

Desenvolvendo um projeto com Node.js, Express, MySQL, Sequelize, Bootstrap e EJS

🛒 Products		
<input type="text" value="Search product"/>		<input type="button" value="+ New product"/>
Description	Quantity	Actions
PLAYSTATION 1	1	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
PS VITA 1000	6	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
PS5	1	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
PS4 PRO 1TB	3	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
PS4 SLIM 500GB	4	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
PLAYSTATION 3 500GB SUPER SLIM	8	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
PS VITA 2000	7	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
PS2 SLIM	5	<input type="button" value="Edit"/> <input type="button" value="Delete"/>
PS2 CLASSIC	2	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

mpsxdeveloper
<https://github.com/mpsxdeveloper>

Introdução

Nesse tutorial vamos desenvolver um simples projeto para a web com o padrão MVC.

Utilizaremos as tecnologias Node.js com Express, banco de dados com MySQL e o ORM Sequelize, Bootstrap e EJS para lidar com a exibição das informações.

É necessário um prévio conhecimento básico das tecnologias citadas anteriormente para um melhor acompanhamento desse tutorial. Utilizamos o VSCode na versão para o Sistema Windows para o desenvolvimento do sistema explicado nesse tutorial.

Criação do projeto e instalação das dependências

Primeiramente, vamos criar nosso banco de dados e tabela de produtos. Embora o ORM Sequelize possa fazer esse trabalho, criaremos essa estrutura por conta própria, o que vai facilitar algumas coisas adiante. Sendo assim, utilize o script SQL a seguir para criar o banco e tabela no MySQL:

```
CREATE DATABASE products_db;  
USE products_db;  
  
CREATE TABLE products (  
  id INT NOT NULL AUTO_INCREMENT,  
  description VARCHAR(30) NOT NULL,  
  quantity INT NOT NULL,  
  PRIMARY KEY(id)  
);
```

Podemos agora iniciar a criação de nosso projeto. Crie uma pasta chamada list-products-node, e acesse essa pasta através do editor VSCode.

Abra um novo terminal, e digite `npm init -y` (“npm init -y” fará que um novo projeto Node seja criado com as perguntas preenchidas com uma resposta padrão).

Nossa próxima etapa é instalar as dependências necessárias ao projeto, para isso digite os seguintes comandos no terminal:

```
npm i express  
npm i sequelize  
npm i mysql2
```

```
npm i bootstrap@5.3.0-alpha1
npm i bootstrap-icons
npm i ejs
```

Agora que temos tudo o que precisamos para esse tutorial (note que uma pasta chamada `node_modules` foi adicionada ao projeto), vamos iniciar algumas configurações. Verifique que no arquivo **package.json**, as configurações do projeto mostram as dependências instaladas e suas versões, e também outros detalhes importantes. A linha `"main": "index.js"`, informa que esse é o arquivo que inicia a execução de nosso projeto. Precisamos criar então esse arquivo, na raiz do projeto, crie um arquivo chamado `index.js`, mas antes de editarmos esse arquivo, ainda em `package.json`, verifique a linha `"test": "echo \"Error: no test specified\" && exit 1"`. Altere essa linha para que fique igual a `"start": "node index"`. Ao rodarmos esse script chamado **start**, o node vai executar o arquivo de entrada de nossa aplicação, que é o `index.js` (não foi necessário informar a extensão `js` em `package.json`).

Podemos testar se tudo está funcionando ao rodarmos o comando `npm run start`, e não exibirá nenhum erro no terminal, mas também não vai acontecer nada demais (ainda não escrevemos nada em `index.js`).

Voltando para nossa estrutura de pastas no VSCode, crie na raiz do projeto uma pasta chamada `database` e dentro dessa pasta `database`, crie um arquivo chamado `config.js` e insira o código a seguir:

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

const connection = new Sequelize('products_db', 'root', 'senha', {
  host: 'localhost',
  dialect: 'mysql',
  logging: false
})

module.exports = connection
```

Com isso temos uma nova instância do Sequelize com uma conexão ao banco de dados que criamos anteriormente. Altere os valores `'root'` e `'senha'` para o usuário e senha que você utiliza na sua instalação do MySQL. Se seu banco não está rodando em uma máquina local, altere o `host: 'localhost'` para que reflita sua configuração. Com a conexão exportada, podemos criar um Model para nossa tabela produtos. Crie uma pasta chamada `models` e dentro dessa pasta crie um arquivo chamado `Product.js`, e digite o código a seguir nele:

```

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

const Product = sequelize.define('products', {
  id: {
    type: Sequelize.INTEGER,
    primaryKey: true,
    autoIncrement: true,
    allowNull: false
  },
  description: {
    type: Sequelize.STRING,
    allowNull: false
  },
  quantity: {
    type: Sequelize.INTEGER,
    allowNull: false
  },
}, {
  timestamps: false
})

module.exports = Product

```

Esse model Product reflete a estrutura da tabela produtos, que deve ser levada em consideração quando formos realizar operações na tabela, como salvar, editar, etc. Os tipos de dado (INTEGER, STRING) e permitir nulo (allowNull: false) são regras que devem ser obedecidas ou ocorrerá falhas na hora de lidarmos com as informações no banco de dados. Note o parâmetro de configuração timestamps: false, pois por padrão o Sequelize cria na tabela colunas created_at e updated_at, que são colunas para datas, utilizadas quando um item da tabela é salvo ou atualizado, respectivamente. Quando criamos nossa tabela (sem utilizar o Sequelize), não criamos esses campos (não serão necessários nesse tutorial), portanto devemos informar ao Sequelize que não os considere na hora de lidar com nossa tabela.

Podemos agora criar o controlador do nosso aplicativo, para isso crie uma pasta chamada *controllers* e nessa pasta crie um arquivo chamado *productController.js* e preencha com o código a seguir:

```

const { Sequelize } = require('sequelize')
const Product = require('../models/Product')

const product_create = (req, res) => {
  res.render('create')
}

const product_list = async (req, res) => {

  const slug = req.params.slug

  await Product.findAll({
    order: [
      ['id', 'DESC']
    ],
    limit: 10
  })
  .then((data) => {
    res.render('home', { products: data })
  })
  .catch((error) => { console.log(error) })

}

const product_save = async (req, res) => {

  const { description, quantity } = req.body
  await Product.create({ description, quantity })
  .then((data) => {
    res.send(data)
  })
  .catch((error) => console.log('error creating product' + error))

}

const product_read = async (req, res) => {

  await Product.findOne({
    where: {
      id: req.params.id
    }
  })
  .then((data) => {
    res.render('edit', { product: data })
  })
}

```

```

    })
    .catch((error) => { console.log(error) })

}

const product_update = async (req, res) => {

    const { id, description, quantity } = req.body

    await Product.update(
        { description, quantity },
        { where: { id } }
    )
    .then((data) => {
        res.send(data)
    })
    .catch((error) => { console.log(error) })

}

const product_delete = async (req, res) => {

    await Product.destroy({
        where: { id: req.params.id }
    })
    .then((data) => {
        res.redirect('/')
    })
    .catch((error) => { console.log(error) })

}

const product_search = async (req, res) => {

    const slug = req.params.slug
    const Op = Sequelize.Op

    await Product.findAll({
        where: {
            description: {
                [Op.like]: '%' + slug + '%'
            }
        }
    })
}

```

```

.then((data) => {
  if(data.length > 0) {
    res.send({ products: data })
  }
  else {
    res.send({ products: null })
  }
})
.catch((error) => console.log(error))

}

module.exports = {
  product_list,
  product_create,
  product_save,
  product_read,
  product_update,
  product_delete,
  product_search
}

```

O controlador `productController` realiza todas as operações no `model Product` que foi importado no começo do arquivo. O controlador recebe através das rotas os `objects req e res` (`Request e Response`) e também tem a tarefa de redirecionar para uma página específica dependendo do conteúdo passado pelas rotas. Quando o usuário clicar em um botão para registrar um novo produto, o método `product_create` será invocado, e tudo o que ele faz é redirecionar para a página de criação de um produto com `res.render('create')`.

O método `product_list` listará os 10 últimos produtos registrados em ordem descendente com base no `id` do produto. Note que aqui em `res.render('home', { products: data })`, a página `home` será chamada e passamos como parâmetro a lista de produtos `data`, que é representada pelo parâmetro `products`. Esse parâmetro `products` será utilizado pela engine `EJS` na página `home` para fazer a listagem dos produtos.

Em `product_save`, temos uma situação diferente. Em vez de redirecionar para uma página, estamos apenas retornando o resultado da operação de salvar um produto em `res.send(data)`. Isso significa estamos apenas retornando o resultado do processo para a mesma página que invocou esse método (página `create`), e o retorno são as próprias informações do produto que acabou de ser salvo no banco de dados. Então a página vai decidir o que fazer com essa informação (nesse exemplo, veremos que a página `create` apenas redirecionará para a página `home` em caso de sucesso).

Seguindo esse mesmo raciocínio, em `product_search`, retornamos os produtos que correspondem com a pesquisa feita na página home. Esses produtos retornados deverão ser atualizados diretamente na página home sem recarregar a página, o que significa que a tabela que estiver sendo exibida, deverá ser refeita para mostrar somente os resultados da pesquisa.

Agora crie uma pasta chamada `routes`, e dentro dessa pasta crie um arquivo chamado `productRoutes.js`, preenchendo com o código a seguir:

```
const express = require('express')
const productController = require('../controllers/productController')

const router = express.Router()

// list
router.get("/", productController.product_list)
// create
router.get('/product', productController.product_create)
// save
router.post('/product', productController.product_save)
// read
router.get('/product/:id', productController.product_read)
// update
router.put('/product', productController.product_update)
// delete
router.get('/product/delete/:id', productController.product_delete)
// search
router.get('/product/search/:slug', productController.product_search)

module.exports = {
  router
}
```

Cada requisição do usuário é mapeada por uma rota, que invoca o método correspondente do controlador (o controlador foi importado no início do arquivo). Quando o usuário acessa a raiz do projeto (`http://localhost:3000/`) o código `router.get("/", productController.product_list)` pega essa requisição e chama o método `product_list` do `productController`. Vimos anteriormente, que esse método no controlador apenas redireciona para a página home, com a listagem dos últimos 10 produtos registrados.

Em `router.get('/product', productController.product_create)` e `router.post('/product', productController.product_save)` a rota possui a mesma chamada através do endpoint `‘/product’`, mas os métodos HTTP invocados são diferentes GET e POST, portanto não há nenhum problema.

Observe que em `router.get('/product/:id', productController.product_read)`, além da URL, esse endpoint também recebe um valor `:id`, esse valor será passado dinamicamente através da aplicação. Através desse `id` o método do controlador poderá buscar no banco as informações do produto que possui esse `id`.

Agora podemos configurar nosso arquivo `index.js`. Eis o código a seguir:

```
const express = require('express')
const path = require('path')
const { router } = require('./routes/productRoutes')

const app = express()
const PORT = 3000

app.use('/css', express.static(path.join(__dirname, "node_modules/bootstrap/dist/css")))
app.use('/font', express.static(path.join(__dirname, "node_modules/bootstrap-icons/font")))
app.use(express.json())
app.use(express.urlencoded({extended: true}))
app.use(router)

app.set('view engine', 'ejs')

app.listen(PORT, () => {
  console.log('Server started...')
})
```

Importamos o `express`, o arquivo com as rotas e `path` (para ajudar a montar caminhos para os assets necessários para nossa aplicação). Em `app.use('/css', express.static(path.join(__dirname, "node_modules/bootstrap/dist/css")))` estamos informando o caminho para utilizar o `css` do `bootstrap`, e na linha abaixo, fazemos o mesmo, dessa vez para as utilizar os ícones do `bootstrap`.

Em `app.use(express.json())` estamos configurando o `express` para fazer `parse` do `json` vindo das requisições. Com `app.use(express.urlencoded({extended: true}))` o `express` vai fazer o `parse` do corpo das requisições via formulário.

Em seguida, configuramos o `express` para usar o arquivo de rotas que criamos com `app.use(router)`.

Na linha `app.set('view engine', 'ejs')` estamos configurando o express para utilizar a engine EJS. Por padrão, o EJS vai procurar pelas páginas na pasta `views` na raiz do projeto. Se nossas páginas estiverem em uma pasta com outro nome, ou outro caminho, precisaríamos configurar o caminho dessa pasta.

Finalmente, em `app.listen(PORT, () => { console.log('Server started...') })` o express vai iniciar o servidor e começar a escutar requisições através da porta que configuramos antes e exibe uma mensagem no console ao iniciar.

Para concluir, podemos nos concentrar na camada view do MVC, criando as páginas do nosso aplicativo. Para isso, crie uma pasta chamada `views` e crie 3 arquivos dentro dessa pasta, que são: `create.ejs`, `edit.ejs` e `home.ejs`. Começando com o arquivo `create.ejs`, a seguir temos o código necessário para seu funcionamento (antes de copiar o código adiante, você pode digitar ! na página vazia e o editor vai sugerir criar um template html para você, isso vai facilitar muito a produtividade):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="/css/bootstrap.min.css" />
  <link rel="stylesheet" href="/font/bootstrap-icons.css" />
  <title>Create</title>
</head>

<body>

  <a href="/">
    <div class="container w-50 bg-primary p-3 mt-2">
      <h1 class="text-center text-light"><i class="bi bi-cart4"></i> Products</h1>
    </div>
  </a>

  <div class="container mt-5 w-50">
    <form>
      <legend class="text-center border border-primary border-2 rounded mb-5">
        Create a product
      </legend>
      <div class="mb-3">
        <label for="description" class="form-label">Description</label>
```

```

        <input type="text" class="form-control" name="description" placeholder="Describe
the product" maxlength="30" required />
    </div>
    <div class="mb-3">
        <label for="quantity" class="form-label">Quantity</label>
        <input type="number" class="form-control" name="quantity" placeholder="Quantity of
product" required />
    </div>
    <a href="/" class="btn btn-danger float-start mt-3">
        <i class="bi bi-x-circle"></i> Cancel Save
    </a>
    <button type="submit" class="btn btn-primary float-end mt-3">
        <i class="bi bi-device-hdd"></i> Save Product
    </button>
</form>
</div>

```

```
<script>
```

```

const form = document.querySelector('form')

form.addEventListener('submit', async (e) => {
    e.preventDefault()
    const description = form.description.value.trim()
    const quantity = form.quantity.value.trim()
    if(description === "" || quantity === "") {
        return
    }
    try {
        const res = await fetch('/product', {
            method: 'POST',
            body: JSON.stringify({ description, quantity }),
            headers: { 'Content-Type': 'application/json' }
        })
        const data = await res.json()
        if(data) {
            location.assign('/')
        }
    }
    catch (error) {
        console.log(error)
    }
})

```

```
</script>
```

```
</body>
```

```
</html>
```

Nessa página, temos um simples formulário para preencher a descrição e quantidade do produto. O código javascript utiliza fetch para fazer uma requisição assíncrona, note que os parâmetros são '/product' como URL, método POST, e no corpo da requisição passamos os dados do formulário convertidos para um JSON em formato de String. O método é POST e a URL é '/product', e o sistema de rotas possui uma rota "router.post('/product', productController.product_save)", então esses dados serão tratados pela rota e enviados ao método product_save do productController. Se o retorno for o que método fetch está esperando, o redirecionamento é feito para a página home, que nada mais é que uma nova chamada para outra rota.

Agora crie uma página chamada *edit.js* e insira o código abaixo:

```
<!DOCTYPE html>
```

```
<html lang="en">
```

```
<head>
```

```
  <meta charset="UTF-8">
```

```
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <link rel="stylesheet" href="/css/bootstrap.min.css" />
```

```
  <link rel="stylesheet" href="/font/bootstrap-icons.css" />
```

```
  <title>Edit</title>
```

```
</head>
```

```
<body>
```

```
  <a href="/">
```

```
    <div class="container w-50 bg-primary p-3 mt-2">
```

```
      <h1 class="text-center text-light"><i class="bi bi-cart4"></i> Products</h1>
```

```
    </div>
```

```
  </a>
```

```
  <div class="container mt-5 w-50">
```

```
    <form>
```

```
      <legend class="text-center border border-primary border-2 rounded mb-5">
```

```
        Edit a product
```

```

</legend>
<input type="hidden" name="id" value="<%= product.id %>" />
<div class="mb-3">
  <label for="description" class="form-label">Description</label>
  <input type="text" class="form-control" name="description" placeholder="Describe
the product" value="<%= product.description %>" maxlength="30" required />
</div>
<div class="mb-3">
  <label for="quantity" class="form-label">Quantity</label>
  <input type="number" class="form-control" name="quantity" placeholder="Quantity of
product" value="<%= product.quantity %>" required />
</div>
<a href="/" class="btn btn-danger float-start mt-3">
  <i class="bi bi-x-circle"></i> Cancel Edit
</a>
<button type="submit" class="btn btn-primary float-end mt-3">
  <i class="bi bi-pencil"></i> Edit Product
</button>
</form>
</div>

```

```
<script>
```

```

const form = document.querySelector('form')

form.addEventListener('submit', async (e) => {
  e.preventDefault()
  const id = form.id.value
  const description = form.description.value
  const quantity = form.quantity.value
  try {
    const res = await fetch('/product', {
      method: 'PUT',
      body: JSON.stringify({ id, description, quantity }),
      headers: { 'Content-Type': 'application/json' }
    })
    const data = await res.json()
    if(data) {
      location.assign('/')
    }
  }
  catch (error) {
    console.log(error)
  }
}

```

```
} )
```

```
</script>
```

```
</body>
```

```
</html>
```

Nessa página podemos editar o produto através do formulário. Quando o usuário seleciona um produto na página home para editar, a rota leva ao controlador o id do produto, então o controlador recupera as informações desse produto específico e os envia para a página de edição.

Nos campos dos formulário, a engine EJS exibe as informações atuais do produto através das tags `<%= product.id %>`, `<%= product.description %>` e `<%= product.quantity %>`.

Concluindo, temos que fazer a página home, que exibe uma lista com os 10 últimos produtos registrados, exibe um campo para pesquisa de produtos, além da opção de selecionar um produto para edição ou exclusão. Crie uma página *home.ejs* dentro da pasta views, e deixe-o como no código a seguir:

```
<!DOCTYPE html>
```

```
<html lang="en">
```

```
<head>
```

```
  <meta charset="UTF-8">
```

```
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <link rel="stylesheet" href="/css/bootstrap.min.css" />
```

```
  <link rel="stylesheet" href="/font/bootstrap-icons.css" />
```

```
  <title>Home</title>
```

```
</head>
```

```
<body>
```

```
  <a href="/">
```

```
    <div class="container w-50 bg-primary p-3 mt-3">
```

```
      <h1 class="text-center text-light"><i class="bi bi-cart4"> </i> Products</h1>
```

```
    </div>
```

```
  </a>
```

```
  <div class="container w-50">
```

```
    <div class="row mb-3 mt-3">
```

```
      <div class="col-9">
```

```
        <div class="input-group">
```

```

        <input type="text" class="form-control" placeholder="Search product"
id="searchInput" />
        <button class="btn btn-outline-secondary" type="button" id="searchBtn">
            <i class="bi bi-search"></i>
        </button>
    </div>
</div>
<div class="col-3">
    <a href="/product" class="btn btn-success float-end">
        <i class="bi bi-plus"></i> New product
    </a>
</div>
</div>
<% if(products.length > 0) { %>
    <table class="table table-sm">
        <thead>
            <tr>
                <th scope="col">Description</th>
                <th scope="col">Quantity</th>
                <th scope="col">Actions</th>
            </tr>
        </thead>
        <tbody>
            <% products.forEach(function(product) { %>
                <tr>
                    <td><%= product.description %></td>
                    <td><%= product.quantity %></td>
                    <td colspan="2">
                        <a href="/product/<%= product.id %>" class="btn btn-sm btn-warning"><i
class="bi bi-pencil"></i> Edit</a>
                        <a href="/product/delete/<%= product.id %>" class="btn btn-sm btn-
danger"><i class="bi bi-trash"></i> Delete</a>
                    </td>
                </tr>
            <% }) %>
        </tbody>
    </table>
    <% } else { %>
        <div class="alert alert-danger mt-3 w-75 float-end fw-bold">There are not products
registered yet</div>
    <% } %>
</div>

<script>

```

```

const btn = document.querySelector('#searchBtn')

btn.addEventListener('click', async (e) => {

  const slug = document.querySelector('#searchInput').value

  if(slug.trim() === '') {
    return
  }

  try {
    const res = await fetch('/product/search/' + slug, {
      method: 'GET',
      headers: { 'Content-Type': 'application/json' }
    })
    const data = await res.json()
    const existingData = data
    if(data.products.length > 0) {
      const table = document.querySelector('table')
      const tbody = document.querySelector('tbody')
      table.innerHTML = '';
      let thead = table.insertRow()
      thead.insertCell().innerHTML = '<b><th>Description</th></b>'
      thead.insertCell().innerHTML = '<b><th>Quantity</th></b>'
      thead.insertCell().innerHTML = '<b><th colspan="2">Actions</th></b>'
      data.products.forEach(product => {
        const row = table.insertRow()
        const cellDescription = row.insertCell()
        cellDescription.innerHTML = product.description
        const cellQuantity = row.insertCell()
        cellQuantity.innerHTML = product.quantity
        const cellActions = row.insertCell()
        const linkEdit = document.createElement('a')
        linkEdit.href = `/product/${product.id}`
        linkEdit.classList.add('btn', 'btn-sm', 'btn-warning')
        linkEdit.innerHTML = `<i class="bi bi-pencil"></i> Edit`
        cellActions.appendChild(linkEdit)
        const linkDelete = document.createElement('a')
        linkDelete.href = `/product/delete/${product.id}`
        linkDelete.classList.add('btn', 'btn-sm', 'btn-danger', 'ms-1')
        linkDelete.innerHTML = `<i class="bi bi-trash"></i> Delete`
        cellActions.appendChild(linkDelete)
      })
    }
  }
}

```



```

    }
  }
  catch (error) {
    console.log(error)
  }

})

</script>

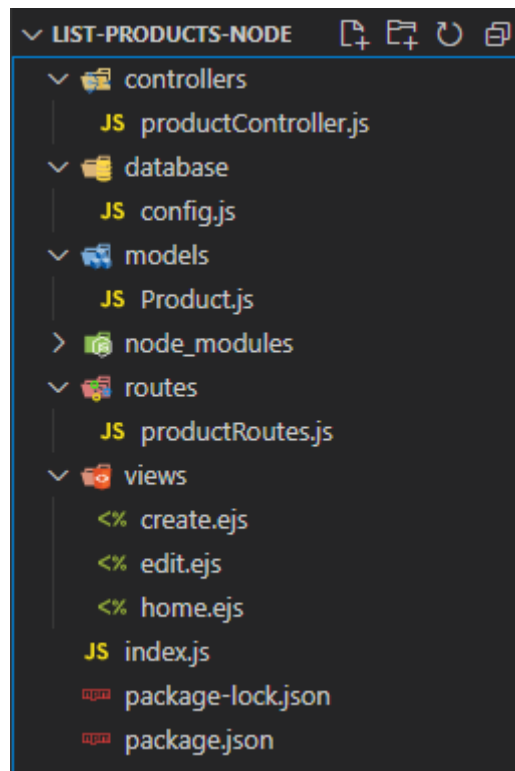
</body>

</html>

```

Nessa página temos a pesquisa por produtos e um botão para criar um produto (este apenas é uma rota `/product` que o controlador vai levar até a página `create.ejs`). Os últimos 10 produtos salvos serão exibidos na tabela, note a linha de código `<% if(products.length > 0) { %>` checando se existem produtos antes de exibir na tabela. Repare que aqui é uma estrutura condicional (if), por isso não se usa a tag `<%= %>`, que é para exibir valores. O loop para preencher a tabela, vem logo a seguir com `<% products.forEach(function(product) { %>`. Verifique as tags de fechamento das condicionais, por exemplo, em `<% } else { %>` usaremos para informar que não há produtos registrados no teste anterior (if). O loop `forEach` só vai acontecer se existirem produtos salvos no banco de dados. Devido ao id do produto ser único, podemos criar dinamicamente as rotas para editar e deletar cada produto dentro da tabela. O código para efetuar uma pesquisa é feito através de uma requisição ajax com `fetch` e se houver retorno de produtos nessa pesquisa, utilizamos javascript simples para reconstruir a tabela somente com os produtos que retornaram da pesquisa, mas mantendo as rotas dinâmicas para deletar e editar cada produto.

A estrutura do projeto ficou como na figura a seguir:



Agora que tudo foi configurado, podemos testar novamente o comando `npm run start` no terminal. Digite a URL `http://localhost:3000/` no navegador e faça os testes de criar, editar, deletar, pesquisar, etc.

Concluindo esse tutorial, uma sugestão de melhoria para o aplicativo desenvolvido seria agrupar trechos que se repetem em várias páginas em templates com a extensão `.ejs`, e utilizar o `include` do EJS `<%- include('exemplo.ejs'); %>` onde for necessário cada um desses templates, evitando assim repetição de código, facilitando a manutenção do aplicativo.

Repositório do projeto no Github: <https://github.com/mpsxdeveloper/list-products-node>