Tutorial Completo: Autenticación y Autorización por Roles en NestJS

Índice

1. Conceptos Fundamentales
2. Preparación del Proyecto
3. Configuración de la Entidad User
4. Creación del Módulo Auth
5. Implementación de JWT
6. Estrategia JWT
7. Decoradores Personalizados
8. Guards Personalizados
9. Protección de Endpoints
10. Testing y Validación

---

1. Conceptos Fundamentales

¿Qué es JWT?

JSON Web Token es un estándar para transmitir información de forma segura. En nuestro caso, solo contendrá el **ID del usuario** para mantener el payload mínimo y seguro.

¿Entidad vs DTO?

- **Entidad (User)**: Representa la tabla en la base de datos
- **DTO (Data Transfer Object)**: Define qué datos acepta/envía cada endpoint

Ejemplo práctico:

- `User` entity: Tiene password, isActive, createdAt, etc.
- `LoginDto`: Solo acepta email y password
- `RegisterDto`: Acepta email, password, fullName (sin campos internos)

¿Por qué separar Auth de Users?

- **Auth**: Maneja login, registro, tokens
- **Users**: Maneja CRUD de usuarios (admin)

---

2. Preparación del Proyecto

2.1 Instalación de Dependencias

```
# Dependencias principales para JWT y autenticación
npm install @nestjs/jwt @nestjs/passport passport passport-jwt
```

```
# Para encriptar contraseñas
npm install bcryptjs

# Dependencias de desarrollo (tipados)
npm install --save-dev @types/passport-jwt @types/bcryptjs
```

2.2 Variables de Entorno

Agregar al archivo `.env`:

```
# JWT Configuration - Solo necesitamos el secret y tiempo de expiración
JWT_SECRET=tu_clave_secreta_muy_segura_aqui_minimo_32_caracteres
JWT_EXPIRED_TIME=24h
```

¿Por qué estas variables?

- `JWT_SECRET`: Clave para firmar tokens (debe ser secreta)
- `JWT_EXPIRED_TIME`: Tiempo de vida del token

---

3. Configuración de la Entidad User

3.1 Crear Enum de Permisos

Comando:

```
# Crear el enum manualmente en la carpeta users/enums
mkdir src/users/enums
```

Archivo: `src/users/enums/permissions.enum.ts`

```
export enum PermissionsTypes {
  ADMIN = 'admin',
  USER = 'user',
  MODERATOR = 'moderator',
  SUPER_ADMIN = 'super_admin'
}
```

¿Por qué un enum?

- Evita errores de tipeo
- Autocompletado en el IDE
- Fácil refactoring

3.2 Crear la Entidad User

Archivo: `src/users/entities/user.entity.ts`

```
import { Entity, PrimaryGeneratedColumn, Column, BeforeInsert, BeforeUpdate } from
'typeorm';

@Entity('users')
export class User {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ unique: true, type: 'text' })
  email: string;

  // select: false = No se incluye en consultas automáticas (por seguridad)
  @Column('text', { select: false })
  password: string;

  @Column('text')
  fullName: string;

  @Column({ default: true, type: 'bool' })
  isActive: boolean;

  // array de strings para múltiples permisos
  @Column({ type: 'text', array: true, default: ['user'] })
  permissions: string[];

  @Column({ type: 'timestamp', default: () => 'CURRENT_TIMESTAMP' })
  createdAt: Date;

  @BeforeInsert()
  checkFieldsBeforeInsert() {
    this.email = this.email.toLowerCase().trim();
  }

  @BeforeUpdate()
  checkFieldsBeforeUpdate() {
    this.checkFieldsBeforeInsert();
  }
}
```

Explicación de decoradores:

- `@Entity('users')`: Nombre de la tabla en DB
- `@PrimaryGeneratedColumn('uuid')`: ID único automático
- `@Column({ unique: true })`: Email único en DB
- `{ select: false }`: Password no se incluye en consultas por defecto
- `{ array: true }`: PostgreSQL array para múltiples permisos

4. Creación del Módulo Auth

4.1 Generar el Módulo

```
# Generar módulo completo con controlador y servicio
nest g res auth --no-spec
```

¿Qué crea este comando?

- `auth.module.ts`
- `auth.controller.ts`
- `auth.service.ts`
- Carpeta `dto/`
- Carpeta `entities/`

4.2 Crear Carpetas Adicionales

```
# Crear carpetas para organizar el código
mkdir src/auth/interfaces
mkdir src/auth/strategies
mkdir src/auth/decorators
mkdir src/auth/guards
```

4.3 Crear DTOs

¿Por qué DTOs separados?

- **Seguridad**: Solo exponemos campos necesarios
- **Validación**: Cada endpoint valida solo lo que necesita
- **Flexibilidad**: Diferentes endpoints, diferentes datos

Comando para crear DTOs:

```
# Los DTOs se crean manualmente en auth/dto/
```

Archivo: `src/auth/dto/login.dto.ts`

```
import { IsEmail, IsString, MinLength } from 'class-validator';

export class LoginDto {
  @IsEmail({}, { message: 'Debe ser un email válido' })
  email: string;

  @IsString({ message: 'La contraseña debe ser texto' })
  @MinLength(6, { message: 'La contraseña debe tener mínimo 6 caracteres' })
```

```
  password: string;
}
```

Archivo: `src/auth/dto/register.dto.ts`

```
import { IsEmail, IsString, MinLength, IsOptional, IsArray } from 'class-
validator';
import { PermissionsTypes } from '../../users/enums/permissions.enum';

export class RegisterDto {
  @IsEmail({}, { message: 'Debe ser un email válido' })
  email: string;

  @IsString({ message: 'La contraseña debe ser texto' })
  @MinLength(6, { message: 'La contraseña debe tener mínimo 6 caracteres' })
  password: string;

  @IsString({ message: 'El nombre completo es requerido' })
  fullName: string;

  // Opcional: Si no se envía, se asigna 'user' por defecto
  @IsOptional()
  @IsArray({ message: 'Los permisos deben ser un array' })
  permissions?: PermissionsTypes[];
}
```

¿Por qué estas validaciones?

- `@IsEmail()`: Valida formato de email
- `@MinLength(6)`: Contraseña mínima segura
- `@IsOptional()`: Campo no obligatorio
- Mensajes personalizados para mejor UX

4.4 Crear Interface JWT

Archivo: `src/auth/interfaces/jwt-payload.interface.ts`

```
// SOLO el ID del usuario en el payload (como solicitaste)
export interface JwtPayload {
  id: string;
}
```

¿Por qué solo el ID?

- **Seguridad**: Menos información en el token
- **Performance**: Payload más pequeño
- **Flexibilidad**: Los datos del usuario pueden cambiar

- **Escalabilidad**: Token no crece con más campos

---

5. Implementación de JWT

5.1 Configurar AuthModule

Archivo: `src/auth/auth.module.ts`

```
import { Module } from '@nestjs/common';
import { ConfigModule, ConfigService } from '@nestjs/config';
import { TypeOrmModule } from '@nestjs/typeorm';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';

import { AuthController } from './auth.controller';
import { AuthService } from './auth.service';
import { User } from '../users/entities/user.entity';
import { JwtStrategy } from './strategies/jwt.strategy';

@Module({
  controllers: [AuthController],
  providers: [AuthService, JwtStrategy],
  imports: [
    ConfigModule,
    // Importar la entidad User para usar el repositorio
    TypeOrmModule.forFeature([User]),
    // Configurar Passport con JWT como estrategia por defecto
    PassportModule.register({ defaultStrategy: 'jwt' }),
    // Configurar JWT de forma asíncrona para usar variables de entorno
    JwtModule.registerAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: (configService: ConfigService) => ({
        secret: configService.get('JWT_SECRET'),
        signOptions: {
          expiresIn: configService.get('JWT_EXPIRED_TIME'),
        },
      }),
    }),
  ],
  // Exportar para usar en otros módulos
  exports: [JwtStrategy, PassportModule, JwtModule, AuthService],
})
export class AuthModule {}
```

Explicación paso a paso:

1. `TypeOrmModule.forFeature([User])`: Permite inyectar el repositorio de User
2. `PassportModule.register()`: Configura Passport con JWT
3. `JwtModule.registerAsync()`: Configuración asíncrona para leer .env

4. `exports`: Permite usar estos servicios en otros módulos

5.2 Implementar AuthService

Archivo: `src/auth/auth.service.ts`

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { JwtService } from '@nestjs/jwt';
import { Repository } from 'typeorm';
import * as bcrypt from 'bcryptjs';

import { User } from '../users/entities/user.entity';
import { LoginDto } from './dto/login.dto';
import { RegisterDto } from './dto/register.dto';
import { JwtPayload } from './interfaces/jwt-payload.interface';
import { PermissionsTypes } from '../users/enums/permissions.enum';

@Injectable()
export class AuthService {
  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>,
    private readonly jwtService: JwtService,
  ) {}

  async register(registerDto: RegisterDto) {
    const { password, permissions, ...userData } = registerDto;

    // Crear usuario con contraseña encriptada
    const user = this.userRepository.create({
      ...userData,
      password: await bcrypt.hash(password, 10),
      permissions: permissions?.length ? permissions : [PermissionsTypes.USER],
    });

    await this.userRepository.save(user);

    // Retornar datos seguros + token
    return {
      id: user.id,
      email: user.email,
      fullName: user.fullName,
      permissions: user.permissions,
      token: this.getJwtToken({ id: user.id }), // Solo ID en el payload
    };
  }

  async login(loginDto: LoginDto) {
    const { email, password } = loginDto;

    // Buscar usuario incluyendo password (select: false por defecto)
```

```
    const user = await this.userRepository.findOne({
      where: { email: email.toLowerCase().trim() },
      select: ['id', 'email', 'password', 'fullName', 'permissions', 'isActive'],
    });

    if (!user) {
      throw new UnauthorizedException('Credenciales no válidas');
    }

    if (!user.isActive) {
      throw new UnauthorizedException('Usuario inactivo, contacte al
administrador');
    }

    // Verificar contraseña
    if (!bcrypt.compareSync(password, user.password)) {
      throw new UnauthorizedException('Credenciales no válidas');
    }

    // Retornar datos seguros + token
    return {
      id: user.id,
      email: user.email,
      fullName: user.fullName,
      permissions: user.permissions,
      token: this.getJwtToken({ id: user.id }), // Solo ID en el payload
    };
  }

  async checkAuthStatus(user: User) {
    // Renovar token con usuario actual
    return {
      id: user.id,
      email: user.email,
      fullName: user.fullName,
      permissions: user.permissions,
      token: this.getJwtToken({ id: user.id }), // Solo ID en el payload
    };
  }

  // Método privado para generar JWT
  private getJwtToken(payload: JwtPayload) {
    return this.jwtService.sign(payload);
  }
}
```

Puntos clave:

1. **Solo ID en JWT**: `{ id: user.id }` - Como solicitaste
2. **Contraseña encriptada**: `bcrypt.hash(password, 10)`
3. **Validación de usuario activo**: Previene login de usuarios deshabilitados
4. **Select explícito**: Para incluir password en login

5. **Datos seguros**: Nunca retornamos password al cliente

---

6. Estrategia JWT

6.1 Crear JwtStrategy

Comando:

```
# Crear archivo manualmente
```

Archivo: src/auth/strategies/jwt.strategy.ts

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { PassportStrategy } from '@nestjs/passport';
import { InjectRepository } from '@nestjs/typeorm';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { Repository } from 'typeorm';

import { User } from '../../users/entities/user.entity';
import { JwtPayload } from '../interfaces/jwt-payload.interface';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>,
    configService: ConfigService,
  ) {
    super({
      // Extraer token del header Authorization: Bearer <token>
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false, // Validar expiración
      secretOrKey: configService.get('JWT_SECRET'),
    });
  }

  // Este método se ejecuta automáticamente cuando se valida el token
  async validate(payload: JwtPayload): Promise<User> {
    const { id } = payload; // Solo tenemos el ID

    // Buscar usuario completo en DB
    const user = await this.userRepository.findOne({
      where: { id },
    });

    if (!user) {
      throw new UnauthorizedException('Token no válido');
    }
```

```
    if (!user.isActive) {
      throw new UnauthorizedException('Usuario inactivo, contacte al
administrador');
    }

    // El usuario se agrega automáticamente a req.user
    return user;
  }
}
```

¿Cómo funciona?

1. Cliente envía: `Authorization: Bearer <token>`
2. Passport extrae y valida el token
3. Si es válido, ejecuta `validate()` con el payload
4. `validate()` busca el usuario en DB con el ID
5. Si todo está bien, el usuario se agrega a `req.user`

Ventajas de solo usar ID:

- Si cambian los permisos del usuario, se reflejan inmediatamente
- Token más pequeño y rápido
- Información siempre actualizada desde DB

---

7. Decoradores Personalizados

7.1 Decorador para Obtener Usuario

Comando:

```
nest g d auth/decorators/get-user --no-spec
```

Archivo: `src/auth/decorators/get-user.decorator.ts`

```
import { createParamDecorator, ExecutionContext, InternalServerErrorException }
from '@nestjs/common';

export const GetUser = createParamDecorator(
  (data: string, ctx: ExecutionContext) => {
    const req = ctx.switchToHttp().getRequest();
    const user = req.user;

    if (!user) {
      throw new InternalServerErrorException('Usuario no encontrado (request)');
    }

    // Si se especifica un campo, retornar solo ese campo
```

```
    return data ? user[data] : user;
  },
);
```

Uso del decorador:

```
// Obtener usuario completo
getProfile(@GetUser() user: User) { ... }

// Obtener solo el ID
getProfile(@GetUser('id') userId: string) { ... }

// Obtener solo el email
getProfile(@GetUser('email') email: string) { ... }
```

7.2 Decorador para Protección por Permisos

Comando:

```
nest g d auth/decorators/role-protected --no-spec
```

Archivo: `src/auth/decorators/role-protected.decorator.ts`

```
import { SetMetadata } from '@nestjs/common';
import { PermissionsTypes } from '../../users/enums/permissions.enum';

// Clave para almacenar metadata de permisos (consistente con nuestro modelo)
export const META_PERMISSIONS = 'permissions';

// Decorador que guarda los permisos requeridos en metadata
export const RoleProtected = (...args: PermissionsTypes[]) => {
  return SetMetadata(META_PERMISSIONS, args);
};
```

¿Por qué META_PERMISSIONS?

- Consistente con el campo `permissions` en la entidad User
- Más claro semánticamente (permisos vs roles)
- Evita confusión en el código

---

8. Guards Personalizados

8.1 Crear UserRoleGuard

Comando:

```
nest g gu auth/guards/user-role --no-spec
```

Archivo: src/auth/guards/user-role.guard.ts

```
import { Injectable, CanActivate, ExecutionContext, BadRequestException,
ForbiddenException } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Observable } from 'rxjs';

import { META_PERMISSIONS } from '../decorators/role-protected.decorator';

@Injectable()
export class UserRoleGuard implements CanActivate {
  constructor(private readonly reflector: Reflector) {}

  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    // Obtener permisos requeridos del decorador @RoleProtected()
    const validPermissions: string[] = this.reflector.get(META_PERMISSIONS,
context.getHandler());

    // Si no hay permisos definidos, permitir acceso
    if (!validPermissions || validPermissions.length === 0) {
      return true;
    }

    // Obtener usuario de la request (viene del JwtStrategy)
    const req = context.switchToHttp().getRequest();
    const user = req.user;

    if (!user) {
      throw new BadRequestException('Usuario no encontrado en la request');
    }

    // Verificar si el usuario tiene alguno de los permisos requeridos
    for (const permission of user.permissions) {
      if (validPermissions.includes(permission)) {
        return true; // Usuario autorizado
      }
    }

    // Usuario no tiene permisos suficientes
    throw new ForbiddenException(
      `El usuario ${user.fullName} necesita uno de estos permisos:
[${validPermissions.join(', ')}]`
    );
  }
}
```

¿Cómo funciona el Guard?

1. Se ejecuta después del JwtStrategy
2. Lee los permisos requeridos del decorador (que ya debe existir)
3. Compara con los permisos del usuario
4. Permite o deniega el acceso

¿Por qué crear el decorador ANTES del guard?

- El guard importa `META_PERMISSIONS` del decorador
- Sin el decorador, el guard no puede compilar
- Es una dependencia directa que debe existir primero

```
### 8.2 Decorador Compuesto Auth

**Comando:**
```bash
nest g d auth/decorators/auth --no-spec
```

Archivo: `src/auth/decorators/auth.decorator.ts`

```
import { applyDecorators, UseGuards } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

import { UserRoleGuard } from '../guards/user-role.guard';
import { RoleProtected } from './role-protected.decorator';
import { PermissionsTypes } from '../../users/enums/permissions.enum';

// Decorador que combina autenticación y autorización
export function Auth(...permissions: PermissionsTypes[]) {
  return applyDecorators(
    RoleProtected(...permissions), // Definir permisos requeridos
    UseGuards(AuthGuard(), UserRoleGuard), // Aplicar guards
  );
}
```

¿Por qué un decorador compuesto?

- **Simplicidad**: Un solo decorador en lugar de tres
- **Consistencia**: Siempre se aplican los guards correctos
- **Mantenibilidad**: Cambios centralizados

Uso:

```
// Solo autenticación (cualquier usuario logueado)
@Auth()
getProfile() { ... }
```
```

```
// Solo administradores
@Auth(PermissionsTypes.ADMIN)
getAllUsers() { ... }

// Administradores o moderadores
@Auth(PermissionsTypes.ADMIN, PermissionsTypes.MODERATOR)
moderateContent() { ... }
```

---

9. Protección de Endpoints

9.1 Implementar AuthController

Archivo: `src/auth/auth.controller.ts`

```
import { Controller, Post, Body, Get } from '@nestjs/common';

import { AuthService } from './auth.service';
import { LoginDto } from './dto/login.dto';
import { RegisterDto } from './dto/register.dto';
import { Auth } from './decorators/auth.decorator';
import { GetUser } from './decorators/get-user.decorator';
import { User } from '../users/entities/user.entity';

@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  // Endpoint público para registro
  @Post('register')
  register(@Body() registerDto: RegisterDto) {
    return this.authService.register(registerDto);
  }

  // Endpoint público para login
  @Post('login')
  login(@Body() loginDto: LoginDto) {
    return this.authService.login(loginDto);
  }

  // Endpoint protegido para renovar token
  @Get('check-status')
  @Auth() // Solo requiere estar autenticado
  checkAuthStatus(@GetUser() user: User) {
    return this.authService.checkAuthStatus(user);
  }

  // Endpoint para obtener perfil del usuario actual
  @Get('profile')
  @Auth() // Solo requiere estar autenticado
```

```
  getProfile(@GetUser() user: User) {
    return {
      id: user.id,
      email: user.email,
      fullName: user.fullName,
      permissions: user.permissions,
      isActive: user.isActive,
      createdAt: user.createdAt,
    };
  }
}
```

9.2 Proteger Endpoints en UsersController

Archivo: `src/users/users.controller.ts`

```
import { Controller, Get, Post, Body, Patch, Param, Delete } from
'@nestjs/common';

import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import { Auth } from '../auth/decorators/auth.decorator';
import { GetUser } from '../auth/decorators/get-user.decorator';
import { PermissionsTypes } from './enums/permissions.enum';
import { User } from './entities/user.entity';

@Controller('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  // Solo administradores pueden crear usuarios
  @Post()
  @Auth(PermissionsTypes.ADMIN)
  create(@Body() createUserDto: CreateUserDto) {
    return this.usersService.create(createUserDto);
  }

  // Administradores y moderadores pueden ver todos los usuarios
  @Get()
  @Auth(PermissionsTypes.ADMIN, PermissionsTypes.MODERATOR)
  findAll() {
    return this.usersService.findAll();
  }

  // Solo administradores pueden ver un usuario específico
  @Get(':id')
  @Auth(PermissionsTypes.ADMIN)
  findOne(@Param('id') id: string) {
    return this.usersService.findOne(id);
  }
```

```
  // Solo administradores pueden actualizar usuarios
  @Patch(':id')
  @Auth(PermissionsTypes.ADMIN)
  update(
    @Param('id') id: string,
    @Body() updateUserDto: UpdateUserDto,
    @GetUser() currentUser: User // Usuario que hace la petición
  ) {
    return this.usersService.update(id, updateUserDto, currentUser);
  }

  // Solo administradores pueden eliminar usuarios
  @Delete(':id')
  @Auth(PermissionsTypes.ADMIN)
  remove(
    @Param('id') id: string,
    @GetUser() currentUser: User // Usuario que hace la petición
  ) {
    return this.usersService.remove(id, currentUser);
  }

  // Cualquier usuario puede cambiar su propia contraseña
  @Patch('change-password')
  @Auth() // Solo requiere estar autenticado
  changePassword(
    @Body() changePasswordDto: any, // Crear este DTO
    @GetUser() user: User
  ) {
    return this.usersService.changePassword(user.id, changePasswordDto);
  }
}
```

9.3 Importar AuthModule en UsersModule

Archivo: `src/users/users.module.ts`

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { User } from './entities/user.entity';
import { AuthModule } from '../auth/auth.module'; // Importar AuthModule

@Module({
  controllers: [UsersController],
  providers: [UsersService],
  imports: [
    TypeOrmModule.forFeature([User]),
    AuthModule, // Importar para usar decoradores y guards
```

```
  ],
  exports: [UsersService], // Exportar si otros módulos lo necesitan
})
export class UsersModule {}
```

---

10. Testing y Validación

10.1 Endpoints para Probar

1. Registro de Usuario

```
POST http://localhost:3000/auth/register
Content-Type: application/json

{
  "email": "admin@test.com",
  "password": "123456",
  "fullName": "Admin User",
  "permissions": ["admin"]
}
```

Respuesta esperada:

```
{
  "id": "uuid-generado",
  "email": "admin@test.com",
  "fullName": "Admin User",
  "permissions": ["admin"],
  "token": "jwt-token-generado"
}
```

2. Login

```
POST http://localhost:3000/auth/login
Content-Type: application/json

{
  "email": "admin@test.com",
  "password": "123456"
}
```

3. Verificar Status (con token)

```
GET http://localhost:3000/auth/check-status
Authorization: Bearer <tu-jwt-token>
```

4. Obtener Usuarios (solo admin)

```
GET http://localhost:3000/users
Authorization: Bearer <tu-jwt-token>
```

10.2 Casos de Prueba

Escenario	Endpoint	Token	Rol	Resultado Esperado
Sin token	GET /users	❌	-	401 Unauthorized
Token inválido	GET /users	❌	-	401 Unauthorized
Usuario normal	GET /users	☑	user	403 Forbidden
Administrador	GET /users	☑	admin	200 OK
Token expirado	GET /users	❌	-	401 Unauthorized
Usuario inactivo	GET /users	☑	admin	401 Unauthorized

10.3 Comandos de Testing

```
# Ejecutar tests unitarios
npm run test

# Ejecutar tests e2e
npm run test:e2e

# Ejecutar tests con coverage
npm run test:cov
```

10.4 Verificar JWT Payload

Puedes verificar que el JWT solo contiene el ID usando jwt.io:

1. Copia el token generado
2. Pégalo en jwt.io
3. Verifica que el payload solo contenga:

```
{
  "id": "uuid-del-usuario",
  "iat": 1234567890,
```

```
  "exp": 1234567890
}
```

Resumen de Archivos Creados

```
src/
├── auth/
│   ├── decorators/
│   │   ├── auth.decorator.ts          # Decorador compuesto
│   │   ├── get-user.decorator.ts      # Obtener usuario de request
│   │   └── role-protected.decorator.ts # Definir roles requeridos
│   ├── dto/
│   │   ├── login.dto.ts               # Datos para login
│   │   └── register.dto.ts            # Datos para registro
│   ├── guards/
│   │   └── user-role.guard.ts         # Validar roles de usuario
│   ├── interfaces/
│   │   └── jwt-payload.interface.ts   # Estructura del JWT (solo ID)
│   ├── strategies/
│   │   └── jwt.strategy.ts            # Validación de JWT
│   ├── auth.controller.ts             # Endpoints de autenticación
│   ├── auth.module.ts                 # Configuración del módulo
│   └── auth.service.ts                # Lógica de negocio
└── users/
    ├── enums/
    │   └── permissions.enum.ts        # Tipos de permisos
    └── entities/
        └── user.entity.ts             # Entidad de base de datos
```

Comandos Utilizados

```
# Generar módulo auth
nest g res auth --no-spec

# Generar guard
nest g gu auth/guards/user-role --no-spec

# Generar decoradores
nest g d auth/decorators/get-user --no-spec
nest g d auth/decorators/role-protected --no-spec
nest g d auth/decorators/auth --no-spec

# Crear carpetas
mkdir src/auth/interfaces
mkdir src/auth/strategies
mkdir src/users/enums
```

Próximos Pasos

1. **Implementar refresh tokens** para mayor seguridad
2. **Agregar rate limiting** para prevenir ataques de fuerza bruta
3. **Implementar 2FA** para usuarios administradores
4. **Agregar logs de auditoría** para acciones sensibles
5. **Crear middleware de logging** para requests autenticados

¡Tu sistema de autenticación está completo y listo para producción! 🚀