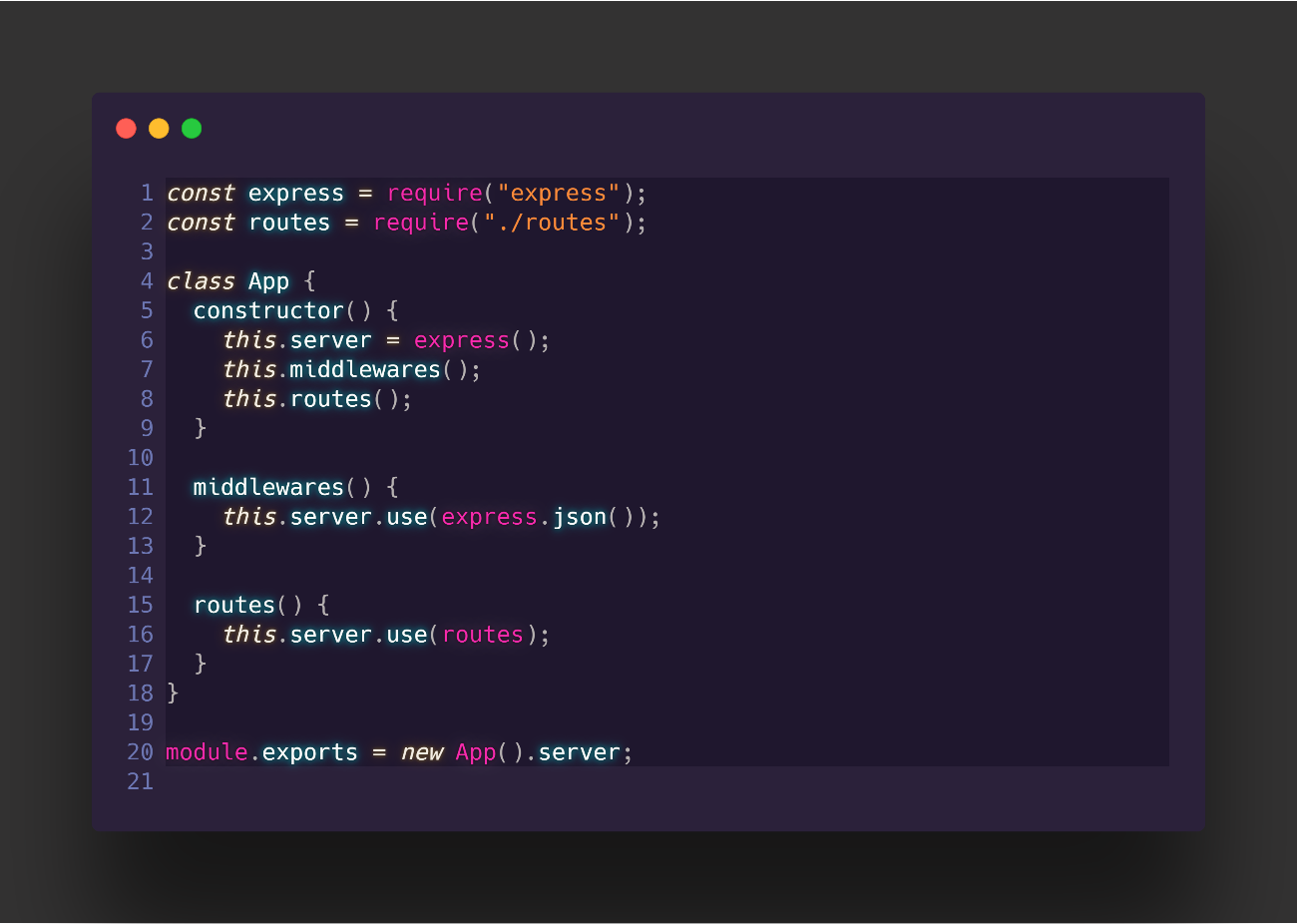
1. **ocConfigurando a estrutura do Projeto**
   1. Crie uma, sugiro que tenha o nome **backendgeek**, para criar via cmd ou terminal use: **mkdir backendgeek**
   2. yarn init -y (para inicializar o projeto)
   3. yarn add express (instalar o pacote do express)
   4. criar pasta **src**
   5. dentro criaremos alguns arquivos:
      1. app.js
      2. server.js
      3. routes.js
   6. no arquivo **app.js,** configuremos a seguinte maneira:

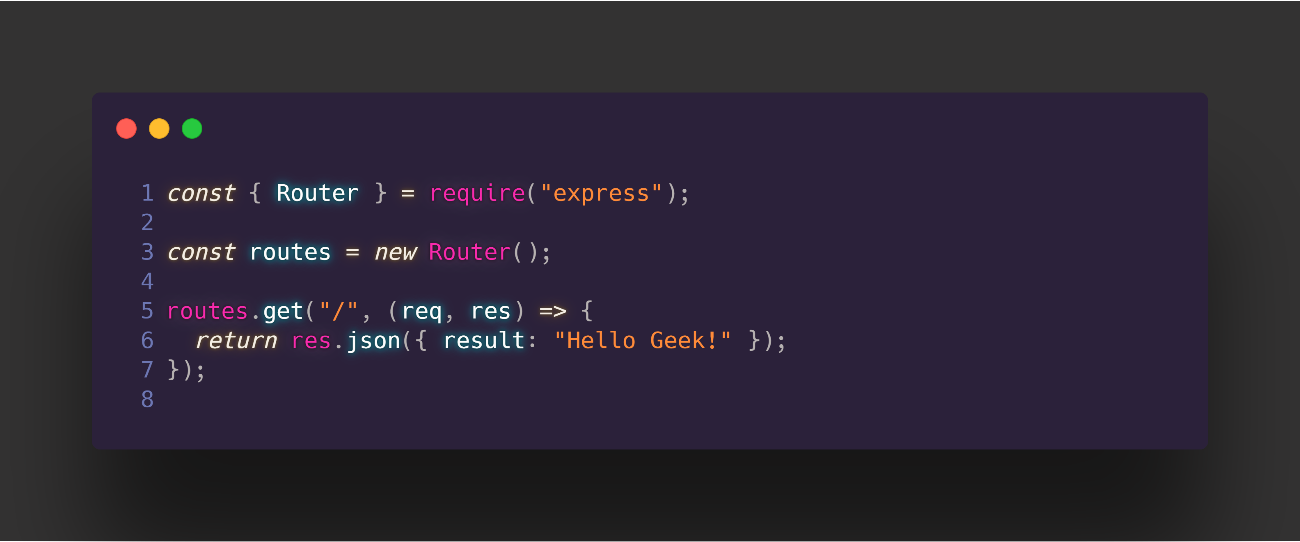


Note que importamos o nosso arquivo de rotas.

* 1. Agora nós iremos trabalhar dentro do nosso **server.js**

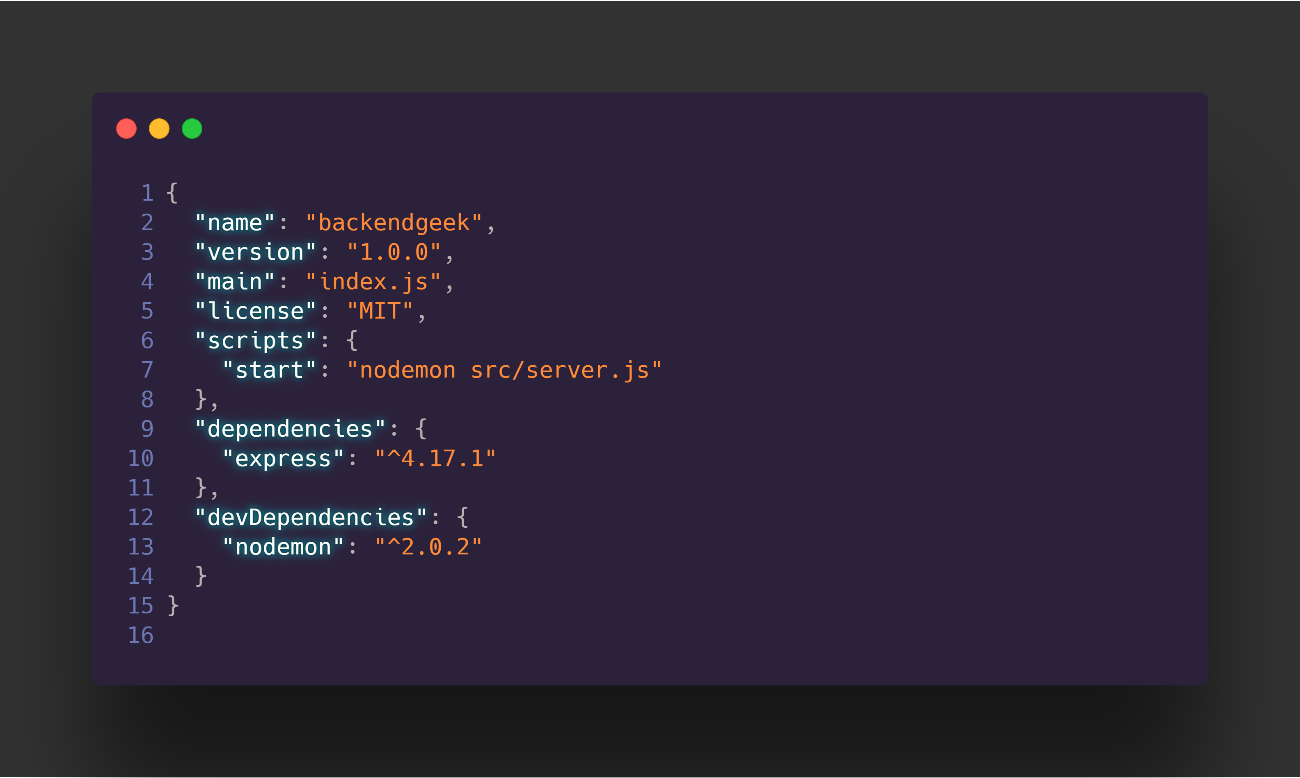


* 1. O nosso arquivo **routes.js** é onde ficará as nossas rotas, você pode cria-lo da seguinte maneira (note que já temos uma rota para testar):



* 1. Agora rodamos no terminal node src/server.js
  2. Basta acessar o seu navegador e digitar localhost:3333 que o result será exibido.

1. **Nessa aula nós iremos adicionar o NODEMON para rodar o nosso servidor.**
   1. yarn add nodemon -D
   2. agora vá até o Package.json e insira os **scripts**, o seu arquivo ficará assim:



* 1. agora você já pode rodar no terminal o comando **yarn start** assim o seu servidor ficará online.
  2. Para utilizarmos import e export default nós iremos instalar o sucrase
     1. Yarn add sucrase -D
     2. Na raiz do projeto criamos um arquivo chamado nodemon.json, que ficará assim:



1. **Nessa aula nós iremos configurar o Eslint, Prettier e Editorconfig**
   1. yarn add eslint -D
   2. yarn eslint --init
      1. To check syntax, find problems, and enforce code style
      2. JavaScript modules (import/export)
      3. None of these
      4. Does your project use TypeScript? (y/N) colocamos **N**
      5. ( ) Browser - (\*) Node (selecionamos o node com a barra de espaço)
      6. Use a popular style guide
      7. Airbnb: <https://github.com/airbnb/javascript>
      8. JavaScript
      9. Would you like to install them now with npm? (Y/n) **Y**
      10. agora deletamos o arquivo Package-lock.json e rodamos o comando **yarn**
   3. agora vamos configurar o **prettier** para trabalhar junto do **eslint**
      1. yarn add prettier
      2. yarn add eslint-config-prettier
      3. yarn add eslint-plugin-prettier
      4. vamos até o arquivo .eslintrc.js e o deixamos da seguinte maneira:
      5. agora criamos um arquivo chamado **.prettierrc** na raiz do nosso projeto, e o deixamos com as seguintes configurações:

module.exports = {

env: {

es6: true,

node: true,

},

extends: [

'airbnb-base', 'prettier'

],

plugins: ['prettier'],

globals: {

Atomics: 'readonly',

SharedArrayBuffer: 'readonly',

},

parserOptions: {

ecmaVersion: 2018,

sourceType: 'module',

},

rules: {

'prettier/prettier': ['error'],

'class-methods-use-this':'off',

'no-param-reassign': "off",

'camelcase': 'off',

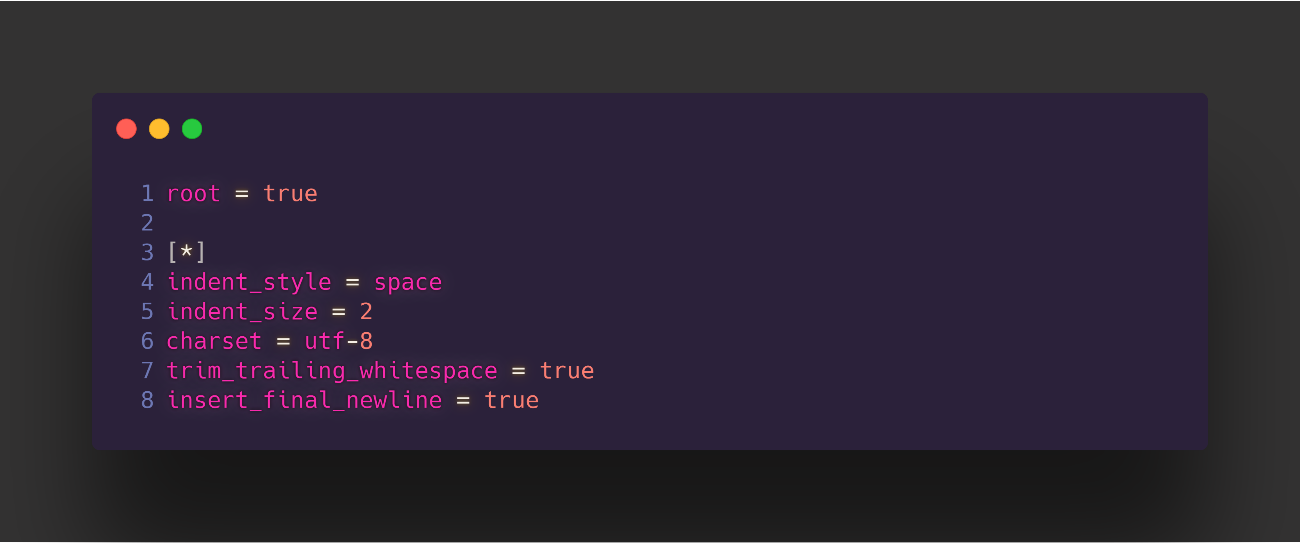
'no-unused-vars': ['error', {"argsIgnorePattern":"next"}]

},

};



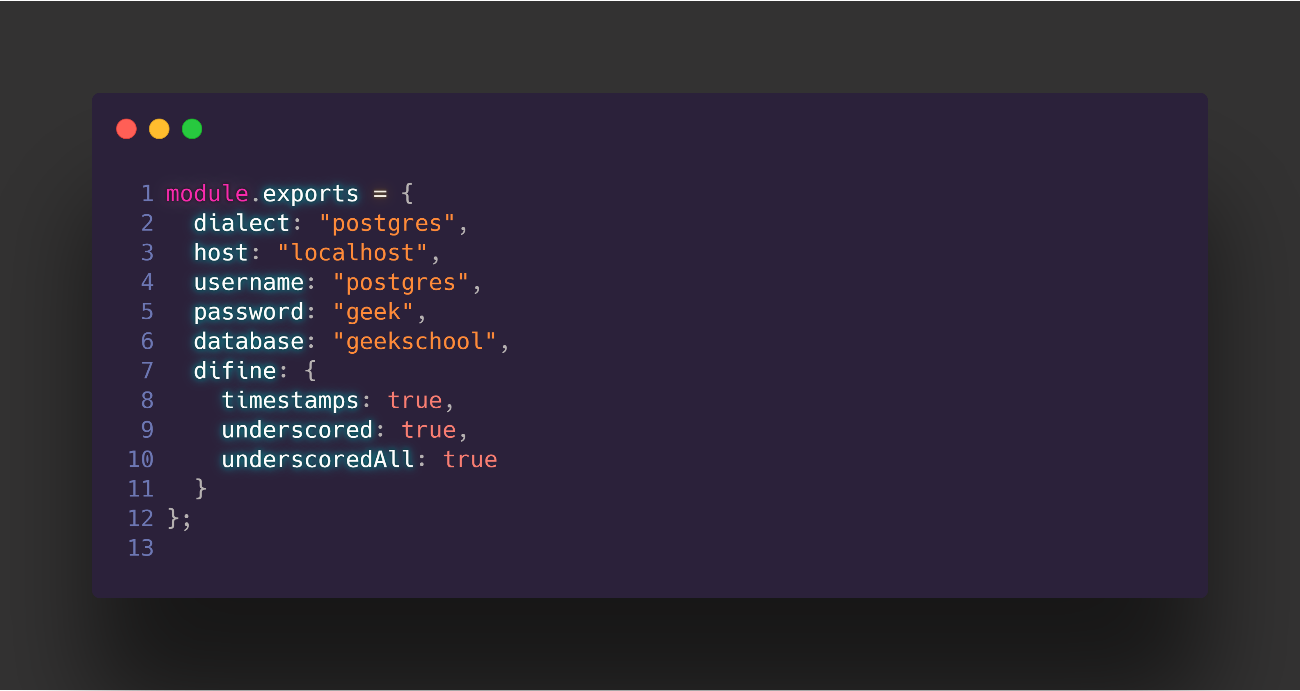
* 1. Agora iremos configurar o editorconfig (é o mais fácil)
     1. Clicamos com o botão direito na raiz do projeto e depois em: Generate editorconfig
     2. Abrimos o arquivo gerado na raiz (.editorconfig) e deixamos da seguinte maneira:



1. **Nessa aula nós iremos trabalhar com Docker**
   1. Criamos o nosso container para o banco de dados mysql
      1. **docker run --name** database **-e POSTGRES\_PASSWORD=**docker -p 5432:5432 **-d postgres**  
         O que está em **azul** é comando do Docker e o que está em **vermelho** é parâmetro.
   2. Agora digite docker ps e veja se o nosso banco de dados **geekdb** está funcionando.
2. **Nessa aula nós iremos configurar o nosso ORM Sequelize**
   1. começamos instalando duas dependências:
      1. yarn add sequelize
      2. yarn add sequelize-cli -D
   2. agora iremos adicionar algumas pastas a estrutura do nosso projeto (pastas dentro de **src**):
      1. criamos uma pasta chamada: **configs**
         1. dentro colocamos um arquivo **db.js**
      2. criamos uma pasta **database**
         1. dentro criamos uma pasta **migrations**
      3. agora criamos uma pasta chamada **apps**
         1. dentro de **apps** criamos uma pasta **models** e outra **controllers**
   3. nesse ponto devemos criar um arquivo na raiz do projeto chamado: **.sequelizerc,** esse arquivo ficará da seguinte maneira:



1. yarn add pg (para usar o postgres)
2. yarn add pg-hstore (para usar o postgres)
3. dentro do nosso arquivo src/configs/db.js faremos a seguinte configuração:



1. **Nessa aula nós iremos criar a nossa primeira migration (users)**
   1. Para criar utilizamos o seguinte comando:
      1. yarn sequelize migration:create --name=create-users
         1. agora dentro da pasta src/database/migrations nós teremos um novo arquivo com um nome cheio de números, esse arquivo ficará assim:



* 1. agora rodamos o seguinte comando: **yarn sequelize db:migrate**
     1. se tudo correr bem, não teremos nenhum erro no console.
     2. Agora basta abrir pgadmin4 e verificar se existe uma tabela **users** lá.

1. **Nessa aula nós iremos criar o nosso Model e Controller de users**
   1. Para isso vamos até a pasta src/apps/models e criamos um arquivo chamado **Users.js**, esse arquivo ficará assim:

import Sequelize, {Model} from 'sequelize'

class User extends Model {

static init (*sequelize*){

**super**.init (

{

name: Sequelize.STRING,

email: Sequelize.STRING,

password\_hash: Sequelize.STRING,

admin: Sequelize.BOOLEAN

},

{

sequelize

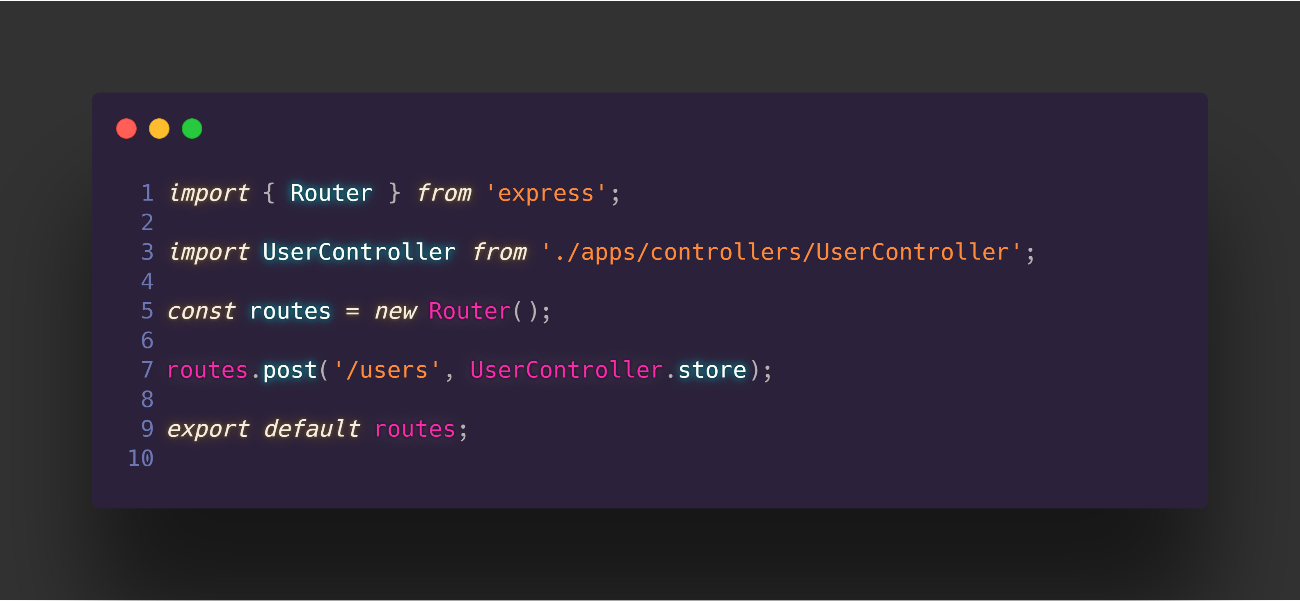
}

);

}

}

export default User;

* 1. Agora criaremos um arquivo UserController.js dentro de src/apps/controllers, nós já criaremos o nosso método store, que ficará assim: 
  2. Depois iremos em routes.js e criaremos a nossa primeira rota para inserir um usuário, que ficará assim:
  3. Agora iremos criar um arquivo que irá fazer a “conexão” entre o nosso banco de dados e o nosso model, esse arquivo ficará dentro de src/database e se chamará **index.js**, o seu conteúdo será o seguinte:

**import Sequelize from 'sequelize'**

import databaseConfig from '../configs/db'

import User from '../apps/models/User'

const models = [User];

class Database {

constructor() {

**this**.init();

}

init() {

**this**.connection = new Sequelize(databaseConfig)

models.map(*model* => model.init(**this**.connection))

}

}

export default new Database();

* 1. Agora dentro de routes.js importamos o arquivo index.js da pasta database, faremos isso da seguinte maneira:

import './database';

1. **Nessa aula nós iremos realizar o teste de inserção de usuário no banco de dados:**
   1. Para isso vamos até o postman e criaremos uma nova rota do tipo **POST:**

{

"name":"Camila Mancini",

"email":"camilamancini@hotmail.com",

"password\_hash":"123456",

"admin":false

}

* 1. agora iremos gerar uma hash para a senha, ou seja, o usuário digitará a senha e nós iremos armazenar essa senha criptografada:
     1. yarn add bcryptjs
     2. agora iremos fazer algumas alterações no nosso model de usuário:

import Sequelize, {Model} from 'sequelize'

import bcrypt from 'bcryptjs'

class User extends Model {

static init (*sequelize*){

**super**.init (

{

name: Sequelize.STRING,

email: Sequelize.STRING,

password: Sequelize.VIRTUAL,

password\_hash: Sequelize.STRING,

admin: Sequelize.BOOLEAN

},

{

sequelize

}

);

**this**.addHook('beforeSave', async (*user*) => {

if (user.password){

user.password\_hash = await bcrypt.hash(user.password\_hash, 8)

}

});

return **this**;

}

}

export default User;

* + 1. No nosso postman agora ao invés de password\_hash, mandaremos um password apenas.
    2. Para testar basta você fazer a inserção de um novo usuário, você verá que agora temos uma senha encriptada.

1. **Nessa aula nós iremos trabalhar com JWT, ou seja a nossa autenticação do usuário:**
   1. Vamos adicionar uma nova extensão, yarn add jsonwebtoken
   2. Para isso começaremos criando um AuthController.
   3. O nosso AuthController ficará da seguinte maneira:

import jwt from 'jsonwebtoken'

import bcrypt from 'bcryptjs'

import User from '../models/User'

class AuthController {

async store (*req*, *res*){

const {email, password} = req.body

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

if (!user){

return res.status(401).json({error: 'User not found!'})

}

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

return res.status(401).json({error: 'Password does not match!'})

}

const {id, name} = user;

return res.json({

user: {

id,

name,

email

},

token: jwt.sign({id}, 'geekplay', {

expiresIn: '3d',

})

})

}bcrypt.compare(password, user.password\_hash

}

export default new AuthController();

* 1. agora vamos em routes.js e adicionaremos mais uma rota

import AuthController from './apps/controllers/AuthController';

routes.post('/auth', AuthController.store);

* 1. agora basta ir no postman e criar uma nova request

{{base\_url}}/auth

{

"email":"joao@gmail.com",

"password":"123456"

}

1. **Nessa aula nós iremos validar se o token está funcionando**
   1. Primeiramente vamos separar as configurações do nosso token , tirando de dentro do nosso AuthController e passando para um outro arquivo externo.
   2. Criaremos um arquivo chamado authConfig.js dentro da pasta configs, esse ficará assim:

export default {

secret: 'geekplay',

expiresIn: '3d'

}

* 1. O nosso AuthController ficará da seguinte maneira:

return res.json({

user: {

id,

name,

email

},

token: jwt.sign({id}, authConfig.secret, {

expiresIn: authConfig.expiresIn,

})

})

* 1. agora nós iremos criar um novo método dentro do nosso UserController. Para teste deixaremos ele da seguinte maneira:

async update (*req*, *res*) {

return res.json({ok: 'Ok'})

}

* 1. criamos agora uma pasta dentro de apps, chamada middlewares, dentro dessa pasta criamos um arquivo authentication.js (esse será responsável por validar o token que gerarmos no sistema), ele ficará da seguinte maneira:

import jwt from 'jsonwebtoken'

import authConfig from '../../configs/authConfig'

function verifyJWT(*req*, *res*, *next*){

const authHeader = req.headers.authorization;

const [, token] = authHeader.split(' ');

jwt.verify(token,authConfig.secret,function(*err*,*verifiedJwt*){

if(err){

console.log(err); *// Token has expired, has been tampered with, etc*

}else{

console.log(verifiedJwt); *// Will contain the header and body*

req.id = verifiedJwt.id;

next();

}

});

}

export default verifyJWT;

* 1. agora iremos em routes e iremos importar algumas novas funcionalidades, bem como criar uma nova rota.

import AuthMiddleware from './apps/middlewares/authentication';

*adicionamos abaixo de todas as outras rotas*

routes.use(AuthMiddleware);

routes.put('/users', UserController.update);

* 1. agora iremos até o nosso postman e faremos o teste, criando um nova rota para testar a *chamada*  do nosso método update do UserController, essa rota ficará assim:

{{base\_url}}/users

1. **Nessa aula nós iremos fazer o método de update do usuário:**
   1. O nosso método ficará da seguinte maneira:

async update (*req*, *res*) {

const {email, oldPassword, password} = req.body;

const user = await User.findByPk(req.id);

if (email && email !== user.email){

const userExists = await User.findOne({where: {email}})

if (userExists){

return res.status(400).json({error: 'User already exists.'})

}

}

if (oldPassword && ! (await bcrypt.compare(oldPassword, user.password\_hash))){

return res.status(401).json({error: 'Password does not match!'})

}

const {id, name, admin} = await user.update(req.body)

return res.json({

id,

name,

email,

admin

})

}

* 1. No postman nós iremos adicionar um json no body da nossa request:

{

"name":"Daniel",

"email":"joao@gmail.com",

"oldPassword":"123456",

"password":"lop32145"

}

1. **Agora nós iremos fazer a validação dos dados que inserimos no nosso banco de dados:**
   1. yarn add @hapi/joi
   2. agora iremos no nosso UserController e modificaremos nosso método store para:

async store(*req*, *res*) {

const schema = Joi.object({

name: Joi.string().required(),

password: Joi.string().required(),

email: Joi.string()

.email()

.required(),

admin: Joi.boolean(),

}).required();

const validator = schema.validate(req.body);

if (validator.error) {

return res.status(400).json({ error: 'Validation Fails.' });

}

const userExists = await User.findOne({ where: { email: req.body.email } });

if (userExists) {

return res.status(400).json({ error: 'User already exists.' });

}

const { id, name, email, admin } = await User.create(req.body);

return res.json({

id,

name,

email,

admin,

});

}

* 1. faremos o mesmo para o método update:

async update(*req*, *res*) {

const { email, oldPassword, password, confirmPassword } = req.body;

const schema = Joi.object({

name: Joi.string().required(),

password: Joi.string().required(),

email: Joi.string()

.email()

.required(),

admin: Joi.boolean(),

oldPassword: Joi.string(),

confirmPassword: Joi.string(),

}).required();

const user = await User.findByPk(req.id);

const validator = schema.validate(req.body);

if (validator.error) {

return res.status(400).json({ error: 'Validation Fails.' });

}

if (email && email !== user.email) {

const userExists = await User.findOne({ where: { email } });

if (userExists) {

return res.status(400).json({ error: 'User already exists.' });

}

}

if (oldPassword && !(await bcrypt.compare(oldPassword, user.password\_hash))) {

return res.status(401).json({ error: 'Password does not match!' });

}

if (oldPassword && password && confirmPassword && !(password === confirmPassword)) {

return res.status(401).json({ error: 'The confirmation password failed!' });

}

const { id, name, admin } = await user.update(req.body);

return res.json({

id,

name,

email,

admin,

});

}

* 1. Por ultimo iremos adicionar essa funcionalidade ao nosso authcontroller, no método store:

const schema = Joi.object({

email: Joi.string()

.email()

.required(),

password: Joi.string().required(),

}).required();

const validator = schema.validate(req.body);

if (validator.error) {

return res.status(400).json({ error: 'Validation Fails.' });

}