



JAVASCRIPT ASSÍNCRONO (PROMESSAS / API FETCH / ASYNC - AWAIT)

FAPESC – DESENVOLVEDORES PARA TECNOLOGIA DA INFORMAÇÃO

HERCULANO DE BIASI
herculano.debiasi@unoesc.edu.br



TÓPICOS

- Promessas (*Promises*)
- API Fetch
- *Async / Await*

PROMESSAS (*PROMISES*)

- Promise (promessa) é um padrão de desenvolvimento muito usado no JavaScript que representa fluxos de ações em operações assíncronas, mais especificamente, o tratamento da conclusão destas operações
- Um objeto *promise* guarda a promessa de que a função que o gerou irá, eventualmente, em algum momento desconhecido no futuro terminar e retornar um resposta
- Uma *promise* representa a eventual conclusão (ou falha) de uma operação assíncrona e seu valor resultante
- Elas foram implementadas no ES6, antes disso a maioria das funções utilizava callbacks (funções de retorno de chamada) assíncronos
- Antes do ES6 elas apareceram em bibliotecas como a Dojo Toolkit e jQuery
- Uma promessa define uma ação que será executada no futuro, ou seja, ela pode ser resolvida (com sucesso) ou rejeitada (com erro)

PROMESSAS (*PROMISES*)

- Ideia geral, estados (*pending*, *fulfilled* e *rejected*) e parâmetros (*resolve* e *reject*)
 - Quando uma promessa foi cumprida (*fulfilled*) ou rejeitada (*rejected*) é dito que ela passou para o estado *settled* (resolvido decidido, assentado, determinado)

... ➔ Um valor que pode estar disponível agora, no futuro ou nunca

Promise



... ➔ Status de Promise

Pending (pendente): Estado inicial, que não foi realizada nem rejeitada
Fulfilled (realizada): Sucesso na operação
Rejected (rejeitado): Falha na operação

PROMESSAS (*PROMISES*)

■ Verificação de estado: Pendente

```
aula25 - 1_promessa_pendente.js

1  let promessa = new Promise((resolve, reject) => {
2    try {
3      // Promise executada com sucesso
4      setTimeout(() => {
5        // Ponto de solução da promessa
6        resolve('Promessa concluída com sucesso!');
7      }, 3000);
8    } catch (e) {
9      // Erro na execução da promessa
10     setTimeout(() => {
11       // Promessa rejeitada
12       reject(e);
13     }, 3000);
14   }
15 });
16
17 console.log(promessa);
```

PROBLEMAS SAÍDA CONSOLE DE DEPURAÇÃO TERMINAL SQL CONSOLE

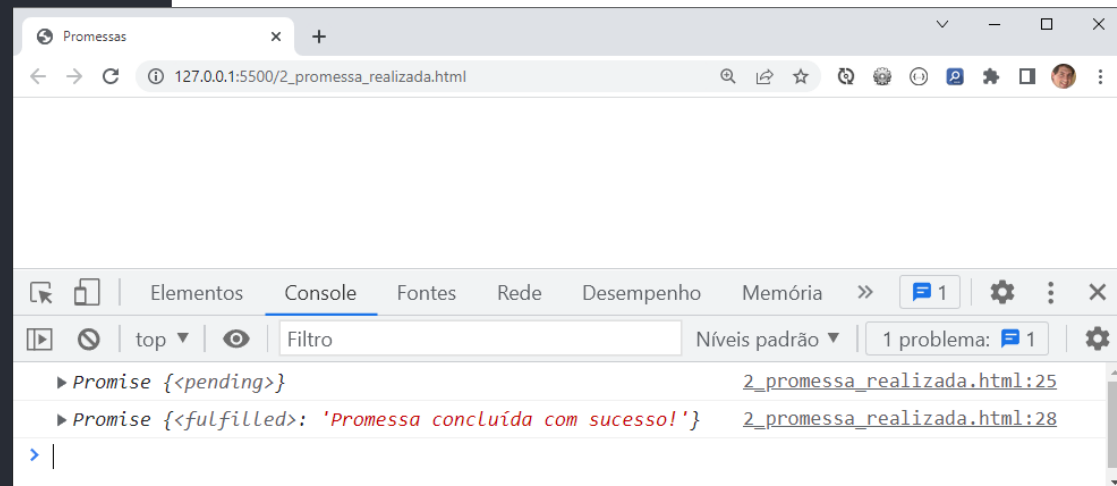
```
[Running] node "c:\html\aula25\1_promessa_pendente.js"
Promise { <pending> }
```

PROMESSAS (*PROMISES*)

■ Verificação de estado: Realizada com sucesso

aula25 - 2_promessa_realizada.html

```
1 <!DOCTYPE html>
2 <html lang="pt-BR">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <title>Promessas</title>
9
10  <script>
11    let promessa = new Promise((resolve, reject) => {
12      try {
13        // Promise executada com sucesso
14        setTimeout(() => {
15          resolve('Promessa concluída com sucesso!');
16        }, 3000);
17      } catch (e) {
18        // Erro na execução da promessa
19        setTimeout(() => {
20          reject(e);
21        }, 3000);
22      }
23    });
24
25    console.log(promessa);
26
27    setTimeout(() => {
28      console.log(promessa);
29    }, 5000);
30  </script>
31 </head>
32
33 <body>
34 </body>
35
36 </html>
```



PROMESSAS (*PROMISES*)

■ Verificação de estado: Rejeitada

Definição promessa: 14:00:00

▼ Promise {<pending>} ⓘ

- [[Prototype]]: Promise
- [[PromiseState]]: "rejected"
- [[PromiseResult]]: Error: Forçando um erro! at http://127.0.0.1:5500/3_promessa_rejeitada.html:14:23

Dentro do catch 3s: 14:00:03

✖ Uncaught (in promise) Error: Forçando um erro!
at 3_promessa_rejeitada.html:14:23
at new Promise (<anonymous>)
at 3_promessa_rejeitada.html:11:24

Bloco 5s: 14:00:05

Promise {<rejected>}: Error: Forçando um erro!
at http://127.0.0.1:5500/3_promessa_rejeitada.html:14:23
at new Promise (<anonymous>) ⓘ

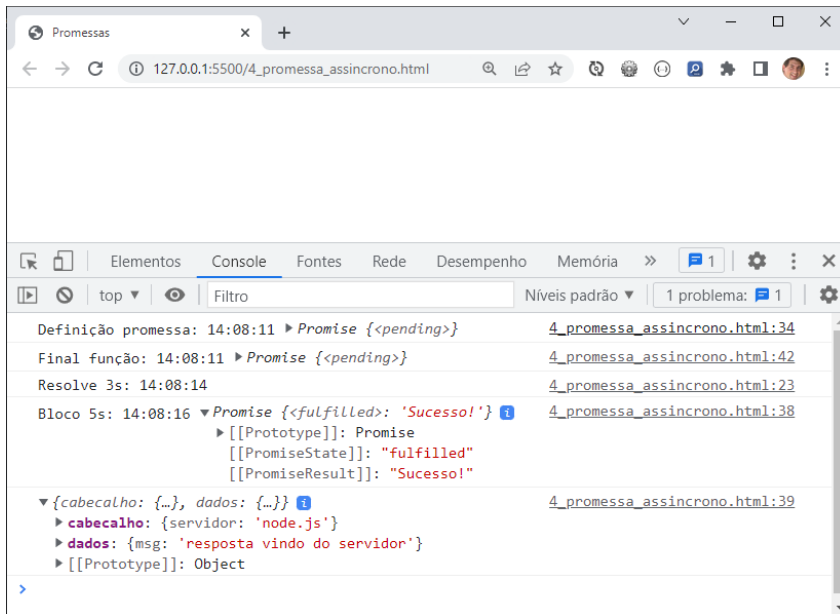
- [[Prototype]]: Promise
- [[PromiseState]]: "rejected"
- [[PromiseResult]]: Error: Forçando um erro! at http://127.0.0.1:5500/3_promessa_rejeitada.html:14:23 at new Promise (<anonymous>) at http://127.0.0.1:5500/3_promessa_rejeitada.html:11:24

```
aula25 - 3_promessa_rejeitada.html

1 <!DOCTYPE html>
2 <html lang="pt-BR">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <title>Promessas</title>
9
10  <script>
11    let promessa = new Promise((resolve, reject) => {
12      try {
13        // Forçando uma condição de erro
14        throw new Error('Forçando um erro!');
15
16        // Promise executada com sucesso (não será executado)
17        setTimeout(() => {
18          resolve('Promessa concluída com sucesso!');
19        }, 3000);
20      } catch (e) {
21        // Erro na execução da promessa
22        setTimeout(() => {
23          console.log('Dentro do catch 3s:', new Date().toISOString().substr(11,8));
24          reject(e);
25          }, 3000);
26      }
27    });
28
29    console.log('Definição promessa:', new Date().toISOString().substr(11,8), promessa);
30
31    setTimeout(() => {
32      console.log('Bloco 5s:', new Date().toISOString().substr(11,8), promessa);
33    }, 5000);
34  </script>
35 </head>
36
37 <body>
38 </body>
39
40 </html>
```

PROMESSAS (PROMISES)

■ Exemplo de simulação de uma requisição assíncrona



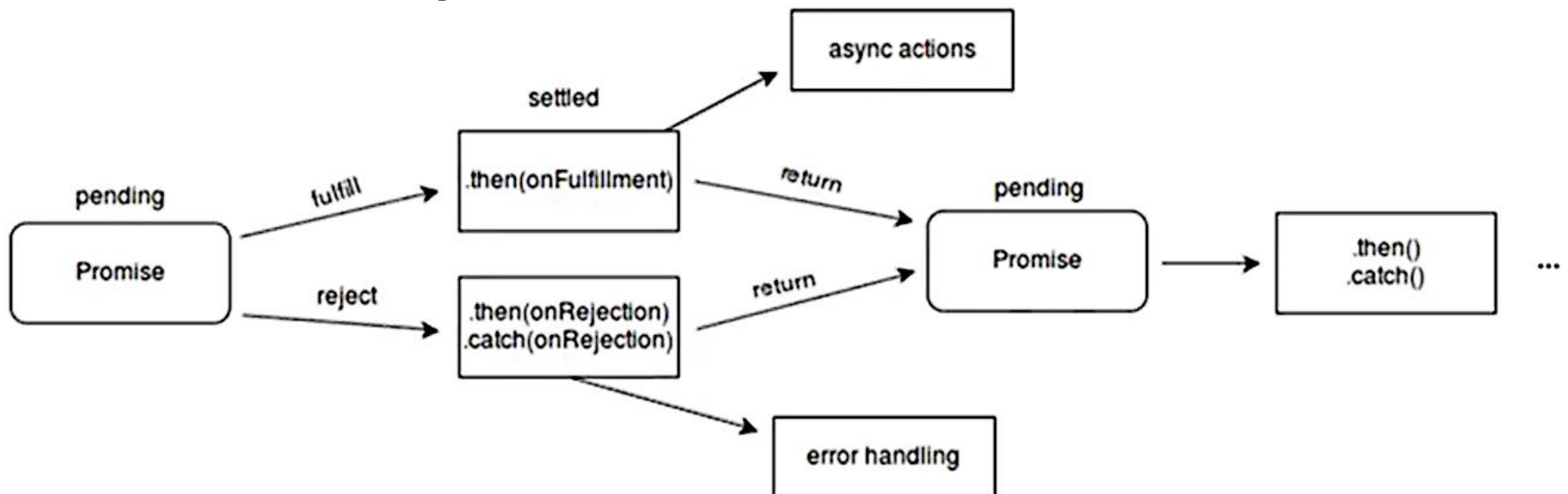
```
aula25 - 4_promessa_assincrono.html

1 <!DOCTYPE html>
2 <html lang="pt-BR">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <title>Promessas</title>
9
10  <script>
11    // Simulando uma resposta de uma requisição HTTP
12    let resposta = {};
13
14    let promessa = new Promise((resolve, reject) => {
15      try {
16        // Promise executada com sucesso (não será executado)
17        setTimeout(() => {
18          resposta = {
19            cabecalho: { servidor: 'node.js' },
20            dados: { msg: 'resposta vindo do servidor' }
21          };
22
23          console.log('Resolve 3s:', new Date().toISOString().substr(11,8));
24          resolve('Sucesso!');
25        }, 3000);
26      } catch (e) {
27        // Erro na execução da promessa
28        setTimeout(() => {
29          reject(e);
30        }, 3000);
31      }
32    });
33
34    console.log('Definição promessa:',
35      new Date().toISOString().substr(11,8),
36      promessa);
37
38    // Bloco 5s
39    setTimeout(() => {
40      console.log('Bloco 5s:', new Date().toISOString().substr(11,8), promessa);
41      console.log(resposta);
42    }, 5000);
43
44    console.log('Final função:', new Date().toISOString().substr(11,8), promessa);
45  </script>
46 </head>
47
48 <body>
49 </body>
50
51 </html>
```


PROMESSAS (*PROMISES*)

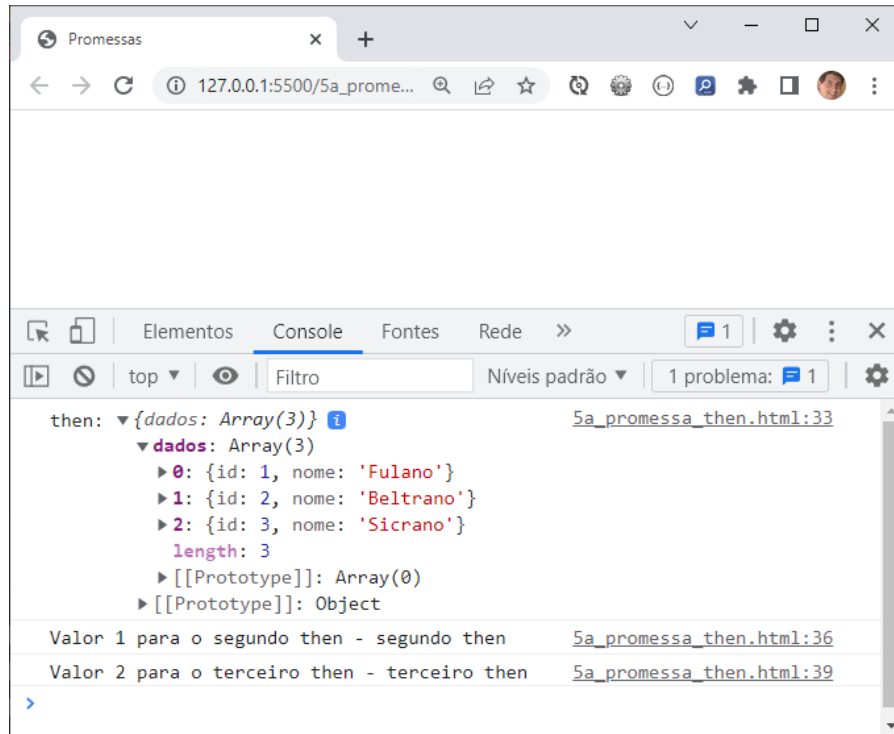
■ Funcionamento

- Um resultado bem sucedido, gerado pela chamada da função `resolve`, desencadeia a chamada do método `then`, ou seja, o método `then` executa uma função *callback* mediante a **resolução** da promessa
- Um resultado mal sucedido, gerado pela função `reject`, desencadeia a chamada do método `catch`, ou seja, o método `catch` executa uma função *callback* em caso de **rejeição** da promessa
- Qualquer valor passado para a função `resolve` será acessível como parâmetro da função `then`, podendo ser uma `String`, `Number`, `Booleano`, `Function`, `Object`, etc – o mesmo ocorre entre `reject` e `catch`



PROMESSAS (PROMISES)

■ Exemplo: then ()

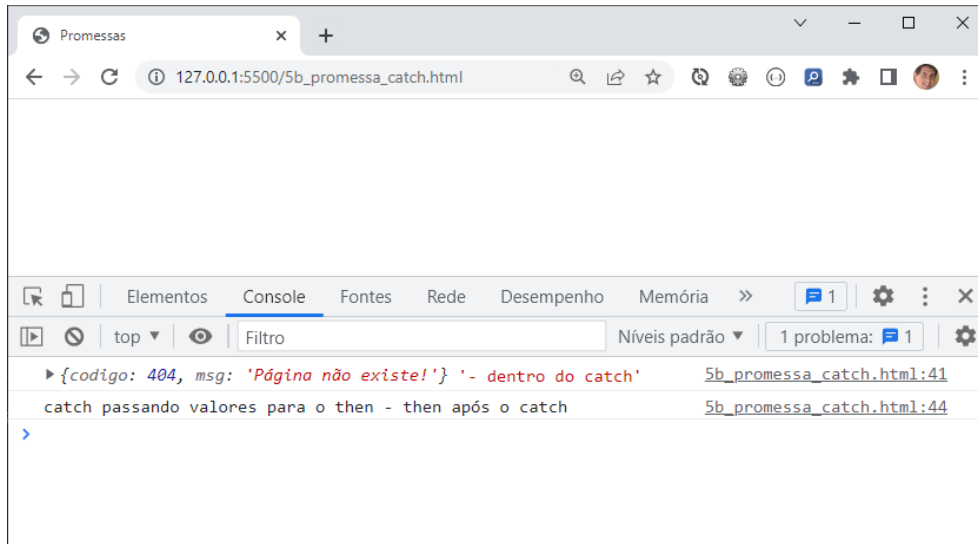


```
aula25 - 5a_promessa_then.html

1  <!DOCTYPE html>
2  <html lang="pt-BR">
3
4  <head>
5      <meta charset="UTF-8">
6      <meta http-equiv="X-UA-Compatible" content="IE=edge">
7      <meta name="viewport" content="width=device-width, initial-scale=1.0">
8      <title>Promessas</title>
9
10     <script>
11         let promessa = new Promise((resolve, reject) => {
12             // Simulação de uma requisição HTTP
13             setTimeout(() => {
14                 // Requisição foi realizada
15                 let resposta = {};
16                 const erro_requisicao = false;
17
18                 // Erro na requisição
19                 if (erro_requisicao) {
20                     resposta = { codigo: 404, msg: 'Página não existe!' };
21                     reject(resposta);
22                 }
23
24                 resposta = { dados: [
25                     { id: 1, nome: 'Fulano' },
26                     { id: 2, nome: 'Beltrano' },
27                     { id: 3, nome: 'Sicrano' }
28                 ]};
29
30                 resolve(resposta);
31             }, 4000);
32         }).then(dados => {
33             console.log('then:', dados);
34             return('Valor 1 para o segundo then')
35         }).then(dados => {
36             console.log(dados, '- segundo then');
37             return('Valor 2 para o terceiro then')
38         }).then(dados => {
39             console.log(dados, '- terceiro then');
40         });
41     </script>
42 </head>
43
44 <body>
45 </body>
46
47 </html>
```

PROMESSAS (PROMISES)

■ Exemplo: `catch()`



```
1 <!DOCTYPE html>
2 <html lang="pt-BR">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <title>Promessas</title>
9
10  <script>
11    let promessa = new Promise((resolve, reject) => {
12      // Simulação de uma requisição HTTP
13      setTimeout(() => {
14        // Requisição foi realizada
15        let resposta = {};
16        const erro_requisicao = true;
17
18        // Erro na requisição
19        if (erro_requisicao) {
20          resposta = { codigo: 404, msg: 'Página não existe!' };
21          reject(resposta);
22        }
23
24        resposta = { dados: [
25          { id: 1, nome: 'Fulano' },
26          { id: 2, nome: 'Beltrano' },
27          { id: 3, nome: 'Sicrano' }
28        ]};
29
30        resolve(resposta);
31      }, 4000);
32    }).then(dados => {
33      console.log('then:', dados);
34      return('Valor 1 para o segundo then')
35    }).then(dados => {
36      console.log(dados, '- segundo then');
37      return('Valor 2 para o terceiro then')
38    }).then(dados => {
39      console.log(dados, '- terceiro then');
40    }).catch(erro => {
41      console.log(erro, '- dentro do catch');
42      return('catch passando valores para o then')
43    }).then(dados => {
44      console.log(dados, '- then após o catch');
45    });
46  </script>
47 </head>
48
49 <body>
50 </body>
51
52 </html>
```

PROMESSAS (*PROMISES*)

■ Exemplo: Divisão

```
[Running] node "c:\html\aula25\6_promessa_divisao.js"  
Divisão normal de 10 por 2 = 5  
Divisão normal de 10 por 0 = Infinity  
Divisão normal de 0 por 0 = NaN  
Sucesso: 5
```

```
aula25 - 6_promessa_divisao.js  
  
1  let log = console.log;  
2  
3  function dividir(a, b) {  
4      return a / b;  
5  }  
6  
7  log('Divisão normal de 10 por 2 = ' + dividir(10, 2));  
8  log('Divisão normal de 10 por 0 = ' + dividir(10, 0));  
9  log('Divisão normal de 0 por 0 = ' + dividir(0, 0));  
10 log('');  
11  
12 //-----  
13  
14 function promessaDivisao(a, b) {  
15     return new Promise(function (resolve, reject) {  
16         if (b === 0) {  
17             reject(new Error('Não é possível dividir por 0 !'));  
18             return;  
19         }  
20  
21         resolve(a / b);  
22     });  
23 }  
24  
25 promessaDivisao(10, 2).then(function (resultado) {  
26     log(`Sucesso: ${resultado}`);  
27 }).catch(function (erro) {  
28     log('Erro na divisão');  
29     log(erro);  
30 });
```

PROMESSAS (*PROMISES*)

■ Exemplo: Divisão

[Running] node "c:\html\aula25\6_promessa_divisao.js"

Divisão normal de 10 por 2 = 5

Divisão normal de 10 por 0 = Infinity

Divisão normal de 0 por 0 = NaN

Erro na divisão

Error: Não é possível dividir por 0 !

```
at c:\html\aula25\6_promessa_divisao.js:17:20
at new Promise (<anonymous>)
at promessaDivisao (c:\html\aula25\6_promessa_divisao.js:15:12)
at Object.<anonymous> (c:\html\aula25\6_promessa_divisao.js:25:1)
at Module._compile (node:internal/modules/cjs/loader:1126:14)
at Object.Module._extensions..js (node:internal/modules/cjs/loader:1180:10)
at Module.load (node:internal/modules/cjs/loader:1004:32)
at Function.Module._load (node:internal/modules/cjs/loader:839:12)
at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
at node:internal/main/run_main_module:17:47
```

```
aula25 - 6_promessa_divisao.js

1  let log = console.log;
2
3  function dividir(a, b) {
4      return a / b;
5  }
6
7  log('Divisão normal de 10 por 2 = ' + dividir(10, 2));
8  log('Divisão normal de 10 por 0 = ' + dividir(10, 0));
9  log('Divisão normal de 0 por 0 = ' + dividir(0, 0));
10 log('');
11
12 //-----
13
14 function promessaDivisao(a, b) {
15     return new Promise(function (resolve, reject) {
16         if (b === 0) {
17             reject(new Error('Não é possível dividir por 0 !'));
18             return;
19         }
20
21         resolve(a / b);
22     });
23 }
24
25 promessaDivisao(10, 0).then(function (resultado) {
26     log(`Sucesso: ${resultado}`);
27 }).catch(function (erro) {
28     log('Erro na divisão');
29     log(erro);
30 });
```

PROMESSAS (*PROMISES*)

■ Exemplo

```
aula25 - 7a_limpar_quartojs

1  let promessaDeLimparQuarto = new Promise((resolve, reject) => {
2      /* Processo: Limpando o quarto
3          .
4          .
5          .      */
6      let estaLimpo = true;
7
8      if (estaLimpo) {
9          resolve('está limpo');
10     } else {
11         reject('não está limpo');
12     }
13 });
14
15 promessaDeLimparQuarto.then(function(doResolve) {
16     console.log('O quarto ' + doResolve)
17 }).catch(function(doReject) {
18     console.log('O quarto ' + doReject)
19 });
```

PROMESSAS (*PROMISES*)

- Dependências, execução sequencial e paralelismo de promessas com `all` e `race`

```
aula25 - 7b_limpar_casa.js

1 let limparQuarto = new Promise((resolve, reject) => {
2   resolve('Limpei o quarto! ');
3 });
4
5 let retirarLixo = function (mensagem) {
6   return new Promise((resolve, reject) => {
7     resolve(mensagem + '-> Retirei o lixo! ')
8   });
9 };
10
11 let ganharSorvete = function (mensagem) {
12   return new Promise((resolve, reject) => {
13     resolve(mensagem + '-> Ganhei sorvete! ')
14   });
15 };
16
17 limparQuarto.then(resultado => {
18   return retirarLixo(resultado);
19 }).then(resultado => {
20   return ganharSorvete(resultado);
21 }).then(resultado => {
22   console.log(resultado + '-> Fim!');
23 });
24
25 Promise.all([limparQuarto, retirarLixo(), ganharSorvete()])
26   .then(() => console.log('Todas as tarefas foram finalizadas!'));
27
28 Promise.any([limparQuarto, retirarLixo(''), ganharSorvete('')])
29   .then(tarefa => console.log(`Uma das tarefas - ${tarefa}- encerrada!\n`));
```

API FETCH

- Em 2015 foi lançada a Fetch API, que possui funcionalidades semelhante ao objeto XHR (XMLHttpRequest) mas de mais alto nível e baseado em promessas

```
fetch('send-ajax-data.php')  
  .then(data => console.log(data))  
  .catch (error => console.log('Error:' + error));
```


API FETCH

- Consumindo a API ([webservice](#)) de consulta de CEPs da [ViaCEP](#)



The screenshot shows a web browser window with the URL <https://viacep.com.br>. The page has a green header with the **VIA CEP** logo and the text "Consulte CEPs de todo o Brasil". The main content area is white and contains the following text:

Procurando um [webservice](#) gratuito e de alto desempenho para consultar Códigos de Endereçamento Postal (CEP) do Brasil? Utilize nosso serviço, melhore a qualidade de suas aplicações web e colabore para manter esta base de dados atualizada.

Acessando o webservice de CEP

Para acessar o webservice, um CEP no formato de **{8}** dígitos deve ser fornecido, por exemplo: "01001000". Após o CEP, deve ser fornecido o tipo de retorno desejado, que deve ser "json", "xml", "piped" ou "query".

Exemplo de pesquisa por CEP:
viacep.com.br/ws/01001000/json/

Validação do CEP

Quando consultado um CEP de formato inválido, por exemplo: "950100100" (9 dígitos), "95010A10" (alfanumérico), "95010 10" (espaço), o código de retorno da consulta será um **400** (Bad Request). Antes de acessar o webservice, valide o formato do CEP e certifique-se que o mesmo possua **{8}** dígitos. Exemplo de como validar o formato do CEP em javascript está disponível nos exemplos abaixo.

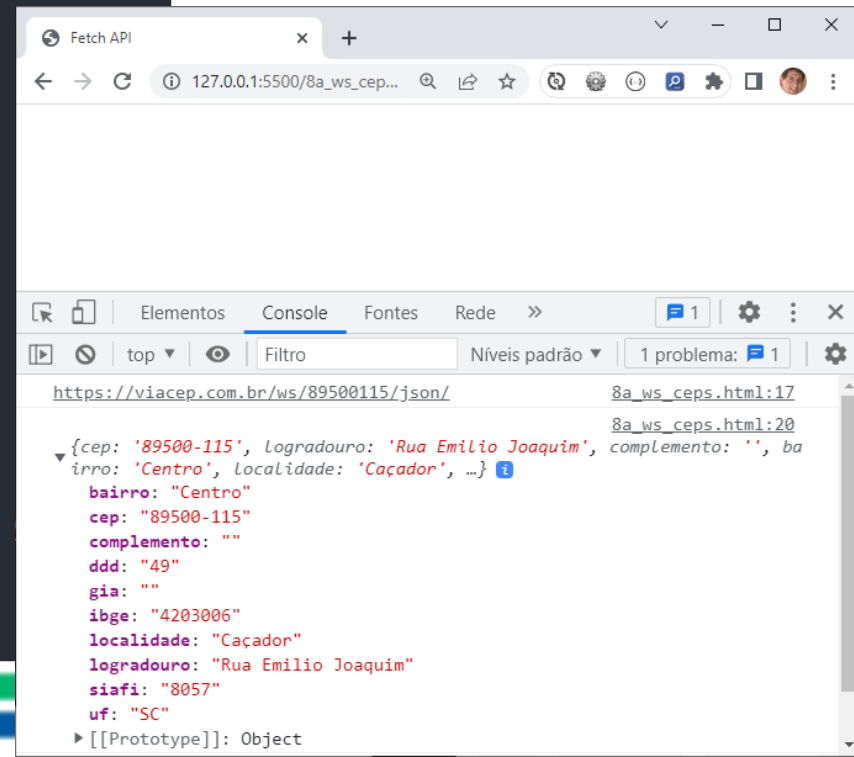
Quando consultado um CEP de formato válido, porém inexistente, por exemplo: "99999999", o retorno conterá um valor de "erro" igual a "true". Isso significa que o CEP consultado não foi encontrado na base de dados. Veja como manipular este "erro" em javascript nos exemplos abaixo.

API FETCH

■ Versão mais simples

```
aula25 - 8a_ws_ceps.html

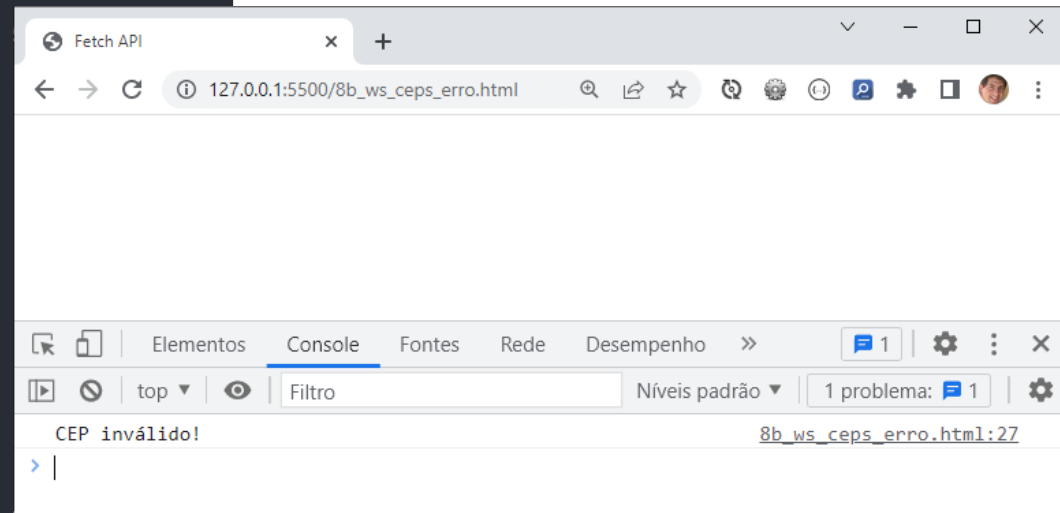
1 <!DOCTYPE html>
2 <html lang="pt-BR">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <title>Fetch API</title>
9 </head>
10
11 <body>
12   <ul id="usuarios"></ul>
13
14   <script>
15     const CEP = '89500115';
16     const URL = `https://viacep.com.br/ws/${CEP}/json`;
17
18     console.log(URL);
19
20     fetch(URL)
21       .then(response => response.json())
22       .then(dados => console.log(dados))
23   </script>
24 </body>
25
26 </html>
```



API FETCH

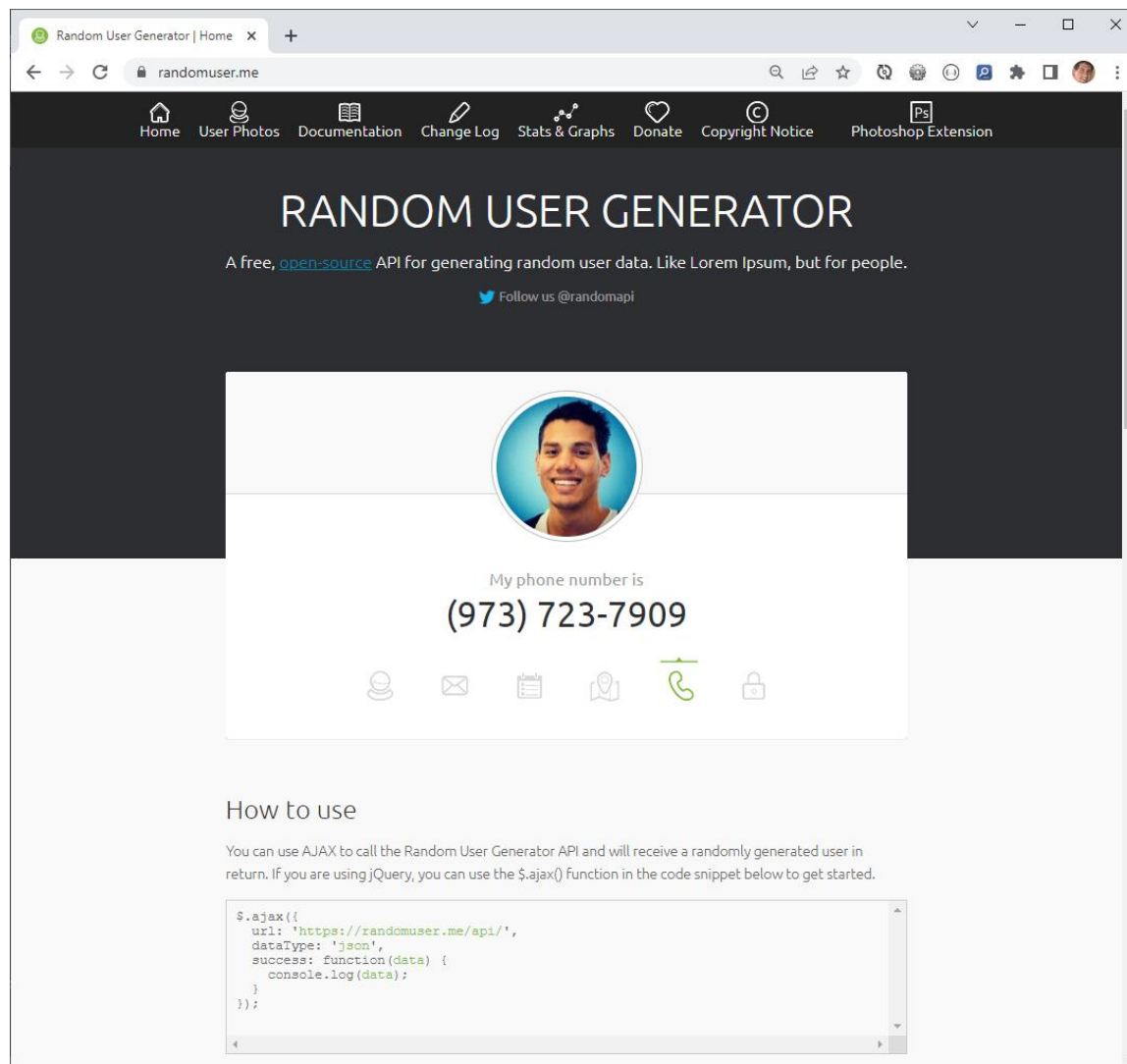
■ Versão mais completa, com opções e tratamento de erro

```
aula25 - 8b_ws_ceps_erro.html
1 <!DOCTYPE html>
2 <html lang="pt-BR">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <title>Fetch API</title>
9 </head>
10
11 <body>
12   <ul id="usuarios"></ul>
13
14   <script>
15     const CEP = '89500000';
16     const URL = `https://viacep.com.br/ws/${CEP}/json/`;
17
18     const opcoes = {
19       method: 'GET',
20       mode: 'cors',
21     };
22
23     fetch(URL, opcoes)
24       .then(response => response.json())
25       .then(dados => {
26         if (dados.erro) {
27           console.log('CEP inválido!');
28         }
29       })
30   </script>
31 </body>
32
33 </html>
```



API FETCH

- Acessando uma API que gera dados de usuários aleatórios

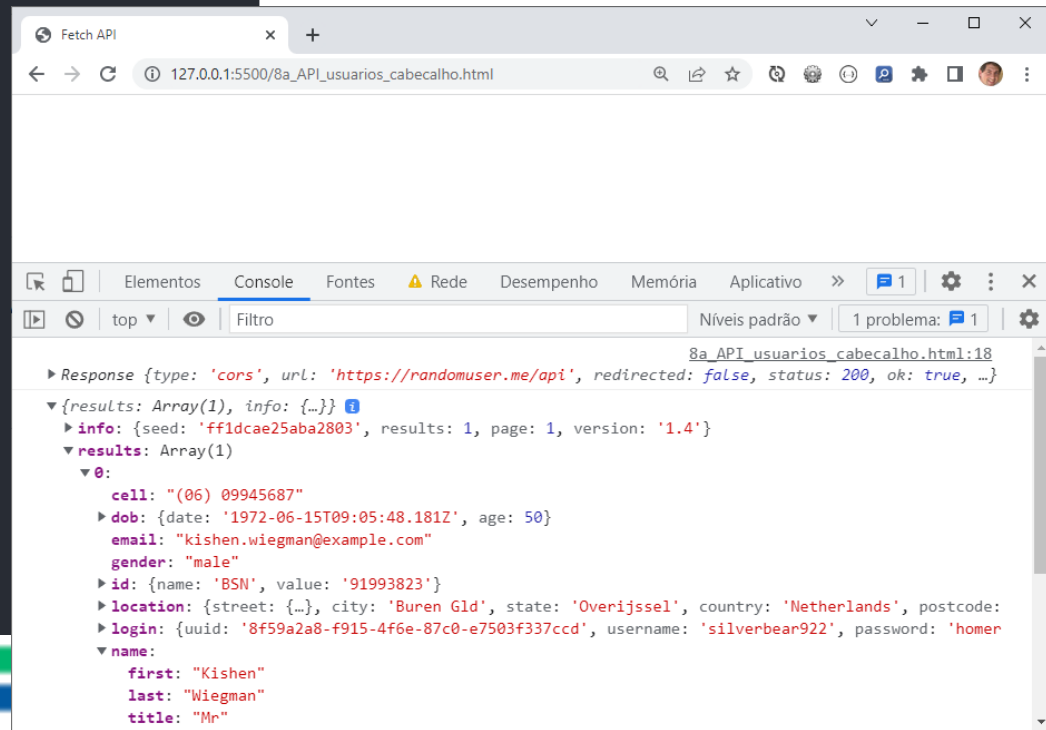


API FETCH

■ Mostrando a resposta com seu cabeçalho

aula25 - 8a_API_usuarios_cabecalho.html

```
1 <!DOCTYPE html>
2 <html lang="pt-BR">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <title>Fetch API</title>
9 </head>
10
11 <body>
12   <ul id="usuarios"></ul>
13
14   <script>
15     const ul = document.getElementById('usuarios');
16
17     fetch('https://randomuser.me/api')
18       // Cabeçalho
19       .then(resposta => {
20         console.log(resposta);
21         return resposta;
22       })
23       // Converte para JSON
24       .then(resposta => resposta.json())
25       .then(console.log)
26   </script>
27 </body>
28
29 </html>
```

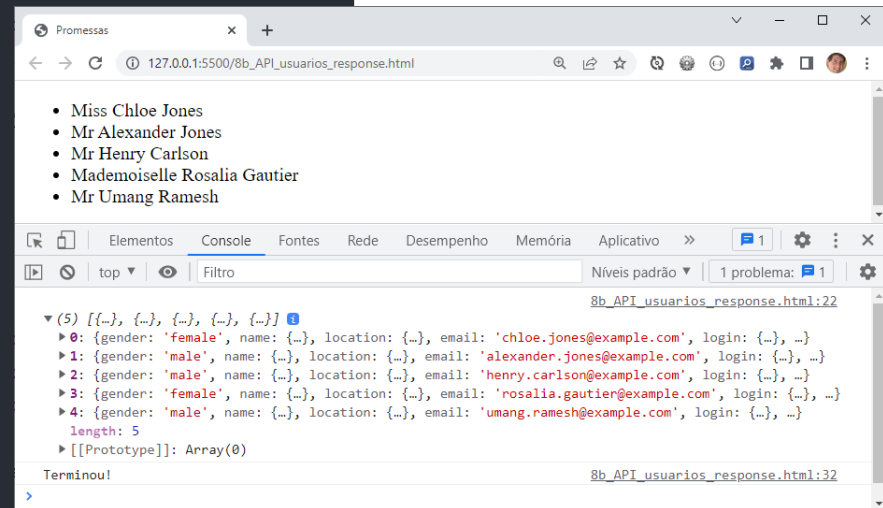


API FETCH

■ Tratando os dados

```
aula25 - 8b_API_usuarios_response.html

1 <!DOCTYPE html>
2 <html lang="pt-BR">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <title>Promessas</title>
9 </head>
10
11 <body>
12   <ul id="usuarios"></ul>
13
14   <script>
15     const ul = document.getElementById('usuarios');
16
17     fetch('https://randomuser.me/api?results=5')
18       .then(resposta => resposta.json())
19       .then(resposta => {
20         // throw new Error('teste');
21         usuarios = resposta.results;
22         console.log(usuarios);
23
24         for (usuario of usuarios) {
25           let li = document.createElement('li');
26           li.innerHTML = `${usuario.name.title} ${usuario.name.first} ${usuario.name.last}`;
27           ul.appendChild(li);
28         }
29
30       })
31       .catch(erro => console.log('Oops!' + erro))
32       .finally(() => console.log('Terminou!'));
33   </script>
34 </body>
35
36 </html>
```



ASYNC / AWAIT

- Os métodos async e await foram implementados no ES8 (ES2017) e constituem uma sintaxe que simplifica a programação assíncrona, facilitando o fluxo de escrita e leitura do código; assim é possível escrever código que funciona de forma assíncrona, porém lido e estruturado de forma síncrona
- O async/await também trabalha com o código assíncrono baseado em promessas, porém esconde as promessas para que a leitura e a escrita seja mais fluídas
- Definindo uma função como async, pode-se utilizar a palavra-chave await antes de qualquer expressão que retorne uma promessa; dessa forma, a execução da função externa (a função async) será pausada até que a promessa seja resolvida – ou seja, quando se usa await, o JavaScript vai aguardar até que a promessa finalize
- A palavra-chave await recebe uma promessa e a transforma em um valor de retorno (ou lança uma exceção em caso de erro)
 - Se a promessa for finalizada com sucesso (*fulfilled*), o valor obtido é retornado
 - Se a promessa for rejeitada (*rejected*), é retornado o erro lançado pela exceção

ASYNC / AWAIT

■ Leitura de arquivos: Versão com promessas (*promises*)

```
1  const fs = require('fs')
2
3  const leArquivo = arquivo => new Promise((resolve, reject) => {
4    fs.readFile(arquivo, (erro, conteudo) => {
5      if(erro) {
6        reject(erro)
7      } else {
8        resolve(conteudo)
9      }
10   })
11 })
12
13 // Encadeamento
14 const promessa = leArquivo('./in1.txt')
15 // console.log(promessa)
16 .then(conteudo => {
17   console.log(String(conteudo))
18   return leArquivo('./in2.txt')
19 })
20 .then(conteudo => {
21   console.log(String(conteudo))
22 })
23 .finally(() => {
24   console.log('Terminou!')
25 })
26
27 setTimeout(() => console.log(promessa), 1000)
```



ASYNC / AWAIT

■ Leitura de arquivos: Versão com *async/await*

```
1  const fs = require('fs')
2
3  const leArquivo = arquivo => new Promise((resolve, reject) => {
4    fs.readFile(arquivo, (erro, conteudo) => {
5      if(erro) {
6        reject(erro)
7      } else {
8        resolve(conteudo)
9      }
10   })
11 })
12
13 init = async (valor) => {
14   try {
15     let conteudo1 = await leArquivo('./in1.txt')
16     let conteudo2 = await leArquivo('./in2.txt')
17     return String(conteudo1) + '\n' + String(conteudo2)
18   } catch(erro) {
19     console.log(erro)
20   }
21 };
22
23 init().then( conteudo => console.log(conteudo))
24 console.log('init', init())
```

ASYNC / AWAIT

■ Async/Await podem ser usadas para escrever Promises de forma mais concisa

```
1 const random = () => {
2   return Promise.resolve(Math.trunc(Math.random()*100))
3 }
4
5 const sumRandomAsyncNums = () => {
6   let first, second, third
7
8   return random()
9     .then(v => {
10      first = v
11      return random()
12    })
13    .then(v => {
14      second = v
15      return random()
16    })
17    .then(v => {
18      third = v
19      return first + second + third
20    })
21    .then(v => {
22      console.log(`${first}+${second}+${third}=${v}`)
23    })
24 }
25
26 sumRandomAsyncNums()
```

```
1 const random = () => {
2   return Promise.resolve(Math.trunc(Math.random()*100))
3 }
4
5 const sumRandomAsyncNums = async() => {
6   const first = await random()
7   const second = await random()
8   const third = await random()
9
10  const v = first + second + third
11
12  console.log(`${first}+${second}+${third}=${v}`)
13 }
14
15 sumRandomAsyncNums()
```

Apoiadores:



ASYNC / AWAIT

- navigator: Informações sobre o navegador
- Recurso de geolocalização

```
1 <!DOCTYPE html>
2 <html lang="pt-BR">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <title>BOM - Navigator - Geolocalização</title>
9
10  <script>
11    let local;
12
13    const requisitaPosicao = () => {
14      const opcoes = {
15        enableHighAccuracy: true,
16        timeout: 5000,
17        maximumAge: 0
18      };
19
20      return new Promise(function (resolve, reject) {
21        navigator.geolocation.getCurrentPosition(
22          pos => {
23            resolve(local = `Latitude : ${pos.coords.latitude} x Longitude: ${pos.coords.longitude}`);
24          },
25          err => { reject(err) },
26          opcoes
27        );
28      });
29    }
30
31    async function infosNavegador() {
32      await requisitaPosicao();
33
34      console.log(local);
35
36      infos = `appName: ${navigator.appName} <br>`;
37      infos += `appVersion : ${navigator.appVersion}<br>`;
38      infos += `appName : ${navigator.appCodeName}<br>`;
39      infos += `product : ${navigator.product}<br>`;
40      infos += `userAgent : ${navigator.userAgent}<br>`;
41      infos += `platform : ${navigator.platform}<br>`;
42      infos += `language : ${navigator.language}<br>`;
43      infos += `online? ${navigator.onLine ? 'Sim' : 'Não'}<br>`;
44      infos += `Java habilitado? ${navigator.javaEnabled() ? 'Sim' : 'Não'}<br>`;
45      infos += `Conexão: ${navigator.connection}<br>`;
46      infos += `Memória: ${navigator.deviceMemory} GB <br>`;
47      infos += `Geolocalização: ${local}<br>`;
48
49      document.getElementById('infos').innerHTML = infos;
50    }
51  </script>
52 </head>
53
54 <body style="text-align: center;">
55   <button onclick="infosNavegador()">Informações Navegador</button>
56   <div id="infos"></div>
57 </body>
58
59 </html>
```