INSTITUTO FEDERAL
Paraná
Campus Paranaguá

Ministério da Educação

ENSINO MÉDIO INTEGRADO: INFORMÁTICA
Disciplina de Desenvolvimento Web
Aula 15: Node/Express - CRUD (Banco de Dados / Sequelize)

Gil Eduardo de Andrade

Conceitos Preliminares

(https://expressjs.com/pt-br/)
(https://sequelize.org/docs/)
(https://dev.mysql.com/doc/)

Introdução

O material a seguir tem por objetivo fornecer um guia completo para a criação de uma aplicação CRUD (Create, Read, Update, Delete) utilizando o Node.js, o framework Express.js e o banco de dados relacional MySQL. Para tal será utilizada uma arquitetura em camadas (boas práticas), tendo como foco as camadas de Controle, Serviço e Acesso a Dados, no intuito de garantir manutenibilidade, escalabilidade e organização do código. A aplicação em questão é a mesma demonstrada na aula anterior, mas com a utilização do banco de dados MySQL.

MySQL: Definição

O MySQL é um sistema de gerenciamento de banco de dados relacional (RDBMS) de código aberto, desenvolvido originalmente pela empresa sueca MySQL AB e atualmente mantido pela Oracle Corporation. É um dos bancos de dados mais populares do mundo, amplamente utilizado em aplicações web, desde pequenos projetos pessoais até grandes sistemas corporativos.

MySQL: Principais Características

- Confiabilidade e Estabilidade: é conhecido por sua robustez e capacidade de lidar com grandes volumes de dados e transações simultâneas sem comprometer a integridade dos dados.

- Performance Otimizada: oferece excelente performance para operações de leitura e escrita, com suporte a índices avançados, otimização de consultas e cache de resultados.

- Escalabilidade: suporta tanto escalabilidade vertical (aumento de recursos do servidor) quanto horizontal (replicação e particionamento).

- Compatibilidade com SQL: implementa o padrão SQL (Structured Query Language), facilitando a migração de outros bancos de dados relacionais.

- Segurança Robusta: Oferece recursos avançados de segurança, incluindo criptografia SSL, autenticação baseada em plugins e controle granular de privilégios.

- Multiplataforma: Funciona em diversos sistemas operacionais, incluindo Linux, Windows, macOS e Unix.

Sequelize: Definição

Sequelize é um **ORM(*)** (Object-Relational Mapping) moderno para Node.js, escrito em TypeScript e JavaScript. Ele fornece uma abstração de alto nível sobre bancos de dados relacionais, permitindo que desenvolvedores trabalhem com dados usando objetos JavaScript em vez de escrever SQL diretamente.

*__ORM:__ é uma técnica de programação que mapeia objetos de uma aplicação para tabelas de um banco de dados relacional e vice-versa. Em outras palavras, ele cria uma camada de abstração entre a aplicação e o banco de dados, permitindo que os desenvolvedores manipulem dados usando objetos e classes em vez de escrever consultas SQL diretamente.

Sequelize: Principais Características

- Suporte Multi-Banco: compatível com PostgreSQL, MySQL, MariaDB, SQLite, Microsoft SQL Server e Oracle Database.

- Validações Robustas: sistema de validação integrado que garante a integridade dos dados no nível da aplicação.

- Associações Complexas: suporte completo para relacionamentos entre tabelas (one-to-one, one-to-many, many-to-many).

- Migrações e Seeders: ferramentas para versionamento e evolução do esquema do banco de dados.

- Transações: suporte completo a transações ACID (Atomicidade, Consistência, Isolamento e Durabilidade) para garantir consistência dos dados. As propriedades ACID são essenciais para garantir que as operações de banco de dados sejam tratadas como uma unidade única, e que o estado do banco de dados seja sempre válido, mesmo em caso de falhas

- Hooks e Middleware: sistema de hooks que permite executar código antes ou depois de operações específicas.

- Query Builder: interface fluente para construção de consultas complexas.

- Connection Pooling: gerenciamento automático de pool de conexões para otimização de performance.

RECRIANDO O CRUD “USERS”

UTILIZANDO PADRÃO: Controlador / Serviço / Repositório

Pré-requisitos

- Node.js instalado (versão 14 ou superior)
- NPM (instalado automaticamente com o Node.js)
- MySQL Server (versão 5.7 ou superior)
- Editor de código (VSCode, outros)
- Terminal ou prompt de comando

1ª ETAPA: Configuração Inicial do Projeto

1.1 Criando o diretório do projeto
(no terminal de comando)

Shell

```
mkdir my-crud-mysql
```

```
cd my-crud-mysql
```

1.2 Inicializando o projeto
(no terminal de comando)

```Shell
npm init -y
```

1.3 Instalando as dependências
(no terminal de comando)

```Shell
npm install express@4.21.2 sequelize mysql2 dotenv
```

mysql2
O ***mysql2*** é um pacote do Node.js que fornece uma interface para interagir com bancos de dados MySQL, permitindo a execução de consultas SQL, gerenciamento de conexões com o banco e manipulação de dados de forma eficiente em aplicações Node.js. Em outras palavras, ele facilita a comunicação entre seu código JavaScript e um servidor MySQL.

dotenv
O ***dotenv*** é um pacote npm que permite carregar variáveis de ambiente, a partir de um arquivo ***.env***, para dentro de um projeto Node.js. Tal característica ajuda a manter informações sensíveis, como chaves de API e configurações específicas do ambiente, separadas do código-fonte.

1.4 Criando a estrutura de diretórios
(no terminal de comando)

```Shell
mkdir -p src/controllers src/services src/repositories
          src/routes src/models src/config
```

2ª ETAPA: Instalando e Configurando MySQL

2.1 Instalação

Linux Ubuntu / Debian

```shell
sudo apt update
sudo apt install mysql-server
```

MacOS

```shell
brew install mysql
```

Windows

Baixar o MySQL Installer do site oficial: https://dev.mysql.com/downloads/installer/

OBS.: este tutorial considera que todas as possíveis configurações, necessárias para cada um dos ambientes, foram efetuadas pelo aluno, e que o MySQL está funcionando corretamente.

3ª ETAPA: Configurando as Variáveis e Ambiente

3.1 Criando o arquivo oculto *".env"*

```shell
touch .env
```

3.2 Editando o arquivo oculto *".env"*

```javascript
# Configurações do Banco de Dados MySQL
DB_HOST=localhost
```

```
DB_PORT=3306
DB_NAME=nome_base_dados
DB_USER=usuario_bd
DB_PASSWORD=senha_bd
# Configurações do Servidor
PORT=3000
NODE_ENV=development
```

3.1 Criando o arquivo oculto para documentação *“.env.example”*

Shell

```
touch .env.example
```

3.2 Editando o arquivo oculto para documentação *“.env.example”*

JavaScript

```
# Configurações do Banco de Dados MySQL
DB_HOST=localhost
DB_PORT=3306
DB_NAME=crud_app
DB_USER=root
DB_PASSWORD=

# Configurações do Servidor
PORT=3000
NODE_ENV=development

# Para MySQL em produção, configure:
# DB_HOST=seu-host-mysql
# DB_USER=seu-usuario
# DB_PASSWORD=sua-senha-segura
# DB_NAME=nome-do-banco

# Para usar com Docker MySQL:
# DB_HOST=mysql-container
# DB_USER=root
# DB_PASSWORD=rootpassword
```

```
# DB_NAME=crud_app
```

4ª ETAPA: Configurando a Conexão com MySQL

4.1 Criando o arquivo *"src/config/database.js"*

JavaScript

```
const { Sequelize } = require('sequelize');

// Configuração da conexão com MySQL
const sequelize = new Sequelize(
    process.env.DB_NAME || 'crud_app',
    process.env.DB_USER || 'crud_user',
    process.env.DB_PASSWORD || 'crud_password',
    {
        host: process.env.DB_HOST || 'localhost',
        port: process.env.DB_PORT || 3306,
        dialect: 'mysql',
        logging: process.env.NODE_ENV === 'development' ? console.log
: false,
        pool: {
            max: 5, // número máximo de conexões
            min: 0, // número mínimo de conexões
            acquire: 30000, // tempo máximo para uma conexão
            idle: 10000 // tempo máximo de uma conexão ociosa
        },
        define: {
            timestamps: true, //cria colunas createdAt e updatedAt
            underscored: false, //formato colunas: snake_case ou camelCase
            freezeTableName: true //pluralização automática nome das tabelas
        }
    }
);

const connectDB = async () => {
    try {
        await sequelize.authenticate();
```

```
            console.log('✅ Conexão com MySQL estabelecida com
sucesso.');

        // Sincronizar modelos com o banco de dados
        await sequelize.sync({
            force: false, // Não recriar tabelas se já existirem
            alter: process.env.NODE_ENV === 'development'
            // Alterar estrutura apenas em desenvolvimento
        });
             console.log('✅ Modelos sincronizados com o banco de
dados.');

    } catch (error) {
                console.error('❌ Erro ao conectar com MySQL:',
error.message);
        process.exit(1);
    }
};

// Graceful shutdown
process.on('SIGINT', async () => {
    try {
        await sequelize.close();
             console.log('🔌 Conexão com MySQL fechada devido ao
encerramento da aplicação');
        process.exit(0);
    } catch (error) {
        console.error('❌ Erro ao fechar conexão:', error);
        process.exit(1);
    }
});

module.exports = { sequelize, connectDB };
```

- pool: configurações do pool de conexões para otimizar performance;
- logging: habilita os logs SQL apenas em desenvolvimento;
- sync(): sincroniza modelos com o banco de dados;
- Graceful shutdown: fecha conexões adequadamente ao encerrar a aplicação;

5ª ETAPA: Definindo o Modelo de Dados

5.1 Criando o arquivo *"src/models/User.js"*

JavaScript

```
const { DataTypes } = require('sequelize');
const { sequelize } = require('../config/database');

const User = sequelize.define('User', {
    id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
    },
    name: {
        type: DataTypes.STRING(50),
        allowNull: false,
        validate: {
            notEmpty: {
                msg: 'Nome é obrigatório'
            },
            len: {
                args: [2, 50],
                msg: 'Nome deve ter entre 2 e 50 caracteres'
            }
        }
    },
    email: {
        type: DataTypes.STRING(100),
        allowNull: false,
        unique: {
            name: 'unique_email',
            msg: 'Email já está em uso'
        },
        validate: {
            notEmpty: {
                msg: 'Email é obrigatório'
            },
            isEmail: {
                msg: 'Email deve ser válido'
            }
        },
```

```
        set(value) {
            // Converter para lowercase antes de salvar
            this.setDataValue('email', value.toLowerCase().trim());
        }
    }
}, {
    tableName: 'users',
    timestamps: true, // Adiciona createdAt e updatedAt
    indexes: [
        {
            unique: true,
            fields: ['email']
        }
    ]
});

module.exports = User;
```

Características da Model:

- Define estrutura da tabela “users”
- Inclui validações de dados
- Normaliza e-mail para lowercase
- Adiciona índice único para email
- Timestamps automáticos

6ª ETAPA: Implementando a Camada Repositório

6.1 Criando o arquivo *“src/repositories/userRepository.js”*

JavaScript

```
const User = require('../models/User');
const { Op } = require('sequelize');

class UserRepository {
    async create(userData) {
        try {
```

```
            return await User.create(userData);
        } catch (error) {
            throw error;
        }
    }

    async findAll(options = {}) {
        try {
            const { limit, offset, order } = options;
            return await User.findAll({
                limit,
                offset,
                order: order || [['createdAt', 'DESC']],
                    attributes: { exclude: [] } // Incluir todos os
campos
            });
        } catch (error) {
            throw error;
        }
    }

    async findById(id) {
        try {
            return await User.findByPk(id);
        } catch (error) {
            throw error;
        }
    }

    async findByEmail(email) {
        try {
            return await User.findOne({
                where: {
                    email: email.toLowerCase().trim()
                }
            });
        } catch (error) {
            throw error;
        }
    }
```

```
    async update(id, updateData) {
        try {
             const [updatedRowsCount] = await User.update(updateData,
{
                where: { id },
                returning: true
            });

            if (updatedRowsCount === 0) {
                return null;
            }

            // Buscar o registro atualizado
            return await this.findById(id);
        } catch (error) {
            throw error;
        }
    }

    async delete(id) {
        try {
            const deletedRowsCount = await User.destroy({
                where: { id }
            });
            return deletedRowsCount > 0;
        } catch (error) {
            throw error;
        }
    }

    async count(whereCondition = {}) {
        try {
            return await User.count({
                where: whereCondition
            });
        } catch (error) {
            throw error;
        }
    }

    async findAndCountAll(options = {}) {
```

```
        try {
            const { limit, offset, where, order } = options;
            return await User.findAndCountAll({
                where,
                limit,
                offset,
                order: order || [['createdAt', 'DESC']]
            });
        } catch (error) {
            throw error;
        }
    }

    async search(searchTerm, options = {}) {
        try {
            const { limit, offset } = options;
            return await User.findAll({
                where: {
                    [Op.or]: [
                        {
                            name: {
                                [Op.like]: `%${searchTerm}%`
                            }
                        },
                        {
                            email: {
                                [Op.like]: `%${searchTerm}%`
                            }
                        }
                    ]
                },
                limit,
                offset,
                order: [['createdAt', 'DESC']]
            });
        } catch (error) {
            throw error;
        }
    }
}
```

```
module.exports = new UserRepository();
```

Funcionalidades do repositório:

- Operações CRUD básicas
- Busca por email
- Contagem de registros
- Busca textual
- Paginação
- Tratamento de erros

7ª ETAPA: Implementando a Camada de Serviço

7.1 Criando o arquivo *"src/services/userService.js"*

JavaScript

```
const userRepository = require("../repositories/userRepository");
const { ValidationError, UniqueConstraintError } = require('sequelize');

class UserService {
    async createUser(userData) {
        try {
            // Validação básica
            if (!userData.name || !userData.email) {
                throw new Error("Nome e email são obrigatórios.");
            }

            // Verificar se o email já existe
            const existingUser = await userRepository.findByEmail(userData.email);
            if (existingUser) {
                throw new Error("Email já está em uso.");
            }

            return await userRepository.create(userData);
```

```
        } catch (error) {
            // Tratar erros específicos do Sequelize
            if (error instanceof ValidationError) {
                        const messages = error.errors.map(err =>
err.message);
                throw new Error(messages.join('. '));
            }

            if (error instanceof UniqueConstraintError) {
                throw new Error("Email já está em uso.");
            }

            throw error;
        }
    }

    async getAllUsers(options = {}) {
        try {
            const { page, limit } = options;

            if (page && limit) {
                const offset = (page - 1) * limit;
                return await userRepository.findAndCountAll({
                    limit: parseInt(limit),
                    offset: parseInt(offset)
                });
            }

            return await userRepository.findAll();
        } catch (error) {
                    throw new Error("Erro ao buscar usuários: " +
error.message);
        }
    }

    async getUserById(id) {
        try {
            // Validar se o ID é um número válido
            if (!id || isNaN(id)) {
                throw new Error("ID de usuário inválido.");
            }
```

```
            const user = await userRepository.findById(parseInt(id));
            if (!user) {
                throw new Error("Usuário não encontrado.");
            }
            return user;
        } catch (error) {
            throw error;
        }
    }

    async updateUser(id, userData) {
        try {
            // Validar se o ID é um número válido
            if (!id || isNaN(id)) {
                throw new Error("ID de usuário inválido.");
            }

            // Validações de negócio antes de atualizar
            if (!userData.name && !userData.email) {
                 throw new Error("Pelo menos um campo (nome ou email)
deve ser fornecido para atualização.");
            }

            // Se está atualizando o email, verificar se já existe
            if (userData.email) {
                                        const existingUser = await
userRepository.findByEmail(userData.email);
                if (existingUser && existingUser.id !== parseInt(id))
{
                    throw new Error("Email já está em uso por outro
usuário.");
                }
            }

                                       const updatedUser = await
userRepository.update(parseInt(id), userData);
            if (!updatedUser) {
                     throw new Error("Usuário não encontrado para
atualização.");
            }
```

```
            return updatedUser;
        } catch (error) {
            // Tratar erros específicos do Sequelize
            if (error instanceof ValidationError) {
                        const messages = error.errors.map(err =>
err.message);
                throw new Error(messages.join('. '));
            }

            if (error instanceof UniqueConstraintError) {
                    throw new Error("Email já está em uso por outro
usuário.");
            }

            throw error;
        }
    }

    async deleteUser(id) {
        try {
            // Validar se o ID é um número válido
            if (!id || isNaN(id)) {
                throw new Error("ID de usuário inválido.");
            }

                                        const deleted = await
userRepository.delete(parseInt(id));
            if (!deleted) {
                    throw new Error("Usuário não encontrado para
exclusão.");
            }
            return { message: "Usuário deletado com sucesso." };
        } catch (error) {
            throw error;
        }
    }

    async getUsersCount() {
        try {
            return await userRepository.count();
        } catch (error) {
```

```
                throw new Error("Erro ao contar usuários: " +
error.message);
        }
    }

    async searchUsers(searchTerm, options = {}) {
        try {
            if (!searchTerm || searchTerm.trim() === '') {
                throw new Error("Termo de busca é obrigatório.");
            }

            const { page, limit } = options;
            let searchOptions = {};

            if (page && limit) {
                searchOptions.offset = (page - 1) * limit;
                searchOptions.limit = parseInt(limit);
            }

                return await userRepository.search(searchTerm.trim(),
searchOptions);
        } catch (error) {
                throw new Error("Erro ao buscar usuários: " +
error.message);
        }
    }
}

module.exports = new UserService();
```

Responsabilidades do serviço:

- Validações de negócio
- Verificação de duplicatas
- Tratamento de erros do Sequelize
- Lógica de paginação
- Coordenação de operações

8ª ETAPA: Implementando a Camada de Controle

8.1 Criando o arquivo *"src/controllers/userController.js"*

JavaScript

```
const userService = require("../services/userService");

class UserController {
  async createUser(req, res) {
    try {
      const newUser = await userService.createUser(req.body);
      res.status(201).json({
        success: true,
        message: 'Usuário criado com sucesso',
        data: newUser
      });
    } catch (error) {
      res.status(400).json({
        success: false,
        message: error.message
      });
    }
  }

  async getAllUsers(req, res) {
    try {
      const { page, limit } = req.query;
      const result = await userService.getAllUsers({ page, limit });

      // Se foi paginado, retornar com metadados
      if (result.count !== undefined) {
        const totalPages = Math.ceil(result.count / limit);
        res.status(200).json({
          success: true,
          data: result.rows,
          pagination: {
            currentPage: parseInt(page),
            totalPages,
            totalItems: result.count,
            itemsPerPage: parseInt(limit)
          }
```

```
      });
    } else {
      res.status(200).json({
        success: true,
        data: result
      });
    }
  } catch (error) {
    res.status(500).json({
      success: false,
      message: error.message
    });
  }
}

async getUserById(req, res) {
  try {
    const user = await userService.getUserById(req.params.id);
    res.status(200).json({
      success: true,
      data: user
    });
  } catch (error) {
    res.status(404).json({
      success: false,
      message: error.message
    });
  }
}

async updateUser(req, res) {
  try {
    const updatedUser = await userService.updateUser(
      req.params.id,
      req.body
    );
    res.status(200).json({
      success: true,
      message: 'Usuário atualizado com sucesso',
      data: updatedUser
    });
```

```
    } catch (error) {
      res.status(400).json({
        success: false,
        message: error.message
      });
    }
  }

  async deleteUser(req, res) {
    try {
      const result = await userService.deleteUser(req.params.id);
      res.status(200).json({
        success: true,
        message: result.message
      });
    } catch (error) {
      res.status(404).json({
        success: false,
        message: error.message
      });
    }
  }

  async getUsersCount(req, res) {
    try {
      const count = await userService.getUsersCount();
      res.status(200).json({
        success: true,
        data: { count }
      });
    } catch (error) {
      res.status(500).json({
        success: false,
        message: error.message
      });
    }
  }

  async searchUsers(req, res) {
    try {
      const { q: searchTerm, page, limit } = req.query;
```

```
      if (!searchTerm) {
        return res.status(400).json({
          success: false,
          message: 'Parâmetro de busca "q" é obrigatório'
        });
      }

      const users = await userService.searchUsers(searchTerm, { page,
limit });
      res.status(200).json({
        success: true,
        data: users,
        searchTerm
      });
    } catch (error) {
      res.status(400).json({
        success: false,
        message: error.message
      });
    }
  }
}

module.exports = new UserController();
```

Características dos controladores:

- Formato de resposta padronizado
- Códigos de status HTTP apropriados
- Tratamento de parâmetros de query
- Metadados de paginação
- Validação de entrada

9ª ETAPA: Configurando as Rotas

9.1 Criando o arquivo *"src/routes/userRoutes.js"*

```JavaScript
// src/routes/userRoutes.js

const express = require("express");
const userController = require("../controllers/userController");

const router = express.Router();

// Rotas específicas devem vir antes das rotas com parâmetros
router.get("/users/count", userController.getUsersCount);
router.get("/users/search", userController.searchUsers);
router.post("/users", userController.createUser);
router.get("/users", userController.getAllUsers);
router.get("/users/:id", userController.getUserById);
router.put("/users/:id", userController.updateUser);
router.delete("/users/:id", userController.deleteUser);

module.exports = router;
```

Mapeamento de rotas:

- GET /api/users - Listar usuários
- GET /api/users/:id - Buscar por ID
- POST /api/users - Criar usuário
- PUT /api/users/:id - Atualizar usuário
- DELETE /api/users/:id - Deletar usuário
- GET /api/users/count - Contar usuários
- GET /api/users/search?q=termo - Buscar usuários

10ª ETAPA: Configurando as Rotas

10.1 Criando o arquivo *"src/app.js"*

```JavaScript
const express = require("express");
const { connectDB } = require("./config/database");
const userRoutes = require("./routes/userRoutes");
```

```
const app = express();

// Conectar ao banco de dados
connectDB();

// Middleware para parsear JSON no corpo das requisições
app.use(express.json({ limit: '10mb' }));

// Middleware para parsear dados de formulário URL-encoded
app.use(express.urlencoded({ extended: true, limit: '10mb' }));

// Middleware para CORS (se necessário)
app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', '*');
  res.header('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE,
OPTIONS');
            res.header('Access-Control-Allow-Headers',      'Origin,
X-Requested-With, Content-Type, Accept, Authorization');

  if (req.method === 'OPTIONS') {
    res.sendStatus(200);
  } else {
    next();
  }
});

// Rota de saúde da aplicação
app.get('/health', (req, res) => {
  res.status(200).json({
    success: true,
    status: 'OK',
    message: 'Aplicação funcionando corretamente',
    timestamp: new Date().toISOString(),
    database: 'MySQL',
    version: '1.0.0'
  });
});

// Rota raiz
app.get('/', (req, res) => {
  res.status(200).json({
```

```
    success: true,
    message: 'API CRUD com Node.js, Express e MySQL',
    endpoints: {
      health: '/health',
      users: '/api/users',
      documentation: 'Consulte o README para mais informações'
    }
  });
});

// Monta as rotas de usuário sob o prefixo /api
app.use("/api", userRoutes);

// Middleware para rotas não encontradas
app.use('*', (req, res) => {
  res.status(404).json({
    success: false,
    message: 'Rota não encontrada',
    path: req.originalUrl,
    method: req.method
  });
});

// Middleware de tratamento de erros genérico
app.use((err, req, res, next) => {
  console.error('Erro na aplicação:', err.stack);

  // Erro de validação do Sequelize
  if (err.name === 'SequelizeValidationError') {
    const messages = err.errors.map(error => error.message);
    return res.status(400).json({
      success: false,
      message: 'Erro de validação',
      errors: messages
    });
  }

  // Erro de constraint única do Sequelize
  if (err.name === 'SequelizeUniqueConstraintError') {
    return res.status(400).json({
      success: false,
```

```
      message: 'Violação de restrição única',
      field: err.errors[0]?.path || 'unknown'
    });
  }

  // Erro de conexão com banco de dados
  if (err.name === 'SequelizeConnectionError') {
    return res.status(503).json({
      success: false,
      message: 'Erro de conexão com o banco de dados'
    });
  }

  // Erro de sintaxe SQL
  if (err.name === 'SequelizeDatabaseError') {
    return res.status(500).json({
      success: false,
      message: 'Erro interno do banco de dados'
    });
  }

  // Erro genérico
  res.status(500).json({
    success: false,
    message: 'Erro interno do servidor'
  });
});

module.exports = app;
```

Configurações importantes:

- Conexão com banco de dados
- Middlewares de parsing
- CORS habilitado
- Tratamento de erros específicos do Sequelize
- Rotas de saúde e documentação

(*) **Rota de saúde** é um endpoint simples que serve para verificar se a sua aplicação está funcionando corretamente. Pense nela como um "teste de requisição" para o seu servidor. A rota

geralmente é um GET para um caminho como **/health** ou **/status**. Quando um serviço externo faz uma requisição para essa rota, a aplicação deve responder com **status 200 OK** se tudo estiver bem.

(*) **Rota de documentação** serve para expor a lista de todos os endpoints da API, seus parâmetros, o que eles esperam e o que eles retornam. É como um manual de instruções para os desenvolvedores que precisam usar a API. Você pode criar uma rota, como /docs, que serve uma página HTML ou JSON que descreve toda a sua API. Existem ferramentas populares, como Swagger/OpenAPI ou Postman, que podem gerar essa documentação de forma automática a partir dos seus comentários no código ou de um arquivo de configuração.

11ª ETAPA: Criando o ponto de Entrada

11.1 Criando o arquivo (na raiz) *"index.js"*

JavaScript

```
// Carregar variáveis de ambiente
require('dotenv').config();

const app = require("./src/app");

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`🚀 Servidor rodando na porta ${PORT}`);
  console.log(`📍 Acesse: http://localhost:${PORT}/api/users`);
  console.log(`💚 Health check: http://localhost:${PORT}/health`);
  console.log(`🌍 Ambiente: ${process.env.NODE_ENV || 'development'}`);
  console.log(`🗄️  Banco de dados: MySQL`);
});
```

12ª ETAPA: Executando e Testando a Aplicação

12.1 Iniciando o Servidor

(no terminal de comando)

Shell

```
node index.js
```

```
 % node index.js
[dotenv@17.2.1] injecting env (7) from .env -- tip: 🛠️ run anywhere with `dotenvx run -- yourcommand`
🚀 Servidor rodando na porta 3000
📍 Acesse: http://localhost:3000/api/users
💚 Health check: http://localhost:3000/health
🌎 Ambiente: development
🗄️ Banco de dados: MySQL
Executing (default): SELECT 1+1 AS result
✅ Conexão com MySQL estabelecida com sucesso.
```

12.2 Testando os Endpoints

Verificar se aplicação está funcionando
(no terminal de comando)

Shell
```
curl http://localhost:3000/health
```

```
 % curl http://localhost:3000/health
{"success":true,"status":"OK","message":"Aplicação funcionando corretamente",
```

Criar Usuário
(no terminal de comando)

Shell
```
curl -X POST http://localhost:3000/api/users \
-H "Content-Type: application/json" \
-d '{"name": "João Silva", "email": "joao@email.com"}'
```

```
 % curl -X POST http://localhost:3000/api/users \
-H "Content-Type: application/json" \
-d '{"name": "João Silva", "email": "joao@email.com"}'
{"success":true,"message":"Usuário criado com sucesso","data":{"id":1,"name":"João Silva","email":
"joao@email.com","updatedAt":"2025-08-07T22:38:39.095Z","createdAt":"2025-08-07T22:38:39.095Z"}}%
```

+ Opções

	id	name	email	createdAt	updatedAt
☐ Edita Copiar Apagar	1	João Silva	joao@email.com	2025-08-07 22:38:39	2025-08-07 22:38:39

Listar Usuários
(no terminal de comando)

Shell
```
curl http://localhost:3000/api/users
```

```
 % curl http://localhost:3000/api/users
{"success":true,"data":[{"id":1,"name":"João Silva","email":"joao@email.com","createdAt":"20
25-08-07T22:38:39.000Z","updatedAt":"2025-08-07T22:38:39.000Z"}]}%
```

Buscar Usuário por ID
(no terminal de comando)

Shell

```
curl http://localhost:3000/api/users/1
```

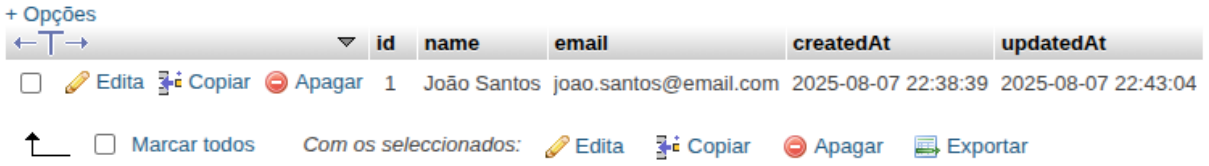
```
 % curl http://localhost:3000/api/users
{"success":true,"data":[{"id":1,"name":"João Silva","email":"joao@email.com","createdAt":"20
25-08-07T22:38:39.000Z","updatedAt":"2025-08-07T22:38:39.000Z"}]}%
```

Atualizar Usuário
(no terminal de comando)

Shell

```
curl -X PUT http://localhost:3000/api/users/1 \
-H "Content-Type: application/json" \
-d '{"name": "João Santos", "email": "joao.santos@email.com"}'
```

```
 % curl -X PUT http://localhost:3000/api/users/1 \
-H "Content-Type: application/json" \
-d '{"name": "João Santos", "email": "joao.santos@email.com"}'
{"success":true,"message":"Usuário atualizado com sucesso","data":{"id":1,"name":"João Santo
s","email":"joao.santos@email.com","createdAt":"2025-08-07T22:38:39.000Z","updatedAt":"2025-
08-07T22:43:04.000Z"}}%
```



Remover Usuário
(no terminal de comando)

Shell

```
curl -X DELETE http://localhost:3000/api/users/1
```

```
 % curl -X DELETE http://localhost:3000/api/users/1
{"success":true,"message":"Usuário deletado com sucesso."}%
```

id	name	email	createdAt	updatedAt

Estrutura Final do Projeto

Textproto

```
my-crud-app/
├── src/
│   ├── config/
│   │   └── database.js
│   ├── controllers/
│   │   └── userController.js
│   ├── models/
│   │   └── User.js
│   ├── repositories/
│   │   └── userRepository.js
│   ├── services/
│   │   └── userService.js
│   ├── routes/
│   │   └── userRoutes.js
│   └── app.js
├── .env
├── .env.example
├── index.js
├── package.json
└── package-lock.json
```