**REACT – REACTGRAM**

**Setup do projeto**

**npm init -y** – instalação de pacotes necessários

reactgram>backend>Terminal: **npm init -y**

**npm i bcryptjs** – biblioteca que permite manipular melhor as senhas(do usuário) que vai ter autenticação;

**cors** – nos permite receber requisições da mesma origem, pois trabalharemos com 2 aplicações localhost;

**dotenv** – onde ficará as variáveis de ambiente da aplicação, poderemos salvar por exemplo, o domínio do banco de dados aqui e conseguir resgatar ele em qualquer lugar facilmente;

**express** – framework back-end para criar de fato a API;

**express-validator** –trabalha entre as requisições para validar os dados, isso tira um pouco a lógica de validações dentro dos controllers;

**jsonwebtoken** –essencial para autenticação, gera e verifica tokens para autenticar usuários e deixar tokens ativos;

**mongoose** –ferramenta que será utilizada para trabalhar com banco de dados(não relacional) - MongoDB

**multer** – será utilizado para trabalhar com upload de imagens no geral.

**npm i --save-dev nodemon –** simulará um servidor local

 "server": "nodemon ./app.js"

Executando pacote nodemon no arquivo app.js

**Configurando o dotenv**

***app.js***

require("dotenv").config(); //nos da acesso ao arquivo '.env' onde estão as variáveis de ambiente

const port = process.env.PORT

***.env***

PORT=5000

**Rota de teste da API**

***routes>Router.js***

const express = require("express"); //chamada express

const router = express(); //chamada router

//test route

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

  //função anônima que aceita req e res é padrão

  res.send("API Working!");

})

***app.js – chamada da rota***

const router = require("./routes/Router.js");

app.use(router); //todas as rotas colocadas em routes>Router.js, servirão como base para nossa aplicação

**Postman**Tela de celular com aplicativo aberto

Descrição gerada automaticamenteTela de celular com aplicativo aberto

Descrição gerada automaticamenteImagem de vídeo game

Descrição gerada automaticamente com confiança média

**Importando middlewares**

***app.js***

//Solve CORS - quando as requisições são executadas pelo mesmo domínio

app.use(cors({ credentials: true, origin: "http://localhost:3000" })); //origin - de onde esta vindo a requisição

//Upload diretório - diretório de upload de imagens

app.use("/uploads", express.static(path.join(\_\_dirname, "/uploads")))//juntando nome do diretório atual(\_dirname) com /uploads

**Conexão com o banco**

- Criado novo projeto com o nome de ReactGram no MongoDB;

***config>db.js***

const dbUser = process.env.DB\_USER;

const dbPassword = process.env.DB\_PASSWORD;

//trazendo de '.env'

const conn = async () => {

  try {

    const dbConn = await mongoose.connect(//conectando ao banco de dados

      `mongodb+srv://${dbUser}:${dbPassword}@cluster.fvnirbj.mongodb.net/`

    );//concatenando variáveis q contém user e senha do mongodbproj

    console.log("Coonectou ao banco!")

    return dbConn;

  } catch (error) {//caso de erro na conexão

    console.log(error);

  }

};

conn();

module.exports=conn

***.env***

DB\_USER=aninha

DB\_PASSWORD=iYZJQzn7qExUPv7n

**Criando Model do usuário**

***models>User.js***

const {Schema} = mongoose

//o models é dividido em duas partes: //Schema- esquema(como ele é constituído)

//model - objeto que possui os métodos(inserir, deletar, ler)

const userSchema = new Schema({//dados do usuário

    name: String,

    email: String,

    password: String,

    profileImage: String,//foto de perfil do usuário

    bio: String

}, {

    timestamps:true//2 campós serão criados no model(createdate/updatedate)-com isso qnd usuário é criado e atualizado, o horário e a data são marcados

})

const User = mongoose.model("User", userSchema); //definindo model com o nome de "User" e com o Schema 'userSchema'

module.exports = User;//exportando user

**Criando Model da foto**

***models>Photo.js(mesma estrutura do usuário)***

const photoSchema = new Schema(

  {

    image: String, //será salvo o caminho da imagem, pois no bd não salva imagens

    title: String,

    likes: Array, //array porque os interesses são a quantidade e informações do usuário, isso nos leva a um array de objetos

    comments: Array, //qtd+conteudo\_coment+infor.user

    userId: mongoose.ObjectId, //id-usuario que inseriu a foto - .ObjectId: indica que não é uma string comum e sim de idmongoose(mais complexo)

    userName: String, //nome usuario

  },

  {

    timestamps: true,

  }

);

**Criando Controller do usuário**

***controllers>UserController.js***

***.env***

JWT\_SECRET=thisisoursecret

Quando trabalha-se com JWT é preciso ter um **SECRET**, é uma forma das pessoas que também usam JWT não hackearem o token.

Não há um valor ideal, mas quanto mais complexa(string), o token fica mais seguro.

Isso será utilizado na criação do token e na hora de descriptografar o token.

***UserController.js***

const User = require("../models/User") //importando model

const bcrypt = require("bcryptjs")

const jwt = require("jsonwebtoken")

const { default: mongoose } = require("mongoose")

const jwtSecret = process.env.JWT\_SECRET

//Generate user token

const generateToken = (id) => {

  //espera um id pois ele será inserido no token

  return jwt.sign({ id }, jwtSecret, {

    expiresIn: "7d",

  }) //gerando token e passando id(ser possível fazer alguma manipulação eventualmente), o jwtSecret(criação desse token) e data de expiração(logout automático depois deste tempo)

}

//Register user and sign in

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

  res.send("Registro")

}

module.exports = {

  register

}

***Router.js***

//ROTAS DA APLICAÇÃO

const express = require("express") //chamada express

const router = express() //chamada router

router.use("/api/users", require("./UserRoutes"))//prefixo de api

//test route

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

  //função anônima que aceita req e res é padrão

  res.send("API Working!")

})

module.exports = router//export routers

***Postman***

Tela de celular com aplicativo aberto

Descrição gerada automaticamente

**Iniciando validações**

**middlewares** – **no meio** **de** algumas **requisições** **vai acontecer algo** (por ex.: **entre** uma rota e um registro de fato, acontece algo (esse é o middle));

middler de validação – **pegar erros** que aconteceram nas validações e **devolver de uma forma mais ‘amigável’** para a aplicação no front-end;

Será utilizado o **mesmo código** **para todos** os **middlers** **de validação**;

As **funções de validação** serão utilizadas **nas rotas**.

***middlewares>handleValidation.js***

const { validationResult } = require("express-validator")//método 'validationResult' veio do express-validator

const validate = (req, res, next) => {//os parâmetros recebidos é o padrão do middler

  //next - quando prosseguimos ou não baseado em algum fato que aconteceu na requisição(deixar, prosseguir, parar)

  const errors = validationResult(req) //erros vem da requisição

  //toda requisição que obter um middler de validação retornará possíveis erros

  if (errors.isEmpty()) {

    return next()

  } //se nao tem erro prosseguir//.isEmpty()- verifica se a string de entrada está vazia ou não

  const extractedErrors = []//erros extraídos

  errors.array().map((err) => extractedErrors.push(err.msg))//erros trasnformados em array, logo em seguida é feito um .map e cada erro vira 'err'

  //a mensagem de cada um desses erros vai para 'extractedErrors'

  return res.status(422).json({//todos os erros são retornados e um status é exibido, nesse caso o 422(respostas são passadas em json)

    errors: extractedErrors, //no front-end essa variável será consumida para saber o erro que ocorreu

  })

}

module.exports = validate

***routes>UserRoutes.js***

//Middlewares

const validate = require("../middlewares/handleValidation");

router.post("/register", ***validate***, register); //rota de post+caminho+funcao usada para registro

**Validação de criação do usuário**

***middlewares>useValidations.js***

const { body } = require("express-validator") //entrega tudo oq vem do corpo da requisição

const userCreateValidation = () => {

  //validação da criação de usuário

  return [

    //retorna possíveis erros baseados no body

    body("name") //nome do usuário

      .isString() //validando se nome é uma string

      .withMessage("O nome é obrigatório!!")

      .isLength({ min: 3 }) //nome precisa ter no min 3 letras

      .withMessage("O nome precisa ter no mínimo 3 caracteres."),

..........

 body("confirmpassword")

      .isString()

      .withMessage("A confirmação de senha é obrigatória!!")

***.custom((value, { req }) => {***

***//value-valor recebido do campo***

***if (value != req.body.password) {***

***//comparando valor com senha enviada pela requisição***

***throw new Error("As senhas não são iguais")***

***}***

        return true

      }),

  ]

}

***routes>UserRoutes.js***

const { userCreateValidation } = require("../middlewares/useValidations")

//Routes

router.post("/register", userCreateValidation(), validate, register)

**Registro usuário**

***controllers>UserController.js***

  const { name, email, password } = req.body //usando variáveis vindas do corpo da requisição

//check if user existts-verificando se usuário existe

  const user = await User.findOne({ email }) //findOne- retorna o primeiro documento que ele encontrar na collection que corresponda com o campo name.

  //verificando se model encontra usuário pelo email(pois email é único)

//Generate password hash / gerar hash da senha - uma hash é o que fica salvo no bd depois de salvar uma senha

  const salt = await bcrypt.genSalt() //gera string aleatória

  const passwordHash = await bcrypt.hash(password, salt) //gera senha aleatória para não haver por exemplo de alguém pegar uma senha pelo que há no banco de dados(acessando o banco de dados de forma indevida)

 //create user

  const newUser = await User.create({

    name,

    email,

    password: passwordHash,

  })

//checagem para saber se usuário foi criado com sucesso, retorna o token

  if (!newUser) {

  } //se user n foi criado com sucesso retorna msg de erro(json) + err422

  res.status(201).json({

    \_id: newUser.\_id,

    token: generateToken(newUser.\_id),

  }) //se foi criado com sucesso retorna 201(status de criacao com sucess)+obj em formato json com id do usuário e o token(criado a partir do id do usuário)

***Postman***

Tela de celular

Descrição gerada automaticamente

**Validação login**

***middlewares>userValidations.js***

const loginValidation = () => {

  //validando login(email, senha)

  return [

    body("email")

      .isString()

.......................

module.exports = {

  loginValidation,

}

***controllers>UserController.js***

//fazer login do usuário

const login = (req, res) => {

  res.send("Login")

}

***routes>UserRoutes.js***

const {

  userCreateValidation, // validacao criação de user

  loginValidation, //validacao login

} = require("../middlewares/userValidations");

//Routes

router.post("/login", loginValidation(), validate, login);

//Controller

const { register, login }

**Login user**

***controllers>UserController.js***

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

  const { email, password } = req.body; //chaves vindas do body necessárias para fazer login

  const user = await User.findOne({ email });

  //check if user exists

  if (!user) {

    res.status(404).json({ errors: ["Usuário não encontrado"] });

    return;

  } //se usuário n existir será exibido um err404 + mensagem de erro

  //check if password matches-check se senhas combinam

  if (!(await bcrypt.compare(password, user.password))) {

    //comparando senhas, se não combinam é executado o if

    res.status(422).json({ errors: ["Senha inválida"] });

    return;

  }

  //return user with token(deu tudo certo, não houve erros)

  res.status(200).json({

    \_id: user.\_id,

    profileImage: user.profileImage,

    token: generateToken(user.\_id),

  });

};

**Criando validação de autenticação**

const User = require("../models/User")

const jwt = require("jsonwebtoken") //p ser possível fazer a comparação do token//biblioteca

const jwtSecret = process.env.JWT\_SECRET //apenas com ele é possível realizar de fato o secret//valor

const authGuard = async (req, res, next) => {

  //funcao recebe valores padrão do middler

  //requisições com esse token terá alguns padrões

  const authHeader = req.headers("authorization") //headers terão o 'authorization'

  const token = authHeader && authHeader.split("")[1] //verifica se authHeader existe, pois requisições que não tem o authorization nao tem o token

  //split-permite dividir/separar strings//com("")a string será dividida entre as palavras e com o índice[1] a string fica dividida por letras

  //check if header has a token - verificando se header contém o token

  if (!token) {

    return res.status(401).json({ errors: ["Acesso negado!"] })

  }

  //check if token is valid - checando se token é válido

  try {

    const verified = jwt.verify(token, jwtSecret)//método verify irá comparar o token com o jwtSecret, se tem esse Secret o tokne é válido

    req.user = await User.findById(verified.id).select("-password")//será retornado um obj com todas propriedades do token e senha não será passada por não ser necessário  //findById-usado para buscar um doc ou um registro em um bd pelo id

    //obtém user vindo do id que veio do token, assim é possível extrair dados desse user sem ter que consultar o banco para utilizar user novamente, isso é passado pra frente, logo, outras funções conseguem utilizar esse use vindo da req

    next()

  } catch (error) {

    res.status(401).json({ errors: ["Token inválido!"] })

  }

}

module.exports = authGuard