Treinamento Teknisa

Node.js

Rômulo A. Lousada

I. O que é Node.js?

II. Iniciando um Projeto em Node.js.

III. Instalando pacotes.

IV. Criando a Primeira Rota.

V. Criando um CRUD.

I. O que é Node.js

O que é Node.js

Criado em 2009, é um software de código aberto, multiplataforma e baseado no interpretador V8 do Google, permitindo a execução de códigos Javascript fora de um navegador web.

Não é uma linguagem de programação.

Não é um framework.

Utiliza o Javascript, a mesma linguagem usada há décadas pelo client-side dos navegadores.

Interpretado pelo V8, o código é entregue como server-side, tornando o Node.js muito eficiente.

O que é Node.js

Vantagens

I. **Javascript**: Por conta de ser uma linguagem existente há décadas, e ter milhões de programadores pelo mundo, é uma linguagem simples, de fácil aprendizado, e que ainda é atualizada buscando melhorias.

II. **Javascript Full-Stack**: Antes do Node.js, era necessário trabalhar com outra linguagem para o backend quando ia desenvolver uma aplicação web, como por exemplo, PHP.

III. **Leve e Multiplataforma**: Permite rodar os projetos em servidores abertos e com o sistema operacional que quiser.

O que é Node.js

Desvantagens

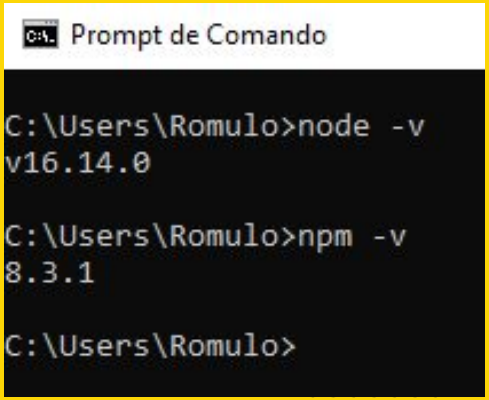
I. **Javascript**: Tem as suas vantagens, porém, javascript também possui suas desvantagens, como uma linguagem antiga, fracamente tipada (Typescript).

II. **Ecossistema Grande**: Alguns podem ver como vantagem, mas também existem muitas opções e pacotes que fazem a mesma coisa ou é muito semelhante, dificultado a escolha e decisão do que usar no projeto.

III. **Assíncrono**: Pode ser complexo inicialmente, porém, com a evolução do javascript através do ES6 (Promises) e ES7 (Async/Await), tende a tornar mais fácil o entendimento.

II. Iniciando um Projeto em Node.js

Iniciando um Projeto em Node.js



Para começar, é necessário realizar o download e instalação do Node.js na máquina.

Link do site: https://nodejs.org/en/

Após o término da instalação, para garantir que foi corretamente instalado, tente executar no cmd alguns comandos básicos, para teste:

node -v

npm -v

Problemas Comuns

Após executar os comandos node -v e npm -v:

Caso retorne o número da versão em ambos, tudo está correto.

Caso retorne a mensagem de comando não reconhecido, tente reiniciar o VSCode, o Terminal do VSCode e/ou o computador.

Se ainda não reconhecer o comando, desinstale e instale novamente o programa.

Em último caso, após reinstalar e o problema persistir, verificar as variáveis de ambiente e confirmar que o PATH foi criado corretamente.

Iniciando um Projeto em Node.js

- Criar uma pasta vazia, onde o projeto será iniciado.
  - Após selecionar o local onde o projeto será criado, abrir a pasta no VSCode.
  - A estrutura dentro da pasta deixaremos o mais simples possível, criando apenas uma pasta chamada "src" dentro da pasta principal.

- Navegar até a pasta “src” que foi criada usando o terminal.
  - Iniciar o npm:
    - Para iniciar o npm, execute o comando npm init
    - Este comando irá te guiar para criar o arquivo package.json
    - Podemos criar usando todas as opções padrões, e depois alterar diretamente o arquivo, se necessário.

III. Instalando Pacotes

Instalando Pacotes

- Agora é possível instalar pacotes necessários usando o npm para o projeto. Para teste, instale o pacote chamado "express".
    - Execute o comando npm install express
    - Após executar o comando, verifique o que foi alterado no projeto:
        - O arquivo package.json agora possui um objeto chamado "dependencies", com o nome do pacote e versão instalado.
        - Foi criada uma pasta chamada "node_modules", contendo todos os arquivos necessários para que o pacote que instalado funcione.

IV. Criando a Primeira Rota

Criando a Primeira Rota

- Para testar se o pacote foi corretamente baixado, crie um arquivo .html na raiz da pasta chamado chamado index.html

index.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport"
content="width=device-width,
initial-scale=1.0">

        <title>Hello World Simple App</title>
    </head>
    <body>
        <div>Hello World!</div>
    </body>
</html>
```

Criando a Primeira Rota

- Agora, crie um arquivo .js chamado server.js
  - Adicione a dependência do express que foi instalado.
  - Crie uma instância do objeto Express.
  - Defina a porta que será usada. Neste exemplo, será usada a 5000.
  - O express dá acesso a diversas funções para roteamento. Dentre elas, a get(), uma das mais comuns, dispara uma requisição GET.

server.js

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

const app = express();
const port = 5000;
```

Criando a Primeira Rota

server.js

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

const app = express();
const port = 5000;

app.get('/', (req, res) => {
  res.sendFile('index.html', {root: __dirname});
});
```

- O express dá acesso a diversas funções para roteamento. Dentre elas, a get(), uma das mais comuns, dispara uma requisição GET.

- Essa função get recebe dois argumentos:
  - ‘/’ é o caminho
  - Uma função de callback que será disparada.

Criando a Primeira Rota

- Por último, chame a função listen() do express, para começar a escutar por requisição na porta e rota criados.

server.js

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

const app = express();
const port = 5000;

app.get('/', (req, res) => {
  res.sendFile('index.html', {root: __dirname});
});

app.listen(port, () => {
  console.log(`Now listening on port ${port}`);
});
```

Iniciando!

Com o código necessário criado para iniciar o projeto, no terminal, rode o comando node para começar a escutar por requisições.

node server.js

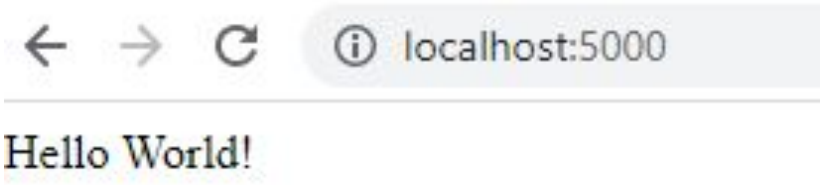
Para facilitar, abra o package.json e crie um novo script chamado "start", e coloque o comando usado anteriormente.

Execute o comando abaixo para iniciar:

npm start

Iniciando!

Abra o navegador e acesse: localhost:<port number>



No exemplo acima, localhost:5000

Isso é o suficiente para iniciar um projeto básico em Node.js!

IV. Criando um CRUD

Criando um CRUD

C - Create
R - Retrieve
U - Update
D - Delete

Para os próximos passos do CRUD, mais alguns pacotes vão ser necessários.

Estes pacotes vão permitir armazenar as informações em uma base de dados, e também auxiliar nas requisições, facilitando a forma que as informações vão ser enviadas no body de cada requisição.

Os pacotes necessários são:

npm install sequelize sqlite3

npm install body-parser

CRUD - Database

- Crie uma pasta chamada “database” dentro da pasta “src”.
- Dentro da “database”, crie um arquivo chamado “db.js”.
- Neste arquivo, carregar a dependência do sequelize.

db.js

```
const sequelize = require('sequelize');
```

CRUD - Database

- Crie uma nova instância do sequelize, passando ao seu construtor as configurações necessárias.

- Dentro da pasta “database”, crie uma pasta chamada “storage”, onde será salvo o arquivo.

- Por fim, exporte a instância criada.

db.js

```
const sequelize = require('sequelize');

const database = new sequelize({
  dialect: 'sqlite',
  storage: './database/storage/database.sqlite'
});

module.exports = database;
```

CRUD - Tables

- Crie agora uma pasta “tables” dentro de “database”. Aqui dentro será salvo o arquivo de cada tabela que deve existir no banco.

- Neste caso, crie apenas um arquivo chamado “programmer.js”, pois será a única tabela necessária.

CRUD - Tables

programmer.js

- Comece importando os arquivos necessários, no caso, o sequelize e o db.js.

- Depois, defina as colunas que vão existir na tabela, indicando o seu nome, tipo de dados, entre outras informações. Neste exemplo, vamos precisar apenas de id, nome e as linguagens que o programador conhece.

- No final, exporte a tabela criada.

```
const sequelize = require('sequelize');
const database = require('../db');

const programmer = database.define('programmer', {
  id: {
    type: sequelize.INTEGER,
    autoIncrement: true,
    allowNull: false,
    primaryKey: true
  },
  name: {
    type: sequelize.STRING,
    allowNull: false
  },
  python: {
    type: sequelize.BOOLEAN,
    allowNull: false
  },
  javascript: {
    type: sequelize.BOOLEAN,
    allowNull: false
  },
  java: {
    type: sequelize.BOOLEAN,
    allowNull: false
  }
});

module.exports = programmer;
```

CRUD - Body

- Voltando ao server.js, antes de prosseguir com a criação das rotas, é necessário importar o body-parser instalado.

- Após iniciar o express(), adicione o seguinte trecho para que ele consiga suportar JSON no body e receber as informações necessárias para as rotas que serão criadas.

server.js

```
const express = require('express');
const bodyParser = require('body-parser');
const programmer = require('./database/tables/programmer');

const app = express();
const port = 5000;

app.use(bodyParser.json());

app.listen(port, () => {
  console.log(`Now listening on port ${port}`);
});
```

CRUD - DB Sync

- A primeira rota a ser criada é do tipo GET, com o nome syncDatabase, responsável por fazer a sincronização da base de dados, caso o arquivo já de banco de dados já exista.

- O trecho ao lado cria a rota, importa o arquivo db.js e tenta chamar a função sync(). Por fim, envia a response a requisição informando o sucesso ou erro.

server.js

```
app.get('/syncDatabase', async (req, res) => {
  const database = require('./database/db');

  try {
    await database.sync();

    res.send(`Database succesfully sync'ed`);
  } catch (error) {
    res.send(error);
  }
});
```

CRUD - Create

- Para criar um novo registro, adicione a tabela “programmer.js” e depois pegue as informações do body da requisição.

- Antes de adicionar o novo programador, verifique que todos os campos foram informados na requisição, e use a função create().

- Ao final, responda a requisição com o novo programador criado.

server.js

```
app.post('/createProgrammer', async (req, res) => {
  try {
    const params = req.body;

    const properties = ['name', 'python', 'java', 'javascript'];

    const check = properties.every((property) => {
      return property in params;
    });

    if (!check) {
      const propStr = properties.join(', ');
      res.send(`All parameters needed to create a programmer must be sent: ${propStr}`);
      return;
    }


    const newProgrammer = await programmer.create({
      name: params.name,
      python: params.python,
      javascript: params.javascript,
      java: params.java
    });

    res.send(newProgrammer);
  } catch (error) {
    res.send(error);
  }
});
```

CRUD - Retrieve

- A busca de registros na base permite que seja ou não enviado um ID para filtrar.
- Esse id será enviado no body, e caso ele exista, a função findByPk() é invocada, recebendo o ID.
- Se não for enviado o ID, a função findAll() vai retornar todos os registros da base de dados.

server.js

```
app.get('/retrieveProgrammer', async (req, res) => {
  try {
    const params = req.query;

    if ('id' in params) {
      const record = await programmer.findByPk(params.id);

      if (record) {
        res.send(record);
      } else {
        res.send('No programmer found using received ID');
      }

      return;
    }

    const records = await programmer.findAll();

    res.send(records);
  } catch (error) {
    res.send(error);
  }
});
```

CRUD - Update

- Para atualizar é obrigatório que seja informado o ID do registro e que este ID exista na base de dados.

- Após as validações iniciais, atualizar as informações de acordo com o que foi recebido no body e chamar a função save().

- Ao final, informar que o registro foi corretamente atualizado.

server.js

```
app.put('/updateProgrammer', async (req, res) => {
  try {
    const params = req.body;

    if (!('id' in params)) {
      res.send(`Missing 'id' in request body`);
      return;
    }

    const record = await programmer.findByPk(params.id);

    if (!record) {
      res.send(`Programmer ID not found.`);
      return;
    }

    const properties = ['name', 'python', 'java', 'javascript'];

    const check = properties.some((property) => {
      return property in params;
    });

    if (!check) {
      const propStr = properties.join(', ');
      res.send(`Request body doesn't have any of the following properties: ${propStr}`);
      return;
    }

    record.name = params.name || record.name;
    record.python = params.python || record.python;
    record.java = params.java || record.java;
    record.javascript = params.javascript || record.javascript;

    await record.save();

    res.send(`${record.id} ${record.name} - Updated successfully`);
  } catch (error) {
    res.send(error);
  }
});
```

CRUD - Delete

- Para deletar, também é obrigatório receber um ID como parâmetro, e esse ID deve retornar um registro da base de dados.
- Após as validações, a função destroy() irá remover aquele registro.
- Por último, informar a requisição que o registro foi deletado.

server.js

```
app.delete('/deleteProgrammer', async (req, res) => {
  try {
    const params = req.body;

    if (!('id' in params)) {
      res.send(`Missing 'id' in request body`);
      return;
    }

    const record = await programmer.findByPk(params.id);

    if (!record) {
      res.send(`Programmer ID not found.`);
      return;
    }

    await record.destroy();

    res.send(`${record.id} ${record.name} - Deleted successfully`);
  } catch (error) {
    res.send(error);
  }
});
```

CRUD - Otimização validateId

- Tanto o Update quanto o Delete possuem um mesmo trecho de código, responsável por validar se um ID foi enviado, e se este ID consegue encontrar um registro na base de dados.

- Para reduzir algumas linhas de código através da reutilização, o correto seria extrair essa parte semelhante em uma função a parte, responsável por essa validação.

server.js

```
const validateID = async (params) => {
  try {
    if (!('id') in params) {
      throw `Missing 'id' in request body`;
    }

    const record = await programmer.findByPk(params.id);

    if (!record) {
      throw `Programmer ID not found.`;
    }

    return record;
  } catch (error) {
    throw error;
  }
}
```

CRUD - Otimização validateId

- Com a mudança anterior, a rota “updateProgrammer” ficaria da seguinte forma, fazendo uso da nova função de validação de ID.

server.js

```
app.put('/updateProgrammer', async (req, res) => {
  try {
    const params = req.body;

    const record = await validateID(params);

    const properties = ['name', 'python', 'java', 'javascript'];

    const check = properties.some((property) => {
      return property in params;
    });

    if (!check) {
      res.send(`Request body doesn't have any of the following properties:
${properties.join(', ')}`);
      return;
    }

    record.name = params.name || record.name;
    record.python = params.python || record.python;
    record.java = params.java || record.java;
    record.javascript = params.javascript || record.javascript;

    await record.save();

    res.send(`${record.id} ${record.name} - Updated successfully`);
  } catch (error) {
    res.send(error);
  }
});
```

CRUD - Otimização validateId

- E o mesmo trecho deve ser alterado na rota “deleteProgrammer”.

- Agora, caso alguma nova regra de validação precise ser inserida, ou alguma regra já existente precise ser alterada, basta fazer a alteração em apenas um lugar.

server.js

```
app.delete('/deleteProgrammer', async (req, res) =>
{
  try {
    const params = req.body;

    const record = await validateID(params);

    record.destroy();

    res.send(`${record.id} ${record.name} - Deleted
successfully`);
  } catch (error) {
    res.send(error);
  }
});
```

CRUD - Otimização validateProperties

- Outra otimização possível é o trecho que faz a validação das propriedades que estão sendo recebidas no body da requisição.

- Neste exemplo, existe uma variação apenas da função que é chamada para varrer o array, onde em um exemplo é a “some” e no outro é a “every”

server.js

```
const validateProperties = (properties, params, fn) => {
  try {
    const check = properties[fn]((property) => {
      return property in params;
    });

    if (!check) {
      const propStr = properties.join(', ');
      throw `Request body doesn't have any of the following
properties: ${propStr}`;
    }

    return true;
  } catch (error) {
    throw error;
  }
}
```

CRUD - Otimização validateProperties

- O primeiro lugar que é necessário alterar é na rota createProgrammer, fazendo a substituição do trecho que antes fazia a validação dos parâmetros.

- A função usada para varrer o array é a “every”, e deve ser informada na chamada da função validateProperties.

server.js

```
app.post('/createProgrammer', async (req, res) => {
  try {
    const params = req.body;

    const properties = ['name', 'python', 'java', 'javascript'];

    validateProperties(properties, params, 'every');

    const newProgrammer = await programmer.create({
      name: params.name,
      python: params.python,
      javascript: params.javascript,
      java: params.java
    });

    res.send(newProgrammer);
  } catch (error) {
    res.send(error);
  }
});
```

CRUD - Otimização validateProperties

- Por último, a rota updateProgrammer também fazia a validação dos parâmetros recebidos no body, com a diferença que aqui a função chamada é a “some”.

server.js

```
app.put('/updateProgrammer', async (req, res) => {
  try {
    const params = req.body;

    const record = await validateID(params);

    const properties = ['name', 'python', 'java', 'javascript'];

    validateProperties(properties, params, 'some');

    record.name = params.name || record.name;
    record.python = params.python || record.python;
    record.java = params.java || record.java;
    record.javascript = params.javascript || record.javascript;

    await record.save();

    res.send(`${record.id} ${record.name} - Updated successfully`);
  } catch (error) {
    res.send(error);
  }
});
```

Dúvidas