Programação Web com Linguagens de Script

Thiago Rodrigues

Desenvolvimento

- Iniciando aplicação
  - npm init -y

- Utilizando o tipo de importação de pacotes module.
  - No arquivo package.json, adicione: "type": "module“.

- Instalando dependências
  - npm i express bcrypt multer jsonwebtoken dotenv mongoose
  - npm i -D nodemon

Desenvolvimento

- **Estrutura de pacotes**

  src
  - model
  - controller
  - service
  - routes
  - db
  - helpers
  - public
    - images
      - users
      - products

Desenvolvimento

- Criando arquivo .env
  - O arquivo .env deve ser criado na raiz do projeto.
  - Por padrão o nome das propriedades de um arquivo .env deve ser todo em caixa alta.

- Criando arquivo app.js
  - O arquivo app.js deve ficar dentro da pasta ./src

- Criando atalho para execução do projeto
  - No arquivo package.json adicione no objeto “scripts”
    - “start”:”nodemon -r dotenv/config ./src/app.js”

- Executando projeto
  - npm run start

Desenvolvimento

- Adicionando propriedades no arquivo .env
  - Adicione as propriedades de porta e ambiente
    - PORT=3000
    - NODE_ENV=development

Desenvolvimento

- Arquivo app.js

```
// Importa o módulo 'express', uma framework de servidor web para Node.js
import express from "express";

// Cria uma instância da aplicação Express
const app = express();

// Adiciona um middleware para interpretar requisições com JSON no corpo
app.use(express.json());

// Inicia o servidor, ouvindo na porta definida na variável de ambiente 'PORT'
// O callback é executado quando o servidor começa a ouvir (inicializar) e imprime uma mensagem no console
app.listen(process.env.PORT, ()=>{
   console.log(`Server running in port: ${process.env.PORT}`);
});
```

Desenvolvimento

- Abrindo conexão com o banco de dados
  - Crie uma variável de ambiente chamada MONGO_URL
    - MONGO_URL="mongodb+srv://**<username>**:**<password>**@aula-pos.tamaygj.mongodb.net/**<collection_name>**?retryWrites=true&w=majority&appName=aula-pos"

Desenvolvimento

- Criando arquivo db.js

```
// Importa o módulo 'mongoose', uma biblioteca ODM (Object Data Modeling) para MongoDB e Node.js
import mongoose from "mongoose";

// Declara uma função assíncrona para conectar ao banco de dados MongoDB
const connectDB = async() => {
   // Inicia um bloco try/catch para lidar com possíveis erros de conexão
   try {
      // Tenta conectar ao MongoDB usando a URL fornecida na variável de ambiente 'MONGO_URL'
      const conn = await mongoose.connect(process.env.MONGO_URL);
      // Se a conexão for bem-sucedida, imprime no console uma mensagem com o namespace do banco de dados conectado
      console.log(`MongoDB connected: ${conn.connection.db.namespace}`);
   } catch (error) {
      // Se ocorrer um erro, imprime o erro no console
      console.log(error);
      // Encerra o processo com um código de falha (1)
      process.exit(1);
   }
}

// Exporta a função 'connectDB' para que possa ser usada em outros arquivos
export default connectDB;
```

Desenvolvimento

- Alterando inicialização do servidor no arquivo app.js

```
// Importa o módulo 'express', uma framework de servidor web para Node.js
import express from "express";

// Chama a função 'connectDB' para conectar ao banco de dados MongoDB
connectDB().then(() => {
    // Após a conexão bem-sucedida ao banco de dados, inicia o servidor, ouvindo na porta definida na variável de ambiente 'PORT'
    // O callback é executado quando o servidor começa a ouvir (starta) e imprime uma mensagem no console
    app.listen(process.env.PORT, () => {
        console.log(`Server running in port: ${process.env.PORT}`);
    });
});
```

Desenvolvimento

- Crie os arquivos para representar os modelos das entidades que serão salvas no banco de dados:
  - User.js
  - Product.js

Desenvolvimento

- Arquivo User.js

```
// Importa o 'Schema' do módulo 'mongoose' para definir a estrutura dos documentos no MongoDB
import { Schema } from "mongoose";
// Importa o módulo 'mongoose' para interagir com o MongoDB
import mongoose from "mongoose";

// Define o modelo 'User' com um novo esquema
const User = mongoose.model(
   "User", // Nome do modelo
   new Schema({
      // Definição dos campos do esquema 'User'
      name: {
         type: String, // Tipo do campo é String
         required: true, // Campo obrigatório
         maxlength: 80 // Tamanho máximo de 80 caracteres
      },
      email: {
         type: String, // Tipo do campo é String
         required: true, // Campo obrigatório
         unique: true, // Campo Único
         maxlength: 100 // Tamanho máximo de 100 caracteres
      },
      password: {
         type: String, // Tipo do campo é String
         required: true, // Campo obrigatório
         maxlength: 80 // Tamanho máximo de 80 caracteres
      },
      image: String, // Campo opcional de tipo String para armazenar a URL da imagem do usuário
      phone: {
         type: String, // Tipo do campo é String
         required: true, // Campo obrigatório
         maxlength: 20 // Tamanho máximo de 20 caracteres
      },
      address: {
         type: String, // Tipo do campo é String
         required: true, // Campo obrigatório
         maxlength: 180 // Tamanho máximo de 180 caracteres
      }
   },
   {timestamps: true} // Adiciona campos 'createdAt' e 'updatedAt' automaticamente
   )
);

// Exporta o modelo 'User' para que possa ser usado em outros arquivos
export default User;
```

Desenvolvimento

- Arquivo Product.js

```
// Importa o módulo 'Schema' do Mongoose para definir a estrutura do modelo de dados
import { Schema } from "mongoose";
// Importa o módulo 'mongoose' para interagir com o MongoDB
import mongoose from "mongoose";

// Define o modelo 'Product' usando o método 'mongoose.model()'
const Product = mongoose.model(
  "Product", // Nome do modelo
  new Schema({
    // Define os campos do modelo 'Product' e suas configurações
    name: {
      type: String, // Tipo do campo é String
      required: true // Campo obrigatório
    },
    description: {
      type: String, // Tipo do campo é String
      required: true // Campo obrigatório
    },
    images: {
      type: Array, // Tipo do campo é Array
      required: true // Campo obrigatório
    },
    available: Boolean, // Tipo do campo é Boolean (opcional)
    state: {
      type: String, // Tipo do campo é String
      enum: ["good", "fair", "bad"] // Valor permitido deve ser um dos especificados
    },
    owner: {
      type: Schema.Types.ObjectId, // Tipo do campo é ObjectId, referenciando outro documento
      ref: "User" // Referência ao modelo 'User'
    },
    receiver: {
      type: Schema.Types.ObjectId, // Tipo do campo é ObjectId, referenciando outro documento
      ref: "User" // Referência ao modelo 'User'
    },
    purchased_at: Date, // Campo de data para a data de compra
    donated_at: Date // Campo de data para a data de doação
  }, { timestamps: true }) // Opções adicionais, neste caso, para adicionar automaticamente campos de
'createdAt' e 'updatedAt'
);

// Exporta o modelo 'Product' para ser usado em outros arquivos
export default Product;
```

Desenvolvimento

- Criando lógica para rotas de usuário
  - Crie o arquivo UserRoutes.js, UserController.js e UserService.js

- Para o **controller** e o **service** crie as respectivas classes e métodos estáticos.

Desenvolvimento

- **Lógica para registrar um usuário**
  - Receber dados da requisição
  - Validar todos os dados
  - Verificar se a senha é igual a confirmação de senha
  - Verificar se já existe um usuário com e-mail informado
  - Criar um hash da senha
  - Salvar o usuário
  - Criar o token de acesso
  - Retornar o token e o id do usuário

Desenvolvimento

- Arquivo UserController.js

```
// Importa a função 'createUserToken' do arquivo '../helpers/manage_jwt.js'
import { createUserToken } from "../helpers/manage_jwt.js";
// Importa o serviço 'UserService' do arquivo '../service/UserService.js'
import UserService from "../service/UserService.js";

// Exporta a classe 'UserController' que contém métodos para lidar com as requisições relacionadas aos usuários
export default class UserController {
  // Método estático assíncrono para registrar um novo usuário
  static async register(req, res) {
    try {
      // Extrai os campos 'name', 'email', 'phone', 'address', 'password' e 'confirmpassword' do corpo da requisição
      const { name, email, phone, address, password, confirmpassword } = req.body;
      // Chama o método 'register' do serviço 'UserService' para registrar o usuário com os dados fornecidos
      const savedUser = await UserService.register(name, email, phone, password, confirmpassword, address);
      // Cria um token de autenticação para o usuário registrado usando a função 'createUserToken'
      const token = createUserToken(savedUser);
      // Retorna uma resposta com status 201 (Created), incluindo o token e o ID do usuário
      res.status(201).json({ token, userId: savedUser._id });
    } catch (error) {
      // Se ocorrer um erro, define o status do erro com base no código de status fornecido ou 500 (Internal Server Error) por padrão
      error.statusCode = error.statusCode || 500;
      // Retorna uma resposta de erro com o status e uma mensagem de erro
      res.status(error.statusCode).json({ error: error.message });
    }
  }
}
```

Desenvolvimento

- Arquivo UserService.js

```
// Importa o modelo de usuário ('User') do arquivo '../model/User.js'
import User from "../model/User.js";
// Importa o módulo 'bcrypt' para criptografar senhas
import bcrypt from "bcrypt";

// Exporta a classe 'UserService' que contém métodos relacionados à lógica de negócios do usuário
export default class UserService {
   // Método estático assíncrono para registrar um novo usuário
   static async register(name, email, phone, password, confirmpassword, address) {
      // Verifica se o campo 'name' está presente
      if (!name) {
         const error = new Error("O nome é obrigatório.");
         error.statusCode = 422;
         throw error;
      }

      // ...

      // Verifica se já existe um usuário com o mesmo email
      const userExists = await User.findOne({ email });

      if (userExists) {
         const error = new Error("Por favor, utilize outro e-mail.");
         error.statusCode = 422;
         throw error;
      }

      // Gera um salt (um valor aleatório usado como parte do processo de criptografia)
      const salt = await bcrypt.genSalt(12);
      // Cria o hash da senha usando o salt
      const passwordHash = await bcrypt.hash(password, salt);

      // Cria uma instância do modelo 'User' com os dados fornecidos
      const user = new User({ name, email, password: passwordHash, phone, address });
      // Salva o usuário no banco de dados
      const savedUser = await user.save();

      // Retorna o usuário salvo
      return savedUser;
   }
}
```

Desenvolvimento

- Arquivo UserRoutes.js

```
// Importa o módulo 'Router' do Express para criar rotas
import { Router } from "express";
// Importa o controlador 'UserController' do arquivo '../controller/UserController.js'
import UserController from "../controller/UserController.js";

// Cria uma instância do Router do Express
const router = Router();

// Define uma rota POST para o registro de usuários, que chama o método 'register' do controlador 'UserController'
router.post("/register", UserController.register);

// Exporta o router para ser usado em outros arquivos, como no arquivo principal da aplicação
export default router;
```

Desenvolvimento

- Adicionando rotas no arquivo app.js

```
import UserRoutes from "./routes/UserRoutes.js";

app.use("/users", UserRoutes);
```

Desenvolvimento

- Adicionando propriedade de chave privada nas variáveis de ambiente
  - PRIVATE_KEY="..."

https://www.lastpass.com/pt/features/password-generator#generatorTool

Desenvolvimento

- Arquivo manage_jwt.js

```
// Importa o módulo 'jsonwebtoken' para lidar com geração de tokens JWT (JSON Web Tokens)
import jwt from "jsonwebtoken";

// Define a função 'createUserToken', que recebe um usuário como argumento e retorna um token JWT
export const createUserToken = (user) => {
   // Gera um token JWT com os dados do usuário (nome e ID) e a chave privada definida na variável de ambiente 'PRIVATE_KEY'
   const token = jwt.sign({
      name: user.name, // Nome do usuário
      id: user._id // ID do usuário
   }, process.env.PRIVATE_KEY); // Chave privada usada para assinar o token

   // Retorna o token gerado
   return token;
}
```

Desenvolvimento

- **Lógica para login do usuário**
  - Receber dados da requisição
  - Validar todos os dados
  - Verificar se já existe um usuário com e-mail informado
  - Validar o hash da senha
  - Salvar o usuário
  - Criar o token de acesso
  - Retornar o token e o id do usuário

Desenvolvimento

- Arquivo UserController.js

```
// Método estático assíncrono 'login' que recebe requisição (req) e resposta (res)
static async login(req, res){
    try {
        // Extrai o email e a senha do corpo da requisição
        const { email, password } = req.body;
        // Chama o método 'validateLoginData' do serviço 'UserService' para validar os dados de login
        const user = await UserService.validateLoginData(email, password);
        // Cria um token de autenticação para o usuário validado usando a função 'createUserToken'
        const token = createUserToken(user);
        // Retorna uma resposta com status 200 (OK), incluindo o token e o ID do usuário
        res.status(200).json({ token, userId: user._id });
    } catch (error) {
        // Se ocorrer um erro, define o status do erro com base no código de status fornecido ou 500 (Internal Server Error) por padrão
        error.statusCode = error.statusCode || 500;
        // Retorna uma resposta de erro com o status e uma mensagem de erro
        res.status(error.statusCode).json({ error: error.message });
    }
}
```

Desenvolvimento

- Arquivo UserService.js

```
// Método estático assíncrono 'validateLoginData' que recebe um email e uma senha como argumentos
static async validateLoginData(email, password){
   // Verifica se o email foi fornecido
   if(!email){
      const error = new Error("O email é obrigatório.");
      error.statusCode = 422; // Define o status do erro como 422 (Unprocessable Entity)
      throw error; // Lança o erro
   }
   // ...
   // Procura um usuário no banco de dados com o email fornecido
   const user = await User.findOne({email});

   // Se nenhum usuário for encontrado com o email fornecido, lança um erro de autenticação
   if(!user){
      const error = new Error("Email ou senha inválidos.");
      error.statusCode = 401; // Define o status do erro como 401 (Unauthorized)
      throw error; // Lança o erro
   }

   // Compara a senha fornecida com a senha armazenada no banco de dados para o usuário encontrado
   const checkPassword = await bcrypt.compare(password, user.password);

   // Se as senhas não coincidirem, lança um erro de autenticação
   if(!checkPassword){
      const error = new Error("Email ou senha inválidos.");
      error.statusCode = 401; // Define o status do erro como 401 (Unauthorized)
      throw error; // Lança o erro
   }

   // Se o email e a senha forem válidos, retorna o usuário
   return user;
}
```

Desenvolvimento

- Arquivo UserRoutes.js

```
router.post("/login", UserController.login);
```

Desenvolvimento

- **Lógica para resgatar usuário atual**
  - Extrair dados do token de autenticação
  - Buscar usuário na base de dados

Desenvolvimento

- Arquivo manage_jwt.js

```
// Exporta a função middleware 'verifyJWT' que verifica a validade de um token JWT
export const verifyJWT = async (req, res, next) => {
    // Obtém o cabeçalho de autorização da requisição
    const headerAuth = req.headers.authorization;
    // Extrai o token do cabeçalho de autorização (formato esperado: "Bearer <token>")
    const token = headerAuth?.split(" ")[1];

    // Se não houver token, retorna uma resposta de erro 401 (Unauthorized)
    if (!token) {
        res.status(401).json({ message: "Acesso Negado." });
        return; // Interrompe a execução da função
    }

    try {
        // Verifica e decodifica o token JWT usando a chave privada definida na variável de ambiente 'PRIVATE_KEY'
        const payload = jwt.verify(token, process.env.PRIVATE_KEY);
        // Anexa os dados do payload ao objeto de requisição (req) para uso posterior
        req.user = payload;
        // Imprime o payload no console para depuração
        console.log(payload);
        // Chama o próximo middleware ou rota
        next();
    } catch (error) {
        // Se ocorrer um erro ao verificar o token, retorna uma resposta de erro 400 (Bad Request)
        res.status(400).json({ message: "Token Inválido." });
        return; // Interrompe a execução da função
    }
}
```

Desenvolvimento

- Arquivo UserController.js

```
// Método estático assíncrono 'getUser' que recebe requisição (req) e resposta (res)
static async getUser(req, res) {
    try {
        // Chama o método 'getUserByToken' do serviço 'UserService' passando a requisição, para obter os dados do usuário a partir do token
        const user = await UserService.getUserByToken(req);
        // Se a operação for bem-sucedida, retorna uma resposta com status 200 (OK) e os dados do usuário em formato JSON
        res.status(200).json({ user });
    } catch (error) {
        // Se ocorrer um erro, define o status do erro com base no código de status fornecido ou 500 (Internal Server Error) por padrão
        error.statusCode = error.statusCode || 500;
        // Retorna uma resposta de erro com o status e uma mensagem de erro
        res.status(error.statusCode).json({ message: error.message });
    }
}
```

Desenvolvimento

- Arquivo UserService.js

```
// Método estático assíncrono 'getUserByToken' que recebe uma requisição (req)
static async getUserByToken(req) {
   // Verifica se os dados do usuário estão presentes na requisição
   if (!req.user) {
      // Se não estiverem presentes, lança um erro de "Acesso negado" com status 401 (Unauthorized)
      const error = new Error("Acesso negado.");
      error.statusCode = 401;
      throw error;
   }

   // Procura um usuário no banco de dados pelo ID presente no token, excluindo o campo 'password' da seleção
   const user = await User.findOne({ _id: req.user.id }).select("-password");

   // Retorna os dados do usuário encontrado
   return user;
}
```

Desenvolvimento

- Arquivo UserRoutes.js

```
router.get("/currentUser", verifyJWT, UserController.getUser);
```

Desenvolvimento

- **Lógica para realizar atualização de dados do usuário atual**
  - Extrair dados do token de autenticação
  - Buscar usuário na base de dados
  - Receber dados do corpo da requisição
  - Receber dado da imagem
  - Validar atributos
  - Verificar se o email é diferente e já existe na base de dados
  - Verificar se senha e confirmação de senha são iguais e se foram iguais criar hash da senha novamente
  - Atualizar usuário

Desenvolvimento

- Arquivo image_upload.js

```
// Importa a biblioteca multer, que é usada para fazer upload de arquivos
import multer from "multer";

// Importa a biblioteca path, que fornece utilitários para trabalhar com caminhos de arquivos e diretórios
import path from "path";

// Configura a forma como os arquivos de imagem serão armazenados no disco
const imageStore = multer.diskStorage({
   // Define a pasta de destino para os arquivos de imagem
   destination: (req, file, cb) => {
      let folder = "";  // Variável para armazenar o nome da pasta

      // Verifica a URL base da requisição para determinar a pasta correta
      if(req.baseUrl.includes("users")){
         folder = "users";  // Se a URL base incluir "users", define a pasta como "users"
      }else if(req.baseUrl.includes("products")){
         folder = "products";  // Se a URL base incluir "products", define a pasta como "products"
      }

      // Chama o callback para definir o caminho completo da pasta de destino
      cb(null, `src/public/images/${folder}`);
   },
   // Define o nome do arquivo
   filename: (req, file, cb) => {
      // Gera um nome único para o arquivo usando a data atual e um número aleatório
      cb(null, Date.now() + Math.floor(Math.random() * 1000) + path.extname(file.originalname));
   }
});
...
```

Desenvolvimento

- Arquivo image_upload.js

```
...
// Configura o middleware de upload de imagem usando multer
const imageUpload = multer({
   // Define a estratégia de armazenamento configurada anteriormente
   storage: imageStore,
   // Filtra os arquivos para permitir apenas imagens PNG ou JPG
   fileFilter(req, file, cb) {
      // Verifica se o nome original do arquivo corresponde aos formatos permitidos
      if (!file.originalname.match(/\.(png|jpg)$/)) {
         // Se o arquivo não for PNG ou JPG, retorna um erro
         return cb(new Error("Por favor, envie uma imagem jpg ou png"));
      }
      // Se o arquivo for válido, chama o callback sem erros
      cb(undefined, true);
   }
});

// Exporta o middleware de upload de imagem para uso em outras partes da aplicação
export default imageUpload;
```

Desenvolvimento

- Arquivo UserController.js

```
// Método estático assíncrono 'updateUser' que recebe requisição (req) e resposta (res)
static async updateUser(req, res) {
   try {
      // Extrai os campos 'name', 'email', 'phone', 'address', 'password' e 'confirmpassword' do corpo da requisição
      const { name, email, phone, address, password, confirmpassword } = req.body;
      // Obtém os dados do usuário a partir do token presente na requisição
      const userByToken = await UserService.getUserByToken(req);
      // Chama o método 'updateUser' do serviço 'UserService' para atualizar os dados do usuário
      const user = await UserService.updateUser(req, userByToken, name, email, phone, password, confirmpassword, address);
      // Se a operação for bem-sucedida, retorna uma resposta com status 200 (OK) e os dados do usuário atualizado em formato JSON
      res.status(200).json({ user });
   } catch (error) {
      // Se ocorrer um erro, define o status do erro com base no código de status fornecido ou 500 (Internal Server Error) por padrão
      error.statusCode = error.statusCode || 500;
      // Retorna uma resposta de erro com o status e uma mensagem de erro
      res.status(error.statusCode).json({ message: error.message });
   }
}
```

Desenvolvimento

- Arquivo UserService.js

```
// Método estático assíncrono 'updateUser' que recebe requisição (req), usuário (user), nome (name), email (email), telefone (phone), senha (password), confirmação de senha (confirmpassword) e endereço (address)
static async updateUser(req, user, name, email, phone, password, confirmpassword, address) {
    // Verifica se há um arquivo na requisição e, se houver, define a imagem do usuário como o nome do arquivo
    if (req.file) {
        user.image = req.file.filename;
    }

    // Verifica se o nome foi fornecido
    if (!name) {
        const error = new Error("O nome é obrigatório.");
        error.statusCode = 422; // Define o status do erro como 422 (Unprocessable Entity)
        throw error; // Lança o erro
    }

    // Verifica se já existe um usuário com o email fornecido
    const userExists = await User.findOne({ email });

    // Se o email atual do usuário for diferente do novo email e o novo email já estiver em uso, lança um erro
    if (user.email !== email && userExists) {
        const error = new Error("Por favor, informe outro e-mail");
        error.statusCode = 422; // Define o status do erro como 422 (Unprocessable Entity)
        throw error; // Lança o erro
    }

    // Define o email do usuário
    user.email = email;
```

Desenvolvimento

- Arquivo UserService.js

```
 //...
// Verifica se a senha foi fornecida
  if (!password) {
     const error = new Error("A senha é obrigatória.");
     error.statusCode = 422; // Define o status do erro como 422 (Unprocessable Entity)
     throw error; // Lança o erro
  }

  // Verifica se a confirmação de senha foi fornecida
  if (!confirmpassword) {
     const error = new Error("A confirmação de senha é obrigatória.");
     error.statusCode = 422; // Define o status do erro como 422 (Unprocessable Entity)
     throw error; // Lança o erro
  }

  // Verifica se a senha e a confirmação de senha coincidem
  if (password !== confirmpassword) {
     const error = new Error("A senha precisa ser igual a confirmação de senha.");
     error.statusCode = 422; // Define o status do erro como 422 (Unprocessable Entity)
     throw error; // Lança o erro
  } else if (password === confirmpassword && password != null) {
     // Gera um salt e hash para a nova senha
     const salt = await bcrypt.genSalt(12);
     const passwordHash = await bcrypt.hash(password, salt);

     // Define a senha do usuário como o hash da nova senha
     user.password = passwordHash;
  }

  // Atualiza os dados do usuário no banco de dados e retorna o usuário atualizado
  const updatedUser = await User.findByIdAndUpdate(user._id, user, { new: true });

  return updatedUser;
}
```

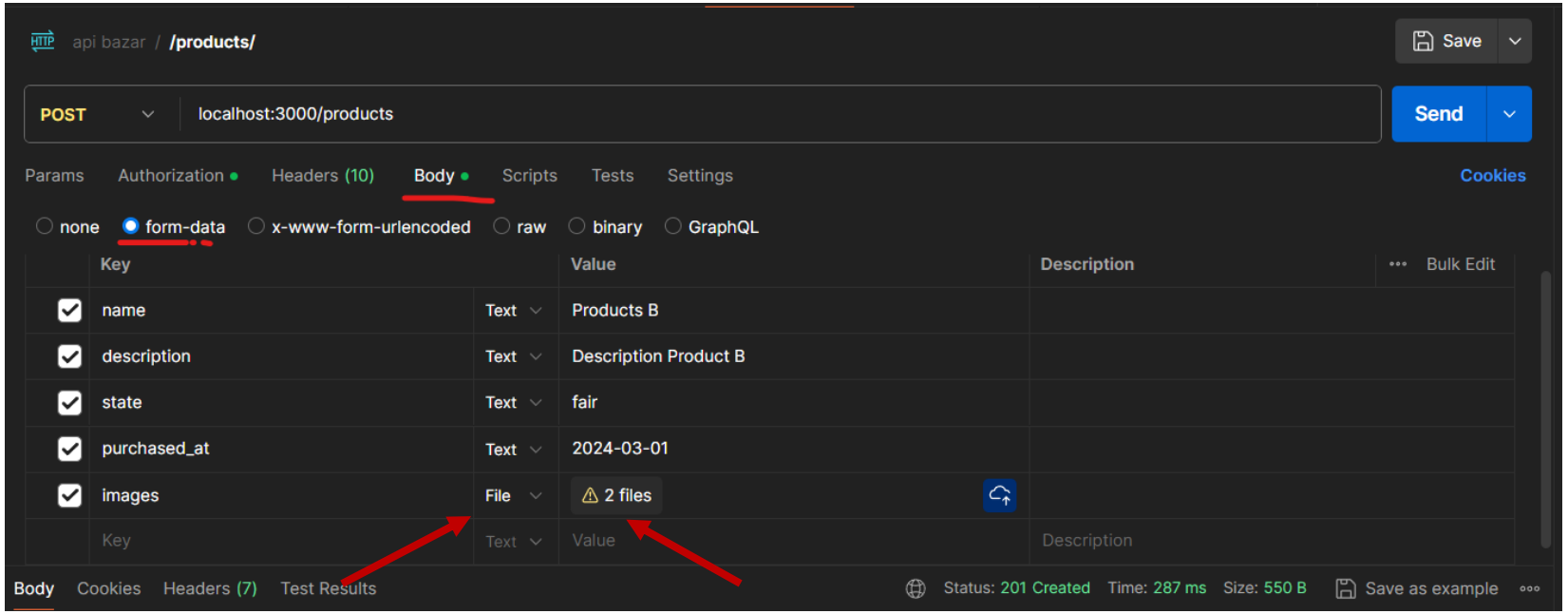
Desenvolvimento

- Arquivo UserRoutes.js

```
router.put("/currentUser", verifyJWT, imageUpload.single("image"), UserController.updateUser);
```

Desenvolvimento

- **Observações**
  - Para as rotas que enviam imagens, utilizar a opção **form-data** do body no postman.



Desenvolvimento

- **Observações**
  - Para as rotas que precisam de autorização, utilizar a opção **Bearer Token** na aba **Authorization** no postman.