

**Unidade Curricular:** Desenvolvimento WEB

**Professor:** Tiago Ravache

# **NodeJS**

Node.js é um ambiente de execução de código Javascript, multiplataforma, que nos permite desenvolver diversas aplicações como: servidores, aplicações de linha de comando (CLI) e aplicações web. De forma escalável e performática.

# Primeira aplicação Node

Nessa aplicação iremos usar o microframework Express.js, para criarmos um servidor, que receberá requisições HTTP e poderá gerar respostas para seus clientes (como o navegador por exemplo).

Para isso precisamos instalar o express em nossa aplicação para isso devemos executar a linha de comando: `npm install express`. No prompt/terminal.

A seguir o passo a passo a ser realizado:

# Primeira aplicação Node

- Criar uma pasta no computador;
- Dentro dela crie um arquivo chamado server.js;
- Abra essa pasta no VSCode;
- Utilize o atalho ctrl+shift+' para abrir o terminal/prompt;
- Execute o comando: npm install express;
- Digite o código a seguir no arquivo server.js

# Primeira aplicação Node

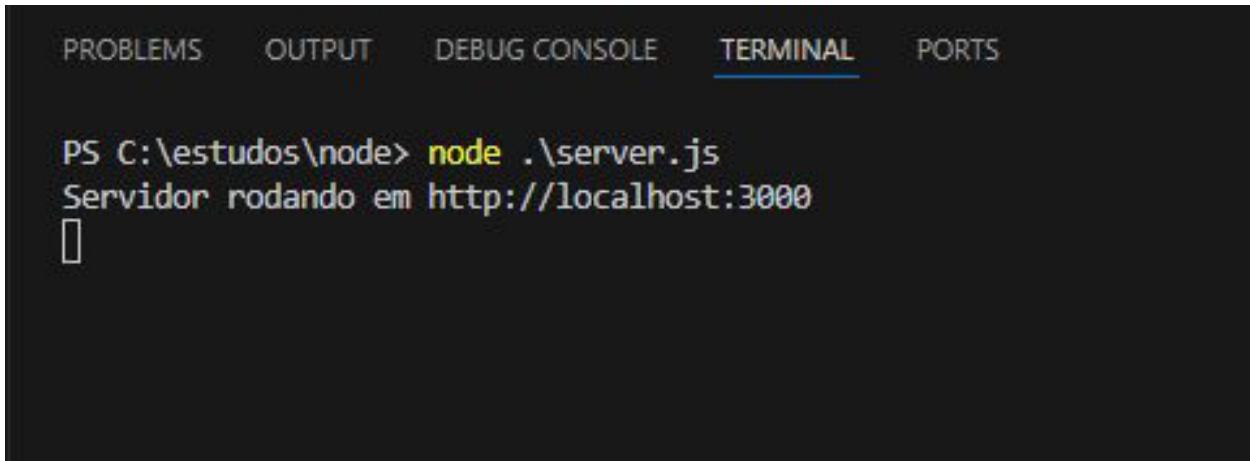
```
JS server.js  X

JS server.js > ...
1 const express = require('express');
2 const app = express();
3 const port = 3000;
4
5 app.get('/', (req, res) => {
6   res.send('Hello world');
7 });
8
9 app.listen(port, () => {
10   console.log(`Servidor rodando em http://localhost:${port}`);
11 });
12
13
14
```

# Rodando a primeira aplicação

No terminal/prompt digite o comando node .\server.js e tecle enter;

A aplicação será iniciada e a mensagem abaixo deverá aparecer no terminal:



The screenshot shows a terminal window with several tabs at the top: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined in blue), and PORTS. The terminal area displays the following text:

```
PS C:\estudos\node> node .\server.js
Servidor rodando em http://localhost:3000
[]
```

## **Rotas**

Vamos entender melhor o que são rotas;

Uma rota é uma associação entre um método HTTP, uma URL e um método de um controlador, responsável por processar a requisição e gerar a resposta dessa requisição.

# Rotas

Em destaque na imagem vemos onde está a definição da rota inicial da aplicação

```
JS server.js  X  
JS server.js > ...  
1  const express = require('express');  
2  const app = express();  
3  const port = 3000;  
4  
5  app.get('/', (req, res) => {  
6    res.send('Hello world');  
7  });  
8  
9  app.listen(port, () => {  
10    console.log(`Servidor rodando em http://localhost:${port}`);  
11  });  
12  
13  
14
```

## Criando uma nova rota

Vamos agora criar uma nova rota no nosso servidor:

- A rota será acessada através da URL: /novarota;
- Deverá usar o método GET;
- Retornar o texto “Nova rota criada”.

## Fazendo requisições HTTP

Para nos auxiliar no processo de envio de requisições HTTP usando node vamos utilizar uma biblioteca chamada ***axios***;

Portanto novamente no terminal/prompt do VSCode, digitamos o seguinte: `npm install axios`;

A seguir o código com a instanciação do axios na nossa aplicação

# Fazendo requisições HTTP

```
JS server.js  X  
JS server.js > ...  
1 const express = require('express');  
2 const axios = require('axios')  
3  
4 const app = express();  
5 const port = 3000;  
6  
7 app.get('/', (req, res) => {  
8     res.send('Hello world');  
9 });  
10  
11 app.listen(port, () => {  
12     console.log(`Servidor rodando em http://localhost:${port}`);  
13 });  
14
```

## Fazendo requisições HTTP

- Agora vamos criar uma rota chamada: /consulta-cep/:cep;
- Deverá usar o método GET;
- Usará o axios para fazer uma chamada GET para a api ViaCep passando o :cep da rota como parametro;
- Retornará o JSON que receberá da ViaCep;

A seguir o código a ser implementado:

# Fazendo requisições HTTP

```
JS server.js  X
JS server.js > ⚏ app.get('/consulta-cep/:cep') callback
1  const express = require('express');
2  const axios = require('axios')
3
4  const app = express();
5  const port = 3000;
6
7  app.get('/', (req, res) => {
8      res.send('Hello world');
9  });
10
11 app.get('/consulta-cep/:cep', async (req, res) => {
12     const cep = req.params.cep; // Obtendo o CEP da URL
13
14     try {
15         // Fazendo a requisição para a API do ViaCEP
16         const response = await axios.get(`https://viacep.com.br/ws/${cep}/json/`);
17         res.json(response.data); // Retorna os dados da resposta
18     } catch (error) {
19         console.error('Erro ao fazer requisição:', error);
20         res.status(500).send('Erro ao consultar o CEP');
21     }
22 });
23
24 app.listen(port, () => {
25     console.log(`Servidor rodando em http://localhost:${port}`);
26 });
27
```

# Fazendo requisições HTTP - Extra

Para encerrar este exercício, vamos adicionar uma validação no cep, para evitarmos enviar um cep inválido para a api que estamos consumindo.

Portanto, será necessário adicionar uma condição que faça um teste entre o cep informado na chamada a nossa api, e uma expressão regular (RegEx) que validará essas informações.

Em Javascript podemos criar uma expressão regular declarando uma variável/constante, passando o texto da expressão entre contra-barras (/).

A expressão regular para validar cep é: ^[0-9]{5}-?[09]{3}\$

No Javascript ficará: const cepRegex = /^[0-9]{5}-?[09]{3}\$/;

Para validar o cep informado contra a expressão regular usaremos cepRegex.test(cep), caso não seja válido deveremos retornar um status http 400 com a mensagem “CEP inválido. Formato: XXXXX-XXX”

# Comunicação com banco de dados

É responsabilidade do back-end também a comunicação com a camada de persistência de dados, seja arquivos em storage ou bancos de dados (SQL ou NO-SQL).

Para seguirmos com nossos exemplos em node vamos instalar uma biblioteca chamada SEQUELIZE que nos permite conectar com diversos SGBD.

Para isso no prompt/terminal do VSCode digite o comando: `npm install sequelize`

Aguarde a instalação.

# Sequelize

Sequelize é um ORM moderno TypeScript e Node.js para Oracle, Postgres, MySQL, MariaDB, SQLite e SQL Server, e mais.

Apresentando suporte sólido a transações, relações, carregamento rápido e lento, replicação de leitura e mais.

ORM – Object-Relational Mapper – é um software que mapeia entidades de bancos relacionais para objetos (OOP)

# Models

Models são a essência do Sequelize. Uma model é uma abstração que representa uma tabela no seu banco de dados. No Sequelize, é uma classe que estende o Model. A model informa ao Sequelize várias coisas sobre a entidade que ele representa, como o nome da tabela no banco de dados e quais colunas ele tem (e seus tipos de dados).

Uma model no Sequelize tem um nome. Este nome não precisa ser o mesmo nome da tabela que ele representa no banco de dados.

Normalmente, as models têm nomes singulares (como User), enquanto as tabelas têm nomes pluralizados (como Users), embora isso seja totalmente configurável.

# Models exemplo

```
1 const { Sequelize, DataTypes } = require('sequelize');
2 const sequelize = new Sequelize('sqlite::memory:');
3
4 const User = sequelize.define(
5   'User',
6   {
7     // Atributos da model são definidos aqui
8     firstName: {
9       type: DataTypes.STRING,
10      allowNull: false,
11    },
12    lastName: {
13      type: DataTypes.STRING,
14      // allowNull tem valor padrão true
15    },
16  },
17  {
18    // Outras opções podem ser inseridas aqui
19  },
20);
21
```

# Models - Endereço

- Agora vamos criar uma model chamada: Endereco;
- Atributos:
  - Id - Integer
  - Cep – String
  - Logradouro – String
  - Numero – Integer
  - Complemento – String
  - Bairro – String
  - Cidade – String
  - Estado - String
  - MunicipioIBGE - String

# Models - Endereço

```
1  const { Model, DataTypes } = require('sequelize');
2
3  class Endereco extends Model {}
4
5  Endereco.init({
6    Id: {
7      type: DataTypes.INTEGER,
8      primaryKey: true,
9      autoIncrement: true,
10 },
11 Cep: {
12   type: DataTypes.STRING,
13   allowNull: false,
14 },
15 Logradouro: {
16   type: DataTypes.STRING,
17   allowNull: false,
18 },
19 Numero: {
20   type: DataTypes.INTEGER,
21   allowNull: false,
22 },
23 Complemento: {
24   type: DataTypes.STRING,
25 },
26 Bairro: {
27   type: DataTypes.STRING,
28   allowNull: false,
29 },
30 Cidade: {
31   type: DataTypes.STRING,
32   allowNull: false,
33 },
34 Estado: {
35   type: DataTypes.STRING,
36   allowNull: false,
37 },
38 MunicipioIBGE: {
39   type: DataTypes.STRING,
40   allowNull: false,
41 },
42 }, {
43   sequelize,
44   modelName: 'Endereco',
45   tableName: 'enderecos', // Define o nome da tabela no banco de dados
46   timestamps: true, // Define se quer ou não os campos createdAt e updatedAt
47 });
48
49 module.exports = Endereco;
50
```

# Criar banco de dados

- Usaremos Postgres
- O banco de dados se chamara api-node
- Create database api-node

# Configurar Banco de dados e conexão

- Criar o arquivo database.js com os dados de conexão conforme segue:

```
module.exports = {  
  ...  
  dialect: 'postgres',  
  host: 'localhost',  
  username: 'usuario',  
  password: 'senha',  
  database: 'api-node',  
  define: {  
    timestamps: true,  
    underscored: true  
  }  
}
```

# Tabela

- Criaremos uma tabela ENDERECO que irá conter os campos:
- Id integer not null PK
- Cep varchar(8) not null
- Logradouro varchar(120) not null
- Numero smallint not null
- Complemento varchar(60)
- Bairro varchar(60) not null
- Cidade varchar(60) not null
- Estado varchar(2) not null
- IBGE varchar(8) not null

# Migrations

Para criar as migrations é necessário inicializar o sequelize-cli para isso execute o comando abaixo no prompt/terminal:

```
npx sequelize-cli init
```

Após isso com o comando abaixo criamos uma migration:

```
npx sequelize-cli migration:generate --name create-endereço
```

O comando acima gerará o arquivo a seguir:

# Migrations

```
1  'use strict';
2
3  module.exports = {
4    up: async (queryInterface, Sequelize) => {
5      await queryInterface.createTable('enderecos', {
6        Id: {
7          type: Sequelize.INTEGER,
8          primaryKey: true,
9          autoIncrement: true,
10         allowNull: false,
11       },
12       Cep: {
13         type: Sequelize.STRING,
14         allowNull: false,
15       },
16       Logradouro: {
17         type: Sequelize.STRING,
18         allowNull: false,
19       },
20       Numero: {
21         type: Sequelize.INTEGER,
22         allowNull: false,
23       },
24       Complemento: {
25         type: Sequelize.STRING,
26       },
27       Bairro: {
28         type: Sequelize.STRING,
29         allowNull: false,
30       },
31       Cidade: {
32         type: Sequelize.STRING,
33         allowNull: false,
34       },
35       Estado: {
36         type: Sequelize.STRING,
37         allowNull: false,
38       },
39       MunicipioIBGE: {
40         type: Sequelize.STRING,
41         allowNull: false,
42       },
43     });
44   },
45
46   down: async (queryInterface, Sequelize) => {
47     await queryInterface.dropTable('enderecos');
48   }
49};
```

# Migrations

Para criar a tabela no banco de dados a partir da migration é necessário rodar o comando:

```
npx sequelize-cli db:migrate
```

Caso necessário desfazer, o comando é:

```
npx sequelize-cli db:migrate:undo
```

# Finalizando a configuração do Banco

Criar o arquivo abaixo na pasta /database

```
const Sequelize = require('sequelize')
const config = require('../config/database')

const Endereco = require('../models/Endereco')

const connection = new Sequelize(config)

Endereco.init(connection)
Endereco.associate(connection.models)

module.exports = connection
```

# Criando uma controller

Uma controller é um arquivo que realiza certo processamento sobre dados requeridos produzindo uma saída aderente as necessidades de negócio

Crie uma pasta na raiz do projeto chamada controllers e dentro dela crie um arquivo chamado EnderecoController

# Código da controller

```
const { Endereco } = require('../models');

// Criação de um novo endereço
exports.createEndereco = async (req, res) => {
  try {
    const { Cep, Logradouro, Numero, Complemento, Bairro, Cidade, Estado, MunicipioIBGE } = req.body;

    const novoEndereco = await Endereco.create({
      Cep,
      Logradouro,
      Numero,
      Complemento,
      Bairro,
      Cidade,
      Estado,
      MunicipioIBGE,
    });

    res.status(201).json(novoEndereco);
  } catch (error) {
    res.status(500).json({ error: 'Erro ao criar endereço', details: error.message });
  }
};
```

# Código da controller

```
// Leitura de todos os endereços
exports.getAllEnderecos = async (req, res) => {
  try {
    const enderecos = await Endereco.findAll();
    res.status(200).json(enderecos);
  } catch (error) {
    res.status(500).json({ error: 'Erro ao buscar endereços', details: error.message });
  }
};
```

# Código da controller

```
// Leitura de um endereço por ID
exports.getEnderecoById = async (req, res) => {
  try {
    const { Id } = req.params;
    const endereco = await Endereco.findByPk(Id);

    if (!endereco) {
      return res.status(404).json({ error: 'Endereço não encontrado' });
    }

    res.status(200).json(endereco);
  } catch (error) {
    res.status(500).json({ error: 'Erro ao buscar endereço', details: error.message });
  }
};
```

# Código da controller

```
// Atualização de um endereço
exports.updateEndereco = async (req, res) => {
  try {
    const { Id } = req.params;
    const { Cep, Logradouro, Numero, Complemento, Bairro, Cidade, Estado, MunicipioIBGE } = req.body;

    const endereco = await Endereco.findByPk(Id);

    if (!endereco) {
      return res.status(404).json({ error: 'Endereço não encontrado' });
    }

    endereco.Cep = Cep;
    endereco.Logradouro = Logradouro;
    endereco.Numero = Numero;
    endereco.Complemento = Complemento;
    endereco.Bairro = Bairro;
    endereco.Cidade = Cidade;
    endereco.Estoado = Estado;
    endereco.MunicipioIBGE = MunicipioIBGE;

    await endereco.save();

    res.status(200).json(endereco);
  } catch (error) {
    res.status(500).json({ error: 'Erro ao atualizar endereço', details: error.message });
  }
};
```

# Código da controller

```
// Exclusão de um endereço
exports.deleteEndereco = async (req, res) => {
  try {
    const { Id } = req.params;

    const endereco = await Endereco.findByPk(Id);

    if (!endereco) {
      return res.status(404).json({ error: 'Endereço não encontrado' });
    }

    await endereco.destroy();

    res.status(204).send(); // Sem conteúdo, pois foi deletado com sucesso
  } catch (error) {
    res.status(500).json({ error: 'Erro ao deletar endereço', details: error.message });
  }
};
```

# Separando a configuração das rotas em um novo arquivo

Crie na raiz do projeto um arquivo chamado routes.js e implemente-o como na imagem a seguir

```
const express = require('express');
const enderecoController = require('./controllers/EnderecoController');

const router = express.Router();

router.post('/enderecos', enderecoController.createEndereco);
router.get('/enderecos', enderecoController.getAllEnderecos);
router.get('/enderecos/:Id', enderecoController.getEnderecoById);
router.put('/enderecos/:Id', enderecoController.updateEndereco);
router.delete('/enderecos/:Id', enderecoController.deleteEndereco);

module.exports = router;
```

# Alterando o arquivo server.js

Por fim alteramos o arquivo server.js como segue:

```
const express = require('express');
const rotas = require('./routes');

const app = express();

app.use(express.json());
app.use('/api', rotas);

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Servidor rodando na porta ${PORT}`));
```

# Por hoje é tudo pessoal



Até a próxima aula!