

JAVASCRIPT ASSÍNCRONO

JavaScript Assíncrono

Síncrono vs Assíncrono

- **Síncrono**

Espera a tarefa acabar para continuar com a próxima.

- **Assíncrono**

Move para a próxima tarefa antes da anterior terminar. O trabalho será executado no 'fundo' e quando terminado, será colocado na fila (Task Queue).

- **Exemplos**

`setTimeout`, `Ajax`, `Promises`, `Fetch`, `Async`.

Vantagens

- **Carregamento no Fundo**

Não trava o script. É possível utilizar o site enquanto o processamento é realizado no fundo.

- **Função ao término**

Podemos ficar de olho no carregamento e executar uma função assim que ele terminar.

- **Requisições ao Servidor**

Não precisamos recarregar a página por completo à cada requisição feita ao serviro.

JAVASCRIPT ASSÍNCRONO

Promises

new Promise()

`Promise` é uma função construtora de promessas. Existem dois resultados possíveis de uma promessa, ela pode ser resolvida, com a execução do primeiro argumento, ou rejeitada se o segundo argumento for ativado.

```
const promessa = new Promise(function(resolve, reject) {  
  let condicao = true;  
  if(condicao) {  
    resolve();  
  } else {  
    reject();  
  }  
});  
  
console.log(promessa); // Promise {<resolved>: undefined}
```

resolve()

Podemos passar um argumento na função `resolve()`, este será enviado junto com a resolução da Promise.

```
const promessa = new Promise(function(resolve, reject) {  
  let condicao = true;  
  if(condicao) {  
    resolve('Estou pronto!');  
  } else {  
    reject();  
  }  
});  
  
console.log(promessa); // Promise {<resolved>: "Estou pronto!"}
```

reject()

Quando a condição de resolução da promise não é atingida, ativamos a função `reject` para rejeitar a mesma. Podemos indicar no console um erro, informando que a promise foi rejeitada.

```
const promessa = new Promise(function(resolve, reject) {  
  let condicao = false;  
  if(condicao) {  
    resolve('Estou pronto!');  
  } else {  
    reject(Error('Um erro ocorreu.'));  
  }  
});  
  
console.log(promessa); // Promise {<rejected>: Error:...}
```

then()

O poder das Promises está no método `then()` do seu protótipo. O Callback deste método só será ativado quando a promise for resolvida. O argumento do callback será o valor passado na função `resolve`.

```
const promessa = new Promise(function(resolve, reject) {  
  let condicao = true;  
  if(condicao) {  
    resolve('Estou pronto!');  
  } else {  
    reject(Error('Um erro ocorreu.'));  
  }  
});  
  
promessa.then(function(resolucao) {  
  console.log(resolucao); // 'Estou pronto!'  
});
```

Assíncrono

As promises não fazem sentido quando o código executado dentro da mesma é apenas código síncrono. O poder está na execução de código assíncrono que executará o `resolve()` ao final dele.

```
const promessa = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Resolvida');
  }, 1000);
});

promessa.then(resolucao => {
  console.log(resolucao); // 'Resolvida' após 1s
});
```

then().then()

O método `then()` retorna outra Promise. Podemos colocar `then()` após `then()` e fazer um encadeamento de promessas. O valor do primeiro argumento de cada `then`, será o valor do retorno anterior.

```
const promessa = new Promise((resolve, reject) => {
  resolve('Etapa 1');
});

promessa.then(resolucao => {
  console.log(resolucao); // 'Etapa 1';
  return 'Etapa 2';
}).then(resolucao => {
  console.log(resolucao) // 'Etapa 2';
  return 'Etapa 3';
}).then(r => r + 4)
  .then(r => {
    console.log(r); // Etapa 34
  });
});
```

catch()

O método `catch()`, do protótipo de Promises, adiciona um callback a promise que será ativado caso a mesma seja rejeitada.

```
const promessa = new Promise((resolve, reject) => {
  let condicao = false;
  if(condicao) {
    resolve('Estou pronto!');
  } else {
    reject(Error('Um erro ocorreu.'));
  }
});

promessa.then(resolucao => {
  console.log(resolucao);
}).catch(reject => {
  console.log(reject);
});
```

then(resolve, reject)

Podemos passar a função que será ativada caso a promise seja rejeitada, direto no método then, como segundo argumento.

```
const promessa = new Promise((resolve, reject) => {
  let condicao = false;
  if(condicao) {
    resolve('Estou pronto!');
  } else {
    reject(Error('Um erro ocorreu.'));
  }
});

promessa.then(resolucao => {
  console.log(resolucao);
}, reject => {
  console.log(reject);
});
```

finally()

`finally()` executará a função anônima assim que a promessa acabar. A diferença do `finally` é que ele será executado independente do resultado, se for resolvida ou rejeitada.

```
const promessa = new Promise((resolve, reject) => {
  let condicao = false;
  if(condicao) {
    resolve('Estou pronto!');
  } else {
    reject(Error('Um erro ocorreu.'));
  }
});

promessa.then(resolucao => {
  console.log(resolucao);
}, reject => {
  console.log(reject);
}).finally(() => {
  console.log('Acabou'); // 'Acabou'
});
```

Promise.all()

Retornará uma nova promise assim que todas as promises dentro dela forem resolvidas ou pelo menos uma rejeitada. A resposta é um array com as respostas de cada promise.

```
const login = new Promise(resolve => {
  setTimeout(() => {
    resolve('Login Efetuado');
  }, 1000);
});

const dados = new Promise(resolve => {
  setTimeout(() => {
    resolve('Dados Carregados');
  }, 1500);
});

const tudoCarregado = Promise.all([login, dados]);

tudoCarregado.then(respostas => {
  console.log(respostas); // Array com ambas respostas
});
```

Promise.race()

Retornará uma nova promise assim que a primeira promise for resolvida ou rejeitada. Essa nova promise terá a resposta da primeira resolvida.

```
const login = new Promise(resolve => {
  setTimeout(() => {
    resolve('Login Efetuado');
  }, 1000);
});

const dados = new Promise(resolve => {
  setTimeout(() => {
    resolve('Dados Carregados');
  }, 1500);
});

const carregouPrimeiro = Promise.race([login, dados]);

carregouPrimeiro.then(resposta => {
  console.log(resposta); // Login Efetuado
});
```

JAVASCRIPT ASSÍNCRONO

Fetch

Fetch API

Permite fazermos requisições HTTP através do método `fetch()`. Este método retorna a resolução de uma Promise. Podemos então utilizar o `then` para interagirmos com a resposta, que é um objeto do tipo Response.

```
fetch('./arquivo.txt').then(function(response) {  
  console.log(response); // Response HTTP (Objeto)  
});
```

Response

O objeto Response, possui um corpo com o conteúdo da resposta. Esse corpo pode ser transformado utilizando métodos do protótipo do objeto Response. Estes retornam outras promises.

```
fetch('./arquivo.txt').then(function(response) {  
  return response.text();  
}).then(function(corpo) {  
  console.log(corpo);  
});
```

Servidor Local

O fetch faz uma requisição HTTP/HTTPS. Se você iniciar um site local diretamente pelo index.html, sem um servidor local, o fetch não irá funcionar.

```
fetch('c:/files/arquivo.txt')
  .then((response) => {
    return response.text();
})
  .then((corpo) => {
    console.log(corpo);
}); // erro
```

Se dermos um espaço após o objeto ou pularmos linha, o método continua funcionando.

.json()

Um tipo de formato de dados muito utilizado com JavaScript é o **JSON** (JavaScript Object Notation), pelo fato dele possuir basicamente a mesma sintaxe que a de um objeto js. **.json()** transforma um corpo em json em um objeto JavaScript.

```
fetch('https://viacep.com.br/ws/01001000/json/')
  .then(response => response.json())
  .then(cep => {
    console.log(cep.bairro, cep.logradouro);
  });
}
```

.text()

Podemos utilizar o `.text()` para diferentes formatos como txt, json, html, css, js e mais.

```
const styleElement = document.createElement('style');

fetch('./style.css')
.then(response => response.text())
.then(style => {
  styleElement.innerHTML = style;
  document.body.appendChild(styleElement);
});
```

HTML e .text()

Podemos pegar um arquivo inteiro em HTML, transformar o corpo em texto e inserir em uma div com o innerHTML. A partir dai podemos manipular esses dados como um DOM qualquer.

```
const div = document.createElement('div');

fetch('./sobre.html')
  .then(response => response.text())
  .then(htmlBody => {
    div.innerHTML = htmlBody;
    const titulo = div.querySelector('h1');
    document.querySelector('h1').innerText = titulo.innerText;
  });
}
```

.blob()

Um blob é um tipo de objeto utilizado para representação de dados de um arquivo. O blob pode ser utilizado para transformarmos requisições de imagens por exemplo. O blob gera um URL único.

```
const div = document.createElement('div');

fetch('./imagem.png')
.then(response => response.blob())
.then(imgBlob => {
  const blobUrl = URL.createObjectURL(imgBlob);
  console.log(blobUrl);
});
```

.clone()

Ao utilizarmos os métodos acima, text, json e blob, a resposta é modificada. Por isso existe o método clone, caso você necessite transformar uma resposta em diferentes valores.

```
const div = document.createElement('div');

fetch('https://viacep.com.br/ws/01001000/json/')
  .then(response => {
    const cloneResponse = response.clone();
    response.json().then(json => {
      console.log(json)
    });
    cloneResponse.text().then(text => {
      console.log(text)
    });
  });
}
```

.headers

É uma propriedade que possui os cabeçalhos da requisição. É um tipo de dado iterável então podemos utilizar o `forEach` para vermos cada um deles.

```
const div = document.createElement('div');

fetch('https://viacep.com.br/ws/01001000/json/')
  .then(response => {
    response.headers.forEach(console.log);
  });
}
```

.status e .ok

Retorna o status da requisição. Se foi 404, 200, 202 e mais. ok retorna um valor booleano sendo true para uma requisição de sucesso e false para uma sem sucesso.

```
const div = document.createElement('div');

fetch('https://viacep.com.br/ws/01001000/json/')
  .then(response => {
    console.log(response.status, response.ok);
    if(response.status === 404) {
      console.log('Página não encontrada')
    }
  });
}
```

.url e .type

`.url` retorna o url da requisição. `.type` retorna o tipo da resposta.

```
const div = document.createElement('div');

fetch('https://viacep.com.br/ws/01001000/json/')
  .then(response => {
    console.log(response.type, response.url);
  });

//types
// basic: feito na mesma origem
// cors: feito em url body pode estar disponível
// error: erro de conexão
// opaque: no-cors, não permite acesso de outros sites
```

Exercícios

```
// Utilizando a API https://viacep.com.br/ws/\${CEP}/json/
// crie um formulário onde o usuário pode digitar o cep
// e o endereço completo é retornado ao clicar em buscar
```

```
// Utilizando a API https://blockchain.info/ticker
// retorne no DOM o valor de compra da bitcoin and reais.
// atualize este valor a cada 30s
```

```
// Utilizando a API https://api.chucknorris.io/jokes/random
// retorne uma piada randomica do chucknorris, toda vez que
// clicar em próxima
```

JAVASCRIPT ASSÍNCRONO

JSON

JSON

JavaScript Object Notation (JSON) é um formato de organização de dados, compostos por um conjunto de chave e valor. As aspas duplas são obrigatórias, tanto na chave quanto no valor quando este for uma string.

```
{  
  "id": 1,  
  "nome": "Andre",  
  "email": "andre@origamid.com"  
}
```

Valores

Os valores podem ser números, strings, boolean, arrays, objetos e null.

```
{  
  "id": 1,  
  "faculdade": true,  
  "pertences": [  
    "lapis",  
    "caneta",  
    "caderno"  
  ],  
  "endereco": {  
    "cidade": "Rio de Janeiro",  
    "pais": "Brasil"  
  },  
  "casado": null  
}
```

Arrays e Objetos

É comum possuirmos array's com objetos em cada valor da array.
Cuidado para não colocar vírgula no último item do objeto ou array.

```
[  
  {  
    "id": 1,  
    "aula": "JavaScript",  
    "tempo": "25min"  
  },  
  {  
    "id": 2,  
    "aula": "HTML",  
    "tempo": "15min"  
  },  
  {  
    "id": 3,  
    "aula": "CSS",  
    "tempo": "10min"  
  }]  
]
```

ORIGAMID

JSON.parse() e JSON.stringify()

`JSON.parse()` irá transformar um texto JSON em um objeto JavaScript. `JSON.stringify()` irá transformar um objeto JavaScript em uma string no formato JSON.

```
const textoJSON = '{"id": 1, "titulo": "JavaScript", "tempo":  
"25min"}';  
const textoOBJ = JSON.parse(textoJSON);  
  
const enderecoOBJ = {  
    cidade: "Rio de Janeiro",  
    rua: "Ali Perto",  
    pais: "Brasil",  
    numero: 50,  
}  
const enderecoJSON = JSON.stringify(enderecoOBJ);
```

Exemplo Real

Podemos guardar por exemplo no localStorage, uma string como valor de uma propriedade. E retornar essa string como um objeto.

```
const configuracoes = {  
    player: "Google API",  
    tempo: 25.5,  
    aula: "2-1 JavaScript",  
    vitalicio: true,  
}  
  
localStorage.configuracoes = JSON.stringify(configuracoes);  
const pegarConfiguracoes =  
JSON.parse(localStorage.configuracoes);
```

JAVASCRIPT ASSÍNCRONO

API e HTTP

API

- Application

Um servidor, aplicativo, objeto JavaScript ou qualquer outra coisa que você interaja através de comandos. Ao digitar uma URL, estamos utilizando a API do browser para se comunicar com a API do servidor.

- Programming

Programação, isso significa que um comando irá encadear uma cadeia de eventos pré-definidos. O resultado esperado é geralmente o mesmo.

- Interface

A interface são os comandos criados para permitir a interação com a aplicação. Ex: `'VIOLAO'.toLowerCase()` é um método que faz parte da interface do objeto String. A interação com a interface retorna um efeito / resposta.

Exemplos de API's

- GitHub

<https://api.github.com/users/origamid>

<https://api.github.com/users/origamid/followers>

- Array / Element

`[].map();`

`[].filter();`

`Element.classList;`

`Element.attributes;`

- Tempo

<https://www.metaweather.com/api/location/455825/>

<https://github.com/toddmotto/public-apis>

HTTP

Hypertext Transfer Protocol é o protocolo utilizado para enviarmos/recebermos arquivos e dados na Web.

```
fetch('https://pokeapi.co/api/v2/pokemon/')
  .then(r => r.json())
  .then(pokemon => {
    console.log(pokemon);
  });
}
```

url e method

Uma requisição HTTP é feita através de uma URL. O método padrão é o GET, mas existem outros como POST, UPDATE, DELETE, HEADER.

```
const url = 'https://jsonplaceholder.typicode.com/posts/';
const options = {
  method: 'POST',
  headers: {
    "Content-Type": "application/json; charset=utf-8",
  },
  body: '{"aula": "JavaScript"}'
}

fetch(url, options)
  .then(response => response.json())
  .then(json => {
    console.log(json);
  });
}
```

method

- GET

Puxa informação, utilizado para pegar posts, usuários e etc.

- POST

Utilizado para criar posts, usuários e etc.

- PUT

Geralmente utilizado para atualizar informações.

- DELETE

Deleta uma informação.

- HEAD

Puxa apenas os headers.

ORIGAMID

GET

GET irá puxar as informações da URL. Não é necessário informar que o método é GET, pois este é o padrão.

```
const url = 'https://jsonplaceholder.typicode.com/posts/';
fetch(url, {
  method: 'GET'
})
.then(r => r.json())
.then(r => console.log(r))
```

POST

POST irá criar uma nova postagem, utilizando o tipo de conteúdo especificado no headers e utilizando o conteúdo do body.

```
const url = 'https://jsonplaceholder.typicode.com/posts/';

fetch(url, {
  method: 'POST',
  headers: {
    "Content-Type": "application/json; charset=utf-8",
  },
  body: '{"titulo": "JavaScript"}'
})
.then(r => r.json())
.then(r => console.log(r))
```

PUT

PUT irá atualizar o conteúdo do URL com o que for informado no conteúdo do body.

```
const url = 'https://jsonplaceholder.typicode.com/posts/1/';

fetch(url, {
  method: 'PUT',
  headers: {
    "Content-Type": "application/json; charset=utf-8",
  },
  body: '{"titulo": "JavaScript"}'
})
.then(r => r.json())
.then(r => console.log(r))
```

HEAD

HEAD puxa apenas os headers. É uma requisição mais leve pois não puxa o body.

```
const url = 'https://jsonplaceholder.typicode.com/posts/1/';

fetch(url, {
  method: 'HEAD',
})
.then(response => {
  response.headers.forEach(console.log);
  console.log(response.headers.get('Content-Type'));
});
```

Headers

- **Cache-Control**

Tempo que o arquivo deve ficar em cache em segundos. Ex: public, max-age=3600

- **Content-Type**

Tipo de conteúdo. Ex: text/html; charset=utf-8. Indicar o tipo de arquivo principalmente em métodos POST e PUT.

- **Lista de Headers**

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>

CORS

Cross-Origin Resource Sharing, gerencia como deve ser o compartilhamento de recursos entre diferentes origens.

É definido no servidor se é permitido ou não o acesso dos recursos através de scripts por outros sites. Utilizando o Access-Control-Allow-Origin.

Se o servidor não permitir o acesso, este será bloqueado. É possível passar por cima do bloqueio utilizando um proxy.

CORS é um acordo entre browser / servidor ou servidor / servidor. Ele serve para dar certa proteção ao browser, mas não é inviolável.

```
const url = 'https://cors-
anywhere.herokuapp.com/https://www.google.com/';
const div = document.createElement('div');

fetch(url)
.then(r => r.text())
.then(r => {
  div.innerHTML = r;
  console.log(div);
})
```

JAVASCRIPT ASSÍNCRONO

ASYNC e AWAIT

async / await

A palavra chave `async` indica que a função possui partes assíncronas e que você pretende esperar a resolução da mesma antes de continuar. O `await` irá indicar a promise que devemos esperar. Faz parte do ES8.

```
async function puxarDados() {  
  const dadosResponse = await fetch('./dados.json');  
  const dadosJSON = await dadosResponse.json();  
  
  document.body.innerText = dadosJSON.titulo;  
}  
  
puxarDados();
```

then / async

A diferença é uma sintaxe mais limpa.

```
function iniciarFetch() {  
    fetch('./dados.json')  
        .then(dadosResponse => dadosResponse.json())  
        .then(dadosJSON => {  
            document.body.innerText = dadosJSON.titulo;  
        })  
}  
  
iniciarFetch();  
  
async function iniciarAsync() {  
    const dadosResponse = await fetch('./dados.json');  
    const dadosJSON = await dadosResponse.json();  
    document.body.innerText = dadosJSON.titulo;  
}  
  
iniciarAsync();
```

Try / Catch

Para lidarmos com erros nas promises, podemos utilizar o `try` e o `catch` na função.

```
async function puxarDados() {  
  try {  
    const dadosResponse = await fetch('./dados.json');  
    const dadosJSON = await dadosResponse.json();  
    document.body.innerText = dadosJSON.titulo;  
  }  
  catch(error) {  
    console.log(error);  
  }  
}  
puxarDados();
```

Iniciar Fetch ao Mesmo Tempo

Não precisamos esperar um fetch para começarmos outro. Porém precisamos esperar a resposta resolvida do fetch para transformarmos a response em `json`.

```
async function iniciarAsync() {  
  const dadosResponse = fetch('./dados.json');  
  const clientesResponse = fetch('./clientes.json');  
  
  // ele espera o que está dentro da expressão () ocorrer  
  // primeiro  
  const dadosJSON = await (await dadosResponse).json();  
  const clientesJSON = await (await clientesResponse).json();  
}  
iniciarAsync();
```

Promise

O resultado da expressão à frente de await tem que ser uma promise. E o retorno do await será sempre o resultado desta promise.

```
async function asyncSemPromise() {  
    // Console não irá esperar.  
    await setTimeout(() => console.log('Depois de 1s'), 1000);  
    console.log(' acabou');  
}  
asyncSemPromise();  
  
async function iniciarAsync() {  
    await new Promise(resolve => {  
        setTimeout(() => resolve(), 1000)  
    });  
    console.log('Depois de 1s');  
}  
iniciarAsync();
```

JAVASCRIPT ASSÍNCRONO

History API

History

É possível acessarmos o histórico de acesso do browser em uma sessão específica através do `window.history`. O objeto history possui diferentes métodos e propriedades.

```
window.history;  
window.history.back(); // vai para a anterior  
window.history.forward(); // vai para a próxima
```

pushState()

A parte interessante de manipularmos o history é que podemos modificar o histórico e adicionar um novo item.

```
window.history.pushState(obj, title, url).
```

```
// Em obj podemos enviar um objeto com dados  
// mas o seu uso é restrito por isso geralmente utilizamos  
// null. O title ainda é ignorado por alguns browsers, também  
// utilizamos null nele. O url que é parte importante.
```

```
window.history.pushState(null, null, 'sobre.html');
```

popstate

O evento `popstate` pode ser adicionado ao objeto `window`. Assim podemos executar uma função toda vez que o usuário clicar no botão de voltar ou próximo.

```
window.addEventListener('popstate', () => {  
  fetchPage(window.location.pathname);  
});
```

COPiar

Fetch e History

Ao puxarmos dados via fetch api, o url da página continua o mesmo. Ao combinar fetch com a history api conseguimos simular uma navegação real entre páginas, sem a necessidade de recarregamento da mesma.

```
async function fetchPage(url) {  
  const pageReponse = await fetch(url);  
  const pageText = await pageReponse.text();  
  window.history.pushState(null, null, url);  
}
```