Linguagem de Programação III - Entrega 3 - Gabriel Michel Braucks

diretório.arquivo: src.entidades.interesse

```
import { BaseEntity, Column, CreateDateColumn, Entity, ManyToOne,
PrimaryGeneratedColumn } from
    "typeorm";

import Locatário from "./locatário";
import Residência from "./residência";

@Entity()
export default class Interesse extends BaseEntity {
    @PrimaryGeneratedColumn()
    id: number;
    @Column()
    necessidade_mobília: boolean;
    @Column()
    justificativa: string;
    @CreateDateColumn()
    data_manifestação: Date;
    @ManyToOne(() => Residência, (residência) => residência.interesses, { onDelete:
"CASCADE" })
    residência: Residência;
    @ManyToOne(() => Locatário, (locatário) => locatário.interesses, { onDelete:
"CASCADE" })
    locatário: Locatário;
}
```

diretório.arquivo: src.entidades.locador

```
import { BaseEntity, Column, Entity, JoinColumn, OneToMany, OneToOne,
PrimaryGeneratedColumn } from
    "typeorm";

import Usuário from "./usuário";
import Residência from "./residência";

@Entity()
export default class Locador extends BaseEntity {
    @PrimaryGeneratedColumn()
    id: number;
    @Column()
    anos_experiência: number;
    @Column()
    número_imóveis: number;
    @OneToMany(() => Residência, (residência) => residência.locador)
    residências: Residência[];
    @OneToOne(() => Usuário, (usuário) => usuário.locador, { onDelete:
"CASCADE" })
    @JoinColumn()
    usuário: Usuário;
}
```

diretório.arquivo: src.entidades.locatário

```
import { BaseEntity, Column, Entity, JoinColumn, OneToMany, OneToOne,
PrimaryGeneratedColumn }
    from "typeorm";

import Usuário from "./usuário";
import Interesse from "./interesse";

export enum RendaMensal { ATE2 = "até_2_salarios", DE2A5 = "de_2_a_5_salarios",
ACIMADE5 = "acima_5_salarios" };

@Entity()
export default class Locatário extends BaseEntity {
    @PrimaryGeneratedColumn()
    id: number;
    @Column({ type: "enum", enum: RendaMensal })
    renda_mensal: RendaMensal;
    @Column()
    telefone: string;
    @OneToMany(() => Interesse, (interesse) => interesse.locatário)
    interesses: Interesse[];
    @OneToOne(() => Usuário, usuário => usuário.locatário, { onDelete:
"CASCADE" })
    @JoinColumn()
    usuário: Usuário;
}
```

diretório.arquivo: src.entidades.residência

```
import { BaseEntity, Column, Entity, ManyToOne, OneToMany,
PrimaryGeneratedColumn } from "typeorm";

import Locador from "./locador";
import Interesse from "./interesse";

export enum Categoria { APARTAMENTO = "Apartamento", CASA = "Casa", KITNET
= "Kitnet", LOFT = "Loft" };

@Entity()
export default class Residência extends BaseEntity {
    @PrimaryGeneratedColumn()
    id: number;
    @Column()
    título: string;
    @Column({ type: "enum", enum: Categoria })
    categoria: Categoria;
    @Column()
    localização: string;
    @Column()
    valor_aluguel: number;
    @Column({ type: "date" })
    data_disponibilidade: Date;
    @Column()
    descrição: string;
    @Column()
    mobiliado: boolean;
    @ManyToOne(() => Locador, (locador) => locador.residências, { onDelete:
"CASCADE" })
    locador: Locador;
    @OneToMany(() => Interesse, (interesse) => interesse.residência)
    interesses: Interesse[];
}
```

diretório.arquivo: src.entidades.usuário

```
import { BaseEntity, Column, CreateDateColumn, Entity, OneToOne, PrimaryColumn }
from "typeorm";

import Locador from "./locador";
import Locatário from "./locatário";

export enum Perfil { LOCATÁRIO = "locatário", LOCADOR = "locador" };
export enum Status { PENDENTE = "pendente", ATIVO = "ativo" };

export enum Cores {
   AMARELO = "yellow", ANIL = "indigo", AZUL = "blue", AZUL_PISCINA = "cyan",
   CINZA_ESCURO = "bluegray", LARANJA = "orange", ROSA = "pink", ROXO =
"purple", VERDE = "green",
   VERDE_AZULADO = "teal"
};

@Entity()
export default class Usuário extends BaseEntity {
   @PrimaryColumn()
   cpf: string;
   @Column({ type: "enum", enum: Perfil })
   perfil: Perfil;
   @Column({ type: "enum", enum: Status, default: Status.PENDENTE })
   status: Status;
   @Column()
   nome: string;
   @Column()
   email: string;
   @Column()
   senha: string;
   @Column()
   questão: string;
   @Column()
   resposta: string;
   @Column({ type: "enum", enum: Cores })
   cor_tema: string;
   @OneToOne(() => Locador, (locador) => locador.usuário)
   locador: Locador;
   @OneToOne(() => Locatário, (locatário) => locatário.usuário)
   locatário: Locatário;
   @CreateDateColumn()
   data_criação: Date;
}
```

diretório.arquivo: src.middlewares.verificar-erro-conteúdo-token

```
import md5 from "md5";
import Usuário from "../entidades/usuário";

export default async function verificarErroConteúdoToken(request, response, next) {
    const cpf_encriptado = md5(request.params.cpf || request.body.cpf);
    const usuário_token = await Usuário.findOne({ where: { email:
request.email_token } });
    const usuário = await Usuário.findOne({ where: { cpf: cpf_encriptado } });
    if (usuário_token.email !== usuário.email) return response.status(401).json
        ("Acesso não autorizado.");
    next();
}
```

diretório.arquivo: src.middlewares.verificar-perfil-locador

```
import { Perfil } from "../entidades/usuário";

export default function verificarPerfilLocador(request, response, next) {
    if (request.perfil === Perfil.LOCADOR) return next();
    else return response.status(401).json({ erro: "Acesso não autorizado." });
};
```

diretório.arquivo: src.middlewares.verificar-perfil-locatário

```
import { Perfil } from '../entidades/usuário';

export default function verificarPerfilLocatário(request, response, next) {
    if (request.perfil === Perfil.LOCATÁRIO) return next();
    else return response.status(401).json({ erro: "Acesso não autorizado." });
};
```

diretório.arquivo: src.middlewares.verificar-token

```
import dotenv from 'dotenv';
import { JwtPayload, TokenExpiredError, verify } from "jsonwebtoken";

dotenv.config();

const SENHA_JWT = process.env.SENHA_JWT;

export default function verificarToken(request, response, next) {
    const header = request.headers.authorization;
    if (!header) return response.status(401).json({ erro: "Token nao informado." });
    const token = header.split(' ')[1];
    try {
        const { perfil, email } = verify(token, SENHA_JWT) as JwtPayload;
        request.perfil = perfil;
        request.email_token = email;
        return next();
    } catch (error) {
        if (error instanceof TokenExpiredError) {
            return response.status(401).json({ erro: "Token expirado, faça login
novamente." });
        }
        return response.status(401).json({ erro: "Token invalido." });
    }
};
```

diretório.arquivo: src.rotas.rotas-locador

```
import { Router } from "express";
import verificarToken from "../middlewares/verificar-token";
import verificarPerfilLocador from "../middlewares/verificar-perfil-locador";
import ServiçosLocador from "../serviços/serviços-locador";
import verificarErroConteúdoToken from "../middlewares/verificar-erro-conteúdo-
token";

const RotasLocador = Router();

export default RotasLocador;

RotasLocador.post("/", ServiçosLocador.cadastrarLocador);
RotasLocador.get("/:cpf", verificarToken, verificarPerfilLocador,
ServiçosLocador.buscarLocador);
RotasLocador.patch("/", verificarToken, verificarPerfilLocador,
ServiçosLocador.atualizarLocador);
RotasLocador.post("/residencias", verificarToken, verificarPerfilLocador,
   ServiçosLocador.cadastrarResidência);
RotasLocador.patch("/residencias", verificarToken, verificarPerfilLocador,
   ServiçosLocador.alterarResidência);
RotasLocador.delete("/residencias/:id", verificarToken, verificarPerfilLocador,
   ServiçosLocador.removerResidência);
RotasLocador.get("/residencias/locador/:cpf", verificarToken, verificarPerfilLocador,
   verificarErroConteúdoToken, ServiçosLocador.buscarResidênciasLocador);
RotasLocador.get("/residencias/localizacoes", verificarToken, verificarPerfilLocador,
   ServiçosLocador.buscarLocalizaçõesResidências);
```

diretório.arquivo: src.rotas.rotas-locatário

```
import { Router } from "express";
import verificarToken from "../middlewares/verificar-token";
import verificarPerfilLocatário from "../middlewares/verificar-perfil-locatário";
import ServiçosLocatário from "../serviços/serviços-locatário";
import verificarErroConteúdoToken from "../middlewares/verificar-erro-conteúdo-
token";

const RotasLocatário = Router();

export default RotasLocatário;

RotasLocatário.post("/", ServiçosLocatário.cadastrarLocatário);
RotasLocatário.patch("/", verificarToken, verificarPerfilLocatário,
ServiçosLocatário.atualizarLocatário);
RotasLocatário.get("/:cpf", verificarToken, verificarPerfilLocatário,
ServiçosLocatário.buscarLocatário);

RotasLocatário.post("/interesses/", verificarToken, verificarPerfilLocatário,
    ServiçosLocatário.cadastrarInteresse);
RotasLocatário.delete("/interesses/:id", verificarToken, verificarPerfilLocatário,
    ServiçosLocatário.removerInteresse);
RotasLocatário.get("/interesses/locatario/:cpf", verificarToken, verificarPerfilLocatário,
    verificarErroConteúdoToken, ServiçosLocatário.buscarInteressesLocatário);
RotasLocatário.get("/interesses/residencias/", verificarToken, verificarPerfilLocatário,
    ServiçosLocatário.buscarResidências);
```

diretório.arquivo: src.rotas.rotas-usuário

```
import { Router } from "express";
import ServiçosUsuário from "../serviços/serviços-usuário";
import verificarToken from "../middlewares/verificar-token";
import verificarErroConteúdoToken from "../middlewares/verificar-erro-conteúdo-
token";

const RotasUsuário = Router();

export default RotasUsuário;

RotasUsuário.post("/login", ServiçosUsuário.logarUsuário);
RotasUsuário.post("/verificar-cpf/:cpf", ServiçosUsuário.verificarCpfExistente);

RotasUsuário.patch("/alterar-usuario", verificarToken,
ServiçosUsuário.alterarUsuário);
RotasUsuário.delete("/:cpf", verificarToken, verificarErroConteúdoToken,
ServiçosUsuário.removerUsuário);

RotasUsuário.get("/questao/:cpf", ServiçosUsuário.buscarQuestãoSegurança);
RotasUsuário.post("/verificar-resposta", ServiçosUsuário.verificarRespostaCorreta);
```

diretório.arquivo: src.serviços.serviços-locador

```
import md5 from "md5";
import { getManager } from "typeorm";
import Usuário, { Status } from "../entidades/usuário";

import Locador from "../entidades/locador";
import ServiçosUsuário from "./serviços-usuário";

import Residência from "../entidades/residência";

export default class ServiçosLocador {
    constructor() { }
    static async cadastrarResidência(request, response) {
        try {
            const { título, categoria, localização, valor_aluguel, data_disponibilidade,
descrição, mobiliado,
                cpf } = request.body;
            const cpf_encriptado = md5(cpf);
            const locador = await Locador.findOne({
                where: { usuário: cpf_encriptado },
                relations: ["usuário"]
            });
            await Residência.create({
                título, categoria, localização, valor_aluguel, data_disponibilidade, descrição,
mobiliado, locador
            }).save();
            return response.json();
        } catch (error) { return response.status(500).json({ erro: "Erro BD :
cadastrarResidência" }); }
    };
    static async alterarResidência(request, response) {
        try {
            const { id, título, categoria, localização, valor_aluguel, data_disponibilidade,
descrição, mobiliado } = request.body;
            await Residência.update(id, {
                título, categoria, localização, valor_aluguel, data_disponibilidade, descrição,
mobiliado
            });
            return response.json();
        } catch (error) { return response.status(500).json({ erro: "Erro BD :
alterarResidência" }); }
    };
    static async removerResidência(request, response) {
        try {
            const id_residência = request.params.id;
            const residência = await Residência.findOne(id_residência);
            await Residência.remove(residência);
            return response.json();
```

```
        } catch (error) { return response.status(500).json({ erro: "Erro BD :
removerResidência" }); }
    };
    static async buscarResidênciasLocador(request, response) {
        try {
            const cpf_encriptado = md5(request.params.cpf);
            const residências = await Residência.find({
                where: { locador: { usuário: cpf_encriptado } },
                relations: ["locador", "locador.usuário"]
            });
            return response.json(residências);
        } catch (error) {
            return response.status(500).json
                ({ erro: "Erro BD : buscarResidênciasLocador" });
        }
    };
    static filtrarLocalizaçõesEliminandoRepetição(residências: Residência[]) {
        let localizações: { label: string, value: string }[];
        localizações = residências.filter((residência, índice, residências_antes_filtrar) =>
            residências_antes_filtrar.findIndex
                (residência_anterior => residência_anterior.localização ===
residência.localização) === índice)
            .map(residência => ({ label: residência.localização, value:
residência.localização }));
        return localizações;
    };
    static async buscarLocalizaçõesResidências(request, response) {
        try {
            const residências = await Residência.find();
            const localizações =
ServiçosLocador.filtrarLocalizaçõesEliminandoRepetição(residências);
            return response.json(localizações.sort());
        } catch (error) {
            return response.status(500).json
                ({ erro: "Erro BD : buscarLocalizaçõesResidências" });
        }
    };
    static async cadastrarLocador(request, response) {
        try {
            const { usuário_info, anos_experiência } = request.body;
            const número_imóveis = 0;
            const { usuário, token } = await
ServiçosUsuário.cadastrarUsuário(usuário_info);
            const entityManager = getManager();
            await entityManager.transaction(async (transactionManager) => {
                await transactionManager.save(usuário);
                const locador = Locador.create({ usuário, anos_experiência,
número_imóveis });
                await transactionManager.save(locador);
```

```
            await transactionManager.update(Usuário, usuário.cpf, { status:
Status.ATIVO });
            return response.json({ status: Status.ATIVO, token });
        });
    } catch (error) {
        return response.status(500).json({ erro: error });
    }
};
static async buscarLocador(request, response) {
    try {
        const cpf_encriptado = md5(request.params.cpf);
        const locador = await Locador.findOne({
            where: { usuário: cpf_encriptado },
            relations: ["usuário"]
        });
        if (!locador) return response.status(404).json({ erro: "Locador não
encontrado." });
        return response.json({
            nome: locador.usuário.nome, email: locador.usuário.email,
            anos_experiência: locador.anos_experiência,
            número_imóveis: locador.número_imóveis
        });
    } catch (error) { return response.status(500).json({ erro: "Erro BD :
buscarLocador" }); }
};

static async atualizarLocador(request, response) {
    try {
        const { cpf, anos_experiência } = request.body;
        const cpf_encriptado = md5(cpf);
        await Locador.update({ usuário: { cpf: cpf_encriptado } },
            { anos_experiência });
        return response.json();
    } catch (error) { return response.status(500).json({ erro: "Erro BD :
atualizarLocador" }); }
};
};
```

diretório.arquivo: src.serviços.serviços-locatário

```
import md5 from "md5";
import { getManager } from "typeorm";
import Usuário, { Status } from "../entidades/usuário";
import Locatário from '../entidades/locatário';
import ServiçosUsuário from "./serviços-usuário";

import Residência from "../entidades/residência";
import Interesse from "../entidades/interesse";

export default class ServiçosLocatário {
   constructor() { }
   static async cadastrarInteresse(request, response) {
      try {
         const { id_residência, necessidade_mobília, justificativa, cpf } = request.body;
         const cpf_encriptado = md5(cpf);
         const locatário = await Locatário.findOne({ where: { usuário:
cpf_encriptado } });
         const residência = await Residência.findOne(id_residência);
         const interesses = await Interesse.find({ where: { locatário, residência } });
         if (interesses.length > 0) return response.status(404).json
            ({ erro: "O locatário já cadastrou interesse para a residência." });
         await Interesse.create({ necessidade_mobília, justificativa, locatário,
residência }).save();
         return response.json();
      } catch (error) { return response.status(500).json({ erro: "Erro BD :
cadastrarInteresse" }); }
   };
   static async removerInteresse(request, response) {
      try {
         const id = request.params.id;
         await Interesse.delete(id);
         return response.json();
      } catch (error) { return response.status(500).json({ erro: "Erro BD :
removerInteresse" }); }
   };
   static async buscarInteressesLocatário(request, response) {
      try {
         const cpf_encriptado = md5(request.params.cpf);
         const interesses = await Interesse.find({
            where: { locatário: { usuário: cpf_encriptado } },
            relations: ["locatário", "locatário.usuário", "residência", "residência.locador",
               "residência.locador.usuário"]
         });
         return response.json(interesses);
      } catch (error) {
         return response.status(500).json
            ({ erro: "Erro BD : buscarInteressesLocatário" });
```

```
        }
    };
    static async buscarResidências(request, response) {
        try {
            const residências = await Residência.find({ relations: ["locador",
"locador.usuário"] });
            return response.json(residências);
        } catch (error) { return response.status(500).json({ erro: "Erro BD :
buscarResidências" }); }
    };
    static async cadastrarLocatário(request, response) {
        try {
            const { usuário_info, renda_mensal, telefone } = request.body;
            const { usuário, token } = await
ServiçosUsuário.cadastrarUsuário(usuário_info);
            const entityManager = getManager();
            await entityManager.transaction(async (transactionManager) => {
                await transactionManager.save(usuário);
                const locatário = Locatário.create({ usuário, renda_mensal, telefone });
                await transactionManager.save(locatário);
                await transactionManager.update(Usuário, usuário.cpf, { status:
Status.ATIVO });
                return response.json({ status: Status.ATIVO, token });
            });
        } catch (error) { return response.status(500).json({ erro: error }); }
    };
    static async atualizarLocatário(request, response) {
        try {
            const { cpf, renda_mensal, telefone } = request.body;
            const cpf_encriptado = md5(cpf);
            await Locatário.update({ usuário: { cpf: cpf_encriptado } }, {
                renda_mensal, telefone
            });
            return response.json();
        } catch (error) { return response.status(500).json({ erro: "Erro BD :
atualizarLocatário" }); }
    };
    static async buscarLocatário(request, response) {
        try {
            const cpf_encriptado = md5(request.params.cpf);
            const locatário = await Locatário.findOne({
                where: { usuário: cpf_encriptado },
                relations: ["usuário"]
            });
            if (!locatário) return response.status(404).json({ erro: "Locatário não
encontrado." });
            return response.json({
                nome: locatário.usuário.nome, email: locatário.usuário.email,
                renda_mensal: locatário.renda_mensal, telefone: locatário.telefone
```

```
            });
        } catch (error) { return response.status(500).json({ erro: "Erro BD :
buscarLocatário" }); }
    };
}
```

diretório.arquivo: src.serviços.serviços-usuário

```
import bcrypt from "bcrypt";
import dotenv from 'dotenv';
import md5 from "md5";
import { sign } from "jsonwebtoken";

import Usuário, { Perfil } from "../entidades/usuário";
import Locador from "../entidades/locador";
import Locatário from "../entidades/locatário";

import { getManager } from "typeorm";

dotenv.config();

const SALT = 10;
const SENHA_JWT = process.env.SENHA_JWT;

export default class ServiçosUsuário {
    constructor() { }

    static async verificarCpfExistente(request, response) {
        try {
            const cpf_encriptado = md5(request.params.cpf);
            const usuário = await Usuário.findOne(cpf_encriptado);
            if (usuário) return response.status(404).json({ erro: "CPF já cadastrado." });
            else return response.json();
        } catch (error) {
            return response.status(500).json({ erro: "Erro BD: verificarCpfCadastrado" });
        }
    };

    static async verificarCadastroCompleto(usuário: Usuário) {
        switch (usuário.perfil) {
            case Perfil.LOCADOR:
                const locador = await Locador.findOne({
                    where: { usuário: usuário.cpf },
                    relations: ["usuário"]
                });
                if (!locador) return false;
                return true;
            case Perfil.LOCATÁRIO:
                const locatário = await Locatário.findOne({
                    where: { usuário: usuário.cpf },
                    relations: ["usuário"]
                });
                if (!locatário) return false;
                return true;
            default: return;
```

```
    }
  };

  static async logarUsuário(request, response) {
    try {
      const { nome_login, senha } = request.body;
      const cpf_encriptado = md5(nome_login);
      const usuário = await Usuário.findOne(cpf_encriptado);
      if (!usuário) return response.status(404).json({ erro: "Nome de usuário não
cadastrado." });
      const cadastro_completo = await
ServiçosUsuário.verificarCadastroCompleto(usuário);
      if (!cadastro_completo) {
        await Usuário.remove(usuário);
        return response.status(400).json
          ({ erro: "Cadastro incompleto. Por favor, realize o cadastro
novamente." });
      }
      const senha_correta = await bcrypt.compare(senha, usuário.senha);
      if (!senha_correta) return response.status(401).json({ erro: "Senha
incorreta." });
      const token = sign({ perfil: usuário.perfil, email: usuário.email }, SENHA_JWT,
        { subject: usuário.nome, expiresIn: "1d" });
      return response.json({
        usuárioLogado: {
          nome: usuário.nome, perfil: usuário.perfil,
          email: usuário.email, questão: usuário.questão, status: usuário.status,
          cor_tema: usuário.cor_tema, token
        }
      });
    } catch (error) { return response.status(500).json({ erro: "Erro BD:
logarUsuário" }); }
  };

  static async cadastrarUsuário(usuário_informado) {
    try {
      const { cpf, nome, perfil, email, senha, questão, resposta, cor_tema } =
usuário_informado;
      const cpf_encriptado = md5(cpf);
      const senha_encriptada = await bcrypt.hash(senha, SALT);
      const resposta_encriptada = await bcrypt.hash(resposta, SALT);
      const usuário = Usuário.create({
        cpf: cpf_encriptado, nome, perfil, email,
        senha: senha_encriptada, questão,
        resposta: resposta_encriptada, cor_tema
      });
      const token = sign({ perfil: usuário.perfil, email: usuário.email }, SENHA_JWT,
        { subject: usuário.nome, expiresIn: "1d" });
      return { usuário, senha, token };
```

```
        } catch (error) {
            throw new Error("Erro BD: cadastrarUsuário");
        };
    };

    static async alterarUsuário(request, response) {
        try {
            const { cpf, senha, questão, resposta, cor_tema, email } = request.body;
            const cpf_encriptado = md5(cpf);
            let senha_encriptada: string, resposta_encriptada: string;
            let token: string;
            const usuário = await Usuário.findOne(cpf_encriptado);
            if (email) {
                usuário.email = email;
                token = sign({ perfil: usuário.perfil, email }, SENHA_JWT,
                    { subject: usuário.nome, expiresIn: "1d" });
            }
            if (cor_tema) usuário.cor_tema = cor_tema;
            if (senha) {
                senha_encriptada = await bcrypt.hash(senha, SALT);
                usuário.senha = senha_encriptada;
            }
            if (resposta) {
                resposta_encriptada = await bcrypt.hash(resposta, SALT);
                usuário.questão = questão;
                usuário.resposta = resposta_encriptada;
            }
            await Usuário.save(usuário);
            const usuário_info = {
                nome: usuário.nome, perfil: usuário.perfil, email: usuário.email,
                questão: usuário.questão, status: usuário.status, cor_tema:
usuário.cor_tema, token: null
            };
            if (token) usuário_info.token = token;
            return response.json(usuário_info);
        } catch (error) { return response.status(500).json({ erro: "Erro BD:
alterarUsuário" }); }
    };

    static async removerUsuário(request, response) {
        try {
            const cpf_encriptado = md5(request.params.cpf);
            const entityManager = getManager();
            await entityManager.transaction(async (transactionManager) => {
                const usuário = await transactionManager.findOne(Usuário,
cpf_encriptado);
                await transactionManager.remove(usuário);
                return response.json();
            });
```

```
        } catch (error) {
            return response.status(500).json({ erro: "Erro BD: removerUsuário" });
        }
    };
    static async buscarQuestãoSegurança(request, response) {
        try {
            const cpf_encriptado = md5(request.params.cpf);
            const usuário = await Usuário.findOne(cpf_encriptado);
            if (usuário) return response.json({ questão: usuário.questão });
            else return response.status(404).json({ mensagem: "CPF não cadastrado" });
        } catch (error) {
            return response.status(500).json
                ({ erro: "Erro BD : buscarQuestãoSegurança" });
        }
    };
    static async verificarRespostaCorreta(request, response) {
        try {
            const { cpf, resposta } = request.body;
            const cpf_encriptado = md5(cpf);
            const usuário = await Usuário.findOne(cpf_encriptado);
            const resposta_correta = await bcrypt.compare(resposta, usuário.resposta);
            if (!resposta_correta) return response.status(401).json({ mensagem:
"Resposta incorreta." });
            const token = sign({ perfil: usuário.perfil, email: usuário.email },
                process.env.SENHA_JWT, { subject: usuário.nome, expiresIn: "1h" });
            return response.json({ token });
        } catch (error) {
            return response.status(500).json({ erro: "Erro BD:
verificarRespostaCorreta" });
        }
    };
};
```

diretório.arquivo: src.servidor

```
import cors from "cors";
import express from "express";
import "reflect-metadata";
import { createConnection } from "typeorm";
import RotasUsuário from "./rotas/rotas-usuário";
import RotasLocador from "./rotas/rotas-locador";
import RotasLocatário from "./rotas/rotas-locatário";

const app = express();
const PORT = process.env.PORT
const CORS_ORIGIN = process.env.CORS_ORIGIN;

app.use(cors({ origin: CORS_ORIGIN }));
app.use(express.json());
app.use("/usuarios", RotasUsuário);
app.use("/locadores", RotasLocador);
app.use("/locatarios", RotasLocatário);
app.listen(PORT || 3333);

const conexão = createConnection();

export default conexão;
```

diretório.arquivo: .env

```
NODE_ENV=development
CORS_ORIGIN=http://localhost:3000
TYPEORM_TYPE=mysql
TYPEORM_HOST=localhost
TYPEORM_PORT=3306
TYPEORM_USERNAME=root
TYPEORM_PASSWORD=admin
TYPEORM_DATABASE=banco
SENHA_SISTEMA=Abracadabra2025
SENHA_JWT=2302867d9f6a2a5a0135c823aa740cf1
```

gov.br
Documento assinado digitalmente
GABRIEL MICHEL BRAUCKS
Data: 05/06/2025 17:59:13-0300
Verifique em https://validar.iti.gov.br

Dourados MS, 05/06/2025 ---