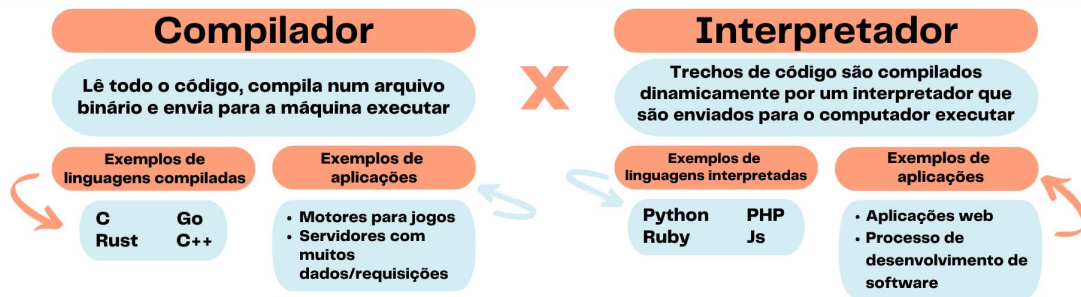


Aula 5 - JS, Node.js, Express e REST

José Glauber - UFCG
2025.1

Javascript



Javascript

LINGUAGENS DE PROGRAMAÇÃO

TIPADAS VS NÃO TIPADAS

As linguagens estaticamente tipadas requerem declarar as **variáveis com seu tipo de dados**.

```
int a = 1;  
String b = "hola";  
bool c = true;
```

Mais verboso, mas menos sujeito a erros de sintaxe (o compilador os detecta).

Você pode saber que tipo de dados uma função retorna.

```
func saludar() string {
```

Não é possível alterar o tipo de dados após a variável ser declarada.

As linguagens dinamicamente tipadas declaram **variáveis sem a necessidade** definir o tipo de dado (o intérprete infere o tipo).

```
let a = 1  
let b = "hola"  
let c = true
```

Código mais legível e curva de aprendizado mais fácil.

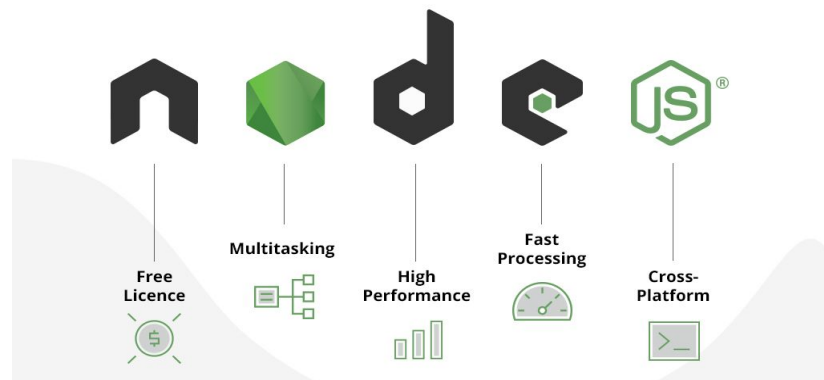
Não sabe qual tipo de dados uma função retorna.

```
def saludar():
```


Você pode alterar o tipo de dados após a variável ser declarada.

Dominina cualquier lenguaje desde cero en:
ed.team/programacion

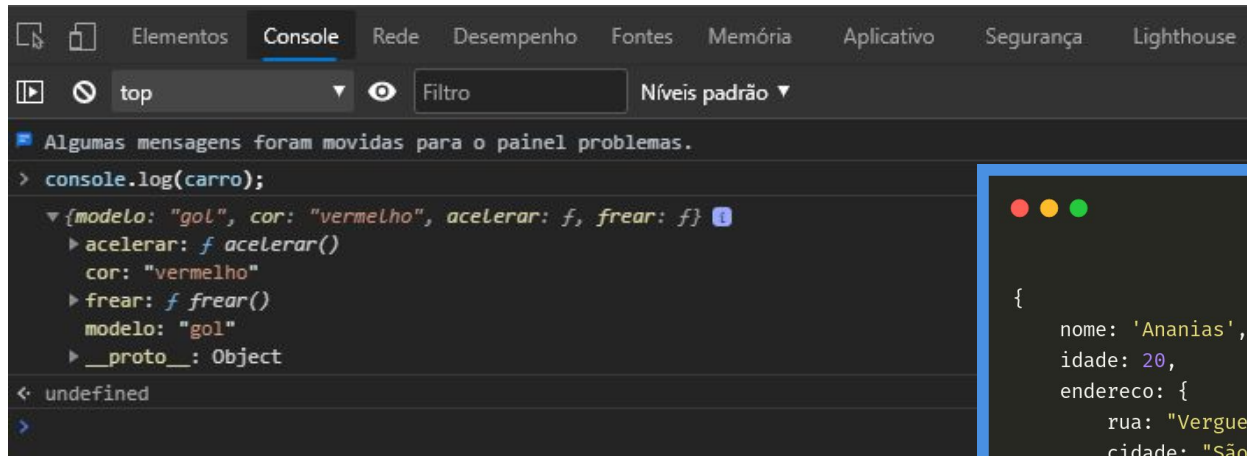
EDteam



Javascript

 <p>Beetle 1954</p>	Propriedades	Métodos
	car.name = Beetle	car.start()
	car.type = Sedan	car.drive()
	car.weight = 840kg	car.brake()
	car.color = Pink	car.stop()

Javascript



```
{
  nome: 'Ananias',
  idade: 20,
  endereco: {
    rua: 'Vergueiro',
    cidade: 'São Paulo'
  },
  telefones: ['91234-5678', '98765-4321'],
  "calcula-ano-nascimento": function(){
    //obtem ano atual
    var anoAtual = new Date().getFullYear();
    return anoAtual - this.idade;
  }
}
```

Javascript

JSON e Objeto são a mesma coisa?

Um JSON tem algumas restrições:

- não pode ter funções
- não pode ter comentários

```
{  
  "nome": "Ananias",  
  "idade": 20,  
  "endereço": {  
    "rua": "Vergueiro",  
    "cidade": "São Paulo"  
  },  
  "telefones": ["91234-5678", "98765-4321"]  
}
```

Conceitos básicos

```
function soma(a, b) {  
  return a + b;  
}
```

- pode ser usada antes de ser definida
- ideal para funções disponíveis em todo o escopo

```
const saudacao = function (nome) {  
  return `Olá, ${nome}!`;  
};
```

- só pode ser usada após ser definida
- pode ser atribuída a variáveis, o que permite manipular funções como objetos

```
const multiplicar = (a, b) => a * b;
```

- Sintaxe mais concisa
- Não tem seu próprio this ou arguments. Herda do contexto criada.

Diferenças?

O que o ES6+ trouxe de mais importante?

- Classes
 - sintaxe para criação de objetos e herança baseada em protótipos
- Modules
 - permitiu a exportação e importação de código entre arquivos;

```
1 class Person {
2   constructor(name, age) {
3     this.name = name;
4     this.age = age;
5   }
6   ..
7   message() {
8     return `My name is ${this.name} and i have ${this.age} old.`;
9   }
10 }
11
12 const p1 = new Person("Alice", 25);
13 console.log(p1.message());
```

```
1 export const name = "Alice";
2
3 export function message() {
4   console.log("Hello world!");
5 }
6
7 import { name, message } from './modulo.js';
8 message();
```


Classes

- Forma de implementar POO em Javascript;

```
1 class Person {
2   constructor(name, age) {
3     this.name = name;
4     this.age = age;
5   }
6
7   message() {
8     return `My name is ${this.name} and i have ${this.age} old.`;
9   }
10 }
11
12 const p1 = new Person("Alice", 25);
13 console.log(p1.message());
```

- Introduziu os conceitos de herança

```
1 class Student extends Person {
2   constructor(name, age, course) {
3     super(name, age);
4     this.course = course;
5   }
6
7   StudentInfo() {
8     return `${this.message()}. I'm studying ${this.course}`;
9   }
10 }
11
12 const Student1 = new Student("Bob", 22, "Prog web");
13 console.log(Student1.StudentInfo());
```

Qual foi o verdadeiro ganho das Promises em Javascript?

- Código assíncrono era frequentemente gerenciado por meio de callbacks;



Complexidade do código e a difícil manutenção

```
1 doSomething(function (result) {  
2   doSomethingElse(  
3     result,  
4     function (newResult) {  
5       doThirdThing(  
6         newResult,  
7         function (finalResult) {  
8           console.log(finalResult);  
9         },  
10        failureCallback  
11      );  
12    },  
13    failureCallback  
14  );  
15 }, failureCallback);
```

```
1 doSomething()  
2   .then((result) => doSomethingElse(result))  
3   .then((newResult) => doThirdThing(newResult))  
4   .then((finalResult) => console.log(finalResult))  
5   .catch((error) => console.error(error));
```

- Melhor tratamento de erros
- Encadeamento mais simples;
- Operações paralelas;
- Compatibilidade com async/await

```

1  const url = "https://api.exemplo.com/data";
2
3  function searchData(url) {
4    return new Promise((resolve, reject) => {
5      fetch(url)
6        .then((response) => {
7          if (!response.ok) {
8            throw new Error("Network error");
9          }
10         return response.json();
11       })
12       .then((data) => resolve(data))
13       .catch((error) => reject(error));
14    });
15  }
16
17  searchData(url)
18    .then((data) => {
19      console.log("response data:", data);
20    })
21    .catch((error) => {
22      console.error("Error to get data:", error);
23    });

```

Promise.all
 Promise.race
 Promise.allSettled
 Promise.any

Async/await

- Introduzida no ES8;
- Permitiu que código assíncrono fosse escrito de maneira síncrona;

```
1  async function fetchData() {  
2    try {  
3      const response1 = await fetch("url1");  
4      const data1 = await response1.json();  
5  
6      const response2 = await fetch("url2");  
7      const data2 = await response2.json();  
8  
9      console.log("Data:", data1, data2);  
10   } catch (error) {  
11     console.error("Erro to get data info:", error);  
12   }  
13 }  
14  
15 fetchData();
```

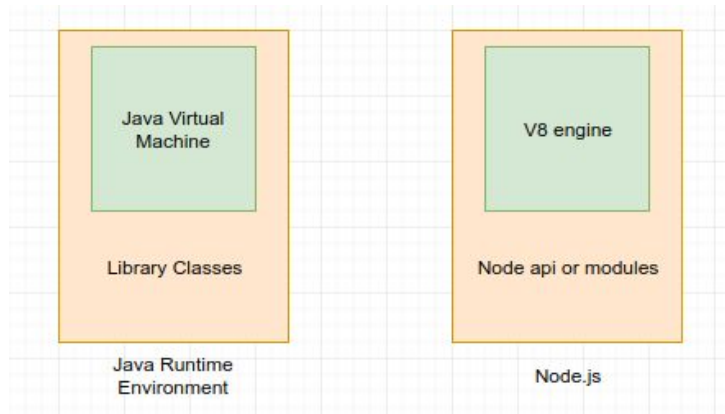
Async/await

```
fetch('url')  
  .then(response =>  
response.json())  
  .then(data => console.log(data))  
  .catch(err => console.error(err));
```

```
async function fetchData() {  
  try {  
    const response = await fetch('url');  
    const data = await response.json();  
    console.log(data);  
  } catch (err) {  
    console.error(err);  
  }  
}  
fetchData();
```

Node.js

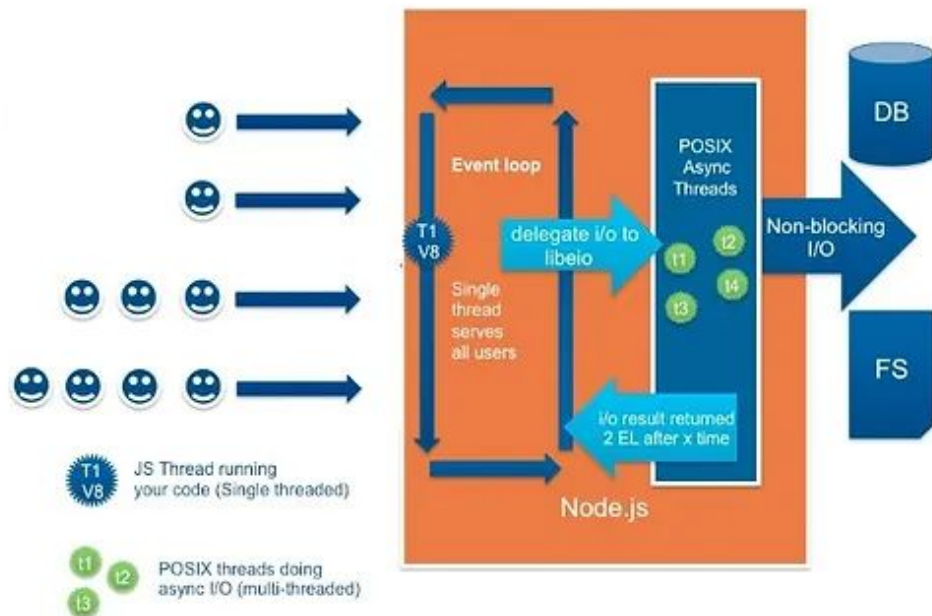
- Ambiente de execução JavaScript assíncrono baseado no V8 do Google Chrome;
- Sair do navegador e executar em máquina próprias, como uma aplicação autônoma;



- V8 engine transmite código baixo nível para o computador, que não tem a necessidade de interpretá-lo primeiramente;

Node.js

Node.js - Single Thread, Event



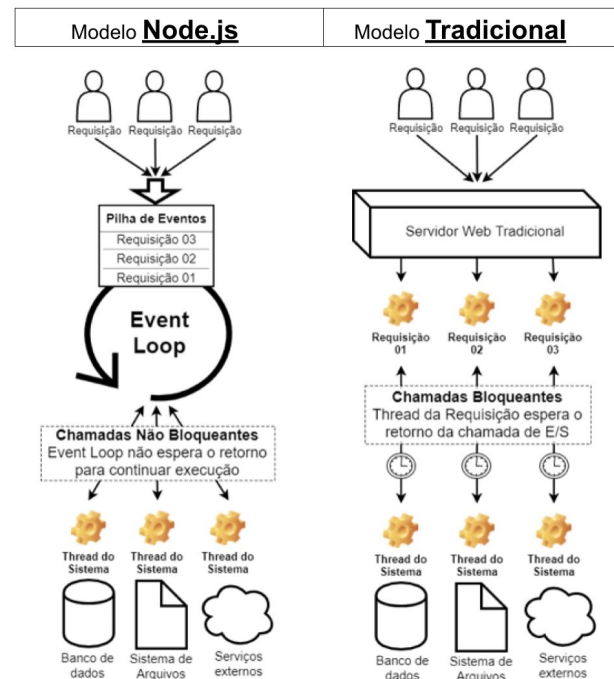
Node.js - NPM (node package manager)

- Maior ecossistema de bibliotecas de código aberto do mundo;
- Facilita a instalação e gerenciamento de pacotes;

```
{
  "name": "my_project",
  "version": "1.0.0",
  "description": "A little description of my project",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified.\\\" && exit 1"
  },
  "author": "Glauber",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```


Node.js - vantagens

- Alta performance devido ao V8 e ao modelo non-blocking I/O
- Escalabilidade
- Linguagem server-side e client-side;



Express

- Framework web para Node.js e fornece um conjunto robusto de recursos;
- Abstrai funcionalidades como:
 - roteamento e middleware

```
app.get("/", (req, res) => {  
  res.send("Página Inicial");  
});  
  
app.post("/submit", (req, res) => {  
  res.send("Dados enviados");  
});  
  
app.put("/update/:id", (req, res) => {  
  res.send(`Atualizar item com ID: ${req.params.id}`);  
});  
  
app.delete("/delete/:id", (req, res) => {  
  res.send(`Deletar item com ID: ${req.params.id}`);  
});
```

```
1  const express = require('express');  
2  const app = express();  
3  
4  // Application Middleware  
5  app.use((req, res, next) => {  
6    console.log(`${req.method} ${req.url}`);  
7    next();  
8  });  
9  
10 // Route Middleware  
11 app.get('/usuario/:id', (req, res, next) => {  
12   const userId = req.params.id;  
13   if (isNaN(userId)) {  
14     res.status(400).send('UserId invalid');  
15   } else {  
16     next();  
17   }  
18 }, (req, res) => {  
19   const userId = req.params.id;  
20   res.send(`Usuário ID: ${userId}`);  
21 });  
22  
23 // Manipulate error Middleware  
24 app.use((err, req, res, next) => {  
25   console.error(err.stack);  
26   res.status(500).send('Something wrong!');  
27 });
```

Express

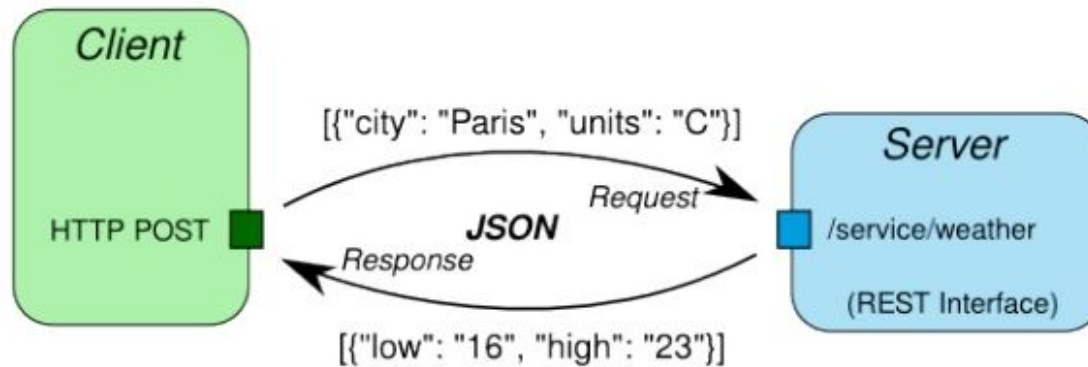
- Trabalhando com dados:
 - O Express facilita a manipulação de dados em requisições HTTP usando middlewares como body-parser

```
1  const bodyParser = require("body-parser");
2  app.use(bodyParser.json());
3
4  app.post("/users", (req, res) => {
5    const user = req.body;
6    res.send(`User ${user.name} was created!`);
7  });
8
```

Arquitetura REST

Arquitetura REST

JSON / REST / HTTP



Lab práctico

Link:

<https://docs.google.com/document/d/1t0ZPb069tE1K6eXV4fcBpkek7KpIPaSi/edit?usp=sharing&ouid=113884329448078609087&rtpof=true&sd=true>

Referências

- FLANAGAN, David. JavaScript: o guia definitivo. Bookman Editora, 2012.
- <https://www.digitalocean.com/community/tutorials/node-js-architecture-single-threaded-event-loop>
- https://developer.mozilla.org/en-US/docs/Learn/Server-side/Node_server_without_framework
- <https://docs.npmjs.com/>