Andrés Camilo Romero Ruiz

Camilo Carmona Valencia

**Informe de funcionalidades**

Se realizó el backend en nest.js, las pruebas fueron realizadas con jest, la base de datos fue desplegada en neon tech, y la aplicación fue desplegada en koyeb.

En este informe se explicará el acercamiento a los diferentes problemas del desarrollo backend. Se explicara nuestro acercamiento a la autorización mediante JWT, la persistencia de datos, y los endpoints creados para cada modulo.

**Autenticación**

La autenticación fue manejada mediante el uso de javascript web tokens (JWT), dond eun usuario se autentica mediante su correo y su contraseña. Para la verificación se uso bycript para comparar la contraseña ingresada con la contraseña ya encriptada en la base de datos de la siguiente manera:

        if(!user || !bcrypt.compareSync(password, user.password))

            throw new UnauthorizedException('Invalid credentials');

Si la autenticación es exitosa entonces se genera un token que será retornado al usuario para que este lo use para acceder a los métodos protegidos.

Dentro del token se encuentra el id del usuario como se ve a continuación:

        return {user\_id: user.id, email: user.email,

            token: this.jwtService.sign({user\_id: user.id})};

El id del usuario será usado para autenticar al usuario en los diferentes endpoints que no están abiertos, al igual que para verificar sus roles. Los roles se verifican de la siguiente manera:

const validRoles = this.reflector.get<string[]>('roles', context.getHandler());

    if(!validRoles || validRoles.length===0){

      return true;

    }

….

    for (const role of user.roles) {

      if(validRoles.includes(role)){

        return true;

      }

    }

Esto se encuentra dentro de una guard, que es usada como decorator para los metodos protegidos, así:

    @Auth()

    myInfo(@Request() req)

Si no está limitado por roles, y:

    @Auth(validRoles.admin)

    findAll(@Query() paginationDto: PaginationDto)

Si está limitado por roles.

Mediante los roles del usuario es determinado que puede hacer dentro de la aplicación, pero en situaciones donde un rol no es suficiente es necesario buscar al usuario directamente, este es el caso con la modificación de productos, dado que un usuario solo deberia ser capaz de modificar sus producto, no los productos de otro, con esto en mente asi se usa el token para la verificación:

    update(@Request() req, @Param('id', ParseUUIDPipe) id:string, @Body() body:UpdateProductDto){

        return this.productsService.update(id,body,req.user.id);

    }

Y dentro de ProductService:

        if (productUpdate.owner.id !== userId){

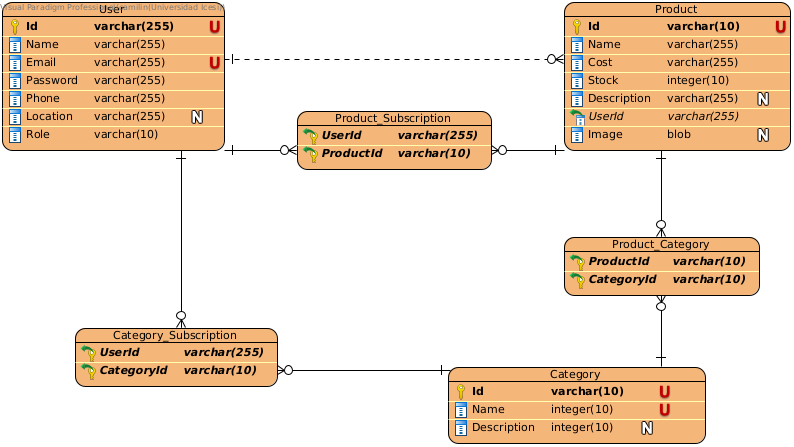
        throw new UnauthorizedException("Users may only update their products")

        }

**Persistencia de datos**

Como los datos que estamos trabajando son en su mayoria regulares, sin la necesidad de documentos o alta disponibilidad, se opto por una base de datos relacional, especificamente por postgres dado que es una plataforma de fuente abierta soportada por muchas plataformas de deployment.

En cuanto a nuestra implementación, tenemos tres tablas principales, la tabla de usuarios, la tabla de productos y la tabla de categorias, nuestro modelo de datos se ve de la siguiente manera:

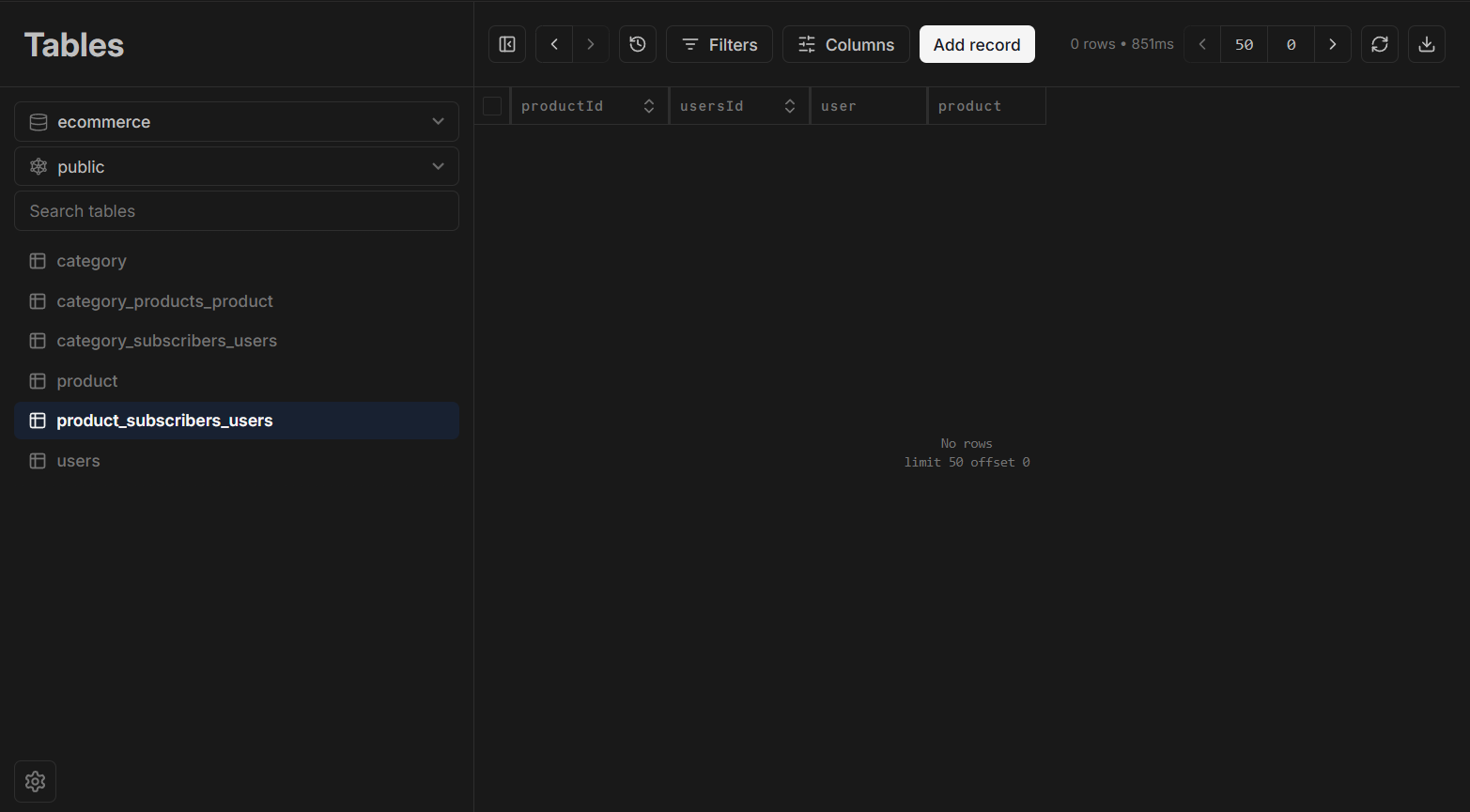
Donde las relaciones son de suscripción y del vendedor a sus productos, estas relaciones se implementaron como relaciones muchos a muchos dentro de las entidades:

    @ManyToMany(() => User, (user) => user.product\_subscriptions, {nullable:true, cascade:true})

    @JoinTable()

    subscribers: User[]

Estas relaciones generan una tabla pivote en la base de datos :



Por otro lado, se utilizo el servicio neon tech para desplegar la base de datos, y para conectarnos se utilizan las variables de entorno para guardar el enlace a la base de datos de manera segura:

 useFactory: (configService: ConfigService) => ({

        type: 'postgres',

        url: configService.get<string>('DATABASE\_URL'),

        autoLoadEntities: true,

        synchronize: true,

        ssl: {

          rejectUnauthorized: false,

        },

**Módulos**

**AuthModule**

El módulo de autenticación tiene toda la lógica relacionada al manejo de usuarios y sus interacciones directas con el sistema, sus endpoints son los siguientes:

* **createUser(/auth/register (POST)):**

Utilizado para registrar los usuarios, esta desprotegido, no tiene parámetros de url y la controladora toma estos parametros:

    @Post('register')

    createUser(@Body() creatUserDto: CreateUserDto)

El dto de creación de usuario tiene el el nombre, correo y contraseña del usuario, el telefono y ubicación son columnas exclusivas del usuario vendedor, para la creación del usuario se encripta la contraseña antes de guardar el usuario en la base de datos:

                user = this.userRepository.create({

                    password: bcrypt.hashSync(password, 10),

                    ...userData

                })

Esto tiene dos posibles retornos, o se retorna el usuario creado sin la contraseña con un código 201, o un codigo 400 si el usuario ya existia en la base de datos, determinado por el correo.

* **loginUser(/auth/login (POST)):**Utilizado para la autenticación de usuarios, está desprotegido, no tiene parámetros de url y la controladora toma estos parametros:

    @Post('login')

    loginUser(@Body() loginUserDto: LoginUserDto)

El dto de login tiene el correo y la contraseña del usuario, el correo será usado para encontrar al usuario correspondiente mientras que la contraseña será usada para validar el ingreso, a la hora de ingresar se generará un jwt como fue discutido anteriormente.

Como salida tiene una respuesta con codigo 201 si se ingreso con credenciales válidos, dentro de la cual se encontrará la información del usuario junto con el token generado. O se generará una respuesta con codigo 401 si los credenciales no son validos.

* **becomeSeller(/auth/seller (POST)):**

Utilizado para registrar a un usuario como vendedor, no tiene parámetros de url y la controladora toma estos parámetros:

    @Post('seller')

    @Auth()

    becomeSeller(@Request() req, @Body() sellerDto: SellerDto)

El dto de seller tiene el numero de telefono y la ubicación, estos datos son vitales para el manejo de productos dado que la plataforma no tiene dentro de su alcance el pago dentro de la aplicación, por lo tanto los vendedores tiene que resolver sus ventas por medios externos, y la ubicación es para facilitar que los posibles vendedors los encuentren.

Como salida tiene una respuesta con codigo 201 con los datos actualizados del vendedor como cuerpo.

* **myInfo(/auth/info (GET)):**

Utilizado para que un usuario pueda obtener su información, no tiene parámetros de url y la controladora tiene estos parámetros:

    @Get('info')

    @Auth()

    myInfo(@Request() req)

Como el propósito de este endpoint es recuperar la información del mismo usuario que esta autenticado, el id del usuario se encuentra dentro del token por lo que no es necesario enviarlo dentro de los parámetros, el id se recupera de esta forma:

        const userId = req.user.id;

Este endpoint tiene como retorno una respuesta con codigo 200 con la información encontrada del usuario.

* **findAll(/auth/users (GET)):**

Utilizado para obtener un listado con todos los usuarios, solo puede acceder un usuario con rol admin, sus parámetros de url son los de paginación (offset y limit) y la controladora tiene estos parámetros:

    @Get('users')

    @Auth(validRoles.admin)

    findAll(@Query() paginationDto: PaginationDto)

Los parámetros son offset y limit, con el propostio de paginar a los usuarios, dado el caso que no sean incluidos, por defecto offset tiene un valor de 0 y limit de 10.

Como retorno tiene una respuesta con codigo 200 que contiene la cantidad de usuarios especificada.

* **findByName(/auth/:name (GET)):**

Metodo usado para obtener a los usuarios que tengan un nombre similar al nombre enviado, tiene como parámetros de url tiene el nombre del usuario y la controladora toma estos parámetros:

    @Get(':name')

    @Auth(validRoles.admin)

    findByName(@Param('name') name:string, @Query() paginationDto: PaginationDto)

Como es posible que más de un usuario cumpla con la condición del nombre, entonces es necesario enviar parametros de paginación para limitar la cantidad de objetos a enviar, nuevamente los parametros de paginación por defecto son 0 para offset y 10 para limit.

Como retorno tiene una respuesta con codigo 200 con todos los usuarios con nombre similar al provisto.

* **update(/auth (PUT)):**

Metodo usado para que los usuarios puedan actualizar sus propios datos, no recibe parametros de url y la controladora toma los siguientes parametros:

    @Put()

    @Auth()

    update(@Request() req, @Body() updateUserDto: UpdateUserDto)

Al igual que en el endpoint seller, en este endpoint se obtiene el id del usuario mediante el jwt generado al momento del login, mientras que la información actualizada se encuentra en el cuerpo de la request, UpdateUserDto tiene los datos del usuario a excepción del id.

Como retorno tiene una respuesta con codigo 200 y los datos actualizados.

* **delete(/auth (DELETE)):**

Metodo usado para eliminar un usuario, solamente un administrador tiene el permiso para eliminar a un usuario, tiene como parámetro de url el id del usuario a borrar, y la controladora toma estos parametros:

    @Delete(':id')

    @Auth(validRoles.admin)

    delete(@Param('id') id: string)

Este metodo no elimina realmente al usuario de la base de datos, sino que actualiza el estado del usuario, de activo a inactivo.

Tiene como retorno una respuesta con codigo 200 con la información del usuario eliminado.

**CategoryModule**

El modulo de categorias tiene la logica del manejo de las categorias, siendo estas una forma de juntar los productos para una búsqueda más sencilla, sus endpoints son los siguientes:

* **create(/categories (POST)):**

Metodo usado para la creación de categorias, no toma parámetros por url, solamente los vendedores pueden crear categorias, y la controladora toma los siguientes parámetros:

  @Post()

  @Auth(validRoles.seller)

  create(@Body() createCategorydDto: CreateCategoryDto)

Una categoria tiene solamente dos componentes, un nombre y una descripción, estos datos son enviados mediante el cuerpo de la request, la categoría creada no es asignada a ningún producto hasta que sea añadida manualmente a un producto.

Como retorno tiene una respuesta con codigo 201 con la información de la categoría creada.

* **findAll(/categories (GET)):**

Metodo usado para obtener a todas las categorias creadas dentro del sistema, toma parametros de paginación por url y la controladora toma los siguientes parametros:

  @Get()

  @Auth()

  findAll(@Query() paginationDto: PaginationDto)

Tiene como retorno una respuesta con codigo 200 con la información de todas las categorias dentro de los limites de la paginación.

* **findOne(/categories/:term (GET))**

Método usado para obtener una sola categoría, ya sea mediante su identificador único, o mediante su nombre, esto es posible porque las categorias tienen un nombre único, y permite una url más limpia, sus endpoints son:

  @Get(':term')

  @Auth()

  findOne(@Param('term') term: string)

El término que llega como parámetro puede ser o el id o el nombre, se determina cual de las dos es usada de la siguiente manera:

if(isUUID(term))

Tiene como retorno una respuesta con codigo 200 y la información de la categoría encontrada, o una respuesta con codigo 400 si la categoría que se busca no existe.

* **update(/auth/:id (PATCH))**

Metodo usado para modificar uno de los componentes de una categoría, toma como parámetro de url el id de la categoría, y la controladora toma los siguientes parametros:

  @Patch(':id')

  @Auth(validRoles.admin)

  update(@Param('id',ParseUUIDPipe) id: string, @Body() updateCategoryDto: UpdateCategoryDto)

UpdateCategoryDto tiene los datos de la categoría a excepción del slug y del id dado que estos no se pueden cambiar de manera explícita.

Como retorno tiene una respuesta con codigo 200 con los datos actualizado si se encuentra la categoría, o una respuesta con codigo 404 si no se encuentra la categoría.

* **remove(/auth/:id (DELETE)):**

Metodo usado para eliminar una categoría, toma el id de la categoría como parámetro de url, y la controladora toma los siguientes parametros:

  @Delete(':id')

  @Auth(validRoles.admin)

  remove(@Param('id',ParseUUIDPipe) id: string)

Como retorno tiene una respuesta con codigo 200 y la información de la categoría eliminada.

* **subscribe(/categories/subscribe (POST)):**

Metodo usado para añadir el usuario autenticado a la lista de usuarios interesados en la categoría con el propósito de recibir actualizaciones de los productos dentro de la categoría, no tiene parámetros de url y estos son los parámetros que toma la controladora:

  @Post('subscribe')

  @Auth()

  subscribe(@Request() req, @Body() subscribeCategoryDto:SubscribeCategoryDto)

El usuario es determinado mediante el token, y SubscribeCategoryDto contiene el id de la categoria.

Como retorno tiene una respuesta con codigo 201 con la información de la categoría a la cual el usuario se suscribió.

**ProductModule**

El modulo de productos es el encargado del manejo de los productos, también es el encargado de notificar a los usuarios dado que las condiciones de notificación dependen de eventos de los productos, sus endpoints son los siguientes:

* **findProducts(/products (GET)):**

Metodo usado para obtener todos los productos disponibles aplicando un filtro, por url se envían parametros para filtrar por nombre, un limite inferior y superior de costo, un arreglo de categorias, y la posibilidad de ver solamente los productos con unidades, además de los parametros de paginación. Los parametros de la controladora son:

    @Get()

    async findProducts(@Query() filter: FilterProductDto)

Como retorno tiene una respuesta con codigo 200 junto con los productos que cumplen con los los filtros especificados.

* **subscribe(/products/subscribe POST)):**

Metodo usado para añadir el usuario autenticado a la lista de usuarios interesados en el producto con el propósito de recibir actualizaciones del estado de su inventario, no tiene parámetros de url y estos son los parámetros que toma la controladora:

    @Auth()

    @Post('subscribe')

    subscribe(@Request() req, @Body() subscribeProductDto:SubscribeProductDto)

El usuario es determinado mediante el token, y SubscribeProductDto contiene el id del producto.

Como retorno tiene una respuesta con codigo 201 con la información del producto al cual el usuario se suscribió.

* **create(/products (POST)):**

Metodo usado para crear un producto, solamente los vendedores pueden acceder a este endpoint, no tiene parámetros de url y la controladora toma como parámetro:

    @Post()

    @UsePipes(ValidationPipe)

    @Auth(validRoles.seller)

    async create(@Request() req, @Body() product:CreateProductDto)

Un producto tiene como parámetros de creación un nombre, una descripción, un costo, y un arreglo de categorias, todo esto está incluido dentro de CreateProductDto.

Cuando se crea un producto se hace un llamado a las apis de generación de correos y sms para enviar una notificación que indique al usuario que una categoría en la que está interesado tiene un producto nuevo, de esta manera:

            if ((Date.now()-user.lastNotified.getTime()) >= 10800000){

                this.mailService.sendEmail(user.email, "Te puede interesar", message)

                this.smsService.sendSms("573022852699", message)

Los usuarios solo pueden ser notificados cada 3 horas.

Como retorno tiene una respuesta con codigo 201 con la información del producto creado.

* **findById(/products/:id (GET)):**

Metodo usado para encontrar a los productos con base en su id, por url toma como parámetro el id del producto y la controladora toma como parámetros:

    @Get(':id')

    async findById(@Param('id', ParseUUIDPipe)id :string)

Como retorno tiene una respuesta con codigo 200 y los datos del producto encontrado.

* **delete(/products/:id (DELETE)):**

Metodo usado para eliminar productos, toma el id del producto por la url, solamente los vendedores pueden acceder a este endpoint y la controladora toma estos parametros:

    @Delete(':id')

    @Auth(validRoles.seller)

    async delete(@Param('id', ParseUUIDPipe)id :string)

Como retorno tiene una respuesta con codigo 200 y los datos del producto borrado.

* **update(/products/:id (PATCH)):**

Metodo usado para modificar datos específicos de un producto, solamente los vendedores pueden acceder a este endpoint, toma como parámetro de url el id del producto, y la controladora toma estos parametros:

    @Patch(':id')

    @Auth(validRoles.seller)

    update(@Request() req, @Param('id', ParseUUIDPipe) id:string, @Body() body:UpdateProductDto)

Se usa el jwt para verificar que el usuario que está modificando sea el dueño del producto, dentro de UpdateProductDto son todas las características del producto excepto el id y el dueño.

Si se actualiza el estado de “inStock” a verdadero, se envia una notificación a los usuarios suscritos al producto, para avisarles que el producto esta de vuelta en stock.

Como retorno tiene una respuesta con codigo 200 y los datos actualizados del producto.