Configurando MongoDB Atlas – mongoose – express – morgan – bycryptjs - jsonwebtoken.

Parte – 3

Una vez configurado Docker y operativa la imagen de mongodb, ahora nos toca hacer la conexión con la base de datos.

Para hacer gestionar mongo vamos a instalar mongoose desde el terminal de Code.

npm install mongoose

Este módulo es necesario para gestionar el mongodb desde nuestro fichero de javascript, es decir, hacer la conexión y hacer consultas CRUD.

El siguiente código lo voy a escribir en el fichero "**bd.js**" y nos muestra la importación del módulo "mongoose". La conexión a la base de datos mediante un string denominado "**connectionString**" y un mensaje por consola que nos avisa de la conexión con la base de datos creada en mongo.

Fijaros que introduzco el código entre el try{}catch{}, puesto que es buena práctica controlar posibles errores en la conexión.

```
import mongoose from "mongoose"; //para gestionar todas las operaciones CRUD sobre la BBDD de MongoDB.

export const connectDB = async() => {  //función flecla que voy a llamar desde index.js después de haber arrancado el servidor de express.
    //Utilizar el try-catch para capturar los posibles errores de conexión a nuestra BBDD.
    try{
        //const                    connectionString                    = "mongodb+srv://mariosantos:20232024Raquel@cluster0.mmvisjt.mongodb.net/";
        const connectionString = "mongodb://localhost/dam2"  //Creo el string de la base de datos y que voy a utilizar como parámetro de entrada en mongoose.connect()
        mongoose.connect(connectionString); //lanzamos la conexión.
        console.log("Base de Datos Conectada");
    } catch (error) {
        console.log(error);  //visualizamos el posible error.
    }
}
```


IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891

Para arrancar la conexión con la BBDD, importamos la variable connectDB() en el fichero index.js.

```
import aplicacion from "./aplicacion.js";
import {connectDB} from "./db.js";


aplicacion.listen(4000); //Le decimos a nuestro pequeño servidor que se quede escuchando en el puerto 3000.
console.log("Servidor abierto en puerto 4000"); //Ponemos un mensaje de comprobación.

connectDB(); //Conexión a la base de datos.
```

Fijaros que en este caso, cuando importamos la variable "connectDB", lo añadimos entre llaves porque a la hora de exportar la variable no le hemos puesto el "default".

En el momento en el que queremos registrar/guardar datos en nuestra base de datos debemos tener en cuenta que mongodb tiene alguna peculiaridad como puede ser que puedo mandar únicamente algunos campos de una colección de datos o incluso, campos nuevos. Esto podría ser un problema a la hora de tener cierta consistencia en nuestros clientes. Para solucionar esto, lo que vamos a hacer es un esquema del modelo de datos que queremos guardar. Esto significa que vamos a verificar los datos antes de mandarlos al backend para que se guarden.

Vamos a crear el fichero "**user.model.js**" que se guardará dentro de nuestra carpeta de proyecto "**modelos**".

```
import mongoose from 'mongoose'; //importamos el módulo para trabajar con mongo.

const modeloUsuario = new mongoose.Schema({     //Creamos modelUsuario, que es el modelo de datos a verificar antes de mandar al backend
    username: {                                 // Campo username de tipo string.
        type: String,
    },
    email: {                                    //Campo Email de tipo string.
        type: String,
    },
       password: {                                      //Campo password de tipo
      string.
        type: String,
    }
}, {
    timestamps: true,
});
```


IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891

```
export default mongoose.model("usuario", modeloUsuario); //exportamos el modelo de datos "modeloUsuario" y le vamos a denominar
                                                         //usuario que es con la que trabajaremos desde el fichero autor.controlador.js.
```

Este es el modelo de datos que vamos a utilizar para verificar los datos. Como podéis observar, tengo dos conjuntos de datos:

- **username, mail, password**: los valores con los que trabajamos con operaciones CRUD.
- **timestamps**: vamos a registrar la fecha de creación.

En este modelo de datos, lo único que hacemos es declarar los campos y el tipo de datos que vamos a guardar que, en este caso, son string. Si lo dejamos así, podríamos dejar campos sin añadir. Por ejemplo, el siguiente ejemplo de registro de datos desde nuestro cliente thunder client, es correcto:

```
{
 "username" : "guillermo",
 "mail" : "guillermo@odoo.com",
 "password" : "guillermo123"
}
```

Pero, el siguiente ejemplo, también es correcto:

```
{
 "username" : "sofía",
 "mail" : "",
 "password" : ""
}
```

Esto puede ocasionar algún problema, como inconsistencia de datos en nuestra BBDD. Para solucionar esto, le podemos meter más atributos a nuestros campos para hacerlos obligatorios, eliminar espacios en blanco, etc…

Nuestro fichero “user.model.js” quedaría de la siguiente forma:

IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891

```
import mongoose from 'mongoose'; //importamos el módulo para trabajar con mongo.

const modeloUsuario = new mongoose.Schema({     //Creamos modelUsuario, que es el modelo de datos a verificar antes de mandar al backend
    username: {                                 // Campo username de tipo string.
        type: String,
        require: true,
        trim: true,    //quita los espacios en blanco innecesarios.
    },
    email: {                                    //Campo Email de tipo string.
        type: String,
        require: true,
        trim: true,
        unique: true,
    },
    password: {                                 //Campo password de tipo string.
        type: String,
        require: true,
    }

}, {
    timestamps: true,
});

export default mongoose.model("usuario", modeloUsuario); //exportamos el modelo de datos "modeloUsuario" y le vamos a denominar
                                                         //usuario que es con la que trabajaremos desde el fichero autor.controlador.js.
```

Observar que los parámetros que acompañan a los campos de datos:

- **type**: tipo de dato.
- **require**: campo obligatorio si le pasamos un valor de true.
- **unique**: no se puede repetir en las diferentes colecciones de datos.
- **trim**: eliminamos espacios en blanco que se han insertado de forma innecesaria.

Una vez configurado el fichero .js que vamos a utilizar como verificador de datos, estamos listos para hacer nuestro fichero controlador de rutas para que, ahora si, inserte los datos en nuestra bbdd.

En este punto, vamos a trabajar sobre el fichero “autor.controlador.js”. Para empezar, vamos a modificar nuestra función flecha “register”. Recordar que esta función es ejecutada cuando desde el front hacemos mención a la ruta “**localhost://4000/register**”.

En principio, tenemos el fichero de la siguiente forma:

```
import usuario from "../modelos/user.model.js";
import bcrypt from "bcryptjs";
import {crearTokenAcceso} from "../librerias/jwt.js";
import userModel from "../modelos/user.model.js";

export const login = async (req, res) => res.send("legeando");
export const register = async(req, res) => res.send("registrando");
export const logout = async(req, res) => res.send("saliendo");
Vamos a comentar la función register y vamos a crear esta otra:
```


IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891

```
export const register = async (req, res) => {
    const {email, password, username} = req.body;

    try{
        const nuevoUsuario = new usuario({
            username,
            email,
            password,
        });

        const usuarioSalvado = await nuevoUsuario.save();

        res.json({
            message: "Usuario Creado Satisfactoriamente",
            id: nuevoUsuario._id,
            username: nuevoUsuario.username,
            email: nuevoUsuario.email,
            createdAt: nuevoUsuario.createdAt,
            updatedAt: nuevoUsuario.updatedAt,
        });
    } catch (error){
        res.status(500).json({ message: error.message});
    }

}
```

Como observamos en esta función flecha con lo primero que nos encontramos es con "**const {email, password, username} = req.body**". Añadimos a nuestras variables email, password y username, los valores que pasamos desde nuestro cliente front thunder client.

```
{
 "username" : "guillermo",
 "mail" : "guillermo@odoo.com",
 "password" : "guillermo123"
}
```

Fijaros que cuando recogemos los datos del front, utilizamos el parámetro de entrada "req" y cuando le queremos enviar datos al cliente, utilizamos el parámetro "res".

Al tratarse de una conexión a una base de datos situada en un servidor, lo correcto es la utilización del try-catch, de tal manera que podamos capturar el evento de error generado en el caso de que algo no funcione correctamente.

IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891

A continuación, en la parte del “try” nos encontramos con una declaración de una variable del tipo que hemos declarado anteriormente en el fichero “user.model.js” y que va a tomar los valores que hemos introducido en nuestro frontend “thunder client”, y nos va a servir para verificar los datos antes de mandarlos al backend.

Una vez verificados esos datos, los mandamos a guardar como una colección nueva en nuestra bbdd. Si no se cumple alguna de las condiciones que hemos configurado, no se guardará la colección de datos. Para guardar los datos utilizamos la orden:

```
const usuarioSalvado = await nuevoUsuario.save();
```

Le pasamos un await, para pausar la ejecución del código hasta que se escriben los datos en la bbdd.

Por último, lo que hace el código es imprimir el json con los datos que se han guardado en la colección:

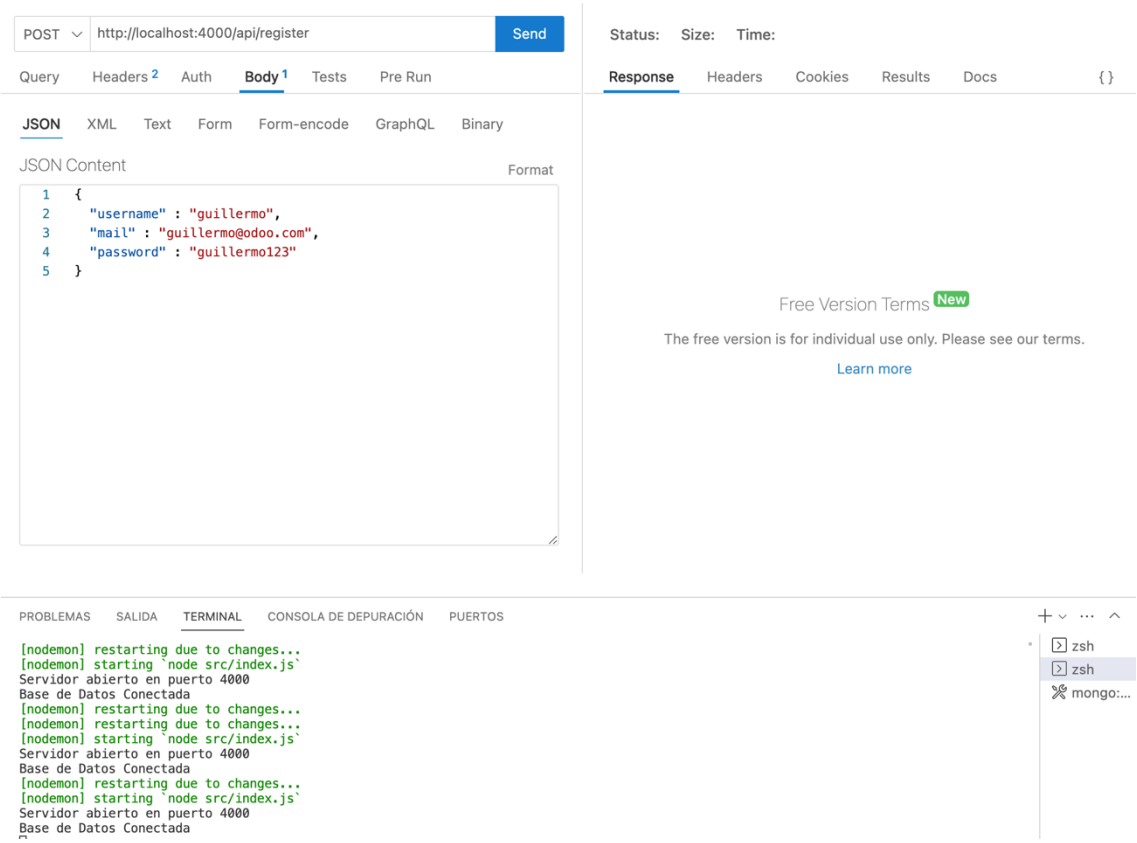
```
res.json({
            message: "Usuario Creado Satisfactoriamente",
            id: nuevoUsuario._id,
            username: nuevoUsuario.username,
            email: nuevoUsuario.email,
            createdAt: nuevoUsuario.createdAt,
            updatedAt: nuevoUsuario.updatedAt,
});
```

En la parte del código perteneciente a “catch”, lo que hacemos es capturar ese evento de error e imprimir el mensaje de este por consola.

Una vez terminada esta función flecha “register” tendremos capacidad de guardar los datos en la bbdd.

En el siguiente documento, vamos a hablar de lo que es un token y de cómo podemos configurarlo en nuestro backend ejemplo.

IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891



La función flecha “login”, quedaría de una forma muy similar a la función flecha de “register”, pero antes de verla, vamos a encriptar y a configurar un “token”.

El paso de encriptación lo vamos a realizar en nuestra función flecha “register”, que es desde la que enviamos la información al backend.

Lo primero que hacemos es instalar, desde nuestro terminal de Visual Code, el módulo de **bcryptjs**.

npm i bcryptjs

En el fichero “autor.controlador.js” vamos a importar este módulo:

```
import bcrypt from "bcryptjs";
```

Lo siguiente es encriptar nuestra “password”. Insertamos la siguiente orden:

```
const contrasenaHash = await bcrypt.hash(password, 10); //Encriptamos nuestra contraseña.
```

Creamos la variable “contrasenaHash” y llamamos a la función “bcrypt.hash()” que es una función asíncrona y le pasamos el await para que haga una pausa hasta que encripta

IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891

el password. A esta función se le pasa dos parámetros, uno es la password que le pasamos a la función req.body de la función register(). Luego es el nivel de seguridad, este valor es numérico. Si el valor es muy alto, el nivel de seguridad es más alto, pero también tardará más. Lo normal y el valor que viene por defecto es 10.

A partir de este momento, tenemos encriptada la password, ahora se lo pasamos a nuestro modelo de datos, antes de generar la colección de datos en nuestra BBDD.

El código de la función register() nos quedaría de la siguiente manera.

```
export const register = async (req, res) => {
    const {email, password, username} = req.body;

    const contrasenaHash = await bcrypt.hash(password, 10); //Encriptamos
nuestra contraseña.

    try{
        const nuevoUsuario = new usuario({
            username,
            email,
            password: contrasenaHash, //enviamos al backend nuestra contraseña
encriptada.
        });

        const usuarioSalvado = await nuevoUsuario.save();

        res.json({
            message: "Usuario Creado Satisfactoriamente",
            id: nuevoUsuario._id,
            username: nuevoUsuario.username,
            email: nuevoUsuario.email,
            password: nuevoUsuario.password,
            createdAt: nuevoUsuario.createdAt,
            updatedAt: nuevoUsuario.updatedAt,
        });
    } catch (error){
        res.status(500).json({ message: error.message});
    }
}
```

Una vez configurada la password, vamos a crear un “token”.

token: Si buscamos por internet nos encontramos con que un token es una unidad de valor basada en criptografía y emitida por una entidad privada en una blockchain. Nosotros lo utilizamos a modo de pase, es decir, el backend le pasa un valor de “acceso”

IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891

al frotend y cuando el frontend quiere enviar algo al backend, debe mostrar ese valor de “acceso”. En nuestro miniproyecto, lo que hacemos es pasarlo al frontend y guardarlo en una cookie.

En este punto, vamos a trabajar con los siguientes ficheros:

- fichero “jwt.js” dentro de la carpeta “librerías” de nuestro proyecto.
- Fichero “config.js”.
- Fichero “autor.controlador.js”.

Lo primero es cargar el módulo de “**jsonwebtoken”** que es el que nos va a permitir trabajar con diferentes token’s.

npm i jsonwebtoken

Empezamos con el fichero “jwt.js”:

```
import jwt from "jsonwebtoken"   //Importamos el modulo para creación de
token.
import { TOKEN_SECRET_KEY } from "../config.js"; //Creamos una variable dentro
del fichero config.js con el valor de la clave.
export function crearTokenAcceso(payload){  //Creamos nuestra función que
recibe como parámetro de entrada el id del usuario.

    return new Promise((resolve, reject) => { //Utilizamos una función
Promise, si todo ha ido bien nos devuelve el resolve y si hay algún problema,
nos devuelve el reject.
        jwt.sign(  //Para crear nuestro token, usamos el método sign que le
pasamos el parámetro de entrada (id), la clave que vamos a utilizar y el
tiempo de vida.
            payload,
            TOKEN_SECRET_KEY,
            {
                expiresIn: "1d",
            },
            (err, token) => {
                if(err) reject(err); //Si ocurre algún error, nos devuelve
reject(error) pero si todo ha ido bien, nos devuelve el token ya codificado.
                resolve(token)
            }
        );
    });
}
```

La variable que hemos creado en el fichero **config.js** queda de la siguiente manera:

```
export const TOKEN_SECRET_KEY = "contrasena123";
```


IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891

En nuestro fichero "**autor.controlador.js**" vamos a llamar a la función creada en el punto anterior:

```
const token = await crearTokenAcceso({id: usuarioSalvado._id}) //le pasamos
como parámetro el id del usuario.
res.cookie('token', token) //Creamos la cookie que va a guardar el valor del
token
```

La función register(), queda como se muestra a continuación:

```
export const register = async (req, res) => {
    const {email, password, username} = req.body;

    const contrasenaHash = await bcrypt.hash(password, 10); //Encriptamos
nuestra contraseña.

    try{
        const nuevoUsuario = new usuario({
            username,
            email,
            password: contrasenaHash, //enviamos al backend nuestra contraseña
encriptada.
        });

        const usuarioSalvado = await nuevoUsuario.save();

        const token = await crearTokenAcceso({id: usuarioSalvado._id})

        res.cookie('token', token)
        res.json({
            message: "Usuario Creado Satisfactoriamente",
            id: nuevoUsuario._id,
            username: nuevoUsuario.username,
            email: nuevoUsuario.email,
            password: nuevoUsuario.password,
            createdAt: nuevoUsuario.createdAt,
            updatedAt: nuevoUsuario.updatedAt,
        });
    } catch (error){
        res.status(500).json({ message: error.message});
    }
```

A partir de aquí, podemos terminar de configurar nuestras funciones de "login" y de "logout".

IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891

La función de login() es similar a la de register() y nos quedaría de la siguiente manera:

```
export const login = async (req, res) => {
    const { email, password } = req.body;

    //const contrasenaHash = await bcrypt.hash(password, 10);

    try{

        const usuarioEncontrado = await usuario.findOne({email}); //Buscamos
el usuario.

        if (!usuarioEncontrado) return res.status(400).json({message: "Usuario
no encontrado"});

        const coincidencia = await bcrypt.compare(password,
usuarioEncontrado.password); //comparamos las contraseñas.

        if (!coincidencia) return res.status(400).json({ message: "Contraseña
incorrecta"});

        const token = await crearTokenAcceso({id: usuarioEncontrado._id})

        res.cookie('token', token)
        res.json({
            message: "Usuario encontrado Satisfactoriamente",
            id: usuarioEncontrado._id,
            username: usuarioEncontrado.username,
            email: usuarioEncontrado.email,
            createdAt: usuarioEncontrado.createdAt,
            updatedAt: usuarioEncontrado.updatedAt,
        });
    } catch (error){
        res.status(500).json({ message: error.message});
    }
```

Fijaros que cuando nos logeamos, lo primero que hacemos es buscar en nuestra bbdd si existe o no ese usuario. Esto lo hacemos porque hemos importado el modelo de datos:

```
import usuario from "../modelos/user.model.js";
```

y hacemos la búsqueda a través del comando:

```
const usuarioEncontrado = await usuario.findOne({email}); //Buscamos el
usuario.
```


IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891

El resto de la función es más o menos igual que el de register().

Por último, nos queda la función logout() en la que borramos la cookie que guarda el token.

```
export const logout = (req, res) => {
    res.cookie("token", "", {
        expires: new Date(0),
    });
    return res.sendStatus(200);
}
```


IES VLLABLANCA
C/Villablanca, 79 - 28032 Madrid
Teléfono 917 752 214 – Fax 917 766 577
e-mail ies.villablanca.madrid@educa.madrid.org
C.C.: 28039891