Proyecto Fullstack:  
Se utiliza mongodb como base de datos y estaremos utilizando el orm de moongose

Utilizamos postman

Utilizamos compas

Primeros pasos:  
Creamos una carpeta para el backend y el frontend.

Despues entramos a nuestra carpeta de backend desde la terminal y creamos un package.json con el comando:  
*npm init*

Despues creamos en nuestra carpeta de backend un index.js.

Para poder comprobar que si este funcionando el index.js, podemos crear en package.json en la parte de scripts un start y correr index.js:

{

  "name": "apv-backend",

  "version": "1.0.0",

  "description": "Aprendiendo js fullstack",

  "main": "index.js",

  "scripts": {

    "start": "node index.js"

  },

  "author": "Ian Martinez",

  "license": "ISC"

}

Y ya para comprobar, escribes un console log en index.js y ejecutas el comando:

*npm run start ó npm start*

y te deberá de mostrar en consola el texto que hayas escrito.

Después instalamos la dependencia de express:  
*npm i express*

Después para agregamos los imports y exports de node.js en la parte de package.json, y colocar “type”:”module”, para indicarle que serán por medios de modulos:

{

  "name": "apv-backend",

  "version": "1.0.0",

  "type":"module",

  "description": "Aprendiendo js fullstack",

  "main": "index.js",

  "scripts": {

    "start": "node index.js"

  },

  "author": "Ian Martinez",

  "license": "ISC",

  "dependencies": {

    "express": "^4.21.2"

  }

}

Y ya de esa forma ya puedes agregar express a tu index.js con la sintaxis de import.

Despues creamos una variable para obtener la funcionalidad de express y asi poder crear nuestro servidor, en donde la pasamos el puerto en donde se ejecutara y después un arrowFunction en donde si se ejecuta correctamente se mostrara un mensaje:

const app = express(); //En esta variable de app vamos a tener toda la funcionalidad que requeriamos para crear el servirdor.

app.listen(4000, ()=>{

    console.log('Servidor funcionando en el puerto 4000')

});

Despues para manejar el routing utilizamos el ‘app.use’ que es la forma en la que express lo maneja (request es lo que tu estas enviando, response es lo que tu obtienes del servidor): (recuerda reiniciar el servidor)

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

    res.send('Hola mundo');

})

Para no estar reiniciando el servidor vamos a instalar una dependencia de desarrollo:

*npm i -D nodemon*

Después agregamos a package.json nodemon en la parte de scripts para que lo corra con nodemon en lugar de con node:

{

  "name": "apv-backend",

  "version": "1.0.0",

  "type": "module",

  "description": "Aprendiendo js fullstack",

  "main": "index.js",

  "scripts": {

    "start": "node index.js",

    "dev":"nodemon index.js"

  },

  "author": "Ian Martinez",

  "license": "ISC",

  "dependencies": {

    "express": "^4.21.2"

  },

  "devDependencies": {

    "nodemon": "^3.1.9"

  }

}

De esta forma en lugar de darle npm start, le daremos *npm run dev* para que de esa forma corra el servidor con nodemon.

Despues pasamos a la conexión de nuestra base de datos:  
Vamos a crear una carpeta llamada ‘config’ en donde ahí introduciremos nuestro archivo: db.js.

Para conectarnos a la base de datos vamos a utilizar un orm que se llama ‘mongoose’:

npm i mongoose

Despues en nuestro archivo db.js agregamos la configuración para poder conectar a nuestra base de datos:  
Primero importamos mongoose.

Despues en definimos una variable en donde hacemos la conexión con un async await y un try catch, en donde adentro del try agregamos una variable llamada db en donde colocamos el await y de adentro mongoose nos proporciona un método llamado ‘connect’ en donde adentro del paréntesis agregamos el string de conexión a nuestra base de datos de mongoAtlas. Despues agregamos una variable llamada url. En donde le asignamos: ${db.connection.host} que nos va a dar una url y ${db.connection.port} que nos va a dar el puerto.

import mongoose from "mongoose";

const conectarDB = async () => {

    try {

        const db = await mongoose.connect("mongodb+srv://root:root@cluster0.l4t1c.mongodb.net/");

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

        console.log(`MongoDB conectado en: ${url}`);

    } catch (error) {

        console.log(`Error: ${error.message}`);

        process.exit(1); // Detener la aplicación en caso de error

    }

};

export default conectarDB;

Despues en index.js le indicamos que el anterior archivo (db.js) existe:

Que esto se hace importando el archivo conectarDB y utilizándolo en index:

import express from "express";

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

const app = express(); //En esta variable de app vamos a tener toda la funcionalidad que requeriamos para crear el servirdor.

conectarDB();

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

    res.send('Hola mundos');

})

app.listen(4000, ()=>{

    console.log('Servidor funcionando en el puerto 4000')

});

Creación de variables de entorno:  
Tenemos que indicarle a express donde encontrar las variables de entorno con una dependencia:

*npm i dotenv*

después en el index.js agregamos esta dependencia:

import dontenv from 'dotenv'

Y justo después de la variable app colocamos:

dontenv.config();

Despues en el index.js agregamos la variable PORT, ya que cuando hagamos el deployment en un servidor especializado de js, esta variable se asigna automáticamente:

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

Despues se crean los modelos, en donde se recomienda que los nombres de estos empiecen con mayuscula.

Despues en el archivo se importa mongoose y se crea el schema que es la estructura que va a tener los datos en el modelo de veterinario:

import mongoose, { mongo } from "mongoose";

const veterinarioSchema = mongoose.Schema({

})

**IMPORTANTE:** Conforme se vayan agregando los registros, mongodb automáticamente le agrega el id entonces no requieres asignar el id en el modelo.

Datos que soporta mongoose:  


Esta es una forma de indicarle al schema que tipo de datos serán y sus características:

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,

        trim:true

    },

    web:{

        type: String,

        default:null,

    },

    token:{

        type: String,

    },

    confirmado:{

        type: Boolean,

        default:false

    },

});

Despues tenemos que registrarlo como modelo:

const Veterinario = mongoose.model("Veterinario",veterinarioSchema);

En donde definimos una variable, después utilizamos el método model, en donde dentro de los paréntesis le indicamos que el modelo se va a llamar “Veterinario” y el segundo argumento es la forma que van a tener los datos. Y ya por utlimo lo exportamos :

export default Veterinario;

Después creamos una carpeta en donde vamos a ir metiendo las rutas:

En donde los archivos que introduzcamos adentro serán los administraran las rutas (por ejemplo “veterinariosRoutes.js”). la sintaxis es la siguiente:

import express from 'express';

const router = express.Router();

export default router;

Despues importamos este anterior archivo en index.js:

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

Y declaramos que una url para que cuando la visitemos mandamos a llamar el routing que tengamos en el archivo:

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

Y en el archivo “veterinarioRoutes” vamos a colocar las rutas que van a seguir la sintaxis antiror:

router.get('/', (req,res) =>{ //que esto es como si pusiéramos: “/api/veteriarinarios/”

    res.send('Hola mundo');

})

Despues creamos los controllers:

En donde vamos a crear una carpeta que se llame “controllers”, después de esto vamos a crear un archivo llamado “veterinarioControllers.js”. Despues nos movemos a “verinarioRoutes.js”.

Y en lugar de que en “verinarioRoutes.js” tengamos:

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

    res.send('desde /api/veterinarios/');

});

Vamos a tener:

router.get('/', registrar);

Y en “veterianrioControllers.js” vamos a tener lo siguiente:

const registrar = (req,res) =>{

    res.send('desde /api/veterinarios/');

}

export {registrar};

De esta forma se comunican las rutas con los controladores.

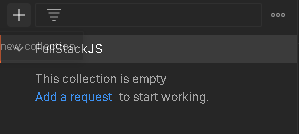
Send envia datos en forma de texto a la pantalla, pero lo que queremos es que los envie como json, para eso utilizamos:

const perfil = (req,res) =>{

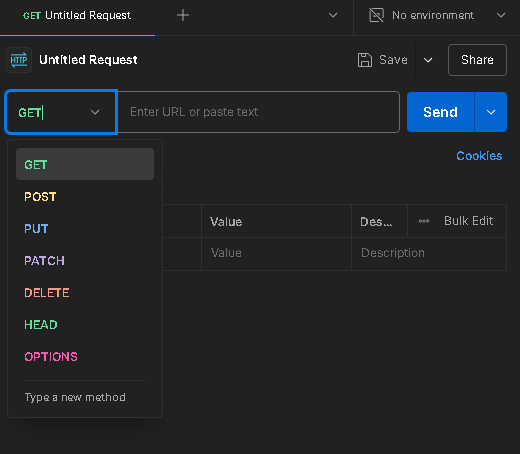
    res.json({url: "desde api/veterinarios/perfil"});

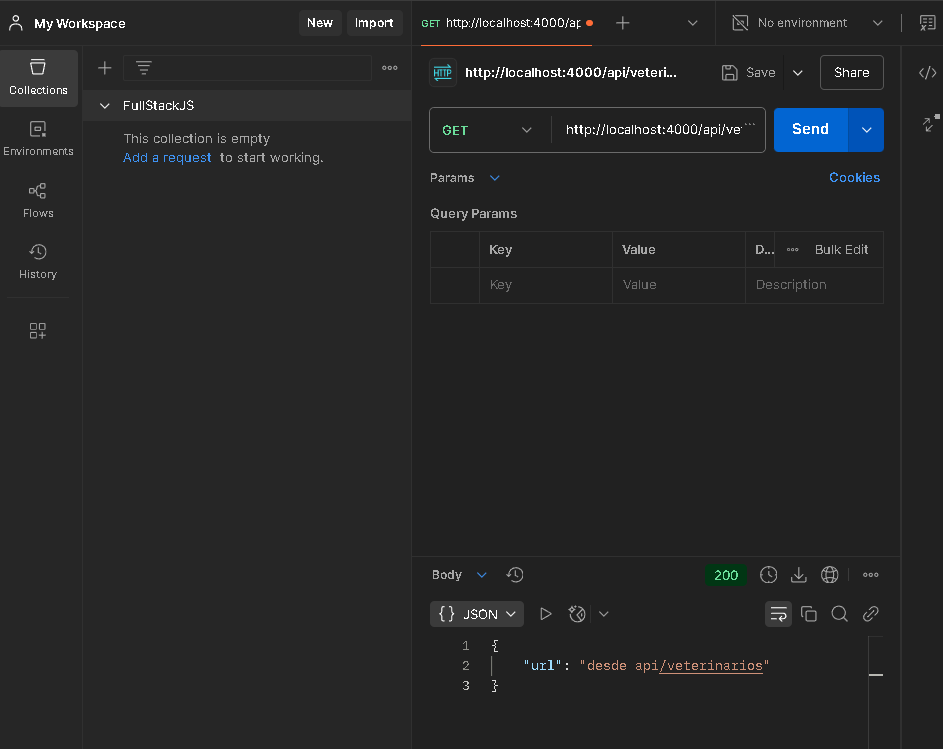
}

Despues de hacer eso utilizamos la herramienta “Postman” para poder hacer las pruebas de envio y recepción de datos, primero creamos una nueva colección en el botón de ‘+’ y le asignamos un nombre:

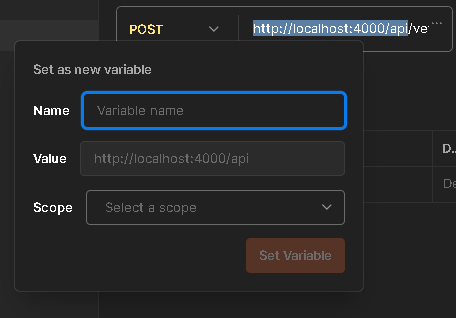


Despues en el signo de mas en la pestaña que sale al seleccionar ‘fullstackjs’ podemos crear diferentes request con los diferentes verbos para enviar y recibir peticiones:



Despues introducimos la url y le damos a “send”, lo que nos va a dar una respuesta de tipo json como definimos anteriormente en VS:  


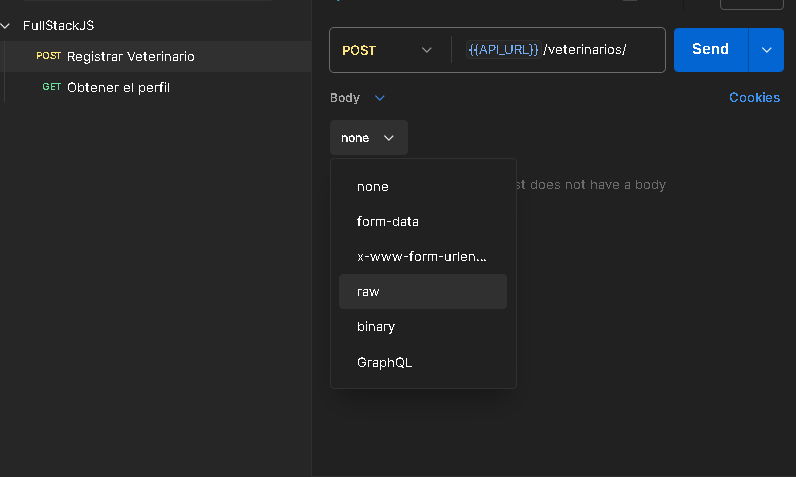
Para almacenas los request podemos precionarle a un lado de la url y darle “save” y asi darle un nombre.

Despues también si hay una url que se parezca hasta cierto punto y que otros request compartan, lo podemos settear como variable:  


En scope la guardamos como un collection y le damos un nombre:  


Como enviar datos con postman?

Das clic en ‘body’ y después en ‘raw’ y lo colocas por tipo ‘json’



Cuando tu envias información hacia una api de express, eso se guarda en ‘request.body’

Pero te va a marcar ‘undefined’, esto pasa porque le tienes que indicar a express que vas a enviar datos de .json, y eso lo logras en index.js colocando después de definir app la siguiente línea:

app.use(express.json())

Almacenando registros en la BD

Primero creamos una nueva instancia, por lo que la creamos con lo “req.body” ya que ahí viene la información que le estamos pasando:

const veterinario = new Veterinario(req.body);

Despues lo guardamos con la siguiente sintaxis:

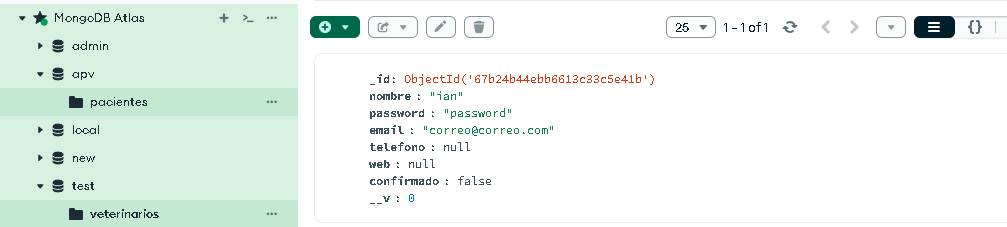
const veterinarioGuardado = await veterinario.save();

Nota: .save() es uno del los métodos de mongoose para crear un nuevo registro, es útil si tienes un objeto y o vas a guardar en la base de datos o si tienes un objeto, lo modificas y después lo guardas.

IMPORTANTE - RECUERDA:  
Si dejas tu conexión a la base de datos en tu .env de esta forma:

MONGO\_URI=mongodb+srv://root:root@cluster0.l4t1c.mongodb.net/

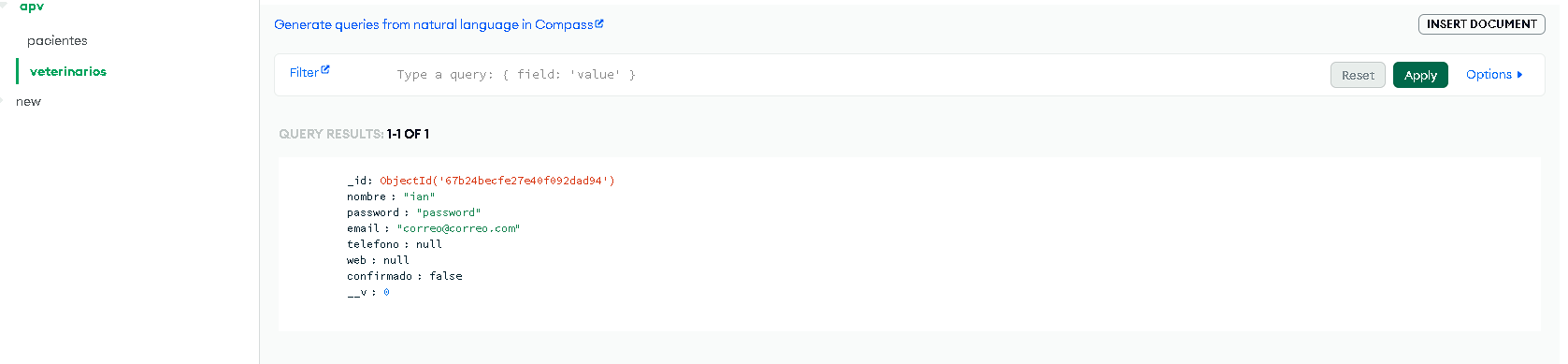
Te creara una nueva base de datos. que en este caso se llamo como ‘test’:



Para evitar esto y meterla a la base da datos “apv” lo que tienes que hacer es al final del texto de tu variable de entorno, después del ‘/’ agregarle el nombre de la base de datos en donde quieres que se guarde el modelo (recuerda que para borrar la anterior bd que se creo necesitas acceder al cluster desde la pagina web de mongo):

MONGO\_URI=mongodb+srv://root:root@cluster0.l4t1c.mongodb.net/apv

De esta forma se guardara en ‘apv’:



Con el método “.find( )” te puedes traer todos los registros para buscar entre ellos, en cambio con “.findOne( )” te permite buscar por los diferentes atributos que tenga cada uno de los registros de la base de datos, y “.findById( )” se trae un registro por su id.

Ejemplo de findOne:

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

Nota: Cuando vayas a modificar una colección, es necesario que elimines la colección anterior, esto porque ya que se registra un modelo se queda almacenado todo lo que se tiene.

Creacion de helpers:  
Estas son algunas funciones que puedo importar en diferentes archivos, esto sirve para no cargar demasiado los archivos. De esta forma separas las funciones, se recomienda 1 helper por archivo, ya que por la forma en la que funcionan los imports, si importas 1 con 100 helpers, asi solamente uses 1 funcion va a escanear todo el archivo.

Creando un routing dinamico:  
Para eso express te permite agregarle después de definir la ruta un “:”:

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

Despues le puedes agregar el nombre que quieras:

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

Nota: Siempre que tu leas datos de la URL, lo vas a poder leer con: req.params

Y ya simplemente para leerlo colocas “.” Y el nombre que le diste:

console.log(req.params.token);

Hashear los passwords:

Los passwords se hashean en el modelo.

Para hashear los passwords se utiliza una dependencia (tienes que estas al pendiente de estas dependencias, ya que si estas no se actualizan y les dejan de dar soporte, pueden ser vulneradas). La dependencia se llama ‘bcrypt’ y la instalamos en las dependencias normales. Bcrypt permite hashear los passwords pero también comprobar un password contra lo que este hasheado en la base de datos.

En mongoose tenemos acceso a ciertos hooks, y hay acciones que se hacen antes o después de que se guarde un registro y ahí puedes insertar código para que haga ciertas acciones.

Nota: el hasheo se hace antes de almacenar el registro.

Por ejemplo aquí usamos un código que se obtiene de la documentación de mongoose:

veterinarioSchema.pre('save',function(){

    console.log("antes de almacenar");

})

Que lo que nos esta indicando es que como es ‘pre’ es una funcion que se hace antes de instertar el objeto en la base de datos. Y en este caso como es ‘.pre’ y dentro del paréntesis viene la palabra ‘save’ es previo al guardado.

Como saber si algo ya fue modificado, en este caso el password:

veterinarioSchema.pre('save', async function(next){

    if(!this.isModified("password")){ //Esto sirve para que si despues hay un cambio y password ya esta hasheado no lo vuelva a hashear

        next(); //Next sirve para que pase al siguiente middleware

    }

Autenticar el usuario:  
Para eso se necesita leer lo que el usuario este mandando, y eso se obtiene con request.body:

Console.log(req.body)

Para simular el envio y ver que todo este bien, puede ir a postman y simular un envio con esa dirección.

Autenticar usuario:

Para eso podemos ocupar :

veterinarioSchema.methods

Que sirve para que las funciones se ejecuten únicamente en este schema o en este

modelo.

Y después de eso lo puedes nombrar como tu quieras, en este caso será nombrado asi:

veterinarioSchema.methods.comprobarPassword

Despues para compararlo, lo vamos a hacer con un método de bcrypt, que se llama

“compare”, y compare toma 2 valores, el primero es el password del formulario y el segundo es el password que el usuario guardo (se puede comparar, aunque el password este hasheado).

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

    return await bcrypt.compare(passwordFormulario, this.password)

}

Ahora para verificar si el password es correcto o no lo podemos hacer con un if

if(await usuario.comprobarPassword(password)){

        //usamos await porque tiene que buscar en la base de datos

        console.log("password correcto");

    }else{

        const error = new Error('El password es incorrecto');

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

    }

Podemos usar usuario.comprobarPassword porque arriba tenemos una instancia, y que era una instancia?, era lo siguiente:

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

(en si es porque usamos ‘Veterinario” ).

Instalacion de Jsonwebtoken:

Se usa

npm i jsonwebtoken

Esto sirve para generar los json web tokens pero también comprobar que sean correctos.

Despues nos vamos a las variables de entorno y creamos un JWT\_SECRET:

JWT\_SECRET=palabrasecreta

Después generamos un archivo dentro de los helpers que se va a llamar “generarJWT.js”:

En donde importamos jwt y le pasamos la sintaxis para poder generar uno:

import jwt from 'jsonwebtoken';

const generarJWT = () =>{

    return jwt.sign({nombre: 'Juan'}, process.env.JWT\_SECRET,{ expiresIn:'30d',});

}

export default generarJWT;

En donde en el primer objeto tu puedes colocar lo que tu quieras, en este caso de ejemplo se coloco el nombre, pero puedes colocar el id, entre otras cosas (se recomienda no dar información sensible).

Creacion de un middleware para ver si el paciente esta autenticado:  
Primero creamos una carpeta que se llamara Middleware, para después creemos la función que vamos a colocar en el archivo, esta función la a vamos a utilizar para poder autenticar al usuario.

Este seria un ejemplo de middleware:

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

    console.log("desde mi middleware");

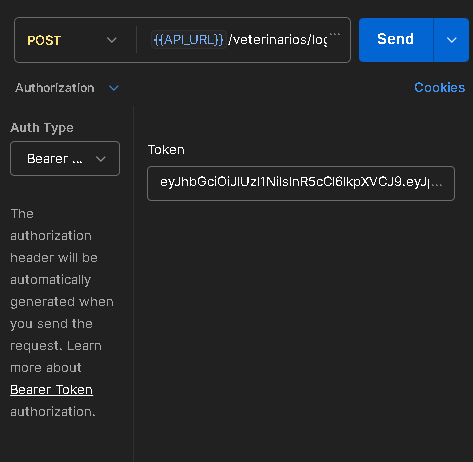
    next();

};

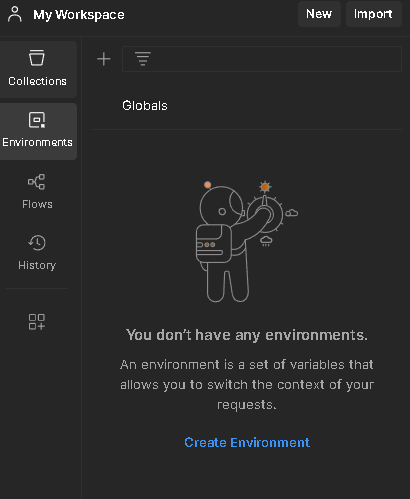
En donde el next(); sirve para saltar al siguiente middleware (que no se sabe cual es, pero que sirve para que no se quede únicamente ahí).

Comprobando el Token en el Middlewar:

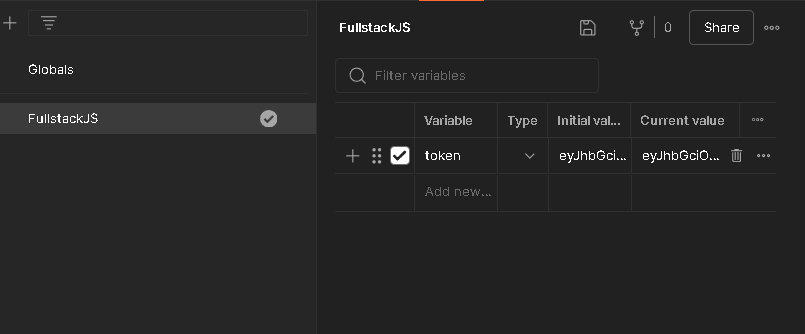
En donde se crean envía el Json web token, este se envia normalmente en header, pero en esta caso vamos a utilizar “bearer token” en la parte de “autorizacion”, en donde vamos a colocar el token.

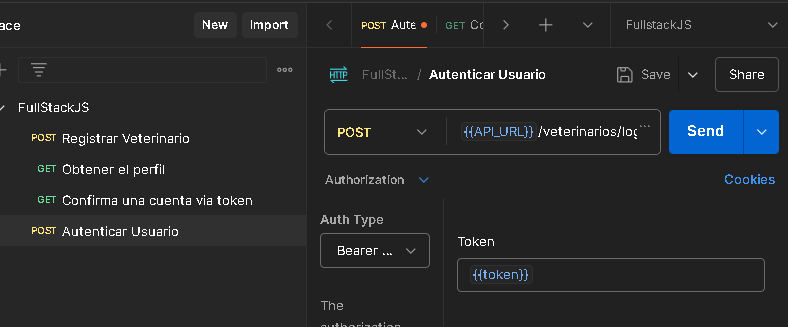
Para esto vamos a probar nos vamos a postman y colocamos el token:  


Pero, se recomienda que esto se guarde en variables, asi que nos vamos a “environments” y creamos una nueva variable



Una vez creada podemos introducir variables en la tabla y en “initial value” pegamos el token:



Y para usar la variable en las colecciones simplemente seleccionamos el enviroment en la parte superior derecha y con “{{ }}” dentro colocamos la variable:  


Y ya para ver la información del bearer web token al llamar a la funcion de auth, podemos poner req.headers de la siguiente manera (ojo, el bearer token se tiene que estar en la parte de autorización de postman en donde se hace la petición, en este caso en “obtener perfil”:

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

    console.log(req.headers);

    next();

};

Y de esta forma en consola nos saldrá la informacion que necesitamos saber.

Y para ver que tenga un json web token y que tenga el bearer podemos usar el siguiente código:

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');

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

    next();

};

Comprobar que el token sea correcto:

token = req.headers.authorization.split(' ')[1]; //De esta forma solo aparece el token sin el bearer

const decoded = jwt.verify(token, process.env.JWT\_SECRET); //Para verificar, colocamos el token y la palabra secreta de las variables de entorno

Despues para iniciar sesión con los datos que se tiene, se puede hacer lo siguiente (cabe mencionar que el token tiene el id del usuario, entonces es por eso que lo encuentra):

req.veterinario = await Veterinario.findById(decoded.id).select(

                "-password -token -confirmado"

            );

            return next();

En donde con req.veterinario se inicia una sesión, y con select quitar los datos que no quieres que se muestren, después con next() pasas al siguiente middleware.

Entonces en pocas palabras lo que hace esto es con postman al momento de ingresar y con el token que generaste anteriormente gracias a que autenticaste al usuario, ya que este token se genera con el id, se puede buscar y encontrar entonces ya solo se guarda en la variable decoded y después de inicia sesión con sus datos.

Accediendo a la información del usuario autenticado:

const perfil = (req,res) =>{

    const {veterinario} = req; //Podemos hacer esto ya que esto ya esta almacenado en node js porque se guardo la sesion.

    res.json({perifl: veterinario});

}

Comprobar el token:  
Para eso tenemos que sacar el token de la url, que lo hacemos con req.params.

Despues buscamos a un usuario que tenga ese token en la bd.

Si lo encuentra manda un mensaje que lo encontró, pero si no manda un error de que no lo encontró.

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

    const {token} = req.params;

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

    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 nuevo password:  
Primero obtenemos el token de la url y el password de password del body

Después comprobamos si el token existe y lo buscamos, si no existe manda un error.

Pero si existe entonces el token se vuelve null para que nadie mas lo pueda volver a usar y se cambia el password al introducido en req.body.

Se guarda y arroja un mensaje si es que se cumple correctamente.

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.error(error)

    }

};

Creación del modelo de pacientes:

Para esto necesitamos un nombre, propietario, email, alta, síntomas y el object id del veterinario.

import mongoose from "mongoose";

const pacientesSchema = mongoose.Schema({

    nombre:{

        type: String,

        required: true

    },

    propietario:{

        type: String,

        required: true

    },

    email:{

        type: String,

        required: true

    },

    alta:{

        type: Date,

        required: true

    },

    sintomas:{

        type: String,

        required: true

    },

    veterinario:{

        type:mongoose.Schema.Types.ObjectId,

        ref:'Veterinario'

    }

})

La referencia es el nombre que tu le hayas puesto al generar el schema, por ejemplo:

const Veterinario = mongoose.model("Veterinario",veterinarioSchema);

Despues también le pasamos la referencia y como van a ir los datos, además de el timestamps para que asi nos cree las columnas de editado y creado:

import mongoose from "mongoose";

const pacientesSchema = mongoose.Schema({

    nombre:{

        type: String,

        required: true

    },

    propietario:{

        type: String,

        required: true

    },

    email:{

        type: String,

        required: true

    },

    alta:{

        type: Date,

        required: true

    },

    sintomas:{

        type: String,

        required: true

    },

    veterinario:{

        type:mongoose.Schema.Types.ObjectId,

        ref:'Veterinario'

    }

},{

    timestamps:true //Esto para que nos cree las columnas de editado y creado

});

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

export default Paciente;

Despues agregamos en el index la ruta que será para los pacientes:

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

Una vez hecho eso creamos el archivo del controlador del paciente en la carpeta de “controllers”:

const agregarPaciente = (req, res) =>{

};

const obtenerPacientes = (req, res) =>{

};

export {agregarPaciente, obtenerPacientes};

Para después crear el archivo para pacientes de las rutas en la carpeta “routes” y agregar las respectivas rutas:

import express from 'express';

import { agregarPaciente, obtenerPacientes } from '../controllers/PacienteController.js';

const router = express.Router();

router.route('/')

    .post(agregarPaciente)

    .get(obtenerPacientes);

export default router;

De esta forma ya podemos empezar a trabajar con lo que va a hacer cada ruta con su respectivo controlador.

Para empezar a agregar pacientes el veterinario tiene que estar autenticado, es por eso que necesitamos checar si esto ocurre con el middleware que creamos, esto lo hacemos importando el middleware y colocándolo antes de “agregarPaciente”

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

const router = express.Router();

router.route('/')

    .post(checkAuth,agregarPaciente)

    .get(obtenerPacientes);

export default router;

Al tu utilizar el middleware, tienes que checar si este esta autorizado con el bearerToken, entonces lo que tienes que hacer es ver si el veterinario no está autenticado, tienes que autenticar su usuario en postman con el request de “Autenticar Usuario”, obtener el bearerToken, después en “crear un nuevo paciente” ingresas el bearer token de ese veterinario en la parte de “autorización” para que asi puedas obtener los datos de este, entre estos su id.

De esta forma ya puedes agregar el id del veterinario al registro del paciente:

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

    const paciente = new Paciente(req.body);

    paciente.veterinario = req.veterinario.\_id;

    try {

        const pacienteAlmacenado = await paciente.save();

        res.json(pacienteAlmacenado);

    } catch (error) {

        console.log(error)

    }

};

Obtener los pacientes del veterinario:

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

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

    res.json(pacientes);

};

Se usa la anterior sintaxis, en donde el ‘where’ se refiere a la columna donde va a buscar, en este caso la columna de veterinario:  


Y “equals(req.veterinario); es un filtro adicional para que solo se encuentren pacientes cuyo campo ‘veterinario’ coincida con el valor de req.veterinario, que en este caso es el que contiene la información del veterinario.

Obtener un paciente en específico:

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

    const {id} = req.params;

    const paciente = await Paciente.findById(id);

    if(paciente.veterinario.\_id.toString() !== req.veterinario.\_id.toString()){ //Cuando compares ids de mongodb, conviertelos a .toString para que puedas compararlos

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

    }

    if(paciente){

        res.json(paciente);

    }

};

En el código anterior, en const {id} = req.params; tomamos el id de req.params, que es el url, y de donde sacamos el id? Lo sacamos de routes, en donde indicamos que lo siguiente a pacientes/, va a ser el id:

 router.route('/:id')

Despues buscamos ese paciente por el id y lo guardamos en ‘paciente’

Si el id del campo ‘veterinario’ del paciente, y el id del token del veterinario que se obtiene por medio de la autorización no son iguales entonces arroja el mensaje ‘Accion no valida’.

Si son iguales retorna al paciente.

Actualizar paciente:

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

    const {id} = req.params;

    const paciente = await Paciente.findById(id);

    if(!paciente){

        return res.status(404).json({msg: 'No encontrado'});

    }

    if(paciente.veterinario.\_id.toString() !== req.veterinario.\_id.toString()){ //Cuando compares ids de mongodb, conviertelos a .toString para que puedas compararlos

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

    }

    //Actualizar paciente

    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 pacienteActializado = await paciente.save();

        res.json(pacienteActializado);

    } catch (error) {

        console.log(error)

    }

};

En el anterior código primero buscamos el paciente por el id, lo almacenamos, después si no lo encuentra manda un error que indica que no lo encontró.

Después al igual igual que el de obtener paciente, si el veterinario no es el mismo, entonces no puede actualizar al paciente.

Despues se actualiza al paciente con los datos que se le indiquen, si hay un campo del cual no se ingresa un valor, entonces se coloca el que estaba antes.

Se coloca un trycatch para guardar los datos, si se guarda te arroja los pacientes actualizados, pero si no, entonces te marcaria un error.

Eliminar un paciente:

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

    const {id} = req.params;

    const paciente = await Paciente.findById(id);

    if(!paciente){

        return res.status(404).json({msg: 'No encontrado'});

    }

    if(paciente.veterinario.\_id.toString() !== req.veterinario.\_id.toString()){ //Cuando compares ids de mongodb, conviertelos a .toString para que puedas compararlos

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

    }

    try {

        await paciente.deleteOne();

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

    } catch (error) {

        console.log(error);

    }

};

Para eliminar un paciente, es igual que los anteriores códigos, solamente que para eliminar el paciente es donde cambiaria, ya que se utiliza .deleteOne, hay otras formas que permiten borrarlo, que se pueden encontrar en la documentación de mongoose.

Creación del frontend:

Abrimos una terminal y usamos:  
npm init vite@latest

Después entramos al proyecto y ejecutamos el comando para instalar los packages:  
npm install

Después de eso podemos borrar app.css, borrar los iconos, borrar lo que hay dentro de index.css, y borrar lo que hay dentro de app.jsx.

Usualmente todo el trabajo lo vamos a hacer en src, es donde vamos a colocar hojas de estilo, stylecomponents, layout, components, etc.

Después instalamos tailwind y configurándolo:

Para instalarlo tenemos que poner en la consola:  
npm install -D tailwindcss@3 postcss autoprefixer

Después tenemos que configurarlo con:  
npx tailwindcss init -p

Esto lo que va a hacer es crearnos 2 documentos:

Postcss.config.js y Tailwind.confing.js

En Tailwind.confing.js ponemos:

/\*\* @type {import('tailwindcss').Config} \*/

export default {

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

  theme: {

    extend: {},

  },

  plugins: [],

}

Y colocamos en index.css:

@tailwind base;

@tailwind components;

@tailwind utilities;

Despues instalamos react-router-dom con el comando:

npm i react-router-dom

Despues importamos lo siguientes:

import {BrowserRouter, Route, Routes} from 'react-router-dom';

Routes: Permite agrupar diferentes rutas.

Route: Para una ruta en especifico.

Ahora con este código:

import {BrowserRouter, Route, Routes} from 'react-router-dom';

import AuthLayout from './layout/AuthLayout';

function App() {

  return (

    <BrowserRouter>

      <Routes>

        <Route path='/' element={<AuthLayout />}>

        </Route>

      </Routes>

    </BrowserRouter>

  )

}

export default App

El path es la dirección de la pagina y el element es lo que quieras que este renderice.

En authlayout cargamos esto:

<>

    <h1>Desde Auth</h1>

</>

A los signos vacios se les conoce como ‘fragment’, y no genera mas código html.

Ahora para renderizar lo mismo en todas las paginas o algo que se repita, vamos a colocar como ejemplo la de login:

import {BrowserRouter, Route, Routes} from 'react-router-dom';

import AuthLayout from './layout/AuthLayout';

import Login from './pages/Login';

function App() {

  return (

    <BrowserRouter>

      <Routes>

        <Route path='/' element={<AuthLayout />}>

          <Route index element={<Login />}/>

        </Route>

      </Routes>

    </BrowserRouter>

  )

}

export default App

Route define y agrupa que todas las paginas que esten dentro de el tendrán el authLayout.

Despues el Route que esta dentro, la etiqueta ‘index’ define que ese va a ser el contenido principal y ‘element’ define la pagina que se va a cargar.

Para que cargue el contenido, tenemos que poner el authlayout el outlet:

import React from 'react'

import { Outlet } from 'react-router-dom'

function AuthLayout() {

  return (

    <>

    <h1>Desde Auth</h1>

    <Outlet/>

    </>

  )

}

export default AuthLayout

Que lo que hace es que en la parte del outlet mete por ejemplo lo que hay en login.

Creacion de paginas:  
Creamos las paginas en la carpeta de pages del front, después los importamos al index para crear el routing:

import {BrowserRouter, Route, Routes} from 'react-router-dom';

import AuthLayout from './layout/AuthLayout';

import Login from './pages/Login';

import Registrar from './pages/Registrar';

import OlvidePassword from './pages/OlvidePassword';

import ConfirmarCuenta from './pages/ConfirmarCuenta';

function App() {

  return (

    <BrowserRouter>

      <Routes>

        <Route path='/' element={<AuthLayout />}>

          <Route index element={<Login />}/>

          <Route path='registrar' element={<Registrar />}/>

          <Route path='olvide-password' element={<OlvidePassword />}/>

          <Route path='confirmar/:id' element={<ConfirmarCuenta />}/>

        </Route>

      </Routes>

    </BrowserRouter>

  )

}

export default App

El “:id” se maneja al igual que en la parte del backend.

Usar useState para la parte del registro: