Documentação de API para e-commerce EssentialsDrawing

APIs (Interfaces de Programação de Aplicação) referem-se a softwares com determinadas funções, cuja interface interliga dois serviços de comunicação, solicitações do usuário (cliente) e respostas a ele (servidor). Mais especificamente será utilizada uma API do tipo REST, pois o cliente enviará dados ao servidor e este executará funções internas para retorná-los ao cliente.

APIs REST possuem um conjunto de funções essenciais para o controle de banco de dados desde o básico até o avançado, como o GET, POST, PUT, PATCH e DELETE. Os clientes podem utilizar essas funções para acessar dados no servidor através do HTTP, protocolos para comunicação entre eles, assim como URLs digitadas no navegador para visitar um site qualquer.

A seguinte documentação apresenta informações de como foi desenvolvida a estrutura de uma API para atender uma contratação de serviço da loja EssentialsDrawing para administrar a venda de imagens e pacotes de imagens geradas por IA (Inteligência Artificial), de modo que o administrador possa consultar, adicionar, atualizar e deletar os produtos, além de que será possível fazer registro, login, recuperação de senha, editar dados cadastrados, comprar imagens e/ou pacotes, visualizar pedidos e filtrar pesquisas por parte do cliente (usuário comum). Cada produto possuirá um código único, nome, sua descrição, preço e categoria.

A estrutura do projeto, pastas, arquivos e processos, organizada de maneira eficiente facilita o desenvolvimento e a manutenção posterior da API. Abaixo está a estrutura deste projeto com algumas informações importantes de pastas e arquivos utilizados para o desenvolvimento da API:

|  |  |
| --- | --- |
| ESSENTIALSDRAWINGS | Pasta raiz |
| ├── config | Pasta para a configuração do banco de dados |
| │ └── db.js | Arquivo para conexão com o banco de dados |
| ├── controllers | Pasta para configurar os controladores das rotas |
| │ ├── authControl.js | Arquivo de controladores de autenticação |
| │ ├── routesAdm.js | Arquivo de controladores de administrador |
| │ ├── routesCart.js | Arquivo de controladores do carrinho |
| │ └── routesUser.js | Arquivo de controladores do usuário comum |
| ├── middlewares | Pasta para configurar a autorização de acesso às rotas |
| │ └── authMiddleware.js | Arquivo de autorização de acesso às rotas |
| ├── routes | Pasta para as rotas da API |
| │ ├── auth.js | Arquivo de rotas de autenticação |
| │ └── routes.js | Arquivo das demais rotas (admin, cart e user) |
| ├── services | Pasta para configurar o envio de e-mails |
| │ └── emailService.js | Arquivo de envio de e-mail para recuperação de senha |
| ├── .env | Arquivo de variáveis de ambiente |
| ├── package-lock.json | Arquivo de configuração do npm |
| ├── package.json | Arquivo que contém a lista de módulos do Node |
| └── server.js | Arquivo de códigos do servidor |

Será utilizado a linguagem de programação em JavaScript, para isso deve ser necessário está instalado na máquina o Node.js e o seu arquivo de gerenciador de pacotes, o npm que permite instalar bibliotecas, frameworks e outras ferramentas para o projeto e que são disponibilizados para uso público.

Para iniciar o projeto primeiro deve-se instalar as dependências que incluem as bibliotecas que serão utilizadas durante os processos de análise de dados no servidor para responder ao cliente de maneira fluida e eficiente, o código abaixo refere-se às instalações.

npm init -y

npm install express cors body-parser dotenv mysql2

npm install nodemon –save-dev

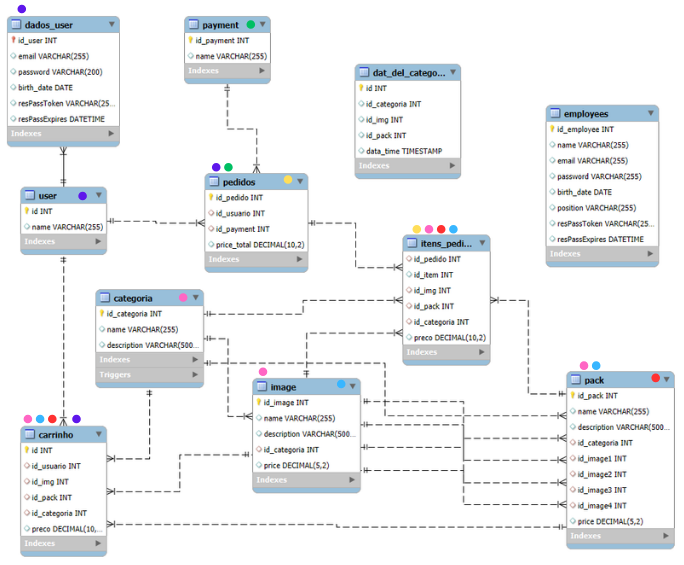
Essas dependências são: express, cors, body-parser, dotenv, nodemon e mysql2, uma framework para criar servidores web e APIs, um middleware que permite requisições externas ao servidor, outro middleware que analisa o corpo das requisições HTTP que podem ser acessadas em notação de objeto (.json), um módulo responsável por carregar as variáveis de ambiente, uma ferramenta de desenvolvimento que reinicia o servidor toda vez que um arquivo é modificado e um pacote que oferece recursos melhorados de autenticação à banco de dados, respectivamente.

Também deve ser criado o banco de dados, dados organizados e armazenados digitalmente em um sistema relacional com linhas e tabelas que poderão ser consultados, armazenados, modificados ou deletados, exatamente apropriado para a administração de um inventário de qualquer empresa.

Um SGBD (Sistema de Gerenciamento de Banco de Dados) compatível com as instalações feitas anteriormente é o MySQL que foi projetado e otimizado para aplicações web e pode ser executado em qualquer plataforma, possui segurança, é fácil integração, além de contar com opções de ajustes, backups e recuperações de dados. O MySQL trabalha com banco de dados relacionais casando com o projeto proposto de criação de API para controle de vendas.

Para armazenar os produtos (imagens e pacotes) da loja foi criado um banco de dados nomeado de **essentialsdrawing\_db** e várias tabelas, que serão necessárias para o funcionamento da loja, no MySQL com algumas características já citadas anteriormente, código único sendo uma chave primária para consultas específicas e interligações com outras tabelas, nome, descrição, categoria, preço de cada produto.

Abaixo está o diagrama EER com todas as tabelas do banco de dados. Para melhor entender foi colocado círculos coloridos nas tabelas, se o círculo estiver dentro, é uma tabela principal e fora é uma tabela que está recebendo dados da tabela principal, ou seja, a tabela payment possui o id e o nome do pagamento disponível para o usuário e a tabela pedidos vai ter o id do pagamento escolhido pelo usuário quando ele finalizar o pedido, portanto elas estão relacionadas entre si.



Com o banco de dados criado, agora devemos adicionar as variáveis de ambiente no arquivo .env, na raíz do projeto, que serão carregadas pelo dotenv e utilizadas na configuração da conexão com o banco de dados, ou seja, o .env permite que as variáveis sejam carregadas no process.env para uso do código e caso o banco de dados não conecte-se ao servidor, o erro pode está nos dados das variáveis de ambiente. Esses dados são sigilosos, pois possuem user e password do responsável pela loja. A seguir o código:

DB\_HOST = localhost

DB\_USER = root

DB\_PASS = \*\*\*\*\*\*

DB\_NAME = essentialsdrawing\_db

PORT = 5000

JWT\_SECRET = \*\*\*\*\*\*\*\*\*\*

EMAIL\_USER = essentialsdrawing@gmail.com

EMAIL\_PASS = \*\*\*\*\*\*\*\*\*\*

Antes de conectarmos ao banco de dados, precisamos iniciar o servidor. No arquivo server.js utilizaremos algumas dependências instaladas anteriormente, o express, cors, body-parser e dotenv. O express é responsável pela criação e gerenciamento do servidor da API, o dotenv vai gerenciar as variáveis de ambiente, cors vai permitir requisições externas e o body-parser analisa o corpo das requisições e permite que o servidor manipule os dados enviados pelo cliente.

Devemos importá-las e armazená-las em variáveis específicas e carregar as variáveis de ambiente do arquivo .env para que o código process.env possa permitir chamar as variáveis em qualquer parte do código inteiro.

const express = require('express');

const dotenv = require('dotenv');

const cors = require('cors');

const bodyParser = require('body-parser');

dotenv.config();

Depois inicializar a variável express em uma função para chama-lá também quando quisermos em outro lugar no código, usaremos esta função para usar o cors para habilitar todas as rotas da aplicação e o body-parser para analisar as requisições em formato JSON enviadas pelo cliente.

const app = express();

app.use(cors());

app.use(bodyParser.json());

Então definiremos a rota inicial para teste do servidor usando a aplicação com uma requisição e uma resposta juntamente com uma porta específica ou padrão para o servidor rodar.

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

res.send('Servidor OK!');

});

const port = process.env.port || 3000

app.listen(port, () => {

console.log(`Conectado ao servidor na porta ${port} 😊👇

Acesse 👉 http://localhost:${port}`)

})

Para que o código em server.js execute devemos adicioná-lo no arquivo package.json juntamente com o nodemon, então quando fizermos alterações no código, o servidor sempre irá iniciar automaticamente.

"scripts": {

"start": "nodemon server.js"

},

Com o comando **npm start** no terminal a seguinte mensagem deve aparecer: “Conectado ao servidor na porta 5000. Acesse http://localhost:5000”. Caso ocorra os erros: **ReferenceError** algo pode estar escrito de maneira incorreta, alguma variável, e **TypeError** alguma função pode também estar escrita de maneira errada.

Agora depois de o servidor está rodando iniciaremos a conexão com o banco de dados na pasta config, arquivo db.js com a importação da biblioteca mysql2, armazenando-a em uma variável, e o código para criar a conexão do servidor com o banco de dados criado anteriormente com as tabelas no MySQL, junto com o endereço do servidor do banco de dados, o nome e a senha do usuário responsável pelo banco e o nome deste.

const mysql = require('mysql2');

const\_db = mysql.createConnection({

host: process.env.DB\_HOST,

user: process.env.DB\_USER,

password: process.env.DB\_PASS,

database: process.env.DB\_NAME

});

Para iniciar a conexão, o seguinte código irá pegar a variável criada anteriormente e conectá-la apresentando duas respostas, uma caso dê erro e outra de sucesso, e para podermos usá-la em outros arquivos exportamo-la.

inventory\_db.connect((err) => {

if (err) {

return console.error('Erro na conexão com o banco de dados!', err);

}

console.log(`Conectado com sucesso ao banco de dados ${process.env.DB\_NAME}!`)

});

module.exports = inventory\_db;

Voltando no arquivo server.js adicionaremos o código para importar e utilizar a conexão com o banco de dados quando o servidor iniciar.

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

Inserindo novamente o comando **npm start** no terminal a seguinte mensagem deve aparecer: “Conectado ao servidor na porta 5000. Acesse <http://localhost:5000> Conectado com sucesso ao banco de dados essentialsdrawing\_db!”. Caso ocorra os erros: **ReferenceError** algo pode estar escrito de maneira incorreta, alguma variável, **TypeError** alguma função pode também estar escrita de maneira errada, **code: 'ETIMEDOUT'** o endereço host está incorreto, **code: 'ER\_ACCESS\_DENIED\_ERROR'** o nome ou a senha do usuário está incorreta, e **code: 'ER\_BAD\_DB\_ERROR'** o banco de dados não foi encontrado, o nome deve estar incorreto.

Com o servidor online e conectado ao banco de dados corretamente é possível fazer as operações ditas anteriormente, cadastro, login, resetar senha, consultar, adicionar, atualizar de modo parcial e deletar os produtos da loja. Para executar essas operações precisamos criar rotas, que são os caminhos definidos no código para responder as requisições HTTP, ou seja, cada rota possui uma função que só será executada se ela for acessada em determinada URL, e controladores, que são funções lógicas para manipular as requisições e respostas HTTP, portanto são responsáveis por processar os dados, interagir com o banco de dados e retornar as respostas solicitadas para o cliente. Alguns dos códigos de status dessas respostas são as seguintes para esta API:

* 200 OK: Requisição bem-sucedida.
* 201 Created: Recurso criado com sucesso.
* 400 Bad Request: Requisição inválida.
* 401 Unauthorized: Autenticação necessária.
* 404 Not Found: Recurso não encontrado.
* 500 Internal Server Error: Erro no servidor.

Esta API já vem pré-definida com um usuário do tipo admin cadastrado no banco de dados na tabela employees. Este admin poderá ter acesso ao sistema somente depois de redefinir sua senha, para maior segurança, pois assim ela estará criptografada com base nos códigos posteriores. Precisamos criar o código de autorização com requerimento de autenticação do usuário (comum e admin) para poder acessar todas as rotas disponíveis para cada tipo de usuário.

É necessário instalar as bibliotecas jsonwebtoken e bcrypt para criptografar senhas (no terminal) e uma chave secreta (JWT\_SECRET) que está no arquivo .env, para assinar tokens. O seguinte código executado no terminal permite obter essa chave:

npm install jsonwebtoken bcrypt

node -e "console.log(require('crypto').randomBytes(64).toString('hex'))"

Então no arquivo authMiddleware.js importamos a biblioteca jsonwebtoken e armazenamos em uma constante, criamos outra constante para uma função que intercepta as requisições antes de passar para as rotas protegidas. Nela obtemos o token no cabeçalho de autorização da requisição que vamos fazer, se não houver token vamos ter um erro (401) de acesso negado. Caso tenha, o token deve ser validado e verificado com a chave secreta, se ocorrer tudo certo o conteúdo ou ação é mostrado para o usuário, executando a rota, caso não, será retornado o erro (400) de token inválido. E por fim o código deve ser exportado para ser utilizado em outro lugar.

const jwt = require('jsonwebtoken')

const authMiddleware = (req, res, next) => {

const token = req.header('Authorization').replace('Bearer ', '')

if (!token) {

return res.status(401).send('Acesso negado. Nenhum token fornecido.')

}

try {

const decoded = jwt.verify(token, process.env.JWT\_SECRET)

req.user = decoded

next()

} catch (err) {

res.status(400).send('Token inválido.')

}

};

module.exports = authMiddleware

Depois disso configuramos o código para envio de e-mail de redefinição de senha. É preciso a biblioteca nodemailer que configura um transporte de e-mail utilizando o serviço do Gmail e define uma função para enviar e-mails.

npm install nodemailer

No arquivo emailService.js importamos, configuramos o transporte do e-mail com service(gmail) – serviço de e-mail, host – o servidor SMTP, port – porta padrão para conexões SMTP, secure – para garantir que a conexão seja feita através de SSL e auth (user e pass) – do arquivo .env. Criamos uma função para enviar o e-mail com o remetente, destinatário, assunto e conteúdo. Também criamos outra função para efetivamente enviar o e-mail com a função anterior e tratamento de erro e resposta, por fim há a exportação do código.

const nodemailer = require('nodemailer')

const transporter = nodemailer.createTransport({

service: 'gmail',

host: 'smtp.gmail.com',

port: 465,

secure: true,

auth: {

user: process.env.EMAIL\_USER,

pass: process.env.EMAIL\_PASS

}

})

const sendEmail = (to, subject, text) => {

const mailOptions = {

from: process.env.EMAIL\_USER,

to,

subject,

text

}

transporter.sendMail(mailOptions, (error, info) => {

if (error) {

return console.log('Erro ao enviar e-mail:', error);

}

console.log('E-mail enviado:', info.response)

})

}

module.exports = { sendEmail };

Agora poderemos criar os códigos das rotas dos usuários, primeiro as rotas de autenticação (registro, login, solicitação de redefinição de senha e a redefinição de senha). No arquivo authControl.js há importações de módulos (bibliotecas) do node.js: crypto – cria dados aleatórios para a criptografia, como o JWT, bcrypto – criação de hashes de senhas seguras e jsonwebtoken – verifica tokens para autorização e autenticação; e a importação do banco de dados e do serviço de e-mail.

const crypto = require('crypto')

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

const bcrypt = require('bcrypt')

const sendEmail = require('../services/emailService').sendEmail

const jwt = require('jsonwebtoken')

Criamos uma função assíncrona de cadastro do usuário, que vai pegar nome, e-mail, senha e data de aniversário do corpo da requisição e entra em um bloco try ... catch (para execução e tratamento de erro adequado) que verifica se já existe um usuário cadastrado com o e-mail fornecido, caso haja retorna o erro(400) de usuário já cadastrado, caso não, esperamos o bcrypt criptografar a senha fornecida 10x, para melhor segurança, esperamos por meio de uma operação assíncrona no banco de dados a inserção do nome do usuário na tabela user e os demais dados na tabela dados\_user e recebemos uma mensagem (201) de usuário registrado com sucesso. Caso dê algo errado será exibido o erro(500) onde possivelmente há um erro no código.

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

const { name, email, password, birth\_date } = req.body

try {

const [existingUser] = await db.promise().query('SELECT \* FROM dados\_user WHERE email = ?', [email])

if (existingUser.length > 0) {

return res.status(400).send('Usuário já registrado!')

}

const hashedPassword = await bcrypt.hash(password, 10)

await db.promise().query('INSERT INTO user (name) VALUES (?)', [name])

await db.promise().query(`INSERT INTO dados\_user (email, password, birth\_date)

VALUES (?, ?, ?)`, [email, hashedPassword, birth\_date])

res.status(201).send('Usuário registrado com sucesso!')

} catch (err) {

console.error('Erro ao registrar usuário:', err)

res.status(500).send('Erro ao registrar usuário.')

}

}

A função assíncrona login irá pegar o e-mail, senha e função(role) do corpo da requisição e entrar no bloco try ... catch, no bloco try há a constante query que vai ser diferente para cada role, se user(usuário comum) a query irá fazer uma consulta no banco de dados na tabela dados-user e pegar o e-email fornecido, caso o role seja admin, irá fazer a mesma coisa na tabela employees.

Com o resultado (e-mail) iremos esperar a consulta no banco de dados e verificar se ele existe ou não, se não retornará erro(400) credencial inválida, se sim segue para a comparação de senha no banco com a senha fornecida, se não forem iguais retornará o mesmo erro anterior, se sim o jsonwebtoken irá retornar um token para o primeiro resultado encontrado, sendo user ou admin com expiração em 1hr, caso ocorra algum erro o bloco catch será acionado com erro(500) erro de autenticação.

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

const { email, password, role } = req.body

try {

let query;

if (role === 'user') {

query = 'SELECT \* FROM dados\_user WHERE email = ?';

} else if (role === 'employee') {

query = 'SELECT \* FROM employees WHERE email = ?';

} else {

return res.status(400).send('Posição inválida');

}

const [result] = await db.promise().query(query, [email])

if (result.length === 0) {

return res.status(400).send('Credenciais Inválidas')

}

const isMatch = await bcrypt.compare(password, result[0].password)

if (!isMatch) {

return res.status(400).send('Credenciais Inválidas')

}

const token = jwt.sign({ userId: role === 'user' ? result[0].id : result[0].id\_employee},

process.env.JWT\_SECRET, { expiresIn: '1h' });

res.json({ token })

} catch (err) {

console.error('Erro na autenticação', err)

res.status(500).send('Erro na autenticação')

}

}

Na função assíncrona de solicitação de redefinição de senha somente o e-mail vai ser pego do corpo da requisição, no bloco try .. catch esperamos a consulta no banco de dados na tabela dados\_user (se o e-email for do user), se encontrar vai ser gerado um token hexadecimal de 40 caracteres com 20 bytes aleatórios, gerado uma data e hora de expiração do token (1hr), então esperamos o código atualizar a tabela com esse token e essa data, depois será enviado por e-mail o link de redefinição de senha e receberemos a mensagem de sucesso, esse processo também é o mesmo para o admin, a diferença está na tabela, que será a employees. Caso não seja encontrado o e-mail fornecido será enviado o erro(400) usuário não encontrado e caso dê algo errado o bloco catch retornará o erro(500) erro ao solicitar redefinição de senha.

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

const { email } = req.body;

try {

const [user] = await db.promise().query('SELECT \* FROM dados\_user WHERE email = ?', [email]);

if (user.length > 0) {

const token = crypto.randomBytes(20).toString('hex');

const expireDate = new Date(Date.now() + 3600000);

await db.promise().query('UPDATE dados\_user SET resPassToken = ?, resPassExpires = ? WHERE email = ?', [token, expireDate, email]);

const resetLink = `http://localhost:5000/reset-password/${token}`;

sendEmail(email, 'Recuperação de Senha', `Por favor, clique no link para redefinir sua senha: ${resetLink}`);

return res.send('E-mail de recuperação de senha enviado');

}

const [employee] = await db.promise().query('SELECT \* FROM employees WHERE email = ?', [email]);

if (employee.length > 0) {

const token = crypto.randomBytes(20).toString('hex');

const expireDate = new Date(Date.now() + 3600000);

await db.promise().query('UPDATE employees SET resPassToken = ?, resPassExpires = ? WHERE email = ?', [token, expireDate, email]);

const resetLink = `http://localhost:5000/reset-password/${token}`;

sendEmail(email, 'Recuperação de Senha', `Por favor, clique no link para redefinir sua senha: ${resetLink}`);

return res.send('E-mail de recuperação de senha enviado');

}

return res.status(404).send('Usuário não encontrado');

} catch (err) {

console.error('Erro ao solicitar redefinição de senha:', err);

return res.status(500).send('Erro ao solicitar redefinição de senha');

}

};

E por último a função assíncrona de redefinição de senha pega o token da função anterior e a nova senha do corpo da requisição, no bloco try .. catch esperamos a seleção da tabela dados\_user do token e da data maior que o momento da execução do código e compara-se se realmente é o token enviado pelo link, caso for a senha fornecida é criptografada com segurança reforçada (10x) e esperamos a atualização dos dados na tabela com uma resposta de sucesso, o mesmo ocorre caso seja o admin com diferença apenas na tabela employees. Caso não seja o token ou a data seja menor que a do momento será retornado o erro(400) de token inválido ou expirado, e se dê algo errado o bloco catch retorna o erro(500) erro ao redefinir senha.

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

const { token, newPassword } = req.body;

try {

const [user] = await db.promise().query('SELECT \* FROM dados\_user WHERE resPassToken = ? AND resPassExpires > NOW()', [token]);

if (user.length > 0) {

const hashedPassword = await bcrypt.hash(newPassword, 10);

await db.promise().query('UPDATE dados\_user SET password = ?, resPassToken = NULL, resPassExpires = NULL WHERE id\_user = ?', [hashedPassword, user[0].id\_user]);

return res.send('Senha redefinida com sucesso');

}

const [employee] = await db.promise().query('SELECT \* FROM employees WHERE resPassToken = ? AND resPassExpires > NOW()', [token]);

if (employee.length > 0) {

const hashedPassword = await bcrypt.hash(newPassword, 10);

await db.promise().query('UPDATE employees SET password = ?, resPassToken = NULL, resPassExpires = NULL WHERE id\_employee = ?', [hashedPassword, employee[0].id\_employee]);

return res.send('Senha redefinida com sucesso');

}

return res.status(400).send('Token inválido ou expirado');

} catch (err) {

console.error('Erro ao redefinir senha:', err);

return res.status(500).send('Erro ao redefinir senha');

}

};

Por fim exportamos todos os códigos para serem chamados nas rotas de acesso do usuário ou admin:

module.exports = {

registerUser,

login,

requestPasswordReset,

resetPassword

}

No arquivo routesAdm.js contém todos os códigos essenciais de administrador (os dados de registro e sua edição; criação, consulta, edição e deleção de categorias, pagamentos, imagens e pacotes de imagens; criação de pedidos, consulta deles; consulta de usuários; consulta de categorias deletadas; filtro de categorias, imagens e pacotes; e consulta de pesquisa por nome ou descrição de imagens e pacotes conjuntamente. É necessário a importação do banco de dados:

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

Na função de dados do admin será requerido da URL (parâmetro) o id do admin para a consulta no banco de dados na tabela employee, se retornar um resultado é porque existe o usuário admin então irá retornar o resultado em formato json com todos os dados do admin, caso não encontre o usuário, retornará o erro(400) usuário não encontrado e caso dê algum erro retornará o erro(500).

const dadosEmployee = (req, res) => {

const { id\_employee } = req.params;

db.query(`SELECT id\_employee, name, password, birth\_date, position

FROM employees WHERE id\_employee = ?`, [id\_employee], (err, results) => {

if (err) {

console.error('Erro ao obter seus dados.', err);

return res.status(500).send('Erro ao obter seus dados.');

}

if (results.length > 0) {

return res.json(results);

} else {

return res.status(404).send('Usuário não encontrado.');

}

});

};

Na função de editar dados do admin é preciso o id do admin da URL e os campos que serão atualizados do corpo da requisição. Uma matriz armazenará cada campo que será modificado e outra para cada valor desses juntamente com o id. Depois conforme esses dados haverá uma atualização na tabela employees onde o id do admin for correspondente com o da tabela, se sim, receberemos uma resposta de sucesso na edição, caso o id for diferente a resposta será erro(404) de usuário não encontrado e caso ocorra algum erro, erro(500) no servidor.

const editDatasEmployee = (req, res) => {

const { id\_employee } = 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\_employee);

db.query(`UPDATE employees SET ${query.join(', ')} WHERE id\_employee = ?`, values,

(err, results) => {

if (err) {

console.error('Erro ao atualizar os dados', err);

return res.status(500).send('Erro ao atualizar os dados');

}

if (results.affectedRows === 0) {

return res.status(404).send('Usuário não encontrado');

} else {

return res.send('Dados atualizados com sucesso!');

}

}

);

};

A função de adicionar uma categoria precisa de nome e descrição do corpo da requisição para primeiro consultar se já existe uma categoria igual registrada ou não, se sim retornará erro(400) já registrada e caso dê algum erro na consulta o erro(500), se não será inserido na tabela categoria os dados do corpo da requisição, se sucesso retornará status(201) se erro, erro(500).

const addCategory = (req, res) => {

const { name, description } = req.body;

db.query('SELECT \* FROM categoria WHERE name = ? AND description = ?', [name, description],

(err, results) => {

if (err) {

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

return res.status(500).send('Erro ao verificar a categoria');

};

if (results.length > 0) {

return res.status(400).send('Categoria já registrada');

};

}

);

db.query('INSERT INTO categoria (name, description) VALUES (?, ?)', [name, description],

(err, results) => {

if (err) {

console.error('Erro ao adicionar a categoria.', err);

return res.status(500).send('Erro ao adicionar a categoria.');

};

res.status(201).send('Categoria adicionada!');

}

);

};

A função categorias retorna uma consulta no banco de dados na tabela categoria de todas as categorias e seus dados registrados e caso dê algum erro retornará o erro(500).

const categories = (req, res) => {

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

if (err) {

console.error('Erro ao obter as categorias', err);

return res.status(500).send('Erro ao obter as categorias');

};

res.json(results);

});

};

A função editar categoria é bem semelhante a função editar dados do admin, com diferença apenas no id que será da tabela categoria.

const editCategory = (req, res) => {

const { id\_categoria } = 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\_categoria);

db.query(`UPDATE categoria SET ${query.join(', ')} WHERE id\_categoria = ?`, values,

(err, results) => {

if (err) {

console.error('Erro ao atualizar a categoria', err);

return res.status(500).send('Erro ao atualizar a categoria');

};

if (results.affectedRows === 0) {

return res.status(404).send('Categoria não encontrada');

};

res.send('Categoria atualizada com sucesso!');

}

);

};

A função deletar categoria também precisará do id da categoria da URL para deletar apenas o que for inserido lá, na tabela categoria, caso não seja encontrado o id será retornado o erro(404), se dê erro na deleção, erro(500) e se tudo dê certo uma mensagem será enviada de que a categoria foi deletada.

const delCategory = (req, res) => {

const { id\_categoria } = req.params;

db.query('DELETE FROM categoria WHERE id\_categoria = ?', [id\_categoria],

(err, results) => {

if (err) {

console.error('Erro ao deletar a categoria', err);

return res.status(500).send('Erro ao deletar a categoria');

};

if (results.affectedRows === 0) {

return res.status(404).send('Categoria não encontrada');

};

res.send('Categoria deletada.');

}

);

};

Aqui quando ocorrer a deleção de alguma categoria e alguma imagem ou pack tiver essa categoria deletada, os dados dos produtos que sofreram alteração por causa da deleção serão inseridos em uma tabela de backup de deleção de categoria, nessa tabela o admin pode visualizar quando ocorreu essa deleção e os produtos afetados, podendo assim atualizá-los posteriormente com uma nova categoria, isso é graças ao trigger antes de deletar (before delete) inserido no banco de dados. O código segue a mesma lógica de consulta de todas as categorias cadastradas, com exceção de que agora é na tabela dat\_del\_category.

const categoriaDeletada = (req, res) => {

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

if (err) {

console.error('Erro ao obter os dados de backup de categoria deletada', err);

return res.status(500).send('Erro ao obter os dados de backup de categoria deletada');

};

res.json(results);

});

};

Os códigos a seguir das funções de cadastro, consulta, edição e deleção de método de pagamento, imagens e pacotes seguem a mesma lógica das funções de categoria, mudando apenas os dados do corpo da requisição e da URL e as tabelas correspondentes, payment, image e pack.

const addPayment = (req, res) => {

const { name } = req.body;

db.query('SELECT \* FROM payment WHERE name = ?', [name],

(err, results) => {

if (err) {

console.error('Erro ao verificar o método de pagamento:', err);

return res.status(500).send('Erro ao verificar o método de pagamento');

};

if (results.length > 0) {

return res.status(400).send('Método de pagamento já registrado');

};

}

);

db.query('INSERT INTO payment (name) VALUES (?)', [name],

(err, results) => {

if (err) {

console.error('Erro ao adicionar o método de pagamento.', err);

return res.status(500).send('Erro ao adicionar o método de pagamento.');

};

res.status(201).send('Método de pagamento adicionado!');

}

);

};

const payments = (req, res) => {

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

if (err) {

console.error('Erro ao obter métodos de pagamento', err);

return res.status(500).send('Erro ao obter métodos de pagamento');

}

res.json(results);

});

};

const editPayment = (req, res) => {

const { id\_payment } = 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\_payment);

db.query(`UPDATE payment SET ${query.join(', ')} WHERE id\_payment = ?`, values,

(err, results) => {

if (err) {

console.error('Erro ao atualizar a categoria', err);

return res.status(500).send('Erro ao atualizar a categoria');

};

if (results.affectedRows === 0) {

return res.status(404).send('Categoria não encontrada');

};

res.send('Método de pagamento atualizado com sucesso!');

}

);

};

const delPayment = (req, res) => {

const { id\_payment } = req.params;

db.query('DELETE FROM payment WHERE id\_payment = ?', [id\_payment],

(err, results) => {

if (err) {

console.error('Erro ao deletar o método de pagamento', err);

return res.status(500).send('Erro ao deletar o método de pagamento');

}

if (results.affectedRows === 0) {

return res.status(404).send('Categoria não encontrada');

};

res.send('Método de pagamento deletado.');

}

);

};

const addImg = (req, res) => {

const { name, description, price, id\_categoria } = req.body;

db.query('SELECT \* FROM image WHERE name = ? AND description = ? AND price = ? AND id\_categoria = ?',

[name, description, price, id\_categoria],

(err, results) => {

if (err) {

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

return res.status(500).send('Erro ao verificar as imagens');

};

if (results.length > 0) {

return res.status(400).send('Imagem já cadastrada');

};

}

);

db.query('INSERT INTO image (name, description, price, id\_categoria) VALUES (?, ?, ?, ?)',

[name, description, price, id\_categoria],

(err, results) => {

if (err) {

console.error('Erro ao adicionar a imagem.', err);

return res.status(500).send('Erro ao adicionar a imagem.');

};

res.status(201).send('Imagem adicionada!');

}

);

};

const images = (req, res) => {

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

if (err) {

console.error('Erro ao obter as imagens', err);

return res.status(500).send('Erro ao obter as imagens');

};

res.json(results);

});

};

const editImg = (req, res) => {

const { id\_image } = 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\_image);

db.query(`UPDATE image SET ${query.join(', ')} WHERE id\_image = ?`, values,

(err, results) => {

if (err) {

console.error('Erro ao atualizar a imagem.', err);

return res.status(500).send('Erro ao atualizar a imagem.');

};

if (results.affectedRows === 0) {

return res.status(404).send('Imagem não encontrada.')

};

res.send('Imagem atualizada com sucesso!')

}

);

};

const delImg = (req, res) => {

const { id\_image } = req.params;

db.query('DELETE FROM image WHERE id\_image = ?', [id\_image],

(err, results) => {

if (err) {

console.error('Erro ao deletar a imagem.', err);

return res.status(500).send('Erro ao deletar a imagem.');

};

if (results.affectedRows === 0) {

return res.status(404).send('Imagem não encontrada.')

};

res.send('Imagem deletada com sucesso!')

}

);

};

const addPack = (req, res) => {

const { name, description, price, id\_categoria, id\_image1, id\_image2, id\_image3, id\_image4 } = req.body;

db.query('SELECT \* FROM pack WHERE name = ? AND description = ? AND price = ? AND id\_categoria = ?',

[name, description, price, id\_categoria],

(err, results) => {

if (err) {

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

return res.status(500).send('Erro ao verificar o pack');

};

if (results.length > 0) {

return res.status(400).send('Pack já cadastrado');

};

}

);

db.query(`INSERT INTO pack (name, description, price, id\_categoria, id\_image1, id\_image2, id\_image3, id\_image4)

VALUES (?, ?, ?, ?, ?, ?, ?, ?)`, [name, description, price, id\_categoria, id\_image1, id\_image2, id\_image3, id\_image4],

(err, results) => {

if (err) {

console.error('Erro ao adicionar o pack.', err);

return res.status(500).send('Erro ao adicionar o pack.');

};

res.status(201).send('Pack adicionado!')

}

);

};

const packs = (req, res) => {

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

if (err) {

console.error('Erro ao obter os packs', err);

return res.status(500).send('Erro ao obter os packs');

}

res.json(results);

});

};

const editPack = (req, res) => {

const { id\_pack } = 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\_pack);

db.query(`UPDATE pack SET ${query.join(', ')} WHERE id\_pack = ?`, values,

(err, results) => {

if (err) {

console.error('Erro ao atualizar o pack.', err);

return res.status(500).send('Erro ao atualizar o pack.');

};

if (results.affectedRows === 0) {

return res.status(404).send('Pack não encontrado.');

};

res.send('Pack atualizado com sucesso!');

}

);

};

const delPack = (req, res) => {

const { id\_pack } = req.params;

db.query('DELETE FROM pack WHERE id\_pack = ?', [id\_pack],

(err, results) => {

if (err) {

console.error('Erro ao deletar o pack.', err);

return res.status(500).send('Erro ao deletar o pack.');

};

if (results.affectedRows === 0) {

return res.status(404).send('Pack não encontrado.')

};

res.send('Pack deletado com sucesso!');

}

);

};

A função adicionar pedido precisa do id do usuário(comum) e do id do método de pagamento escolhido por ele do corpo da requisição para primeiro fazer uma consulta e obter uma resposta de soma total dos itens do carrinho quando o usuário concluir a compra. Essa soma é armazenada em uma constante e inserida na tabela pedidos junto com os dois id’s anteriores resultando assim em um id de pedido que será armazenado em uma constante. Depois é inserida junto com os dados do carrinho que contém o id do usuário na tabela itens\_pedido e por fim os itens do carrinho são deletados, caso ocorra algum erro durante o processo será retornado o erro(500) e se não, uma mensagem de pedido criado com sucesso.

const addOrder = (req, res) => {

const { id\_usuario, id\_payment } = req.body;

db.query('SELECT SUM(preco) AS price\_total FROM carrinho WHERE id\_usuario = ?', [id\_usuario],

(err, totalResult) => {

if (err) {

console.error('Erro ao somar o total dos itens do carrinho', err);

return res.status(500).send('Erro ao somar o total dos itens do carrinho');

};

const price\_total = totalResult[0].price\_total;

db.query('INSERT INTO pedidos (id\_usuario, price\_total, id\_payment) VALUES (?, ?, ?)',

[id\_usuario, price\_total, id\_payment], (err, result) => {

if (err) {

console.error('Erro ao inserir o pedido', err);

return res.status(500).send('Erro ao inserir o pedido');

};

const id\_pedido = result.insertId;

db.query(` INSERT INTO itens\_pedido (id\_pedido, id\_item, id\_img, id\_pack, preco, id\_categoria)

SELECT ?, id, id\_img, id\_pack, preco, id\_categoria FROM carrinho WHERE id\_usuario = ?`,

[id\_pedido, id\_usuario], (err, result) => {

if (err) {

console.error('Erro ao inserir os itens do carrinho', err);

return res.status(500).send('Erro ao inserir os itens do carrinho');

};

db.query('DELETE FROM carrinho WHERE id\_usuario = ?', [id\_usuario], (err, result) => {

if (err) {

console.error('Erro ao deletar os itens do carrinho', err);

return res.status(500).send('Erro ao deletar os itens do carrinho');

};

res.status(201).send('Pedido criado com sucesso!');

});

}

);

}

);

}

);

};

As funções a seguir, todos os pedidos, todos os usuários cadastrados, todas as imagens (por id) e todos os pacotes (por id) seguem a mesma lógica que a função todas as categorias, mudando apenas a seleção do id das duas últimas e o nome das tabelas para pedidos, user, image e pack.

const orders = (req, res) => {

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

if (err) {

console.error('Erro ao obter os pedidos processados', err);

return res.status(500).send('Erro ao obter os pedidos processados');

};

res.json(results);

});

};

const users = (req, res) => {

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

if (err) {

console.error('Erro ao obter os usuários cadastrados', err);

return res.status(500).send('Erro ao obter os usuários cadastrados');

};

res.json(results);

});

};

const filtroImages = (req, res) => {

db.query(`SELECT id\_image FROM image`, (err, results) => {

if (err) {

console.error('Erro ao filtrar por imagem', err);

return res.status(500).send('Erro ao filtrar por imagem');

};

res.json(results);

});

};

const filtroPacks = (req, res) => {

db.query(`SELECT id\_pack FROM pack`, (err, results) => {

if (err) {

console.error('Erro ao filtrar por pack', err);

return res.status(500).send('Erro ao filtrar por pack');

};

res.json(results);

});

};

A função filtro por categoria precisará do id da categoria da URL para selecionar todas as imagens e pacotes que possuem aquela categoria vinculada, caso dê erro durante a consulta o erro será erro(500), caso não encontre a categoria ou a categoria não possua produtos será erro(404) e caso dê tudo certo, retornará o resultado em formato json.

const filtroCategories = (req, res) => {

const { id\_categoria } = req.params;

db.query(`SELECT image.id\_image, pack.id\_pack FROM image INNER JOIN

pack ON image.id\_categoria = pack.id\_categoria WHERE image.id\_categoria = ?`,

[id\_categoria], (err, results) => {

if (err) {

console.error('Erro ao filtrar por categoria', err);

return res.status(500).send('Erro ao filtrar por categoria');

};

if (results.length > 0) {

return res.json(results);

} else {

return res.status(404).send('Nenhuma imagem ou pack encontrado.');

}

}

);

};

E por último a função pesquisar vai precisar de um termo de pesquisa que será inserido na URL para que possa ser consultado todas as imagens e pacotes que possuem algum caractere igual aos do termo no nome ou descrição desses e será retornado dados como o tipo, id, nome, descrição e preço dos produtos em formato json, caso dê erro durante a consulta o erro será erro(500) e caso não encontre a nenhum produto com os caracteres inseridos será erro(404).

const search = (req, res) => {

const searchTerm = req.query.termo;

const likeSearchTerm = `%${searchTerm}%`;

db.query(`SELECT 'image' AS type, id\_image AS id, name, description, price FROM image

WHERE name LIKE ? OR description LIKE ? UNION

SELECT 'pack' AS type, id\_pack AS id, name, description, price FROM pack

WHERE name LIKE ? OR description LIKE ?;`,

[likeSearchTerm, likeSearchTerm, likeSearchTerm, likeSearchTerm], (err, results) => {

if (err) {

console.error('Erro ao fazer a busca:', err);

return res.status(500).json({ error: 'Erro ao fazer a busca' });

}

if (results.length > 0) {

return res.json(results);

} else {

return res.status(404).send('Nenhuma imagem ou pack encontrado.');

}

});

};

Por fim a exportação de todas as funções para adicioná-las às rotas:

module.exports = {

dadosEmployee,

editDatasEmployee,

addCategory,

categories,

editCategory,

delCategory,

addPayment,

payments,

editPayment,

delPayment,

addImg,

images,

editImg,

delImg,

addPack,

packs,

editPack,

delPack,

addOrder,

orders,

users,

categoriaDeletada,

filtroCategories,

filtroImages,

filtroPacks,

search

}

No arquivo routesCart.js possui as funções de adicionar itens ao carrinho, consultar o carrinho e deletar itens do carrinho, as duas últimas funções seguem a mesma lógica que a função todas as categorias e deletar categoria, mudando apenas a seleção do id e o nome das tabelas para carrinho. E por último a exportação dos códigos para serem utilizados nas rotas.

A função adicionar itens ao carrinho funciona de maneira semelhante a função adicionar categoria, mudando os campos do corpo da requisição, consultando se alguma imagem ou pacote já foi inserido no carrinho (erro 400), não deixando o usuário inserir mais de um item igual, caso não tenha item repetido adiciona o mesmo à tabela carrinho (status 201) e caso ocorra algum problema o erro(500) é retornado.

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

const addCart = (req, res) => {

const { id\_usuario, id\_img, id\_pack, preco, id\_categoria } = req.body;

db.query('SELECT \* FROM carrinho WHERE id\_img = ? OR id\_pack = ?', [id\_img, id\_pack],

(err, results) => {

if (err) {

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

return res.status(500).send('Erro ao verificar o carrinho');

};

if (results.length > 0) {

return res.status(400).send('Imagem ou pack já adicionado');

} else {

db.query('INSERT INTO carrinho (id\_usuario, id\_img, id\_pack, preco, id\_categoria) VALUES (?, ?, ?, ?, ?)',

[id\_usuario, id\_img, id\_pack, preco, id\_categoria], (err, results) => {

if (err) {

console.error('Erro ao adicionar o item.', err)

return res.status(500).send('Erro ao adicionar o item.')

}

res.status(201).send('Item adicionado ao carrinho!')

}

)

}

}

);

}

const cart = (req, res) => {

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

if (err) {

console.error('Erro ao obter os itens do carrinho', err)

return res.status(500).send('Erro ao obter os itens do carrinho')

}

res.json(results)

})

}

const delItemCart = (req, res) => {

const { id } = req.params

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

if (err) {

console.error('Erro ao excluir o item do carrinho', err)

return res.status(500).send('Erro ao excluir o item do carrinho')

}

if (results.affectedRows === 0) {

return res.status(404).send('Item não encontrado');

} else {

return res.status(201).send('Item excluido do carrinho!');

}

})

}

module.exports = {

addCart,

cart,

delItemCart

}

No arquivo routesUser.js estam as funções de consultar os dados do usuário, editar esses dados, consultar os pedidos, filtrar as imagens e os pacotes por categoria e consultar todas as imagens ou packs já adquiridos.

das transações precisamos atualizar o código no arquivo server.js adicionando a importação das rotas e configurando o servidor para usar as rotas em todas as requisições que começarem com */api/transactions* e então assim está concluído o processo para fazer as operações que o administrador do inventário nos requereu.

//Conectando o banco de dados e requerendo transações

const inventory\_db = require('./config/inventory\_db');

const transactionsRoutes = require('./routes/transactions');

app.use('/api/transactions', transactionsRoutes);

Quando o usuário pedir todas as requisições propostas o servidor irá retornar o código 200 OK ou 200 e uma mensagem de sucesso, se a URL for a correta para a transação. Abaixo uma tabela com todas as requisições e seus possíveis erros.

|  |  |
| --- | --- |
| Requisição | Resposta |
| Método POST:  URL: http://0.0.0.0:3000/api/transactions  Body: {  "name": "Açaí combo 1",  "description": "Açaí com guloseimas e calda de morango",  "category": "Açaí",  "price": "9.99",  "stock": 6,  "expiry\_date": "2025-04-20"  } | 201 Created – Transação adicionada com sucesso!;  400 Bad Request – SyntaxError (Algo está escrito incorretamente no body, no JSON);  500 Internal Server Error – ReferenceError (Algo está escrito de maneira errada na função em questão no arquivo de controle); |
| Método GET:  URL: http://0.0.0.0:3000/api/transactions | 200 OK:  [  {  "id": 1,  "name": "Creme de avelã",  "description": "Creme de leite com avelã",  "category": "Creme",  "price": "4.99",  "stock": 20,  "expiry\_date": "2025-06-25T03:00:00.000Z"  },  {  "id": 2,  "name": "Creme com passas",  "description": "Creme de leite com passas",  "category": "Creme",  "price": "4.99",  "stock": 15,  "expiry\_date": "2025-06-01T03:00:00.000Z"  }  ]  500 InternalServerError – Erro ao obter todas as transações (No terminal do servidor poderá ter o erro 'ER\_NO\_SUCH\_TABLE', que mostra que a tabela em questão não existe. |
| Método PUT:  URL: http://0.0.0.0:3000/api/transactions/id  Body:  {  "name": "Picolé de abóbora",  "description": "Picolé sabor abóbora",  "category": "Picolé",  "price": "2.99",  "stock": 15,  "expiry\_date": "2024-10-28"  } | 200 OK – Transação atualizada com sucesso!  400 Bad Request – SyntaxError (Algo está escrito incorretamente no body, no JSON);  500 Internal Server Error – ReferenceError (Algo está escrito de maneira errada na função em questão no arquivo de controle);  404 Not Found (Há um erro de sintaxe na URL); |
| Método PATCH:  URL: http://0.0.0.0:3000/api/transactions/id  Body:  {  "expiry\_date": "2024-12-21"  } | 200 OK – Transação atualizada com sucesso!  400 Bad Request – SyntaxError (Algo está escrito incorretamente no body, no JSON);  500 Internal Server Error – ReferenceError (Algo está escrito de maneira errada na função em questão no arquivo de controle);  404 Not Found (Há um erro de sintaxe na URL); |
| Método DELETE:  URL: http://0.0.0.0:3000/api/transactions/id | 200 OK – Transação deletada com sucesso!  500 Internal Server Error – ReferenceError (Algo está escrito de maneira errada na função em questão no arquivo de controle);  404 Not Found (Há um erro de sintaxe na URL);  Error: Couldn't connect to server (Erro ao conectar o servidor, a porta deve está escrita errada na URL); |

Feito todos esses procedimentos a API está feita e pronta para pegar as requisições do usuário, estas mencionadas anteriormente: consulta de todos os produtos cadastrados (GET), adicionar um novo produto (POST), atualizar todas as informações (PUT) e atualizar de forma parcial um produto (PATCH) e deletar um produto (DELETE) e retornar as devidas respostas ao cliente.