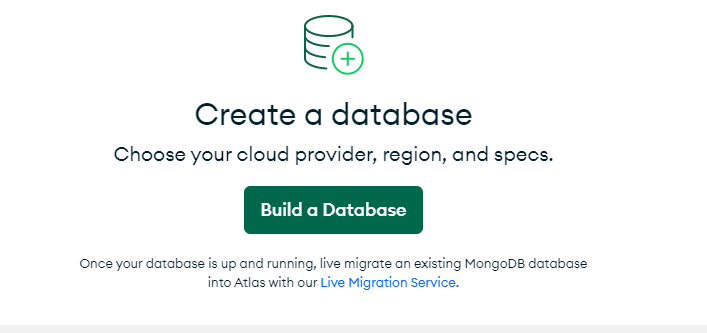
**Username emaLopez pass Fi0n4**

**Creamos y nos conectamos a la base de datos:**

En mongo db le damos a build database



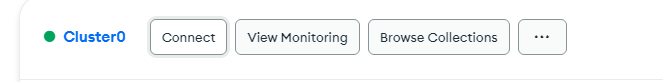
Créate el gratis

Todo por default y crear cluster hasta 500mb gratis nos da nos autenticamos por usuario

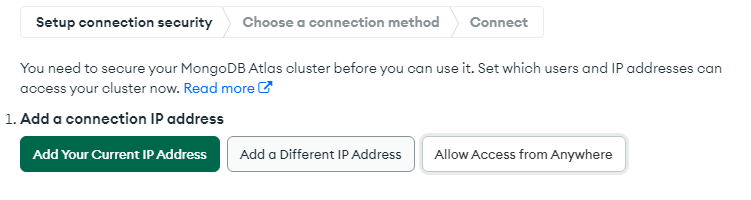
Créate user

A la izquierda en deployment database podes monitorear toda la aplicación.

Darle acá a connect



En este caso vamos a permitir el acceso de cualquier ip pero no se recomienda



Add IP

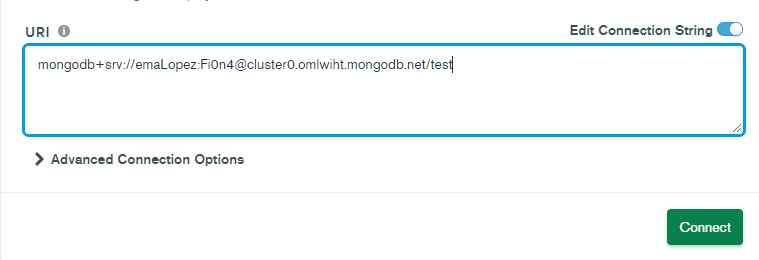
Vamos a choose a connection method

Y connect using mongo db compass

Copiamos el link y lo pegamos en compass

mongodb+srv://emaLopez:<password>@cluster0.omlwiht.mongodb.net/test

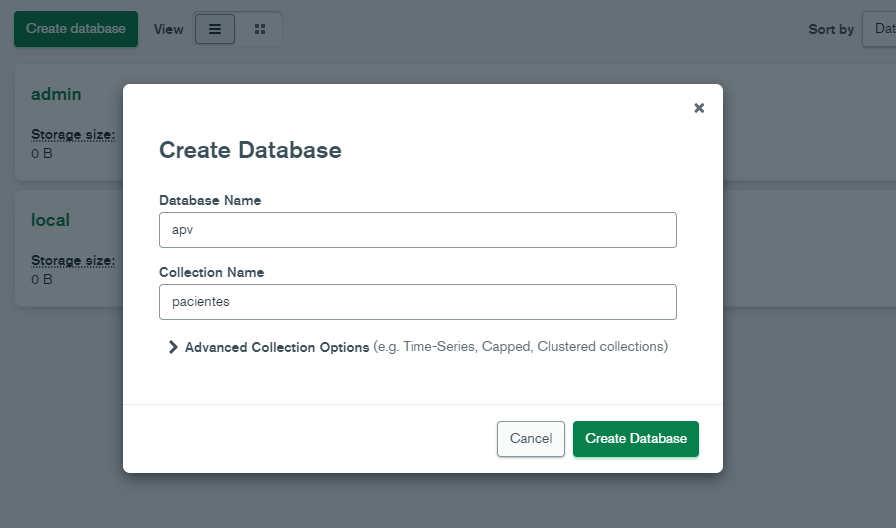
reemplazamos la contraseña



Click a connect

Vamos a databases, create database

Llenamos los datos



Nos va a aparecer la BD creada a la izquierda con la colección



**Creamos la app:**

Llamada fullstackjs y dentro vamos a tener una carpeta para backend y otra para frontend.

Dentro de la de backend hacemos

Npm init y le ponemos los datos

Dentro del backend creamos un index.js y en el package.json que se creo creamos un script llamado **start: “node index.js”**

Lo corremos entonces con **npm run start**

**Creamos un servidor:**

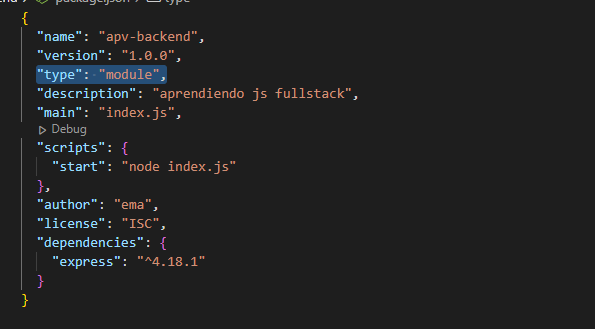
Vamos a trabajar con la dependencia express

Entonces hacemos **npm i express** siempre en la carpeta de backend

**Como hacer para que se puedan usar los imports en node?**

Vamos al package.json y agregamos

**“type”: “module”**



En el index.js ponemos

//aca configuramos el servidor

import express from "express";

const app = express();

app.use('/', (req, res) =>{

    res.send('Hola mundo'); // esto se muestra si entras a localhost:4000

})

app.listen(4000, () => {

    console.log('servidor funcionando en puerto 4000')

});

**Evitar que tener que andar frenando el npm run start y volver a cargar para ver los cambios:**

npm i --save-dev nodemon

el –save-dev se intala como dependencia de desarrollo, si se pasa a producción las dependencias de desarrollo no se suben.

Eso nos va a poner las devDependencies en el package.json

En scripts agregamos esto:

  "scripts": {

    "start": "node index.js",

    "dev": "nodemon index.js"

  },

Si hacemos npm run start va a ejecutar con node y con npm run dev va a ejecutar con nodemon

Al correr con nodemon vamos a poder hacer que el servidor se reinicie automáticamente con cada cambio.

**Conectar base de datos de mongo db con express:**

Creamos una carpeta en backend llamada config y dentro un archivo bd.js

Vamos a trabajar con la dependencia **mongoose**

Así que en backend hacemos

**Npm i mongoose**

En bd.js hacemos:

import mongoose from 'mongoose';

const conectarDB = async () => {

    try {

        const db = await mongoose.connect('')

    } catch (error) {

        console.log(error.message);

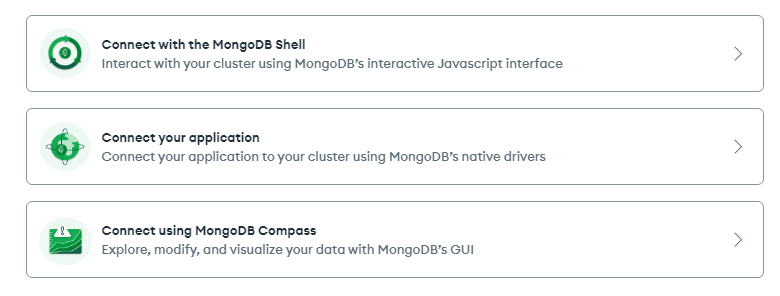
        process.exit(1); // imprime un mensaje de error

    }

}

export default conectarDB;

Vamos al browser a connect en mongodb y le damos a connect your application



Copiamos el codigo que aparece y lo pegamos en el connect del archivo bd.js, le cambiamos la password

Nos va a quedar así con otras cosas mas agregadas

import mongoose from 'mongoose';

const conectarDB = async () => {

    try {

        const db = await mongoose.connect('mongodb+srv://emaLopez:Fi0n4@cluster0.omlwiht.mongodb.net/?retryWrites=true&w=majority',{

            useNewUrlParser: true,

            useUnifiedTopology: true

        })

        const url = `${db.connection.host}:${db.connection.port}`;

        console.log(`mongo db conectado en ${url}`)

    } catch (error) {

        console.log(error.message);

        process.exit(1); // imprime un mensaje de error

    }

}

export default conectarDB;

ahora al index.js le tenemos que indicar que este db.js existe entonces lo importamos y llamamos a conectarDB()

en el import si agregas archivos que vos creaste necesitas la extensión

import express from "express";

import conectarDB from "./config/db.js";

const app = express();

conectarDB();

si corremos en npm run dev tenemos que ver en consola:



**Configurar variables de entorno, evitar que se vean los datos de la conexión a la base de datos:**

Creamos en la carpeta backend un archivo llamado .env ahí creamo la variable de entorno con lo que habíamos puesto en el db.js

MONGO\_URI=mongodb+srv://emaLopez:Fi0n4@cluster0.omlwiht.mongodb.net/?retryWrites=true&w=majority

Para leer las variables de entorno se usa process.env.nombredevariable

Para decirle a express donde encontrar las variables de entorno se hace con otra dependencia

En backend ponemos

**Npm i dotenv**

En el index.js de backend agregamos después del import de express

import express from "express";

import dotenv from 'dotenv';

y antes de conectarDB() hacemos dotenv.config()

dotenv.config();

conectarDB();

y en el archivo db.js nos va a quedar así con la variable de entorno:

        const db = await mongoose.connect(process.env.MONGO\_URI,{

            useNewUrlParser: true,

            useUnifiedTopology: true

        })

Y en el index.js ósea en el servidor también tenemos el numero de puerto 4000 que lo vamos a reemplazar por una variable de entorno, no esta definida en el .env entonces si no encuentra esa variable va a ser 4000 cuando se suba a algún host se le asigna un puerto

const PORT = process.env.PORT || 4000;

app.listen(PORT, () => {

    console.log(`servidor funcionando en puerto ${PORT}`)

});

**Tipos de datos que soporta mongoose:**



**Definir el modelo de veterinarios:**

Creamos una carpeta llamada models en backend y dentro un archivo llamado veterinario.js

Adentro va a llevar esto que va a ser la estructura que va a tener en la base de datos

import mongoose from "mongoose";

//esto crea el esquema y es la forma que van a tener los datos

const veterinarioSchema = mongoose.Schema({

    nombre: {

        type: String,

        required: true,

        trim: true,

    },

    password: {

        type: String,

        required: true,

    },

    email: {

        type: String,

        required: true,

        unique: true,

        trim: true,

    },

    telefono: {

        type: String,

        default: null, // le ponemos un default porque no es obligatorio(required)

        trim: true,

    },

    web: {

        type: String,

        default: null,

    },

    token: {

        type: String,

    },

    confirmado: {

        type: Boolean,

        default: false,

    }

})

const Veterinario = mongoose.model('Veterinario', veterinarioSchema); // esto lo registra como modelo, primer parametro nombre del modelo(creo), se recomienda usar el mismo que la variable y el segundo el nombre del esquema de arriba

export default Veterinario;

**Primeras rutas o routing:**

Creamos en backend un carpeta llamada routes y dentro file llamado veterinarioRoutes.js

Inicialmente vamos a tener algo así:

import express from "express";

const router = express.Router();

// aca adentro van a ir todas las rutas relacionadas a veterinario

export default router;

en el index.js vamos a tener que importar el veterianrioRoutes

import veterinarioRoutes from "./routes/veterinarioRoutes.js";

el veterinarioRoutes es el export default router, no tiene que llamarse igual que el otro archivo.

Y definimos que cuando vaya a la ruta api/veterinarios va a conectarse con el archivo de ruta

app.use('/api/veterinarios', veterinarioRoutes);

Ahora Podemos agregar distintas rutas a veterinarioRoutes.js

import express from "express";

const router = express.Router();

// aca adentro van a ir todas las rutas relacionadas a veterinario

router.get('/', (re, res) =>{ // esto se va a ejecutar cuando se llame a la ruta api/veterinarios

    res.send('Desde API/VETERINARIOS')

})

router.get('/login', (re, res) =>{ // esto se va a ejecutar cuando se llame a la ruta api/veterinarios/login

    res.send('Desde API/VETERINARIOS/LOGIN')

})

export default router;

El **send** servia para imprimir en pantalla

**Incorporar controladores:**

Creamos una carpeta en backend llamada controllers y dentro el archivo veterinarioController.js

Del veterinario routes vamos a sacar la parte del callback esta:

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

    res.send('Desde API/VETERINARIOS')

})

Y la pegamos en una variable dentro de veterinario Controller y lo exportamos

const registrar = (re, res) =>{

    res.send('Desde API/VETERINARIOS')

}

export {

    registrar

}

Entonces desde el veterianrio routes lo importamos

import { registrar } from '../controllers/veterinarioController.js';

y ponemos el registrar en la ruta

router.get('/', registrar) // esto se va a ejecutar cuando se llame a la ruta api/veterinarios

lo mismo hacemos con el otro pero lo dejamos como el routing del perfil

router.get('/perfil', perfil) //esto se va a ejecutar cuando se llame a la ruta api/veterinarios/perfil

**Configurar postMan para enviar peticiones:**

Requerimos respuestas de tipo json, que el json va a ser un objeto y lo vamos a convertir en respuesta json para que pueda ser interpretado en cualquier frontend.

En el veterinarioController cambiamos los send por json y dentro le pasamos un objeto con url

const registrar = (req, res) =>{

    res.json({ url: "Desde API/VETERINARIOS"})

}

const perfil = (req, res) =>{

    res.json({ url: "Desde API/VETERINARIOS/PERFIL"})

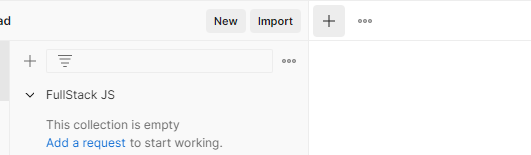
}

Abrimos postman que es una herramienta para ir probando las APIS,

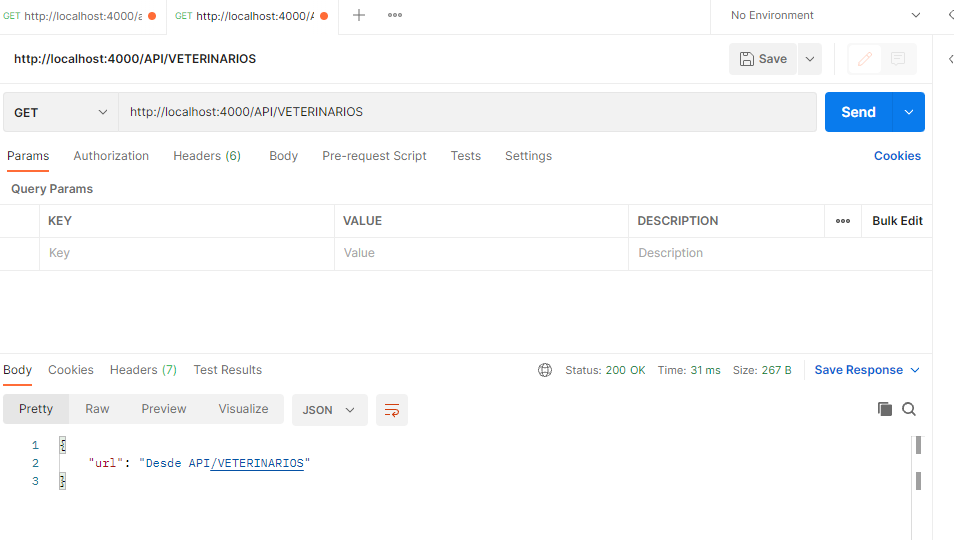
En el apartado collections le damos al + y lo llamamos FullStack JS

Entonces ahí podemos probar el registrar por ejemplo

En el signo de mas de la derecha podemos definir un nuevo request



Ponemos la URL y vamos a poder ver la respuesta dándole a send:



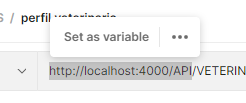
Nota: se cambio el get por post del registrar que es lógico ya que se envían datos de un formulario y el mensaje

const registrar = (req, res) =>{

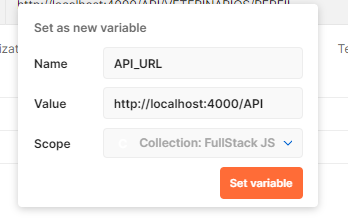
    res.json({ msg: "Registrando usuario"})

}

Seleccionamos hasta API y le damos setear como variable



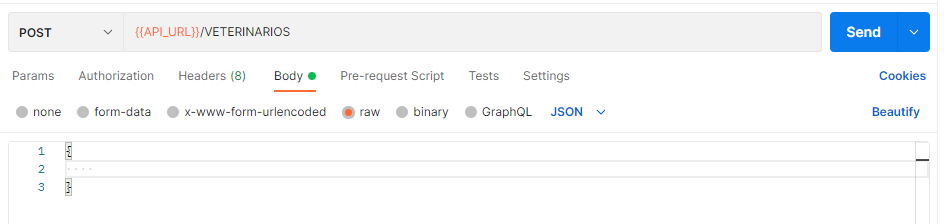
En el +



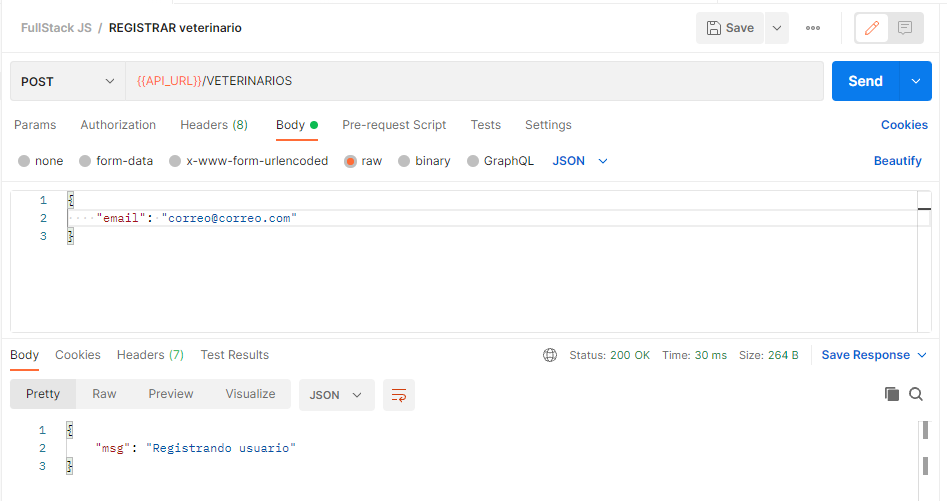
Con eso creamos una variable con esa ruta para no tener que escribirlo siempre

**Como leer datos enviados hacia la API?**

Vamos al que tiene el verbo POST, ponemos body, raw y JSON.



Mandamos datos y podemos ver la respuesta:



console.log(req.body); // esto va a traer lo que se mando en el body

pero para que esto del req.body funcione hay que poner el el index app.use(express.json())

const app = express();

app.use(express.json());

entonces cuando hacemos el send en el postam nos tiene que leer en consola el body que se envió

const registrar = (req, res) =>{

    console.log(req.body); // esto va a traer lo que se mando en el body

    res.json({ msg: "Registrando usuario"})

}

Podemos hacer destructuring para sacar cada dato que se le pase

    console.log(req.body); // esto va a traer lo que se mando en el body

    const {email, password } = req.body;

    console.log(email);

    console.log(password);

**Almacenar los registros en la BD:**

Tenemos que importar el modulo de Veterinario (donde tenemos la estructura de la BD) en el controller de veterinarios.

import Veterinario from "../models/Veterinario.js";

y nos queda así el registrar

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

    try {

        //Guardar un nuevo veterinario

        const veterinario = new Veterinario(req.body); // esto crea un nuevo objeto de veterinarios (viene del modelo la "clase")

        const veterinarioGuardado = await veterinario.save(); // para guardar en la base de datos

        res.json(veterinarioGuardado); // esto es lo que va a traer de respuesta

    } catch (error) {

        console.log(error);

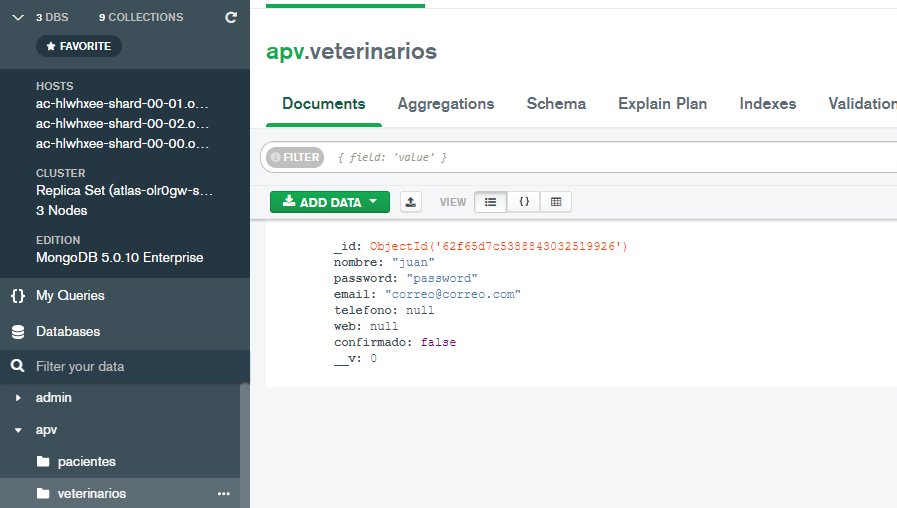
    }

}

Algo importante es ir al archivo .env y al MONGO\_URI hay que agregarle el nombre de la base de datos que creamos en este caso apv

MONGO\_URI=mongodb+srv://emaLopez:Fi0n4@cluster0.omlwiht.mongodb.net/apv?retryWrites=true&w=majority

Frenar npm run dev y volver a correr, enviar los datos por postman y ver en la base de datos si se cargó



**Prevenir usuarios registrados:**

Los usuarios no pueden tener el mismo mail, entonces hay que tener la ejecución de la API si hay un error, para que no quede dando vueltas.

El registrar del controller nos queda así:

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

    const { email } = req.body;

    //revisar si un usuario esta duplicado

    const existeUsuario = await Veterinario.findOne({email}) // podes buscar por los atributos con findOne

    if(existeUsuario){

        const error = new Error('Usuario ya registrado');

        return res.status(400).json({msg: error.message}); //salimos de la funcion con el mensaje de error

    }

    try {

        //Guardar un nuevo veterinario

        const veterinario = new Veterinario(req.body); // esto crea un nuevo objeto de veterinarios (viene del modelo la "clase")

        const veterinarioGuardado = await veterinario.save(); // para guardar en la base de datos

        res.json(veterinarioGuardado); // esto es lo que va a traer de respuesta

    } catch (error) {

        console.log(error);

    }

}

**Generar un token único:**

Borro la colección de veterinarios en la base de datos, cuando se hacen cambios en el modelo se recomienda eliminar la colección, xq cuando registras algo se queda registrado como esta en el modelo, entonces si le haces cambios nuevos no se van a modificar los registros viejos.

En el modelo en el token le puso un valor único con date.now()

    token: {

        type: String,

        default: Date.now()

    },

Pero ese no es el mejor approach, entonces, vamos a crear una carpeta llamada helpers en backend

Creamos un archivo llamado generarId.js vamos a usarlo para importar funciones y tener un código mas claro.

Dentro vamos a crear una función para que retorne un token generado random con esto:

Date.now().toString(32) + Math.random().toString(32).substring(2);

const generarId = () =>{

    //Date.now() trae la fecha con toString(32) lo convierte a numeros y letras

    //Math.random trae valor random con toString lo pasamos a caracteres y con substring(2) le sacamos los dos primeros numeros

    return Date.now().toString(32) + Math.random().toString(32).substring(2); // esta es una forma de hacer un token que sean distintos unos de otros y complejos

}

export default generarId;

importamos la función en el modulo

import generarId from "../helpers/generarId.js";

Entonces ponemos en el token así:

    token: {

        type: String,

        default: generarId(),

    },

Va a crear algo así cuando se haga la llamada



**Primeros pasos para confirmar la cuenta del usuario:**

Como por ejemplo cuando te llega el mail y verificas que sos vos, ahí el estado confirmado pasa a ser true.



En veterianrioRoutes agregamos

router.get('/confirmar', confirmar);

En el controller creamos la función y la exportamos

const confirmar = (req, res) =>{

    res.json({msg: 'Confirmando cuenta'})

}

export {

    registrar,

    perfil,

    confirmar

}

Y lo importamos en veterianrio routes

Ahora si ponemos en postman esa url un get y le damos a send vamos a tener el mensaje de confirmando cuenta

Hay que pasar el token entonces tenemos que crear un routing dinámico.

Podemos crear un parámetro dinámico poniendole dos puntos y el nombre de el parámetro después

router.get('/confirmar/:token', confirmar);

para leer eso, osea para leer datos de la url usamos req.params y después el nombre del parámetro que le pusimos en este caso token, en la función que creamos de confirmar

const confirmar = (req, res) =>{

    console.log(req.params.token);

    res.json({msg: 'Confirmando cuenta'})

}

Entonces, lo que se mande acá



Se va a imprimir acá



Por ahí se va a enviar el token.

**Encontrar el usuario a confirmar, validar y mas:**

Nos va a quedar así el confirmar

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

    const {token} = req.params;

    const usuarioConfirmar = await Veterinario.findOne({token}); // buscamos en la BD por token

    if(!usuarioConfirmar){

        const error = new Error('Token no valido');

        return res.status(404).json({msg: error.message}); // si el token que se envia por la url no encuentra el token en la DB

    }

    try {

        usuarioConfirmar.token = null;

        usuarioConfirmar.confirmado = true;

        await usuarioConfirmar.save(); // aca guardamos los cambios en la DB

        res.json({msg: 'Usuario confirmado correctamente'})

    } catch (error) {

        console.log(error);

    }

}

**Hashear passwords de los usuarios:**

Un buen lugar para hashear la contraseña es en el modelo. Vamos a instalar una dependencia, hay que revisar que la dependencia se este actualizando para que el heasheo no quede vulnerable.

Hacemos **npm i bcrypt** en backend

Importamos en el modelo

import bcrypt from "bcrypt";

Esto nos permite que antes de almacenar el registro lo vamos a hashear

Después de veterinarioSchema podemos hacer esto:

veterinarioSchema.pre('save', async function(next) { // aca no hay que usar arrow function

    //el this va a tener todos los datos del objeto que se va a guardar

    if(!this.isModified('password')){ // esto es para saber si un password ya esta hasheado que lo ignore

        next();

    }

    const salt = await bcrypt.genSalt(10);

    this.password = await bcrypt.hash(this.password, salt);

})



**Primeros pasos con el login:**

Creamos la función para autenticar a los usuarios en el controller

Creamos la funciona autenticar le ponemos en la respuesta un mensaje de autenticando, lo exportamos, lo importamos en las rutas y hacemos un post

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

**Identificando si un usuario existe para autenticarlo:**

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

    const {email} = req.body;

    //comprobar si el usuario existe

    const usuario = await Veterinario.findOne({email})

    if(!usuario){

        const error = new Error('El usuario no existe');

        return res.status(403).json({msg: error.message}); //salimos de la funcion con el mensaje de error

    }

    //Confirmar si el usuario esta confirmado

}

**Revisar si un usuario ha confirmado su cuenta:**

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

    const {email} = req.body;

    //comprobar si el usuario existe

    const usuario = await Veterinario.findOne({email})

    if(!usuario){

        const error = new Error('El usuario no existe');

        return res.status(403).json({msg: error.message}); //salimos de la funcion con el mensaje de error

    }

    //Confirmar si el usuario esta confirmado

    if(!usuario.confirmado){

        const error = new Error("Tu cuenta no ha sido confirmada")

        return res.status(403).json({msg: error.message})

    }

    //Autenticar al usuario

}

**Comprobar el password del usuario:**

Vamos a hacer la lógica en el modelo

veterinarioSchema.methods.comprobarPassword = async function(passwordFormulario){

    //retorna true o false

    // passwordFormulario es el que ingresa en el form, this.password el que esta en la base

    return await bcrypt.compare(passwordFormulario, this.password) // compare compara lo que se manda con lo de la base, aunque este hasheado

}

En el controlador ahora en la función autenticar nos va quedando así

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

    const {email, password} = req.body;

    //comprobar si el usuario existe

    const usuario = await Veterinario.findOne({email})

    if(!usuario){

        const error = new Error('El usuario no existe');

        return res.status(403).json({msg: error.message}); //salimos de la funcion con el mensaje de error

    }

    //Confirmar si el usuario esta confirmado

    if(!usuario.confirmado){

        const error = new Error("Tu cuenta no ha sido confirmada")

        return res.status(403).json({msg: error.message})

    }

    //Revisar el password

    if(await usuario.comprobarPassword(password)){

        //Autenticar

    }else{

        const error = new Error("El password es incorrecto")

        return res.status(403).json({msg: error.message})

    }

}

**Que es un JWT (Json Web token) y como generarlos:**

Una vez que un usuario se autentica se inicia una sesión en el servidor. Con esa sesios se puede verificar a que rutas el usuario tiene acceso y a cuales no

Instalamos una dependencia

Npm i jsonwebtoken

En la carpeta de Helpers vamos a hacer un archivo nuevo llamado generarJWT.js

Creamos una función

const generarJWT = ()=>{

}

export default generarJWT;

en .env vamos a crear otra variable de entorno:

JWT\_SECRET=palabrasupersecreta

La función nos va quedando así:

import jwt from 'jsonwebtoken';

const generarJWT = ()=>{

    return jwt.sign({nombre: 'Juan'}, process.env.JWT\_SECRET, {

        expiresIn: "30d", // el tiempo que tarda en expirar el jwt 30 dias

    })

}

export default generarJWT;

En el controlador importamos

import generarJWT from "../helpers/generarJWT.js";

y el autenticar nos queda así:

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

    const {email, password} = req.body;

    //comprobar si el usuario existe

    const usuario = await Veterinario.findOne({email})

    if(!usuario){

        const error = new Error('El usuario no existe');

        return res.status(403).json({msg: error.message}); //salimos de la funcion con el mensaje de error

    }

    //Confirmar si el usuario esta confirmado

    if(!usuario.confirmado){

        const error = new Error("Tu cuenta no ha sido confirmada")

        return res.status(403).json({msg: error.message})

    }

    //Revisar el password

    if(await usuario.comprobarPassword(password)){

        //Autenticar

        res.json({token: generarJWT()})

    }else{

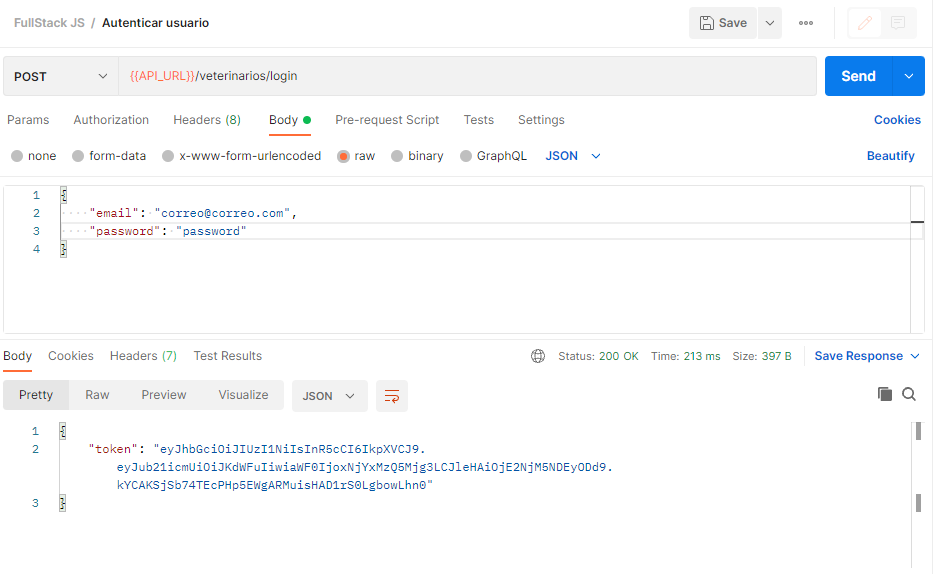
        const error = new Error("El password es incorrecto")

        return res.status(403).json({msg: error.message})

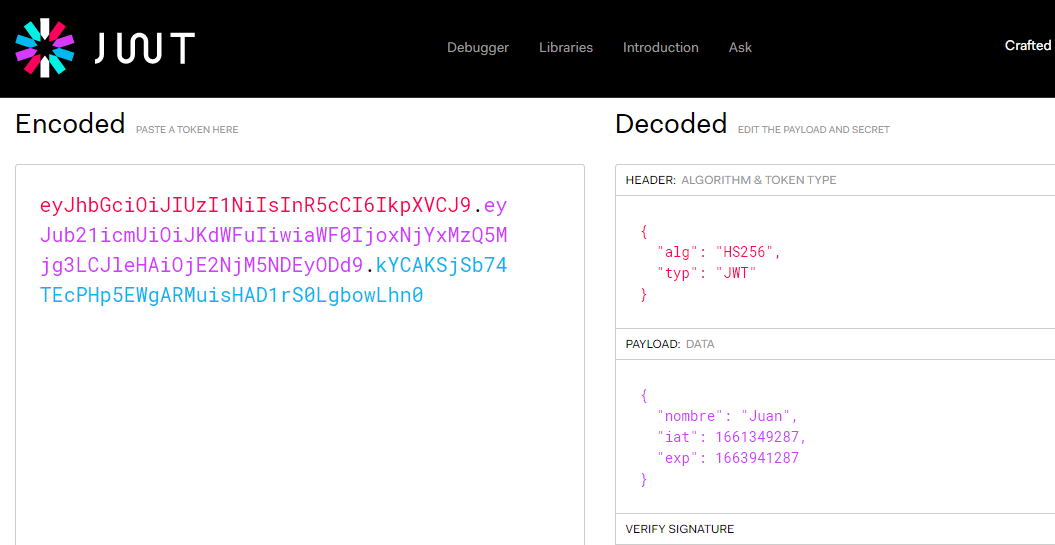
    }

}

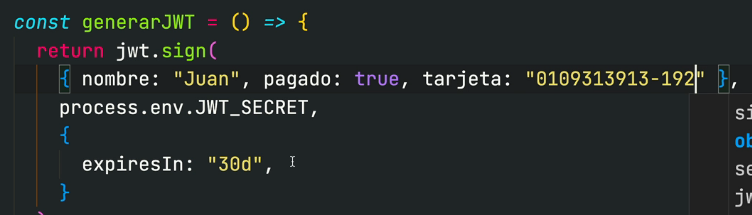
Entonces si mandamos autenticar usuario bien, nos va a devolver un token



Si ese token lo ponemos en twt la pagina vemos que manda el nombre Juan



No hay que poner info sensible acá , como pagado o números de tarjeta



Lo que vamos a guardar ahí va a ser el ID del usuario

Entonces le mandamos el id del usuario

res.json({token: generarJWT(usuario.id)})

y nos queda así la función

const generarJWT = (id)=>{

    return jwt.sign({id}, process.env.JWT\_SECRET, {

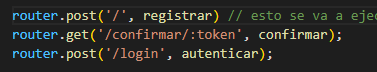
        expiresIn: "30d", // el tiempo que tarda en expirar el jwt 30 dias

    })

}

**Crear rutas protegidas con Node - Middleware:**

Para estas rutas x ejemplo no se requiere iniciar sesión para poder verlas



Para ver el perfil si hace falta que la persona este autenticada

Para proteger las paginas se crea un custom middleware

En backend entonces creamos una carpeta nueva llamada middleware y dentro un archivo authMiddleware.js

Dentro esto

const chekAuth = (req, res, next) =>{

    console.log('Desde mi middleware');

    next()

}

export default checkAuth;

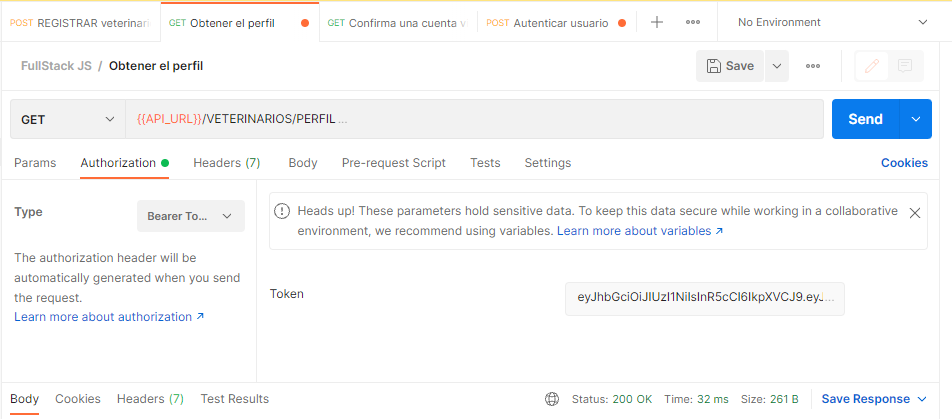
importamos en las rutas y nos queda así:

router.get('/perfil', checkAuth, perfil);

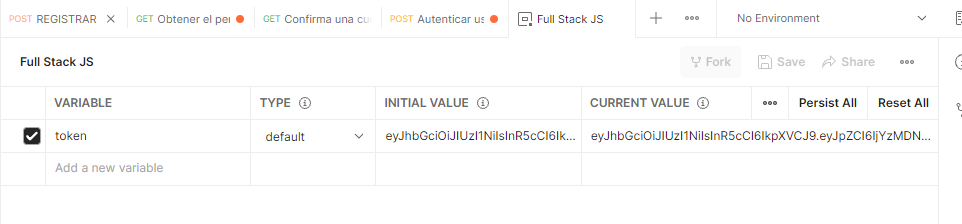
primero se va a ejecutar el checkAuth para ver si esta autenticado, cuando en esa función llega al next se ejecuta perfil.

**Comprobar el token en el middleware:**

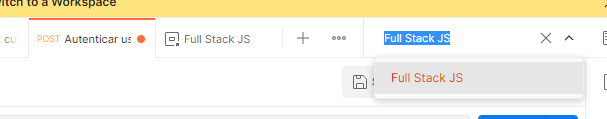
Para mandar el token vamos a obtener el perfil ponemos auth, seleccionamos bearer token y ahí pasamos el token generado



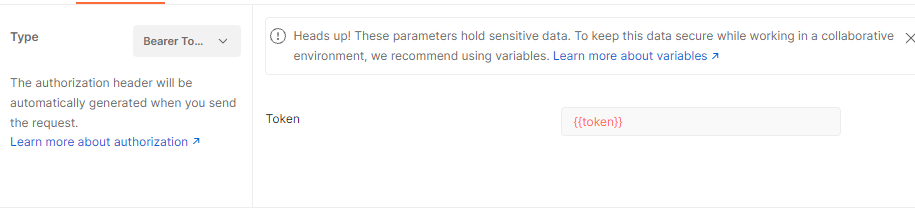
Lo recomendable es ir a enviroments crear uno nuevo, y creamos la variable token



Y después lo podés seleccionar acá arriba



Entonces acá se pone así:



Entonces así para poder obtener el token desde el checkAuth function podemos hacerlo con

req.headers.authorization

Nos queda así:

const checkAuth = (req, res, next) =>{

    if(req.headers.authorization && req.headers.authorization.startsWith('Bearer')){

        console.log('Si tiene el token con bearer')

    }

    const error = new Error("Token no valido o inexistente"); // entra aca si no tiene token valido

    res.status(403).json({msg: error.message});

    next();

}

export default checkAuth;

**Validar y autenticar al usuario con el JWT:**

import jwt from 'jsonwebtoken';

import Veterinario from '../models/Veterinario.js';

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

    let token;

    if(req.headers.authorization && req.headers.authorization.startsWith('Bearer')){

        try {

            token = req.headers.authorization.split(' ')[1]; // sacamos el bearer

            const decoded = jwt.verify(token, process.env.JWT\_SECRET);

            req.veterinario = await Veterinario.findById(decoded.id).select("-password -token -confirmado");

            //trae los datos de veterinario menos el password ni token ni confirmado

            return next();

        } catch (error) {

            const e = new Error("Token no valido");

            res.status(403).json({msg: e.message});

        }

    }

    if(!token){

        const error = new Error("Token no valido o inexistente");

        res.status(403).json({msg: error.message});

    }

    next();

}

export default checkAuth;

**Accediendo a la información del usuario autenticado:**

En el controller hacemos destructuring para sacar lo que pasamos de veterinario

const perfil = (req, res) =>{

    const { veterinario } = req;

    res.json({perfil: veterinario})

}

**Creando routing para resetear password:**

No hay forma de revertir un password hasheado, creamos un post en las rutas

router.post('olvide-password', olvidePassword);

lo creamos en el controller con el export, importamos en las rutas

Van a terminar siendo 3 rutas

router.post('/olvide-password', olvidePassword); // validar email de usuario

router.get('/olvide-password/:token', comprobarToken); // leer el token

router.post('/olvide-password/:token', nuevoPassword); // almacenar nuevo password

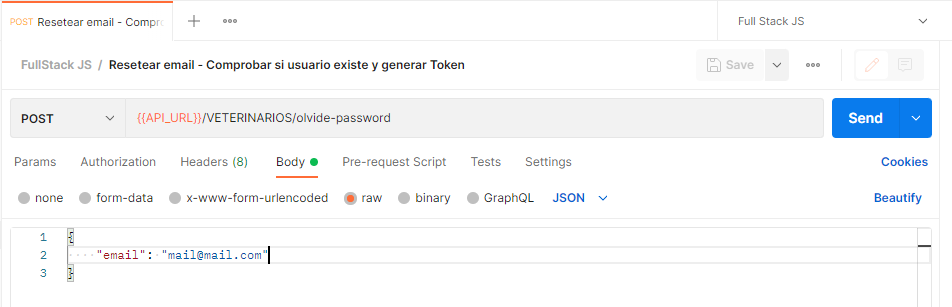
Otra forma de hacer los últimos dos es así:

router.route("/olvide-password/:token").get(comprobarToken).post(nuevoPassword);

De nuevo, creamos en el controlador e importamos en las rutas.

**Revisando si el usuario que solicita el email existe:**

Se creo nuevo post en postman para olvidePassword



Notar que pone raw json y el email

La ruta nos queda así:

router.post('olvide-password', olvidePassword); // validar email de usuario

y la function asi:

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

    const { email } = req.body;

    const existeVeterinario = await Veterinario.findOne({email});

    if(!existeVeterinario){

        const error = new Error("El ususario no existe");

        return res.status(400).json({msg: error.message})

    }

    try {

        existeVeterinario.token = generarId();

        await existeVeterinario.save();

        res.json({msg: "Hemos enviado un mail con las instrucciones"});

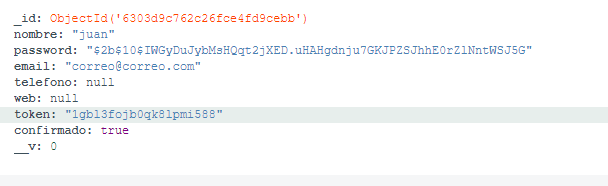
    } catch (error) {

        console.log(error)

    }

};

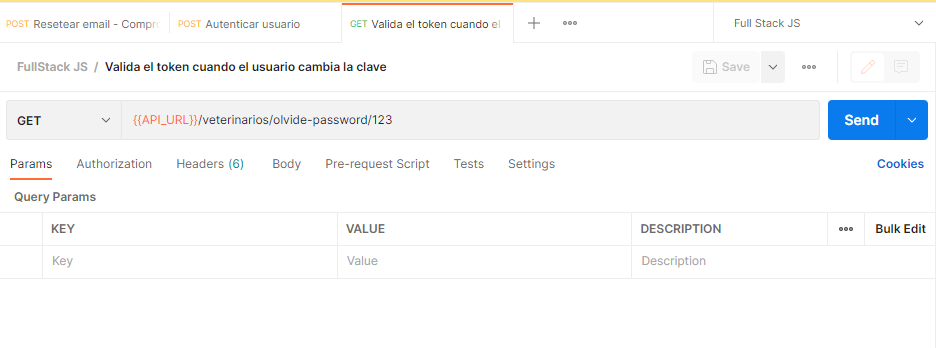
Con eso se nos va a generar este token:



Con ese token vamos a validar si el token es correcto cuando se envia el mail para verificar, y después vamos a guardar la nueva contraseña

**Comprobando si el token es valido:**

Creamos un get nuevo y al final le pasamos el token ejemplo acá 123

****

Nos queda así el comprobarToken

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

    const {token} = req.params ; // para sacar el token de la url

    const tokenValido = await Veterinario.findOne({ token }); // busca en la bd si hay un token como ese

    if(tokenValido){

        //El token es valido el usuario existe

        res.json({msg: "Token valido y el usuario existe"});

    }else{

        const error = new Error('Token no valido');

        return res.status(400).json({msg: error.message});

    }

};

**Almacenar el nuevo password y borrar token**

Vamos a borrar el token y guardar el nuevo password

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

    const {token} = req.params;

    const { password } = req.body;

    const veterinario = await Veterinario.findOne({token});

    if(!veterinario){

        const error = new Error('Hubo un error');

        return res.status(400).json({msg: error.message});

    }

    try {

        veterinario.token = null;

        veterinario.password = password;

        await veterinario.save();

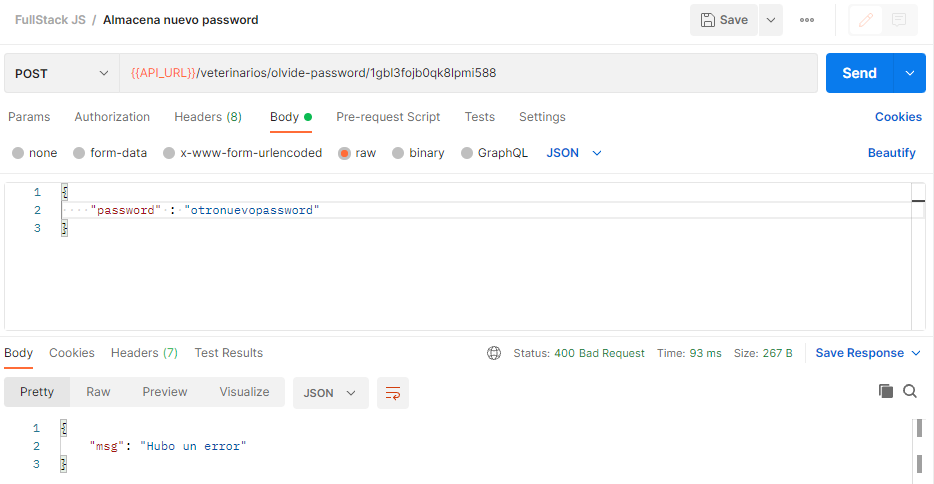
        res.json({msg: "Password modificado correctamente"});

    } catch (error) {

        console.log(error);

    }

};



**Empezamos con los pacientes!**

**Creando modelo de pacientes**

En Models creamos Paciente.js

Nos queda algo así:

import mongoose from 'mongoose';

const pacientesSchema = mongoose.Schema({

    nombre: {

        type: String,

        required: true

    },

    propietario: {

        type: String,

        required: true,

    },

    email: {

        type: String,

        required: true

    },

    fecha: {

        type: Date,

        required: true,

        default: Date.now()

    },

    sintomas: {

        type: String,

        required: true

    },

    veterinario: {

        type: mongoose.Schema.Types.ObjectId, // para sacar el id del veterinario

        ref: 'Veterinario',

    },

}, {

    timestamps: true,

})

const Paciente = mongoose.model('Paciente', pacientesSchema);

export default Paciente;

**Crear routing de pacientes:**

En routes creamos pacienteRoutes.js y en controllers creamos pacienteController.js

En pacienteRoutes hacemos

import express from "express";

const router = express.Router();

export default router;

En el index.js general importamos pacienteRoutes

import pacienteRoutes from "./routes/pacienteRoutes.js";

Hacemos el use con la ruta de pacienteRoutes

app.use('/api/veterinarios', veterinarioRoutes);

app.use('/api/pacientes', pacienteRoutes);

En el paciente routes agregamos

router.route('/') //Esto va a ser /paciente

    .post(agregarPaciente)

    .get(obtenerPacientes)

En el controller creamos las funciones y las exportamos

const agregarPaciente = (req,res) =>{

}

const obtenerPacientes = (req, res) =>{

}

export {

    agregarPaciente,

    obtenerPacientes

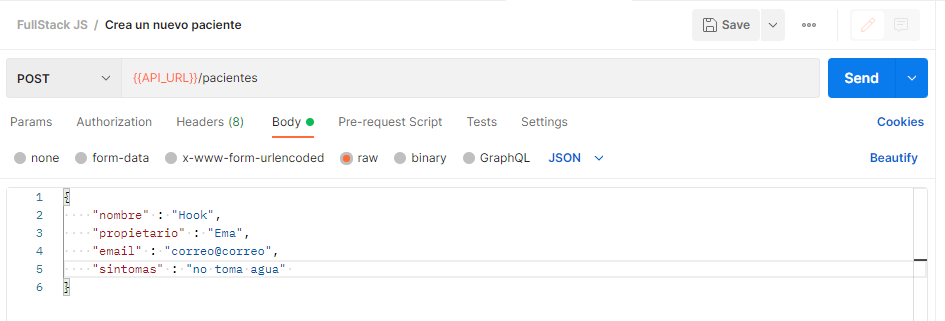
}

Las importamos en routes

import { agregarPaciente, obtenerPacientes } from "../controllers/pacienteController.js";

**Enviar petición para crear pacientes:**

Creamos una request, la fecha no se pone porque tiene el default Date.now



En el controller importamos el modelo de Paciente y vamos creando la estructura, nos falta agregar al modelo el veterinario que lo vemos después

import Paciente from "../models/Paciente.js";

const agregarPaciente = (req,res) =>{

    const paciente = new Paciente(req.body);

    try {

    } catch (error) {

        console.log(error);

    }

}

const obtenerPacientes = (req, res) =>{

}

export {

    agregarPaciente,

    obtenerPacientes

}

**Almacenar veterinario en el registro de paciente:**

Importamos en las rutas

import checkAuth from '../middleware/authMiddleware.js';

Protegemos el endpoint , va a hacer falta así autenticar al usuario y mandarle el token

router.route('/') //Esto va a ser /paciente

    .post(checkAuth,agregarPaciente)

    .get(obtenerPacientes)

Si queremos usar postman vamos a tener que usar un nuevo token sacado de autenticar usuario y pasárselo en el request en authorization,

Entonces con req.veterinario.\_id vamos a poder acceder al id de la persona que esta autenticada

Nos queda en el controller así:

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

    const paciente = new Paciente(req.body);

    paciente.veterinario = req.veterinario.\_id //paciente.veterinario es la columna de la BD

    try {

        const pacienteAlmacenado = await paciente.save();

        res.json(pacienteAlmacenado)

    } catch (error) {

        console.log(error);

    }

}

**Obtener los pacientes de Veterinario:**

Agregamos el checkAuth

router.route('/') //Esto va a ser /paciente

    .post(checkAuth,agregarPaciente)

    .get(checkAuth, obtenerPacientes)

Nos va quedando Asiel obtener todos los pacientes:

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

    const pacientes = await Paciente.find().where('veterinario').equals(req.veterinario);

    res.json(pacientes);

}

**Obtener un paciente en especifico:**

Agregamos rutas

    router.route('/:id') // /paciente/:id

        .get(checkAuth, obtenerPaciente)

        .put(checkAuth, actualizarPaciente)

        .delete(checkAuth, eliminarPaciente)

Las creamos, exportamos e importamos en las rutas

const obtenerPaciente = (req, res) =>{}

const actualizarPaciente = (req, res) => {}

const eliminarPaciente = (req, res) => {}

export {

    agregarPaciente,

    obtenerPacientes,

    obtenerPaciente,

    actualizarPaciente,

    eliminarPaciente

}

El obtener paciente:

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

    const { id } = req.params;

    const paciente = await Paciente.findById(id);

    if(!paciente){

        return res.status(400).json({msg: "no encontrado"})

     }

    if(paciente.veterinario.\_id.toString() !== req.veterinario.\_id.toString() ){

        return res.json({msg: 'Accion no valida'});

    }

    res.json(paciente);

}

**Actualizar pacientes:**

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

    const { id } = req.params;

    const paciente = await Paciente.findById(id);

    if(!paciente){

       return res.status(400).json({msg: "no encontrado"})

    }

    if(paciente.veterinario.\_id.toString() !== req.veterinario.\_id.toString() ){

        return res.json({msg: 'Accion no valida'});

    }

    //actualizar paciente

    // se guarda lo que se manda en el body si no se mandaese dato se usa el que ya tiene

    paciente.nombre = req.body.nombre || paciente.nombre;

    paciente.propietario = req.body.propietario || paciente.propietario;

    paciente.email = req.body.email || paciente.email;

    paciente.fecha = req.body.fecha || paciente.fecha;

    paciente.sintomas = req.body.sintomas || paciente.sintomas;

    try {

        const pacienteActualizado = await paciente.save();

        res.json(pacienteActualizado);

    } catch (error) {

        console.log(error);

    }

}

**Eliminar pacientes:**

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

    const { id } = req.params;

    const paciente = await Paciente.findById(id);

    if(!paciente){

       return res.status(400).json({msg: "no encontrado"})

    }

    if(paciente.veterinario.\_id.toString() !== req.veterinario.\_id.toString() ){

        return res.json({msg: 'Accion no valida'});

    }

    try {

        await paciente.deleteOne();

        res.json({msg: "paciente eliminado"});

    } catch (error) {

        console.log(error);

    }

}

**FRONTEND CON REACT**

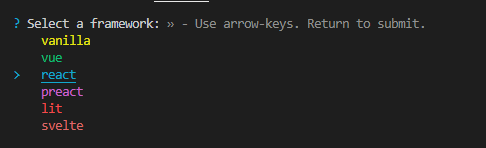
**CREAR LA APP:**

Corremos en la carpeta raíz:

Npm init vite@latest

Ponemos nombre de la carpeta frontend

Noda da las opciones de varios frameworks vite es de vue , pero vamos a usar react



Vamos a la carpeta frontend y después hacemos npm install para instalar las dependencias

Corremos con npm run dev en la carpeta frontend

Vamos a trabajar dentro de la carpeta src en donde tenemos

Eliminamos el contenido del css

Cambiamos el title del index.html

Borramos los css el contenido, y nos queda el app.jsx así: solo con esto

function App() {

  return (

    <h1>Hola Vite</h1>

  )

}

export default App

instalamos tailwind, parecido a Bootstrap, con Bootstrap compras la mesa, con tailwind compraslas partes para armar la mesa.

npm i --save-dev tailwindcss postcss autoprefixer

npx tailwindcss init -p

este ultimo crea un postcss.config y un tailwind.config

en el tailwind configuramos esta línea

  content: ["index.html", "./src/\*\*/\*.jsx"],

en el index.css ponemos

@tailwind base;

@tailwind components;

@tailwind utilities;

**Instalar react router DOM 6:**

npm i react-router-dom

En el app.jsx vamos a llamar las rutas y decir que componente se va a mostrar:

Acá en la ruta principal se va a mostrar el componente AuthLayout

Para vincular front con el back:

Npm i axios

Nodemailer: permite enviar emails usando nodejs

Mailtrap: permite probar el envio de emails

Se crea un inbox y le das a inboxes entras y seleccionas nodemailer

Npm i nodemailer en el backend

No lo pude seguir.