**Javascript**

**Frontend**

Front-End Developer: Em tradução literal significa desenvolvedor da “parte da frente”. O que geralmente entra nessa categoria: HTML, CSS, Sass, Less, Bootstrap, JavaScript, ES5, HTML DOM, JSON, XML, jQuery, Angular, React, Ember.js, Redux, Storybook.

**Backend**

Back-End Developer: Em tradução literal significa desenvolvedor da “parte de traz”. O que geralmente entra nessa categoria: PHP, ASP, C++, C#, Java, Python, Node.js, Backbone.js, Express.js, Meteor.js, Grunt, GraphQL, Ruby, REST, GO, SQL, SQLite, MongoDB, [Firebase](http://firebase.com/), [Parse.com](http://parse.com/), PaaS (Azure and Heroku)

**FULLSTACK**

Full-Stack Developer: Em tradução literal significa desenvolvedor da “pilha completa”. O que geralmente entra nessa categoria: tecnologias Front-End, tecnologias Back-End, pilhas (LAMP, WAMP, LEMP, MEAN, Django Stack, Ruby on Rails)

**Notas**:

1. Não é porque uma tecnologia faz parte de uma categoria, que o desenvolvedor daquela categoria (ex. Front-End) vai saber de todas as tecnologias que existem naquela categoria.
2. Dentro das pilhas ainda deve se considerar coisas como o Sistema Operacional (Linux, Windows, etc), o servidor HTTP (Apache, Nginx, etc), e tecnologias relacionadas.

**Banco de Dados**

Javascript surgiu em 1995.

Surgiu com o nome de Mocha.

Foi lançado com o nome de Livescript, ainda em 1995. Brendan Eich, criador do Javascript.

Em 1995, Brendan Eich trabalhava na Netscape. Eles decidiram lançar um navegador chamado Netscape Navigator.

Mas tinha um grande problema. Na época as coisas não eram tão dinâmicas, era necessário requisitar dados do servidor a todo instante.

Foi necessário criar uma tecnologia mais dinâmica pra rodar nos navegadores. E assim surgiu o Javascript. Mas peraí, o nome não era “Livescript”?

**O Java era a tecnologia mais querida em 1995**. Então a Netscape aproveitou o “hype” e alugou o nome Java na sua tecnologia, transformando assim em Javascript.

**Javascript** e **Java** são completamente diferentes.

**Typescript** é o Javascript modificado.

O Javascript transformou a web. Antigamente as tecnologias eram propriedades de empresas, cada navegador rodava determinadas tecnologias.

O Javascript nasceu aberto, de código livre, qualquer um pode ver, alterar, melhorar, etc.

Antes, a preocupação dos navegadores era dar suporte ao máximo de tecnologias.

Depois do Javascript, a preocupação é interpretar o Javascript da maneira mais rápida e eficiente.

Ecmascript é uma especificação. É do ecmascript que saem todas as definições, atualizações e novas melhorias no Javascript.

O mais famoso deles é o ecmascript 6, ou ES6, que trouxe diversas melhorias.

O navegador nem sempre se adapta ao ecmascript, por isso nós temos “transpiladores” que convertem o código novo em um que os navegadores compreendem.

**ALGORITIMO**. O computador é burro. Precisamos passar para ele uma receita, passo a passo, do que precisa ser feito.

**console.log( ):** imprime na tela oque precisa ser escrito ou aparecer. Exemplo: **console.log(“Hello Word, Javascript”)**

**Variáveis** são espaços na memória do computador onde você pode armazenar dados. Você começa declarando uma variável com a palavra-chave “**var**” (menos recomendado, se aprofunde mais para uma explicação) ou a palavra chave “**let**”, seguida por qualquer nome que você queira chamá-la: **let minhaVariavel;**

1. **var**
2. **const**
3. **let**

**OS TIPOS**

1. **let string = “STRING**” (palavra)
2. **let number = 10** (número)
3. **let decimal = 10.10** (número com ponto ou virgula)
4. **let boolean = true** (ou false)
5. **let undef = undefinid** (nunca foi definido)
6. **let nullexample = null** (não tem valor)

**COMPARATIVOS**

* **if** (se verdadeiro)
* **else** (se for falso)
* **%** (resto)

**CONDICIONAIS**

* **=** (não é igual, mas sim recebe)
* **==** (significa igual)
* **===** (compara também o tipo, não só a variável)
* **!==** (diferente)
* **SWITCH...CASE**

Quando há varias coisas ou informações a serem comparadas. É uma alternativa ao if else. Sempre colocar o **break** no final. Também é mais fácil se precisar mudar uma variável, pois alteraríamos apenas em um lugar.

* **default:** quando não entrar em nenhum caso, entra no defalt.

**OPERADORES LÓGICOS**

* Ou = **||:** a condição é verdadeira se UM ou OUTRO for verdadeiro.
* E = **&&:** a condição é verdadeira se TODOS forem verdadeiros.

**LAÇOS**

Enquanto a condição for verdadeira, o ciclo se repete infinitamente:

* **while:** A declaração while cria um laço que executa uma rotina especifica enquanto a condição de teste for avaliada como verdadeira. A condição é avaliada antes da execução da rotina:

while (condição) {

rotina

}

* **do:** é um while invertido: faça isso enquanto...

num = 0

do {

console.log{num}

num++

} while (num <=10}

* **for:** A instrução for cria um loop que consiste em três expressões opcionais, dentro de parênteses e separadas por ponto e vírgula, seguidas por uma declaração ou uma sequência de declarações executadas em sequência:
* for ([*inicialização*]; [*condição*]; [*expressão final*])

*declaração*

Usando **for**

A declaração **for** começa declarando a variável i e inicializando-a como 0. Ela verifica se i é menor que nove, executa as duas instruções subsequentes e incrementa 1 a variável i após cada passagem pelo loop.

for (var i = 0; i < 9; i++) {

console.log(i);

// more statements

}

* **for..in:** O laço for...in interage sobre propriedades enumeradas de um objeto, na ordem original de inserção. O laço pode ser executado para cada propriedade distinta do objeto. **Tem a ver com a posição do item.**
* **for...of:** O loop for...of percorre objetos iterativos (incluindo Array, Map, Set, o objeto arguments e assim por diante), chamando uma função personalizada com instruções a serem executadas para o valor de cada objeto distinto. **Tem a ver com o valor do item.**

**Array:** O objeto Array do JavaScript é um objeto global usado na construção de 'arrays': objetos de alto nível semelhantes a **listas:**

var frutas = ['Maçã', 'Banana'];

* **array[ 1 ]:** define o item Banana que está na posição 1.
* **array.push(‘Laranja’):** adiciona um item a lista.
* **array.pop( ):** remove o último elemento do array.
* **array.length( ):** A propriedade length representa um inteiro de 32-bit sem sinal, que especifíca o número de elementos em um array.

**Exemplo de média:**

const numbers = [10, 20, 30]

let sum = 0

**for** (**let** num **of** numbers) {

sum = sum + num

}

const media = sum / numbers.legth

console.log(media)

**OBJETOS**

É outro tipo de variável (além das variáveis simples dos arrays)

**const** num = 10

**const** array = [1, 2, 3]

**const** pessoa {

nome: “José”

idade: 49

casado: true

notas: [5, 8, 100]

}

**console.log**(pessoa[“nome”] ou **(pessoa.nome)**

**console.log**(pessoa[“idade”] ou **(pessoa.idade)**

**console.log**(pessoa[“notas”][2]: **(pessoa.notas[2])**

**FUNÇÕES**

Funções são blocos de construção fundamentais em JavaScript. Uma função é um procedimento de JavaScript - um conjunto de instruções que executa uma tarefa ou calcula um valor. Para usar uma função, você deve defini-la em algum lugar no escopo do qual você quiser chamá-la.

A definição da função (também chamada de declaração de função) consiste no uso da palavra chave function (en-US), seguida por:

* Nome da Função.
* Lista de argumentos para a função, entre parênteses e separados por vírgulas.
* Declarações JavaScript que definem a função, entre chaves { }.

Exemplo de função básica:

**function** ola ( ) {

console.log(“OLA”)

}

ola( )

**Arrow functions**

Uma **expressão arrow function** possui uma sintaxe mais curta quando comparada a uma expressão de função ([*function expression*](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/function)) e não tem seu próprio [*this*](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/this), [*arguments*](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Functions/arguments), [*super*](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/super) ou [*new.target*](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/new.target). Estas expressões de funções são melhor aplicadas para funções que não sejam métodos, e elas não podem ser usadas como construtoras (constructors).

**FUNÇÕES RECURSIVAS**

Recursão

O ato de uma função chamar a si mesma, a recursão é usada para resolver problemas que contêm subproblemas menores. Uma função recursiva pode receber duas entradas: um caso base (termina a recursão) ou um caso recursivo (retoma a recursão).

[Exemplos](https://developer.mozilla.org/en-US/docs/Glossary/Recursion#examples)

[A função recursiva chama a si mesma até que a condição seja atendida](https://developer.mozilla.org/en-US/docs/Glossary/Recursion#recursive_function_calls_itself_until_condition_met):

function fatorial (*n*) {

    if (*n* <= 2) return *n*

    return fatorial(*n*-1) \* *n*

}

const numero = 5

const resposta = fatorial(numero)

console.log(resposta)

Resposta no console: 120

**Hoisting (Elevação)**

JavaScript **Hoisting** refere-se ao processo pelo qual o interpretador parece mover a *declaração* de funções, variáveis ​​ou classes para o topo de seu escopo, antes da execução do código.

O içamento permite que as funções sejam usadas com segurança no código antes de serem declaradas.

*Declarações* de variáveis ​​e classes também são içadas, então elas também podem ser referenciadas antes de serem declaradas. Observe que isso pode levar a erros inesperados e geralmente não é recomendado.

// Hoisting

ola()

function ola () {

    console.log("OLA")

}

**DESTRUCTOR E CONCATENAÇÃO**

É uma forma alternativa de ler dados dentro dos objetos e dos arrays.

/\*

const pessoa = {

    nome: "Marcos",

    sobrenome: "Rosalino",

    idade:49

}

const {nome, sobrenome, idade} = pessoa

//const nome = pessoa.nome

//const sobrenome = pessoa.sobrenome

//const idade = pessoa.idade

console.log(nome)

console.log(sobrenome)

console.log(idade)

const array = [1, 2, 3, 4]

const [a, b] = array

console.log(a)

console.log(b)

const pessoaComMaisDados = {

    ...pessoa

}

console.log(pessoaComMaisDados)

const pessoaAtualizada = {

    ...pessoa,

    idade: 50,

    senha: "121212"

}

console.log(pessoaAtualizada)

\*/

const array = [1, 2, 3, 4]

const [a, b] = array

const newArray = [0 ,...array, 100]

console.log(newArray)

**Array.prototype.forEach()**

O **forEach()**método executa uma função fornecida uma vez para cada elemento da matriz.

const array = [1, 5, 4, 3, 10, 7, 8, 20, 12]

// forEach: passa por todos os elementos do array

array.forEach((*elemento*, *posicao*) => {

    //console.log(`${elemento} - posicao ${posicao}`)

})

/\*

find: retorna o elemento

const valor = array.find((elemento) => {

    //return elemento === 10

    if (elemento === 10) return true

    return false

})

console.log(valor)

\*/

// findIndex: retorna a posição do elemento

const index = array.findIndex(*elemento* => elemento === 10)

    //if (elemento === 12) return true

    //return false

console.log(index)

// some: retorna true se pelo menos um número satisfazer a condição.

const some = array.some(*elemento* => elemento >= 10)

console.log(some)

// every: retorna true se todos os números respeitarem a condição.

const every = array.every(*elemento* => elemento >= 1)

console.log(every)

const newArray = [1, 2, 3, 4, 5, 10, 24, 80]

// splice: corta o array. O primeiro parametro é o inicio e o segundo é a quantidade.

//console.log(newArray.splice(4, 1))

/\*console.log(newArray.map((elemento) => {

    return elemento \* 2

}))

\*/

//filter: filtrar os elementos dentro do array.

console.log(newArray.filter((*elemento*) => {

    return elemento % 2 === 1

}))

// reduce: reduz o array para algum outro elemeto.

console.log(newArray.reduce((*acumulado*, *elemento*) => {

    return acumulado + elemento

}, 0))

**OBJETOS E FUNÇÕES**

**Object.keys()**

O **Object.keys()**método retorna uma matriz dos próprios **nomes** de propriedade enumeráveis ​​de um determinado objeto , iterados na mesma ordem que um loop normal faria.

**Object.entries()**

O **Object.entries()**método retorna uma matriz dos próprios [key, value]pares de propriedades com chave de string enumeráveis ​​de um determinado objeto. Isso é o mesmo que iterar com um [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in)loop, exceto que um for...inloop também enumera propriedades na cadeia de protótipos.

A ordem da matriz retornada por Object.entries()é a mesma fornecida por um [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in)loop. Se houver necessidade de ordenação diferente, a matriz deve ser classificada primeiro, como Object.entries(obj).sort((a, b) => b[0].localeCompare(a[0]));.

**Object.values()**

O **Object.values()**método retorna uma matriz dos próprios valores de propriedade enumeráveis ​​de um determinado objeto, na mesma ordem que a fornecida por um [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in)loop. (A única diferença é que um for...inloop também enumera propriedades na cadeia de protótipos.)

const person = {

    name: "Marcos",

    lastName: "Rosalino"

}

// Object.keys() - Cria um array com todas as chaves do objeto.

console.log(Object.keys(person))

// Object.values() - Cria um array com todas os valores do objeto.

console.log(Object.values(person))

// Object.entries() - Cria um array com um array de objetos e valores.

console.log(Object.entries(person))

const livros = {

    'Livro A': 19.90,

    'Livro B': 97.00,

    'Livro C': 20.00

}

const values = Object.values(livros)

//console.log(values)

const sum = values.reduce((*val*, *acc*) => val + acc, 0)

console.log(sum)

**FUNÇÕES ASSÍNCRONAS**

**função assíncrona**

Uma função assíncrona é uma função declarada com a palavra-chave **async** e a palavra-chave **await** é permitida dentro dela. As palavras-chave **async** e **await** permitem que o comportamento assíncrono e baseado em promessas seja escrito em um estilo mais limpo, evitando a necessidade de configurar explicitamente as cadeias de promessas. São funções que tem um formato especial que vão buscar os dados e aguardar até que esses dados estejam disponiveis e possam ser usados.

* **API:** API significa Application Programming Interface (Interface de Programação de Aplicação/Aplicativos). No contexto de APIs, a palavra Aplicação refere-se a qualquer software com uma função distinta. A interface pode ser pensada como um contrato de serviço entre duas aplicações. O intuito de uma API é trocar dados entre sistemas diferentes, na maior parte das vez essas trocas de dados tem como objetivo automatizar processos manuais e/ou permitir a criação de novas funcionalidades. Esta interface de programação é um conjunto de padrões de programação que permitem a construção de aplicativos e a sua utilização.

Qual a diferença entre **APP** e **API**?

No app podem ser utilizados ‘enes’ APIs. Uma **API** (Application Programming Interface) é um conjunto de comandos, funções, protocolos e objetos que os programadores podem usar para criar softwares ou interagir com sistemas externos.

**O que é API?**

API ou *Application Programming Interface*, que em português quer dizer Interface de Programação de Aplicações, **é um conjunto de funções e procedimentos que permitem a integração de sistemas**, permitindo a reutilização das suas funcionalidades por outras aplicações ou software.

Uma API é utilizada para trocar dados entre diferentes tipos de software a fim de automatizar procedimentos e desenvolver novas funcionalidades.

**Como a API funciona?**

Uma **API é uma espécie de ponte que liga diferentes tipos de software ou aplicações e pode ser criada em várias linguagens de programação**. Para além de um bom desenvolvimento, uma API deve ter uma documentação clara e objetiva para facilitar a sua implementação.

Além disso, é frequentemente utilizado um formato de dados predefinido para compartilhar informação entre sistemas, com o objetivo de [**obter a integração**](https://www.sydle.com/br/blog/integracao-de-sistemas-6140d39a84679b13bf127a93/) entre eles. Os mais usados são XML (*Extensible Markup Language*), YAML (originalmente *Yet Another Markup Language*, mas oficialmente *YAML Ain’t Markup Language*) e JSON (*JavaScript Object Notation*) para as aplicações web.

Existe também um padrão nas APIs web chamado REST (*Representational State Transfer*), que é um conjunto de regras e definições que permite o desenvolvimento de projetos com interfaces bem definidas.

**Quais tipos de API existem?**

Existem basicamente **quatro tipos de API considerando suas políticas de compartilhamento**, como veremos abaixo.

**API segundo suas políticas de uso**

1. **APIs públicas ou abertas**

As APIs públicas são também conhecidas como APIs abertas e **estão disponíveis para outros usuários ou programadores utilizarem com restrições mínimas** ou, em alguns casos, de forma totalmente acessível.

2. **APIs privadas ou internas**

As APIs privadas ou internas são ocultadas aos usuários externos e **são expostas apenas para os sistemas internos de uma organização**. São utilizadas para o desenvolvimento interno da empresa, otimizando a produtividade e a reutilização dos serviços.

3. **APIs de parceiros de negócio**

As APIs de parceiros comerciais **são aquelas que são expostas entre os membros de um acordo comercial**. Uma vez que não estão disponíveis para todos, é necessária uma autorização especial para sua utilização.

4. **APIs compostas**

As APIs compostas **utilizam diferentes dados ou APIs de serviços** e permitem aos programadores acessar múltiplos pontos finais.

Também podemos dividir as **APIs em quatro de acordo com o que oferecem ou com seus casos de uso**, como você verá agora.

**APIs de acordo com os seus casos de uso**

**1. API de dados**

As APIs de dados fornecem a vários bancos de dados ou provedores [**SaaS**](https://www.sydle.com/br/blog/software-as-a-service-60228fe96d79af5e998f9bc5/) (*Software as a Service* ou Software como Serviço) um **acesso CRUD (*Create, Read, Update, Delete*) a conjuntos de dados subjacentes**, permitindo a comunicação entre uma aplicação e um sistema de gestão de base de dados.

**2. API de sistemas operacionais**

Este grupo de APIs **define como as aplicações utilizam os recursos e serviços disponíveis do sistema operacional**. Assim, cada OS (*Operative System*) possui um conjunto de APIs, por exemplo, a API do Windows ou a API do Linux possuem *kernel-user space API* e *kernel internal API*.

**3. APIs remotas**

Este grupo define os padrões de interação que as aplicações têm em diferentes dispositivos, ou seja, **um software acessa determinados recursos localizados fora do dispositivo requerente**, como o nome indica. Como duas aplicações se ligam remotamente através de uma rede, as APIs remotas utilizam protocolos para conseguir a ligação.

**4. APIs web**

Esta classe de APIs é a mais comum, uma vez que as APIs web fornecem dados que os dispositivos podem ler e transferir entre sistemas baseados na web ou arquitetura cliente-servidor.

**Protocolos de API**

Os protocolos de API **permitem padronizar a troca de dados entre diferentes serviços web**. Isto proporciona a oportunidade de acessar capacidades em diferentes sistemas, por meio de diferentes linguagens de programação e em diferentes sistemas operacionais.

Os mais importantes estão listados abaixo:

***Remote Procedure Call* (RPC)**

O chamado de procedimento remoto ou RPC permite que as APIs web **possam se aderir aos princípios de intercâmbio de recursos**. O objetivo deste protocolo é definir a interação entre aplicações com base num programa que solicita dados - cliente - e outro que os fornece - servidor - remotamente.

***Service Object Access Protocol* (SOAP)**

É um protocolo verdadeiramente leve para o intercâmbio de informação estruturada num ambiente descentralizado e distribuído. As suas especificações contêm as regras de sintaxe dos pedidos e respostas enviadas pelos pedidos.

As aplicações que cumprem estes princípios permitem o envio de mensagens XML entre o sistema via HTTP (*Hypertext Transfer Protocol*) ou SMTP (**Simple Mail Transfer Protocol**).

***Representational State Transfer* (REST)**

REST é um estilo de arquitetura de software com seis restrições para a criação de aplicações que correm sobre HTTP, especialmente serviços web.

É considerada **uma alternativa SOAP**, uma vez que muitos programadores têm dificuldade em utilizá-la porque têm de escrever grandes quantidades de código para realizar uma tarefa. Por outro lado, a REST segue uma lógica diferente, uma vez que facilita a disponibilidade de dados como recursos.

**GraphQL**

O GraphQL surgiu em resposta à necessidade de desenvolvimento mais rápido, carregamento de dados mais eficiente e maior adaptabilidade móvel.

Esta linguagem de consulta para API **permite ao cliente detalhar os dados de que necessita e simplifica a adição de informação** por meio de múltiplas fontes.

**Quais são as vantagens das APIs?**

Como quase qualquer elemento ou solução da [**Indústria 4.0**](https://www.sydle.com/br/blog/tecnologias-industria-4-0-5fa55ba5d70a374444371979/), as APIs são capazes de proporcionar vários benefícios quando utilizadas.

A seguir estão os principais:

* **Aplicações**: o acesso às APIs assegura uma maior flexibilidade nos processos de transferência de informação.
* **Alcance:** por meio delas, é possível criar camadas de aplicações com o objetivo de distribuir informação a diferentes audiências.
* **Personalização:** pode também servir como uma solução para criar experiências diferenciadas do usuário, permitindo que os protocolos, funções e comandos sejam adaptados de acordo com requisitos específicos.
* **Eficiência:** ao terem conteúdos publicados automaticamente e disponibilizados em múltiplos canais simultaneamente, as APIs permitem uma distribuição mais eficiente dos dados.
* **Adaptação:** um dos grandes benefícios das APIs é a sua capacidade de adaptação às mudanças através da migração de dados e flexibilidade de serviços.

**Exemplos de API**

A seguir estão alguns dos exemplos mais populares de API:

* **Google Maps:** graças às normas aplicadas pelo Google, a maioria dos sites pode utilizar as APIs do Google Maps para integrar mapas.
* **Vulcan:** esta API multiplataforma permite aos programadores criar interfaces gráficas de alta qualidade em tempo real nas aplicações, proporcionando uma comunicação mais rápida e mais eficiente entre as aplicações e as unidades de processamento gráfico.
* **Skyscanner:** esta plataforma de metabusca facilita aos viajantes a procura de melhores tarifas para os seus voos. Também fornece uma API para parceiros comerciais que suporta XML e JSON para troca de dados.
* **Weather API:** é um fornecedor de serviços de geolocalização e informação meteorológica com diversas APIs que vão desde previsões meteorológicas a pesquisas de fuso horário, astronomia e muito mais.

**Inovação com APIs**

Finalmente, explicaremos como uma API pode se tornar uma ferramenta ideal para facilitar a integração com aplicações.

**APIs para pagamentos**

Com este tipo de API, o seu negócio pode integrar sistemas dentro do website ou aplicação da sua empresa com a intenção de **expandir os métodos de pagamento** para os seus produtos e serviços.

**Redes sociais**

As redes sociais também oferecem inovações com APIs para enriquecer a experiência do usuário e **incorporar funcionalidades para obter informações sobre visitantes ou criar usuários e perfis no seu site** a partir do Facebook, contas Google, entre outras.

**Localização**

Algumas das APIs mais populares que já mencionámos são aquelas que permitem oferecer informações e serviços específicos aos usuários num local específico, melhorando a experiência.

**Próximos passos**

Tendo em conta o conceito de APIs e os exemplos apresentados, podemos compreender a sua utilidade para as aplicações, dado que fornecem um grande número de recursos para reforçar as funcionalidades que oferecem, especialmente as que permitem aos sistemas se comunicarem.

Como podemos ver, implementar soluções que facilitem a integração de sistemas pode ser uma vantagem competitiva e um diferencial em termos de experiência do usuário. Portanto, venha ler sobre [**plataformas** ***low-code e no-code***](https://www.sydle.com/br/blog/low-code-no-code-614375cce83193407de36d19/) para continuar impulsionando a sua organização.

* **fetch( )**

O método global **fetch()**inicia o processo de buscar um recurso da rede, retornando uma promessa que é cumprida assim que a resposta estiver disponível.

A promessa resolve para o [Response](https://developer.mozilla.org/en-US/docs/Web/API/Response)objeto que representa a resposta à sua solicitação.

Uma [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch)promessa só é rejeitada quando um erro de rede é encontrado (o que geralmente ocorre quando há um problema de permissão ou similar). Uma [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch)promessa *não* rejeita erros HTTP ( 404, etc.). Em vez disso, um then()manipulador deve verificar as propriedades [Response.ok](https://developer.mozilla.org/en-US/docs/Web/API/Response/ok)e/ou [Response.status](https://developer.mozilla.org/en-US/docs/Web/API/Response/status).

WindowOrWorkerGlobalScopeé implementado por [Window](https://developer.mozilla.org/en-US/docs/Web/API/Window)e [WorkerGlobalScope](https://developer.mozilla.org/en-US/docs/Web/API/WorkerGlobalScope), o que significa que o fetch()método está disponível em praticamente qualquer contexto no qual você queira buscar recursos.

O fetch()método é controlado pela connect-srcdiretiva da [Política de Segurança de Conteúdo](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy) em vez da diretiva dos recursos que está recuperando.

**Nota:** Os fetch()parâmetros do método são idênticos aos do [Request()](https://developer.mozilla.org/en-US/docs/Web/API/Request/Request)construtor.

* **JSON( )**

**JSON** é basicamente um formato leve de troca de informações/dados entre sistemas. Mas **JSON** significa JavaScript Object Notation, ou seja, só posso usar com JavaScript correto? Na verdade, não, e alguns ainda caem nesta armadilha. O **JSON** além de ser um formato leve para troca de dados é também muito simples de ler.

**R**[**esumo**](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/JSON#summary)

O Objeto **JSON** contém métodos para parsing [JavaScript Object Notation](http://json.org/) ([JSON](https://developer.mozilla.org/pt-BR/docs/Glossary/JSON)) e conversão de valores para JSON. Ele não pode ser chamado ou construído e, além de suas propriedades de dois métodos, ele não possui uma funcionalidade interessante.

[**Descrição**](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/JSON#description)

[**JavaScript Object Notation**](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/JSON#javascript_object_notation)

JSON é uma sintaxe para serialização de objetos, matrizes, números, strings, booleanos, e [null](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/null). Baseia-se em sintaxe Javascript, mas é distinta desta: alguns Javascript não são JSON, e alguns JSON não são Javascript.

* **AWAIT**

O **await** operador é usado para esperar por um arquivo [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise). Ele só pode ser usado dentro de um [async function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) código JavaScript regular; no entanto, ele pode ser usado sozinho com [módulos JavaScript.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules)

[**Sintaxe**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await#syntax)

[rv] = await expression.

**expression**

A [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)ou qualquer valor para esperar.

**rv**

Retorna o valor cumprido da promessa ou o próprio valor se não for um Promise.

[**Descrição**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await#description)

A await expressão faz async com que a execução da função seja pausada até que a Promise seja resolvida (ou seja, cumprida ou rejeitada) e retome a execução da async função após a execução. Quando retomado, o valor da await expressão é o do preenchido Promise.

Se Promise for rejeitado, a await expressão lançará o valor rejeitado.

Se o valor da *expressão* após o await operador não for um Promise, ele será convertido em um [Promise resolvido](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve) .

Um await fluxo de execução dividido, permitindo que o chamador da função assíncrona retome a execução. Após o await adiamento da continuação da função assíncrona, a execução das instruções subsequentes ocorre. Se esta await for a última expressão executada por sua função, a execução continua retornando ao chamador da função uma pendência Promise para conclusão da await função 's e retomando a execução desse chamador.

const URL = 'https://jsonplaceholder.typicode.com/users'

async function teste () {

const resposta = await fetch(URL)

const dados = await resposta.json()

const usersEmails = dados.map ((obj) => {

return {

email: obj.email

}

})

console.log(usersEmails)

//console.log(dados)

}

teste()

**TRATAMENTO DE ERROS**

* **try...catch**

A instrução try...catch é composta por um bloco try e um bloco catch, um bloco finally ou ambos. O código no bloco try é executado primeiro e, se lançar uma exceção, o código no bloco catch será executado. O código no bloco finally sempre será executado antes que o fluxo de controle saia de toda a construção.

const URL = 'https://jsonplaceholder.typicode.com/userss'

const teste = async () => {

try {

const resposta = await fetch(URL)

if (resposta.status === 404) throw new Error ("Não encontrado")

const dados = await resposta.json()

const usersEmails = dados.map (({email}) => ({email}))

console.log(usersEmails)

} catch (error) {

console.log(error.message)

}

}

teste()

**Callback e promises**

Uma função de retorno de chamada é uma função passada para outra função como um argumento, que é então invocado dentro da função externa para completar algum tipo de rotina ou ação.

function greeting(name) {

alert('Hello ' + name);

}

function processUserInput(callback) {

var name = prompt('Please enter your name.');

callback(name);

}

processUserInput(greeting);

function upload () {

    return **new** *Promise* ((*resolve*, *reject*) => {

        setTimeout (() => {

            console.log("02 - UPLOAD Concluído")

            resolve()

        }, 2000)

    })

}

async function uploadDefoto () {

    console.log("01 - UPLOAD Iniciando")

    await upload()

    console.log("03 - Execução finalizada")

}

uploadDefoto()

**FACTORY.JS**

function Pessoa (*nome*, *sobrenome*) {

    let pessoa = {

        nome,

        sobrenome

    }

    pessoa.nomeCompleto = `${*nome*} ${*sobrenome*}`

    return pessoa

}

const pessoaA = Pessoa ("Marcos", "Rosalino")

const pessoaB = Pessoa ("Valdeli", "Rosalino")

console.log(pessoaA, pessoaB)

console.log(pessoaA.nomeCompleto)