



Programação para Internet

Módulo 7

Técnica Ajax e Requisições Assíncronas - Parte 2

(JavaScript Promises, API Fetch, Async/Await)

Prof. Dr. Daniel A. Furtado - FACOM/UFU

Conteúdo protegido por direito autoral, nos termos da Lei nº 9 610/98

A cópia, reprodução ou apropriação deste material, total ou parcialmente, é proibida pelo autor

Conteúdo do Módulo

Parte 1

- Introdução, ideia geral, aplicações
- Requisições HTTP, análise no navegador
- Ajax com o [XMLHttpRequest](#): recursos, exemplos, JSON

Parte 2

- JavaScript [Promises](#)
- Ajax com a API [Fetch](#): conceitos, recursos, exemplos
- API Fetch com [async](#) / [await](#)

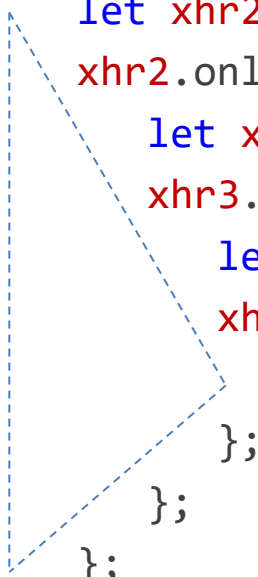
Ajax com a **API Fetch**

API Fetch

- Outra forma de realizar requisições Ajax
- Mais nova que o XMLHttpRequest
- Maior facilidade para encadear tarefas assíncronas
- Maior clareza e simplicidade com `async` / `await`
- Utiliza o conceito de `promise` do JavaScript

Callback Hell

```
let xhr1 = new XMLHttpRequest();
xhr1.onload = function () {
  let xhr2 = new XMLHttpRequest();
  xhr2.onload = function () {
    let xhr3 = new XMLHttpRequest();
    xhr3.onload = function () {
      let xhr4 = new XMLHttpRequest();
      xhr4.onload = function () {
        console.log(xhr4.response);
      };
    };
  };
};
```

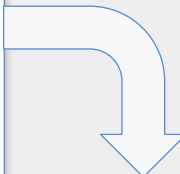


Este exemplo ilustra um possível encadeamento de requisições Ajax utilizando o XMLHttpRequest. Repare que há diversas chamadas em cascata de funções de callback (callback hell), tornando o código complexo e de difícil manutenção. O conceito de **promise** em conjunto com a API **Fetch** permite evitar situações como esta.

Evitando Callback Hell

```
1. let xhr1 = new XMLHttpRequest();
2. xhr1.open("GET", "URL1");
3. xhr1.responseType = 'json';
4. xhr1.onload = function () {
5.     const data1 = xhr1.response;
6.     let xhr2 = new XMLHttpRequest();
7.     xhr2.open("GET", "URL2");
8.     xhr2.responseType = 'json';
9.     xhr2.onload = function () {
10.         const data2 = xhr2.response;
11.         let xhr3 = new XMLHttpRequest();
12.         xhr3.open("GET", "URL3");
13.         xhr3.responseType = 'json';
14.         xhr3.onload = function () {
15.             const data3 = xhr3.response;
16.             console.log(data3);
17.         }
18.         xhr3.onerror = function () {
19.             console.error("Erro de rede XHR3");
20.         };
21.         xhr3.send();
22.     }
23.     xhr2.onerror = function () {
24.         console.error("Erro de rede XHR2");
25.     };
26.     xhr2.send();
27. }
28. xhr1.onerror = function () {
29.     console.error("Erro de rede XHR1");
30. };
31. xhr1.send();
```

Encadeando Requisições com XHR



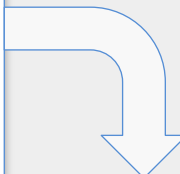
```
1. function getJSON(URL) {
2.     return fetch(URL)
3.         .then(response => response.json());
4. }
5. getJSON('URL1')
6.     .then(data1 => getJSON('URL2'))
7.     .then(data2 => getJSON('URL3'))
8.     .then(data3 => console.log(data3))
9.     .catch(error => console.error(error));
```

Código equivalente utilizando **Fetch/Promises**

Evitando Callback Hell

```
1. let xhr1 = new XMLHttpRequest();
2. xhr1.open("GET", "URL1");
3. xhr1.responseType = 'json';
4. xhr1.onload = function () {
5.     const data1 = xhr1.response;
6.     let xhr2 = new XMLHttpRequest();
7.     xhr2.open("GET", "URL2");
8.     xhr2.responseType = 'json';
9.     xhr2.onload = function () {
10.         const data2 = xhr2.response;
11.         let xhr3 = new XMLHttpRequest();
12.         xhr3.open("GET", "URL3");
13.         xhr3.responseType = 'json';
14.         xhr3.onload = function () {
15.             const data3 = xhr3.response;
16.             console.log(data3);
17.         }
18.         xhr3.onerror = function () {
19.             console.error("Erro de rede XHR3");
20.         };
21.         xhr3.send();
22.     }
23.     xhr2.onerror = function () {
24.         console.error("Erro de rede XHR2");
25.     };
26.     xhr2.send();
27. }
28. xhr1.onerror = function () {
29.     console.error("Erro de rede XHR1");
30. };
31. xhr1.send();
```

Encadeando Requisições com XHR



```
function getJSON(URL) {
    return fetch(URL)
        .then(response => response.json());
}

async function getData() {
    try {
        let data1 = await getJSON('URL1');
        let data2 = await getJSON('URL2');
        let data3 = await getJSON('URL3');
        console.log(data3);
    }
    catch (error) {
        console.error(error);
    }
}
```

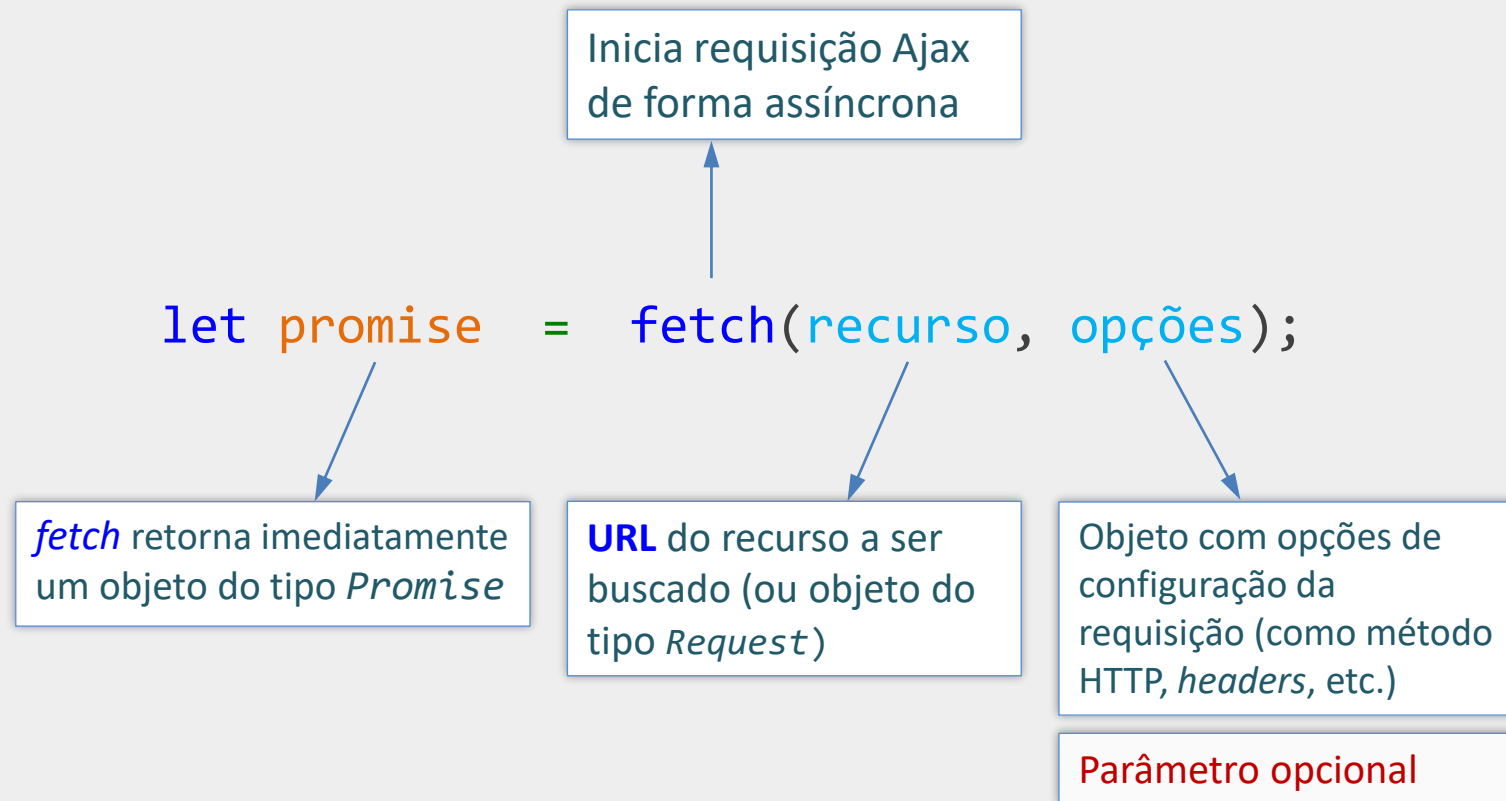
Código equivalente com Fetch e async/await

Introdução à Promises

- Promises facilitam o uso de **métodos assíncronos**
- Tais métodos são executados em **segundo plano** (em outra *thread*) e não retornam um valor final imediatamente
- Mas podem retornar um objeto do tipo **promise**, representando uma "promessa" de fornecer o valor final em um momento posterior

Em outras palavras, uma **promise** é um objeto que representa uma tarefa assíncrona a ser finalizada no futuro

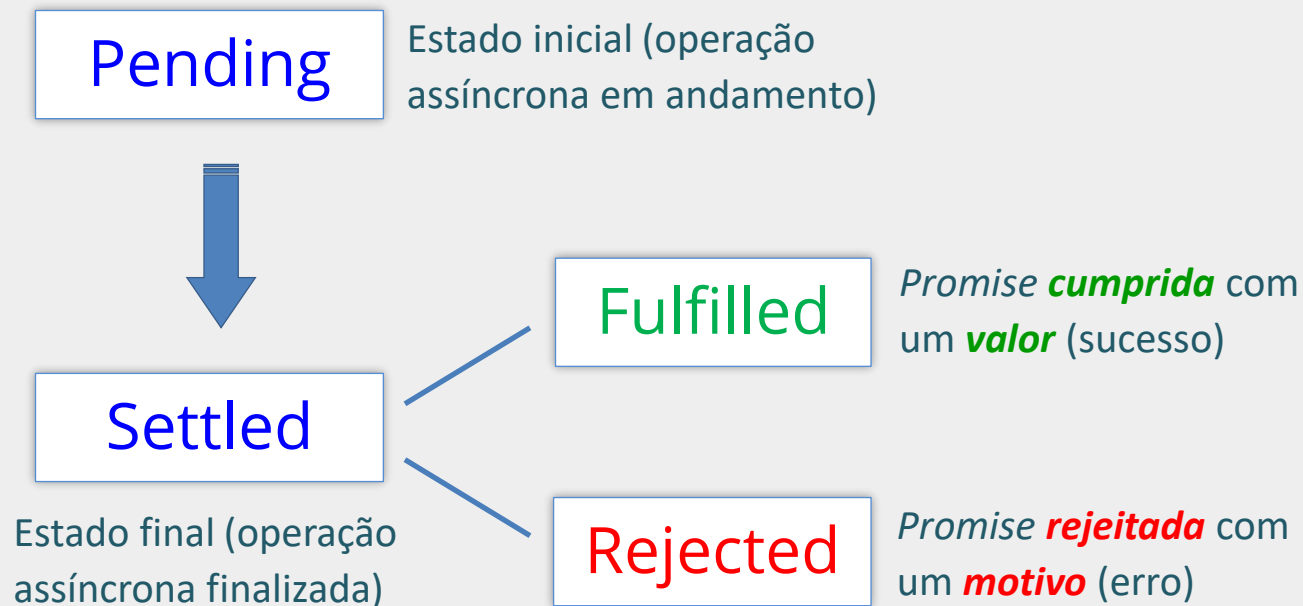
Introdução à Promises - Método fetch



Introdução à Promises

- Se finalizada com **sucesso**, a promise produzirá um **valor**
- Se finalizada com **falha**, produzirá um **motivo** (erro)
- Funções de callback são indicadas para tratar o valor/erro

Estados de uma Promise



Método then

- `then` é um método do objeto `promise`
- Permite resgatar o resultado da `promise`:
 - Pela indicação de função de callback de `sucesso`
 - Pela indicação de função de callback de `erro`
- Retorna uma nova `promise`

Funções de Callback

```
let promise = fetch(URL);  
promise.then(trataResultadoSucesso, trataResultadoErro);
```

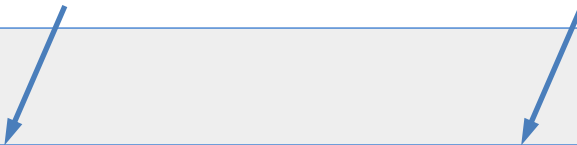
Indique uma função de *callback* a ser chamada quando a promise finalizar com sucesso (*fulfills*)

Indique uma função de *callback* a ser chamada quando a promise finalizar com erro (*rejects*)

Opcional

Funções de Callback

```
let promise = fetch(URL);  
promise.then(trataResultadoSucesso, trataResultadoErro);
```



As funções de callback recebem por parâmetro o resultado obtido pela operação assíncrona.

Funções de Callback - Exemplo

```
promise.then(  
  function (result) {  
    console.log(result);  
  },  
  function (error) {  
    console.log(error);  
  }  
);
```

Funções de Callback - Exemplo

```
promise.then(  
  result => console.log(result) ,  
  error => console.log(error)  
);
```

Utilizando *arrow function*

Funções de Callback - Garantias

```
let promise = fetch(URL);  
promise.then(trataResultSucesso, trataResultErro);
```

No momento da chamada do método **then** é possível que a promise já tenha sido finalizada (*fulfilled* ou *rejected*). Ainda assim, a função de callback indicada será chamada (de sucesso ou de falha, respectivamente).

Concatenando Múltiplos then's

```
promise.then(f1).then(f2).then(f3);
```

- A concatenação de múltiplos **then's** permite executar tarefas assíncronas em sequência
- No exemplo acima, uma tarefa assíncrona em **f2** seria iniciada apenas após conclusão da tarefa assíncrona associada a **f1**. E **f3** seria iniciada após conclusão da tarefa assíncrona associada a **f2**
- O valor obtido com o cumprimento da **promise** anterior é passado para a função de callback seguinte
- Isto é possível porque o método **then** sempre retorna uma nova **promise**, que está associada à finalização de suas callbacks

Concatenando Múltiplos then's

```
promise.then(f1).then(f2).then(f3);
```



```
promise  
  .then(f1)  
  .then(f2)  
  .then(f3);
```

Indentação mais comum com `.then` em nova linha

Concatenando Múltiplos then's

```
promiseA  
  .then(f1 , e1)  
  .then(f2)  
  .then(f3 , e3)
```

Funções de tratamento de erro podem ser adicionadas em cada .then, caso o erro precise ser tratado imediatamente.

*Neste exemplo, se a *promiseA* for rejeitada, o erro será tratado por *e1*, e se a promise retornada pelo 1º *then* for rejeitada, o erro será tratado por *e3* (*f2* será ignorada).*

Método catch

```
promise
  .then(f1)
  .catch(e1)
  .then(f2)
  .then(f3)
  .catch(e2)
```

- O método **catch** é outra forma de indicar função para tratar erros
- Tem papel análogo à “*.then(null, fe)*”
- Mais comumente utilizado no final do encadeamento
- Tratamento de erros concentrado no mesmo bloco
- Não precisa ser único nem usado necessariamente no final

Método catch

```
promise  
  .then(f1)  
  .then(f2)  
  .then(f3)  
  .catch(fe)
```

*Neste exemplo, caso **f1** lance uma exceção ou resulte em uma **promise rejeitada** então as funções **f2** e **f3** serão ignoradas, pois a execução será deslocada para a próxima callback de tratamento de erros (neste caso, a função **fe** do método **.catch**)*

Método finally

```
promise  
  .then(f1)  
  .then(f2)  
  .catch(fe)  
  .finally(f)
```

*O método **finally** permite executar uma ação sempre que a promise **finaliza**, independentemente de ser com sucesso ou não (**cumprida** ou **rejeitada**).*

Método *fetch* - Exemplo

```
fetch("endereco.php?cep=38400-100") // inicia requisição assíncrona
  .then(response => response.json()) // lê string JSON e conv. p/ JS
  .then(data => console.log(data))   // mostra o resultado
  .catch(error => console.error(error)) // mostra eventual erro
```

- *fetch* inicia requisição assíncrona e retorna *promise*
- Se cumprida, a *promise* resultará em um objeto do tipo *Response*
- `response.json()` lê a string JSON e converte em objeto JavaScript
- `response.json()` executa de forma assíncrona e retorna nova *promise*
 - Se cumprida, resultará em objeto JavaScript contendo os dados

Outros Métodos de um Objeto Response

`response.json()`

- Lê, de forma assíncrona, a stream de resposta contendo a string JSON, e a converte em objeto JavaScript
- Retorna `promise` que será cumprida com o objeto JavaScript

`response.text()`

- Lê a stream de resposta no formato textual
- Retorna `promise` que será cumprida com a string resultante

`response.blob()`

- Lê a stream de resposta como um `Blob` (**B**inary **L**arge **O**bject)
- Retorna `promise` que será cumprida com o `blob` resultante

Propriedades Comuns de um Objeto Response

- `response.ok` - true quando o servidor retorna status 200-299
- `response.status` - código de status HTTP retornado pelo servidor
- `response.headers` - informações de cabeçalho retornadas pelo servidor
- `response.url` - URL final da resposta da requisição
- `response.type` - tipo da resposta (basic, cors, etc.)

Confirmando Sucesso da Requisição

```
fetch("endereco.php?cep=38400-100")
  .then(response => {
    if (!response.ok)
      throw new Error("Not ok");

    return response.json();
  })
  .then(endereco => console.log(endereco))
  .catch(error => console.error(error))
```

response.ok
será verdadeira apenas
quando o servidor retornar
um código de status HTTP de
200 a 299 indicando sucesso.

*O lançamento de uma exceção, como neste exemplo, faz com que a promise seja **rejeitada**. Neste caso, a execução prosseguiria para a função de tratamento de erro do método **.catch**.*

Exemplo de Requisições em Sequência

```
fetch(URL1)
  .then(response1 => response1.json())
  .then(data1 => fetch(URL2))
  .then(response2 => response2.json())
  .then(data2 => console.log(data2))
  .catch(error => console.error(error))
```

- A 1ª requisição `fetch` é resolvida com obtenção de `data1`
- A 2ª requisição é iniciada após finalização da 1ª e pode utilizar `data1`
- A 2ª requisição é resolvida com obtenção de `data2`

Atenção para Eventual Necessidade do return

```
fetch(URL1)
  .then(response1 => response1.json())
  .then(data1 => fetch(URL2))
  ...
```

Arrow function com apenas uma declaração: não necessita do **return** na chamada do **fetch**. (**return** implícito)

```
fetch(URL1)
  .then(response1 => response1.json())
  .then(data1 => {
    console.log(data1);
    return fetch(URL2);
  })
  ...
```

Função com mais de uma declaração (com chaves): necessário utilizar explicitamente o **return** neste contexto.

Criando sua Própria Promise

```
let minhaPromise = new Promise((resolve, reject) => {  
    // Chame o método resolve(...) quando suas operações assíncronas  
    // finalizarem com sucesso e produzirem o resultado esperado  
    if (operaçõesAssincExecutadasComSucesso)  
        resolve(resultado);  
  
    // Chame o método reject(...) quando as operações falharem  
    if (operaçõesAssincFalharam)  
        reject("Falha XYZ");  
})  
  
minhaPromise.then(  
    result => console.log(result) ,  
    error => console.log(error)  
);
```

Criando sua Própria Promise - Exemplo

```
function getJSON(url) {  
    return new Promise(function (resolve, reject) {  
        let xhr = new XMLHttpRequest();  
        xhr.open("GET", url);  
        xhr.responseType = "json";  
        xhr.onload = function () {  
            if (xhr.status == 200)  
                resolve(xhr.response);  
            else  
                reject("Not ok: " + xhr.status);  
        };  
        xhr.onerror = function () { reject("Erro de rede"); };  
        xhr.send();  
    });  
}  
getJSON("data.json").then(result => console.log(result));
```

Exemplo simplificado, sem tratar todas as possíveis falhas/exceções

Fetch com Opções de Inicialização

```
// localiza formulário e cria objeto FormData  
let meuForm = document.querySelector("form");  
let formData = new FormData(meuForm);  
  
// opções da requisição  
const options = {  
  method: "POST",  
  body: formData  
}  
  
// inicia requisição  
fetch("processa-form.php", options)  
  .then...
```

Fetch com opções de inicialização - Enviando formulário com **FormData**

Enviando Arquivo com FormData

```
// localiza o campo relativo ao arquivo a ser enviado
let campoArq = document.querySelector('input[type="file"]');

// Cria um objeto FormData e adiciona o arquivo
let formData = new FormData(meuForm);
formData.append("arquivo", campoArq.files[0]);

// opções da requisição
const options = {
  method: "POST",
  body: formData
}

// inicia requisição
fetch("processa-arq.php", options)
  .then...
```

Enviando Objeto JSON

```
// objeto JavaScript contendo os dados de envio  
let dados = {  
    cep : "38400-100",  
    user : "abcd"  
};  
  
// opções da requisição  
const options = {  
    method: "POST",  
    body: JSON.stringify(dados),  
    headers: { 'Content-Type': 'application/json' }  
}  
  
// inicia requisição  
fetch("processa-dados.php", options)  
    .then...
```

Método `Promise.all()`

- Permite executar várias tarefas assíncronas em **paralelo**
- Para situações onde as tarefas são independentes
- Permite agregar os resultados das várias tarefas
- E executar ação quando todas finalizarem com sucesso

Método `Promise.all()`

```
Promise.all([
    tarefaAssinc1(),
    tarefaAssinc2(),
    tarefaAssinc3(),
    tarefaAssincN()
])
.then(results => console.log(results))
.catch(error => console.log(error))
```

O método `Promise.all` retorna uma nova promise que será cumprida apenas quando **todas** as promises do vetor forem cumpridas. Se alguma promise for rejeitada, então a promise retornada também será rejeitada imediatamente. A promise retornada, se cumprida, resolverá em **array** contendo os resultados de **todas** as promises.

Métodos Similares a `Promise.all()`

`Promise.allSettled()`

Retorna uma promise que é *cumprida* quando **todas** as promises recebidas são cumpridas **ou** rejeitadas.

`Promise.any()`

Retorna uma promise que é *cumprida* quando **alguma** das promises recebidas é cumprida. Rejeita quando todas são rejeitadas.

`Promise.race()`

Retorna uma promise que é *cumprida* ou *rejeitada* assim que uma das promises recebidas é cumprida ou rejeitada.

Fetch com **async/await**

async/await

- Parte da ECMAScript 2017
- Possibilita que funções assíncronas sejam chamadas com sintaxe “similar” às síncronas
- Utiliza-se o termo **async** na definição da função e **await** dentro da função na chamada assíncrona

Vantagens de Utilizar `async/await`

- Maior clareza e simplicidade do código
- Dispensa os aninhamentos das promises
- Melhor tratamento de erros com `try/catch`
- Mais fácil de depurar

Função Assíncrona e Expressão await

```
async function funcaoExemplo() {  
    let result1 = await funcAssinc1();  
    let result2 = await funcAssinc2();  
}
```

- Utiliza-se a palavra reservada **async** antes de **function**
- Dessa forma a função pode conter expressões **await**
- Funções assíncronas são chamadas no “estilo” das síncronas

Função Assíncrona e Expressão await

```
async function funcaoExemplo() {  
    let result1 = await funcAssinc1();  
    let result2 = await funcAssinc2();  
}
```

- `await` pode ser usada na chamada de funções que retornam `promises`
- Suspende a execução até que a promise retornada seja `cumprida` ou `rejeitada`
- O valor resolvido da `promise` será o valor de retorno da expressão `await`

Função Assíncrona e Expressão await

```
async function funcaoExemplo() {  
    let result1 = await funcAssinc1();  
    let result2 = await funcAssinc2();  
}
```

- Funções definidas com **async** sempre retornam uma **promise**
- Se o valor de retorno não é explicitamente uma **promise**, então ele será automaticamente encapsulado em uma
- Se uma função **async** não contém **await** então ela é executada de forma síncrona

Função Assíncrona e Expressão *await*

```
async function funcaoExemplo() {  
    let result1 = await funcAssinc1();  
    let result2 = await funcAssinc2();  
}
```

await é permitida apenas dentro de funções *async**

A suspensão com o *await* não causa um bloqueio da thread principal. Isso significa que é possível executar outras funções, tratar eventos, responder à interf. do usuário, etc.

* e também dentro do corpo de módulos

Tratamento de Erros com *async/await*

```
async function funcaoExemplo() {  
  try {  
    let result1 = await funcAssinc1();  
    let result2 = await funcAssinc2();  
  }  
  catch (e) {  
    console.log(e);  
  }  
}
```

*Se a promise vinculada à função assíncrona for rejeitada então a execução será deslocada para o bloco **catch**.*

fetch com async/await

```
async function exemploSimples() {  
  const response = await fetch("endereco.php");  
  const endereco = await response.json();  
  console.log(endereco);  
}
```

Exemplo simplificado, sem tratamento de erros

fetch com async/await

```
async function getAddress(cep) {  
  try {  
    const response = await fetch("endereco.php?cep=" + cep);  
    if (! response.ok)  
      throw new Error("Falha inesperada: " + response.status);  
  
    const endereco = await response.json();  
    console.log(endereco);  
  }  
  catch (e) {  
    console.error(e);  
  }  
}
```

`async/await` possibilitam o tratamento de erros convencional com `try/catch`

Promise.all() com *async/await*

```
try {  
    let [r1, r2, r3] = await Promise.all([  
        tarefa1(),  
        tarefa2(),  
        tarefa3()  
    ]);  
}  
catch (e) {  
    console.error(e);  
}
```

O erro será tratado para a primeira promise rejeitada

Qual a diferença?

```
try {  
  let r1 = await tarefa1();  
  let r2 = await tarefa2();  
  let r3 = await tarefa3();  
}  
catch (e) {  
  console.error(e);  
}
```

X

```
try {  
  const promise1 = tarefa1();  
  const promise2 = tarefa2();  
  const promise3 = tarefa3();  
  
  let r1 = await promise1;  
  let r2 = await promise2;  
  let r3 = await promise3;  
}  
catch (e) {  
  console.error(e);  
}
```

Qual a diferença?

```
try {  
  let r1 = await tarefa1();  
  let r2 = await tarefa2();  
  let r3 = await tarefa3();  
}  
catch (e) {  
  console.error(e);  
}
```

As três tarefas são executadas em segundo plano, mas *uma após a outra*. O tempo total de execução é aprox. a soma dos tempos de cada tarefa.

X

```
try {  
  const promise1 = tarefa1();  
  const promise2 = tarefa2();  
  const promise3 = tarefa3();  
  
  let r1 = await promise1;  
  let r2 = await promise2;  
  let r3 = await promise3;  
}  
catch (e) {  
  console.error(e);  
}
```

As três tarefas são *iniciadas imediatamente* e executadas em segundo plano em “paralelo”. O tempo total é aproximadamente o tempo de execução da mais longa.

Qual a diferença?

```
try {  
  let [r1, r2, r3] = await Promise.all([  
    tarefa1(),  
    tarefa2(),  
    tarefa3()  
  ]);  
}  
catch (e) {  
  console.error(e);  
}
```

X

```
try {  
  const promise1 = tarefa1();  
  const promise2 = tarefa2();  
  const promise3 = tarefa3();  
  
  let r1 = await promise1;  
  let r2 = await promise2;  
  let r3 = await promise3;  
}  
catch (e) {  
  console.error(e);  
}
```

Qual a diferença?

```
try {  
  let [r1, r2, r3] = await Promise.all([  
    tarefa1(),  
    tarefa2(),  
    tarefa3()  
  ]);  
}  
catch (e) {  
  console.error(e);  
}
```

X

```
try {  
  const promise1 = tarefa1();  
  const promise2 = tarefa2();  
  const promise3 = tarefa3();  
  
  let r1 = await promise1;  
  let r2 = await promise2;  
  let r3 = await promise3;  
}  
catch (e) {  
  console.error(e);  
}
```

Em caso de sucesso na execução das tarefas, os códigos se comportaram de maneira similar. Porém, em caso de promise rejeitada na tarefa3, por exemplo, o código da esquerda captura o erro e o trata mais rapidamente (fail fast). No código da direita a exceção será tratada apenas depois que as tarefas 1 e 2 terminarem. Além disso, esse catch irá capturar apenas a 1ª exceção lançada. Caso as outras promises lancem erros adicionais, eles serão propagados, mas não capturados.

Referências

- <https://xhr.spec.whatwg.org/>
- <https://www.ecma-international.org/ecma-262/>
- <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>
- https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- <https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Asynchronous/Concepts>
- **JavaScript and JQuery: Interactive Front-End Web Development**, Jon Duckett.