Programação para Internet

Módulo 8

Requisições HTTP Assíncronas e a Técnica Ajax

(Mensagens HTTP, JSON, Técnica Ajax e conceitos, JavaScript Assíncrono, XMLHttpRequest, JavaScript Promises, API Fetch, Async/await)

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

Conteúdo do Módulo

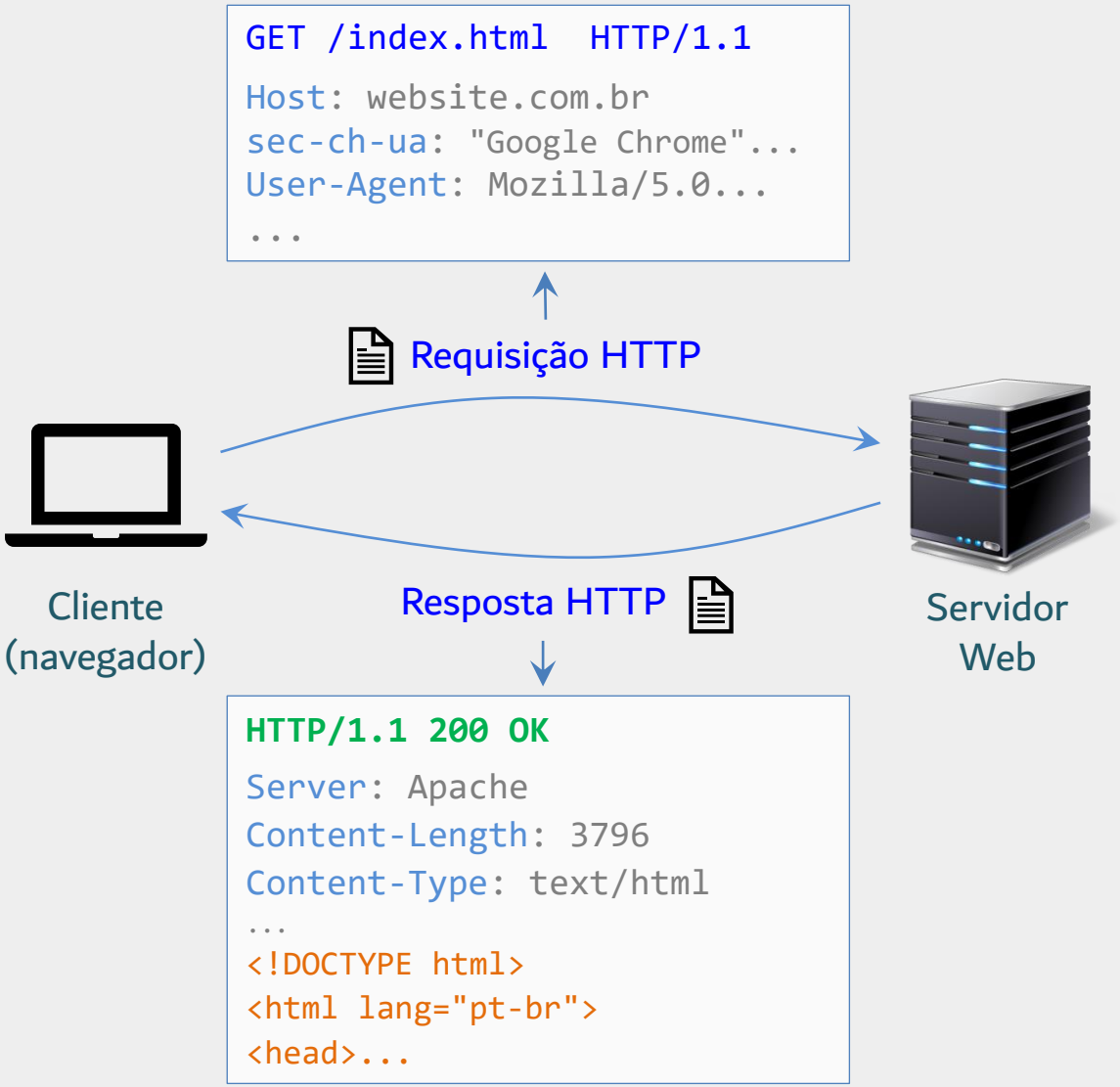
- Parte 1
  - Requisições e Respostas HTTP
  - Recapitulando o formato JSON
- Parte 2
  - Técnica Ajax e conceitos relacionados
  - Requisições assíncronas com o XMLHttpRequest
  - Tratamento de erros: rede x aplicação
  - Requisições com retorno em texto, imagem e JSON
  - Submissão assíncrona de formulários
- Parte 3
  - Requisições assíncronas com à API Fetch
  - JavaScript Promises
  - API Fetch: conceitos, recursos, exemplos
  - API Fetch com async / await

Requisições e Respostas HTTP
Introdução

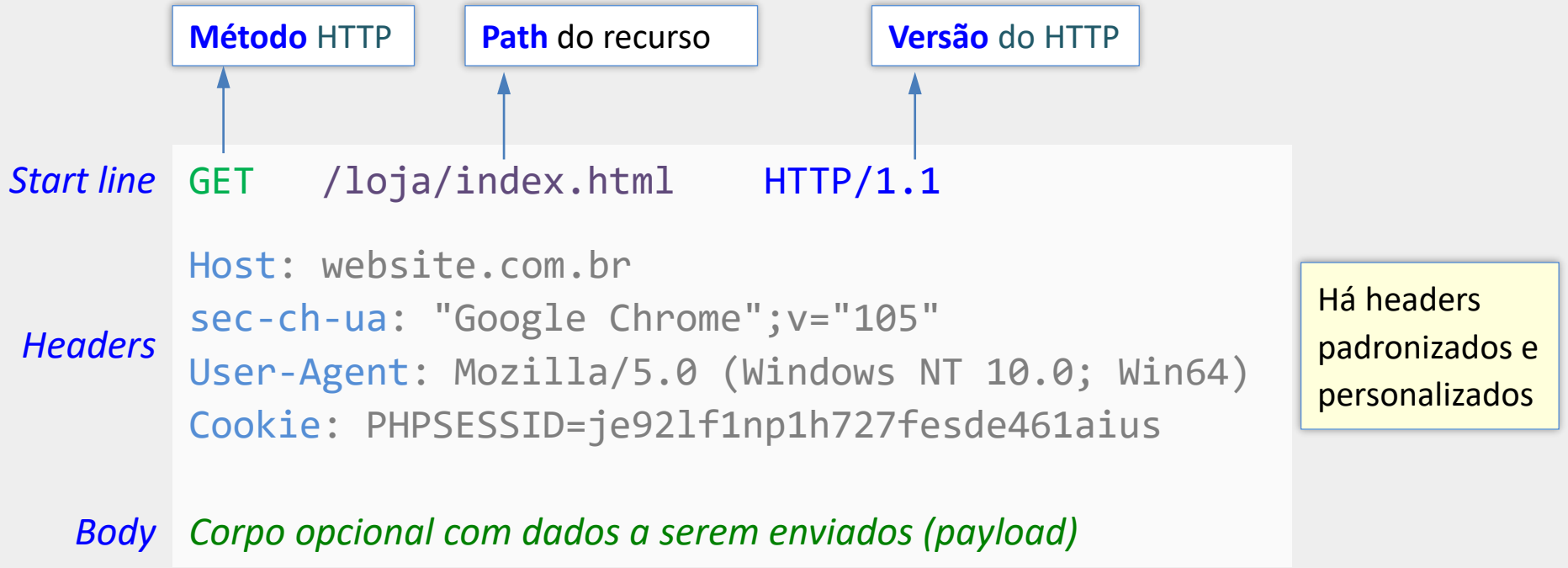
Requisições e Respostas HTTP

- Conceito crucial para entendimento da técnica Ajax
- A comunicação entre o navegador de internet e o servidor web ocorre por meio de uma série de **requisições** e **respostas** HTTP
- Uma requisição HTTP consiste essencialmente de uma mensagem contendo uma série de parâmetros que o navegador envia ao servidor para acessar recursos como arquivos HTML, arquivos de imagens etc.
- O servidor web, por sua vez, responde à requisição enviando de volta uma resposta HTTP, que contém o recurso solicitado juntamente com metadados

Requisições e Respostas HTTP



Exemplo Simplificado de Mensagem de **Requisição HTTP**



Uma mensagem de **requisição HTTP** é organizada em três partes:
1) a primeira linha (**start line** ou **request line**), que contém o método HTTP a ser utilizado e o caminho do recurso no servidor sendo requisitado;
2) as linhas de cabeçalho (**headers**) contendo parâmetros adicionais como a identificação do navegador de internet, dados de cookies etc.;
3) o corpo da requisição (**body**), que é a região da mensagem que pode conter dados da aplicação a serem enviados juntamente com a requisição (como dados de formulários, arquivos etc.)

Exemplo Simplificado de Mensagem de **Requisição HTTP**

Start line

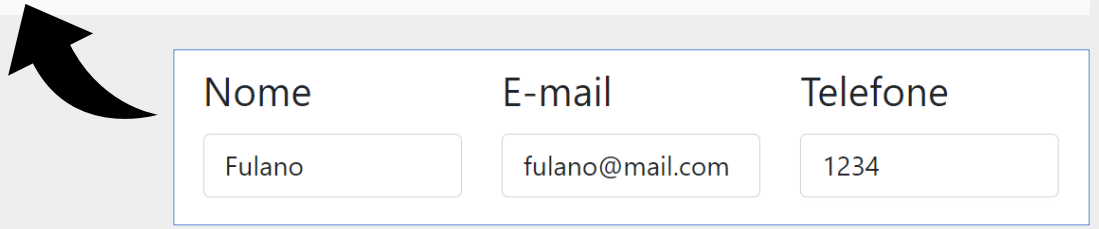
```
POST    /loja/cadastro.php    HTTP/1.1
```

Headers

```
Host: website.com.br
sec-ch-ua: "Google Chrome";v="105"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64)
Content-Type: application/x-www-form-urlencoded
Content-Length: 49
```

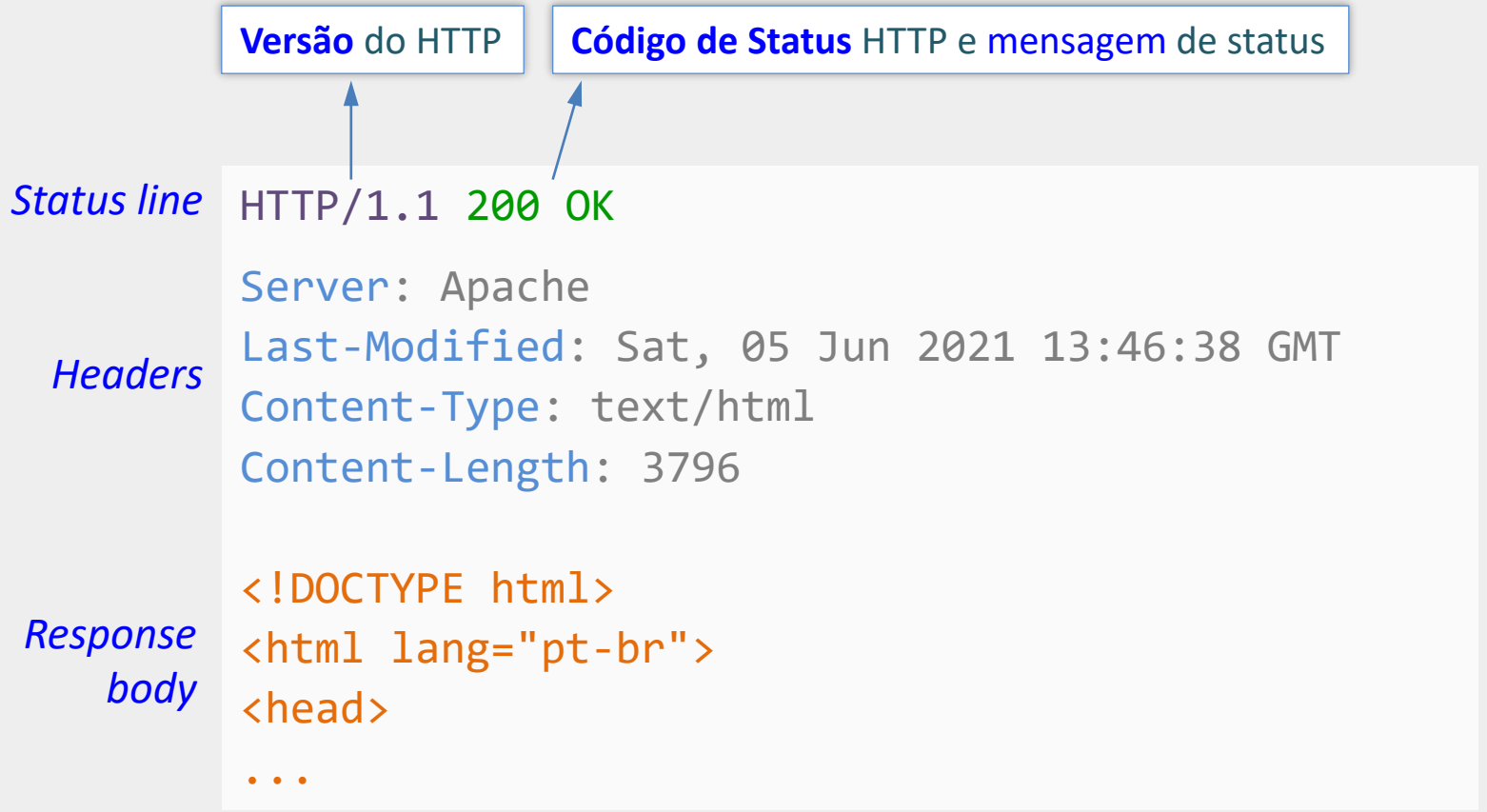
Body

```
nome=Fulano&email=fulano%40mail.com&telefone=1234
```



Quando um formulário HTML é submetido pelo método **POST**, por exemplo, o navegador envia uma requisição HTTP carregando os dados do formulário no **corpo** da mensagem de requisição. Quando o formulário é submetido pelo método **GET**, tais dados são enviados pela própria URL, na primeira linha (**start line**) da mensagem de requisição.

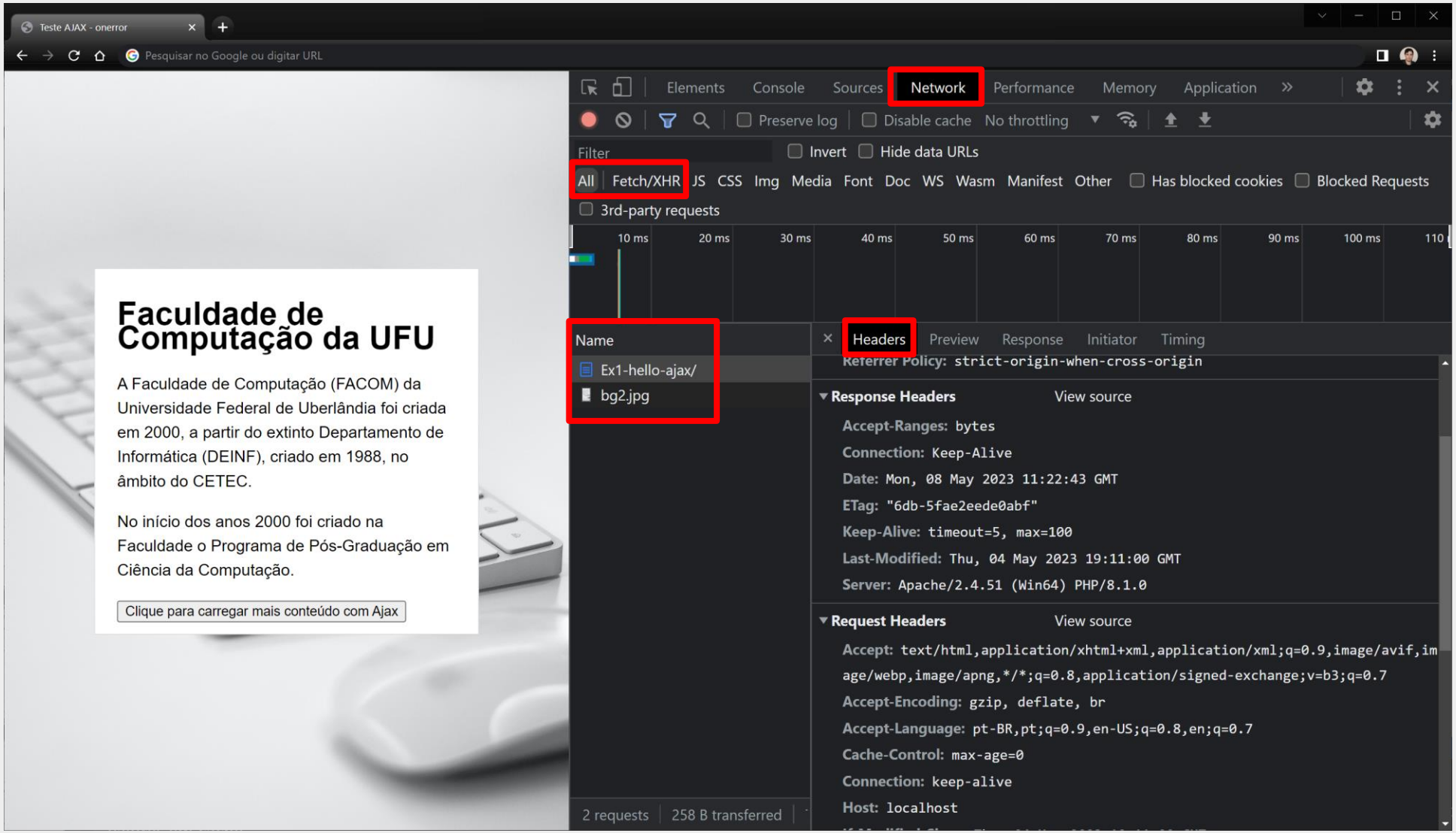
Exemplo Simplificado de Mensagem de **Resposta HTTP**



Uma mensagem de **resposta HTTP** também é organizada em três partes, de maneira similar à mensagem de requisição. A **linha de status** contém o código de status HTTP, que pode indicar sucesso ou falha na obtenção do recurso solicitado. Os **cabeçalhos** contêm metadados identificando, por exemplo, o tipo de conteúdo sendo retornado, o tamanho desse conteúdo e o servidor HTTP que produziu o conteúdo. O **corpo** normalmente contém o recurso que foi solicitado na requisição como código HTML, string JSON, dados de arquivos de imagem etc.

Verificando Dados de uma Requisição HTTP no Navegador

No Google Chrome: F12 → Network → F5 → Sel. Arquivo → Headers → Request Headers → View source



Alguns Códigos de Status HTTP

- 200 OK – resposta padrão para sucesso
- 302 Found – recurso encontrado, seguido por redirecionamento
- 304 Not Modified – recurso não modificado. Utilize a versão em cache
- 403 Forbidden – acesso ao recurso não autorizado
- 404 Not Found – recurso não encontrado
- 500 Internal Server Error – erro interno no servidor

Introdução ao Formato JSON

Introdução ao Formato JSON

- Acrônimo para **J**ava**S**cript **O**bject **N**otation
- É um formato para representação de dados de forma textual
- Por ser textual, é independente de linguagem
- Muito utilizado para intercâmbio de dados
  - Por exemplo, na comunicação cliente / servidor
- Permite a serialização de dados

Introdução ao Formato JSON

- Os dados são organizados em pares do tipo "propriedade" : valor, separados por vírgula;
- Os nomes das propriedades devem usar aspas duplas
- Objetos são representados utilizando chaves { }
- Vetores são representados utilizando colchetes [ ]
- Os valores das propriedades podem ser novos objetos, definidos com chaves

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

Exemplo de Script PHP Retornando Dados em JSON

```php
<?php
require 'conexaoMysql.php';
$pdo = mysqlConnect();

$sql = <<<SQL
  SELECT nome, telefone
  FROM aluno
  LIMIT 2
SQL;

try {
  $stmt = $pdo->query($sql);
  $arrayAlunos = $stmt->fetchAll(PDO::FETCH_OBJ);
  header('Content-Type: application/json; charset=utf-8'); // dados de cabeçalho da resposta HTTP
  echo json_encode($arrayAlunos);  // converte o array de alunos em uma string JSON
}
catch (Exception $e) {
  exit('Falha inesperada: ' . $e->getMessage());
}
```

Este script de exemplo retorna os 2 primeiros registros da tabela **aluno** como um *array* de objetos no formato **JSON**.

Repare que a função header do PHP foi utilizada para definição adequada do cabeçalho **Content-Type** da resposta HTTP para o valor **application/json**, uma vez que será retornado um conteúdo no formato JSON.

O *array* de objetos é convertido em uma string JSON correspondente utilizando a função **json_encode**. Exemplo de saída produzida:
[{"nome": "Fulano", "telefone": "123"}, {"nome": "Ciclano", "telefone": "456"}]

Exemplo de Script PHP Retornando Dados em JSON

```php
<?php
require 'conexaoMysql.php';
$pdo = mysqlConnect();
$cep = $_GET['cep'] ?? '';

$sql = <<<SQL
  SELECT rua, bairro, cidade
  FROM BaseEnderecos
  WHERE cep = ?
SQL;

try {
  $stmt = $pdo->prepare($sql);
  $stmt->execute([$cep]);
  $endereco = $stmt->fetch(PDO::FETCH_OBJ);
  header('Content-Type: application/json; charset=utf-8');
  echo json_encode($endereco);
}
catch (Exception $e) {
  exit('Falha inesperada: ' . $e->getMessage());
}
```

Script simplificado que recebe um CEP pela URL, busca por ele em tabela do banco de dados e retorna os dados do endereço no formato JSON.

Observe que o registro é recuperado como um objeto PHP, pois foi utilizada a opção *PDO::FETCH_OBJ* na chamada do método fetch.

Exemplo de resposta produzida:
{ "rua": "Av Floriano", "bairro": "Centro", "cidade": "Uberlândia" }

Introdução à Técnica Ajax

Surgimento da Técnica Ajax

- No final da década de 1990 e início dos anos 2000 começou a surgir um novo modelo de aplicações web, que ofereciam maior interatividade e responsividade
- Dois exemplos eram o Google Suggest (hoje chamado de Google Autocomplete) e o Google Maps
- Tais aplicações realizavam atualizações na página em segundo plano, de maneira quase instantânea, à medida que o usuário interagia com os elementos da interface

O que é a Técnica Ajax

- Ajax é uma técnica para realizar atualizações incrementais na página web
  - Permite atualizar uma página já carregada no navegador
  - Por meio de busca rápida por conteúdo adicional no servidor
  - E inserção na página dinamicamente (atualizando árvore DOM)
  - Dispensa a necessidade de carregar uma nova página inteiramente
- Utiliza requisições HTTP assíncronas (executadas em 2º plano)
  - Não interrompe a navegação do usuário
  - Não "congela" a interface
- **Objetivo**: aplicação mais ágil, eficiente e melhor experiência do usuário

Surgimento do Termo Ajax em si

adaptive path

Ajax: A New Approach to Web Applications

Jesse James Garrett is a founder of Adaptive Path

February 18, 2005

If anything about current interaction design can be called "glamorous," it's creating Web applications. After all, when was the last time you heard someone rave about the interaction design of a product that wasn't on the Web? (Okay, besides the iPod.) All the cool, innovative new projects are online.

Despite this, Web interaction designers can't help but feel a little envious of our colleagues who create desktop software. Desktop applications have a richness and responsiveness that has seemed out of reach on the Web. The same simplicity that enabled the Web's rapid proliferation also creates a gap between the experiences we can provide and the experiences users can get from a desktop application.

That gap is closing. Take a look at Google Suggest. Watch the way the suggested terms update as you type, almost instantly. Now look at Google Maps. Zoom in. Use your cursor to grab the map and scroll around a bit. Again, everything happens almost instantly, with no waiting for pages to reload.

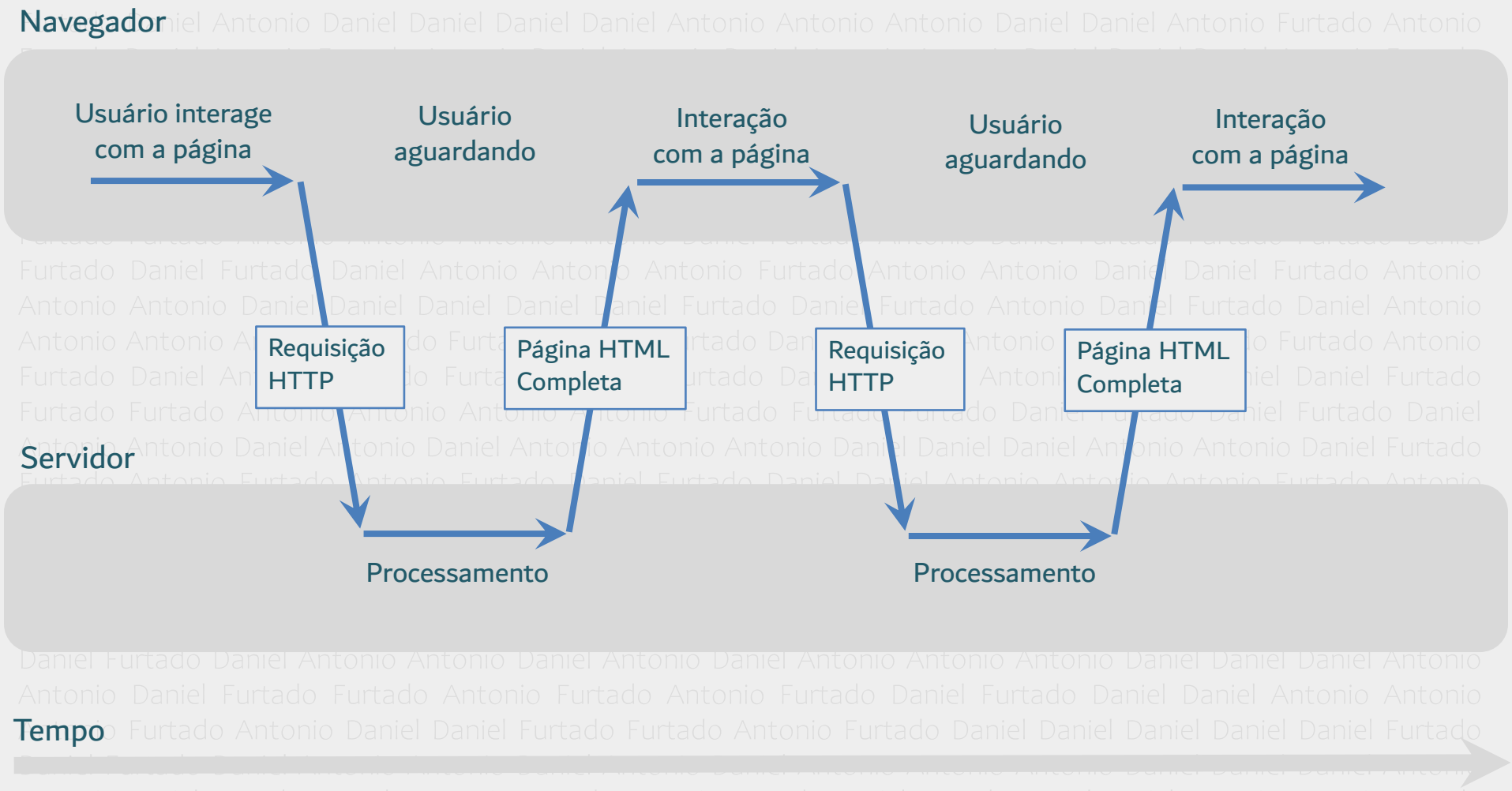
Google Suggest and Google Maps are two examples of a new approach to web applications that we at Adaptive Path have been calling Ajax. The name is shorthand for Asynchronous JavaScript + XML, and it represents a fundamental shift in what's possible on the Web.

Defining Ajax

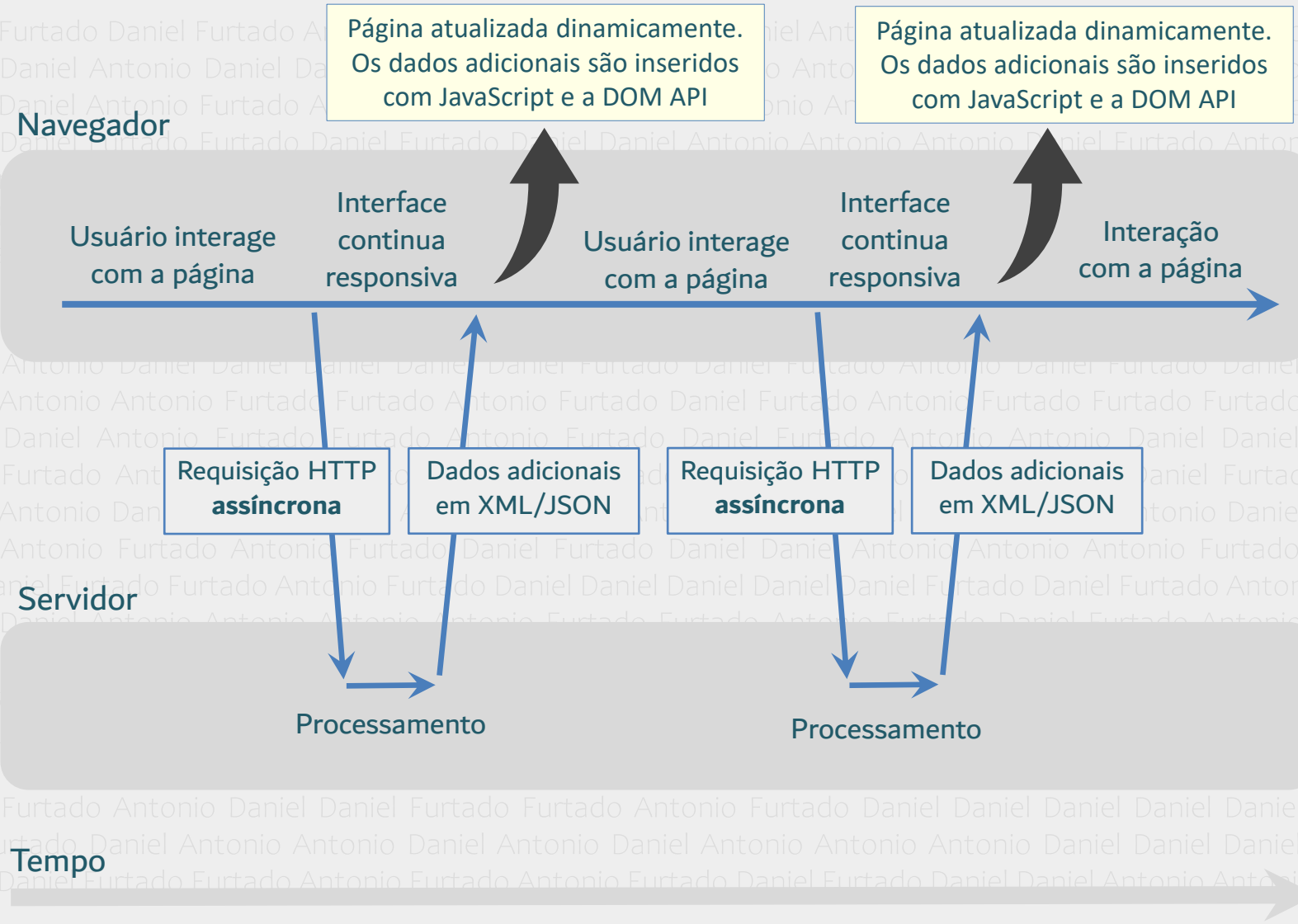
Ajax isn't a technology. It's really several technologies, each flourishing in its own right, coming together in powerful new ways. Ajax incorporates:

- O termo Ajax foi proposto em 2005 pelo canadense Jesse Garrett
- A técnica em sua essência já era utilizada, mas não havia um nome
- Jesse Garrett destaca que a técnica permite tornar as aplicações web mais parecidas com aplicações desktop, melhorando a responsividade

Aplicação Web Convencional (Sem Ajax)



Aplicação Web com Ajax



Exemplo

CEP

38408-100

Endereço

Av João Naves de Ávila

Bairro

Santa Mônica

Cidade

Uberlândia

Estado

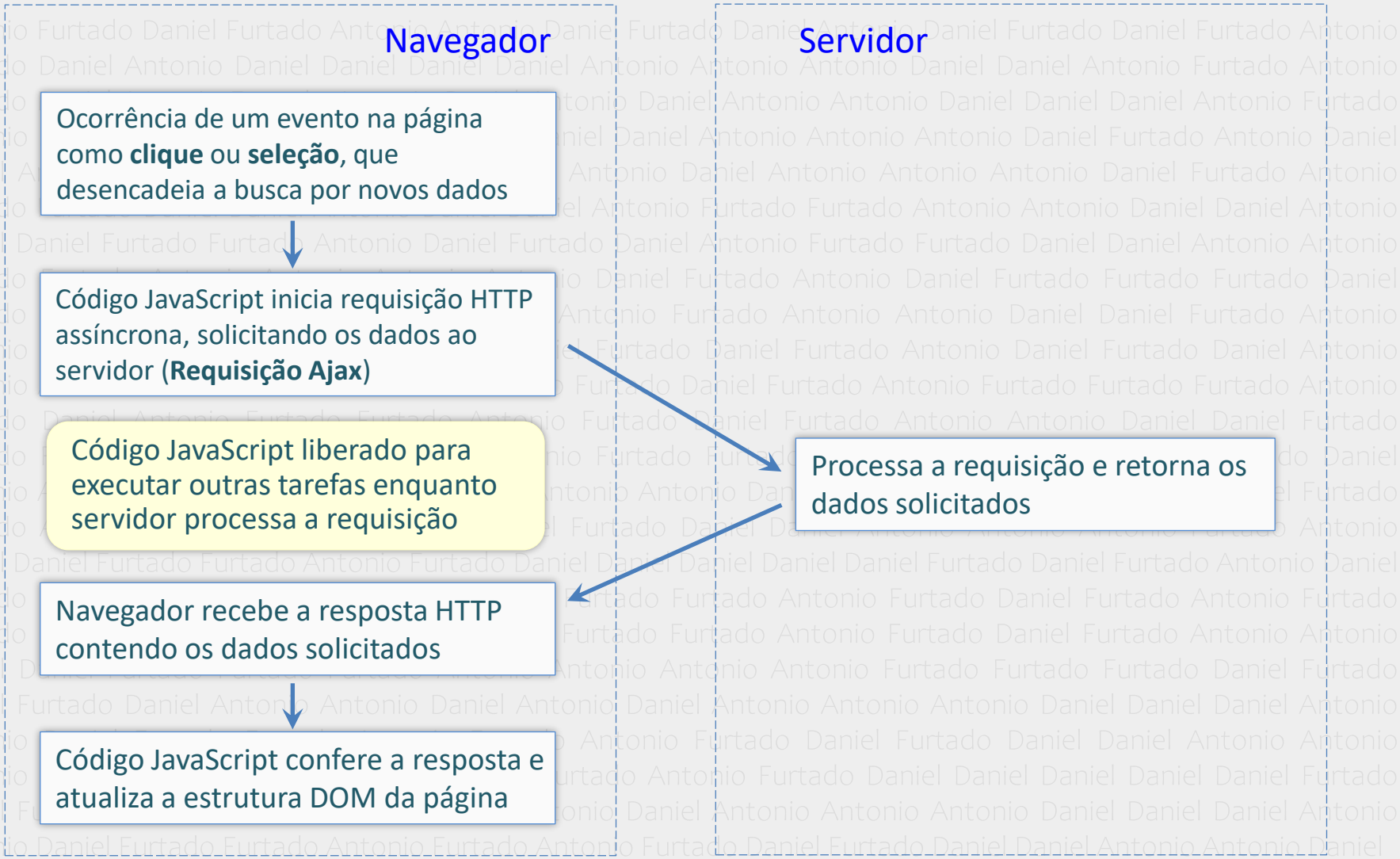
Minas Gerais

Preenchimento automático do formulário assim que o usuário digita o CEP

- Atualização quase instantânea
- Comunicação com servidor eficiente
- Troca de dados essenciais

Código JavaScript insere os dados no formulário (atualizando árvore DOM)

Sequência de Eventos Envolvidos na Técnica Ajax



Informações Adicionais sobre Ajax

- Ajax não é linguagem de programação, API ou biblioteca
- Ajax é uma técnica que combina várias tecnologias como:
  - HTML, CSS
  - XML / JSON
  - JavaScript
  - Árvore DOM
  - XMLHttpRequest / Fetch

Informações Adicionais sobre Ajax

- Principal formato (na época) para troca assíncrona de dados
- Outros formatos são mais comuns atualmente (JSON)

Ajax = **A**synchronous **Ja**vaScript and **X**ML

- Requisições HTTP em segundo plano (outra thread)
- Sem congelamentos da interface do usuário

Outros Exemplos de Aplicações

- Websites do tipo SPA (Single-Page Application)
  - Aplicação de Página Única
  - Conteúdo principal carregado uma única vez
  - Conteúdo adicional carregado dinamicamente com Ajax
    - Os elementos HTML podem ser gerados no próprio no navegador
    - O conteúdo em si pode vir em JSON
    - Traz o esforço de renderização para o lado cliente
- Buscas instantâneas oferecidas por lojas virtuais
- Rolagem infinita da página
  - Redes sociais, listagem de produtos etc.

Como realizar Requisições Ajax com JavaScript?

Nativo

- XMLHttpRequest
- API Fetch

Bibliotecas

- jQuery
- Axios

Ajax com o XMLHttpRequest

Objeto XMLHttpRequest (XHR)

- Projetado para buscar conteúdo em XML via requisições HTTP assíncronas
- Mas também suporta outros formatos como JSON, texto, HTML etc.
- Amplamente suportado pelos navegadores
- É uma API da Web - não faz parte da JavaScript em si
- Código mais longo, mas conceitualmente mais simples
  - Não utiliza Promises (o XHR se baseia em funções de callback)
  - Fácil aprendizado
- Dificuldades
  - Encadeamento de várias requisições de difícil manutenção (callback hell)

Principais Passos para Iniciar Requisição Ajax com o XHR

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

Exemplo Simples de Requisição Ajax sem Tratamento de Erros

O método **open** configura a requisição HTTP:
- O 1º parâmetro indica o método HTTP a ser utilizado (GET, POST, PUT etc.)
- O 2º parâmetro é o endereço no servidor do recurso sendo solicitado (arquivo, script etc.). Pode ser um caminho relativo ou URL completa.
- O 3º parâmetro indica se a requisição deve ser realizada de forma assíncrona (true) ou síncrona (false). Se omitido, será assíncrona (padrão).

A propriedade **onload** permite indicar uma função de callback que será chamada automaticamente quando a resposta enviada pelo servidor terminar de ser carregada pelo navegador. É a função que utilizará os dados requisitados. Deve ser indicada antes do envio da requisição.

Criação do objeto XMLHttpRequest para iniciar requisição Ajax

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

Neste exemplo a propriedade **responseText** é utilizada para acessar a resposta textual enviada pelo servidor (conteúdo do arquivo **dados.txt**).

Envia a requisição HTTP. No caso de requisição assíncrona, o código JavaScript prosseguirá normalmente enquanto a requisição será tratada em segundo plano.

Requisição Assíncrona x Síncrona com o XHR

Requisição Assíncrona

- O código JavaScript prossegue enquanto requisição é gerenciada pelo navegador em segundo plano (outra thread)
- É possível executar outras operações enquanto a requisição é tratada
- O andamento da requisição pode ser monitorado com eventos

Requisição Síncrona

- Considerada obsoleta (mdn web docs)
- O código JavaScript fica "bloqueado", aguardando resposta do servidor
- Não é recomendada, pois pode prejudicar a responsividade
- Se for utilizar, que seja fora da *thread* principal, com Web Workers
- Alguns recursos não estão disponíveis (fora de Web Workers)

Exemplo de Requisição Ajax Buscando Conteúdo Adicional

```html
<main>
  <h1>Faculdade de Computação da UFU</h1>
  <p>A Faculdade de Computação (FACOM) da Universidade...</p>
  <p>No início dos anos 2000 foi criado na Faculdade...</p>
  <button>Clique para carregar mais conteúdo com Ajax</button>
</main>

<script>
  function loadExtraContent() {
    let xhr = new XMLHttpRequest();
    xhr.open("GET", "conteudoAdicional.html", true);
    xhr.onload = function () {
      const main = document.querySelector("main");
      main.insertAdjacentHTML("beforeend", xhr.responseText);
    };
    xhr.send();
  }
  const button = document.querySelector("button");
  button.addEventListener("click", loadExtraContent);
</script>
```

Arquivo **conteudoAdicional.html**

```html
<h2>Cursos de Graduação</h2>
<p>A FACOM oferece os cursos de Bacharelado...</p>

<h2>Cursos de Pós-Graduação</h2>
<p>A oferece também os cursos de
  Mestrado Acadêmico e Doutorado em
  Ciência da Computação.</p>

<address>
  Campus Santa Mônica, Bloco 1A<br>
  Av. João Naves de Ávila, 2121, Santa Mônica<br>
  Uberlândia, MG<br>
  CEP 38400-902
</address>
```

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 erros a nível de rede, como:

- Falha na conexão com a internet
- Servidor não encontrado ou demorando 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 erros/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.

Verificando o código de status HTTP retornado

Código equivalente ao anterior, porém utilizando o método addEventListener

```
xhr.addEventListener("load", function () {
  if (xhr.status == 200)
    console.log(xhr.responseText);
  else
    console.error("Falha: " + xhr.status + xhr.responseText);
};

xhr.addEventListener("error", function () {
  console.log("Erro de rede");
};
```

Conteúdo Complementar

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; // milissegundos
xhr.ontimeout = function () {
  ...
};
```

XMLHttpRequest com JSON Tratamento Adequado da Resposta com Definição de xhr.responseType

Propriedade xhr.responseType

- A propriedade xhr.responseType possibilita especificar o tipo da resposta esperada do servidor, permitindo que os dados recebidos sejam tratados de maneira adequada pelo JavaScript
- Por exemplo, ao definir xhr.responseType = 'json', os dados retornados pelo servidor no formato json serão automaticamente convertidos em um objeto JavaScript correspondente, que estará disponível em xhr.response
- Outros valores comuns para xhr.responseType:
  - 'text' ou string vazia: valor padrão. Neste caso a resposta textual pode ser acessada em xhr.response ou xhr.responseText;
  - 'blob': xhr.response será um objeto Blob contendo dados binários
  - 'document': xhr.response será um objeto Document ou XMLDocument

OBS 1: A propriedade responseType não pode ser alterada quando a requisição for síncrona fora de web workers.
OBS 2: Ao definir responseType para um determinado valor, o desenvolvedor deve certificar-se de que o servidor está realmente enviando uma resposta compatível com esse formato. Se o servidor retornar dados incompatíveis com o responseType definido, o valor de xhr.response será null.

Requisição Ajax com Retorno em JSON

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "endereco.php?cep=38400-100");
xhr.responseType = 'json';
xhr.onload = function () {
  if (xhr.status != 200 || xhr.response === null) {
    console.log("Resposta não obtida");
    return;
  }
  const endereco = xhr.response;
  let form = document.querySelector("#meuForm");
  form.bairro.value = endereco.bairro;
  form.cidade.value = endereco.cidade;
};
xhr.onerror = function () {
  console.error("Requisição não finalizada");
  return;
};
xhr.send();
```

*Ao definir **responseType** com o valor 'json', a string JSON retornada será automaticamente convertida em um objeto JavaScript, que poderá ser resgatado 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 via JavaScript. Porém o desenvolvedor pode utilizar o ambiente de desenvolvimento do navegador para verificar o corpo da resposta HTTP com eventual mensagem de erro.*

Requisição Ajax com Retorno em JSON – Conversão Manual

```
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();
```

Este exemplo ilustra uma requisição Ajax para buscar conteúdo em JSON sem utilizar xhr.responseType = 'JSON'. Observe que neste caso a conversão da string JSON para um objeto JavaScript precisa ser feita manualmente utilizando a função JSON.parse.

Exemplo de Requisição Ajax para Carregar Imagem Dinamicamente e de Forma Assíncrona

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

xhr.onload = function () {
  // recupera os dados binários 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();
```

Este exemplo ilustra como uma imagem muito grande poderia ser carregada **em segundo plano** com a técnica Ajax.

Conteúdo Complementar

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 seria executado de forma **síncrona**, causando o **bloqueio** do JavaScript até a imagem ser carregada por completo (com possível congelamento da interface do usuário).

Conteúdo Complementar

Requisição Ajax Retornando HTML como Objeto Document

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "conteudoAdicional.html");
xhr.responseType = "document";

xhr.onload = function () {
  const doc = xhr.response;
  alert(doc.querySelector("h1").textContent);
};

xhr.send();
```

Mostrará o título "Programação para Internet"

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <title>Uma página de exemplo</title>
</head>

<body>
  <h1>Programação para Internet</h1>
  <p>Olá, mundo!</p>
</body>

</html>
```

Arquivo conteudoAdicional.html

Conteúdo Complementar

Recuperando Dados de Cabeçalho da Resposta HTTP

Ao utilizar o objeto XMLHttpRequest, é possível verificar os dados de cabeçalho da resposta HTTP utilizando o método getResponseHeader

```
...
xhr.onload = function () {
   let contType = xhr.getResponseHeader("Content-Type");
   if (contType != "application/json")
      return;
};
...
```

Exemplo de Requisição Enviando JSON

```
// objeto JavaScript a ser enviado como string JSON
let objetoJS = {
   cep : "38400-100",
   apikey : "abcdefg123456"
};

let xhr = new XMLHttpRequest();
xhr.open("POST", "buscaEndereco.php");
xhr.onload = function () { ... }

// define o cabeçalho HTTP 'Content-Type' para envio de JSON
// caso contrário, seria 'multipart/form-data' (xhr.open("POST",...)
xhr.setRequestHeader("Content-Type", "application/json");

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

setRequestHeader deve ser chamada **depois** do método open e **antes** do método send. A string JSON produzida por JSON.stringify será enviada pelo método send no corpo da mensagem de requisição HTTP.

Exemplo de Script PHP Recebendo JSON

```php
<?php
require "conexaoMysql.php";
$pdo = mysqlConnect();

// carrega a string JSON da requisição
// php://input retorna todos os dados que vem depois das linhas
// de cabeçalho HTTP da requisição, independentemente do tipo do conteúdo
$stringJSON = file_get_contents('php://input');
$dados = json_decode($stringJSON);
$cep = $dados->cep;
$apiKey = $dados->apiKey;

$sql = <<<SQL
  SELECT rua, bairro, cidade
  FROM BaseEnderecos WHERE cep = ?
SQL;
try {
  $stmt = $pdo->prepare($sql);
  $stmt->execute([$cep]);
  $endereco = $stmt->fetch(PDO::FETCH_OBJ);
  header('Content-Type: application/json; charset=utf-8');
  echo json_encode($endereco);
}
catch (PDOException $e) {
  exit('Falha inesperada: ' . $e->getMessage());
}
```

Conteúdo Complementar

Submetendo Formulários com o XHR

Submetendo Formulários com o XHR

- Há duas formas de submeter um formulário com o XHR
  1. Utilizando JavaScript puro (não será apresentada)
  2. Utilizando a API FormData

Submetendo Formulário com FormData

```
// cria-se um objeto FormData utilizando o objeto do formulário
let meuForm = document.querySelector("#meuFormulario");
let formData = new FormData(meuForm);

let xhr = new XMLHttpRequest();
xhr.open("POST", "cadastra.php");
xhr.send(formData); // envia-se o objeto utilizando o método send
```

Submetendo Formulário com FormData

Acrescentando campos com o método append

```
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);
```

Enviando Dados por POST com FormData

```
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 POST em xhr.open
4. Envia-se o objeto como parâmetro do método xhr.send

Enviando Dados por POST sem o FormData

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

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

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

Conteúdo Complementar

Propriedade onreadystatechange

```
let xhr = new XMLHttpRequest();

xhr.onreadystatechange = function () {
   if (this.readystate === this.DONE) {
      console.log(this.responseText);
   }
};

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

xhr.open("GET", "busca.php");
xhr.send();
```

*A propriedade **onreadystatechange** permite monitor o andamento da requisição. O valor de readystate varia de 0 (início) até 4 (término).*

Conteúdo Complementar

Parte 2
Requisições Assíncronas com a API Fetch

API Fetch

- Outra forma de realizar requisições Ajax
- Mais nova que o XMLHttpRequest
- Maior facilidade para encadear tarefas assíncronas (evitando callback hell)
- Maior clareza e simplicidade quando utilizada 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 esta situação.

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 o 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
(Será apresentado em detalhes ao longo deste material)

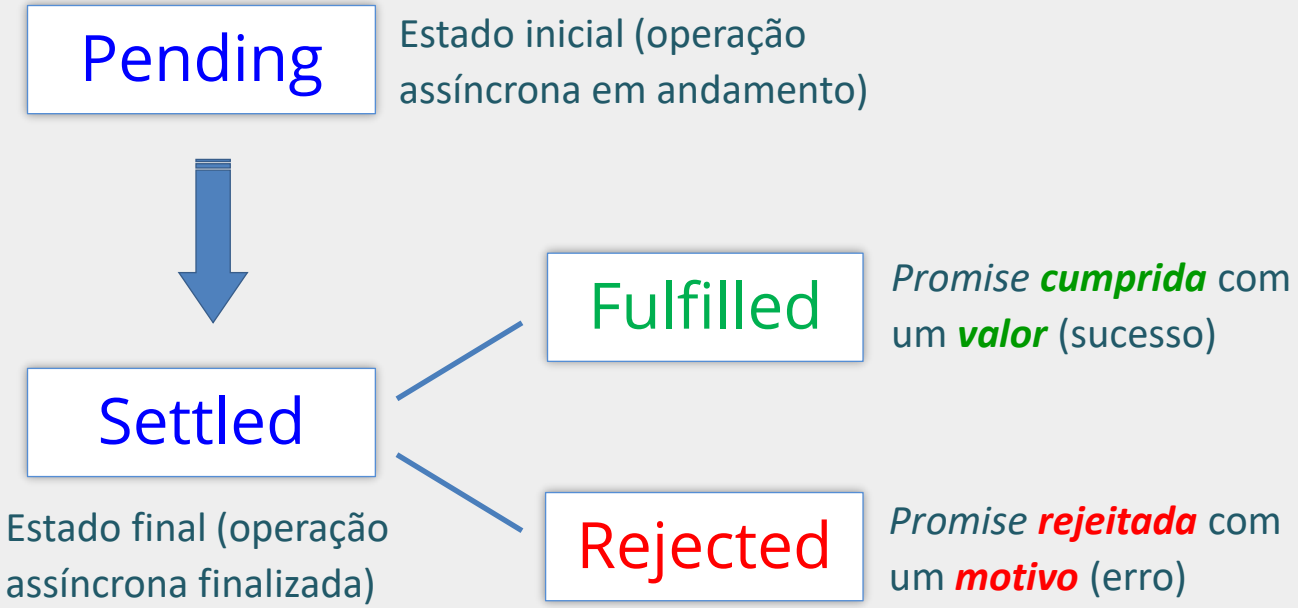
Introdução à Promises

- Promises em JavaScript simplificam o uso de **métodos assíncronos**
- Métodos assíncronos são executados em **segundo plano** (em outra thread)
  - Portanto, não retornam um valor final imediatamente
  - Mas retornam imediatamente um objeto do tipo promise, representando uma "promessa" de fornecer o valor final no futuro
- Em outras palavras, uma promise é um objeto que representa uma tarefa assíncrona a ser finalizada no futuro

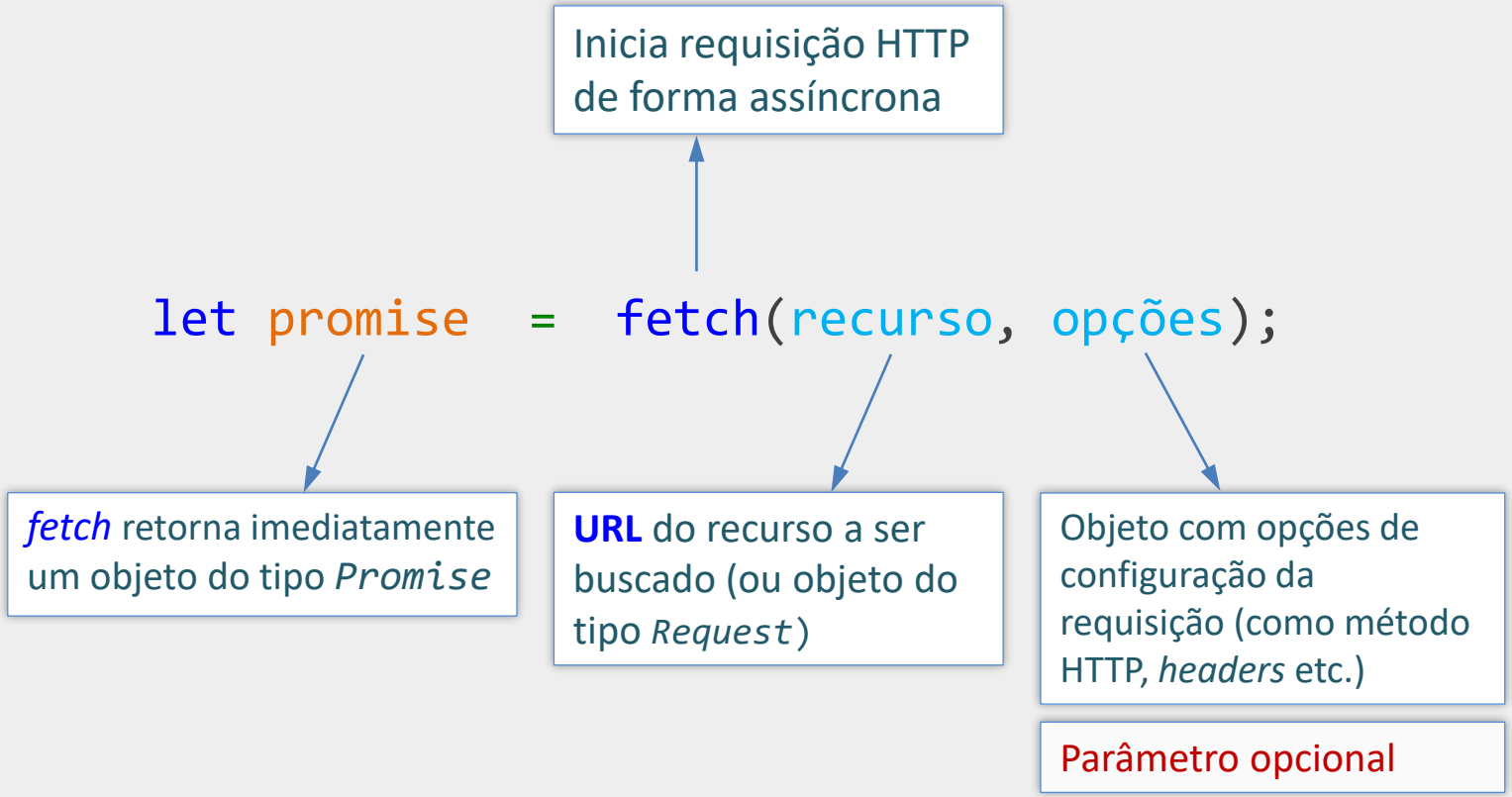
Introdução à Promises

- Se a tarefa assíncrona finalizar com **sucesso**, a promise será **cumprida** e produzirá um **valor** (resultado)
- Se a tarefa assíncrona finalizar com **falha**, a promise será **rejeitada** e produzirá um **motivo** (erro)
- Para processar o resultado ou tratar o erro devem ser indicadas funções de callback utilizando o método then() da promise

Estados de uma Promise



Introdução à Promises - Método fetch



Método then

- then é um método do objeto promise
- then permite resgatar o resultado ou o erro produzido pela tarefa:
  - Duas funções de callback podem ser passadas na chamada do método then;
  - A primeira delas é a callback de sucesso, que será chamada para processar o resultado da tarefa caso ela finalize com sucesso;
  - A segunda é a callback de erro, que será chamada para tratar o erro, caso a tarefa finalize com falha;
- O método then sempre retorna uma nova promise.

Exemplo – Passando callbacks para o método then

```
let promise = fetch(URL);

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

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

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

Opcional

As funções de callback recebem por parâmetro o resultado obtido pela operação assíncrona (ou erro produzido em caso de falha).

Callbacks definidas como funções anônimas

Garantias: 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).

Exemplo – Passando callbacks para o método then

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

Callbacks definidas com *arrow function*

Concatenando Múltiplos then's

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

- A concatenação de múltiplos then's permite executar tarefas assíncronas em sequência
- Neste exemplo, f1 seria executada após conclusão com sucesso da tarefa assíncrona associada à promiseA. Porém f1, por sua vez, pode iniciar outra tarefa assíncrona. Quando essa nova tarefa terminar com sucesso, f2 será executada para tratar o resultado e também poderá iniciar outra tarefa, cujo resultado com sucesso será tratado por f3
- 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 cada `.then` em uma 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 tratato por e1, e se a promise retornada pelo 1º then for rejeitada, o erro será tratado por e3 (f2 será ignorada).

Método catch

- Uma outra forma de indicar função para tratar erros é por meio do método **catch**
- Tem papel análogo à *".then(null, fe)"*
- É comumente utilizado no final do encadeamento para tratamento de erros concentrado no mesmo bloco
- Porém não precisa ser único nem usado necessariamente no final

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

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).*

Conteúdo Complementar

Método *fetch* - Exemplo

A promise retornada pelo método **fetch** será cumprida assim que a linha de status e as linhas de cabeçalho da resposta HTTP ficarem disponíveis. Como resultado será retornado um objeto do tipo **Response**, incluindo tais informações. Esse objeto também disponibiliza métodos para realizar o carregamento do corpo da resposta HTTP.

```
fetch("endereco.php?cep=38400-100")
 .then(response => response.json())
 .then(data => console.log(data))
 .catch(error => console.error(error))
```

O método **json**() do objeto response também é um método **assíncrono**, que retorna uma promise. O método carregará o corpo da resposta HTTP em segundo plano e converterá a string JSON em um **objeto JavaScript** correspondente. Em caso de sucesso a promise será cumprida tendo o objeto JavaScript como resultado.

Objeto JavaScript resultante do cumprimento da promise retornada por response.**json**()

Outros Métodos de um Objeto Response

`response.json()`

- Lê, de forma assíncrona, a stream de resposta (corpo da resp. http) 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

Método *fetch* - Exemplo

```
fetch("endereco.php?cep=38400-100")   // inicia requisição assíncrona
  .then(response => response.text())  // lê corpo da resposta como texto
  .then(bodyText => {
     const objetoJS = JSON.parse(bodyText);
     console.log(objetoJS))
  })
 .catch(error => console.error(error)) // mostra eventual erro
```

Conteúdo Complementar

Propriedades Comuns de um Objeto Response

- response.status - código de status HTTP retornado pelo servidor
- response.ok - true quando o servidor retorna um código de status entre 200 e 299
- response.headers - informações de cabeçalho retornadas pelo servidor

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 exemplo, a execução prosseguiria para a função de tratamento de erro do método .catch.

Preenchimento Automático de Endereço – Exemplo Completo

HTML

```html
<form>
  Informe o CEP no formato xxxxx-xxx (Ex. 38400-100)
  <input type="text" name="cep" id="cep">
  <input type="text" name="rua">
  <input type="text" name="bairro">
  <input type="text" name="cidade">
</form>
```

```html
<script>
  function buscaEndereco(cep) {
    if (cep.length != 9) return;
    let form = document.querySelector("form");
    fetch("busca-endereco.php?cep=" + cep)
      .then(response => {
        if (!response.ok)
          throw new Error(response.status);

        // Atenção para o 'return' aqui
        return response.json();
      })
      .then(endereco => {
        form.rua.value = endereco.rua;
        form.bairro.value = endereco.bairro;
        form.cidade.value = endereco.cidade;
      })
      .catch(error => {
        form.reset();
        console.error('Falha inesperada: ' + error);
      });
  }
  const inputCep = document.querySelector("#cep");
  inputCep.onkeyup = () => buscaEndereco(inputCep.value);
</script>
```

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))
```

- Neste exemplo, a 1ª requisição fetch poderá finalizar com sucesso e o resultado ser convertido no objeto JavaScript data1
- A 2ª requisição será iniciada após finalização da 1ª requisição e poderá utilizar data1
- Se a 2ª requisição finalizar com sucesso, produzirá data2
- Em caso de erro de rede na 1ª ou na 2ª requisição, ele será mostrado na janela de console do navegador

Exemplo de Requisições em Sequência

```
<main>
  <h2>Temperatura Local: <span id="temp">...</span></h2>
  <h2>Velocidade do Vento: <span id="wind">...</span></h2>
</main>
<script>
  // OBS: A função não trata todos os erros
  function buscaClimaLocal() {
    fetch('https://ipapi.co/json/')
      .then(response => response.json())
      .then(data => fetch(`https://api.open-meteo.com/v1/forecast?latitude=${data.latitude}&longitude=$
      .then(response => response.json())
      .then(data => {
        document.getElementById("temp").textContent = data.current_weather.temperature + 'º';
        document.getElementById("wind").textContent = data.current_weather.windspeed + ' km/h';
      })
      .catch(error => console.log(error));
  }
  window.onload = function () {
    buscaClimaLocal();
  }
</script>
```

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)
);
```

Conteúdo Complementar

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

Conteúdo Complementar

Enviando Formulário com Fetch e FormData

```
// 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...
```

Para enviar um formulário pelo método POST utilizando o FormData é necessário utilizar o segundo parâmetro do método fetch e informar um objeto com os detalhes adicionais da requisição. Neste exemplo estamos alterando o método para **POST** e especificando que o objeto **formData** deve ser enviado no corpo da requisição (como payload).

Enviando dados por POST com Fetch e FormData

```
let formData = new FormData();
formData.append("cep", "38400-100");

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

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

Enviando Dados em JSON com Fetch

```
// 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...
```

Este exemplo envia um objeto JavaScript, dados, após convertê-lo para o formato JSON com a função JSON.stringify. Repare que é necessário especificar o tipo dos dados que está sendo enviando no corpo da requisição. Isto é feito acrescentando o cabeçalho HTTP Content-Type com o valor 'application/json', pois estamos enviando um conteúdo no formato JSON.

Enviando Formulário em JSON com Fetch e FormData

```
// localiza formulário e cria objeto FormData
let meuForm = document.querySelector("form");
let formData = new FormData(meuForm);
let strJSON = JSON.stringify(Object.fromEntries(formData));

// opções da requisição
const options = {
   method: "POST",
   body: strJSON,
   headers: { 'Content-Type': 'application/json'  }
}

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

OBS Importante: um objeto FormData pode conter vários valores com a mesma chave. Isso pode ocorrer, por exemplo, quando há campos do tipo checkbox que utilizam o mesmo name. O método Object.fromEntries() descarta todos os registros com o mesmo valor de chave, mantendo apenas o último, o que pode levar à perda de dados.

Conteúdo Complementar

Enviando Arquivo com Fetch e 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...
```

Conteúdo Complementar

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()

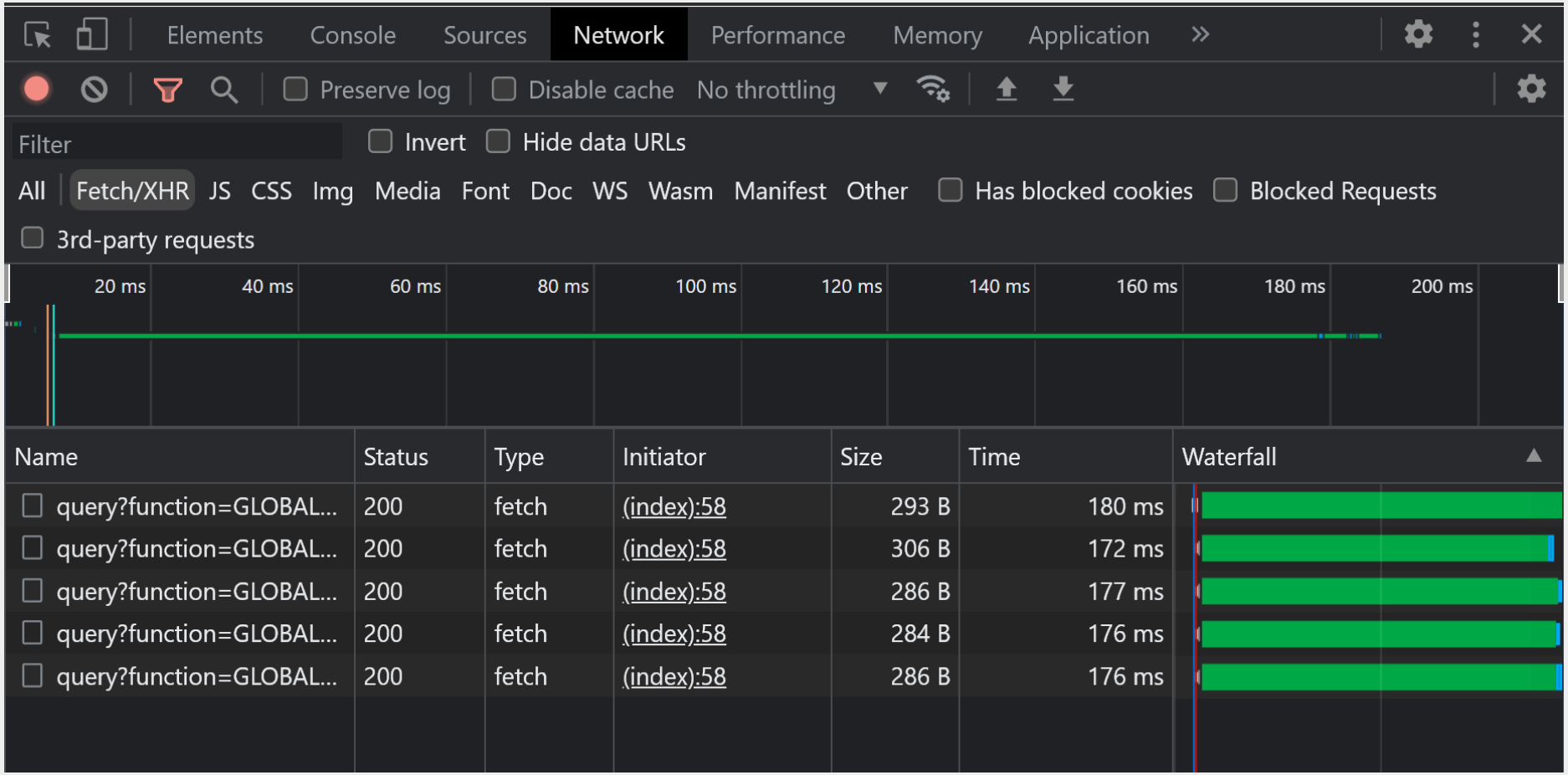
```
Promise.all([
    tarefaAssinc1(),
    tarefaAssinc2(),
    tarefaAssinc3(),
    tarefaAssincN()
])
.then(results => console.log(results)
.catch(error => console.log(error)
```

*O método Promise.all recebe por parâmetro um array de promises e retorna uma nova promise que será cumprida apenas quando **todas** as promises do array forem cumpridas. Se alguma promise for rejeitada, então a promise retornada também será rejeitada imediatamente e retornará o erro associado à primeira promise rejeitada. A promise retornada, se cumprida, resolverá em **array** contendo os resultados de **todas** as promises.*

Método Promise.all() – Exemplo

```
<script>
  function getQuote(codigoAcao) {
    return fetch(`https://www.alphavantage.co/query?function=GLOBAL_QUOTE&symbol=${codigoAcao}
      .then(response => response.json())
      .then(data => data['Global Quote']['05. price']);
  }
  function getPortfolio() {
    Promise.all([
      getQuote('ABEV3'),
      getQuote('ASAI3'),
      getQuote('AZUL4'),
      getQuote('B3SA3'),
      getQuote('BIDI11')
    ])
      .then(cotacoes => console.log(cotacoes))
      .catch(error => console.log(error))
  }
  window.onload = function () {
    getPortfolio();
  }
</script>
```

Analisando o Tempo das Requisições no Navegador



Métodos Similares a Promise.all()

`Promise.allSettled([p1, p2, ...])`

Retorna uma promise que é ***cumprida*** quando **todas** as promises p1, p2 etc. chegam ao estado final settled (são cumpridas **ou** rejeitadas).

A promise retornada é cumprida com array de resultados (e erros) de todas as promises.

`Promise.any([p1, p2, ...])`

Para que a promise retornada por Promise.any seja cumprida, basta que uma das promises p1, p2, ..., seja cumprida. A promise retornada será rejeitada apenas quando todas as promises p1, p2, ..., são rejeitadas.

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 para definir novas funções assíncronas, e o termo await, dentro dessas funções, para chamar outras funções assíncronas
- Não substitui as promises. É um nova forma de utilizá-las

Vantagens de Utilizar async/await

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

Função Assíncrona e Expressões async/wait

Chamada assíncrona utilizando o then

```
function minhaFuncao() {
  funcaoAssincrona("parametros")
   .then(result => console.log(result))
   .catch(error => console.error(error))
}
```

Chamada assíncrona utilizando await

```
async function minhaFuncao () {
   try {
     result = await funcaoAssincrona("parametros");
     console.log(result))
   } catch(e) { console.error(e) }
}
```

- await pode ser usada na chamada de funções que retornam promises
- Permite que funções assíncronas sejam chamadas no "estilo" das síncronas
- Suspende a execução de minhaFuncao 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 Expressões async/wait

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

- await é permitida apenas dentro de funções definidas com async*
  - Utiliza-se a palavra reservada async antes de function, como no exemplo acima
- A suspensão com o await **não** causa um bloqueio da thread principal
  - Causa apenas a suspensão da função async onde ocorre a chamada assíncrona (minhaFuncaoAssincrona)
  - Isso significa que é possível executar outras funções, tratar eventos, responder à interface etc.
- Função async que não contém await é executada de foma síncrona
- Funções definidas com async sempre retornam uma promise
  - Se o valor de retorno não é explicitamente uma promise, então ele será autom. encapsulado em uma

* e também dentro do corpo de módulos

Função Assíncrona e Expressões async/wait – Exemplo

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

Exemplo simplificado, sem tratamento de erros

Qual será a saída apresentada no console?

```
async function buscaEndereco() {
  console.log("A");
  const response = await fetch("endereco.php?cep=38400-100");
  const endereco = await response.json();
  console.log("B");
}
window.onload = function () {
  buscaEndereco();
  console.log("C");
}
```

Qual será a saída apresentada no console?

```
async function buscaEndereco() {
  console.log("A");
  const response = await fetch("endereco.php?cep=38400-100");
  const endereco = await response.json();
  console.log("B");
}
window.onload = function () {
  buscaEndereco();
  console.log("C");
}
```

Considerando que a requisição finalize com sucesso (e o servidor leva algum tempo para responder), será apresentada no console a sequência A C B.

A letra A é apresentada primeiro porque o código é executado de forma síncrona até a declaração **await fetch**. Nesse momento, a execução da função buscaEndereco é **suspensa** enquanto a requisição é tratada de **forma assíncrona** em outra thread. Consequentemente, a execução do código na thread principal prossegue para a próxima linha depois da chamada de buscaEndereco() e apresenta a letra C. Portanto, a thread principal **não fica** bloqueada enquanto a requisição é tratada, mas a função assíncrona contendo a expressão await é suspensa (buscaEndereco).

Finalmente, quando a requisição finalizar retornando o objeto response, a execução em buscaEndereco é retomada e inicia outra operação assíncrona (response.json()), que causa uma nova suspensão de buscaEndereco. Quando a leitura e conversão do JSON finalizar em segundo plano e resultar no objeto endereco, então a execução de buscaEndereco será retomada novamente e apresentará a letra B.

Tratamento de Erros com async/await

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

async/await possibilita o tratamento de erros utilizando um bloco try/catch tradicional. Se a promise vinculada à função assíncrona for **rejeitada** então a **execução será deslocada** para o bloco **catch**. Se isso acontecer com funcAssinc1, por exemplo, então funcAssinc2 não será executada.

Tratamento de Erros com async/await – Exemplo

```
async function buscaEndereço(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);
  }
}
```

Neste exemplo, o bloco catch mostrará um erro nas seguintes situações:

1) Se a requisição não finalizar devido a um erro de rede (a promise retornada pelo fetch é rejeitada)

2) Se a requisição retornar um código de status fora da faixa 200-299

3) Se houver um erro na leitura ou conversão dos dados em JSON

Tratamento de Erros com async/await – Exemplo

```
async function buscaEndereço(cep) {
  try {
    const response = await fetch("endereco.php?cep=" + cep);
    if (! response.ok)
      throw new Error("Falha inesperada: " + response.status);

    const bodyText = await response.text();
    const endereco = JSON.parse(bodyText);
    console.log(endereco);
  }
  catch (e) {
    console.log(bodyText ?? "");
    console.error(e);
  }
}
```

Este exemplo lê o corpo da resposta HTTP de forma textual e converte a string JSON em objeto JavaScript utilizando a função JSON.parse. Caso o servidor não retorne uma string JSON válida (mas uma mensagem de erro de script, por exemplo), o conteúdo textual retornado (mensagem de erro) poderá ser visto no console do navegador. No exemplo do slide anterior, o desenvolvedor precisaria utilizar o ambiente de desenvolvimento do navegador para visualizar o conteúdo da resposta HTTP.

Conteúdo Complementar

Encadeamento de Requisições com fetch e async/await

```
async function buscaClimaLocal() {
  try {
    // busca a latitude e longitude local
    const response1 = await fetch('https://ipapi.co/json/');
    if (! response1.ok) throw new Error(response1.statusText);
    local = await response1.json();

    // busca informações do clima local passando a latitude e a longitude como parâmetro
    const response2 = await fetch(`https://api.open-meteo.com/v1/forecast?latitude=${local.latitude}&lon
    if (! response2.ok) throw new Error(response2.statusText);
    clima = await response2.json();

    // apresenta as informações do clima
    document.getElementById("temp").textContent = clima.current_weather.temperature + 'º';
    document.getElementById("wind").textContent = clima.current_weather.windspeed + ' km/h';
  }
  catch (error) {
    console.log(error);
    alert('Não foi possível obter a temperatura local');
  }
}
```

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 de execução total 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 de execução total pode ser próximo do tempo de execução da tarefa 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);
}
```

Conteúdo Complementar

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, o catch da direita irá capturar apenas a 1º exceção lançada. Caso as outras promises lancem erros adicionais, eles serão propagados, mas não catpurados (podendo aparecer warnings).

Conteúdo Complementar

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
- https://github.com/public-apis/public-apis
- https://rapidapi.com/collection/list-of-free-apis
- Jasse J. Garrett. **Ajax: A New Approach to Web Applications**, Adaptive Path, 2005.
- David Flanagan. **JavaScript: The Definitive Guide**. 7ª ed., 2020.
- Jon Duckett. **JavaScript and JQuery: Interactive Front-End Web Development**.