**Javascript – Módulo 5 – B7Web**

**Sumário**

[Aula 01 – O Que São Requisições 2](#__RefHeading___Toc3287_3265094356)

[Aula 02 – O Que É JSON 2](#__RefHeading___Toc3289_3265094356)

[Aula 03 – JSON.parse 2](#__RefHeading___Toc3291_3265094356)

[Aula 04 – JSON.stringfy 3](#__RefHeading___Toc3293_3265094356)

[Aula 05 – Código Síncrono vs Código Assíncrono 3](#__RefHeading___Toc3295_3265094356)

[Aula 06 – Callbacks 4](#__RefHeading___Toc3297_3265094356)

[Aula 07 – Promises 4](#__RefHeading___Toc3299_3265094356)

[Aula 08 – Fetch (1/2) 6](#__RefHeading___Toc3301_3265094356)

[Aula 09 – Fetch (2/2) 7](#__RefHeading___Toc3303_3265094356)

[Aula 10 – Async e Await 8](#__RefHeading___Toc3305_3265094356)

[Aula 11 – Fetch com POST 10](#__RefHeading___Toc3307_3265094356)

[Aula 12 – Upload de arquivos com JS 10](#__RefHeading___Toc3309_3265094356)

[Aula 13 – Thumbnails com JS 12](#__RefHeading___Toc3311_3265094356)

[Aula 14 – Thumbnails com FileReader 13](#__RefHeading___Toc3313_3265094356)

# **Aula 01 – O Que São Requisições**

Definição Resumida do Bonieky:

O que são requisições?

Toda vez que pedimos uma informação externa, estamos fazendo uma requisição.

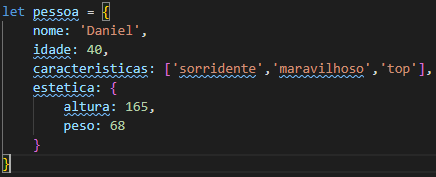
Um **conceito importante** é que para uma requisição acontecer, é necessário que haja **comunicação** entre quem está realizando a requisição e quem está recebendo esta requisição.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# **Aula 02 – O Que É JSON**

**JSON** **→ Javascript Object Notation**

Ele é uma sintaxe de objeto Javascript utilizado para fazer a comunicação entre requisições tanto para guardar como também para enviar e receber informações.



Formas de acesso as informações:

pessoa.nome → Daniel

pessoa.idade → 40

pessoa.caracteristicas → ['sorridente','maravilhoso','top']

pessoa.caracteristicas[0] → sorridente

pessoa.estetica → {altura: 165, peso: 68}

pessoa.estetica.altura → 165

Obs. Normalmente nos arquivos JSON, as declarações de objetos são feitas em linha:

let pessoa = {nome: 'Daniel', idade: 40, caracteristicas: ['sorridente','maravilhoso','top'],estetica: {altura: 165, peso: 68}}

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Aula 03 – JSON.parse

Quando fazemos uma requisição, recebemos um JSON em formato de uma string

'{"nome": "Daniel", "idade": 40}'

Obs.: Aqui todas as palavras contidas nesta string JSON tiveram que ser colocadas entre aspas duplas.

função **parse** → Transforma uma string de JSON em um JSON real.



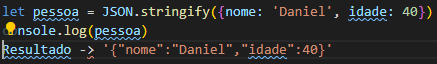
Obs.: O comando JSON.parse reconhece o formato de uma string JSON e só realiza a transformação da string para JSON real quando o conteúdo da string obedece a sintaxe de um JSON.

Exemplo: JSON.parse("bla bla bla") → Isto não é um JSON, portanto, não funcionará.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Aula 04 – JSON.stringfy

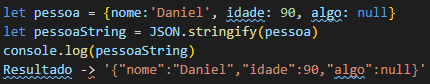
**stringify** → Função **inversa da função parse** no JSON. E função stringify **recebe um JSON real e tranforma em uma string JSON**.



Obs. Aqui, na operação inversa, o stringfy adicionou aspas duplas em todas as palavras encontradas, menos no número 40 que é um integer. Outro fator interessante, é que no conteúdo do item nome, a string ''Daniel' que estava entre aspas simples, foi colocada entre aspas duplas.

Veja como o JSON.parse trabalha inversamente ao JSON,stringify:





**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

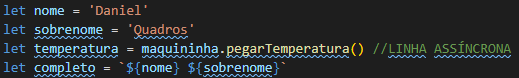
# Aula 05 – Código Síncrono vs Código Assíncrono

Diferença entre os códigos síncronos e assíncronos:

**Síncrono** → Ele roda linha a linha, ou seja, ele só vai para a linha posterior quando a linha atual for totalmente executada. Ex.:



**Assíncrono** → As funções assíncronas em um código é que tornam um código assíncrono. As funções assíncronas são funções que não necessitam obrigatoriamente que sua execução aconteça para que o código continue seu fluxo de execução. Se uma função assíncrona não for executada, o código continua para a próxima linha, e somente quando esta função for solicitada ela acontecerá.



Neste exemplo, em um código assíncrono, ele não espera a maquininha pegar a temperatura para executar a próxima linha. O código segue a diante até a última linha.

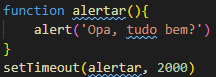
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Aula 06 – Callbacks

**Conceito:** É uma função Javascript que é enviada para outra execução. Geralmente uma execução assíncrona.

CALLBACK → I'm gonna call you back = Eu te ligo de volta (novamente).\*/

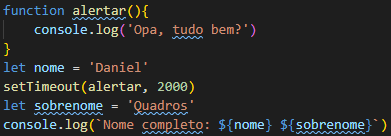
Vamos ver um exemplo:



Neste exemplo acima, a função ‘alertar’ é uma função de callbak, que será acionada pelo comando setTimeout apenas depois de 2000 mile segundos (2 segundos).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Outro exemplo:





Observe que neste código acima, foi executado primeiro a última linha, depois, o callback que necessitava de 2 segundos para acontecer, foi executado.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Aula 07 – Promises

Promises (Promessa) → Quando trabalhamos com funções assíncronas, nós, primeiramente temos promessas de resultados, até que a função assíncrona seja executada.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Promise → É um resultado daquela promessa.

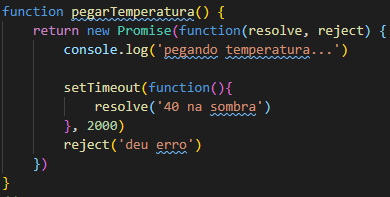
Uma função assíncrona pode ter 3 tipos de resultados:

1 - Nunca obter resultado.

2 - Obter resultado esperado, no momento esperado.

3 - Obter resultado não esperado.

Veja o exemplo:

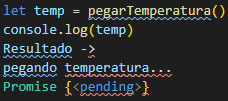


Observe a sintaxe para a criação de uma promisse:

**new Promise(function(resolve, reject) { comandos… } )**

Dentro da Promise, **obrigatoriamente**, preciso ter uma **função** com dois parâmetros, **resolve** (retorno positivo da promessa) e **reject**(quando ocorre algum erro, ou não há retorno)

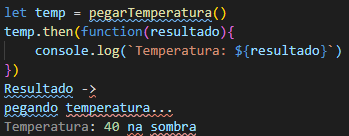
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**USANDO A PROMISE DO EXEMPLO ACIMA:**



Observe que além do console.log padrão da function pegarTemperatura, o console informou que a variável tem é uma Promise e está pendente **<pending>** de conclusão.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**VAMOS UTILIZAR A PROMISSE COM RETORNO ‘RESOLVE’:**



Obs. Aqui, a segunda linha do resultado apareceu apenas 2 segundos após a primeira linha ser executada.

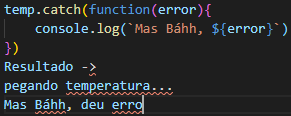
Agora que sabemos que a variável temp é uma Promise, podemos adcionar a esta variável o comando **then (então), ou seja, já que obteve resultado faça.…**

O que fizemos aqui neste exemplo foi chamarmos a nossa Promise (temp) com o then (então). **O then, chama o parâmetro 'resolve' da Promise** e utiliza o resultado obtido do parâmetro 'resolve' como parâmetro da function acionada pelo then. O que neste exemplo, após executar o console.log('pegando temperatura...'), Retornou o console.log da função do then → console.log(`Temperatura: ${resultado}`) trazendo o resultado: Temperatura: 40 na sombra.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**VAMOS UTILIZAR A PROMISSE COM RETORNO ‘REJECT’:**

Obs.: Aqui, para teste, retiramos os comandos do then que utilizamos para acionar o result, assim, podermos testar apenas o reject:



**catch** → O comando 'catch' é vinculado ao parâmetro '**reject**' da nossa Promise. Que neste exemplo. É o que é executado, caso a promise não seja cumprida, ou seja, caso o result não seja executado.

Obs. Em uma sequência, se o then for executado primeiro, o catch não será executado e, o contrário também é verdade. Se o catch for executado primeiro, o then não será executado. Funciona como um verdadeiro ou falso.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Aula 08 – Fetch (1/2)

Site de API feke para estudo, {JSON} Placeholder:

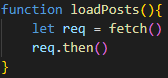
https://jsonplaceholder.typicode.com/

É uma API que retorna dados fictícios para podermos manipular e testar.

**fetch** → função que retorna uma Promise (promessa)

Este exemplo abaixo seria uma forma de executá-la, este é um exemplo bom para visualizarmos o funcionamento, porém não é a forma mais usual e nem a que iremos trabalhar:

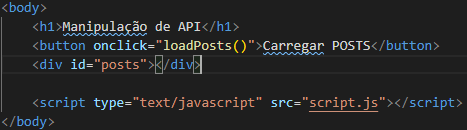
Armazenando o fetch em uma variável e depois, executando o then() para o retorno da Promise.

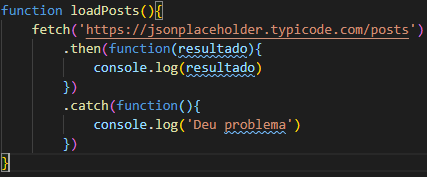


Porém, não trabalharemos desta Forma. E sim, da forma que está abaixo, executando o then diretamente vinculado ao fetch, já que sabemos que o fetch retorna uma Promise:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Para esta aula utilizamos o seguinte HTML:**





**O fetch tem dois parâmetros:**

**1º** - A **url** que queremos fazer a requisição. Que neste exemplo, pegamos a urla da API do site {JSON} Placeholder: e lá pegamos a API de posts para o exercício.

**2º** - O segundo parâmetro **é opcional**. É um objeto que **indica a forma de entrega desta requisição**. Que podem ser de um dos seguintes métodos:

- **GET** em que as informações vão todas via url.

- **POST** em que eu envio internamente (no corpo) os dados da requisição.

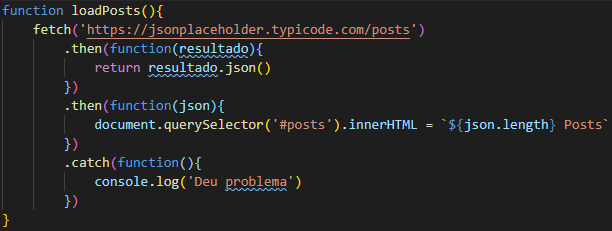
**E existem também os métodos PUT, DELETE** (porém, ainda não foi explicado) **entre outros**.

**Obs.: Se não definirmos nenhum método, por padrão, será utilizado o GET**.

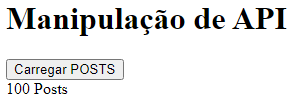
Ao executarmos a função acima, ela já trouxe pra nós a API da url do primeiro parâmetro do fetch. E podemos visualizar esta API no console do navegador contendo um array com 100 objetos (100 posts), que trazem as informações de cada post, e cada objeto no formato de JSON string.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Agora, no exemplo abaixo, utilizamos o nosso HTML para visualizar no navegador a quantidade de objetos que essa API está nos entregando.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**RESULTADO NO NAVEGADOR:**



No exemplo acima, substituímos o console.log que era execução do nosso .then pelo retorno de 'resultado'(API (Recebida em string json)).json

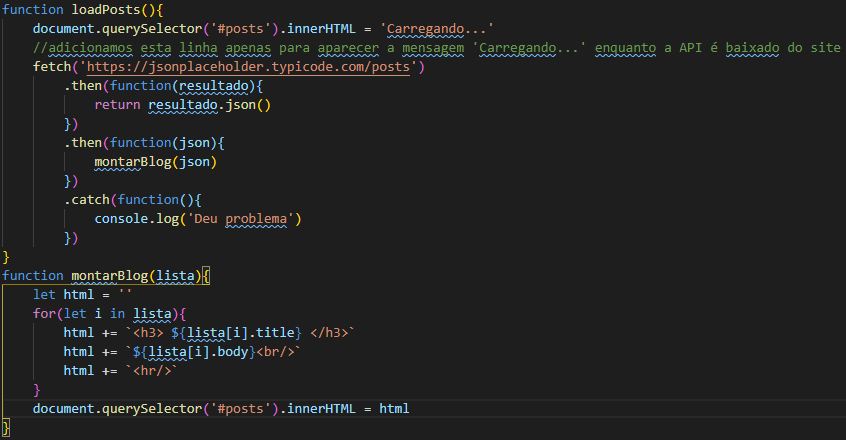
Este .json aqui retorna uma Promise(De acordo com o Boniek, nem ele mesmo entende porque que este comando aqui retorna uma Promise, mas o que interessa aqui é saber que ele faz isso.)

Pelo fato de que este .json retornar uma Promise, adicionamos outro .then após ele, e neste then é que obtivemos o resultado esperado.

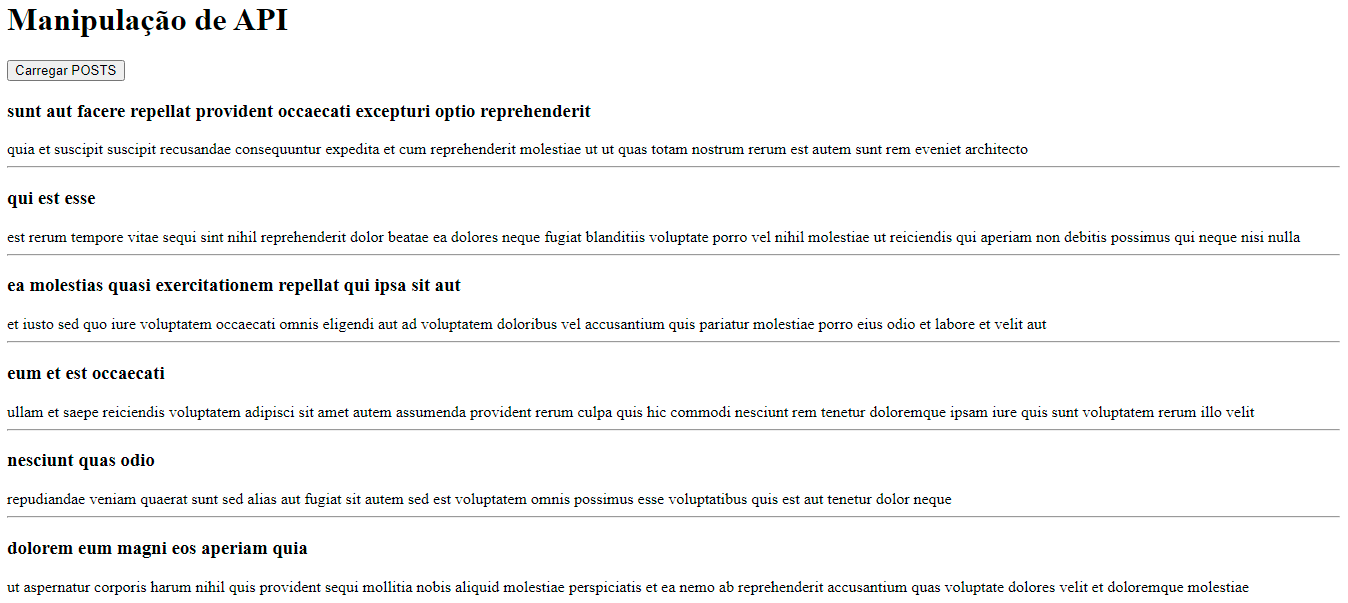
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Aula 09 – Fetch (2/2)

Nesta aula, colocamos em prática as teorias vistas na aula anterior, carregando as informações da nossa API no navegador. Veja o exemplo:



**NAVEGADOR COM OS 100 POSTS CARREGADOS:**



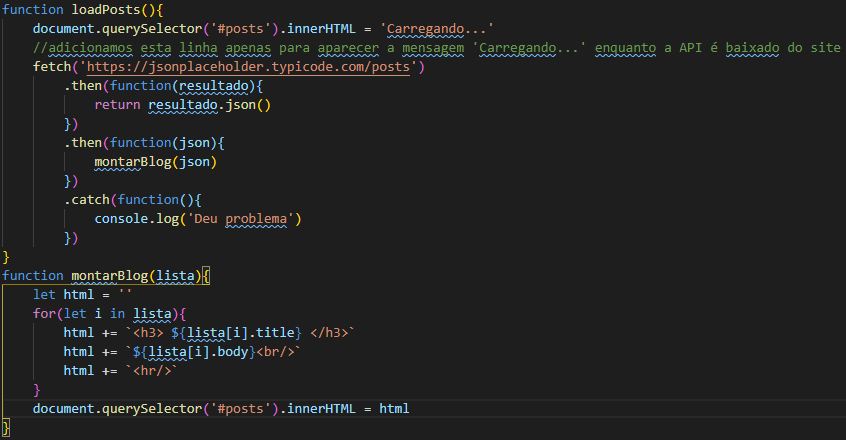
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Aula 10 – Async e Await

**async await** → Permique que juntemos as duas funções. No nosso exemplo anterior (por exemplo) poderiamos juntar a function '.then(function(resultado){' com a function '.then(function(json){' e o nosso código permaneceria limpinho.

Obs.: **O conjuto desses comandos "obrigam" o código a esperar o resultado.**

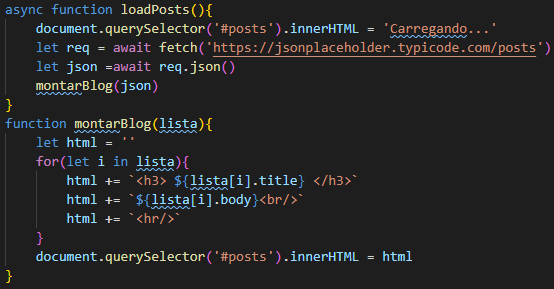
Para uma compreenção melhor destes comandos, vou colocar aqui a imagem do nosso javascript da aula anterior:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Obs.:** Utilizamos o comando async antes do comando function. **O async, sozinho, não tem utilidade**, para que haja o funcionamento, **precisamos utilizar o await em conjunto com ele**. Veja o exemplo:

Agora com o async await, o mesmo código fica com a seguinte sintaxe:



O await posicionado antes de uma promise, faz com que o código fique aguardando a promise ser satisfeita para que o código continue sua execução.

Obs.: Se estivermos criando uma função na sintaxe de variável o async é colocado antes dos parênteses que iniciam uma arrow function. Veja o exemplo:



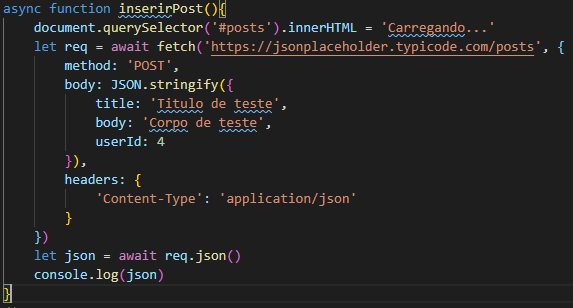
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Aula 11 – Fetch com POST

MÉTODO DE ENVIO COM post:

- **method: 'POST'** → Método de envio da requisição (2º parâmetro do fetch que, por padrão, serve para definir o metodo de envio).

Obs.: Este segundo parâmetro é um objeto, por padrão.



- **body** → As informações que serão enviadas, precisam estar dentro do body e para podermos enviar as informações, além de definirmos o método de envio, **utilizamos o** **JON.stringify**, para transformar o nosso objeto de envio em uma **string JSON**.

- **headers** → Também é um objeto. Além do método e do body, podemos também enviar cabeçalhos, um ou mais headers, quantos eu precisar.

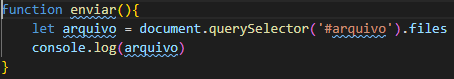
Conteúdos que podem ser enviados no headers:

'**Content-Type': 'application/json**' → Não foi explicado pelo Bonieky nesta aula, ele disse que mais adiante será ensinado.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Aula 12 – Upload de arquivos com JS

Vejamos os exemplos:



**files** → Podemos utilizar este comando em variáveis que venham do HTML através de um **<input type"file">**

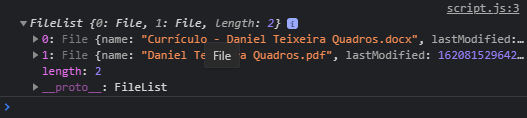
Este comando cria um objeto, permitindo que enviemos vários arquivos ao mesmo tempo.

Obs.: Para enviarmos diversos arquivos de uma só vez, precisamos adicionar o parâmetro '**multiple**’ lá no input do HTML.



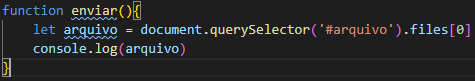
No console do javascript obtivemos a seguinte informação, após efetuarmos o procedimento de envio de arquivo:





\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

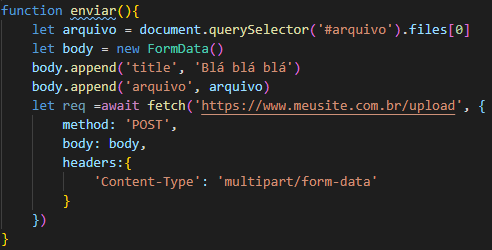
Neste próximo exemplo, retiramos o parâmetro multiple do nosso input no HTML:



Agora, adicionando o **[0]** no comando '**.files**', ao invés de retornar uma lista de arquivo '**Filelist**’ ele retorna um arquivo específico, '**File**'.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Agora, sabendo dessas informações e utilizando o modelo de comando acima, vamos fazer o UPLOAD deste arquivo:



Para fazermos envio de arquivos, não podemos utilizar o JSON.stringfy. Mesmo que com esse arquivo também tenham dados, eu não posso utilizar o JSON.stringify.

Para fazermos envio de arquivos, é preciso utilizar uma classe chamada **FormData()** com a seguinte sintaxe: **new FormData()**

Sobre essa sequência de comandos abaixo, ele não explicou a função do comando .append, embora eu tenha a impressão de já ter visto este comando em algum lugar, não consigo lembrar(

    body.append('title', 'Blá blá blá')

    body.append('arquivo', arquivo)

) Seguindo a lógica, este comando está recebendo dois parâmetros e pelo que vi, o primeiro parâmetro é utilizado para criar um item em um objeto e o segundo parâmetro é o valor que este item recebe. Neste caso, estas duas linhas retornariam o seguinte resultado:

**{title: ‘Bla bla bla’, arquivo: arquivo}**

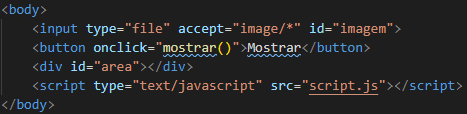
Como estamos enviando arquivos, precisamos modificar também o Content-Type e aqui, usamos o 'Content-Type': 'multipart/form-data' para envio de arquivos.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

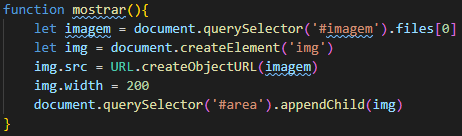
# Aula 13 – Thumbnails com JS

Esta sequência é muito útil, quando o usuário vai fazer um UPLOAD de uma imagem, porém, queremos mostrar pra ele no navegador, uma cópia ou miniatura do que será enviado.

**UTILIZADO O SEGUINTE HTML:**

****

**JAVASCRIPT:**

****

**let imagem = document.querySelector('#imagem').files[0]** → recebe a imagem do input imagem

**let img = document.createElement('img')** → Variável img recebe a criação de uma img para HTML

**img.src = URL.createObjectURL(imagem)** → Salva o endereço(src) da imagem do computador do usuário e cria uma URL com esse endereço, possibilitando a apresentação da mesma no navegador.

**img.width = 200** → Definimos a largura da imagem.

**document.querySelector('#area').appendChild(img)** → **Adiciona** img à <div> area do HTML.

**appendChild(img)** → Utilizamos no lugar do innerHTML. Este comando, ao invés de receber um texto, neste caso, receberá a imagem. Ele pega um conteúdo que já tem e adiciona mais outro.

**NAVEGADOR:**



**NO PRÓXIMO EXEMPLO, MOSTRA COM A ADIÇÃO DE MAIS 1 ARQUIVO DE IMAGEM SEM REMOVER DA TELA O PRIMEIRO ARQUIVO. ESSA POSSIBILIDADE ACONTECE POR CAUSA DO COMANDO appendChild(img)**



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# **Aula 14 – Thumbnails com FileReader**

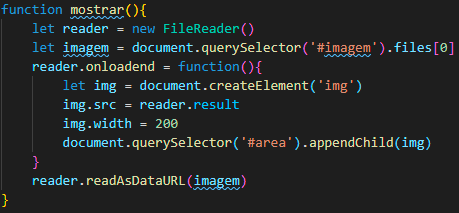
Criaremos uma alternativa para o exercício realizado na aula anterior (Aula 13)

O método que utilizamos na aula passada foi com código síncrono e o que utilizaremos nesta aula será levemente assíncrono.

Obs.: Essa definição de 'levemente assíncrono' foi dado pelo próprio Bonieky'.

Pra esta aula utilizamos o mesmo HTML da aula passada:

Abaixo o Javascript criado:



**let reader = new FileReader() →** Define a classe FileReader para a variável reader.

**let imagem = document.querySelector('#imagem').files[0] →** pega a imagem do input HTML.

**reader.onloadend = function(){ //onloadend →**  Quando o carregamento finalizar.

**let img = document.createElement('img') →** Variável img recebe a criação de uma img para HTML.

**img.src = reader.result →** Vai receber o resultado, ou seja, a url da imagem para podermos exibir.

**reader.readAsDataURL(imagem)//readAsDataURL** → Transforma o endereço da imagem que está no computador para um endereço no formato url.

O Bonieky comentou que pessoalmente ele prefere o método da aula anterior por que é um método síncrono. Mas ele também confirma que é apenas questão de gosto mesmo.

O resultado no navegador, foi exatamente igual ao resultado da aula passada.