Pregunta:

¿Podrías detallarme el punto 4 de la pregunta anterior de forma más extendida?

Respuesta:

¡Absolutamente! Detallar el punto 4 es fundamental, ya que la autenticación es el pilar de la seguridad y la gestión de sesiones en tu aplicación. Vamos a desglosar en profundidad cómo construir un flujo de autenticación robusto y seguro, cubriendo tanto el backend (la lógica) como el frontend (el formulario).

Este proceso se basa en el estándar de la industria: **JSON Web Tokens (JWT)**.

**El Flujo General será:**

1. El usuario introduce su nombre y contraseña en el formulario de login del frontend.
2. El frontend envía estas credenciales al backend.
3. El backend verifica las credenciales contra la **nueva base de datos MariaDB**.
4. Si son correctas, el backend genera un token JWT (una cadena de texto codificada y firmada digitalmente) y se lo devuelve al frontend.
5. El frontend almacena este token de forma segura y lo adjuntará en las cabeceras de todas las futuras peticiones a endpoints protegidos.
6. El backend verificará la validez de este token en cada petición para saber quién es el usuario y si tiene permiso para acceder.

### **Parte 1: Backend (NestJS) - Creando el Cerebro de la Autenticación**

Aquí construiremos toda la lógica para validar usuarios y emitir tokens.

#### **Paso 4.1: Instalación de Dependencias**

Navega a la carpeta de tu API (mi-api-gestion) y ejecuta:

Bash

npm install @nestjs/passport passport @nestjs/jwt passport-jwt passport-local  
npm install --save-dev @types/passport-jwt @types/passport-local

* @nestjs/passport y passport: El núcleo para manejar estrategias de autenticación.
* @nestjs/jwt: Para crear y verificar los JSON Web Tokens.
* passport-jwt y passport-local: Las "estrategias" específicas que usaremos. La local es para el login con usuario/contraseña y la jwt es para proteger endpoints con un token.

#### **Paso 4.2: Creación y Configuración del AuthModule**

1. **Genera el módulo:**  
   Bash  
   nest g module auth
2. **Configura el módulo** para que conozca sobre los usuarios, Passport y JWT.  
   Primero, añade tu secreto JWT en el fichero .env. **Debe ser una cadena larga y aleatoria.**  
   **mi-api-gestion/.env** (añadir estas líneas)  
   Fragmento de código  
   # ... otras variables  
   JWT\_SECRET=ESTE\_ES\_UN\_SECRETO\_MUY\_LARGO\_Y\_SEGURO\_CAMBIAME  
   JWT\_EXPIRATION=3600s # 1 hora  
     
   Ahora, configura mi-api-gestion/src/auth/auth.module.ts:  
   TypeScript  
   import { Module } from '@nestjs/common';  
   import { AuthService } from './auth.service';  
   import { AuthController } from './auth.controller';  
   import { PassportModule } from '@nestjs/passport';  
   import { JwtModule } from '@nestjs/jwt';  
   import { ConfigModule, ConfigService } from '@nestjs/config';  
   import { JwtStrategy } from './strategies/jwt.strategy';  
   import { LocalStrategy } from './strategies/local.strategy';  
   import { UsersModule } from '../users/users.module'; // Importamos el módulo de usuarios  
     
   @Module({  
    imports: [  
    UsersModule, // Para poder usar UsersService  
    PassportModule,  
    JwtModule.registerAsync({  
    imports: [ConfigModule],  
    inject: [ConfigService],  
    useFactory: (configService: ConfigService) => ({  
    secret: configService.get<string>('JWT\_SECRET'),  
    signOptions: { expiresIn: configService.get<string>('JWT\_EXPIRATION') },  
    }),  
    }),  
    ],  
    controllers: [AuthController],  
    providers: [AuthService, LocalStrategy, JwtStrategy], // Registramos las estrategias  
   })  
   export class AuthModule {}

#### **Paso 4.3: Implementación de las Estrategias de Passport**

Crea una carpeta strategies dentro de src/auth.

1. **Estrategia Local (para el login):** Verifica que el usuario y la contraseña son correctos.  
   **mi-api-gestion/src/auth/strategies/local.strategy.ts**  
   TypeScript  
   import { Strategy } from 'passport-local';  
   import { PassportStrategy } from '@nestjs/passport';  
   import { Injectable, UnauthorizedException } from '@nestjs/common';  
   import { AuthService } from '../auth.service';  
     
   @Injectable()  
   export class LocalStrategy extends PassportStrategy(Strategy) {  
    constructor(private authService: AuthService) {  
    // Por defecto, passport-local espera 'username' y 'password'  
    super();  
    }  
     
    async validate(username: string, password: string): Promise<any> {  
    const user = await this.authService.validateUser(username, password);  
    if (!user) {  
    throw new UnauthorizedException('Credenciales incorrectas');  
    }  
    return user;  
    }  
   }
2. **Estrategia JWT (para proteger rutas):** Verifica el token que viene en las peticiones.  
   **mi-api-gestion/src/auth/strategies/jwt.strategy.ts**  
   TypeScript  
   import { ExtractJwt, Strategy } from 'passport-jwt';  
   import { PassportStrategy } from '@nestjs/passport';  
   import { Injectable } from '@nestjs/common';  
   import { ConfigService } from '@nestjs/config';  
     
   @Injectable()  
   export class JwtStrategy extends PassportStrategy(Strategy) {  
    constructor(private configService: ConfigService) {  
    super({  
    jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),  
    ignoreExpiration: false,  
    secretOrKey: configService.get<string>('JWT\_SECRET'),  
    });  
    }  
     
    // Passport verifica el token y si es válido, llama a este método con el payload decodificado  
    async validate(payload: any) {  
    // Lo que retornemos aquí se adjuntará a request.user  
    return { userId: payload.sub, username: payload.username, permisos: payload.permisos };  
    }  
   }

#### **Paso 4.4: Implementación del AuthService y AuthController**

1. **Genera el servicio y el controlador:**  
   Bash  
   nest g service auth --no-spec  
   nest g controller auth --no-spec
2. **Completa el AuthService** con la lógica de validación y creación de tokens.  
   **mi-api-gestion/src/auth/auth.service.ts**  
   TypeScript  
   import { Injectable } from '@nestjs/common';  
   import { JwtService } from '@nestjs/jwt';  
   import { UsersService } from '../users/users.service';  
   import { User } from '../database/entities/user.entity';  
     
   @Injectable()  
   export class AuthService {  
    constructor(  
    private usersService: UsersService,  
    private jwtService: JwtService,  
    ) {}  
     
    async validateUser(username: string, pass: string): Promise<Omit<User, 'password'>> {  
    const user = await this.usersService.findOneByUsername(username);  
    if (user && await user.validatePassword(pass)) {  
    const { password, ...result } = user;  
    return result;  
    }  
    return null;  
    }  
     
    async login(user: any) {  
    const payload = {   
    username: user.username,   
    sub: user.id, // 'sub' es el nombre estándar para el subject (ID del usuario)  
    permisos: user.permisos.map(p => p.tipo) // Incluimos los permisos en el token  
    };  
    return {  
    access\_token: this.jwtService.sign(payload),  
    };  
    }  
   }  
     
   *(Nota: Necesitarás crear el método findOneByUsername en tu UsersService que busca un usuario en la nueva BD)*.
3. **Completa el AuthController** para exponer el endpoint de login.  
   **mi-api-gestion/src/auth/auth.controller.ts**  
   TypeScript  
   import { Controller, Post, UseGuards, Request, Get } from '@nestjs/common';  
   import { AuthGuard } from '@nestjs/passport';  
   import { AuthService } from './auth.service';  
     
   @Controller('auth')  
   export class AuthController {  
    constructor(private authService: AuthService) {}  
     
    // La petición a POST /auth/login primero pasa por el Guard 'local'  
    @UseGuards(AuthGuard('local'))  
    @Post('login')  
    async login(@Request() req) {  
    // Si las credenciales son válidas, req.user es poblado por LocalStrategy  
    return this.authService.login(req.user);  
    }  
     
    // Ejemplo de ruta protegida  
    @UseGuards(AuthGuard('jwt'))  
    @Get('profile')  
    getProfile(@Request() req) {  
    // req.user es poblado por JwtStrategy con el payload del token  
    return req.user;  
    }  
   }

¡El backend para la autenticación está listo!

### **Parte 2: Frontend (Next.js) - Construyendo la Puerta de Entrada**

Ahora crearemos el formulario y la lógica para gestionar el estado de autenticación en la aplicación de React.

#### **Paso 4.5: Instalación de Dependencias del Frontend**

Navega a la carpeta de tu frontend (mi-app-gestion) y ejecuta:

Bash

npm install react-hook-form @hookform/resolvers yup jwt-decode

* react-hook-form: Para gestionar formularios de forma eficiente.
* @hookform/resolvers y yup: Para una validación de esquemas sencilla y potente.
* jwt-decode: Una pequeña utilidad para decodificar el payload del token JWT en el cliente.

#### **Paso 4.6: Creación del Contexto de Autenticación (AuthContext)**

Este es el "cerebro" del estado de autenticación en el frontend.

1. Crea una carpeta context dentro de src.
2. Crea el fichero mi-app-gestion/src/context/AuthContext.tsx:  
   TypeScript  
   'use client';  
   import { createContext, useContext, useState, useEffect, ReactNode } from 'react';  
   import { jwtDecode } from 'jwt-decode';  
   import Cookies from 'js-cookie';  
   import api from '@/lib/api';  
     
   // Define la forma del usuario decodificado del token  
   interface AuthUser {  
    userId: number;  
    username: string;  
    permisos: string[];  
   }  
     
   // Define lo que el contexto proveerá  
   interface AuthContextType {  
    isAuthenticated: boolean;  
    user: AuthUser | null;  
    login: (token: string) => void;  
    logout: () => void;  
    isLoading: boolean;  
   }  
     
   const AuthContext = createContext<AuthContextType | undefined>(undefined);  
     
   export const AuthProvider = ({ children }: { children: ReactNode }) => {  
    const [user, setUser] = useState<AuthUser | null>(null);  
    const [isLoading, setIsLoading] = useState(true);  
     
    useEffect(() => {  
    // Al cargar la app, intenta leer el token de las cookies  
    const token = Cookies.get('token');  
    if (token) {  
    try {  
    const decodedUser: AuthUser = jwtDecode(token);  
    setUser(decodedUser);  
    // Configura axios para que siempre envíe el token  
    api.defaults.headers.common['Authorization'] = `Bearer ${token}`;  
    } catch (error) {  
    console.error("Token inválido", error);  
    logout();  
    }  
    }  
    setIsLoading(false);  
    }, []);  
     
    const login = (token: string) => {  
    const decodedUser: AuthUser = jwtDecode(token);  
    setUser(decodedUser);  
    // Almacena el token en una cookie segura  
    Cookies.set('token', token, { expires: 1, secure: process.env.NODE\_ENV === 'production' });  
    api.defaults.headers.common['Authorization'] = `Bearer ${token}`;  
    };  
     
    const logout = () => {  
    setUser(null);  
    Cookies.remove('token');  
    delete api.defaults.headers.common['Authorization'];  
    };  
     
    const value = {  
    isAuthenticated: !!user,  
    user,  
    login,  
    logout,  
    isLoading  
    };  
     
    return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>;  
   };  
     
   // Hook personalizado para usar el contexto fácilmente  
   export const useAuth = () => {  
    const context = useContext(AuthContext);  
    if (context === undefined) {  
    throw new Error('useAuth debe ser usado dentro de un AuthProvider');  
    }  
    return context;  
   };  
     
   *Nota: He usado js-cookie para una gestión de cookies más sencilla. Instálalo: npm install js-cookie @types/js-cookie.*

#### **Paso 4.7: Integrar AuthProvider y construir la Página de Login**

1. **Envuelve la aplicación con el AuthProvider** en mi-app-gestion/src/app/layout.tsx:  
   TypeScript  
   // ... imports  
   import { AuthProvider } from '@/context/AuthContext';  
     
   export default function RootLayout({ children }: { children: React.ReactNode }) {  
    return (  
    <html lang="es">  
    <body>  
    <AuthProvider>  
    <ThemeRegistry>{children}</ThemeRegistry>  
    </AuthProvider>  
    </body>  
    </html>  
    );  
   }
2. **Crea la página de login** en mi-app-gestion/src/app/login/page.tsx:  
   TypeScript  
   'use client';  
   import { useState } from 'react';  
   import { useForm } from 'react-hook-form';  
   import { Button, TextField, Container, Typography, Box, Alert, CircularProgress } from '@mui/material';  
   import { useAuth } from '@/context/AuthContext';  
   import { useRouter } from 'next/navigation';  
   import api from '@/lib/api';  
     
   export default function LoginPage() {  
    const { register, handleSubmit } = useForm();  
    const { login } = useAuth();  
    const router = useRouter();  
    const [error, setError] = useState<string | null>(null);  
    const [isLoading, setIsLoading] = useState(false);  
     
    const onSubmit = async (data) => {  
    setError(null);  
    setIsLoading(true);  
    try {  
    const response = await api.post('/auth/login', data);  
    const { access\_token } = response.data;  
    login(access\_token);  
    router.push('/dashboard'); // Redirige al dashboard después del login  
    } catch (err) {  
    setError('Usuario o contraseña incorrectos. Por favor, inténtelo de nuevo.');  
    console.error('Fallo en el login', err);  
    } finally {  
    setIsLoading(false);  
    }  
    };  
     
    return (  
    <Container component="main" maxWidth="xs">  
    <Box  
    sx={{  
    marginTop: 8,  
    display: 'flex',  
    flexDirection: 'column',  
    alignItems: 'center',  
    }}  
    >  
    <Typography component="h1" variant="h5">  
    Iniciar Sesión  
    </Typography>  
    <Box component="form" onSubmit={handleSubmit(onSubmit)} sx={{ mt: 1 }}>  
    <TextField  
    margin="normal"  
    required  
    fullWidth  
    id="username"  
    label="Nombre de Usuario"  
    autoFocus  
    {...register('username')}  
    />  
    <TextField  
    margin="normal"  
    required  
    fullWidth  
    label="Contraseña"  
    type="password"  
    id="password"  
    {...register('password')}  
    />  
    {error && <Alert severity="error" sx={{ width: '100%', mt: 2 }}>{error}</Alert>}  
    <Button  
    type="submit"  
    fullWidth  
    variant="contained"  
    sx={{ mt: 3, mb: 2 }}  
    disabled={isLoading}  
    >  
    {isLoading ? <CircularProgress size={24} /> : 'Acceder'}  
    </Button>  
    </Box>  
    </Box>  
    </Container>  
    );  
   }

Con estos pasos, has construido un sistema de autenticación completo y seguro. El usuario puede ahora iniciar sesión, y tu aplicación frontend es consciente de su estado de autenticación, lista para los siguientes pasos como proteger rutas y mostrar contenido basado en permisos.

**Fuentes**

1. <https://github.com/SporosDAO/sweat-token> sujeto a licencia (Apache - 2.0)