SERVER part 2 NODEJS HERRERA

- Vamos a crear una DB con tres colecciones
  - Usuarios
  - Categorías - comparte el ObjectId del Usuario
  - Productos - comparte el ObjectId de la Categoría y el ObjectId del Usuario

---

Configuración de MongoDB

- Creo la DB con Mongo Compass
- El string de conexion es

> mongodb://localhost:27017/NodeServer

- **Cambiar localhost por 127.0.0.1** en .env
- Coloco el string de conexión en una variable de entorno que llamo MONGODB
- Instalo mongoose con npm
- Creo la carpeta /database/config.js

```
import mongoose from 'mongoose'

const dbConnection = async()=>{
    try {
       await mongoose.connect(process.env.MONGODB, {
        useNewUrlParser: true,
        useUnifiedTopology: true
       })  //si esto falla es atrapado por el catch
        console.log("Base de datos conectada!")

    } catch (error) {
        console.log(error)
        throw new Error("Error en la conexión con la DB")
    }
}

export default dbConnection
```

- Uso la función en el server
- Creo el método conectarDB y lo llamo en el constructor

```
import express from 'express'
import cors from 'cors'
import userRouter from '../routes/user.routes.js'
import dbConnection from '../database/config.js'

export class Server {
```

```
    constructor(){
        this.app = express()
        this.port = process.env.PORT
        this.usuariosPath = '/api/usuarios'

        //conexion a la DB
        this.conectarDB()

        //Middlewares ( en el constructor van a ejecutarse al levantar el servidor
)
        this.middlewares()

       //Rutas
        this.routes()
    }

    async conectarDB(){
        await dbConnection()
    }

    middlewares(){
        this.app.use(express.static('public')) //Esto es lo que se va a servir en
'/'
        this.app.use(express.urlencoded({extended: false}))
        this.app.use(express.json())
        this.app.use(cors())
    }

    routes(){

        this.app.use(this.usuariosPath, userRouter)
    }

    listen(){
        this.app.listen(this.port, ()=>{
            console.log(`Server corriendo en puerto ${this.port}`)
        })
    }
}
```

Modelo de Usuario

- Creo en la carpeta /models/usuario.js
- usuario.js

```
import mongoose from 'mongoose'

const usuarioSchema = mongoose.Schema({
```

```
    nombre:{
        type: String,
        required: [true, 'El nombre es obligatorio']
    },

    correo:{
        type: String,
        required: [true, 'El correo es obligatorio'],
        unique: true
    },

    password:{
        type: String,
        required: [true, 'El password es obligatorio']
    },

    img:{
        type: String,
    },

    rol:{
        type: String,
        required: true,
        enum: ['ADMIN_ROLE', 'USER_ROLE'] //el rol tiene que ser uno de estos dos
    },

    estado:{
        type: Boolean,
        default: true
    },

    googgle:{ //si el usuario ha sido creado por google
        type: Boolean,
        default: false
    }

})

const Usuario= mongoose.model('Usuario', usuarioSchema) //Usuario es el nombre de
la colección. Mongoose la eñadirá la s

export default Usuario
```

POST: creando un usuario en la colección

- En el controlador usuariosPost importo el usuario
- **NOTA: se ha cambiado los nombres de los controladores por usuariosGet, usuariosPost, etc**
- Método POST en la ruta '/api/usuarios' en Thunder Client
- Debo introducir todos los datos requeridos por el modelo en Thunder Client (nombre, correo, password)
    - Si no saltará un error

```
const usuariosPost = async (req,res) => {
    const body = req.body
    const usuario = new Usuario(body)
    await usuario.save()

    res.json(usuario)

}
```

- Le paso el JSON a Thunder Client

```
{
 "nombre": "Pedro",
  "correo": "pedro@correo.com",
  "password": "1234",
  "rol": "ADMIN_ROLE"
}
```

- Me devuelve esto:

```
{
  "nombre": "Pedro",
  "correo": "pedro@correo.com",
  "password": "1234",
  "rol": "ADMIN_ROLE",
  "estado": true,
  "googgle": false,
  "_id": "6448f6aa9c59978a0856b149",
  "__v": 0
}
```

- Hacen falta varias cosas: validar el rol, encriptar el password, verificar si es de google...pero al menos ya estamos grabando en la DB!

BcryptJS - encriptando la contraseña

- Nunca confies en el Frontend. Hay que validar todos los endpoints
- Uso desestructuración del body en el controlador con los campos que me interesan
- Instalo bcryptjs con npm y lo importo en usuario.controller.js
    - Primero genero el salt. Tiene 10 vueltas por defecto
    - Hasheo con el password que he desestructurado del body el salt como segundo parámetro
- user.controller.js

```
const usuariosPost = async (req,res) => {
    const {nombre, correo, password, rol}= req.body
    const usuario = new Usuario({nombre, correo, password, rol})

    //verificar si el correo existe

    const salt = bcryptjs.genSaltSync() // tiene el valor 10 por defecto
    usuario.password = bcryptjs.hashSync(password, salt)

    await usuario.save()

    res.json(usuario)  //no debo regresar el password, luego lo configuro

}
```

- password encriptado!

Validar campos obligatorios - Email

- Puede ser que necesite validar el correo en otras pantallas (controladores)
- Para hacer validaciones voy a usar express-validator (lo instalo con npm)
- Tengo que validar además de que sea un correo válido, que exista y que sea único
- Verifico que exista
- user.controller.js

```
const usuariosPost = async (req,res) => {


    const {nombre, correo, password, rol}= req.body
    const usuario = new Usuario({nombre, correo, password, rol})

    //verificar si el correo existe
    const existeEmail = await Usuario.findOne({correo})

    if(existeEmail){
        return res.status(400).json({
            msg: "El correo ya está registrado"
        })
    }

    const salt = bcryptjs.genSaltSync() // tiene el valor 10 por defecto
    usuario.password = bcryptjs.hashSync(password, salt)

    await usuario.save()

    res.json(usuario)
```

```
}
```

- Express-validator es un conjunto de middlewares que puedo ejecutar antes del controlador en el router
- Después del path, dentro de un arreglo coloco el middleware
- uso check, introduzco el nombre del campo como primer argumento y el mensaje de error como el segundo
- uso isEmail para verificar que es un correo válido
- user.routes.js

```
router.post('/', [check('correo', 'El correo no es válido').isEmail()],
userController.usuariosPost)
```

- El check va almacenando todos los errores en la req
- En usuariosPost puedo confirmarlo. Para ello uso la función validationResult

```
const usuariosPost = async (req,res) => {

    //recojo errores del express validator
    const errors = validationResult(req)

    if(!errors.isEmpty()){
        return res.status(400).json({
            errors
        })
    }

    const {nombre, correo, password, rol}= req.body
    const usuario = new Usuario({nombre, correo, password, rol})

    //verificar si el correo existe
    const existeEmail = await Usuario.findOne({correo})

    if(existeEmail){
        return res.status(400).json({
            msg: "El correo ya está registrado"
        })
    }

    const salt = bcryptjs.genSaltSync() // tiene el valor 10 por defecto
    usuario.password = bcryptjs.hashSync(password, salt)

    await usuario.save()

    res.json(usuario)

}
```

- Ahora si introduzco un correo sintácticamente no válido me devuelve esto

```
{
  "errors": {
    "errors": [
      {
        "type": "field",
        "value": "pereTT",
        "msg": "El correo no es válido",
        "path": "correo",
        "location": "body"
      }
    ]
  }
}
```

- hay una manera mejor de manejar los errores que arroja express-validator. Se verá más adelante

---

Validar todos los campos necesarios

- Es hora de validar todos los campos obligatorios en el modelo de Usuario
- Creo otra linea en el arreglo del post para hacer la validación del nombre(en user.routes.js)

```
router.post('/',
[check('correo', 'El correo no es válido').isEmail(),
 check('nombre', 'El nombre es obligatorio').not().isEmpty()
], userController.usuariosPost)
```

- Ahora el password y el role

```
router.post('/',
[check('correo', 'El correo no es válido').isEmail(),
 check('nombre', 'El nombre es obligatorio').not().isEmpty(),
 check('password', 'El password es obligatorio y debe de ser de más de 6
letras').isLength({min: 6}),
 check('rol', "No es un rol válido").isIn(['ADMIN_ROLE', 'USER_ROLE']) //luego se
validará contra la DB

], userController.usuariosPost)
```

- Más adelante se protegerá esta ruta, ya que nadie debería cambiar el role salvo el administrador
- Si tengo que hacer validaciones en otras rutas significaría que tengo que copiar el mismo código varias veces
    - Al menos el de extraer los errores de la función validationResult
- Esto viola el principio DRY (Dont Repeat Yourself)

- Por ello creo la carpeta en la raíz /middlewares/validar-campos.js
- Debp importar validationResult, y pasarle la req, la res, y next para que el middleware pase al siguiente
- validar-campos.js

```
import { validationResult } from "express-validator"


export const validarCampos = (req,res, next)=>{

    const errors = validationResult(req)

    if(!errors.isEmpty()){
        return res.status(400).json({
            errors
        })
    }
    next()

}
```

- Este middleware va a ser el último que coloque después de las validaciones

```
router.post('/',
[check('correo', 'El correo no es válido').isEmail(),
 check('nombre', 'El nombre es obligatorio').not().isEmpty(),
 check('password', 'El password es obligatorio y debe de ser de más de 6
letras').isLength({min: 6}),
 check('rol', "No es un rol válido").isIn(['ADMIN_ROLE', 'USER_ROLE']),
    validarCampos //lo coloco el último porque una vez hechas las validaciones
ejecuto la que va a revisar los errores
], userController.usuariosPost)
```

- Lo siguiente es validar que los roles existan en una DB y no en un arreglo en duro

---

Validar Rol contra DB

- Creo en Mongo Compass una nueva colección llamada roles
- Con Add Data inserto un documento json

```
{
  "rol": "ADMIN_ROLE"
}
```

- Hago lo mismo USER_ROLE y VENTAS_ROLE

- **NOTA**: para que VENTAS_ROLE no de error de momento lo añado al arreglo del enum en el esquema de Usuario
- Hay que crear un modelo. El nombre del archivo tiene el mismo que el de la colección pero sin la s

```
import mongoose from 'mongoose'

const roleSchema = mongoose.Schema({
    rol:{
        type: String,
        required: [true, 'El rol es obligatorio']
    }

})

const Role = mongoose.model('Role', roleSchema)

export default Role
```

- Hago la validación
- Defino el rol con un string vacío para que en caso de que no venga choque con la validación
- Busco con findOne que haga match con el rol
- Si no existe el rol lanzo un error

```
import express from 'express'
import userController from '../controllers/user.controller.js'
import {check} from 'express-validator'
import { validarCampos } from '../middlewares/validar-campos.js'
import Role from '../models/role.js'

const router = express.Router()

router.get('/', userController.usuariosGet)

router.post('/',
[check('correo', 'El correo no es válido').isEmail(),
 check('nombre', 'El nombre es obligatorio').not().isEmpty(),
 check('password', 'El password es obligatorio y debe de ser de más de 6
letras').isLength({min: 6}),
 check('rol').custom(async(rol="")=>{      //validación contra la DB
    const existeRol = await Role.findOne({rol})

    if(!existeRol){
        throw new Error("El rol no existe")
    }

 }),
    validarCampos //lo coloco el último porque una vez hechas las validaciones
ejecuto la que va a revisar los errores
], userController.usuariosPost)
```

```
router.patch('/:id', userController.usuariosPut)

router.delete('/', userController.usuariosDelete)


export default router
```

- Si ahora le paso un rol no válido me lanza el error

```
{
  "errors": {
    "errors": [
      {
        "type": "field",
        "value": "ADMI_ROLE",
        "msg": "El rol no existe",
        "path": "rol",
        "location": "body"
      }
    ]
  }
}
```

Centralizar la validación del rol

- Optimización del código
- Cuando haga el update del usaurio (y el delete) voy a tener que volver a validar el rol
- Creo una nueva carpeta en la raíz llamada helpers/db-validators.js
- Corto la función de validación de rol y la pego en el archivo
-

```
import Role from "../models/role.js"

export const esRolValido = async(rol="")=>{      //validación contra la DB
    const existeRol = await Role.findOne({rol})

    if(!existeRol){
        throw new Error("El rol no existe")
    }

}
```

- Cuando tengo una función (un callback) cuyo primer argumento es el mismo que le paso como parámetro puedo obviarlo

```
rol => esRolValido(rol)  // === esRolValido
```

- user.routes.js

```
router.post('/',
[check('correo', 'El correo no es válido').isEmail(),
 check('nombre', 'El nombre es obligatorio').not().isEmpty(),
 check('password', 'El password es obligatorio y debe de ser de más de 6
letras').isLength({min: 6}),
 //check rol
 check('rol').custom(esRolValido),
    validarCampos
], userController.usuariosPost)
```

- Puedo crear métodos en el modelo. Por ejemplo, para que no me devuelva el password cuando imprimo el usuario creado
- Cuando llamo el modelo y lo quiero imprimir, el modelo llama a .toJSON
- Puedo modificar el modelo para que no devuelve el password. Podría sobreescribir findOne también
- Quiero sobreescribir el .toJSON
- Tiene que ser una función normal porque voy a usar el this, y en una función de flecha el this apunta fuera de la misma

```
import mongoose from 'mongoose'

const usuarioSchema = mongoose.Schema({

    nombre:{
        type: String,
        required: [true, 'El nombre es obligatorio']
    },

    correo:{
        type: String,
        required: [true, 'El correo es obligatorio'],
        unique: true
    },

    password:{
        type: String,
        required: [true, 'El password es obligatorio']
    },

    img:{
        type: String,
    },

    rol:{
        type: String,
```

```
        required: true,
        enum: ['ADMIN_ROLE', 'USER_ROLE', 'VENTAS_ROLE']
    },

    estado:{
        type: Boolean,
        default: true
    },

    googgle:{ //si el usuario ha sido creado por google
        type: Boolean,
        default: false
    }

})

usuarioSchema.methods.toJSON = function(){
    const {__v, password, ...usuario} = this.toObject() //esto me va a generar mi
instancia pero con sus valores respectivos, uso la desestructuración
    return usuario
}

const Usuario= mongoose.model('Usuario', usuarioSchema)

export default Usuario
```

- Ahora, cuando me devuelve el objeto usuario desde el controlador lo hace sin los campos __v y password

---

Custom Validation - EmailExiste

- Le paso el correo y hago la verificación en la DB con async await
- Uso throw new Error para lanzar el error (no tengo el res) en caso de que encuentre el correo
- En db-validators.js

```
import Role from "../models/role.js"
import Usuario from "../models/usuario.js"

export const esRolValido = async(rol="")=>{      //validación contra la DB
    const existeRol = await Role.findOne({rol})

    if(!existeRol){
        throw new Error("El rol no existe")
    }

}

export const existeMail = async(correo)=>{
    const existeEmail = await Usuario.findOne({correo})
```

```
    if(existeEmail){
       throw new Error("El email ya existe")
    }

    return existeMail
}
```

- Lo añado como una validación del express validator con check y custom

```
router.post('/',
[check('correo', 'El correo no es válido').isEmail(),
 check('nombre', 'El nombre es obligatorio').not().isEmpty(),
 check('password', 'El password es obligatorio y debe de ser de más de 6
letras').isLength({min: 6}),
 check('rol').custom(esRolValido),
 check('correo').custom(existeMail),
    validarCampos //lo coloco el último porque una vez hechas las validaciones
ejecuto la que va a revisar los errores
], userController.usuariosPost)
```

PUT - Actualizar información del usuario

- user.routes.js

```
const usuariosPut = async (req,res) =>{
    const {id} = req.params
    const {password, google, ...resto} = req.body //extraigo el viejo password y
lo añado con resto.password en caso de que venga


    if(password){ // si viene el password tengo que volver a encriptarlo
        const salt = bcryptjs.genSaltSync()
        resto.password = bcryptjs.hashSync(password, salt)
    }

    const usuario = await Usuario.findByIdAndUpdate(id, resto) // le paso el id y
el resto como valor a actualizar

    if(!usuario){   // verifico que exista el usuario
        res.status(400).json({
            msg: "No existe usuario"
        })
    }


    res.status(200).json({
        usuario
    })
```

```
}
```

- Para que findByIdAndUpdate me devuelva el objeto actualizado debo añadirle el objeto new: true

> const usuario = await Usuario.findByIdAndUpdate(id, resto,{new: true})

- Si el correo diera problemas porque el correo ya existe, debo sacarlo en la desestructuración y hacer una validación posterior
- Puedo seguir la misma estrategia que con el password y añadirlo con resto.correo = correo
- Si en el body le paso un _id de mongo válido va a chocar con la DB. Hay que evitar que haya un _id en el body
- Lo extraigo de la desestructuración en el controlador. El método put de user.controller.js queda así

```
const usuariosPut = async (req,res) =>{
    const {id} = req.params
    const {_id, password, google,correo, ...resto} = req.body //extraigo el viejo password y lo añado con resto.password en caso de que venga


    if(password){ // si viene el password tengo que volver a encriptarlo
        const salt = bcryptjs.genSaltSync()
        resto.password = bcryptjs.hashSync(password, salt)
    }

    const usuario = await Usuario.findByIdAndUpdate(id, resto, {new: true}) // le paso resto como valor a actualizar

    res.status(200).json({
        usuario
    })

}
```

- Tengo que verificar que el id que le paso en la url sea un id de mongo válido
- Al final de todos mis chequeos debo usar mi helper validarCampos para recoger los errores
- Creo un custom validator en db-validators.js

```
export const userExistsById = async(id)=>{
    const existeUsuario= await Usuario.findById(id)

    if(!existeUsuario) throw new Error("El id no existe")

}
```

- También hay que validar el rol (ya tengo la función en db-validators.js)
- user.routes.js

```
router.put('/:id',[
    check('id', 'No es un id válido').isMongoId(), //verifico que es un id de
Mongo válido
    check('id').custom(userExistsById),  //verifico que existe el usuario
    check('rol').custom(esRolValido),   //verifico que es un rol válido
    validarCampos                       //siempre al final recojo los errores con
validarCampos
], userController.usuariosPut)
```

GET - Obtener todos los usuarios de forma paginada

- Creo 15 usuarios en la DB
- Para hacer el get en el controlador de user.controller.js

```
const usuariosGet = async (req,res) =>{
   const usuarios = await Usuario.find()

   res.json({usuarios})
}
```

- Para hacer la paginación desestructuro los argumentos de la url y uso el método limit

```
const usuariosGet = async (req,res) =>{
   const {limite= 5} = req.query
    const usuarios = await Usuario.find()
        .limit(limite)

   res.json({usuarios})
}
```

- Si lo dejo así y no le paso el query limit en la url me muestra 5 elementos, pero si introduzco un numero revienta
    - Da error de failed to parse porque está esperando un número y cuando lo desestructuro del query viene como string
    - Debo castear el resultado a numero. Puedo usar Number(elemento) o un +

```
const usuariosGet = async (req,res) =>{
   const {limite= 5} = req.query
    const usuarios = await Usuario.find()
        .limit(+limite)   // casteo limite a número

   res.json({usuarios})
}
```

- Escribo en Thunder Client el GET http://localhost:8080/api/usuarios?limite=7 y me muestra los 7 primeros
- Voy a recibir otrto argumento que será desde e iniciará por defecto en 0
- Uso skip para decirle a partir de que usuario quiero mostrar

```
const usuariosGet = async (req,res) =>{
   const {limite= 5, desde = 0} = req.query
    const usuarios = await Usuario.find()
        .skip(desde)
        .limit(+limite)


   res.json({usuarios})
}
```

- Hay que hacer una validación para asegurarnos de que en los query venga un numero

---

Retornar numero total de registros en una colección

- Puedo usar countDocuments

```
const usuariosGet = async (req,res) =>{
   const {limite= 5, desde = 0} = req.query
    const usuarios = await Usuario.find()
        .skip(desde)
        .limit(+limite)

    const total = await Usuario.countDocuments()

   res.json({
    total,
    usuarios})
}
```

- El problema de esto es que voy a hacer un borrado lógico, donde cambiaré el estado de true a false cuando quiera borrar
- Mostrando así el total siempre mostraré los borrados también
- debo añadir la condición en el find y en el count

```
const usuariosGet = async (req,res) =>{
   const {limite= 5, desde = 0} = req.query

    const usuarios = await Usuario.find({estado: true}) //puedo guardarlo en una
constante const query = {estado:true}
        .skip(desde)
        .limit(+limite)
```

```
    const total = await Usuario.countDocuments({estado: true})


   res.json({
    total,
    usuarios})
}
```

- El problema es que el await es un código bloqueante, y si la primera petición tarda un segundo y la segunda otro segundo, la petición tardaría dos segundos. Se puede optimizar con Promise.all

```
const usuariosGet = async (req,res) =>{
   const {limite= 5, desde = 0} = req.query

    const respuesta = await Promise.all([  // va a ejecutar ambas de manera
simultanea y no va a continuar hasta que ambas resuelvan
        Usuario.countDocuments({estado: true}), // si una de las dos falla, todo
falla
        Usuario.find({estado: true})
        .skip(desde)
        .limit(+limite)
    ])

   res.json({respuesta})
}
```

- La respuesta así no luce igual. Está devolviendo un arreglo respuesta donde el primer valor es el total y el segundo un arreglo con los usuarios
- Puedo usar una desestructuración de arreglos (posicional)

```
const usuariosGet = async (req,res) =>{
   const {limite= 5, desde = 0} = req.query

    const [total, usuarios]= await Promise.all([  // va a ejecutar ambas de manera
simultanea y no va a continuar hasta que ambas resuelvan
        Usuario.countDocuments({estado: true}), // si una de las dos falla, todo
falla
        Usuario.find({estado: true})
        .skip(desde)
        .limit(+limite)
    ])

   res.json({
    total, usuarios
   })
}
```

Delete - Borrando un usuario de la base de datos

- Recibo el id desde la url con req.params
- Hago las validaciones del id (que sea válido de Mongo, y que el usuario exista)
- usuario.routes.js
-

```
router.delete('/:id', [
    check('id', 'No es un id válido').isMongoId(),
    check('id').custom(userExistsById),
    validarCampos
],
userController.usuariosDelete)
```

- user.controller.js

```
const usuariosDelete =  async (req,res) =>{
   const {id} = req.params
    const usuario = await Usuario.findByIdAndUpdate(id, {estado:false}) //si
quisiera borrarlo fisicamente usaría finsByIdAndDelete(id)

    res.json({
        msg: "Borrado ok!"
    })

}
```