



Assincronicidade, promessas e *fetch*

THIAGO DELGADO PINTO

versão: 2022.12.08



Licença Creative Commons 4

assincronicidade

adj. – 1. Que ocorre, ou não se processa, em sincronia com algum evento ou processo, ou segundo uma taxa constante em relação à determinada referência; 2. Máquina elétrica rotativa cuja velocidade de rotação não é proporcional à frequência da rede elétrica.

o JavaScript executa em uma **única thread**
no navegador, no NodeJS, no DenoJS, ...

há um **loop de eventos** para processar tarefas

operações síncronas devem executar rápido
pois são bloqueantes

por padrão é bloqueante

exemplos:

ler imagem

```
 bloqueie  
o conteúdo



# forma 1 – declarar scripts após conteúdo

exemplo:

...

```
<body>
```

```
 <h1>Bem-vindo(a)</h1>
```

```
 <!-- mais conteúdo da página aqui -->
```

```
 <!-- ... -->
```

```
 <script src="primeiro.js" ></script>
```

```
 <script src="segundo.js" ></script>
```

```
</body>
```

## forma 2 – usar o atributo defer

```
<head>
```

```
 <script src="grande.js" defer ></script>
```

```
 <script src="pequeno.js" defer ></script>
```

```
</head>
```

```
<body>
```

```
 <h1>Bem-vindo(a)</h1>
```

```
 <!-- mais conteúdo da página aqui -->
```

```
</body>
```

Vai processar o DOM, depois "grande.js" e depois "pequeno.js", mesmo que "pequeno.js" seja baixado primeiro que "grande.js".

## um script com o atributo **defer**...

1. executa após o DOM estar **pronto**  
logo **antes** do evento **DOMContentLoaded**
2. respeita a **ordem de declaração** dos scripts

obs.: atributo **defer** só funciona junto com **src**

atributo **async**

# atributo `async`

faz o **carregamento** do script **não bloquear** a página,  
mas sua **execução pode ser bloqueante**

script irá executar imediatamente após ser carregado  
podendo bloquear a página

**não respeita** ordem de declaração de scripts

útil para scripts externos independentes  
ex. contadores, anúncios

# usando atributo `async` – exemplo

```
<head>
```

```
 <script src="counter.js" async ></script>
```

```
 <script src="ad.js" async > </script>
```

```
</head>
```

```
<body>
```

```
 <h1>Bem-vindo(a)</h1>
```

```
 <!-- mais conteúdo da página aqui -->
```

```
 ...
```

```
</body>
```

portanto...

use o atributo **defer** para executar scripts em ordem, após o carregamento do DOM

use o atributo **async** para executar scripts pequenos e de forma independente

promessas



um modelo para processamento assíncrono

proposto por Friedman & Wise (1976)

também é chamado de "*eventual*" – Hibbard (1976)

é fortemente ligado ao conceito de *futuros*

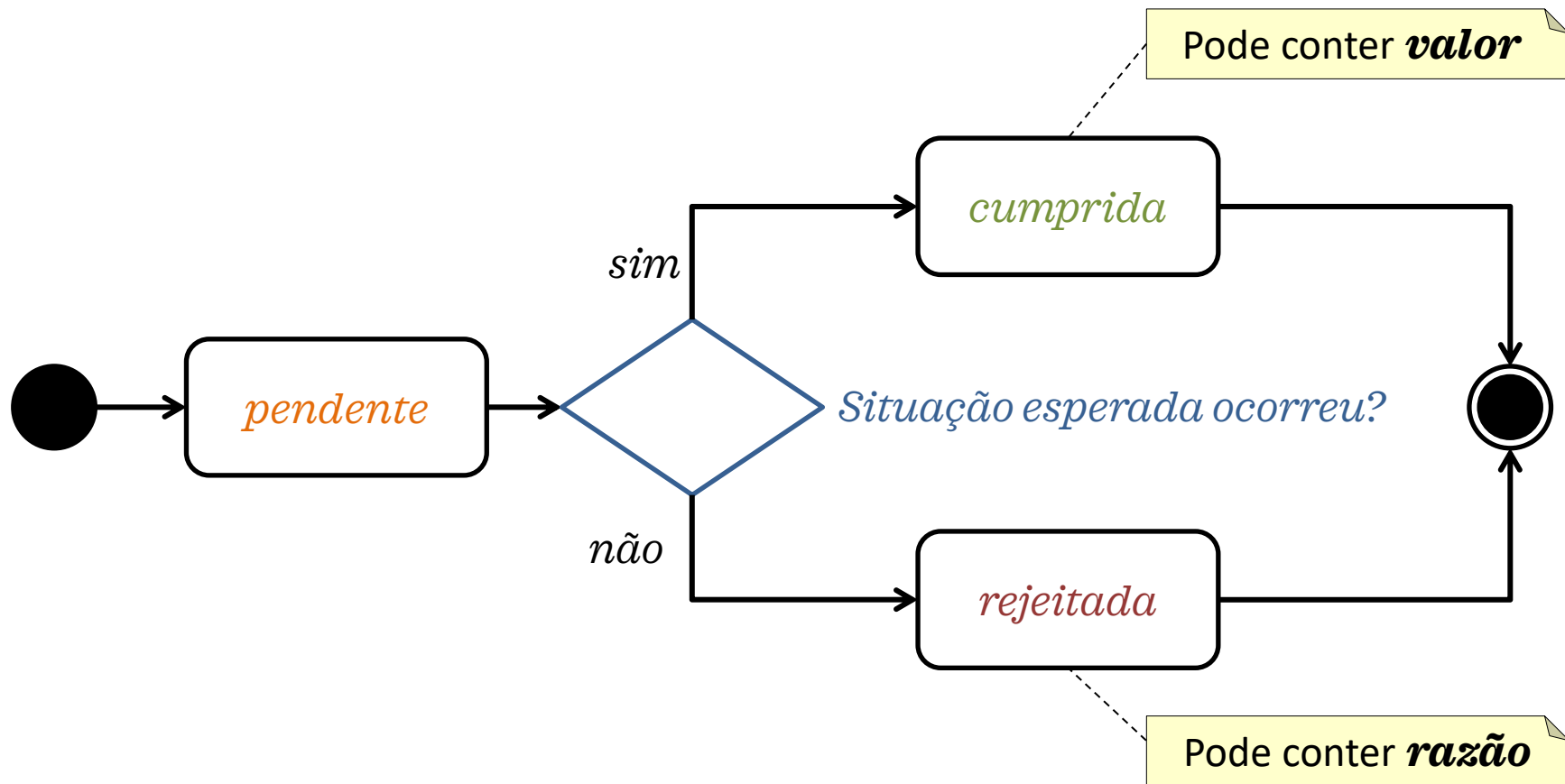
Baker & Hewitt (1977)

FRIEDMAN, Daniel; WISE, David. *The Impact of Applicative Programming on Multiprocessing*. International Conference on Parallel Processing. pp. 263–272. 1976.

HIBBARD, Peter. *Parallel Processing Facilities*. New Directions in Algorithmic Languages, IRIA, 1976.

BAKER, Henry; Carl Hewitt (1977). [The Incremental Garbage Collection of Processes](#). Proceedings of the Symposium on Artificial Intelligence Programming Languages ACM SIGPLAN Notices 12, 8. pp. 55–59. 1977.

*representa a possibilidade de um valor  
estar disponível.*



criado pela comunidade do JavaScript  
após ver várias implementações de Promessa surgirem...

<https://promisesaplus.com/>

provê uma **interface simples**

ganhou popularidade e [várias implementações](#)

bibliotecas conhecidas passaram a usá-la  
ex. jQuery, YUI, Dojo Toolkit



se tornou parte da especificação do EcmaScript  
a partir do ES6 (2015)

ganhou versões para outras linguagens  
ex. PHP, Python, Swift, C#, Java

# classe Promise – sintaxe *à la* TS

```
class Promise {
 private state: "pending" | "fulfilled" | "rejected";
 constructor(
 (resolve?: (value?: any) => void, reject?: (reason?: any) => void)
);
 then(
 onFulfilled: (value?: any) => void,
 onRejected?: (reason?: any) => void
): Promise;
 catch(onRejected: (reason?: any) => void): Promise;
 static resolve(value?: any): Promise;
 static reject(reason?: any): Promise;
 static race(promises: Iterable): Promise; // primeira (res. ou rej.)
 static all(promises: Iterable): Promise; // todas, se resolverem
 static allSettled(promises: Iterable): Promise; // todas sempre
}
```

*se um método retorna uma promessa,  
é **possível** que ele retorne um valor.*

*só se ele **cumprir** a promessa!*

do *cumprir* a promessa,  
o *valor* será passado para um  
*callback* definido no método *then*,  
que é assíncrono.



## conceito 2 – exemplo

```
function carregarFrutas() {
 return Promise.resolve(['Maçã', 'Laranja', 'Goiaba']);
}
```

```
const promessa = carregarFrutas();
promessa.then((frutas) => {
 console.log('As frutas são: ', frutas);
});
console.log('FRUTAS');
```

*ao **rejeitar** a promessa,  
a **razão** será passada para um  
**callback** definido no método **catch**,  
que é assíncrono.*

## conceito 3 – exemplo

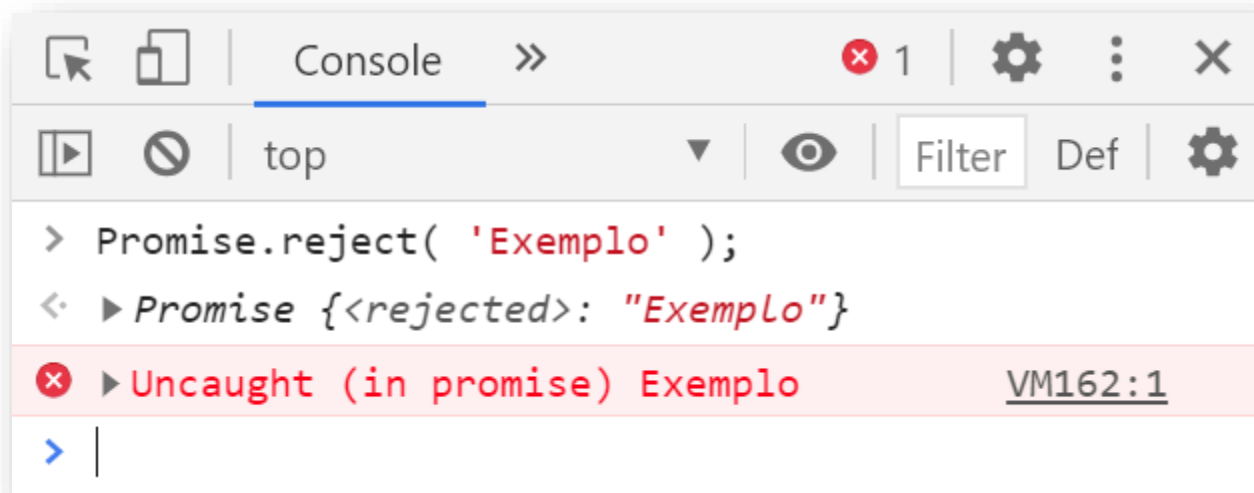
```
function carregarFrutas() {
 if (Math.random() > 0.5) {
 return Promise.resolve(['Maçã', 'Laranja', 'Goiaba']);
 }
 return Promise.reject('As frutas hoje não estão boas.');
}
```

```
const promessa = carregarFrutas();
promessa.then((frutas) => {
 console.log('As frutas são: ', frutas);
});
promessa.catch((razao) => {
 console.log('Erro ao carregar frutas: ', razao);
});
```

*Ao **rejeitar** uma promessa,  
uma **exceção** é lançada.*

*Se um **catch** não for declarado,  
a exceção fica **sem ser tratada**.*

# conceito 4 – exemplo



*Retornar uma Promessa **não garante** que o método se torne **assíncrono**.*

## conceito 5 – exemplo

```
function carregarArquivoGrande() {
 const conteudo = lerArquivoMuitoGrande(); // Síncrono
 if (! conteudo) {
 return Promise.reject('Erro ao carregar o arquivo.');
 }
 return Promise.resolve(conteudo);
}
```

```
const promessa = carregarArquivoGrande(); // Aguarda! ☹️
promessa.then((conteudo) => { console.log(conteudo); });
promessa.catch((razao) => { console.log(razao); });
```

*Um comportamento **encapsulado** em uma **Promessa** é executado **assincronamente**.*



## conceito 6 – exemplo

```
function carregarArquivoGrande() {
 return new Promise((resolve, reject) => {
 const conteudo = lerArquivoMuitoGrande();
 if (! conteudo) {
 return reject('Erro ao carregar o arquivo.');
 }
 return resolve(conteudo);
 });
}
```

```
const promessa = carregarArquivoGrande(); // Não aguarda! 😊
promessa.then((conteudo) => { console.log(conteudo); });
promessa.catch((razao) => { console.log(razao); });
```

*O construtor de Promise invocará o método recebido de forma **assíncrona**.*

*Logo, o comportamento **síncrono** de `carregarArquivoGrande()` é apenas o de criar e retornar a promessa.*

*Por padrão, o navegador roda o código em apenas **uma única thread**.*

*Logo, no momento em que `lerArquivoMuitoGrande()` executar, a thread provavelmente irá **bloquear**.*

*Para executar em **paralelo**, seria necessário usar **web workers**.*

## exercício 1

Crie uma função com comportamento assíncrono que retorne o número 100 após 3 segundos. Use *setTimeout* na solução.

1. Executa o seu *callback*;
2. Se ele lançar **exceção**, cria uma Promessa **rejeitada** com a exceção.
3. Senão, cria uma Promessa **cumprida** contendo o **retorno** do seu *callback*.

# sobre o `then` – exemplo com retorno

```
const promessa1 = Promise.resolve(10);
```

```
const promessa2 = promessa1.then(valor => {
 console.log('Recebi ', valor);
 return 20;
});
```

```
promessa2.then(valor => {
 console.log('Recebi ', valor);
 return 30;
});
```

# sobre o `then` – exemplo simplificado com retorno

```
const promessa = Promise.resolve(10);
```

```
promessa.then(valor => {
 console.log('Recebi ', valor);
 return 20;
}).then(valor => {
 console.log('Recebi ', valor);
 return 30;
});
```

# sobre o `then` – exemplo com exceção

```
function f1() { return Promise.resolve(10); }
```

```
f1().then(valor => {
 console.log('Recebi ', valor);
 return 20;
}).then(valor => {
 console.log('Recebi ', valor);
 if (valor < 50) { throw new Error('Menor que 50'); }
 return 100;
}).then(valor => {
 console.log('Recebi ', valor);
 return 200;
}).catch(razao => {
 console.log('Erro: ', razao.message);
});
```



**Não será executado!**



## sobre o `catch`

1. Captura uma exceção de qualquer **then** anterior.
2. Executa o seu *callback*;
3. Se ele lançar **exceção**, cria uma Promessa **rejeitada** com a exceção.
4. Senão, cria uma Promessa **cumprida** contendo o **retorno** do seu *callback*.

# sobre o catch – exemplo

```
const promessa = Promise.resolve(10);
```

```
promessa.then(valor => {
 if (valor < 50) { throw new Error('Menor que 50'); }
 return 20;
}).catch(razao => {
 console.log('Erro: ', razao.message);
 return 30;
}).then(valor => {
 console.log(valor);
})
```

1. coloca as promessas recebidas para executar
2. retorna a primeira que **cumprir** ou **rejeitar**.

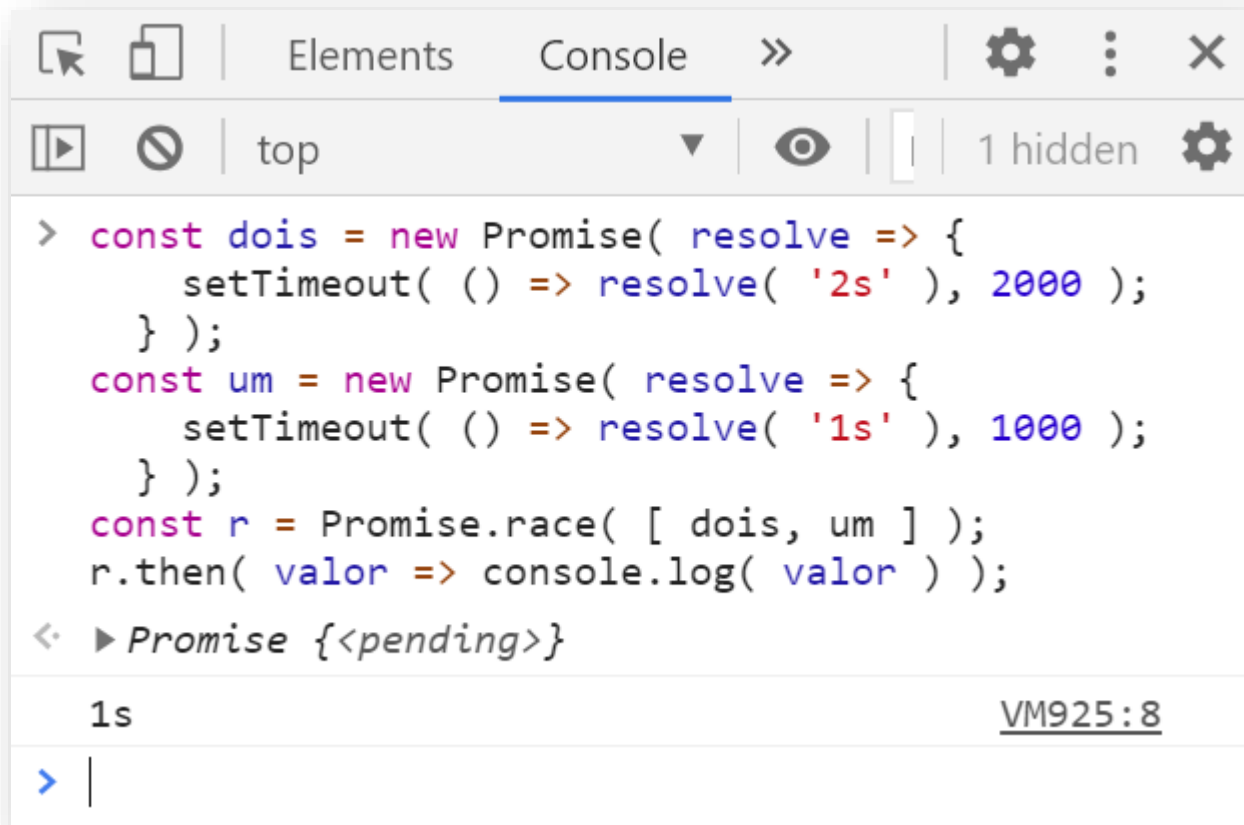
## sobre o race – exemplo

```
const dois = new Promise(resolve => {
 setTimeout(() => resolve('2s'), 2000);
});
```

```
const um = new Promise(resolve => {
 setTimeout(() => resolve('1s'), 1000);
});
```

```
const r = Promise.race([dois, um]);
r.then(valor => console.log(valor)); // 1s
```

# sobre o race – execução exemplo



The screenshot shows a web browser's developer console with the 'Console' tab selected. The code being executed is as follows:

```
> const dois = new Promise(resolve => {
 setTimeout(() => resolve('2s'), 2000);
 });
const um = new Promise(resolve => {
 setTimeout(() => resolve('1s'), 1000);
 });
const r = Promise.race([dois, um]);
r.then(valor => console.log(valor));
```

The execution result is displayed below the code:

```
< ▶ Promise {<pending>}
```

Below the result, the text '1s' is shown, indicating the time taken for the first promise to resolve. The console also shows 'VM925:8' as the source of the code.

## sobre o `all`

1. coloca as promessas recebidas para executar
2. aguarda **todas cumprirem** ou **uma rejeitar**
3. se houver **rejeição**, retorna a promessa rejeitada para ser tratada no **then**, em seu **segundo argumento**
4. se todas **cumprirem**, retorna uma lista com todos os retornos, na ordem das promessas

## sobre o `all` – exemplo 1

```
const dois = new Promise(resolve => {
 setTimeout(() => resolve('2s'), 2000);
});
```

```
const um = new Promise(resolve => {
 setTimeout(() => resolve('1s'), 1000);
});
```

```
const r = Promise.all([dois, um]);
r.then(valor => console.log(valor)); //['2s','1s']
```

## sobre o `all` – exemplo 2

```
const dois = new Promise(resolve => {
 setTimeout(() => resolve('2s'), 2000);
});
const um = new Promise(resolve => {
 setTimeout(() => resolve('1s'), 1000);
});
const tres = new Promise((resolve, reject) => {
 setTimeout(() => reject('Ops!'), 500);
});

const r = Promise.all([dois, um, tres]);
r.then(
 valor => console.log(valor),
 razao => console.log('Erro: ', razao));
```



## sobre o `all` – exemplo 2 alternativo

```
const dois = new Promise(resolve => {
 setTimeout(() => resolve('2s'), 2000);
});
```

```
const um = new Promise(resolve => {
 setTimeout(() => resolve('1s'), 1000);
});
```

```
const tres = new Promise((resolve, reject) => {
 setTimeout(() => reject('Ops!'), 500);
});
```

```
const r = Promise.all([dois, um, tres]);
r.then(valor => console.log(valor), razao => {});
r.catch(razao => console.log('Erro: ', razao));
```

Vazio



Agora o `catch()` consegue capturar o erro

## exercício 2

Simule uma corrida com 5 veículos, representados como Promessas com duração entre 2 e 5 segundos, escolhidos aleatoriamente. Dado que o trajeto percorrido por todos é o mesmo, o veículo considerado mais rápido é aquele que concluir o trajeto em menor tempo. Sendo assim, simule a corrida e informe o veículo mais rápido.

## sobre o `allSettled`

1. coloca as promessas recebidas para executar **até o fim**
2. coleta todos os **resultados como objetos** com a propriedade `status`  
que assume "**fulfilled**" (cumprida) ou "**rejected**" (rejeitada)
3. objetos com o status "**fulfilled**" têm a propriedade "`value`" (valor), enquanto os com "**rejected**" têm a propriedade "`reason`" (motivo)

# sobre o `allSettle` – exemplo

```
const dois = new Promise(resolve => {
 setTimeout(() => resolve('2s'), 2000);
});
const um = new Promise(resolve => {
 setTimeout(() => resolve('1s'), 1000);
});
const tres = new Promise((resolve, reject) => {
 setTimeout(() => reject('Ops!'), 500);
});
```

```
const p = Promise.allSettle([dois, um, tres]);
p.then(values => { console.log(values); });
// [
// { status: "fulfilled", value: "2s" },
// { status: "fulfilled", value: "1s" },
// { status: "rejected", reason: "Ops!" },
//]
```



**Resultado mantém ordem das promessas.**

*fetch*

# introdução

é um método para realizar requisições HTTP

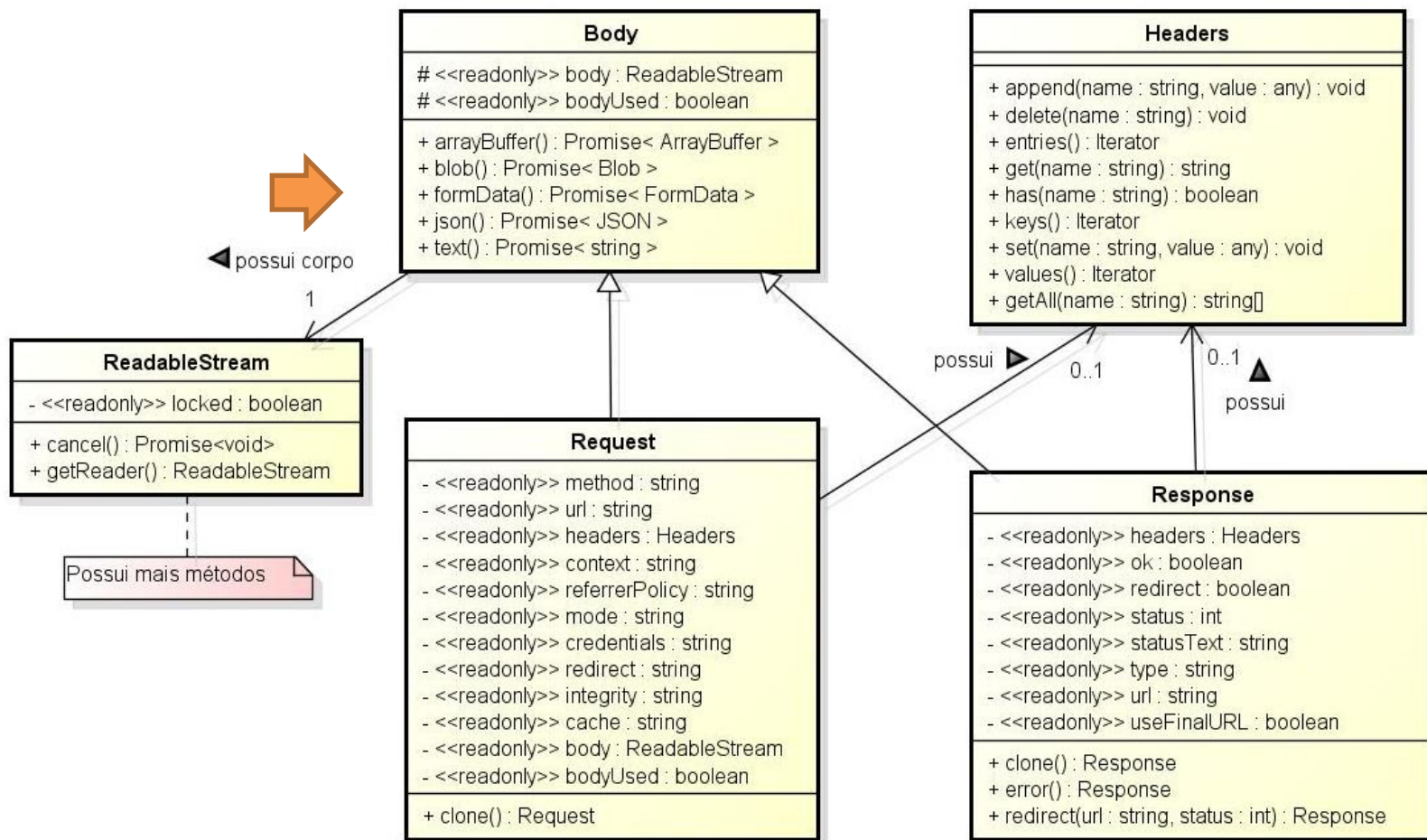
substitui **XMLHttpRequest** (XHR)

mais seguro

ex. controla diretivas de [Content Security Policy](#) (CSP)

disponível para Web Workers

# algumas classes relacionadas



# sintaxe

```
fetch(resource: string | Request,
 init?: {
 method?: string, // ex. "GET", "POST", "DELETE", "PUT"
 headers?: string | Headers,
 body?: string | FormData | Blob | BufferSource | URLSearchParams |
ReadableStream,
 mode?: string, // ex. "cors", "no-cors", "same-origin"
 credential?: string | FederatedCredential | PasswordCredential,
 cache?: string, // ex: "default", "no-cache"
 redirect?: string, // ex: "follow", "manual"
 referrer?: string,
 referrerPolicy?: string, // ex: "same-origin"
 integrity?: string, // hash
 keepalive?: boolean,
 signal?: AbortSignal
 }): Promise< Response >;
```



*veremos apenas o uso mais **básico** de fetch*

# exemplo – obtendo um recurso JSON

```
fetch('https://jsonplaceholder.typicode.com/todos/1',
 { headers: { 'Accept': 'application/json' }, mode: 'cors' })
 .then(response => {
 if (response.status >= 400) {
 throw new Error('Erro ' + response.status);
 }
 return response.json();
 })
 .then(json => console.log(json))
 .catch(err => console.error(err));
```

*fetch não rejeita automaticamente  
respostas HTTP com status  $\geq 400$*

## exemplo – obtendo um recurso textual

```
fetch('http://localhost/arquivo.txt', { mode: 'cors' })
 .then(response => {
 if (response.status >= 400) {
 throw new Error('Erro ' + response.status);
 }
 return response.text();
 })
 .then(txt => console.log(txt))
 .catch(err => console.error(err));
```

# exemplo – obtendo uma imagem

```
fetch(
 'http://www.cefet-rj.br/arquivos_download/logo_cefet__home_site.jpg',
 { mode: 'cors' }
)
 .then(response => {
 if (response.status >= 400) {
 throw new Error('Erro ' + response.status);
 }
 return response.blob();
 })
 .then(blob => {
 const objURL = URL.createObjectURL(blob);
 minhaImagem.src = objURL;
 })
 .catch(err => console.error(err));
```


# enviando dados de um formulário – modo 1

```
// Envia como application/x-www-form-urlencoded
```



```
const obj = {
 'nome' : document.getElementById('nome').value,
 'login' : document.getElementById('login').value,
 'senha' : document.getElementById('senha').value,
};
fetch('https://exemplo.com/usuarios',
 { method: 'POST',
 headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
 body: new URLSearchParams(obj)
 })
 .then(response => {
 if (response.status >= 400) {
 throw new Error('Erro ' + response.status);
 }
 alert('Cadastrado com sucesso.');
 })
 .catch(err => alert(err.message));
```

# enviando dados de um formulário – modo 2

```
// Envia como multipart/form-data 
fetch('https://exemplo.com/usuarios',
 { method: 'POST',
 headers: { 'Content-Type': 'multipart/form-data' },
 body: new FormData(document.getElementById('form-usuario'))
 })
 .then(response => {
 if (response.status >= 400) {
 throw new Error('Erro ' + response.status);
 }
 alert('Cadastrado com sucesso.');
 })
 .catch(err => alert(err.message));
```

# enviando dados de um formulário – modo 3

```
// Envia como application/json 
const obj = {
 nome : document.getElementById('nome').value,
 login : document.getElementById('login').value,
 senha : document.getElementById('senha').value,
};
fetch(
 'https://exemplo.com/usuarios',
 { method: 'POST',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify(obj)
 })
 .then(response => {
 if (response.status >= 400) {
 throw new Error('Erro ' + response.status);
 }
 alert('Cadastrado com sucesso.');
 })
 .catch(err => console.error(err));
```



# removendo um recurso

```
fetch('https://exemplo.com/produtos/105',
 { method: 'DELETE' })
 .then(response => {
 if (response.status >= 400) {
 throw new Error('Erro ' + response.status);
 }
 alert('Removido');
 })
 .catch(err => console.error(err));
```

criaremos um servidor HTTP RESTful servindo JSON  
é necessário ter NodeJS instalado

1. Crie uma pasta e a acesse, ex:

```
mkdir serv1
cd serv1
```

2. Inicialize um pacote NPM

```
npm init --yes
```

3. Instale a dependência JSON Server:

```
npm i -D json-server
```

4. crie o arquivo **recursos.json**, para representar seus recursos em formato JSON. Nele, declare um objeto vazio, ou seja, **{}**.

5. Adicione ao objeto a propriedade "**produtos**", com o seguinte *array* de objetos:

```
[{"id": 1, "descricao": "Água Mineral 1,5L",
 "preco": 2.50, "estoque": 20 },
 {"id": 2, "descricao": "Suco de Uva 1L",
 "preco": 10.00, "estoque": 10 },
 {"id": 3, "descricao": "Cerveja Acme",
 "preco": 7.00, "estoque": 15 }]
```

6. No console, execute:

```
npx json-server --watch recursos.json
```

Tipicamente, o servidor HTTP irá iniciar na porta 3000, estando disponível em <http://localhost:3000>.

O recurso criado, produtos, estará disponível em: <http://localhost:3000/produtos>

Ele aceita requisições GET, POST, PUT e DELETE, ao estilo REST.

*Considere o servidor criado anteriormente.*

1. Crie um arquivo **produtos.html** que carregue e exiba uma listagem dos produtos existentes no servidor, em uma tabela HTML.
2. Modifique a listagem de produtos para permitir a seleção de uma linha, utilizando CSS e o evento de clique. Crie um botão Remover que permita remover o produto selecionado da tabela e do servidor (DELETE).

3. Crie um formulário, acima da listagem de produtos, que permita cadastrar um produto no servidor (POST), ao clicar em Salvar. Acima do formulário, crie o botão Novo, que deve criar um produto novo e limpar o formulário.

4. Usando o recurso de seleção de linha criado no exercício 2, faça com que o formulário exiba o produto selecionado quando a linha for clicada. Ao clicar em salvar, se o produto tiver um id diferente de zero, faça a alteração do produto no servidor (PUT).

# referências

FRIEDMAN, Daniel; WISE, David. *The Impact of Applicative Programming on Multiprocessing*. International Conference on Parallel Processing. pp. 263–272. 1976.

HIBBARD, Peter. *Parallel Processing Facilities*. New Directions in Algorithmic Languages, IRIA, 1976.

BAKER, Henry; Carl Hewitt (1977). [\*The Incremental Garbage Collection of Processes\*](#). Proceedings of the Symposium on Artificial Intelligence Programming Languages ACM SIGPLAN Notices 12, 8. pp. 55–59. 1977.

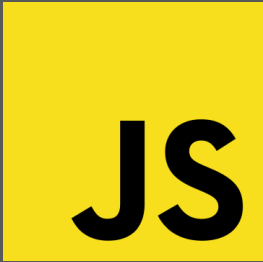
ARCHIBALD, Jake. *In the Loop*. JS Conf Asia, 2008. Disponível em: <https://www.youtube.com/watch?v=cCOL7MC4PI0>

JavaScript.Info. *Script Async Defer*. Disponível em: <https://javascript.info/script-async-defer>

MDN. *Promise*. Disponível em: [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise)

MDN. *Fetch API*. Disponível em: [https://developer.mozilla.org/pt-BR/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/pt-BR/docs/Web/API/Fetch_API)

WHATWG. *Fetch Standard*. Disponível em: <https://fetch.spec.whatwg.org/>



*fim*

2022.12.08 – Melhoria dos slides sobre defer e async.

2021.10.04 – Insere slide 34 de observação e slides 50 e 51 sobre allSettle().

2021.05.20 – Insere slide 58 e ajusta formatação de slides 59 e 60, todos sobre envio de dados de formulário.

2020.11.02 – Inicial.



Licença Creative Commons 4

ESTE MATERIAL PERTENCE AO PROFESSOR THIAGO DELGADO PINTO  
E ESTÁ DISPONÍVEL SOB A LICENÇA CREATIVE COMMONS VERSÃO 4.  
AO SE BASEAR EM QUALQUER CONTEÚDO DELE, POR FAVOR, CITE-O.