JSON

JSON é uma forma de estruturar dados.

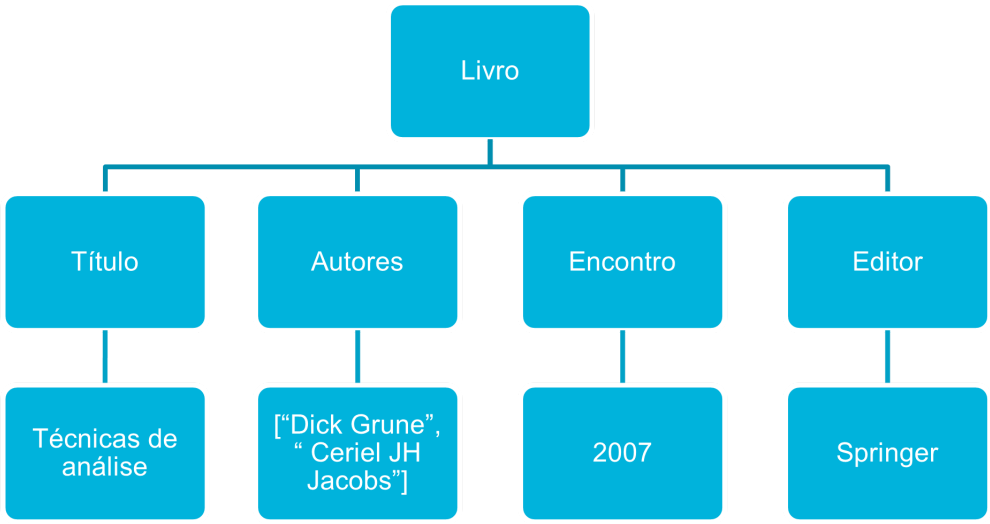
JSON é um formato de dados.

O documento JSON abaixo contém dados sobre um livro: seu título, autores, data de publicação e editora.

JSON

```json
{
    "Livro" :
    {
        "Titulo" : "Técnicas de Análise" ,
        "Autores" : [ "Dick Grune" , "Ceriel JH Jacobs" ] ,
        "Data" : "2007" ,
        "Editora" : "Primavera"
    }
}
```

Um objeto JSON é uma árvore



A estrutura de dados em árvore foi bem estudada por cientistas da computação e matemáticos.

Existem muitos algoritmos bem conhecidos para processar e percorrer árvores e o JSON podem aproveitar isso.

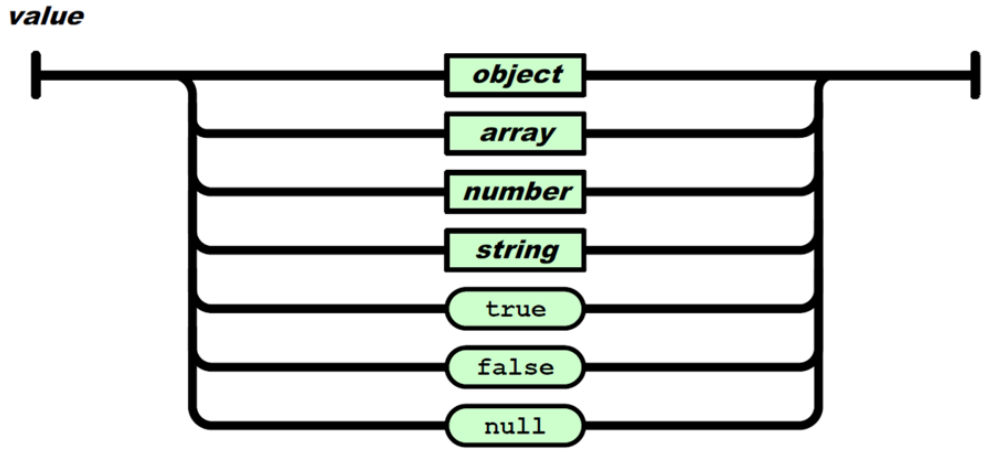
Schema Json

```
{
    "$esquema" : http://json-schema.org/draft-04/schema ",
    "tipo" : "objeto" ,
    "propriedades" : {
        "Livro" : {
            "tipo" : "objeto" ,
            "propriedades" : {
                "Título" : { "tipo" : "cadeia" } ,
                "Autores" : { "tipo" : "matriz" , " minItems " : 1 , "
maxItems " : 5 , "itens" : { "tipo" : "corda" }} ,
                "Data" : { "tipo" : "corda" , "padrão" : "^[0-9]{4}$" } ,
                "Editora" : { "tipo" : "corda" , " enum " : [ "Primavera"
, "MIT Press" , "Harvard Press" ]}
            } ,
            "necessário" : [ "Título" , "Autores" , "Data" ] ,
            "propriedades adicionais" : falso
        }
    } ,
    "necessário" : [ "Livro" ] ,
    "propriedades adicionais" : falso
}
```

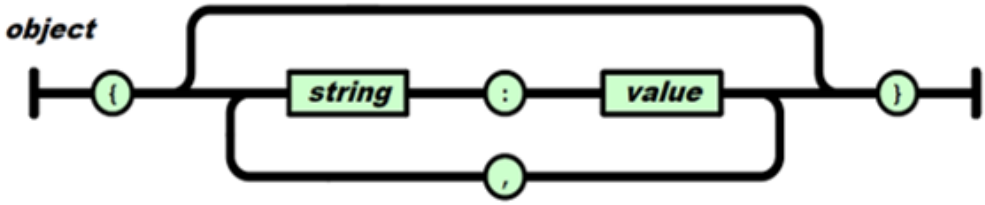
O formato de dados JSON

Uma instância JSON contém um único valor JSON.

Um valor JSON pode ser um object , array , number , string , true, false ou null:



Um objeto JSON tem zero ou mais pares string-colon-value , separados por vírgula e entre chaves:



Exemplo JSON

```
{
    "nome" : "João Doe" ,
    "idade" : 30 ,
    "casado" : true
}
```

Um objeto JSON pode estar vazio.

Este é um objeto JSON

```
{}
```

Você deve considerar os objetos JSON como contendo pares de chave/valor.

Assim como em um banco de dados onde as chaves primárias devem ser exclusivas, também em um objeto JSON as chaves devem ser exclusivas.

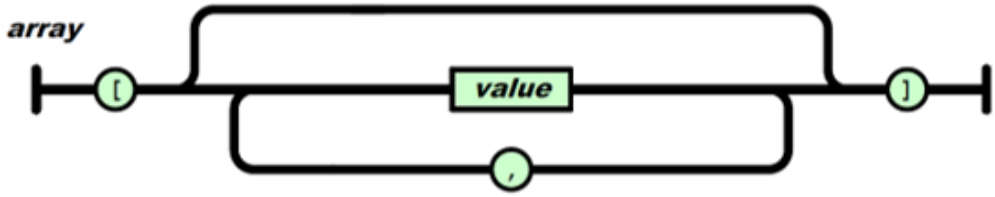
Este objeto JSON tem chaves duplicadas:

Chaves

```
{
    "Título" : " Uma história de Mark Twain " ,
    "Título" : " As Aventuras de Huckleberry Finn "
}
```

Matriz JSON

Uma matriz JSON é usada para expressar uma lista de valores. Uma matriz JSON contém zero ou mais valores, separados por vírgula e entre colchetes:



Matriz JSON

```
{
    "name": "John Doe",
    "age": 30,
    "casado": true,
    "irmãos": [ "John" , "Maria" , "Pat" ]
}
```

Matriz de objetos

Cada item em uma matriz pode ser qualquer um dos sete valores JSON.

Matriz

```
{
    "name": "John Doe",
    "age": 30,
    "casado": true,
    "irmãos": [
        {
            "nome" : "João" ,
            "idade" : 25
        } ,
```

```
        verdade ,
        "Olá Mundo"
    ]
}
```

JSON não permite strings de várias linhas.

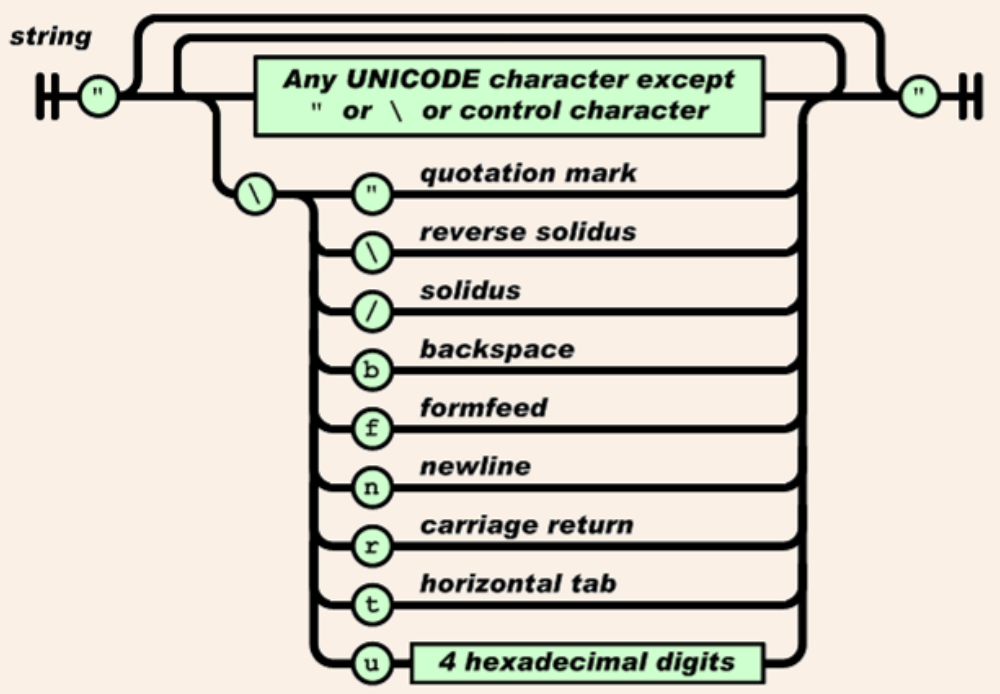
Válido

```
{
    "comentário": "Este é um comentário muito, muito longo"
}

{
    "comment": "Este é um comentário muito, \n muito longo"
}
```

Inválido

```
{
    "comentário": "Este é um
comentário muito, muito longo"
}
```



As strings JSON são sempre delimitadas por aspas duplas.

Strings XML (como valores de atributo) podem ser delimitadas por aspas duplas ou aspas simples.

Espaço em branco é irrelevante

```
{
    "nome" : "João Doe" ,
    "idade" : 30 ,
    "casado" : true
}

{ "nome" : "John Doe" , "idade" : 30 , "casado" : true }
```

Usando JSON você pode definir estruturas arbitrariamente complexas

Estrutura JSON

```
{
    "Livro" :
    {
        "Título" : "Técnicas de Análise" ,
        "Autores" : [ "Dick Grune" , "Ceriel JH Jacobs" ]
    }
}


{
    "Livro" :
    {
        "Título" : "Técnicas de Análise" ,
        "Autores" : [
            { " nome" : "Dick Grune" , "universidade" : " Vrije
Universidade " } ,
            { " nome" : "Ceriel JH Jacobs" , "universidade" : " Vrije
Universidade " }
        ]
    }
}


{
    "Livro" :
    {
        "Título" : "Técnicas de Análise" ,
        "Autores" : [
            {
                "nome" : { " primeiro" : "Dick " , "ultimo" : "Grune " }
,
```

```
                "universidade" : " Vrije Universidade "
            } ,
            {
                "nome" : { " primeiro" : "Ceriel " , " last" : "Jacobs "
} ,
                "universidade" : " Vrije Universidade " }
        ]
    }
}
```

Exercícios

- Escreva um documento JSON que declara uma caderneta de endereços que contém endereços de contato. Cada contato tem um nome, endereço e telefone. O endereço tem o nome da rua, número e CEP.
- Crie o documento JSON que declare um produto, representado por um elemento produto. Produto deve ter um fabricante representado por um elemento fabricante. Utilize detalhes como nome, endereço, etc. como elementos filhos de um elemento fabricante. Cada produto tem os elementos: nome do produto, preço unitário, quantidade e categoria.

Consumindo API

JS síncrono vs assíncrono

Síncrono

Espera a tarefa acabar para continuar com a próxima.

Assíncrono

Move para a próximo 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.

```
JS
```

```
setTimeout(function() {
  console.log('Time 1');
});

setTimeout(function() {
  console.log('Time 2');
}, 100);

console.log('Log 1');

setTimeout(function() {
  console.log('Time 3');
}, 50);

console.log('Log 2');
```

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.

JS assíncrono

Promises

`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.

Promisse

```
const promessa = new Promise(function(resolve, reject) {
  let condicao = true;
  if(condicao) {
    resolve();
  } else {
    reject();
  }
});
```

```
9
10  console.log(promessa); // Promise {<resolved>: undefined}
```

RESOLVE()

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

Resolve

```
1   const promessa = new Promise(function(resolve, reject) {
2     let condicao = true;
3     if(condicao) {
4       resolve('Estou pronto!');
5     } else {
6       reject();
7     }
8   });
9
10  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.

Reject

```
1   const promessa = new Promise(function(resolve, reject) {
2     let condicao = false;
3     if(condicao) {
4       resolve('Estou pronto!');
5     } else {
6       reject(Error('Um erro ocorreu.'));
7     }
8   });
9
10  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.

Then

```
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.

Assinc

```
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.

then.then

```
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.

cathc

```
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.

then

```
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.

Finally

```
 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 reposta é uma array com as respostas de cada promise.

all

```
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.

race

```
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
});
```

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

```
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.

response

```
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.

local

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

json

```
fetch('https://viacep.com.br/ws/36036000/json/')
.then(response => response.json())
.then(cep => {
  console.log(cep.bairro, cep.logradouro);
});
```

.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.

headers

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

fetch('https://viacep.com.br/ws/36036000/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.

status

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

fetch('https://viacep.com.br/ws/36036000/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 reposta.

type

```
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

html

```html
<!DOCTYPE html>
<html>
  <head>
    <title>CEP</title>
    <meta charset="UTF-8" />
  </head>

  <body>
    <form action="" method="get">
      <input type="text" name="cep" id="cep" />
      <button id="buscar">buscar</button>
    </form>
    <pre id="resposta"></pre>

    <script src="src/index.js"></script>
  </body>
</html>
```

js

```js
const buscarBtn = document.getElementById("buscar");
const cep = document.getElementById("cep").value;
const resposta = document.getElementById("resposta");

const handleClick = () => {

};

buscarBtn.addEventListener("click", handleClick);
```

- Utilizando a API https://blockchain.info/ticker retorne no DOM o valor de compra da bitcoin and reais.
- Utilizando a API https://api.chucknorris.io/jokes/random retorne uma piada randomica do chucknorris, toda vez que clicar em próxima

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.

parse

```
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);
```

HTTP

Podemos usar HTTP e HTTPS para realizar nossas requisições

http

```
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.

methos

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

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

- 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.

GET

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

GET

```
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.

POST

```
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.

PUT

```
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))
```

CORS

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

CORS

```
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);
});
```

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.

await

```
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.

async

```
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.

try

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

fetch

```
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.

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');
```

```
13  }
14  iniciarAsync();
```

Exercícios

- Utilizando a API https://pokeapi.co/api/v2/pokemon/ crie um formulário onde o usuário digita o número do pokemon e seu nome é exibido, utilize async-await