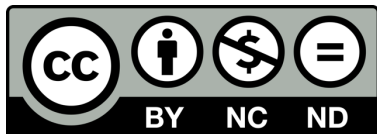

Desenvolvimento *full-stack* com JavaScript

Profa. Dra. Bruna C. Rodrigues da Cunha

bruna.rodrigues@ifsp.edu.br



HTML, CSS, JavaScript...

1



Programação Front-end x Back-end

2

- **Front end**: interface de interação com os usuários; código interpretado pelo cliente (*browser*)
- **Back-end**: programa executado no servidor (e.g., regras de negócio, acesso a banco de dados)

Front-end



- HTML
- CSS
- JavaScript

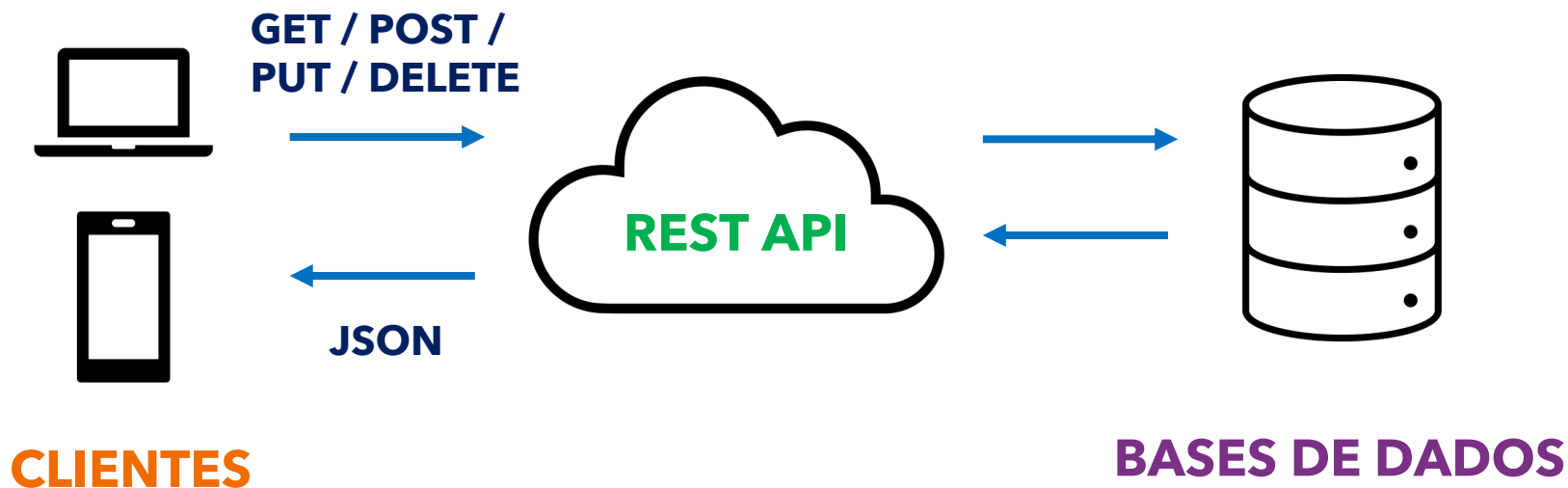
Back-end



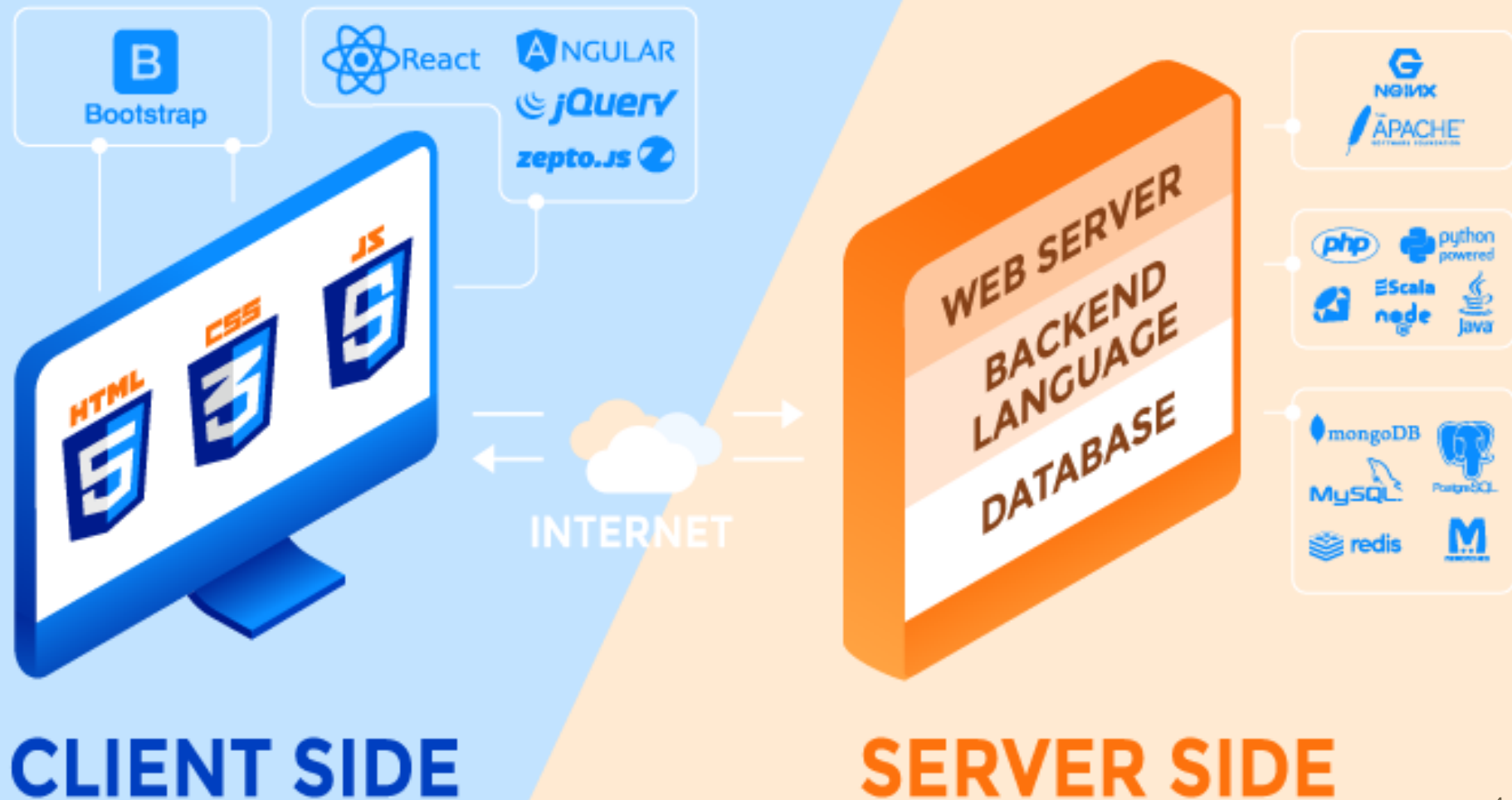
- JavaScript
- Java
- Python
- C#
- Ruby

REST

3

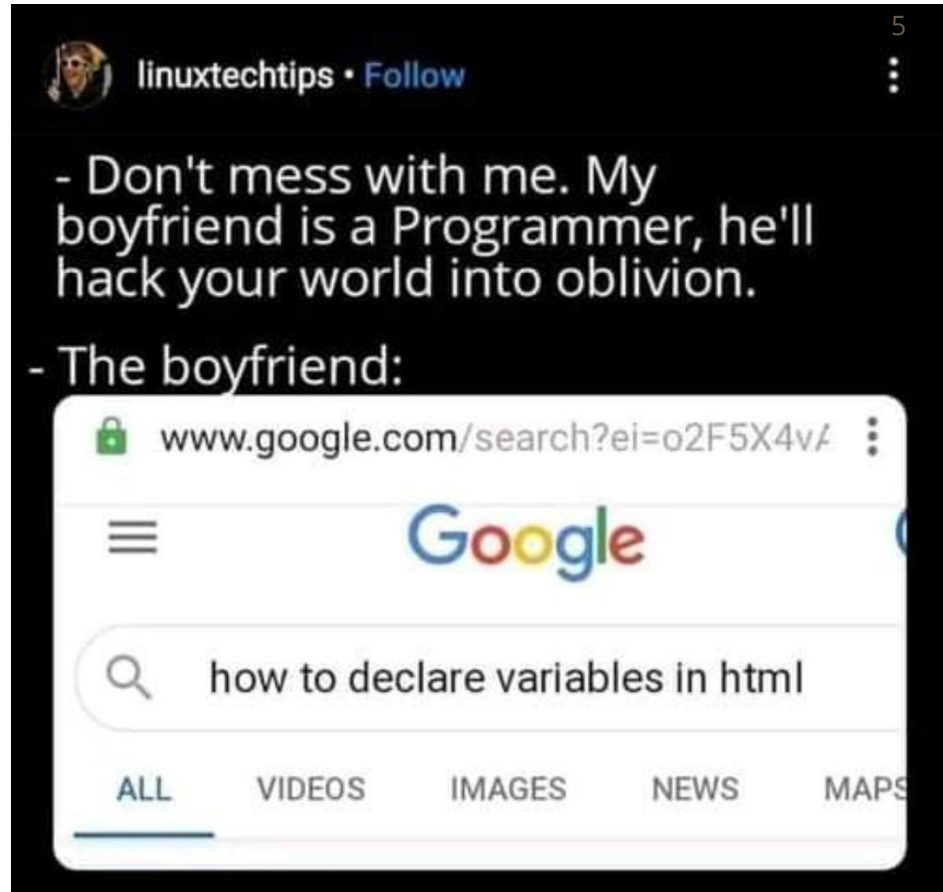


REpresentational State Transfer



Página Dinâmica?

- *"Não mexa comigo. Meu namorado é Programador e vai hackear sua vida até o esquecimento".*
- *O namorado:*
[Google] "como declarar variáveis em html"



JavaScript

6

- Habilita a criação de páginas Web interativas e dinâmicas
- JavaScript permite programar o comportamento de páginas Web
- Interpretada no cliente (no navegador = *front-end*)
- E também no servidor (Node.js)
- Navegadores da Web possuem um mecanismo dedicado à execução de JS.



JavaScript



7

- Criada em 1995.
- JavaScript, abreviado como JS, é uma linguagem de script (que roda em um ambiente de execução) interpretada (ao invés de compilada) em conformidade com a especificação ECMAScript.
- JavaScript suporta estilos de programação orientados a eventos, funcionais e imperativos (incluindo orientação a objetos).
- Inicialmente implementado apenas no lado do cliente em navegadores da Web, JavaScript foi incorporado em outros tipos de software, principalmente em servidores Web.
- Os termos **Vanilla JavaScript** referem-se ao JavaScript não estendido por nenhuma estrutura ou biblioteca adicional.

ECMA Script

- ECMAScript (ou ES) é uma especificação de linguagem de script padronizada pela Ecma International na ECMA-262 e ISO/IEC 16262.
- Foi criada para padronizar o JavaScript, de modo a promover várias implementações independentes.
- O JavaScript permaneceu a implementação mais conhecida do ECMAScript desde que o padrão foi publicado pela primeira vez, com outras implementações conhecidas, incluindo JScript e ActionScript.
- Criada em 1997.

Node.js

9

- Node.js é um ambiente de execução para JavaScript que permite a criação de aplicações para a Web.
- O Node.js permite o uso do JavaScript no lado do servidor, tornando-o uma linguagem *full-stack*.
- O Node.js é construído sobre o motor JavaScript do Google Chrome (V8) e utiliza um modelo de I/O baseado em eventos não-bloqueantes.



Ambiente de Dev

10

- **Visual Studio Code**
- **Node.js**: interpretador de código JavaScript que interpreta o código fora do navegador

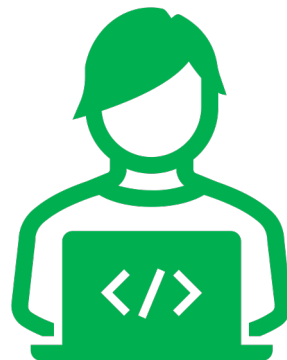
Visual Studio
Code



Ambiente

- node -v
- npm -v

Vamos testar!



```
[brunaru@MacBook-Air-de-Bruna ~ % node -v  
v19.6.0  
[brunaru@MacBook-Air-de-Bruna ~ % npm -v  
9.4.0  
brunaru@MacBook-Air-de-Bruna ~ %
```

JavaScript: Características



12

- Extensão de arquivo: .js
- Sentenças podem ter ou não ter `;` ao final
- Pode usar aspas simples `'` ou aspas dupla `"`
- Blocos de sentença são indicados por colchetes `{ }`
- Comentários são adicionados assim `// comentário` ou `/* comentário */`
- Linguagem de tipagem fraca
- Cuidado, JavaScript é *case sensitive*

Meu Primeiro Código

13

arquivo.js

```
function greetMe(yourName) {  
  console.log('Hello ' + yourName)  
}
```

```
greetMe('World')
```

node arquivo.js



Executa código no terminal

Declaração

14

- Declaração de variáveis: var, let e const
- **var**: declara uma variável **global**, inicializando-a opcionalmente como um valor.
- **let**: declara uma variável **local** com escopo de bloco, inicializando-a opcionalmente como um valor.
- **const**: declara uma variável **local constante** com escopo de bloco, inicializando-a opcionalmente como um valor.

Declaração

15

```
if (true) {  
    var x = 5;  
}  
console.log(x); // x é 5
```

```
if (true) {  
    let y = 5;  
}  
console.log(y); // y é undefined
```


Case Sensitive

16

```
let myValue = 1  
let myvalue = 2  
console.log(myValue)  
console.log(myvalue)
```

Tipagem Fraca

17

```
let answer = 42 // atribuição  
answer = 'O valor é 42...'
```

```
x = 'A resposta é ' + 42 // resultado "A resposta é 42"  
y = 42 + ' é a resposta' // resultado "42 é a resposta"
```

```
'37' - 7 // resultado 30
```

```
'37' + 7 // resultado "377" - concatenação de string
```

Conversão

18

- parseInt(): apenas inteiros sem casas decimais
- parseFloat()

```
parseInt('37') + 7 // resultado 44
```

```
parseFloat('25.78') + 5.5 // resultado 31.28
```

Conversões

19

- `parseFloat()`
- `parseInt()`
- `toString()`

Tipos

20

Sete tipos de dados que são primitivos:

1. **Boolean**: verdadeiro/falso.
2. **null**: palavra-chave que indica um valor nulo. Como o JavaScript diferencia maiúsculas de minúsculas, null não é igual a Null, NULL ou qualquer outra variante.
3. **Undefined**: propriedade cujo valor não está definido.
4. **Number**: número inteiro ou ponto flutuante. Exemplo: 42 ou 3.14159.
5. **BigInt**: número inteiro com precisão arbitrária.
6. **String**: sequência de caracteres que representam um valor de texto.
7. **Symbol**: tipo de dados cujas instâncias são únicas e imutáveis.
8. **Object**: objetos genéricos

Null, Undefined e NaN

21

- **null**: valor primitivo que representa a ausência intencional de um valor de objeto
- **undefined**: valor primitivo utilizado quando uma variável não teve valor atribuído
- **NaN**: "not a number"

Operadores

22

- **Atribuição** `= ; += ; -= ; *= ; /=`
- **Aritméticos** `+, -, *, /, %`
- **Relacionais** `== ; === ; != ; !== ; <= ; >= ; < ; >`
- **Lógicos** `|| ; && ; !`
- **Unários** `++ ; --`

Igual x Estritamente Igual

23

```
if (1 == '1') {  
    console.log('é igual')  
} else {  
    console.log('não é igual')  
}
```

```
if (1 === '1') {  
    console.log('é igual')  
} else {  
    console.log('não é igual')  
}
```


Conditionais

24

```
if (condition) {  
    statement_1;  
} else {  
    statement_2;  
}
```

Condicionais

25

```
if (condition_1) {  
    statement_1;  
} else if (condition_2) {  
    statement_2;  
} else if (condition_n) {  
    statement_n;  
} else {  
    statement_last;  
}
```

Laços

26

```
while (condition) {  
    statement  
}
```

```
let n = 0;  
let x = 0;  
while (n < 3) {  
    n++;  
    x += n;  
}
```

Laços

27

```
for ([initialExpression]; [condition]; [incrementExpression]) {  
    statement  
}
```

```
for (let i = 0; i < 10; i++) {  
    console.log(i);  
}
```

Switch

28

```
switch (expression) {  
    case label_1:  
        statements_1  
        [break;]  
    case label_2:  
        statements_2  
        [break;]  
    ...  
    default:  
        statements_def  
        [break;]  
}
```

Switch

29

```
switch (fruitType) {  
  case 'Oranges':  
    console.log('Oranges are $0.59.')  
    break  
  case 'Apples':  
    console.log('Apples are $0.32.')  
    break  
  case 'Mangoes':  
    console.log('Mangoes are $0.56.')  
    break  
  case 'Papayas':  
    console.log('Mangoes and papayas are $2.79.')  
    break  
  default:  
    console.log('Sorry, we are out of ' + fruitType + '.')  
}  
console.log("Is there anything else you'd like?")
```



Template Strings

30

- Recurso que permite escrever variáveis e sentenças de múltiplas linhas
- Utiliza-se **crase** para delimitar o texto a ser escrito

```
let a = 5
let b = 10
console.log(`Quinze é ${a + b} e
            não é ${2 * a + b}.`)
```

Funções

31

```
function somaImprime (a, b) {  
  console.log(`${a} + ${b}`)  
}
```

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


Funções

32

- Funções anônimas: são funções sem um nome que são atribuídas a uma variável.

```
const somaImprime = function(a, b) {  
  console.log(`${a} + b`)  
}
```

```
somaImprime(3, 4)
```

Funções

33

Motivos para evitar funções anônimas:

- São mais difíceis de depurar ("debugar")
- São mais difíceis de testar
- Não descrevem o papel da função
- Gera código mais confuso



Funções Arrow

34

- Exemplo, essa função:

```
const writeMyName = function(name) {  
    console.log(`Hello my friend ${name}!`)  
}
```

- Pode ser escrita assim:

```
const writeMyName = name => console.log(`Hello my friend ${name}!`)
```

Funções Arrow

35

- Exemplo de Arrow Function com múltiplas linhas:

```
const divide = (a, b) => {  
  if (b !== 0) {  
    return a / b  
  }  
  return 0  
}
```

Funções Arrow

36

- Exemplo de Arrow Function como callback:

```
let nums = [4, 62, 11, 90, 21, 30, 1, 89, 8]
```

```
nums.sort((a,b) => a - b)
```

Objetos

37

```
let myCar = new Object();  
myCar.make = 'Ford';  
myCar.model = 'Mustang';  
myCar.year = 2015;
```

OU

```
let myCar = { make: 'Ford', model: 'Mustang', year: 2015 }
```

- Propriedades não definidas de um objeto são ***undefined***

Objetos

38

- Propriedades de objetos em JavaScript podem também ser acessadas ou alteradas usando-se notação de colchetes. Objetos são às vezes chamados de *arrays associativos*, uma vez que cada propriedade é associada com um valor de string que pode ser usado para acessá-la.

```
let myCar = new Object();  
myCar['make'] = 'Ford';  
myCar['model'] = 'Mustang';  
myCar['year'] = 2015;
```

Objetos

39

Atribuição por referência!

```
let myCar = { make: 'Ford', model: 'Mustang', year: 2015 }  
console.log(myCar)  
let newCar = myCar;  
newCar.price = '$27.000'  
console.log(myCar)
```



Objetos

40

- Objetos também podem possuir funções próprias:

```
var person = {  
  firstName: "Anésia",  
  lastName: "da Silva",  
  age: 87,  
  fullName: function () {  
    return this.firstName + " " + this.lastName;  
  }  
};
```



Classes

41

- Até agora nós vimos como declarar objetos únicos com valores já definidos.
- Classes permitem instanciar objetos que compartilham as mesmas propriedades.
- Em JavaScript as classes são, na verdade, "funções especiais" sendo possível definir expressões e declarações de funções em objetos.
- Classes foram adicionadas ao JS em 2015 pela ES6.
- Para declarar uma classe usamos a propriedade **class** e definimos seu construtor com **constructor**

Classes

42

```
class Hero {  
  constructor(name, level) {  
    this.name = name  
    this.level = level  
  }  
  hello() {  
    return `${this.name} says Everything is impossible until somebody does it.`  
  }  
}
```

```
let myHero = new Hero('Batman', 99)
```



Classes

43

- A herança conta com a declaração **extends**:

```
class Mage extends Hero {  
    constructor(name, level, spell) {  
        super(name, level);  
        this.spell = spell;  
    }  
}
```

Classes

44

- O método clássico para definição de classes de objetos são as **funções construtoras**. A criação de "classes" era realizada dessa forma antes da versão ES6 de 2015. Você pode se deparar com códigos desse tipo:

```
function Person(first, last, age, eyecolor) {  
    this.firstName = first  
    this.lastName = last  
    this.age = age  
    this.eyeColor = eyecolor  
}  
  
var myFather = new Person("John", "Doe", 50, "blue")  
var myMother = new Person("Sally", "Rally", 48, "green")
```

Classes

45

- E como adicionar funções à minha função construtora?

```
function Person(first, last, age, eyecolor) {  
    this.firstName = first  
    this.lastName = last  
    this.age = age  
    this.eyeColor = eyecolor  
    this.name = function() {  
        return this.firstName + " " + this.lastName  
    }  
}
```

Classes

46

- Você também verá códigos assim:

```
function Hero(name, level) {  
    this.name = name  
    this.level = level  
}  
Hero.prototype.hello = function() {  
    return `${this.name} says hello.`  
}
```

- A propriedade **prototype** permite adicionar propriedades e métodos a um objeto. Ela é presente em todos os objetos em JS.

Classes

47

- Priorize a especificação mais recente, ou seja, utilize a declaração com **class** e herança com **extends**!

```
class Hero {  
    constructor(name, level) {  
        this.name = name  
        this.level = level  
    }  
}  
  
class SuperHero extends Hero {  
    constructor(name, level, power) {  
        super(name, level)  
        this.power = power  
    }  
}  
  
let myHero = new SuperHero('Shoto', 25, 'Hot & Cold')
```



Arrays

48

- Heterogêneos: fracamente tipado, ou seja, posso misturar diferentes tipos de conteúdo, como texto, números e objetos. No entanto, é melhor não misturar
- Métodos importantes: `sort`, `pop`, `shift`, `push`, `splice`
- Funções importantes: `forEach`, `filter`

Todos os métodos e funções: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Arrays

49

```
let fruits = ['Apple', 'Orange', 'Banana', 'Mango']
```

```
fruits.push('Blackberry') // adiciona um item
```

```
fruits.sort() // ordena
```

```
let pos = 2, n = 1
```

```
fruits.splice(pos, n) // extrai n itens à partir de pos
```

```
fruits.pop() // extrai o último
```

```
fruits.shift() // extrai o primeiro
```

Arrays

50

```
for (variável of objeto_interável) {  
    sentenças  
}
```

```
let arr = [3, 5, 7, 11];
```

```
for (let i of arr) {  
    console.log(i);  
}
```

Arrays

51

```
// percorre os itens do array  
fruits.forEach(function(item, index, array) {  
    console.log(item, index);  
});
```

Funções Callback

52

- **Callback Functions** são um conceito muito importante em JS.
- Uma função de Callback é uma função passada para outra função como **argumento**, que é então chamada dentro da função que a recebe como parâmetro para concluir algum tipo de rotina ou ação.
- Em JS temos vários métodos que recebem funções como argumento.



Funções Callback

53

- Por exemplo, o método `forEach()` de um Array:

```
let nums = [4, 62, 11, 90, 21, 30, 1, 89]
```

```
let escrever = function(valor, index, nums) {  
    console.log(`O valor do índice ${index} do array é ${valor}`)  
}
```

```
nums.forEach(escrever)
```

Funções Callback

54

- A função callback também pode ser escrita diretamente como argumento:

```
let nums = [4, 62, 11, 90, 21, 30, 1, 89]
```

```
nums.forEach(function(valor, index) {  
    console.log(`O valor ${index} do array é ${valor}`)  
})
```

Funções Callback

55

- Outro exemplo:

```
let nums = [4, 62, 11, 90, 21, 30, 1, 89, 8]
```

```
nums.sort(function(a, b) {  
    return a - b  
})
```

```
console.log(nums)
```

`sort(compareFn?: (a: number, b: number) => number): number[]`

Função usada para determinar a ordem dos elementos. Espera-se que retorne um valor negativo se o primeiro argumento for menor que o segundo argumento, zero se eles forem iguais ou um valor positivo caso contrário. Se omitido, os elementos são classificados em ordem crescente de caracteres ASCII.

Funções Callback

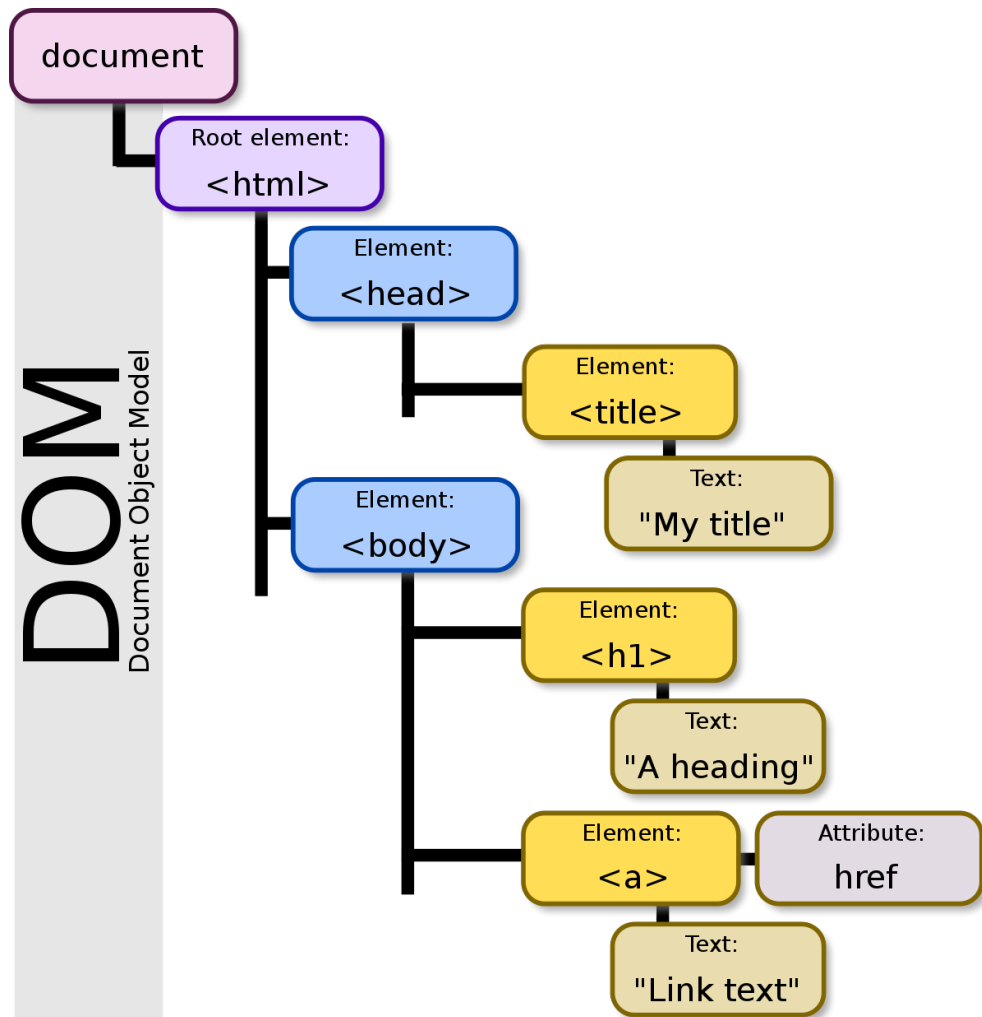
56

- O mais interessante é que podemos utilizar esse recurso ao construirmos nossas funções. Exemplo:

```
function somar(a, b) {  
    return a + b  
}  
  
function imprimeCalculo(meuCalculo, a, b) {  
    const result = meuCalculo(a, b)  
    console.log('Resultado: ' + result)  
}  
  
imprimeCalculo(somar, 34, 12)
```



DOM



<https://nodejs.org/>

58



Node.js® is a JavaScript runtime built on [Chrome's V8 JavaScript engine](#).

Download for macOS (x64)

16.16.0 LTS

Recommended For Most Users

18.7.0 Current

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#)

[Other Downloads](#) | [Changelog](#) | [API Docs](#)

Or have a look at the [Long Term Support \(LTS\) schedule](#)

NPM

59

- O NPM (Node Package Manager) é uma ferramenta de linha de comando que facilita a instalação e gerenciamento de bibliotecas que podem ser usadas em aplicações Node.js.
- O NPM foi criado para ajudar os desenvolvedores a compartilhar e reutilizar código e, desde então, se tornou o maior repositório de pacotes do mundo.
- O NPM também fornece uma série de comandos úteis para executar scripts e tarefas, como *npm start* para iniciar o aplicativo e *npm test* para executar testes. Estes comandos são especificados no arquivo *package.json*.
- Node.js e NPM são ferramentas essenciais para construir aplicações *backend* em JS e gerenciar dependências.

NPM: Comandos Essenciais

60

npm init

- Usado para inicializar um novo projeto Node.js e criar um arquivo package.json.
- Quando você executa npm init, você será guiado por um processo interativo que lhe pedirá informações sobre o seu projeto, como nome, versão, descrição e licenças.
- Ao finalizar, o NPM criará o arquivo **package.json** que contém as informações sobre o seu projeto. Este arquivo é importante por permitir que outros instalem e usem facilmente o seu projeto.

NPM: Comandos Essenciais

61

npm install nome-do-pacote

- Comando usado para instalar pacotes no seu projeto. O NPM procura o pacote especificado na Internet e o baixa para o seu projeto.
- O pacote também é adicionado à seção dependencies do arquivo **package.json**, o que significa que ele será instalado automaticamente quando alguém instalar o seu projeto.
- O comando npm install também pode ser usado sem especificar um pacote específico. Nesse caso, o NPM procurará as dependências especificadas no arquivo package.json e as instalará.
- O comando **npm i** é uma forma abreviada de npm install.

NPM: Comandos Essenciais

62

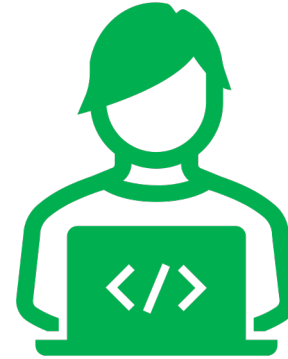
Variações comuns do comando **npm i**:

- **npm i -g <nome-do-pacote>**: instala o pacote globalmente, o que significa que o pacote ficará disponível para todos os projetos em sua máquina, em vez de apenas para o projeto atual.
- **npm i --save-dev <nome-do-pacote>**: instala o pacote e o adiciona à seção devDependencies do arquivo package.json. Isso significa que o pacote será instalado apenas em modo de desenvolvimento, e não em modo de produção.

Ambiente

63

- **node -v**
- **npm -v**
- **npm install -g nodemon**



```
[brunaru@MacBook-Air-de-Bruna ~ % node -v  
v19.6.0  
[brunaru@MacBook-Air-de-Bruna ~ % npm -v  
9.4.0  
brunaru@MacBook-Air-de-Bruna ~ %
```


REST

64

As características básicas do REST incluem:

- 1. Recursos:** os dados e funcionalidades são representados como recursos identificados por URIs (Uniform Resource Identifiers) únicos.
- 2. Métodos HTTP:** usa os métodos HTTP, GET, POST, PUT e DELETE, para realizar operações nos recursos. O método GET é usado para recuperar informações, o POST é usado para criar um novo recurso, o PUT é usado para atualizar um recurso existente e o DELETE é usado para excluir um recurso.
- 3. Representação de recursos:** os recursos são representados como dados estruturados, geralmente em formato JSON ou XML. Isso permite que as aplicações cliente e servidor sejam independentes do formato dos dados.
- 4. Statelessness:** não mantém nenhum estado entre as solicitações. Cada solicitação deve conter todas as informações necessárias para o servidor realizar a ação solicitada, sem depender de informações armazenadas em sessão ou em outro lugar.

Express

65

- O Express é um *framework* de aplicações web para o Node.js que fornece uma maneira conveniente e flexível de criar e gerenciar aplicações web.
- O Express fornece várias funcionalidades, como gerenciamento de rotas, tratamento de erros, integração com banco de dados e recursos de segurança
- Com ele, desenvolvedores podem se concentrar em seus projetos em vez de se preocuparem com tarefas repetitivas.
- Auxilia na criação de APIs REST
- Instalação:
- **npm i express**

Express

66

```
import express from 'express'
const app = express()
const porta = 3000

app.get('/hello', function(request, response) {
  response.send('Olá Mundo!')
})

app.listen(porta, function() {
  console.log('WS rodando na porta ' + porta)
})
```

Documentação do Express

67

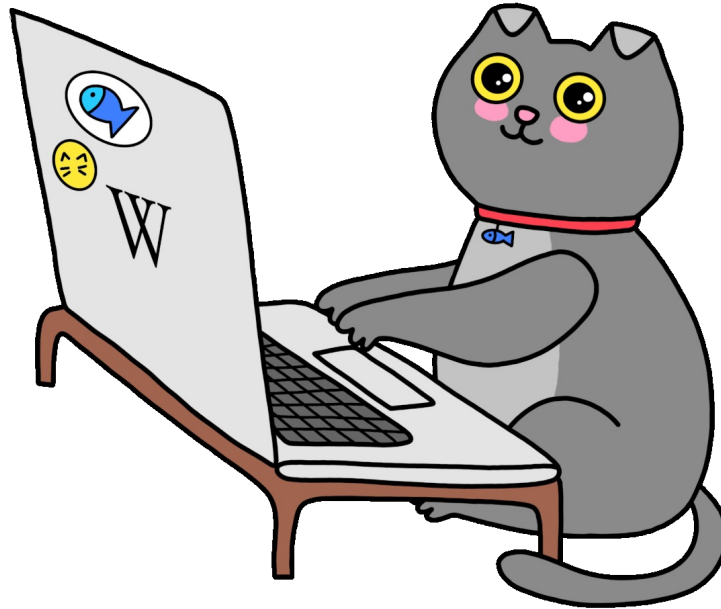
Documentação completa da API:

<https://expressjs.com/en/4x/api.html>

Prática

HELLO, WORLD!

68



require vs import

69

- Em JavaScript, as palavras-chave **require** e **import** são usadas para importar módulos.
- O **import** é uma nova palavra-chave usada para importar módulos no ECMAScript 6 (ES6). Como se trata de um recurso mais novo e poderoso, vamos preferir usar o **import** à partir de agora.

require vs import

70

```
const meuModulo = require('meuModulo')
```

é o mesmo que:

```
import * as myModule from ' meuModulo'
```

ou

```
import meuModulo from 'meuModulo'
```

require vs import

71

```
const express = require('express')
```

=

```
import express from 'express'
```


require vs import

72

Algo bastante útil do **import** é que ele permite importar variáveis e funções específicas, sem precisar trazer um módulo inteiro. Exemplo:

```
// moduloA.js
```

```
export const x = 1
```

```
export const y = 2
```

```
// moduloB.js
```

```
import { x, y } from './moduloA.js'
```

require vs import

73

Importante!!!

Para a importação com o padrão *import funcionar*, temos de adicionar a seguinte linha em nosso **package.json**:

```
"type": "module",
```

Exportar

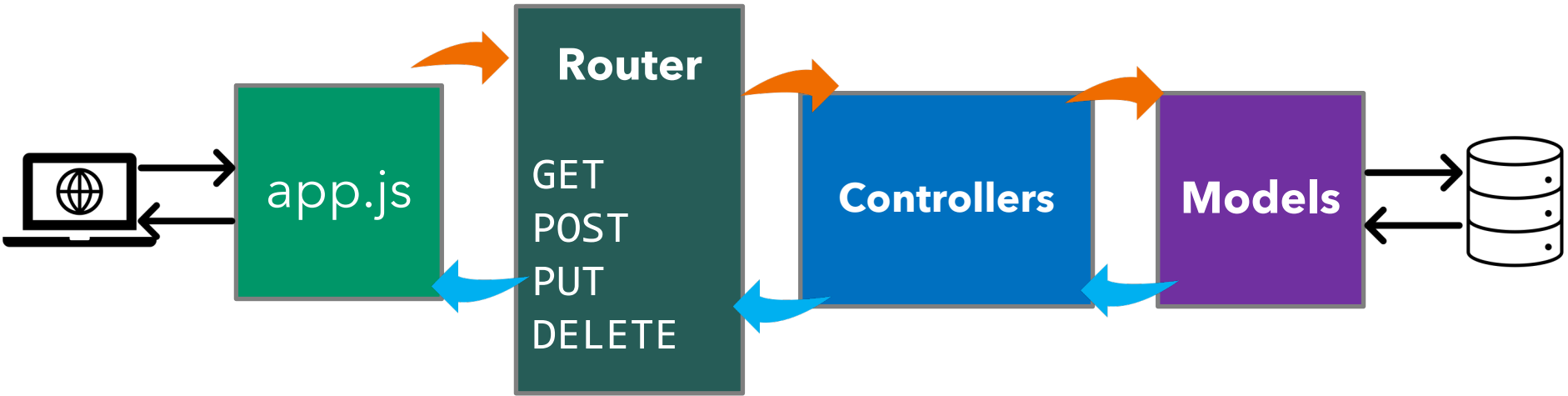
74

- Quando você faz seu próprio módulo (um arquivo .js), é preciso indicar quais elementos podem ser importados pelo outros módulos.
- Para isso usa-se a palavra chave **export**
- Sua variação, **export default**, é usada para indicar o padrão (mínimo) a ser importado. Exemplo:

```
export default {findAll, findByPk, create, update,  
deleteByPk}
```

MVC no Node.js

75



Ponto de Entrada

76

app.js

Responsável por “ouvir” uma determinada porta e configurar a aplicação pelo express.

```
import express from 'express'
import routes from './routes/api.routes.js'

const app = express();

app.use(express.json())
app.use(express.urlencoded({extended: true}))
app.use(routes)

app.listen(3001, () => console.log('Servidor iniciado na porta 3001'))
```



Router

77

routes/api.routes.js

Indica os caminhos de acesso da API Rest. Exemplo:

```
import express from 'express'
import express from 'express'
import breeds from '../controllers/dogbreed.controller.js'

const routes = express.Router()

routes.get('/breed', breeds.getAny)
routes.get('/breeds/:id', breeds.getOne)

export default routes
```

Controller

78

controllers/dogbreed.controller.js

Responsável por conversar com os módulos de model e responder às requisições adequadamente.

```
const breeds = ['Golden Retriever', 'French Bulldog',  
'Cocker Spaniel', 'Shih Tzu', 'Border Collie', 'Beagle', 'German Spitz',  
'Poodle', 'Yorkshire Terrier', 'Welsh Corgi', 'Siberian Husky']  
  
function getAny(request, response) {  
  const i = Math.floor(Math.random() * breeds.length)  
  response.send(breeds[i])  
}  
  
function getOne(request, response) {  
  response.send(breeds[request.params.id])  
}  
  
export default { getAny, getOne }
```

Prática

79



Express

80

- Vamos criar uma API Rest para representar um CRUD (*create, read, update e delete*) dos recursos Cliente e Pet



Documentação do Express

81

- **`app.get(caminho, callback [, callback ...])`**
 - Roteia solicitações HTTP GET para o caminho especificado com as funções de retorno de chamada especificadas.
- **`app.post(caminho, callback [, callback ...])`**
 - Roteia solicitações HTTP POST para o caminho especificado com as funções de retorno de chamada especificadas. Para obter mais informações, consulte o guia de roteamento.
- Notem que podemos chamar um número não determinado de callbacks...

Integrando com o Banco de Dados

82

- Podemos utilizar o módulo do banco e fazer operações diretamente em nosso código.
- Porém essa alternativa não é eficiente nem adequada para projetos atuais.
- Exemplos completos em: <https://blog.logrocket.com/crud-rest-api-node-js-express-postgresql/>

Exemplo de Conexão com Módulo mysql2

83

```
import mysql from "mysql2"

const pool = mysql.createConnection({
  host: "localhost",
  user: "username",
  password: "password"
})

pool.connect(function(err) {
  if (err) throw err;
  console.log("Connected!")
})
```



Exemplo de Conexão com Módulo mysql2

84

```
const getUsers = (request, response) => {  
  pool.query('SELECT * FROM users ORDER BY id ASC', (error, results) => {  
    if (error) {  
      throw error  
    }  
    response.status(200).json(results.rows)  
  })  
}
```

Mapeamento Objeto-Relacional

85

- **Mapeamento Objeto-Relacional (MOR)** ou **Object Relational Mapper (ORM)** é uma técnica de programação que permite relacionar objetos com dados em um banco de dados.
- Desta forma, **não é preciso se preocupar com a escrita de código SQL**, dado que a técnica realiza o mapeamento dos objetos em tabelas e dados.
- Existem diferentes ferramentas de Mapeamento OR. As ferramentas dependem da linguagem de programação utilizada.
- A forma que o mapeamento é configurado depende da ferramenta.
- **A ferramenta ORM mapeia o código ao banco automaticamente por meio dessas configurações.**

Mapeamento Objeto-Relacional

86

- Podemos utilizar diferentes ferramentas ORM para o Node.js.
- No caso de bancos relacionais, a **Sequelize** é uma das soluções mais famosas.



- A Sequelize funciona com Postgres, MySQL, MariaDB, SQLite, DB2, Microsoft SQL Server, Snowflake, Oracle DB e Db2 IBM i.

Mapeamento Objeto-Relacional

87

- Para mapear nosso código com o banco de dados utilizando a **Sequelize**, nós temos de instalar o módulo **sequelize** e o módulo do banco utilizado (no caso, **mysql2**).
- Após inicializar um objeto de conexão, é necessário criar os modelos das entidades que devem persistir no banco.

Inicialização do Banco com Sequelize 88

- Instalar os seguintes módulos:

`npm i express`

`npm i cors`

`npm i sequelize`

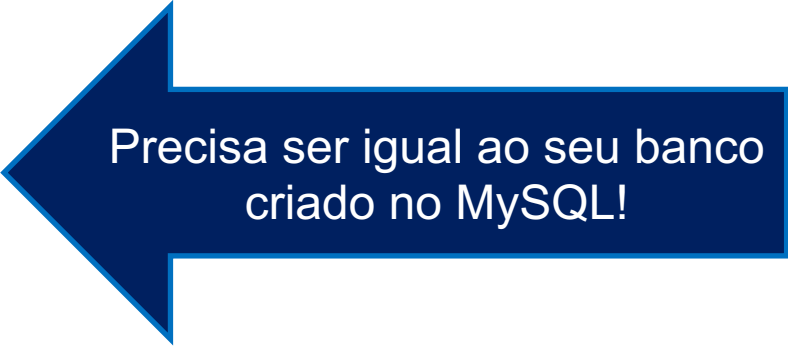
`npm i mysql2`

```
CREATE DATABASE devpet;
```

Inicialização do Banco com Sequelize 90

- Neste exemplo vamos utilizar um arquivo **.env** na raiz do projeto com os dados do nosso banco:

```
const dbName = 'devpet'  
const dbUser = 'userpet'  
const dbHost = '127.0.0.1'  
const dbPassword = '123456'  
const dbPort = 3306
```



Precisa ser igual ao seu banco
criado no MySQL!

Inicialização do Banco com Sequelize 91

db.js

```
import Sequelize from 'sequelize'

const dbName = 'devpet'
const dbUser = 'userpet'
const dbHost = '127.0.0.1'
const dbPassword = '123456'
const dbPort = 3306

const sequelize = new Sequelize(dbName, dbUser, dbPassword, {
  dialect: 'mysql', host: dbHost, port: dbPort
})

export default sequelize
```

Criação de um Modelo

92

models/client.model.js

```
import { Model, DataTypes } from 'sequelize'
import sequelize from '../db.js'

class Client extends Model {}

Client.init( {
  id: { type: DataTypes.INTEGER, autoIncrement: true,
primaryKey: true },
  name: { type: DataTypes.STRING, allowNull: false },
  document: { type: DataTypes.STRING, allowNull: false }
}, { sequelize: sequelize, timestamps: false })

export default Client
```

Testando o Modelo

93

- Com o modelo de Cliente pronto, já podemos fazer certas ações no banco!

`dbsync.js`

```
import Client from './models/client.model.js'

Client.sync()
await Client.create({name: "Endeavor", document: "202.002.002-2"})

Client.findAll().then(function(res) {
  for(let r of res) {
    console.log(r.dataValues)
  }
})
```

- Note que as chamadas são assíncronas!!!**



Promise

94

- As promessas (*promises*) são a base da programação assíncrona no JavaScript. Uma **promise** é um objeto retornado por uma função assíncrona, que representa o estado atual da operação. No momento em que a promessa é retornada ao “chamador”, o objeto da promessa fornece métodos para lidar com o eventual sucesso ou falha da operação.

```
Client.findAll().then(function(res) {  
  for(let r of res) {  
    console.log(r.dataValues)  
  }  
}).catch(function(e) { console.log(e) })
```

Promise e await

95

- O operador **await** é usado para aguardar uma **promise** e obter seu valor de cumprimento. Só pode ser usado dentro de uma função assíncrona ou no nível superior de um módulo.

```
await Client.create({name: "Endeavor", document: "202.002.002-2"})
```


Controller

96

controllers/client.controller.js

```
import Client from '../models/client.model.js'

function findAll(request, response) {
  Client.findAll().then(results => {
    response.json(results).status(200)
  }).catch(e => console.log(e))
}

async function create(request, response) {
  Client.create({ name: request.body.name, document: request.body.document})
  .then((result) => {
    response.status(201).json(result)
  }).catch((e) => console.log(e))
}

export default { findAll, create }
```

Prática

97



Associações

98

- O Sequelize suporta as associações clássicas de Um-Para-Um, Um-Para-Muitos e Muitos-Para-Muitos.
- Isso é feito com 4 tipos de comandos:
 - **HasOne**
 - **BelongsTo**
 - **HasMany**
 - **BelongsToMany**

Associações

99

- Para criar uma relação de Um-Para-Muitos (One-To-Many) entre Pessoa e Endereço (*i.e.*, uma pessoa pode ter vários endereços), utilizo o comando **hasMany**:

```
Pessoa.Enderecos = Pessoa.hasMany(Endereco)
```

- Para criar uma relação de Muitos-Para-Um (Many-To-One) entre Pessoa e endereço (*i.e.*, uma pessoa pode ter vários patrimônios), posso utilizar também o comando **belongsTo**:

```
Patrimonio.belongsTo(Pessoa, { foreignKey: 'responsavelId' })
```

- A configuração **foreignKey** permite mudar o nome da chave estrangeira.

Associações

100

models/pet.model.js

```
import { Model, DataTypes } from 'sequelize'
import database from '../db.js'
import Client from './client.model.js'

class Pet extends Model {}

Pet.init({
  id: { type: DataTypes.INTEGER, autoIncrement: true, primaryKey: true },
  name: { type: DataTypes.STRING, allowNull: false },
  type: { type: DataTypes.STRING, allowNull: false },
  breed: { type: DataTypes.STRING },
  birth: { type: DataTypes.STRING }
}, { sequelize: database, timestamps: false })

Pet.belongsTo(Client)
Client.hasMany(Pet)

export default Pet
```

Rotas

```
import express from 'express'
import breeds from '../controllers/dogbreed.controller.js'
import clientController from '../controllers/client.controller.js'
import petController from '../controllers/pet.controller.js'
```

101

```
const routes = express.Router()
```

```
routes.get('/breed', breeds.getAny)
routes.get('/breeds/:id', breeds.getOne)
```

```
routes.get('/clients', clientController.findAll)
routes.get('/clients/:id', clientController.findByPk)
routes.post('/clients', clientController.create)
routes.put('/clients/:id', clientController.update)
routes.delete('/clients/:id', clientController.deleteByPk)
```

```
routes.get('/pets', petController.findAll)
routes.get('/pets/:id', petController.findByPk)
routes.post('/pets', petController.create)
routes.put('/pets/:id', petController.update)
routes.delete('/pets/:id', petController.deleteByPk)
routes.get('/clients/:id/pets', petController.findPetsOfClient)
```

```
export default routes
```

Documentação do Sequelize

102

<https://sequelize.org/docs/v6/>



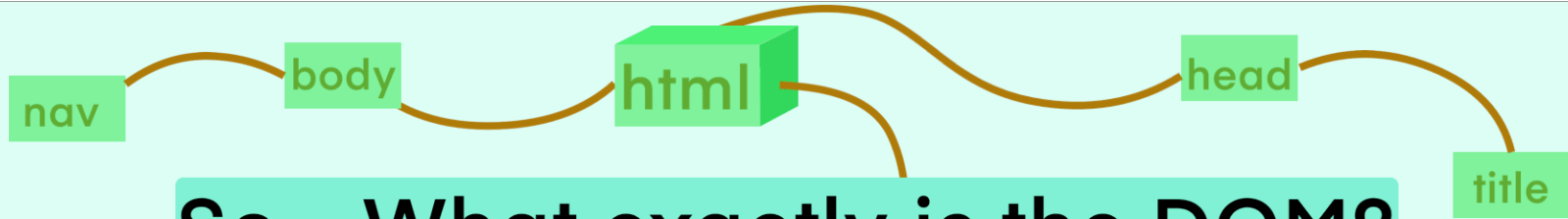
Prática

103

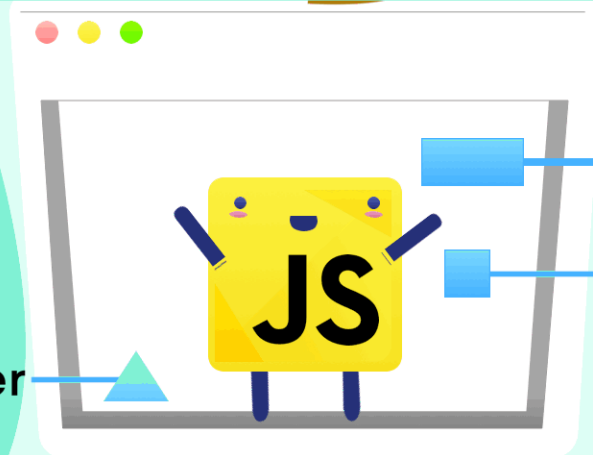


JavaScript: DOM e Eventos

104



So... What exactly is the DOM?



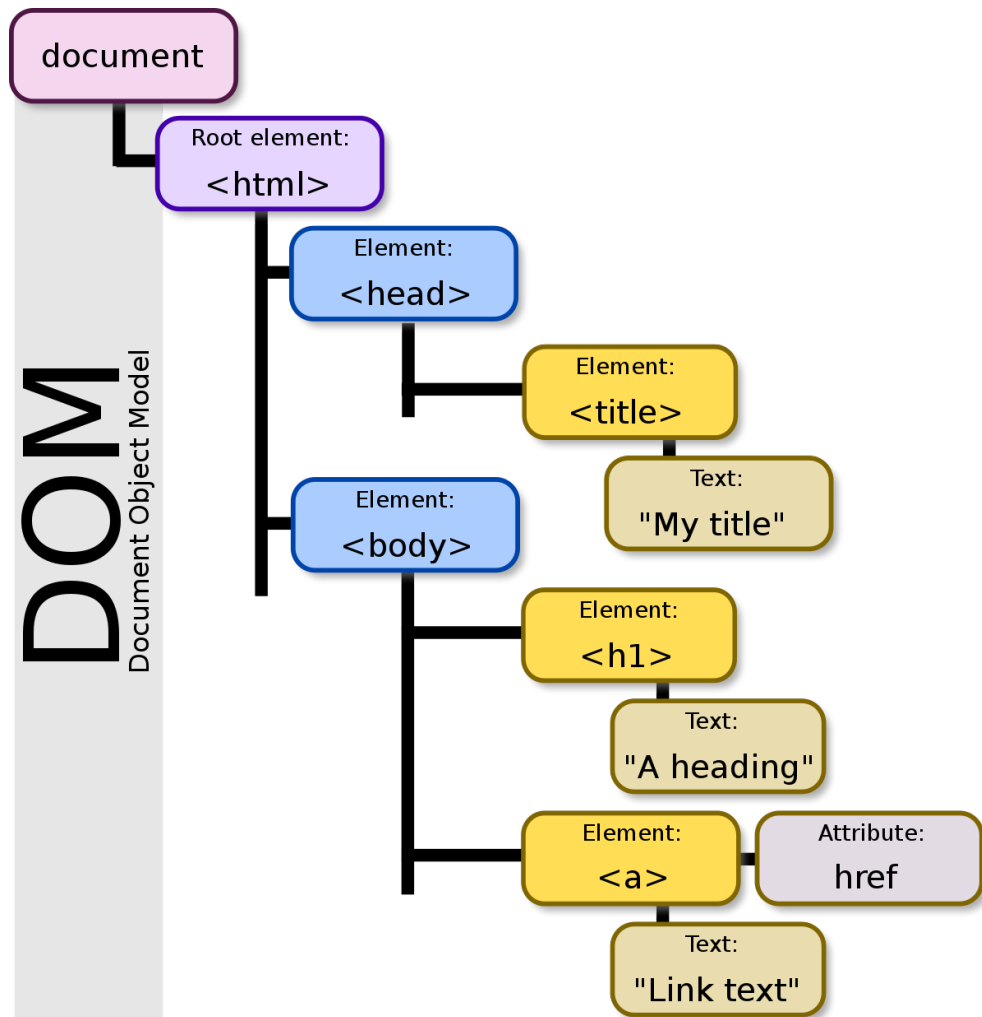
Head

Body

Footer

@codewithcorgis

DOM



DOM (Document Object Model)

106

- O Document Object Model (DOM) é uma interface multiplataforma que trata um documento HTML como uma estrutura em árvore em que cada nó é um objeto que representa uma parte do documento.
- **O DOM representa um documento com uma árvore lógica.** Cada ramificação da árvore termina em um nó e cada nó contém objetos.
- Os métodos DOM permitem acesso programático à árvore; com eles, pode-se alterar a estrutura, estilo ou conteúdo de um documento.

DOM

107

Encontrar elementos

- Pelo id: `document.getElementById(id)`
- Pela tag HTML: `document.getElementsByTagName(name)`
- Pela classe CSS: `document.getElementsByClassName(name)`

DOM

108

```
document.getElementById('client')
```

```
document.getElementsByTagName('main')
```

```
document.getElementsByClassName('destaque')
```

DOM

109

Acessando itens do elemento

- Acesso o conteúdo do elemento: `element.innerHTML`
- Acessa atributo específico: `element.getAttribute('onclick')`

HTML DOM Elements

110

```
document.getElementById('meuId').style.display = 'none'
```

```
document.getElementById('outroId').classList.add('minhaClasse')
```

```
document.getElementsByTagName('main')[0].innerHTML = "Olá Mundo!"
```

```
document.getElementsByClassName('link')[0].target = "_blank"
```

https://www.w3schools.com/jsref/dom_obj_all.asp

DOM

111

- Nós vimos alguns exemplos de métodos seletores como:
 - `document.getElementById`
 - `document.getElementsByTagName`
 - `document.getElementsByName`
 - `document.getElementsByClassName`
- Temos outras duas opções:
 - `document.querySelector`
 - `document.querySelectorAll`

DOM

112

O método **querySelector()** retorna o primeiro elemento que corresponde a um ou mais seletores CSS especificados no documento. O **querySelectorAll()** funciona da mesma forma, mas retorna todos os elementos que possuem os seletores especificados. Isso nos permite criar seleções complexas, como:

```
// Retorna todas as divs que possuem a classe .item:  
document.querySelectorAll('div .item')
```

DOM

113

- Esses métodos retornam um HTMLCollection:
 - `document.getElementsByTagName`
 - `document.getElementsByName`
 - `document.getElementsByClassName`
- Enquanto esse retorna uma NodeList:
 - `document.querySelectorAll`

Eventos

114

- Uma das características da linguagem JS é ser dirigida por eventos: ela foi projetada de modo a reagir sempre que algum evento ocorre.
- Um evento é um acontecimento envolvendo alguma atitude:
 - Do usuário: movimentar o mouse, pressionar uma tecla, enviar um formulário, etc;
 - Do funcionamento do navegador: carregamento de uma página para a exibição, não conseguir carregar uma imagem, um pop-up, etc.

Eventos

115

- O evento mais utilizado é o **onclick**. Ele é acionado quando o usuário clica no elemento.
- Existem duas formas de associar uma função a um evento: pelo atributo onclick ou pela atribuição na DOM no script JS:

1. `<button id="botaoSm" onclick="soma()">Somar</button>`

2. `document.getElementById('botaoSm').setAttribute("onclick","soma")`

Eventos

116

- Outro evento muito utilizado é o **onfocusout**. O onfocusout é acionado quando o usuário tira o foco de um elemento.
- O exemplo mais comum de uso é a validação de campos de um formulário. Toda vez que o usuário sai do campo de entrada de dados, a validação é acionada.

```
<input onfocusout="validar()" type="text" name="nome">
```

Eventos

117

- Os eventos **onmouseover** e **onmouseout** podem ser usados para disparar uma função quando o usuário passa o mouse sobre ou fora de um elemento HTML:

```
<div onmouseover="mOver(this)" onmouseout="mOut(this)">
```

```
Passe o mouse</div>
```

```
<script>
```

```
function mOver(obj) {  
    obj.innerHTML = "Obrigado"  
}
```

```
function mOut(obj) {  
    obj.innerHTML = "Passe o mouse"  
}
```

```
</script>
```

DOM

118

- Além de selecionar elementos, podemos criar!
 - `document.createElement('tag-html')`
- E adicionar ao documento:
 - `document.body.appendChild(e)`

DOM

119

- Exemplo:

```
const paragrafo = document.createElement('p')
paragrafo.innerText = 'Novo Texto'
document.getElementById('div-mensagem').appendChild(paragrafo)
const img = document.createElement('img')
img.setAttribute("src", urlDaImagem)
img.setAttribute("width", "30%")
```


Axios

120

- **Axios** é uma biblioteca popular para fazer solicitações “promise-based” em JavaScript, incluindo aplicativos Node.js.
- Ela oferece uma interface de programação de aplicativos (API) fácil de usar para realizar solicitações HTTP, como GET, POST, PUT, DELETE.
- Além disso, Axios fornece recursos adicionais, como interceptadores de solicitações e respostas, configurações globais e gerenciamento de erros.

- Exemplo API de Previsão do Tempo: <https://open-meteo.com/>
- URL com parâmetros de retorno de temperaturas mínimas e máximas de Capivari - SP:
- https://api.open-meteo.com/v1/forecast?latitude=-23.00&longitude=-47.51&daily=temperature_2m_max,temperature_2m_min&timezone=auto
- **Antes de tudo: Instalar a biblioteca do Axios na sua aplicação, ou incluir no html do website.**

`npm install axios`

OU

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

Sem Axios

122

```
async function getTemperature() {  
  const response = await fetch(url)  
  const data = await response.json()  
  const tempMax = data.daily.temperature_2m_max  
  const tempMin = data.daily.temperature_2m_min  
  console.log(`Temperaturas máximas de ${tempMax} graus Celsius.`)  
  console.log(`Temperaturas mínimas de ${tempMin} graus Celsius.`)  
}
```

```
const url = 'https://api.open-meteo.com/v1/forecast?latitude=-23.00&longitude=-  
47.51&daily=temperature_2m_max,temperature_2m_min&timezone=auto'  
getTemperature(url)
```

Com Axios

123

```
const axios = require('axios')

async function getTemperature() {
  const response = await axios.get(url)
  const tempMax = response.data.daily.temperature_2m_max
  const tempMin = response.data.daily.temperature_2m_min
  console.log(`Temperaturas máximas de ${tempMax} graus Celsius.`)
  console.log(`Temperaturas mínimas de ${tempMin} graus Celsius.`)
}

const url = 'https://api.open-meteo.com/v1/forecast?latitude=-23.00&longitude=-47.51&daily=temperature_2m_max,temperature_2m_min&timezone=auto'
getTemperature(url)
```

Prática

124



Prática

125

- Baixar arquivos HTML e CSS
- <https://github.com/brunaru/petapp-front/blob/main/style.css>
- <https://github.com/brunaru/petapp-front/blob/main/index.html>

- <https://github.com/brunaru/petapp-back>