch(
  "bad-scheme://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
store/products.json",
);

fetchPromise
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    console.log(data[0].name);
  })
  .catch((error) => {
    console.error(`Could not get products: ${error}`);
  });
```



# JS: Combinando Promises

```
const fetchPromise1 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);
const fetchPromise2 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found",
);
const fetchPromise3 = fetch(
  "https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json",
);

Promise.all([fetchPromise1, fetchPromise2, fetchPromise3])
  .then((responses) => {
    for (const response of responses) {
      console.log(`${response.url}: ${response.status}`);
    }
  })
  .catch((error) => {
    console.error(`Failed to fetch: ${error}`);
  });
```

# JS: Combinando Promises

- Às vezes, você pode precisar que qualquer uma de um conjunto de promises seja resolvida, sem se importar qual.
- Nesse caso, você deve usar o ***Promise.any()***. Isso é semelhante ao `Promise.all()`, exceto que ele é resolvido assim que qualquer uma das promises do array for resolvida, ou rejeitado se todas forem rejeitadas.

```
const fetchPromise1 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);
const fetchPromise2 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found",
);
const fetchPromise3 = fetch(
  "https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json",
);

Promise.any([fetchPromise1, fetchPromise2, fetchPromise3])
  .then((response) => {
    console.log(`${response.url}: ${response.status}`);
  })
  .catch((error) => {
    console.error(`Failed to fetch: ${error}`);
  });
```

# JS: Async Await

- Há uma forma ainda mais elegante de trabalhar com código assíncrono no Javascript, ao invés de usar a notação convencional para Promises, vamos utilizar o padrão Async e Await.
- A palavra-chave `async` oferece uma maneira mais simples de trabalhar com código assíncrono baseado em promises. Adicionar `async` no início de uma função a torna uma função assíncrona:

```
async function myFunction() {  
  // This is an async function  
}
```

# JS: Async Await

- Dentro de uma função async, você pode usar a palavra-chave await antes de uma chamada para uma função que retorna uma promise.
- Isso faz com que o código espere nesse ponto até que a promise seja resolvida, momento em que o valor resolvido da promise é tratado como um valor de retorno, ou o valor rejeitado é lançado como um erro.
- Isso permite que você escreva código que utiliza funções assíncronas, mas que se parece com código síncrono. Por exemplo, poderíamos usá-la para reescrever nosso exemplo de fetch.

# JS: Async Await

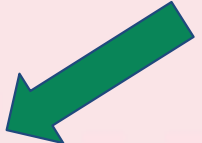
```
async function fetchProducts() {
  try {
    // after this line, our function will wait for the `fetch()` call to be settled
    // the `fetch()` call will either return a Response or throw an error
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    // after this line, our function will wait for the `response.json()` call to be settled
    // the `response.json()` call will either return the parsed JSON object or throw an error
    const data = await response.json();
    console.log(data[0].name);
  } catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

fetchProducts();
```

# JS: Async Await

## Exemplo errado!

```
async function fetchProducts() {  
  try {  
    const response = await fetch(  
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",  
    );  
    if (!response.ok) {  
      throw new Error(`HTTP error: ${response.status}`);  
    }  
    const data = await response.json();  
    return data;  
  } catch (error) {  
    console.error(`Could not get products: ${error}`);  
  }  
}  
  
const promise = fetchProducts();  
console.log(promise[0].name); // "promise" is a Promise object, so this will not work
```




# JS: Async Await

## Exemplo Correto!

```
async function fetchProducts() {
  const response = await fetch(
    "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
  );
  if (!response.ok) {
    throw new Error(`HTTP error: ${response.status}`);
  }
  const data = await response.json();
  return data;
}

const promise = fetchProducts();
promise
  .then((data) => {
    console.log(data[0].name);
  })
  .catch((error) => {
    console.error(`Could not get products: ${error}`);
  });
```





# JS: Async Await - Exemplo Prático Despertador

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <div>
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" size="4" value="Matilda" />
  </div>

  <div>
    <label for="delay">Delay:</label>
    <input type="text" id="delay" name="delay" size="4" value="1000" />
  </div>

  <button id="set-alarm">Set alarm</button>
  <div id="output"></div>

</body>
</html>
```



# JS: Async Await - Exemplo Prático Despertador

```
<script>
  const name = document.querySelector("#name");
  const delay = document.querySelector("#delay");
  const button = document.querySelector("#set-alarm");
  const output = document.querySelector("#output");

  async function alarm(person, delay) {
    if (delay < 0) {
      throw new Error("Alarm delay must not be negative");
    }
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve(`Wake up, ${person}!`);
      }, delay);
    });
  }

  button.addEventListener("click", async () => {
    try {
      const message = await alarm(name.value, Number(delay.value));
      output.textContent = message;
    } catch (error) {
      output.textContent = `Couldn't set alarm: ${error.message}`;
    }
  });

</script>
```

# Exemplos de uso

- <https://hp-api.onrender.com/>
- API IBGE
- API Prefeitura Quixadá

```

async function fetchAllCharacters() {
  try {
    const response = await fetch('https://hp-api.onrender.com/api/characters');
    if (!response.ok) {
      throw new Error(`Erro HTTP: ${response.status}`);
    }
    const characters = await response.json();
    console.log("Personagens:", characters);
    return characters;
  } catch (error) {
    console.error("Falha ao buscar personagens:", error);
  }
}

```

```

async function getDataa(){
  const allChars = await fetchAllCharacters();
  console.log("Primeiro personagem:", allChars[0].name);
}

```

```

getDataa();

```

```

async function fetchCharacterById(id) {
  try {
    const response = await fetch(`https://hp-api.onrender.com/api/character/${id}`);
    if (!response.ok) {
      throw new Error(`Personagem não encontrado (ID: ${id})`);
    }
    const character = await response.json();
    console.log("Personagem encontrado:", character);
    return character;
  } catch (error) {
    console.error("Erro:", error.message);
  }
}

async function getData(){
  const char = await fetchCharacterById("9e3f7ce4-b9a7-4244-b709-dae5c1f1d4a8"); // ID de
  Harry Potter
  console.log(char[0].name);
}

getData();

```

```
async function fetchGryffindorStudents() {  
  try {  
    const response = await  
fetch('https://hp-api.onrender.com/api/characters/house/gryffindor');  
    const students = await response.json();  
    console.log("Alunos da Grifinória:", students);  
  
    // Filtra apenas alunos vivos (opcional)  
    const livingStudents = students.filter(student => !student.alive);  
    console.log("Alunos vivos:", livingStudents);  
  } catch (error) {  
    console.error("Erro ao buscar alunos:", error);  
  }  
}  
  
// Uso  
fetchGryffindorStudents();
```

```
async function fetchSpells() {
  try {
    const response = await fetch('https://hp-api.onrender.com/api/spells');
    const spells = await response.json();

    console.log("Feitiços disponíveis:");
    spells.forEach(spell => {
      console.log(`- ${spell.name}: ${spell.description}`);
    });

    return spells;
  } catch (error) {
    console.error("Erro ao buscar feitiços:", error);
  }
}

// Uso
fetchSpells();
```

# JS: Async Await - Atividade

Crie o código JavaScript e HTML responsável por fazer o download de dados de uma API Web e exibir em algum componente HTML da página, por exemplo, um `<p>`.

URL com os dados: <https://hp-api.onrender.com/api/characters/staff>

# Dúvidas?



[www.shutterstock.com](http://www.shutterstock.com) · 744867163