* Para fazer requisições HTTP, podemos utilizar o **Fetch API**: biblioteca nativa já integrada no JavaScript do navegador
* Existem outras formas de se fazer uma requisição: axios, jQuery, etc
* Para fazermos uma requisição no JavaScript, usamos o método **fetch(url ou definição de uma request)**
* O fetch (é, por padrão, uma requisição get) retorna uma **promise**: elemento importante para lidarmos com assincronismo no JavaScript
* Quando fazemos uma requisição, ela demora um tempo para devolver uma resposta
* **Processamento assíncrono**: envio da requisição – passagem pelo navegador – chegada no servidor – processamento – retorno (processo que é executado e que não temos a resposta de imediato; em algum momento terei a resposta)
* A **promise** é a promessa de um resultado desse processamento assíncrono
* Para processarmos o **sucesso de uma promise**, temos o método **then()**

Exemplo:

fetch(url).**then(function (response) {**

console.log(response);

**})**

* Para verificar as requisições feitas em tempo real, entramos em inspecionar a página, menu Network – Menu Fetch/XHR – clicamos na requisição abaixo do name
* Sempre que utilizamos assincronismo no JavaScript, resolvemos através de **promises**
* Para processarmos o **fracasso de uma promise**, temos o método **catch()**

Exemplo:

fetch(url).**catch(function (error) {**

console.**error**(error);

**})**

* Para processarmos alguma operação toda vez que a requisição terminar, **independente de sucesso ou fracasso**, temos o método **finally()**

Exemplo:

fetch(url).**finally(function () {**

console.**error**(‘Requisição concluída!’);

**})**

* Podemos usar também a estrutura **try catch**

Exemplo:

try {

faz algo

} catch(error) {

para erros

} finally {

independente do sucesso ou do erro do erro ou do sucesso, é executado quando finaliza o processo

}

* Por padrão, o fetch() sempre vai retornar uma **ReadableStream** e podemos trabalhr com essa Stream
* Temos um **JSON** dentro dessa ReadableStrem e precisamos extraí-lo para conseguir acessar e manipular como um objeto JavaScript
* Como fazer isso:

fetch(url).

then(function (response) {

**response.json().then(function (responseBody) {**

**console.log(responseBody);**

**}**

})

* Podemos **encadear** vários métodos **.then()** onde o retorno de um método fica dentro do outro
* Podemos usar as **arrow functions**: sintaxe reduzida da definição de uma função

Exemplo:

.then((response) **=>** {

return response.json();

})

* Essa função é utilizada quando usamos um **call-back** e não precisamos dar contexto a ela
* Se declararmos uma function que não tiver mais de uma linha, não precisamos fragmentá-la para a linha de baixo, como no exemplo visto acima, dando o corpo a ela (envolvido por **chaves)**. Podemos cria-la da seguinte forma:

**.then((response) => response.json())**

* Para debuggar o código no navegador, na tela de inspeção da página entramos no Menu **Sources** – achamos o arquivo JS – colocamos um breakpoint em alguma linha e recarregamos a página
* Se colocarmos a palavra **debugger** dentro de uma função, ao executar nosso código, se o inspecionamento da página estiver aberto, o debugador vai parar o breakpoint onde estiver escrito esse **debugger**
* Esse comando acima é um macete de debugação no navegador, onde não precisamos achar onde está nosso código, o próprio debugger nos guia
* Se usarmos a função **document.getElementsByClassName(‘nome-da-classe’)**, podemos ter como retorno uma lista, pois uma classe pode ter vários subelementos
* Porém, se usarmos o **document.getElementById**, só teríamos 1 retorno, pois somente um elemento da página pode ter aquele ID
* Função **map** – passamos uma função transformadora que transforma um elemento em outro
* O método **Promise.all()** recebe um array de promises e te permite fazer algo somente quando finalizar todas as promises, usando um **then()** no final da estrutura