Para este proyecto vamos a necesitar tener instalado nodejs, typeorm,adicionalmente vamos a instalar express, tenemos que tener instalado typescrip, mysql, para manejar los roles vamos a utilizar jwt.

Primero vamos a instalar de manera global typeorm, para ello abrimos una terminal como administrador.

>>npm i typeorm -g

Con esta dependencia instalada se nos permite ejecutar un comando que nos crea la arquitectura inicial del proyecto node js.

>>typeorm init --name login\_roles --database mysql --express

Ahora vamos aingresar a la carpeta login\_roles

>>cd login\_roles

Vamos a abrir esta carpeta en visual studio code, podemos ver que nos crea los archivos necesarios para comenzar el proyecto, en tsconfig.json tenemos la configuración de typescrip, en el package.json tenemos las dependencias que vamos a necesitar instalar, el ormconfig es la configuración de nuestra base de datos.

Vamos a inicar el proyecto con las dependencias en el package.json

{

   "name": "login\_roles",

   "version": "0.0.1",

   "description": "Awesome project developed with TypeORM.",

   "devDependencies": {

      "ts-node": "3.3.0",

      "@types/node": "^8.0.29",

      "typescript": "3.3.3333"

   },

   "dependencies": {

      "typeorm": "0.2.37",

      "reflect-metadata": "^0.1.10",

      "mysql": "^2.14.1",

      "express": "^4.15.4",

      "body-parser": "^1.18.1"

   },

   "scripts": {

      "start": "ts-node src/index.ts"

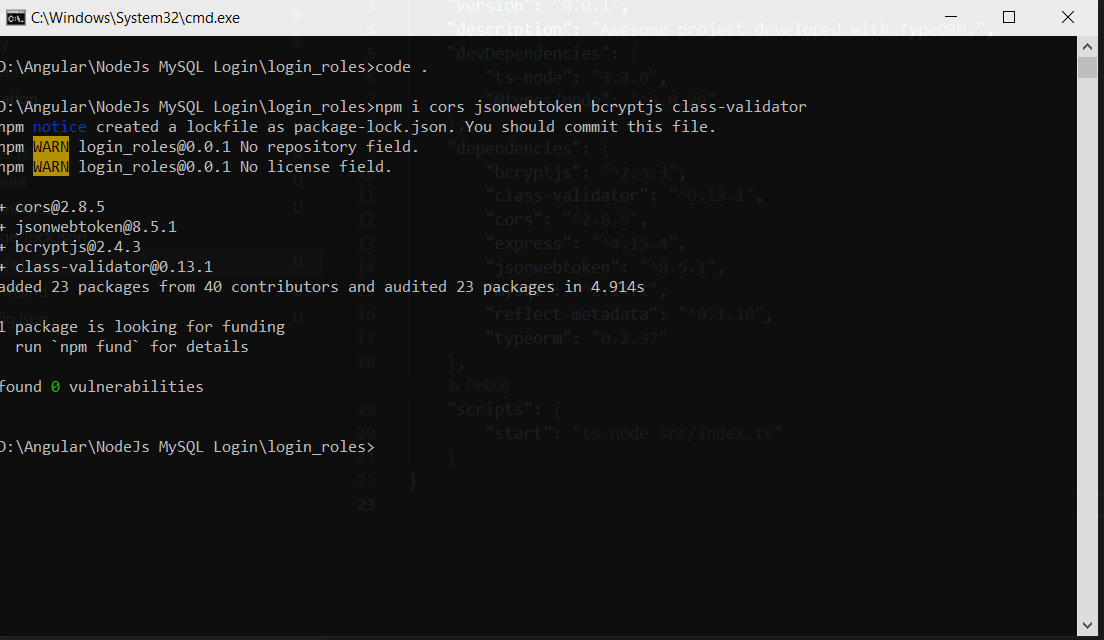
   }

}

Vamos a eliminar la dependencia de typescipt por que quiero utilizar una versión más reciente, también voy a eliminar body-parser esta dependencia lo que hace es que me permite comunicarme con el cliente, pero esto ya lo puedo hacer directamente con express.

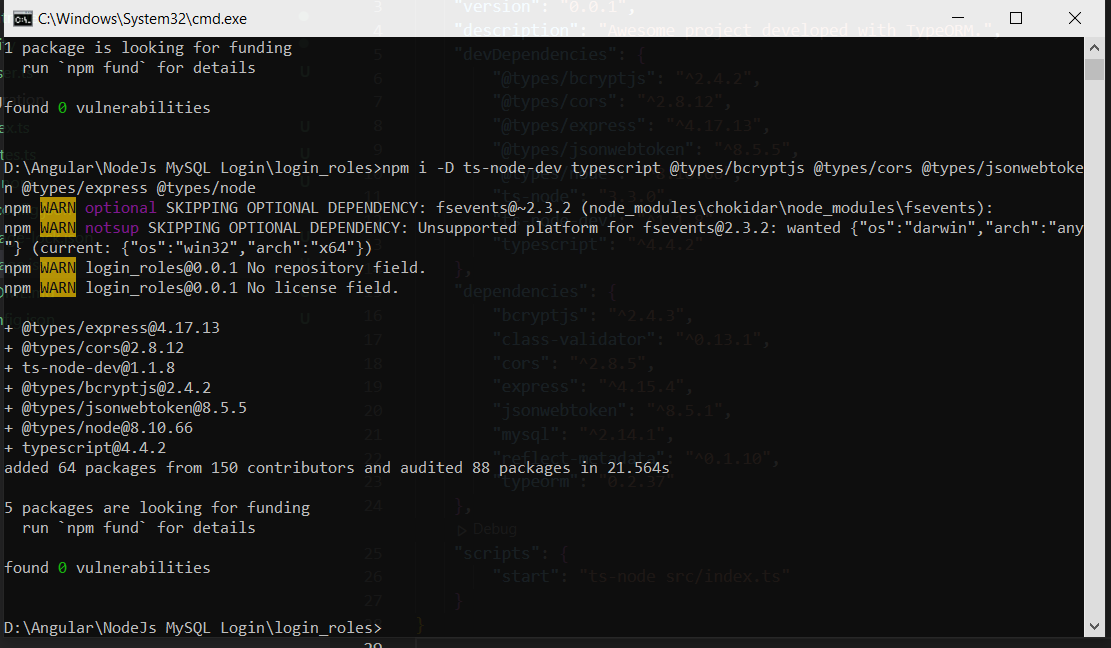
Ahora vamos a instalar las dependencias que nos hacen falta, vamos a instalar cors para poder aceptar todas las peticiones que nos hagan de un frontend; también necesitamos jsonwebtoken que nos va a permitir generar un token para nuestro usuario logeado, bcryptjs para poder encriptar la contraseña en la base de datos, una dependencia que se llama class-validator esta nos ayuda a hacer las validaciones de nuestros campos en la base de datos.

>>npm i cors jsonwebtoken bcryptjs class-validator



También vamos a necesitar otras dependencias pero estas estarán solo en modo de producción por que no queremos compilarlas con el resto del proyecto, vamos a instalar ts-node-dev que nos va a ayudar a reiniciar nuestro servidor, typescript, también vamos a necesitar instalar unos types, @types/bcryptjs, @types/cors, @types/jsonwebtoken, @types/express, @types/node

>>npm i -D ts-node-dev typescript @types/bcryptjs @types/cors @types/jsonwebtoken @types/express @types/node



En el archive de configuracion de typescript podemos ver las opciones, vamos a cambiar el nombre del directorio de compilación en vez de build a dist, vamos a agregar una nueva propiedad que se llama rootDir.

{

   "compilerOptions": {

      "lib": [

         "es5",

         "es6"

      ],

      "target": "es5",

      "module": "commonjs",

      "moduleResolution": "node",

      "outDir": "./dist",

      "rootDir": "./src",

      "emitDecoratorMetadata": true,

      "experimentalDecorators": true,

      "sourceMap": true

   }

}

Como podemos ver en el proyecto ya tenemos la arptea con el node\_modelu pero aun no tenemos todos los modulos instalados, para ello vamos a eliminar esta carptea y ejecutamos el comando npm i

>>npm i

Ahora vamos a configurar los scripts del package.json, primero voy a crear los scripts.

"scripts": {

        "tsc": "tsc",

        "dev": "set debug=\* && ts-node-dev --respawn --transpile-only ./src/index.ts"

    }

Ahora vamos a configurar el orm este nos permite tener incluso mas bases de datos configuradas, de tal manera de tener hasta microservicios corriendo en diferentes servidores.

{

    "type": "mysql",

    "host": "localhost",

    "port": 3306,

    "username": "root",

    "password": "admin",

    "database": "login\_node",

    "synchronize": true,

    "logging": false,

    "entities": [

        "src/entity/\*\*/\*.ts"

    ],

    "migrations": [

        "src/migration/\*\*/\*.ts"

    ],

    "subscribers": [

        "src/subscriber/\*\*/\*.ts"

    ],

    "cli": {

        "entitiesDir": "src/entity",

        "migrationsDir": "src/migration",

        "subscribersDir": "src/subscriber"

    }

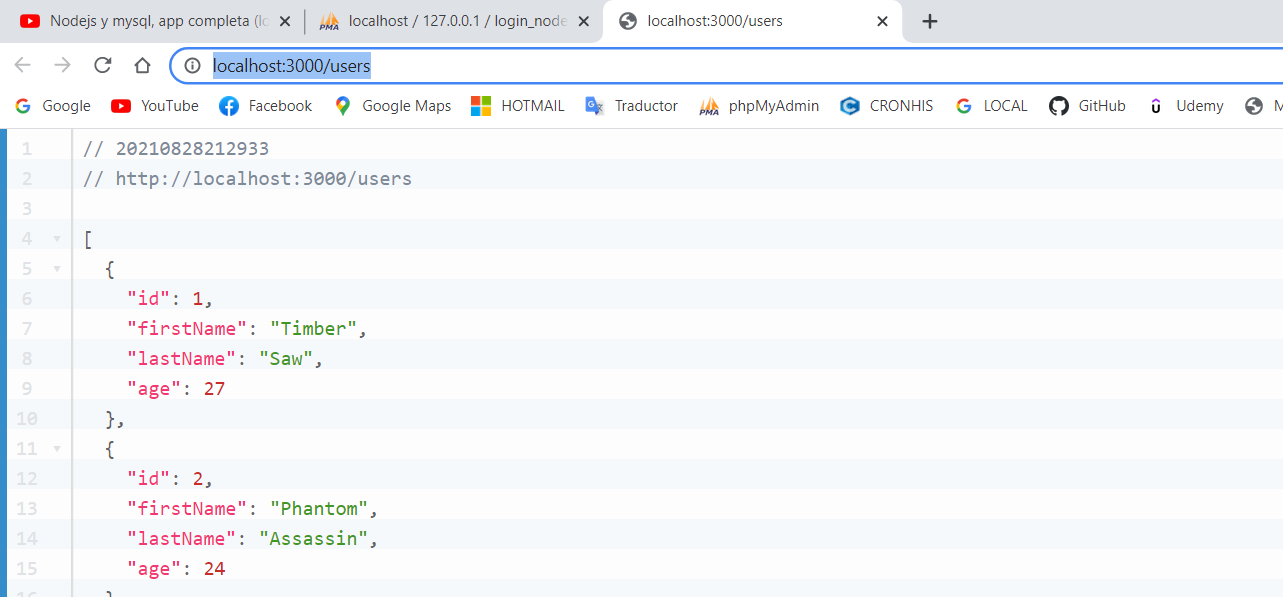
}

Una vez configurada nuestra conexión a la base de datos vamos a ejecutar el comando que creamos.

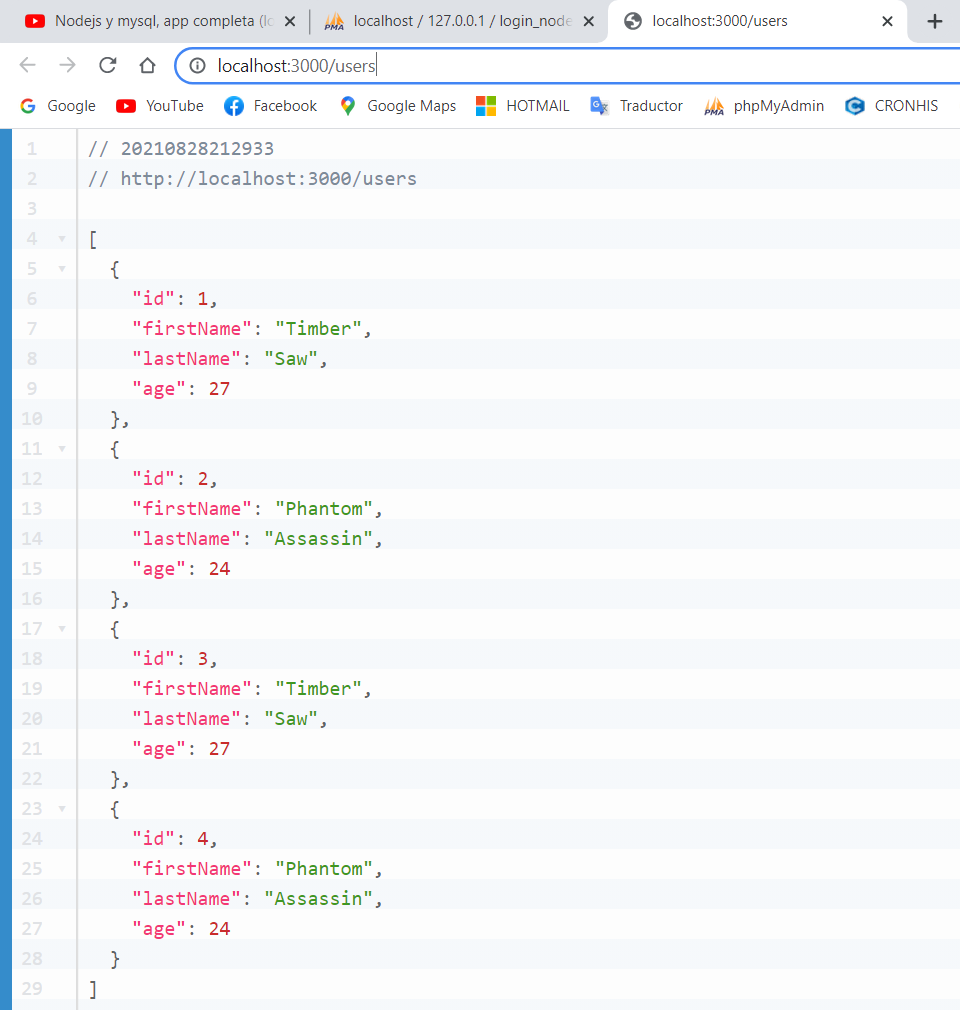
>>npm run dev

Ahora nos dice que si queremos ver el endpoint que ha creado vamos a la siguiente dirección,

<http://localhost:3000/users>



Podemos ver que nos crea los usuarios que está en nuestro código del index, si cancelamos el servicio y lo volvemos a arranar vemos que nos vuelve a crear los usuarios.



Hasta aquí vamos bien se han creado el servidor con las dependencias que necesitamos, vamos a configurar el archivo index.

Vamos a instalar un paquete qye hace que nuestra aplicación sea mas segura

>>npm i helmet

Los imports serian los siguientes

import "reflect-metadata";

import {createConnection} from "typeorm";

import \* as express from "express";

import {Request, Response} from "express";

import \* as cors from "cors";

import \* as helmet from "helmet";

creamos una constante para obtener el puerto si el proyecto esta en un servidor o lo dejamos en el puerto 3000 si estamos en desarrollo.

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

el método de conexión quedaría así.

createConnection().then(async () => {

    // create express app

    const app = express();

    // Middlewares

    app.use(cors());

    app.use(helmet());

    app.use(express.json());

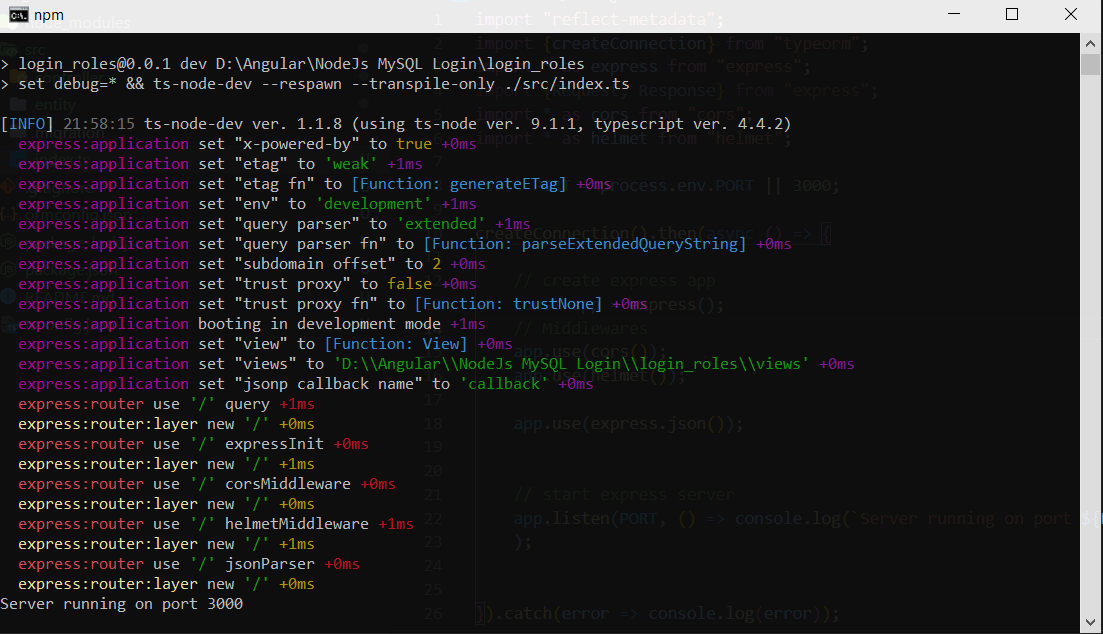
    // start express server

    app.listen(PORT, () => console.log(`Server running on port ${PORT}`)

    );

}).catch(error => console.log(error));

Si lo ejecutamos vemos que nos dice que el servidor esta corriendo en el puerto 3000



Vamos a crear una carpeta de rutas y dentro del tres archivos index.ts, auth.ts, user.ts

Ahora dentro de la carpeta controler vamos a crear el archivo AuthController.ts

Vamos a empezar a trabajar con el index.ts de las rutas.

import { Router } from 'express';

import auth from './auth';

import user from './user';

const routes = Router();

// localhost:3000/auth/login

routes.use('/auth', auth)

// localhost:3000/users

// localhost:3000/users/id

routes.use('/users', user)

export default routes;

este archive es simple y solo tiene la configuración de los nombres de las rutas, ahora vamos a trabajar en las rutas.

El archivo de la ruta auth solo tendrá el método del login que se pasa por el verbo post del html.

import { Router  } from "express";

import AuthController from "../controller/AuthController";

const router = Router()

// login

router.post('/login', AuthController.login);

export default router;

El archivo de la ruta user tendrá el crud de la entidad user.

import { Router } from "express";

import UserController from '../controller/UserController';

const router = Router();

// Get all users

router.get('/', UserController.getAll);

// Get one user

router.get('/:id', UserController.getById);

// Create a new user

router.post('/', UserController.newUser);

// Edit user

router.patch('/:id', UserController.editUser);

// Delete user

router.delete('/:id', UserController.deleteUser);

export default router;

ahora vamos a definir la entidad de los usuarios.

import {Entity, PrimaryGeneratedColumn, Column, Unique, CreateDateColumn, Up

dateDateColumn } from "typeorm";

import { MinLength, IsNotEmpty } from "class-validator";

// TODO IsEmail

@Entity()

@Unique(['username'])

export class User {

    @PrimaryGeneratedColumn()

    id: number;

    @Column()

    @MinLength(6)

    username: string;

    @Column()

    @MinLength(6)

    password: string;

    @Column()

    @IsNotEmpty()

    role: string;

    @Column()

    @CreateDateColumn()

    create\_at: Date;

    @Column()

    @UpdateDateColumn()

    update\_at: Date;

}

Ahora podemos configurar los controladores de auth y user.

El controlador de auth queda asi.

import { getRepository  } from "typeorm";

import { Request, Response } from "express";

import { User } from '../entity/User';

class AuthController {

    static login = async ( req: Request, res: Response )=> {

        const { username, password } = req.body;

        if( !(username && password )){

            return res.status(400).json({message:'User name & password are required!'});

        }

        const userRepository = getRepository(User);

        let user: User;

        try{

            // user = await userRepository.findOneOrFail({ where:{username:username}});

            user = await userRepository.findOneOrFail({ where:{username}});

        }

        catch(e){

            return res.status(400).json({message:'Username or Password incorecct!'})

        }

        res.send(user);

    };

}

export default AuthController;

El controlador de user queda asi.

import { getRepository } from "typeorm";

import { Request, Response } from "express";

import { User } from '../entity/User';

import { validate } from "class-validator";

export class UserController {

    static getAll = async ( req: Request, res: Response ) =>{

        const userRepository = getRepository(User);

        const users = await userRepository.find();

        if( users.length > 0){

            res.send(users);

        } else {

            res.status(404).json({message:'Not results'});

        }

    };

    static getById = async ( req: Request, res: Response ) => {

        const { id } = req.params;

        const userRepository = getRepository(User);

        try {

            const user = await userRepository.findOneOrFail(id);

            res.send(user);

        }

        catch (e){

            res.status(404).json({message:'Not results'})

        }

    };

    static newUser = async (req: Request, res: Response ) => {

        const { username, password, role } = req.body;

        const user = new User();

        user.username = username;

        user.password = password;

        user.role = role;

        // validate

        const errors = validate(user);

        if( (await errors).length > 0 ){

            return res.status(400).json(errors);

        }

        const userRepository = getRepository(User);

        try {

            await userRepository.save(user);

        }

        catch (e) {

            return res.status(409).json({message:'User name already exist'});

        }

        // all ok

        res.send('User created');

    };

    static editUser = async (req: Request, res: Response ) => {

        let user;

        const { id } = req.params;

        const { username, role } = req.body;

        const userRepository = getRepository(User);

        // Try get user

        try{

            user = await userRepository.findOneOrFail(id);

            user.username = username;

            user.role = role;

        }

        catch(e){

            return res.status(404).json({message:'User not found'})

        }

        const errors = await validate(user);

        if( errors.length > 0 ){

            return res.status(400).json(errors);

        }

        // try to save

        try {

            await userRepository.save(user);

        }

        catch(e){

            return res.status(409).json({message:'User name already in use'});

        }

        res.status(201).json({message:'User update'});

    };

    static deleteUser = async (req: Request, res: Response ) => {

        const { id } = req.params;

        const userRepository = getRepository(User);

        let user: User;

        try{

            user = await userRepository.findOneOrFail(id);

        }

        catch(e){

            return res.status(404).json({message:'User not found'});

        }

        // Remove user

        userRepository.delete(id);

        res.status(201).json({message:'User deleted'});

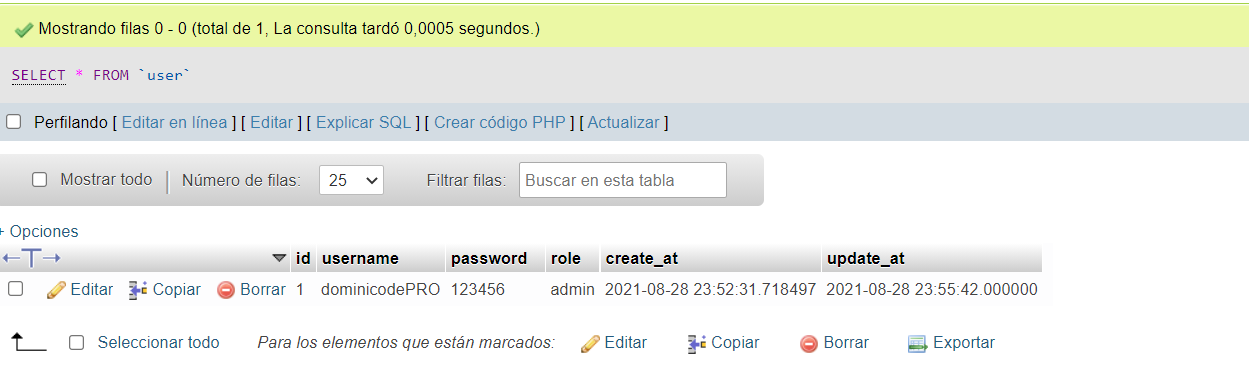
    };

}

export default UserController;

ahora vamos a probar los métodos con postman.

Podemos ver que todos los métodos funcionan en el crud y vemos que en el login nos hace la búsqueda por usuario y nos retorna el objeto con la información, el problema es que aun no se tiene en cuenta la contraseña y además esta esta visible en el servidor la cual debería estar encriptada.



Para esto vamos a utilizar la dependencia bcryptjs, de este paquete vamos a utilizar dos métodos.

hashSync, este método recibe dos parámetros uno es el string al que le vamos a realizar el hash, seria la contraseña y a continuación el salt que es el largo de la encriptación que deseamos realizar que por defecto esta en 10.

compareSync, este método un string, recibe el hash generado y retorna un true o false.

Ahora vamos a la entity de nuestros usuarios, y aquí vamos a crear los dos métodos.

// Metodos

    // Encriptar password

    hashPassword():void{

        this.password = bcrypt.hashSync(this.password, 10 );

    }

    checkPassword( password: string ):boolean{

        return bcrypt.compareSync(password,this.password)

    }

Vamos a poner en práctica estos métodos, para ello vamos al user controler y vamos al método de newUser y en el try de save(user) vamos a poner el método de hashPassword()

const userRepository = getRepository(User);

        try {

            user.hashPassword();

            await userRepository.save(user);

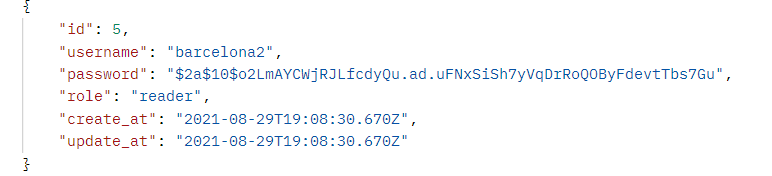
        }

        catch (e) {

            return res.status(409).json({message:'User name already exist'});

        }

Podemos ver que la contraseña de nuestro nuevo usuario ya esta encriptada.



Ahora en el método de login debemos desencriptar la contraseña para poder ingresar, después de comprobar si existe el usuario vamos a verificar que la contraseña sea correcta.

// Check password

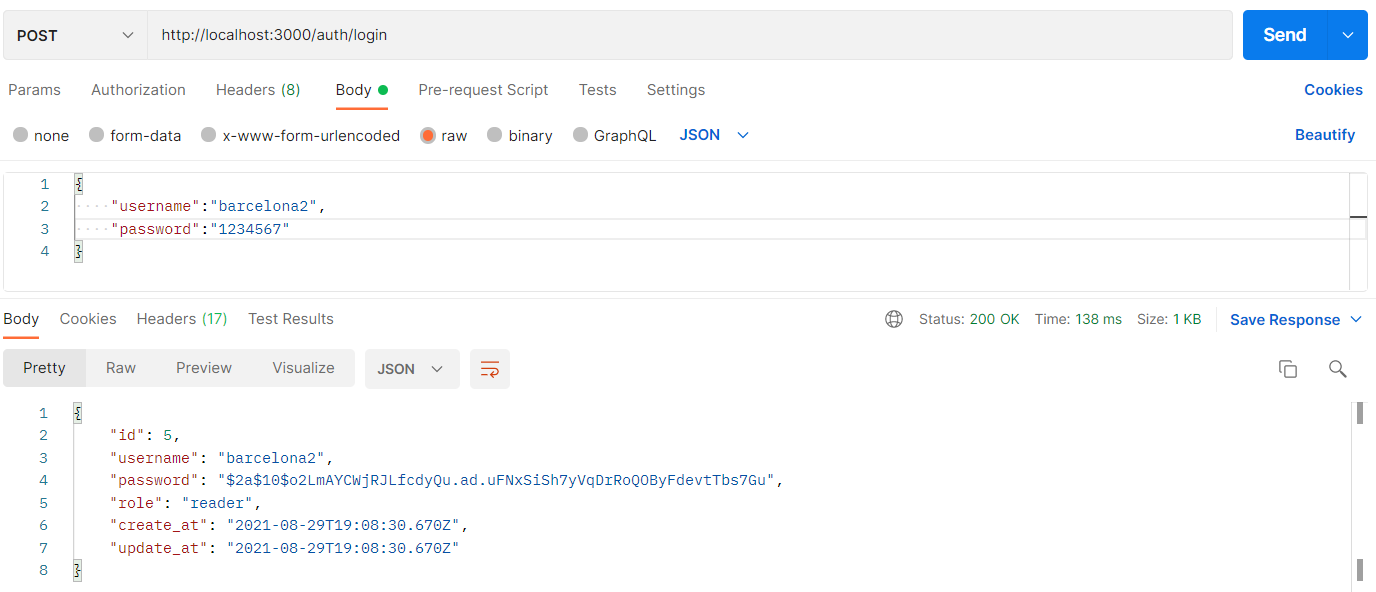
        if( !user.checkPassword(password)){

            return res.status(400).json({message:'Username or Password incorecct!'});

        }

        res.send(user);

ahora comprobamos en postman.



Vemos que nos retorna todo el objeto con el usuario, esto nos indica que ese método esta funcionando, pero no es buena práctica retornar toda esta información por seguridad, pero antes de corregir esto vamos a importar el jwt para poder generar el token.

Vamos a crear el fichero middleware y dentro de el un fichero llamado jwt.ts, ahora en este archivo vamos a hacer las importaciones.

import { Request, Response, NextFunction } from "express";

import \* as jwt from "jsonwebtoken";

import config from "../config/config";

el archive config solo tendrá un código secreto que se define puede ser cualquiera.

export default {

    jwtSecret: 'BDPEK@'

};

Ahora en elarchivojwt.ts vamos a crear el token

export const checkJwt = (req: Request, res: Response, next: NextFunction ) => {

    const token = <string>req.headers['auth'];

    let jwtPayload;

    try{

        jwtPayload = <any>jwt.verify(token, config.jwtSecret );

        res.locals.jwtPayload = jwtPayload;

    }

    catch(e){

        res.status(401).send();

    }

    const { userId, username } = jwtPayload;

    const newToken = jwt.sign({userId,username}, config.jwtSecret, {expiresIn:'1h'});

    res.setHeader('token', newToken );

    //Call next

    next();

};

Ahora debemos pedir este token en las rutas del usuario para así poder restringir que puede hacer el usuario según el rol.

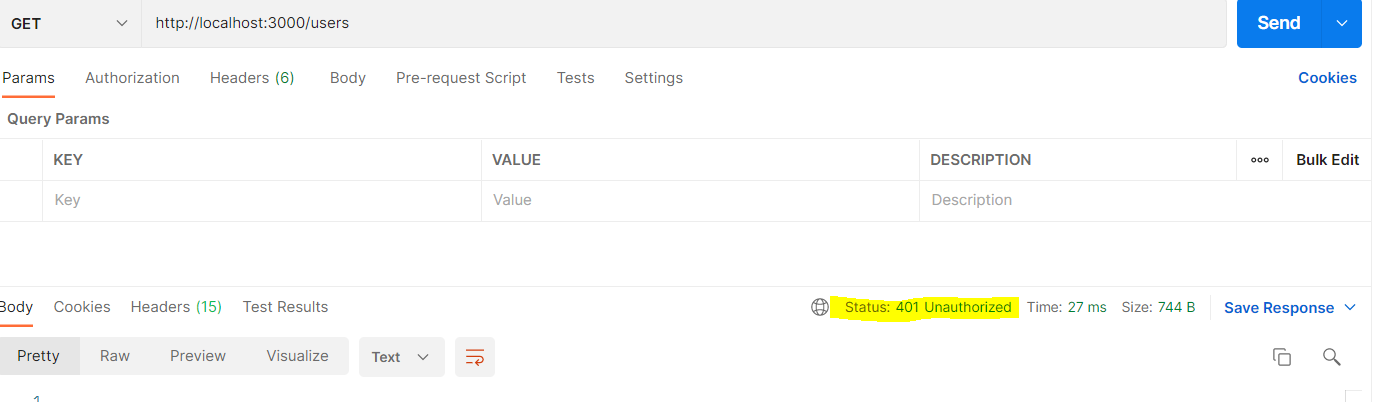
En el archivo de rutas de usuario lo primero que debemos hacer es importar nuestro middleware.

import { checkJwt } from '../middleware/jwt';

vamos a comprobar el check con el método de obtener todos los usuarios y si todo va bien ejecuta la siguiente función para eso utilizamos el next()

// Get all users

router.get('/', [checkJwt], UserController.getAll);

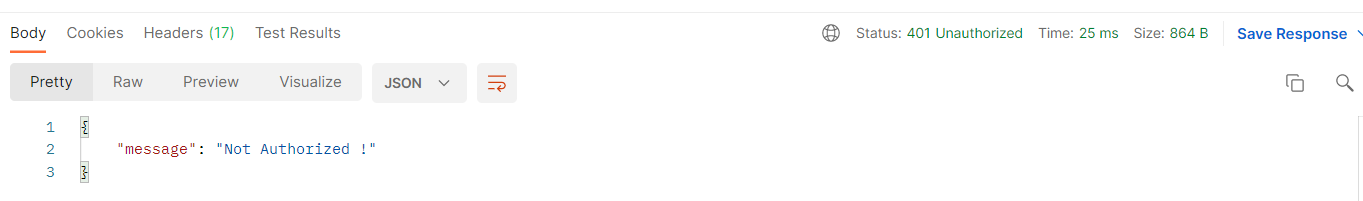


Ahora vemos que nos retorna un error 401 de no autorizado, vamos a modificar este mensaje.

catch(e){

        res.status(401).json({message:'Not Authorized !'});

    }



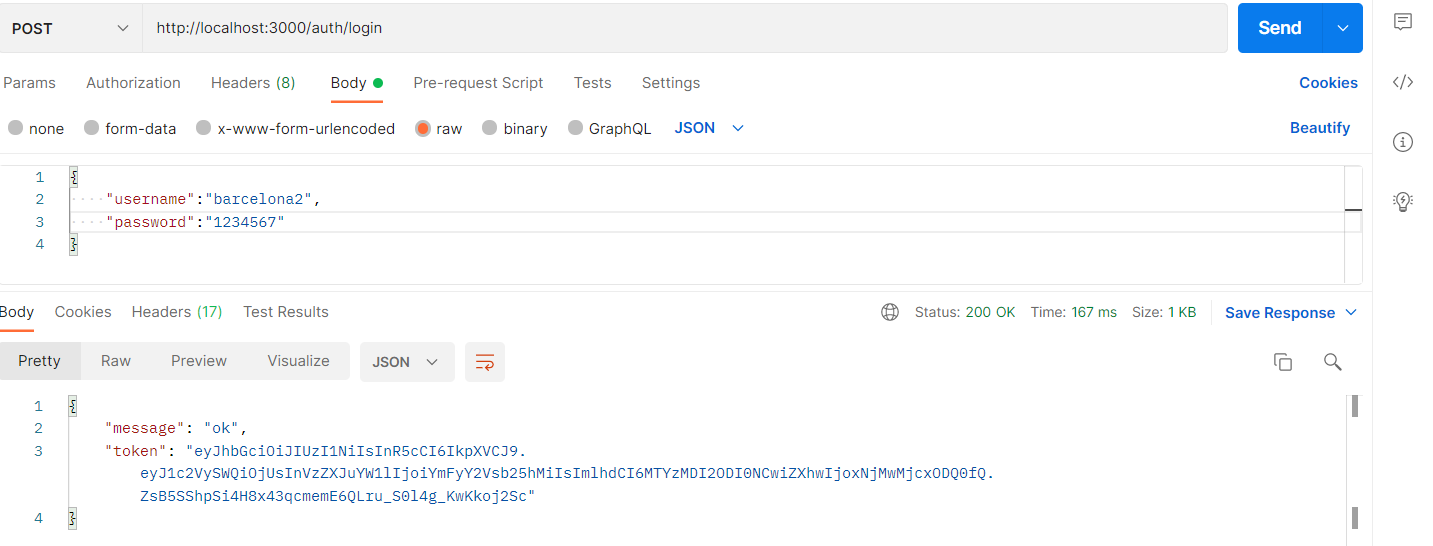
Ahora vamos al auth controller y vamos a decir que una vez la contraseña y el usuario son correctos vamos a crear un token.

const token = jwt.sign({userId: user.id, username: user.username}, config.jwtSecret, {expiresIn: '1h'});

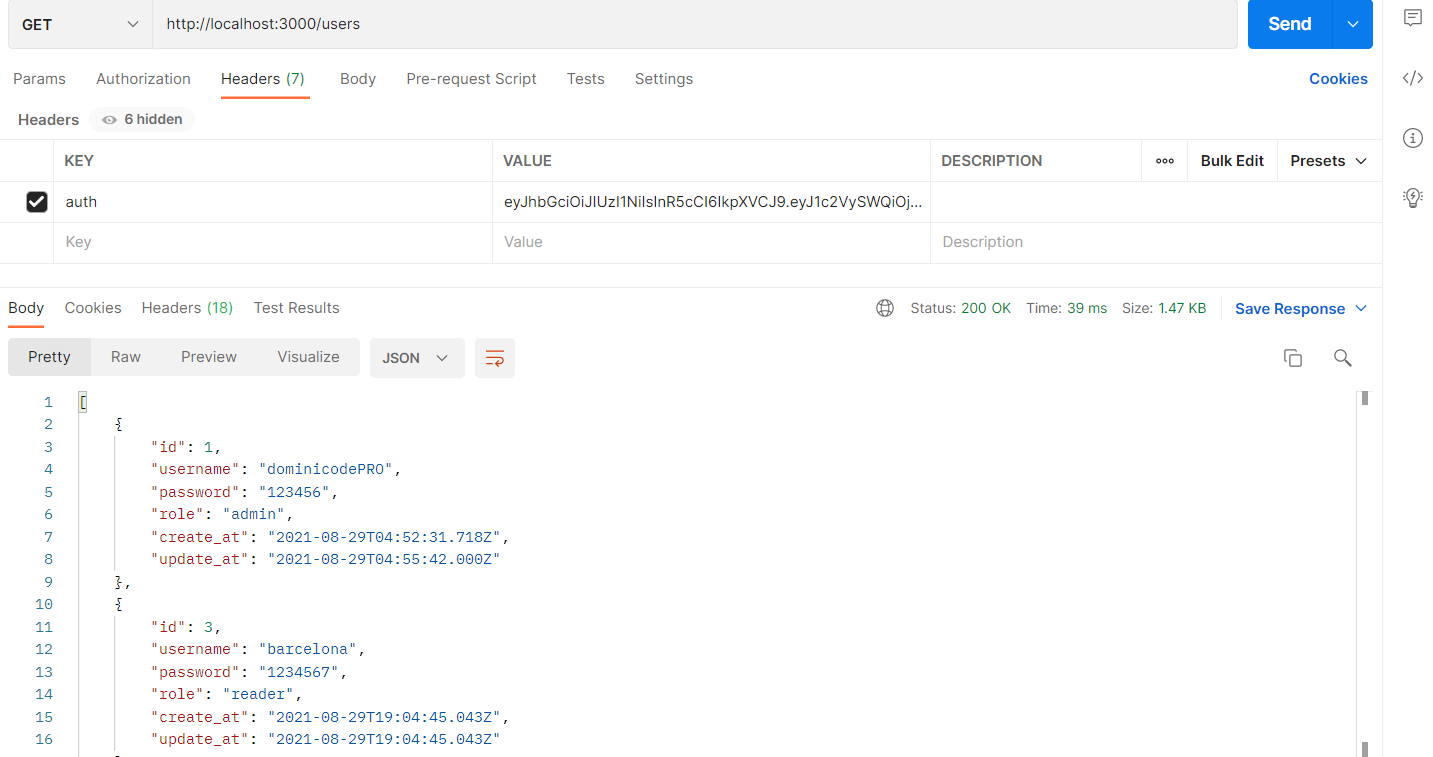
y en el send ya no pasamos todo el objeto sino solo el token.

res.json({message:'ok', token: token });

hacemos la prueba con postman.



Copiamos este token que nos genera y vamos al método de get all users y en la key auth vamos a poner este token.



Ahora ya tenemos los usuarios, lo que sigue es proteger las demás rutas.

Ahora vamos a crear la validación para que solo el usuario admin pueda hacer el crud, en los middleware vamos a crear un archivo llamad role.ts.