



TypeScript

Estendendo o JavaScript

Introdução

- Superset do JavaScript;
 - Conjunto de ferramentas com o objetivo de escrever códigos JavaScript de forma eficiente, segura e dinâmica.
- Compartilha a mesma sintaxe do JavaScript adicionando recursos;
- Foi criado com o objetivo de incluir recursos que não estavam presentes no JavaScript;
- Orientada a objetos;
- Fortemente tipada;
- Não é executado no navegador;
- É preciso compilar o Typescript para o JavaScript;
- Desenvolvida pela Microsoft;

Configuração do arquivo de compilação para JavaScript

- `npx tsc --init`

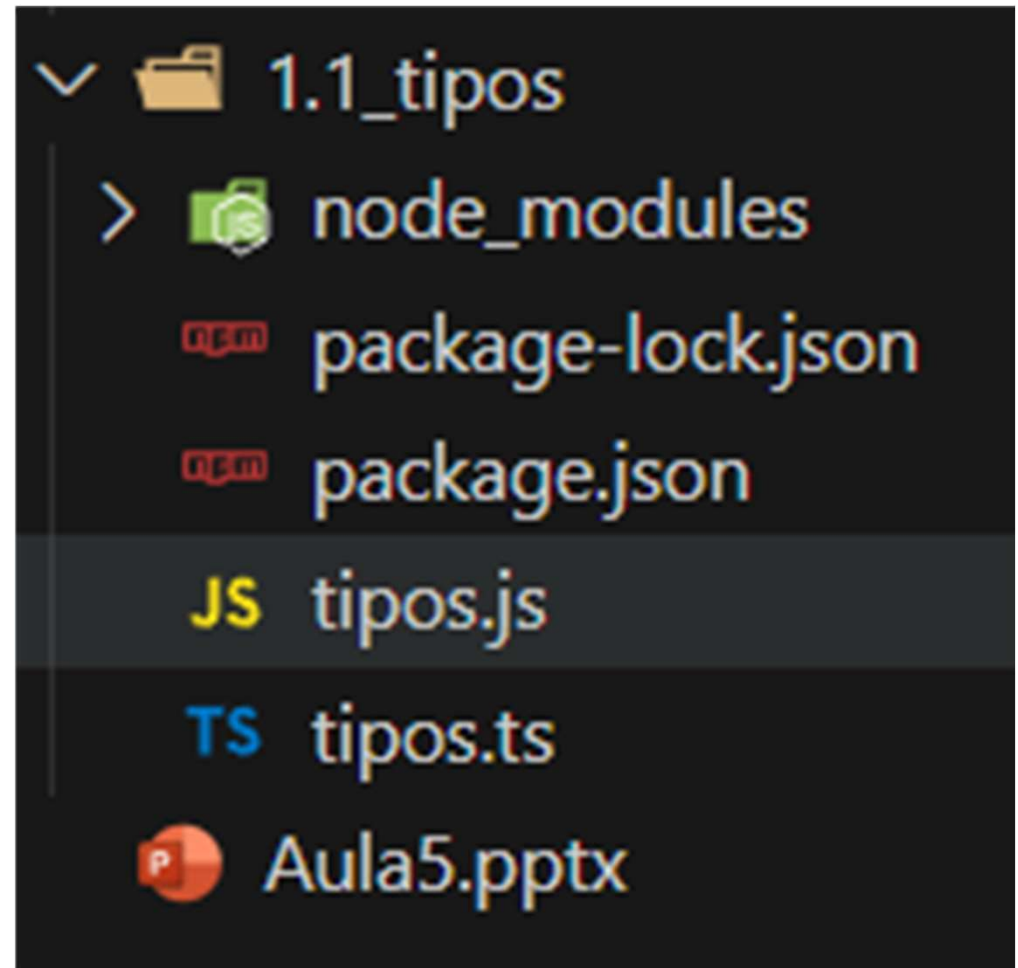
```
// "inlineSources": true, /* Include source code in the sourcemaps instead of just the sourcemap. */
// "emitBOM": true, /* Emit a UTF-8 Byte Order Mark (BOM) in the generated files. */
// "newline": "crlf", /* Set the newline character for emitting files. */
// "stripInternal": true, /* Disable emitting declarations that have '@internal' in their comment. */
// "noEmitHelpers": true, /* Disable generating custom helper functions like `__extends` in production mode. */
// "noEmitOnError": true, /* Disable emitting files if any type checking errors are reported. */
// "preserveConstEnums": true, /* Disable erasing 'const enum' declarations in generated code. */
// "declarationDir": "./", /* Specify the output directory for generated declaration files. */
// "preserveValueImports": true, /* Preserve unused imported values in the JavaScript output if used in a type declaration. */

/* Interop Constraints */
// "isolatedModules": true, /* Ensure that each file can be safely transpiled without relying on other imports. */
// "verbatimModuleSyntax": true, /* Do not transform or elide any imports or exports that are not part of the TypeScript language system. */
// "allowSyntheticDefaultImports": true, /* Allow 'import x from y' when a module does not have a default export. */
"esModuleInterop": true, /* Emit additional JavaScript to ease support for importing CommonJS modules. This enables 'allowSyntheticDefaultImports' for type checking. */
// "preserveSymlinks": true, /* Disable resolving symlinks to their real path. */
"forceConsistentCasingInFileNames": true, /* Ensure that casing is correct in imports. */

/* Type Checking */
"strict": true, /* Enable all strict type-checking options. */
// "noImplicitAny": true, /* Enable error reporting for expressions and declarations with an implied 'any' type. */
// "strictNullChecks": true, /* When type checking, take into account 'null'. */
// "strictFunctionTypes": true, /* When assigning functions, check to ensure parameters and the return values are annotated. */
// "strictBindCallApply": true, /* Check that the arguments for 'bind', 'call' and 'apply' methods are correct. */
// "strictPropertyInitialization": true, /* Check for class properties that are declared but not set in the constructor. */
// "noImplicitThis": true, /* Enable error reporting when 'this' is given without an explicit type. */
// "useUnknownInCatchVariables": true, /* Default catch clause variables as 'unknown'. */
// "alwaysStrict": true, /* Ensure 'use strict' is always emitted. */
// "noUnusedLocals": true, /* Enable error reporting when local variables are not used. */
```

Utilizando o TypeScript

- Node.js
- npm init -y
 - Cria o arquivo package.json
- npm install -save-dev typescript
- npx tsc nomeArquivo.ts
 - ➔ nomeArquivo.js
- node nomeArquivo.js



JavaScript - Tipado dinamicamente

- Tipagem estática: tipo é inferido pela variável e a checagem é feita durante a compilação;
- Tipagem dinâmica: tipo é inferido pelo valor do dado e a checagem é feita durante o tempo de execução;

1.0_staticVsDynamic\typescript.ts

```
//Função javascript padrão
function elevaAoCubo(a){
  return a * a * a;;
}

function somar(a, b){
  return a + b;
}

const resultado = elevaAoCubo('3');

const resultado2 = somar('3', '4');

console.log(resultado);

console.log(resultado2);
```

```
//Função javascript padrão
function elevaAoCubo2(a: number){
  return a * a * a;;
}

function somar2(a: number, b: number){
  return a + b;
}

const resultado_ = elevaAoCubo2(3);

const resultado2_ = somar2(3, 4);

console.log(resultado_);

console.log(resultado2_);
```



Tipos

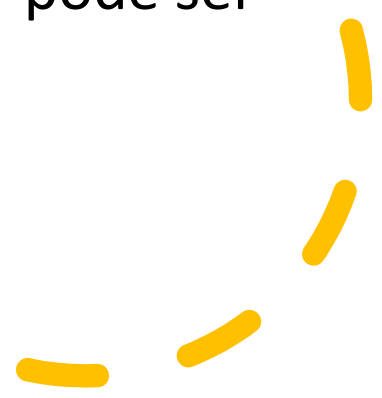
- Primitivos
 - boolean
 - True ou false
 - Number
 - Numeros e números em ponto flutuante
 - String
 - “texto como esse”
 - Bigint
 - Mesma coisa que o tipo number, mas permite números positivos e negativos maiores
 - Symbol
 - São utilizados para criar identificadores universais

```
function printStatusCode(code: string | number) {  
  console.log(`My status code is ${code}.`)  
}  
printStatusCode(404);  
printStatusCode('404');
```

Union Types

- São utilizados quando um valor pode ser mais de um tipo único;

```
//Union Types  
let cpf2: number | string = 12345678900;  
cpf2 = "123.456.789-00";  
//Está ok, pois pode ser number ou string
```



Atribuição de tipo

- Explícito

```
let endereco: string = "Rua dos bobos, 0";
```

- Implícito

```
let endereco = "Rua dos bobos, 0";|
```

- Inferência de tipo é a capacidade de deduzir automaticamente, parcial ou totalmente, o tipo de uma expressão em tempo de compilação

```
let endereco = "Rua dos bobos, 0";  
endereco = 123123;|
```



```

//Tipos primitivos
let cpf:number = 12345678900;
let endereco: string = "Rua dos bobos, 0";
let booleano: boolean = true;
let nulo: null = null;
let indefinido: undefined = undefined;
let numero: number = 3.14;

//Tipos Complexos
//Array
let avioesww2: string[] = ["Spitfire", "Mustang", "Zero"];
//Object
let pessoa: {nome: string, idade: number} = {nome: "João", idade: 20};
//cria a pessoa2 objeto
let pessoa2: {
  nome: string,
  idade: number
};
pessoa2 = {
  nome: "João",
  idade: 20
};
//Array de objetos
let Pessoas: {
  nome: string,
  idade: number
}[] = [pessoa, pessoa2];

console.log(Pessoas);
console.log(Pessoas[0]);
console.log(Pessoas[1]);
console.log(Pessoas[0].nome);
console.log(Pessoas[0].idade);
console.log(Pessoas[1].nome);
console.log(Pessoas[1].idade);

```



```

//Tipos primitivos
var cpf = 12345678900;
var endereco = "Rua dos bobos, 0";
var booleano = true;
var nulo = null;
var indefinido = undefined;
var numero = 3.14;
//Tipos Complexos
//Array
var avioesww2 = ["Spitfire", "Mustang", "Zero"];
//Object
var pessoa = { nome: "João", idade: 20 };
//cria a pessoa2 objeto
var pessoa2;
pessoa2 = {
  nome: "João",
  idade: 20
};
//Array de objetos
var Pessoas = [pessoa, pessoa2];
console.log(Pessoas);
console.log(Pessoas[0]);
console.log(Pessoas[1]);
console.log(Pessoas[0].nome);
console.log(Pessoas[0].idade);
console.log(Pessoas[1].nome);
console.log(Pessoas[1].idade);

```

Classes TypeScript

- TypeScript adiciona tipos e modificadores de visibilidade às classes JavaScript;

```
// Cria uma instância da classe Aviao
let aviao = new Aviao("Boeing 747", 366);
let aviao2 = new Aviao("Boeing 777", 550);
let ww2Aviao = new Aviao("Spitfire", 1);

console.log(aviao. (method) Aviao.getCapacidade(): number
console.log(aviao.getCapacidade());
```

```
class Aviao {
    // Define as propriedades
    private modelo: string;
    private capacidade: number;

    // Define o construtor
    constructor(modelo: string, capacidade: number) {
        this.modelo = modelo;
        this.capacidade = capacidade;
    }

    // Define um método para obter o modelo
    getModelo(): string {
        return this.modelo;
    }

    // Define um método para obter a capacidade
    getCapacidade(): number {
        return this.capacidade;
    }

    // Define um método para exibir informações do avião
    exibirInformacoes(): void {
        console.log(`Modelo: ${this.modelo}, Capacidade: ${this.capacidade}`);
    }
}
```

Type Aliases

- Aliases de tipo permitem definir tipos com um nome personalizado (um Alias);
- Aliases de tipo podem ser usados para primitivos como string ou tipos mais complexos, como objetos e arrays:

```
type CarYear = number
type CarType = string
type CarModel = string
type Car = {
  year: CarYear,
  type: CarType,
  model: CarModel
}

const carYear: CarYear = 2001
const carType: CarType = "Toyota"
const carModel: CarModel = "Corolla"
const car: Car = {
  year: carYear,
  type: carType,
  model: carModel
};
```

Interfaces

- As interfaces são semelhantes aos aliases de tipo (type aliases), exceto que se aplicam apenas a tipos de objetos;
- Classes que implementam interfaces, são obrigadas a ter a estrutura definida na Interface;

Exemplo interface

```
//definição de uma interface
//Tudo que um objeto precisa ter para ser considerado um aviao
interface interfaceAviao {
    modelo: string;
    capacidade: number;
    getModelo(): string;
    getCapacidade(): number;
    exibirInformacoes(): void;
}
```

```
//definição de uma classe que implementa a interface Aviao
class Aviao implements interfaceAviao {
    constructor(public modelo: string, public capacidade: number) {}

    getModelo(): string {
        return this.modelo;
    }

    getCapacidade(): number {
        return this.capacidade;
    }

    exibirInformacoes(): void {
        console.log(`Modelo: ${this.modelo}, Capacidade: ${this.capacidade}`);
    }
}
```

```
//Generics 1
function funcaoGeneric<T>(arg: T): T {
    //Esta função aceita um argumento 'arg' do tipo 'T' e
    //retorna um valor do tipo 'T'.
    //Dentro da função, simplesmente retornamos o argumento.
    //Isso pode não parecer muito útil,
    // mas significa que a função funciona para qualquer tipo 'T'.
    return arg;
}

let saida = funcaoGeneric<string>("Olá");
let saida2 = funcaoGeneric<number>(123);
let saida3 = funcaoGeneric<boolean>(true);
```

Generics

- São uma forma de criar componentes reutilizáveis que podem funcionar em vários tipos, em vez de em um único.

Generics 2

```
//Generics 2
//a classe DataStorage é um tipo genérico
//ela pode ser usada para armazenar qualquer tipo de dados
class DataStorage<T> {
  private data: T[] = [];

  adicionarItem(item: T) {
    this.data.push(item);
  }

  removeItem(item: T) {
    this.data = this.data.filter(i => i !== item);
  }

  getItem() {
    return [...this.data];
  }
}
```

```
//cria um objeto DataStorage para armazenar strings
let texto = new DataStorage<string>();
texto.adicionarItem("oi");
texto.adicionarItem("tudo bom");
console.log(texto.getItem()); // saida: ["oi", "tudo bom"]

//cria um objeto DataStorage para armazenar numeros
let numeros = new DataStorage<number>();
numeros.adicionarItem(10);
numeros.adicionarItem(20);
console.log(numeros.getItem()); // saida: [10, 20]
```


Generics 3

```
//Generics 3

function getTamanho<T>(arg: T[]): number {
    // esta função aceita um argumento 'arg' do tipo 'T[]' e retorna um valor do tipo 'number'.
    return arg.length;
}

let arrayTamanho = getTamanho<number>([1, 2, 3, 4, 5]);
console.log(arrayTamanho); // saída: 5

let stringTamanho = getTamanho<string>(['a', 'b', 'c', 'd', 'e']);
console.log(stringTamanho); // saída: 5
```


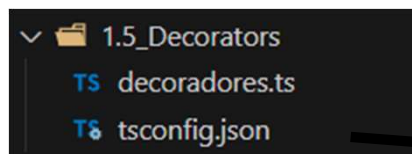

Generics 4

```
//Generics 4
function mergeArrays<T, U>(array1: T[], array2: U[]): (T | U)[] {
    return [...array1, ...array2];
}

let mergedArray = mergeArrays<number, string>([1, 2, 3], ['a', 'b', 'c']);
console.log(mergedArray); // saida: [1, 2, 3, 'a', 'b', 'c']
```

Decoradores (Decorators)

- Um Decorator é um tipo especial de declaração que pode ser anexada a uma declaração de classe, método, propriedade ou parâmetro;
- Decoradores usam a forma @expressão, onde expressão deve ser avaliada como uma função que será chamada em tempo de execução com informações sobre a declaração decorada;
- disponível como um recurso experimental do TypeScript.



```
/* Language and Environment */
"target": "es6",
// "lib": [],
// "jsx": "preserve",
"experimentalDecorators": true,
// "emitDecoratorMetadata": true,
// "jsxFactory": "",
// "jsxFragmentFactory": "",
```

Tipos de decoradores

- Classe: aplicado ao construtor da classe e pode ser usado para observar, modificar ou substituir uma definição de classe;
- Método: aplicado ao descritor de propriedade para o método e pode ser usado para observar, modificar ou substituir uma definição de método;
- Modificadores de acesso (set | get): aplicado ao descritor de propriedade para o modificador de acesso e pode ser usado para observar, modificar ou substituir uma definição de modificadores de acesso;

Tipos de decoradores

- Propriedade: é aplicado à propriedade e pode ser usado para observar, modificar ou substituir uma definição de propriedade;
- Parâmetro: aplicado à função para um construtor de classe ou declaração de método;

Por que usar Decorators?

- Organização de código: podem ajudar a manter o código organizado, permitindo que seja anexado metadados diretamente a classes e membros, em vez de ter que controlá-los separadamente;
- Reutilização de código: podem encapsular comportamento comum e aplicá-lo a múltiplas classes ou membros, promovendo a reutilização de código;
- Separação de preocupações (Separation of concerns): podem separar as preocupações, permitindo que se modifique ou aumente classes e membros sem alterar seu código-fonte;
- Programação Orientada a Aspectos: podem ser usados para implementar técnicas de programação orientada a aspectos, como registro, cache ou gerenciamento de transações.

Decorador de classe @logConstructor

```
function logConstructor(constructor: Function) {  
    console.log('A classe foi criada.',  
        constructor);  
}  
  
@logConstructor  
class Aviao {  
    constructor(public modelo: string, public  
        capacidade: number) {}  
}  
  
let aviao = new Aviao('Boeing 777', 500);
```

```
logConstructor.js  
A classe foi criada. [Function: Aviao]
```

Decorator Factory

- Personalizar como um decorador é aplicado a uma declaração
- Um Decorator Factory é uma função que retorna a expressão que será chamada pelo decorador em tempo de execução.

```
function criarElementoDOM(tag: string, content: string) {  
  return function(constructor: Function) {  
    let elemento = document.createElement(tag);  
    elemento.textContent = `${content} ${constructor.name}`;  
    document.body.appendChild(elemento);  
  }  
}  
  
@criarElementoDOM('h1', 'Definindo a classe:')  
class Planes {  
  constructor(public model: string, public capacity: number) {}  
}  
  
let plane = new Planes('Boeing 777', 500);  
let plane2 = new Planes('pilatus', 2);  
let plane3 = new Planes('Cirrus', 4);
```

Property Decorator

- Declarado pouco antes de uma declaração de propriedade;
- A expressão para o decorador de propriedade será chamada como uma função em tempo de execução com dois argumentos:
 - A função construtora da classe para um membro estático ou o protótipo da classe para um membro de instância.
 - O nome do membro.

Property Decorator

```
let carro1 = new Car('Ford', 'Super Deluxe', 1941);
let carro2 = new Car('Chevrolet', 'Special Deluxe', 1940);
carro1.modelo = 'Mustang';
```

```
• Property Decorator!
Target: undefined
PropertyKey: {
  kind: 'field',
  name: 'fabricante',
  static: false,
  private: false,
  access: { has: [Function: has], get: [Function: get], set: [Function: set] },
  metadata: undefined,
  addInitializer: [Function (anonymous)]
}
```

1.5_Decorators\propertyDecorator.ts

```
function propertyDec(target: any, propertyKey: string) {
  console.log('Property Decorator!');
  console.log('Target: ', target);
  console.log('PropertyKey: ', propertyKey);
}

class Car {
  @propertyDec
  public fabricante: string;
  private _modelo: string;
  public year: number;
  constructor(fabricante: string, model: string, year: number) {
    this.fabricante = fabricante;
    this._modelo = model;
    this.year = year;
  }

  displayCarInfo() {
    console.log(`Fabricante: ${this.fabricante},
                Model: ${this.modelo},
                Year: ${this.year}`);
  }

  get modelo() {
    return this._modelo;
  }

  set modelo(novoModelo: string) {
    if (!novoModelo) {
      throw new Error('Modelo inválido.');
```

Decorator de modificadores de acesso

- Declarado logo antes de uma declaração de modificador de acesso;
- É aplicado ao descritor de propriedade do acessador e pode ser usado para observar, modificar ou substituir as definições de um modificador de acesso;
- Chama como uma função em tempo de execução com três argumentos:
 - a função construtora da classe para um membro estático ou o protótipo da classe para um membro de instância;
 - O nome do membro;
 - O descritor de propriedade do membro.

Decorator de modificadores de acesso

```
function LogAcessor(target: any, nomePropriedade: string, descriptor: PropertyDescriptor) {
    console.log('LogAcessor decorator chamado!');
    console.log('Target: ', target);
    console.log('Nome da propriedade: ', nomePropriedade);
    console.log('Descritor: ', descriptor);
    return descriptor;
}

class Carro {
    private _modelo: string;

    constructor(modelo: string) {
        this._modelo = modelo;
    }

    @LogAcessor
    get modelo() {
        return this._modelo;
    }

    set modelo(valor: string) {
        if (!valor) {
            throw new Error('Modelo inválido.');
        }
        this._modelo = valor;
    }
}
```

Target: [Function: get]
Nome da propriedade: {
 kind: 'getter',
 name: 'modelo',
 static: false,
 private: false,
 access: { has: [Function: has], get: [Function: get] },
 metadata: undefined,
 addInitializer: [Function (anonymous)]
}
Descritor: undefined

Method Decorator

- Declarado logo antes da declaração de um método;
- É aplicado ao descritor de propriedade do método e pode ser usado para observar, modificar ou substituir uma definição de método;
- A expressão para o decorador do método será chamada como uma função em tempo de execução, com os três argumentos:
 - A função construtora da classe para um membro estático ou o protótipo da classe para um membro de instância;
 - O nome do membro;
 - O descritor de propriedade do membro.

Method Decorator

```
function metodoDecorator(target: any, propertyKey: string, descriptor: PropertyDescriptor) {  
    console.log('Método decorator!');  
    console.log('Target: ', target);  
    console.log('PropertyKey: ', propertyKey);  
    console.log('Descriptor: ', descriptor);  
}  
  
class Carros {  
    constructor(public modelo: string) {}  
  
    @metodoDecorator  
    drive(speed: number) {  
        console.log(`${this.modelo} está dirigiendo a ${speed} km/h`);  
    }  
}
```

```
• Método decorator!  
Target: [Function (anonymous)]  
PropertyKey: {  
  kind: 'method',  
  name: 'drive',  
  static: false,  
  private: false,  
  access: { has: [Function: has], get: [Function: get] },  
  metadata: undefined,  
  addInitializer: [Function (anonymous)]  
}  
Descriptor: undefined
```

1.5_Decorators\methodDecorator.ts

Parameter Decorators

- Declarado logo antes da declaração do parâmetro;
- É aplicado à função para um construtor de classe ou declaração de método;
- Não pode ser usado em um arquivo de declaração, em uma sobrecarga ou em qualquer outro contexto de ambiente (como em uma classe de declaração);
- A expressão para o decorador de parâmetros será chamada como uma função em tempo de execução com três argumentos a seguir:
 - A função construtora da classe para um membro estático ou o protótipo da classe para um membro de instância;
 - O nome do membro;
 - O índice ordinal do parâmetro na lista de parâmetros da função

Parameter Decorators

```
function parametroDecorator(target: any, propertyKey: string, parameterIndex: number) {  
    console.log('Parametro Decorator Chamado');  
    console.log('Target: ', target);  
    console.log('PropertyKey: ', propertyKey);  
    console.log('ParameterIndex: ', parameterIndex);  
}  
  
class PC {  
    public modelo: string;  
    constructor(modelo: string) {  
        this.modelo = modelo;  
        console.log('PC criado.');    }  
  
    displayModelo(@parametroDecorator modelo: string) {  
        console.log(`Modelo: ${modelo}`);  
    }  
}
```

Promise

- Objeto que representa uma operação que ainda não foi concluída, mas é esperada no futuro;
- Uma promessa pode estar em um dos três estados:
 - **Pendente (pending)**: O resultado ainda não foi determinado porque a operação assíncrona ainda está em andamento;
 - **Cumprida (fulfilled)**: A operação assíncrona foi concluída e o resultado da promessa é um valor;
 - **Rejeitado**: A operação assíncrona falhou e o resultado da promessa é um erro.


```
'use strict';

// ES5
var myPromise = new Promise(function (resolve, reject) {
    // Your Code which you are unsure about execution time duration.

    // Call resolve() method at the end of successful execution.

    // Call reject() method for your failure case.

    // Catch method will be called automatically, if any error
    occurs.
});

// ES6, Example with Arrow Functions
var myPromise = new Promise((resolve, reject) => { ... })
```

Promise

Promise

- Os dois argumentos (resolve e reject) são pré-definidos por JavaScript;
- Eles não são criados;
- A função executora chama um deles quando estiver pronta;
- Muitas vezes não é preciso chamar a função reject;

Encadeamento de Promise

- Uma necessidade comum é executar duas ou mais operações assíncronas consecutivas;
- Cada operação subsequente começa quando a operação anterior é bem sucedida, com o resultado do passo anterior;
- Isto é alcançado criando uma *cadeia de promises*.
 - função **then** retorna uma nova **promise**, diferente da original
- Sempre retorne um resultado, de outra forma as callbacks não vão capturar o resultado da promise anterior.

Promise

- Funções assíncronas retornam uma promise;

```
// Cria uma nova Promise: Estado: pending
let buscaDados = new Promise((resolve, reject) => {
  // Busca dados da api do microclima da CEPLAC
  //simulando uma requisição assíncrona
  fetch('https://thingspeak.com/channels/1898908/field/2.json')
    .then(response => {
      //https://developer.mozilla.org/pt-BR/docs/Web/API/Response
      // se o response for ok, resolve the Promise com os dados do response convertidos
      if (response.ok) {
        //tipo unknown: Estado: resolved
        return response.json();
      } else {
        // Se o response não esta ok, rejeita a Promise com o status do response
        throw new Error(response.statusText);
      }
    })
    //tipo unknown: Estado: resolved
    .then(data => resolve(data))
    //Estado: rejected
    .catch(error => reject(error));
});
```

```
async function myFunction() {  
  return "Hello";  
}
```



```
function myFunction() {  
  return Promise.resolve("Hello");  
}
```

```
myFunction().then(  
  function(value) { /* code if successful */ },  
  function(error) { /* code if some error */ }  
);
```

Async Await

- O JavaScript moderno adicionou um modo de lidar com call-backs de uma maneira elegante adicionando uma API baseada em Promises;
- Permite tratar código assíncrono como se fosse síncrono;
- A palavra-chave **async** antes de uma função faz com que a função retorne uma promise;

Palavra-chave await

- Só pode ser usada dentro de uma função assíncrona;
- Faz com que a função pause a execução e espere por uma promise resolvida antes de continuar;

```
let value = await promise;
```

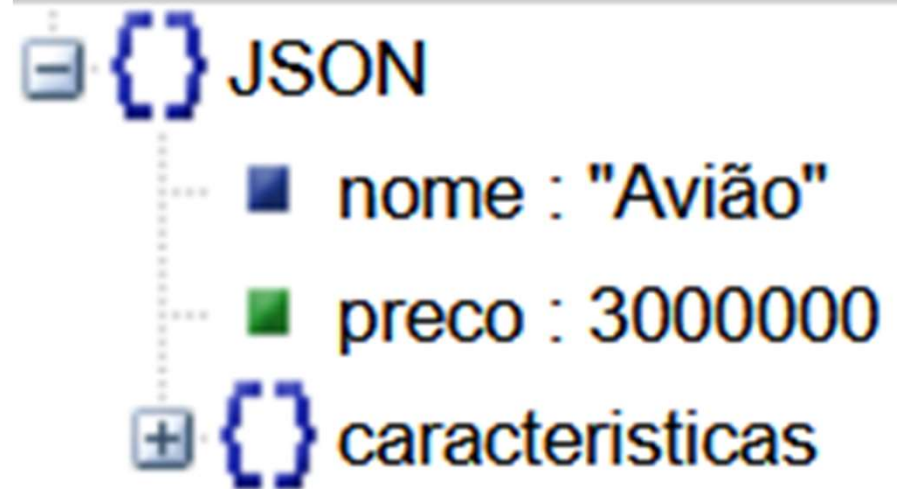
JSON – JavaScript Object Notation

- Maneira de formatar os dados que podem ser enviados pela Internet;
- Baseado em texto padrão para representar dados estruturados com base na sintaxe do objeto JavaScript;
- JavaScript fornece um objeto JSON global que possui métodos disponíveis para conversão entre os dois;
- Converter string em objeto = parse;
- Converter objeto em string para ser transmitida na rede = *stringification*;

JSON – JavaScript Object Notation

- <https://jsonviewer.stack.hu/>

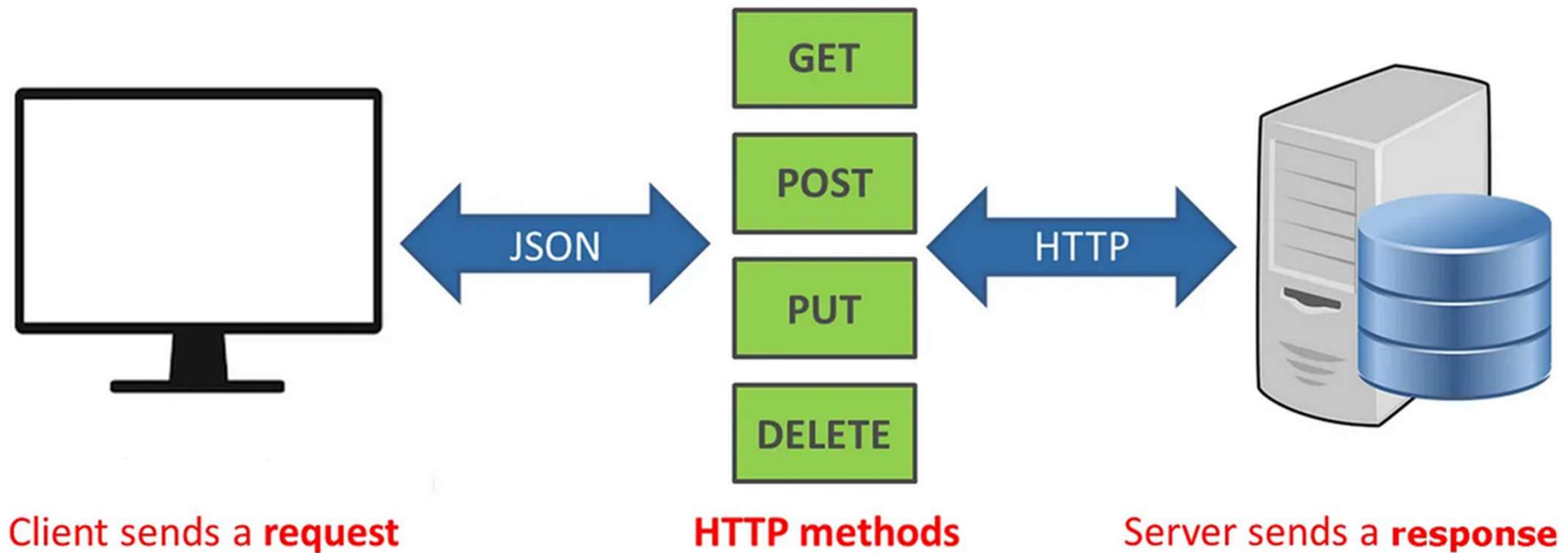
```
{  
  "nome": "Avião",  
  "preço": 3000000,  
  "características": {  
    "motor": "5000cc",  
    "portas": 4  
  }  
}
```



Application Programming Interface (API)

- Conjunto de características e regras existentes em uma aplicação que possibilitam interações com essa através de um software;
- A API pode ser entendida como um simples contrato entre a aplicação que a fornece e outros itens, como outros componentes do software, ou software de terceiros;
- As APIs permitem que os desenvolvedores evitem recriar recursos de aplicativos que já existem.

Application Programming Interface (API)



API Endpoints

- Um endpoint de API é um URL que atua como ponto de contato entre um cliente de API e um servidor de API;
 - BaseURL/Endpoint
- Os clientes da API enviam solicitações aos endpoints da API para acessar a funcionalidade e os dados da API;
 - <https://api.wheretheiss.at/v1/satellites> - Este endpoint retorna uma lista de satélites sobre os quais esta API possui informações;
 - <https://restcountries.com/v3.1/name/brasil> - Este endpoint busca um país pelo nome;

API Endpoints

The screenshot shows a REST client interface with the following components:

- URL Bar:** Displays the URL `https://api.wheretheiss.at/v1/satellites` with a `GET` method selected. A `Send` button is to the right.
- Tabs:** Includes `Params`, `Authorization`, `Headers (6)`, `Body`, `Pre-request Script`, `Tests`, `Settings`, and `Cookies`. The `Params` tab is active.
- Query Params Table:**

	Key	Value	Description	...	Bulk Edit
	Key	Value	Description		
- Body Section:** Includes tabs for `Body`, `Cookies`, `Headers (13)`, and `Test Results`. The `Body` tab is active, showing a JSON response in `Pretty` format:

```
{  "name": "iss",  "id": 25544}
```

. The status bar indicates `Status: 200 OK`, `Time: 786 ms`, and `Size: 442 B`. A `Save as example` button is also present.

Parâmetros de consulta

- Os parâmetros de consulta da API podem ser definidos como pares de chave-valor opcionais que aparecem após o ponto de interrogação no URL;
- São extensões de URL utilizadas para ajudar a determinar conteúdo ou ação específica com base nos dados que estão sendo entregues.

<https://v2.jokeapi.dev/joke/Any?blacklistFlags=religious&type=single>

Parâmetros de consulta

The screenshot shows a REST client interface with the following components:

- URL Bar:** `v2.jokeapi.dev/joke/Any?type=single`
- Method:** `GET`
- Params Tab:** Contains a table with query parameters.
- Body Tab:** Shows the JSON response in 'Pretty' format.

Params Table:

	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	type	single			
	Key	Value	Description		

Body (JSON Response):

```
1  {
2    "error": false,
3    "category": "Programming",
4    "type": "single",
5    "joke": "Two SQL tables sit at the bar. A query approaches and asks \"Can I join you?\"",
6    "flags": {
7      "nsfw": false,
8      "religious": false,
9      "political": false,
10     "racist": false,
11     "sexist": false,
12     "explicit": false
13   },
14   "id": 221,
15   "safe": true,
16 }
```

Status Bar: 200 OK, 281 ms, 1.76 KB

Path parameter

- O Path parameter (parâmetros de caminho) são elementos variáveis de uma URL que é usado para identificar um recurso específico em uma coleção;
- Isso permite enviar uma chamada de API que se refere a um recurso específico, em vez de a uma coleção de recursos;
- Exemplo: Se é preciso recuperar informações sobre um usuário específico, em vez de todos os usuários de uma conta, um path parameter permitirá que isso seja feito.

<https://api.wheretheiss.at/v1/satellites/25544>

Path parameter

The screenshot shows a REST client interface with the following details:

- URL:** `https://api.wheretheiss.at/v1/satellites/25544`
- Method:** `GET`
- Query Params:** A table with 4 columns: Key, Value, Description, and Bulk Edit. It contains one row with 'Key' and 'Value'.
- Body:** A JSON object with the following fields:

```
1  {
2    "name": "iss",
3    "id": 25544,
4    "latitude": 41.446609938551,
5    "longitude": -100.62615475464,
6    "altitude": 420.75674884048,
7    "velocity": 27590.261478907,
8    "visibility": "eclipsed",
9    "footprint": 4511.3114568718,
10   "timestamp": 1701395722,
11   "daynum": 2460279.5801157,
12   "solar_lat": -21.736556652098,
13   "solar_lon": 148.35168037551,
14   "units": "kilometers"
15 }
```
- Status:** 200 OK, 472 ms, 729 B
- Actions:** Save, Send, Bulk Edit, Save as example

Lista de APIs públicas

- Informações sobre países: <https://restcountries.com/>
- Tempo: <https://openweathermap.org/api>
- Imagens de cachorros: <https://thedogapi.com/>
- Piadas: <https://jokeapi.dev/>
- Nasa: <https://api.nasa.gov/>
- Informações de voos em tempo real: <https://aviationstack.com/>
- Mais sobre aviação: <https://www.adsbexchange.com/data/>
- StarWars: <https://pipedream.com/apps/swapi>

API Architecture Styles

- REST;
- SOAP;
- GraphQL;
- gRPC;
- WebSocket;

REST API (*Representational State Transfer*)

- Em uma arquitetura cliente-servidor, o cliente Front end se comunica com o servidor Back end;
- No lado do servidor, diversos serviços são expostos para serem acessados pelo cliente, solicitações de dados do cliente em forma de API com protocolo HTTP;
- REST API permite desenvolver qualquer tipo de aplicativo da web com todas as operações CRUD (criar, recuperar, atualizar, excluir);
- As diretrizes REST sugerem o uso de um método HTTP específico em um tipo específico de chamada feita ao servidor.

REST API (*Representational State Transfer*)

- Quando um cliente faz uma solicitação usando uma API RESTful, essa API transfere uma representação do estado do recurso ao solicitante (endpoint);
- Essa informação / representação) é entregue via HTTP utilizando um dos vários formatos possíveis: Javascript Object Notation (JSON), XML, texto, etc;

Restrições arquitetônicas

- REST define **6 restrições arquitetônicas** que tornam qualquer serviço web – uma API verdadeiramente RESTful.
 - Interface uniforme
 - Cliente - Servidor
 - Sem estado
 - Armazenável em cache
 - Sistema em camadas
 - Código sob demanda (opcional)

Protocolo HTTP

- Para a API ser RESTFull ela deve ser capaz de lidar com todos esses métodos:
 - GET: para recuperar/buscar dados;
 - POST: para criar ou atualizar dados;
 - PUT: para atualização de dados;
 - DELETE: Para exclusão de dados;
 - PATCH: Atualização de dados;

1.10_restfull\app.ts

<https://jasonwatmore.com/post/2021/09/06/fetch-http-get-request-examples>

Integração de Bibliotecas JavaScript

- Importar as bibliotecas do JavaScript;
 - Instalar a biblioteca através do NPM;
 - `npm install <nomeDaBiblioteca>`
 - Importar no seu projeto;
 - **`Import * as _ from 'jquery';`**
- Se a biblioteca não tiver definições de tipo TypeScript:
 - Necessário instalá-las separadamente usando DefinitelyTyped:
 - `npm install @types/library-name`
 - **`npm install @types/jquery`**
 - **`npm install @types/node`**

Exercicio 1

- Crie um mapa interativo utilizando o biblioteca leaflet
 - <https://leafletjs.com/>
- Abra o arquivo app.js na pasta 1.9_Exercicio1
- Carregue uma lista de paises utilizando API restcountries.com e alimente o select “paises” utilizando o DOM em tempo de execução;
- Quando o pais for selecionado no select, associe a um evento “change” e crie uma função que pega o nome do pais e mostra no mapa.
- Index.html já mostra o mapa posicionado na localidade Itabuna/Ilhéus.

1.9_Exercicio1

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find

Referências

- TypeScript Introduction, 2023 - https://www.w3schools.com/typescript/typescript_intro.php
- TypeScript, 2023 - <https://www.typescriptlang.org/>
- Decorators, 2023 - <https://www.typescriptlang.org/docs/handbook/decorators.html>
- REST API introduction - <https://medium.com/@hemilturakhia/rest-api-introduction-eda7dd2c97e8>
- Test your front-end Against a real API - <https://reqres.in/>
- REST Architectural Constraints, 2023 <https://restfulapi.net/rest-architectural-constraints/>
- JavaScript Async, 2023 - https://www.w3schools.com/js/js_async.asp