**Documentação Técnica da API de Controle de Inventário**

**Introdução**

Esta documentação técnica descreve a implementação de uma API RESTful para um sistema de controle de inventário de uma pequena loja.

A API permite adicionar, atualizar, deletar e listar produtos no inventário. Cada produto possui um nome, descrição, categoria, preço, quantidade em estoque e data de validade.

**Estrutura do Projeto**

A estrutura do projeto é organizada de forma a facilitar a manutenção e o desenvolvimento da API.

Abaixo está a estrutura do projeto, juntamente com uma breve descrição de cada pasta e arquivo:

bash

Copiar código

inventory-api/

├── config/ # Pasta para configuração do banco de dados

│ └── db.js # Arquivo de configuração da conexão com o banco de dados

├── controllers/ # Pasta para controladores

│ └── productsController.js # Arquivo de controladores para manipular as requisições relacionadas a produtos

├── routes/ # Pasta para definir as rotas da API

│ └── products.js # Arquivo de rotas para produtos

├── .env # Arquivo de variáveis de ambiente

├── server.js # Arquivo principal do servidor

└── package.json # Arquivo de configuração do npm, listando as dependências do projeto

**Configuração do Ambiente**

**Variáveis de Ambiente**

O arquivo .env na raiz do projeto armazena as variáveis de ambiente que são utilizadas para configurar a conexão com o banco de dados e a porta em que o servidor será executado:

DB\_HOST=localhost

DB\_USER=root

DB\_PASS=your\_password

DB\_NAME=inventory\_db

PORT=3000

**Dependências**

As dependências incluem bibliotecas para criar o servidor, conectar-se ao banco de dados, gerenciar variáveis de ambiente, lidar com CORS e analisar o corpo das requisições HTTP:

npm init -y

npm install express mysql2 dotenv cors body-parser

npm install nodemon --save-dev

**Banco de Dados**

**Estrutura da Tabela**

Para armazenar os produtos no inventário, precisamos criar uma tabela no banco de dados MySQL.

CREATE DATABASE inventory\_db;

USE inventory\_db;

CREATE TABLE products (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

description TEXT NOT NULL,

category VARCHAR(255) NOT NULL,

price DECIMAL(10, 2) NOT NULL,

stock INT NOT NULL,

expiry\_date DATE NOT NULL

);

**Configuração do Projeto**

**Configuração do Banco de Dados**

O arquivo config/db.js configura a conexão com o banco de dados MySQL utilizando as variáveis de ambiente definidas no arquivo .env:

**Arquivo config/db.js**:

const mysql = require('mysql2');

const dotenv = require('dotenv');

dotenv.config();

const db = mysql.createConnection({

host: process.env.DB\_HOST,

user: process.env.DB\_USER,

password: process.env.DB\_PASS,

database: process.env.DB\_NAME

});

db.connect((err) => {

if (err) {

console.error('Erro ao conectar ao banco de dados:', err);

return;

}

console.log('Conectado ao banco de dados MySQL');

});

module.exports = db;

**Controladores**

Os controladores contêm a lógica para manipular as requisições e respostas HTTP. Eles interagem com o banco de dados e retornam as respostas apropriadas para o cliente.

O código do arquivo controllers/productsController.js, contém funções para obter, adicionar, atualizar e deletar produtos:

**Arquivo controllers/productsController.js**:

const db = require('../config/db');

// Função para obter todos os produtos

const getAllProducts = (req, res) => {

db.query('SELECT \* FROM products', (err, results) => {

if (err) {

console.error('Erro ao obter produtos:', err);

res.status(500).send('Erro ao obter produtos');

return;

}

res.json(results);

});

};

// Função para adicionar um novo produto

const addProduct = (req, res) => {

const { name, description, category, price, stock, expiry\_date } = req.body;

// Verificar se o produto já existe

db.query(

'SELECT \* FROM products WHERE name = ? AND description = ? AND expiry\_date = ?',

[name, description, expiry\_date],

(err, results) => {

if (err) {

console.error('Erro ao verificar produto:', err);

res.status(500).send('Erro ao verificar produto');

return;

}

if (results.length > 0) {

res.status(400).send('Produto duplicado');

return;

}

// Se o produto não existe, insira-o no banco de dados

db.query(

'INSERT INTO products (name, description, category, price, stock, expiry\_date) VALUES (?, ?, ?, ?, ?, ?)',

[name, description, category, price, stock, expiry\_date],

(err, results) => {

if (err) {

console.error('Erro ao adicionar produto:', err);

res.status(500).send('Erro ao adicionar produto');

return;

}

res.status(201).send('Produto adicionado com sucesso');

}

);

}

);

};

// Função para atualizar um produto existente (substituição completa)

const updateProductPut = (req, res) => {

const { id } = req.params;

const { name, description, category, price, stock, expiry\_date } = req.body;

db.query(

'UPDATE products SET name = ?, description = ?, category = ?, price = ?, stock = ?, expiry\_date = ? WHERE id = ?',

[name, description, category, price, stock, expiry\_date, id],

(err, results) => {

if (err) {

console.error('Erro ao atualizar produto:', err);

res.status(500).send('Erro ao atualizar produto');

return;

}

if (results.affectedRows === 0) {

res.status(404).send('Produto não encontrado');

return;

}

res.send('Produto atualizado com sucesso');

}

);

};

// Função para atualizar um produto existente (atualização parcial)

const updateProductPatch = (req, res) => {

const { id } = req.params;

const fields = req.body;

const query = [];

const values = [];

for (const [key, value] of Object.entries(fields)) {

query.push(`${key} = ?`);

values.push(value);

}

values.push(id);

db.query(

`UPDATE products SET ${query.join(', ')} WHERE id = ?`,

values,

(err, results) => {

if (err) {

console.error('Erro ao atualizar produto:', err);

res.status(500).send('Erro ao atualizar produto');

return;

}

if (results.affectedRows === 0) {

res.status(404).send('Produto não encontrado');

return;

}

res.send('Produto atualizado com sucesso');

}

);

};

// Função para deletar um produto existente

const deleteProduct = (req, res) => {

const { id } = req.params;

db.query('DELETE FROM products WHERE id = ?', [id], (err, results) => {

if (err) {

console.error('Erro ao deletar produto:', err);

res.status(500).send('Erro ao deletar produto');

return;

}

if (results.affectedRows === 0) {

res.status(404).send('Produto não encontrado');

return;

}

res.send('Produto deletado com sucesso');

});

};

module.exports = {

getAllProducts,

addProduct,

updateProductPut,

updateProductPatch,

deleteProduct

};

**Rotas**

As rotas definem os endpoints da API e associam cada rota a uma função no controlador.

O arquivo routes/products.js contém as definições de rotas para obter, adicionar, atualizar e deletar produtos:

**Arquivo routes/products.js**:

const express = require('express'); // Importa o framework Express

const router = express.Router(); // Cria um novo roteador

const productsController = require('../controllers/productsController'); // Importa o controlador de produtos

// Definindo uma rota para obter todos os produtos

router.get('/', productsController.getAllProducts);

// Definindo uma rota para adicionar um novo produto

router.post('/', productsController.addProduct);

// Definindo uma rota para atualizar um produto existente (substituição completa)

router.put('/:id', productsController.updateProductPut);

// Definindo uma rota para atualizar um produto existente (atualização parcial)

router.patch('/:id', productsController.updateProductPatch);

// Definindo uma rota para deletar um produto existente

router.delete('/:id', productsController.deleteProduct);

// Exportando o roteador

module.exports = router;

**Servidor**

O arquivo server.js configura e inicializa o servidor Express. Ele também define as rotas para a API e inicia o servidor na porta especificada nas variáveis de ambiente:

**Arquivo server.js**:

const dotenv = require('dotenv'); // Importa o pacote dotenv para gerenciar variáveis de ambiente

// Carregar as Variáveis de Ambiente

dotenv.config(); // Carrega as variáveis definidas no arquivo .env para process.env

// Importar as Bibliotecas

const express = require('express'); // Importa o framework Express

const cors = require('cors'); // Importa o pacote cors para permitir requisições de diferentes origens

const bodyParser = require('body-parser'); // Importa o pacote body-parser para analisar o corpo das requisições HTTP

const db = require('./config/db'); // Importa a conexão com o banco de dados

// Inicializar nova aplicação Express

const app = express(); // Inicializa uma nova aplicação Express

// Configurar o CORS e o body-parser

app.use(cors()); // Habilita o CORS para todas as rotas

app.use(bodyParser.json()); // Configura o body-parser para analisar requisições JSON

// Importar as rotas de produtos

const productsRoutes = require('./routes/products'); // Importa as rotas de produtos

// Usar as rotas de produtos para todas as requisições que começam com /api/products

app.use('/api/products', productsRoutes); // Configura o servidor para usar as rotas de produtos

// Rota inicial para testar o servidor

app.get('/', (req, res) => {

res.send('Servidor está rodando'); // Define uma rota inicial para testar o servidor

});

// Configurar o servidor para uma porta específica

const PORT = process.env.PORT || 3000; // Define a porta a partir da variável de ambiente ou usa a porta 3000 como padrão

app.listen(PORT, () => {

console.log(`Servidor rodando na porta ${PORT}`); // Loga uma mensagem informando que o servidor está rodando

});

**Testes da API**

Para garantir que a API está funcionando corretamente, realizamos testes usando o Insomnia. Vamos testar cada endpoint da API e verificar como a API lida com diferentes tipos de requisições, incluindo casos de sucesso e casos de erro.

**Teste com Insomnia**

**1. Adicionar um Novo Produto**

**Requisição:**

* Método: POST
* URL: http://localhost:3000/api/products
* Corpo:

{

"name": "Produto A",

"description": "Descrição do Produto A",

"category": "Categoria A",

"price": 100.0,

"stock": 10,

"expiry\_date": "2023-12-31"

}

**Resposta de Sucesso:**

{

"message": "Produto adicionado com sucesso"

}

**Resposta de Produto Duplicado:**

{

"message": "Produto duplicado"

}

**2. Atualizar um Produto Existente (Substituição Completa)**

**Requisição:**

* Método: PUT
* URL: http://localhost:3000/api/products/1
* Corpo:

{

"name": "Produto A Atualizado",

"description": "Descrição do Produto A Atualizado",

"category": "Categoria A",

"price": 120.0,

"stock": 8,

"expiry\_date": "2023-11-30"

}

**Resposta de Sucesso:**

{

"message": "Produto atualizado com sucesso"

}

**Resposta de Produto Não Encontrado (404):**

{

"message": "Produto não encontrado"

}

**3. Atualizar um Produto Existente (Atualização Parcial)**

**Requisição:**

* Método: PATCH
* URL: http://localhost:3000/api/products/1
* Corpo:

{

"price": 110.0,

"stock": 9

}

**Resposta de Sucesso:**

{

"message": "Produto atualizado com sucesso"

}

**Resposta de Produto Não Encontrado (404):**

{

"message": "Produto não encontrado"

}

**4. Deletar um Produto Existente**

**Requisição:**

* Método: DELETE
* URL: http://localhost:3000/api/products/1

**Resposta de Sucesso:**

{

"message": "Produto deletado com sucesso"

}

**Resposta de Produto Não Encontrado (404):**

{

"message": "Produto não encontrado"

}

**5. Listar Todos os Produtos**

**Requisição:**

* Método: GET
* URL: http://localhost:3000/api/products

**Resposta de Sucesso:**

[

{

"id": 1,

"name": "Produto A",

"description": "Descrição do Produto A",

"category": "Categoria A",

"price": 100.0,

"stock": 10,

"expiry\_date": "2023-12-31"

},

{

"id": 2,

"name": "Produto B",

"description": "Descrição do Produto B",

"category": "Categoria B",

"price": 150.0,

"stock": 5,

"expiry\_date": "2024-01-15"

}

]

**Resposta de Erro Interno do Servidor (500):**

{

"message": "Erro ao obter produtos"

}

**Resolução de Problemas Comuns**

**Erro 404 (Not Found)**

* **Causa**: O recurso solicitado não foi encontrado.
* **Solução**: Verifique se o ID do produto existe no banco de dados e se a URL da requisição está correta.

**Erro 500 (Internal Server Error)**

* **Causa**: Ocorreu um erro no servidor ao processar a requisição.
* **Solução**: Verifique os logs do servidor para identificar a causa do erro. Certifique-se de que a conexão com o banco de dados está configurada corretamente e que a query SQL está correta.

**Considerações Finais**

Após testar todos os endpoints utilizando uma ferramenta como o Insomnia ou Postman, visando garantir que a API esteja funcionando corretamente, foram aplicados os conceitos de tratamento de erros e verificação de duplicidade para manter a integridade dos dados no banco de dados.