



Programação para Internet

Módulo 7

Técnica Ajax

(XMLHttpRequest, Fetch, Promises, Async/Await, JSON, SARS)

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 à técnica Ajax
- Objeto XMLHttpRequest
- Conceitos, requisições HTTP, tratamento de erros, JSON, APIs públicas

Parte 2

- Ajax com a API Fetch e Promises
- Encadeamento de requisições
- API Fetch com async / await

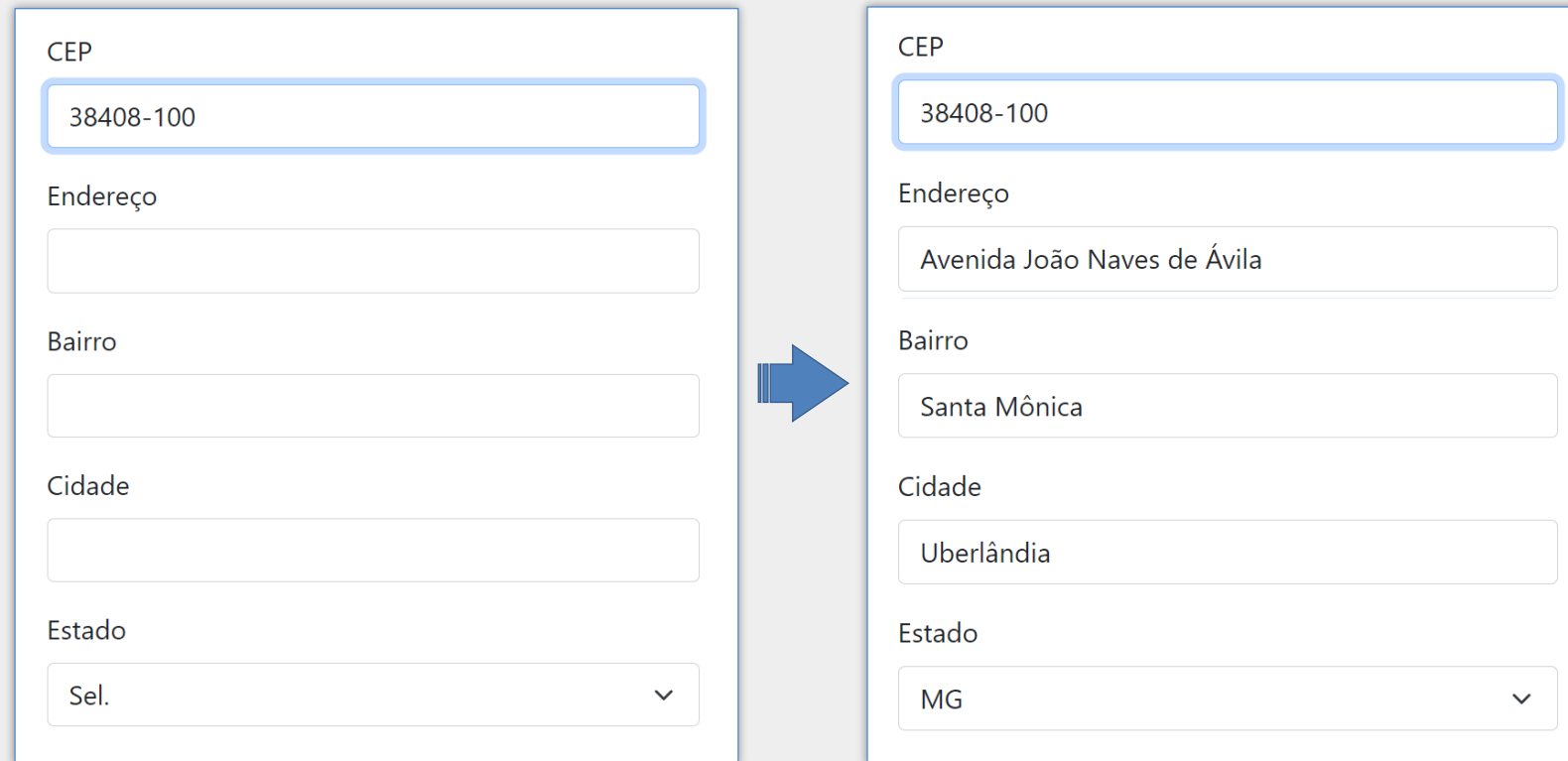
Pré-Requisitos Desejáveis

- Desenvolvimento front-end básico
 - HTML, JavaScript
 - Manipulação da árvore DOM
- Desenvolvimento back-end básico
 - Comunicação com servidor web

O que é Ajax?

- Termo proposto em 2005 por Jesse Garrett
- Técnica que possibilita atualizações incrementais na página, sem a necessidade de recarregá-la completamente
- A atualização da página é feita com requisições HTTP em segundo plano para:
 - buscar rapidamente dados adicionais no servidor,
 - carregá-los dinamicamente na página,
 - sem interromper a navegação do usuário

Exemplo



The diagram illustrates a two-step process for address validation. On the left, a form with fields for CEP, Endereço, Bairro, Cidade, and Estado (a dropdown menu) is shown. The CEP field contains the value '38408-100'. A blue arrow points to the right, where the same form is shown with the other fields automatically populated: 'Avenida João Naves de Ávila' for Endereço, 'Santa Mônica' for Bairro, 'Uberlândia' for Cidade, and 'MG' for Estado.

CEP

38408-100

Endereço

Bairro

Cidade

Estado

Sel. ▼

CEP

38408-100

Endereço

Avenida João Naves de Ávila

Bairro

Santa Mônica

Cidade

Uberlândia

Estado

MG ▼

Preenchimento automático dos campos do endereço com AJAX após preenchimento manual do campo CEP

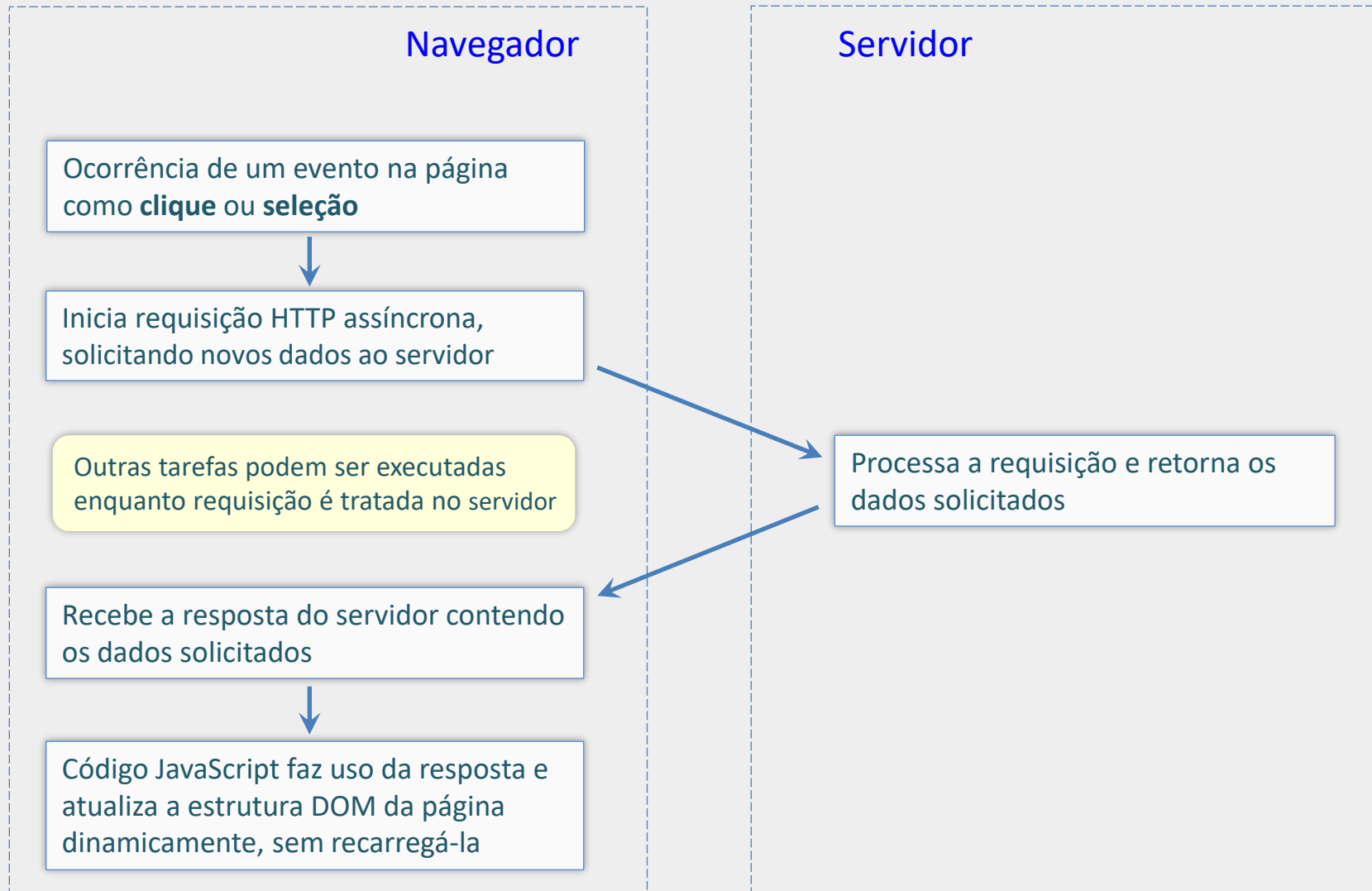
Informações Adicionais sobre Ajax

- Ajax não interrompe a navegação do usuário
- Pode trazer mais agilidade e responsividade às aplicações web
- Ajax não é uma nova tecnologia ou linguagem de programação
- Ajax combina várias tecnologias existentes (HTML, CSS, DOM, JavaScript e o XMLHttpRequest / Fetch)
- Ajax = **A**synchronous **J**avaScript and **X**ML

Outros Exemplos de Soluções com Ajax

- Carregamento dinâmico de caixas de seleção
- Páginas com atualização contínua de dados
- Salvamento automático
- Buscas instantâneas
- Rolagem infinita

Ideia Geral da Técnica Ajax



Como realizar Requisições Ajax com JavaScript?

Nativo

- XMLHttpRequest (XHR)
- API Fetch

Bibliotecas

- jQuery
- Axios

Como realizar Requisições Ajax com JavaScript?

Ajax com o XMLHttpRequest (XHR)

- Disponível na maioria dos navegadores, inclusive antigos
- Utiliza funções de callback
- Conceitualmente mais simples

Ajax com a API Fetch

- Mais nova que o XMLHttpRequest
- Utiliza Promises do JavaScript
- Facilidade para encadear tarefas assíncronas
- Maior clareza com `async` / `await`
- Possibilita requisições no-cors

Visão Geral de Requisições HTTP

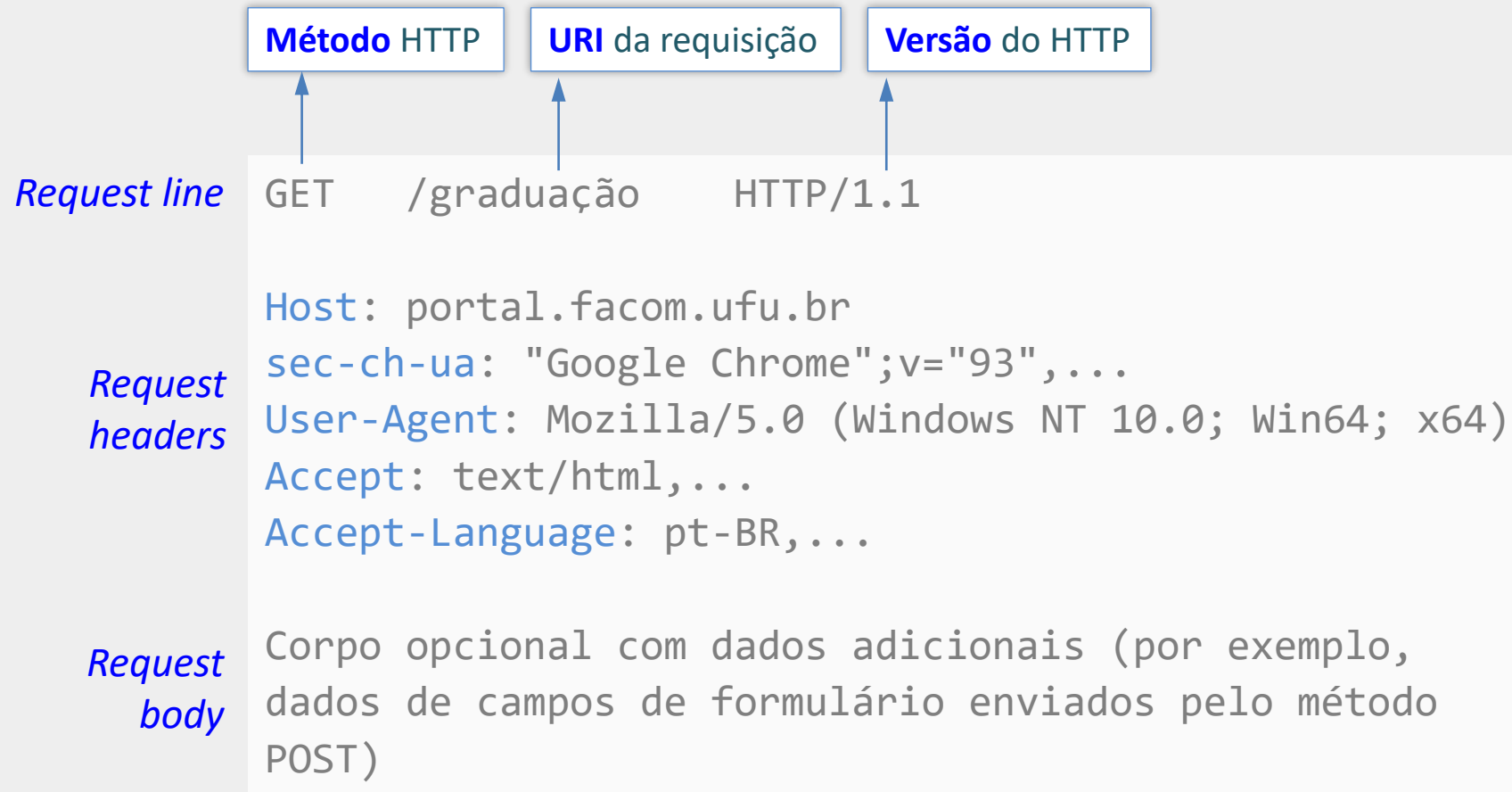
Noção de Requisições HTTP

- Uma **requisição HTTP** é uma mensagem baseada em texto (HTTP 1.1) que é enviada de um cliente a um servidor solicitando algum recurso, como página, imagem, etc.
- O servidor, por sua vez, envia de volta uma **resposta HTTP** (*http response*), contendo eventualmente o recurso solicitado

Noção de Requisições HTTP

- Uma **requisição HTTP** é normalmente composta por 3 partes:
 1. A **linha inicial** da requisição (*request line*)
 2. Uma coleção de **linhas de cabeçalho** (*header lines*)
 3. Um **corpo**, opcional, contendo dados adicionais (*request body*)

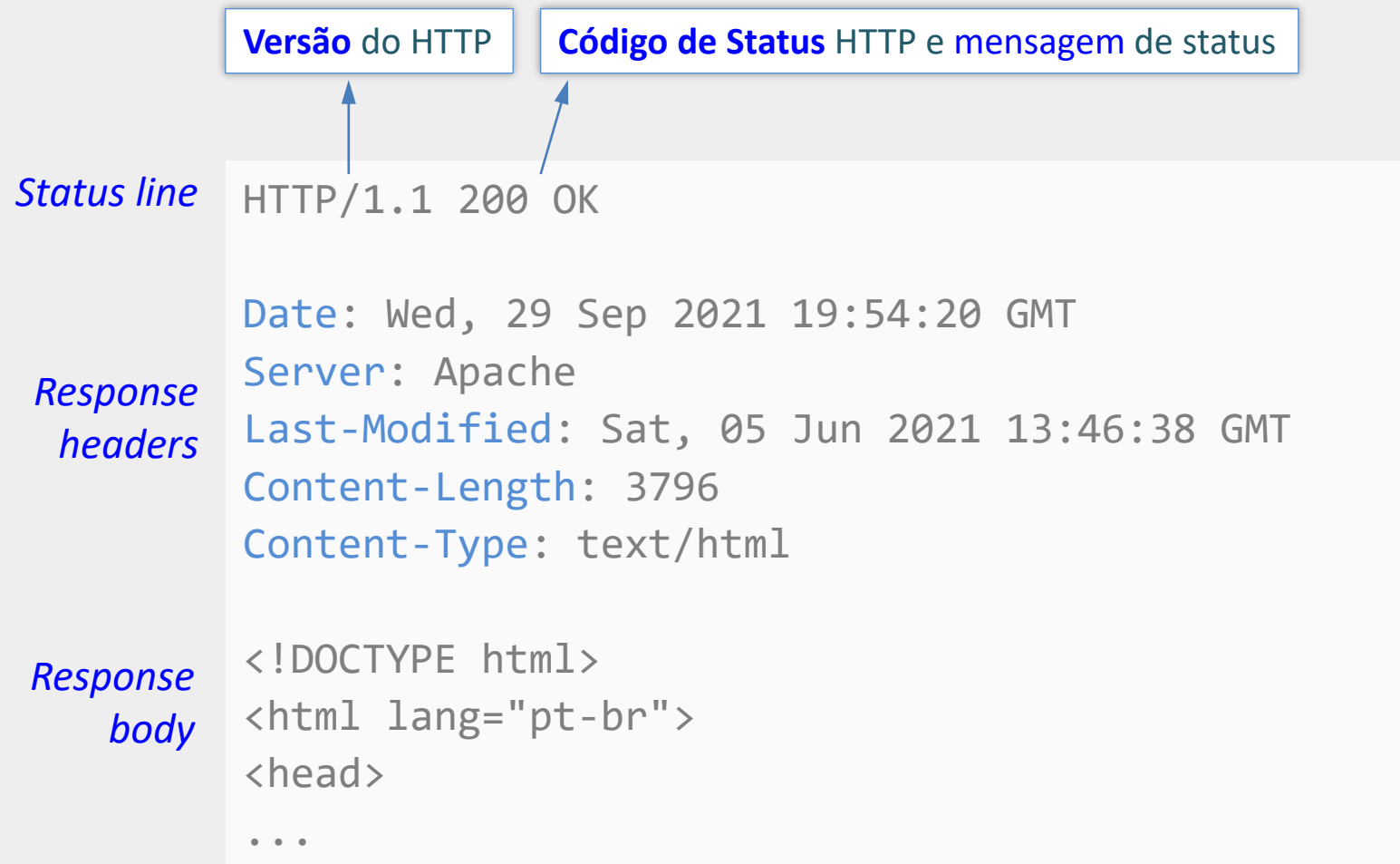
Exemplo de Requisição HTTP



Noção de Respostas HTTP

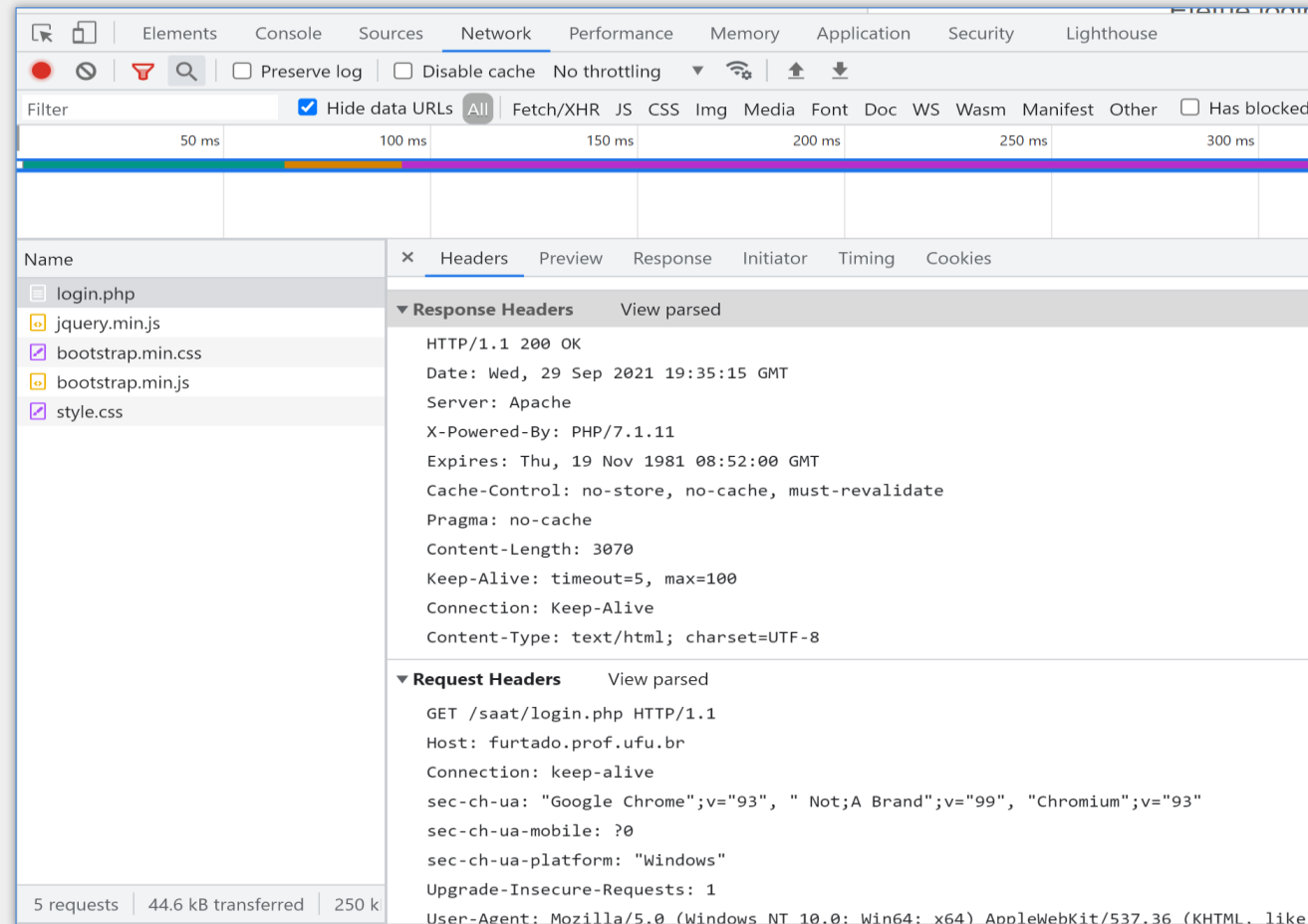
- Uma **resposta HTTP** também é composta por 3 partes:
 1. A **linha de status** da mensagem de resposta (*status line*)
 2. Uma coleção de **linhas de cabeçalho** (*response headers*)
 3. Um **corpo** contendo dados adicionais (*response body*)

Exemplo de Resposta HTTP



Verificando Dados da Requisição no Navegador

Google Chrome: F12 → Network → F5 → Sel. Arquivo → Headers
→ Response Headers (ou Request Headers) → View source



Alguns Códigos de Status HTTP

- 200 OK – resposta padrão para sucesso
- 302 Found – recurso encontrado, seguido por redirecionamento
- 400 Bad Request – possível erro do cliente ao montar a requisição
- 403 Forbidden – acesso ao recurso não autorizado
- 404 Not Found – recurso não encontrado
- 500 Internal Server Error – erro interno no servidor

Ajax com o XMLHttpRequest

Passos Básicos para Iniciar Requisição Ajax

1. Criar objeto `XMLHttpRequest` (XHR)
2. Indicar a URL da requisição - método `open`
3. Indicar função para tratar resposta - propriedade `onload`
4. Enviar a requisição - método `send`

Exemplo Simples de Requisição Ajax

```
let xhr = new XMLHttpRequest();  
xhr.open("GET", "filmes.txt", true);  
xhr.onload = function () {  
    console.log(xhr.responseText);  
};  
xhr.send();
```

*Propriedade **onload**
Permite executar
uma ação quando a
requisição finalizar
e a resposta estiver
pronta*

*Prop. **responseText**
Contém a resposta
textual retornada
pelo servidor*

OBS: Exemplo simples, sem tratamento de erros

Método open

```
xhr.open("GET", "busca.php?id=1", true);
```

Método da requisição

URL da requisição

*Neste exemplo um
parâmetro é enviado
pela própria URL*

true para **assíncrona**
false para **síncrona**

*Caso não informado, o
padrão é **assíncrona***

- O método **open** permite inicializar a requisição, antes de enviá-la, indicando o método HTTP, a URL de destino, e a forma de tratamento da requisição (síncrona ou assíncrona)

Requisição Assíncrona x Síncrona

Requisição Assíncrona

- O código JS prossegue enquanto requisição é gerenciada pelo navegador
- É possível executar outras operações enquanto requisição é tratada
- Andamento da requisição pode ser monitorado com eventos

Requisição Síncrona

- Código JavaScript fica “bloqueado”, aguardando resposta do servidor
- Não recomendada para requisições fora de [Web Workers](#), pois na thread principal pode prejudicar a experiência do usuário
- Alguns recursos não estão disponíveis

Tratando Eventuais Erros de Rede

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "filmes.txt", true);

xhr.onload = function () {
    console.log(xhr.responseText);
};

xhr.onerror = function () {
    console.log("Erro a nível de rede");
};

xhr.send();
```

*Propriedade **onerror**
Permite tratar erros
de rede que tenham
impedido a finalização
da requisição*

Observações sobre **onerror**

```
xhr.onerror = function () {  
    console.log("Erro a nível de rede");  
};
```

Cobre apenas casos de erro a nível de rede, como:

- Falha na conexão de rede/internet
- Servidor demora para responder
- Alguns erros relacionados a permissões de acesso (CORS)

Não disparado em situações como:

- Servidor responde com código de status de erro (500, 404, etc.)
- Servidor responde com dados inesperados
 - Ex.: mensagens de erro/warnings do back-end

Verificando o código de status HTTP retornado

```
xhr.onload = function () {  
    if (xhr.status == 200)  
        console.log(xhr.responseText);  
    else  
        console.error("Falha: " + xhr.status + xhr.responseText);  
};  
  
xhr.onerror = function () {  
    console.log("Erro de rede");  
};
```

`xhr.status` permite verificar o código de status HTTP retornado pelo servidor
`200` é o código de status padrão indicando sucesso/ok.

Outras Propriedades de Evento do XHR

- **onloadstart** – início do carregamento da resposta
- **onloadend** – término do carregamento da resposta
- **onprogress** – permite monitorar o carregamento
- **onreadystatechange** – permite monitorar o andamento da requisição
- **ontimeout** – tempo máximo para encerrar requisição excedido

```
let xhr = new XMLHttpRequest();  
xhr.timeout = 5000;  
xhr.ontimeout = function () {  
    ...  
};
```

Formas de Acessar a Resposta

xhr.responseText

- Acesso à resposta retornada na forma textual

xhr.response

- Utilizada em conjunto com `xhr.responseType`
- Útil para resgatar dados em formatos específicos
 - `ArrayBuffer`, `Blob`, `Document`, `JSON`

xhr.responseXML

- Retorna um conteúdo XML (ou HTML) convertido em um objeto do tipo `Document` (árvore DOM)

xhr.responseURL

- Retorna a URL final depois de eventuais redirecionamentos

Requisição Ajax Retornando Imagem

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "imagemMuitoGrande.jpg");
xhr.responseType = "blob";

xhr.onload = function () {
    const blob = xhr.response;
    const img = document.createElement("img");
    img.src = window.URL.createObjectURL(blob);
    document.body.appendChild(img);
};

xhr.send();
```

Requisição Ajax Retornando Imagem

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "imagemMuitoGrande.jpg");
xhr.responseType = "blob";

xhr.onload = function () {
    // recupera os dados da imagem
    const blob = xhr.response;

    // insere a imagem dinamicamente na página
    const img = document.createElement("img");
    img.src = window.URL.createObjectURL(blob);
    document.body.appendChild(img);
};

xhr.send();
```

Exemplo de uso das propriedades `responseType` e `response` para carregar imagem dinamicamente com Ajax

Inserindo Imagem Dinamicamente sem Ajax

```
...  
const img = document.createElement("img");  
img.src = "images/imagemMuitoGrande.jpg";  
document.body.appendChild(img);  
...
```

Diferente do exemplo anterior, este código é executado de forma síncrona e irá "bloquear" o JavaScript até a imagem ser carregada

Carregando XML como Objeto Document

```
<disciplina>
  <codigo>GSI019</codigo>
  <nome>Program. para Internet</nome>
  <cargahoraria>60</cargahoraria>
  <ementa>ABCDE</ementa>
  <professor>Daniel</professor>
</disciplina>
```

Arquivo no servidor `disciplinaPPI.xml`

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "disciplinaPPI.xml");

xhr.onload = function () {
  const docXml = xhr.responseXML;
  const str = docXml.querySelector("ementa").innerHTML;
  console.log(str);
};

xhr.send();
```

Exemplo de uso da propriedade `responseXML` para carregar documento XML como objeto *Document* (árvore DOM)

Ajax, XHR e o Formato JSON

Introdução ao JSON

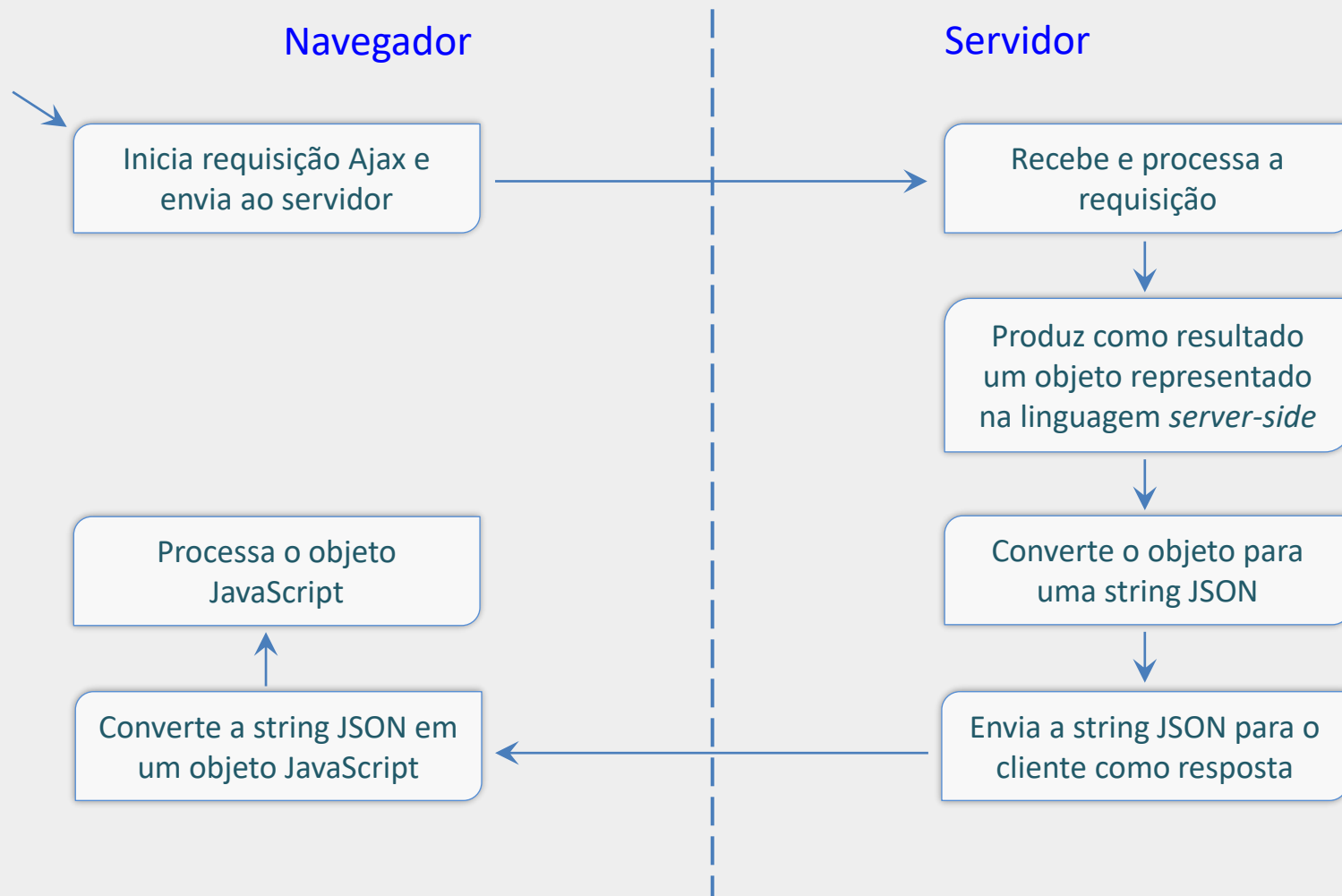
- Formato para representação de dados de forma textual
- Independente de linguagem
- Muito utilizado para intercâmbio de dados
 - Por exemplo, na comunicação cliente / servidor
- Permite a serialização de dados
- Acrônimo para **JavaScript Object Notation**

Formato JSON

```
const strJSON = '{  
  "Disciplina" : "Programação para Internet",  
  "Carga Horária" : 60,  
  "Avaliações" : [ 30, 30, 40 ],  
  "Professor" : "Furtado"  
';
```

- Dados organizados em pares ("*propriedade*" : *valor*)
- Nomes das propriedades devem usar aspas duplas
- Pares separados por vírgula
- **Objetos** são colocados entre **chaves**
- **Vetores** são colocados entre **colchetes**
- Valores das propriedades podem ser novos objetos

Ações Típicas de Requisição Retornando JSON



Duas Formas de Resgatar o Objeto JSON

1. Resgatar a string JSON com a propriedade `responseText` e então convertê-la em objeto JS utilizando `JSON.parse`
2. Definir `xhr.responseType = 'json'` e resgatar o objeto já convertido utilizando `xhr.response`

Requisição GET Retornando JSON - 1ª Forma

Usando `responseText` em conjunto com `JSON.parse`. Suponha que o servidor retornará string JSON com dados de endereço

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "endereco.php?cep=38400-100");
xhr.onload = function () {
    try {
        // JSON.parse converte string JSON em objeto JS
        var endereco = JSON.parse(xhr.responseText);
    }
    catch (e) {
        console.error("JSON inválido: " + xhr.responseText);
        return;
    }

    // insere os dados do endereço no formulário
    form.bairro.value = endereco.bairro;
    form.cidade.value = endereco.cidade;
};
xhr.send();
```

Requisição GET Retornando JSON - 2ª Forma

Usando `responseType` em conjunto com `response`

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "endereco.php?cep=38400-100");
xhr.responseType = 'json';

xhr.onload = function () {
    if (xhr.response === null) {
        console.log("Resposta não obtida");
        return;
    }

    const endereco = xhr.response;
    form.bairro.value = endereco.bairro;
    form.cidade.value = endereco.cidade;
};

xhr.send();
```

Ao definir **responseType** com o valor **'json'**, a string JSON retornada será automaticamente convertida em um objeto JS, que pode ser acessado pela propriedade **response**

Mas atenção: caso haja um erro na conversão da string JSON para o objeto JavaScript, não será possível identificar o erro em detalhes

Exemplo de Código PHP Retornando JSON

```
<?php
class Endereco {
    public $rua;
    public $bairro;
    public $cidade;

    function __construct($rua, $bairro, $cidade) {
        $this->rua = $rua;
        $this->bairro = $bairro;
        $this->cidade = $cidade;
    }
}

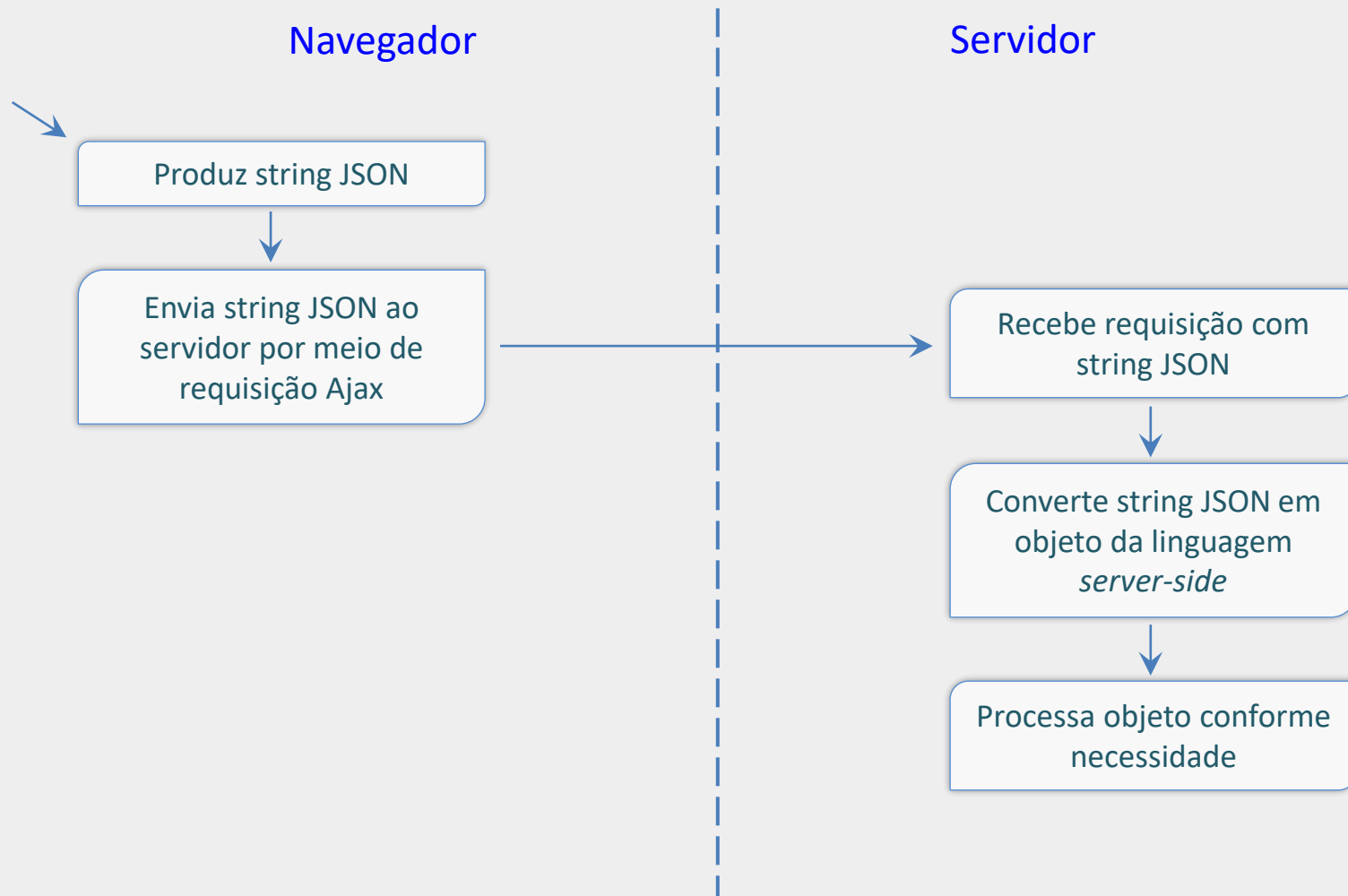
$cep = $_GET['cep'] ?? '';

if ($cep == '38400-100')
    $endereco = new Endereco('Av Floriano', 'Centro', 'Uberlândia');
else if ($cep == '38400-200')
    $endereco = new Endereco('Av Tiradentes', 'Fundinho', 'Uberl');
else
    $endereco = new Endereco('', '', '');

header('Content-type: application/json');
echo json_encode($endereco); // converte objeto PHP em string JSON
```

*Exemplo de uso da função `json_encode` do PHP para converter um objeto da linguagem em uma string **JSON** correspondente, que é enviada como resposta ao cliente.*

Ações Típicas de Requisição Enviando JSON



Exemplo de Requisição POST Enviando JSON

```
// objeto JavaScript a ser enviado
let objetoJS = {
  modelo : "Fusca",
  ano : "1970"
};

let xhr = new XMLHttpRequest();
xhr.open("POST", "cadastra.php");

xhr.onload = function () { ... }

// define cabeçalho HTTP 'Content-Type' para envio de JSON
xhr.setRequestHeader("Content-Type", "application/json");

// JSON.stringify converte um objeto JS para uma string JSON
xhr.send(JSON.stringify(objetoJS));
```

setRequestHeader deve ser chamada **depois** do método *open* e **antes** do método *send*

Submetendo Formulários com o XHR

Submetendo Formulários com o XHR

Há duas formas de submeter um formulário com o XHR

- Utilizando JavaScript puro
- Utilizando a API **FormData**

Submetendo Formulário com o FormData

1. Localiza-se o objeto do formulário
2. Cria-se um objeto `FormData` passando o obj. do formulário
3. Utiliza-se o método `POST` em `xhr.open`
4. Envia-se o objeto criado utilizando o método `xhr.send`

Submetendo Formulário com o FormData

```
let meuForm = document.querySelector("form");  
let formData = new FormData(meuForm);  
  
let xhr = new XMLHttpRequest();  
xhr.open("POST", "cadastra.php");  
xhr.send(formData);
```

Submetendo Formulário com o FormData

```
let meuForm = document.querySelector("form");  
let formData = new FormData(meuForm);  
  
formData.append("id", "123456");  
  
let xhr = new XMLHttpRequest();  
xhr.open("POST", "cadastra.php");  
xhr.send(formData);
```

Acrescentando campos com o método **append**

Enviando Dados por POST - 1ª Forma

```
let formData = new FormData();  
formData.append("modelo", "Fusca");  
formData.append("ano", "1970");  
  
let xhr = new XMLHttpRequest();  
xhr.open("POST", "cadastra.php");  
xhr.send(formData);
```

1. Cria-se um objeto *FormData*
2. Adiciona-se dados (nome, valor) com o método *append*
3. Utiliza-se o método *POST* em *xhr.open*
4. Envia-se o objeto com o método *xhr.send*

Enviando Dados por POST - 2ª Forma

```
let xhr = new XMLHttpRequest();  
xhr.open("POST", "cadastra.php");  
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");  
xhr.send("modelo=Fusca&ano=1970");
```

1. Utiliza-se o método *POST* em *xhr.open*
2. Utiliza-se o *setRequestHeader* para alterar o cabeçalho da requisição
3. Envia-se os dados pelo método *send* na forma de uma string/URL

*Codificação adicional pode ser necessária dependendo dos caracteres da string de dados.
Funções adicionais como **encodeURIComponent** podem ser necessárias.*

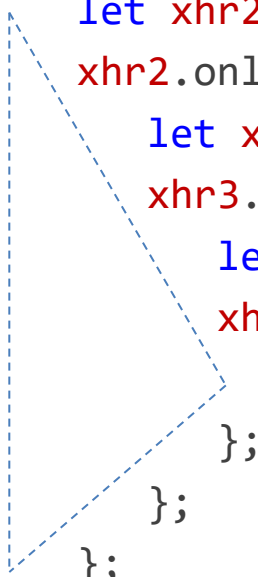
Ajax com a **API Fetch**

API Fetch

- Outra forma de realizar requisições Ajax
- Mais nova que o *XMLHttpRequest*
- Maior facilidade para encadear tarefas assíncronas
- 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);
      };
    };
  };
};
```

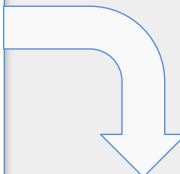


Este exemplo ilustra um possível encadeamento de requisições Ajax utilizando o XMLHttpRequest. O conceito de **promise** permite evitar situações como esta, contendo diversas chamadas em cascata de funções de callback (*callback hell*), o que pode tornar o código complexo e de difícil manutenção.

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



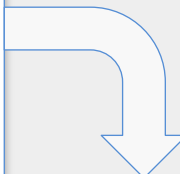
```
1. function getJSON(URL) {
2.     return fetch(URL)
3.         .then(response => response.json());
4. }
5. getJSON('URL1')
6.     .then(data1 => getJSON('URL2'))
7.     .then(data2 => getJSON('URL3'))
8.     .then(data3 => console.log(data3))
9.     .catch(error => console.error(error));
```

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());
}

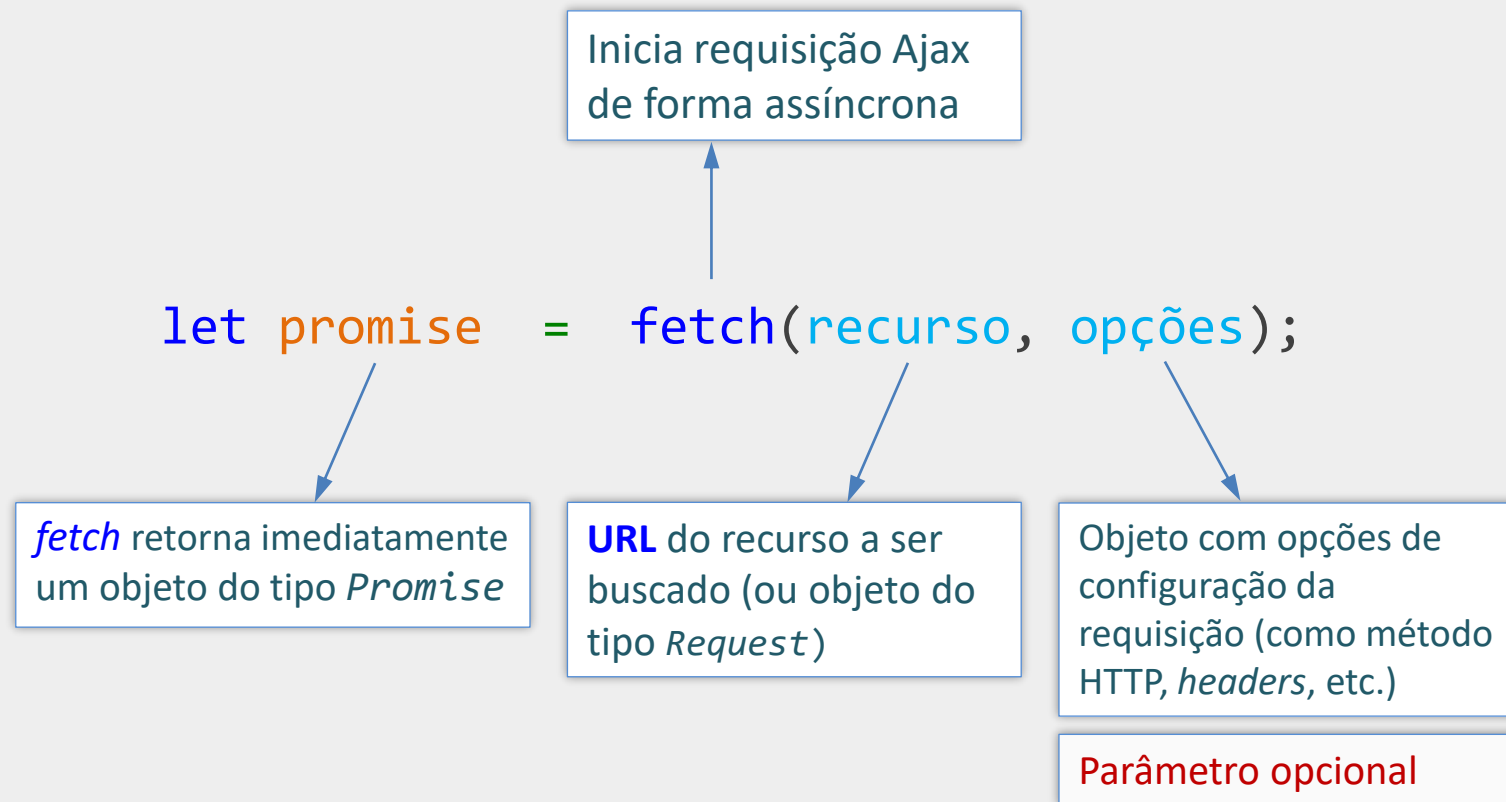
async function getData() {
    try {
        let data1 = await getJSON('URL1');
        let data2 = await getJSON('URL2');
        let data3 = await getJSON('URL3');
        console.log(data3);
    }
    catch (error) {
        console.error(error);
    }
}
```

Código equivalente com Fetch e async/await

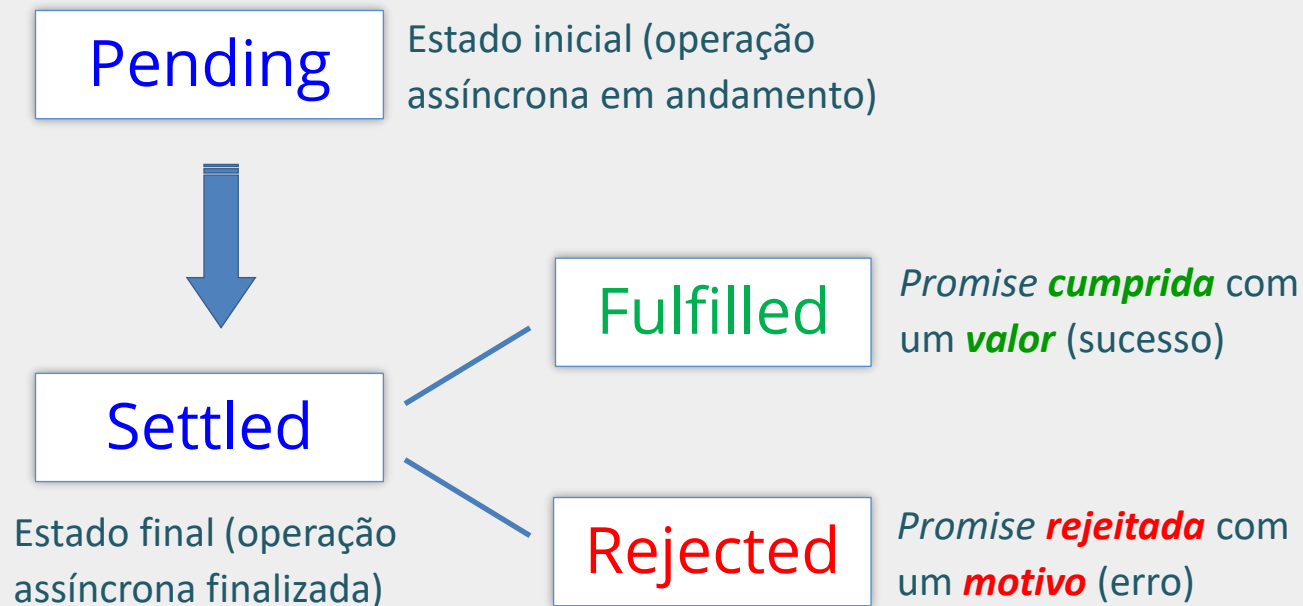
Introdução à Promises

- Em JavaScript, uma **promise** é um objeto que representa o resultado de uma tarefa assíncrona que pode finalizar no futuro
- Repres. uma “promessa” de concluir a tarefa em segundo plano
- Se finalizada com **sucesso**, produzirá um **valor**
- Se finalizada com **falha**, produzirá um **motivo** (erro)
- Funções de callback são indicadas para tratar o **valor/erro**

Introdução à *Promises* - Método *fetch*



Estados de uma *Promise*



Método *then*

- Método de um 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);
```

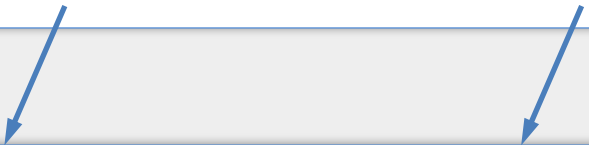
Indique uma função de *callback* a ser chamada quando a promise finalizar com sucesso (*fulfills*)

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).

Múltiplas Funções de *Callback*

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

- É possível adicionar múltiplas funções de *callback* concatenando *then*'s
- As funções serão executadas na ordem de inserção
- Permite iniciar nova tarefa assíncrona assim que terminar a anterior
- O *valor* produzido pela função anterior é passado para a seguinte
- Isto é possível porque o método *then* sempre retorna uma *nova promise*
- Que está associada à finalização de suas *callbacks*

Múltiplas Funções de *Callback*

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



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

Indentação mais comum com `.then` em nova linha

Múltiplas Funções de *Callback*

```
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 a *stream* de resposta contendo *string* JSON e converte em objeto JS
- Retorna *promise* que será cumprida com o objeto JS

`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.