Universidad Icesi

Autenticación y Autorización en NestJS

Seguridad Web, Hashing con bcrypt, Passport, JWT y PermissionsGuard

Desarrollo de Entornos Digitales Web — Universidad Icesi

Universidad ICESI

Agenda y Tres Pilares de la Semana 7

01. Fundamentos & Hashing

- Modelo relacional de persistencia: User, Role y Permission.
- OWASP Top 10: Peligro de contraseñas en texto plano.
- Hashing unidireccional con bcrypt vs. cifrado.
- Salting dinámico y factor de costo (2^k iteraciones).

02. Passport & JWT

- Guards en el ciclo de vida de NestJS (`CanActivate`).
- Passport y el Patrón Strategy desacoplado.
- Anatomía del JWT (RFC 7519): Header, Payload, Firma.
- Arquitectura y pasos para construir el `AuthModule`.

03. Autorización Granular

- De Roles a Permisos atómicos (RBAC vs. PBAC).
- Decorador `@Permissions` y metadatos con `Reflector`.
- Implementación de `PermissionsGuard`.
- Regla de orden en `@UseGuards` y diagnóstico 401/403.

Universidad ICESI

01. Fundamentos de Seguridad y Hashing

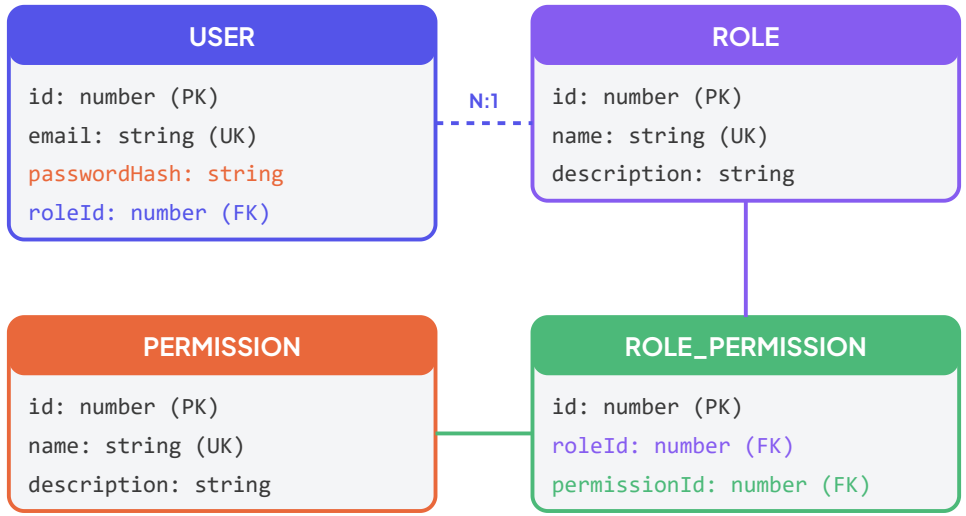
Protección de credenciales mediante funciones criptográficas de derivación, modelo relacional y la frontera entre Autenticación y Autorización.

Prerrequisitos: Modelo Relacional de Persistencia

Para implementar seguridad perimetral robusta, se requiere un **modelo relacional normalizado** que soporte usuarios, roles y permisos atómicos:

- **User:** Identidad única (email), credencial protegida (`passwordHash`) y relación con un rol.
- **Role:** Agrupador de negocio (ej. *Admin*, *Manager*, *Client*).
- **Permission:** Capacidad atómica del sistema (ej. `users:read`, `users:delete`).
- **RolePermission:** Tabla asociativa para relación M:N flexible.

DIAGRAMA DE ENTIDADES DE SEGURIDAD



Universidad icesi

Base de Datos Comprometida

```
ID | Email | Password (Plano)
1 | ana@icesi...| "MiClave123" [!]
```

X

Protección con bcrypt

```
ID | Email | passwordHash
1 | ana@icesi...| $2b$10$vI8a...6zNq
```

Hash irreversible con Salt dinámico

El Riesgo Crítico: Contraseñas en Texto Plano

OWASP Top 10 (A02:2021 — Cryptographic Failures):
Almacenar contraseñas en texto plano es una de las vulnerabilidades más severas en el desarrollo de software.

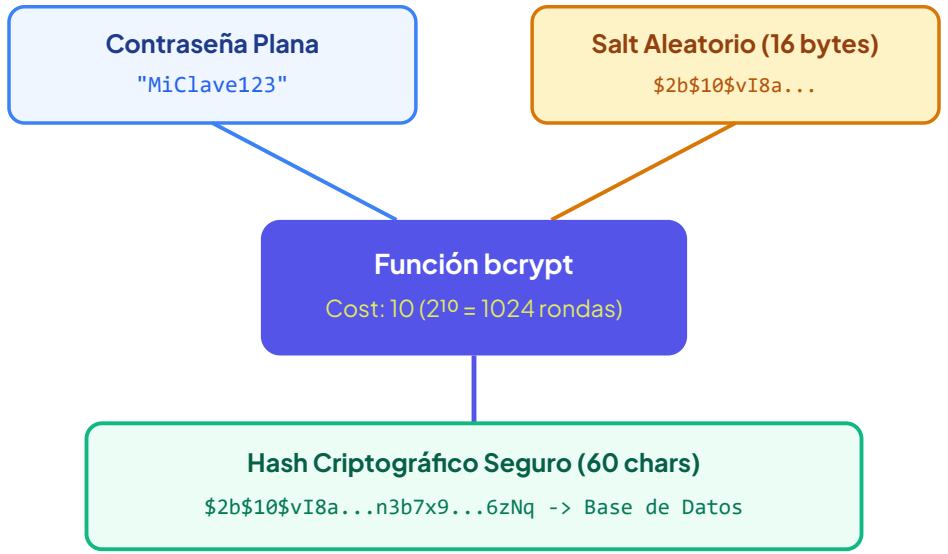
- **Exposición Inmediata:** Una inyección SQL, backup filtrado o brecha física entrega instantáneamente las credenciales de todos los usuarios.
- **Reutilización de Credenciales:** Los usuarios suelen repetir contraseñas en múltiples servicios, multiplicando el impacto del ataque.
- **Cifrado vs. Hash:** El cifrado simétrico/asimétrico es *reversible con una clave*. El hash criptográfico es *estrictamente unidireccional* y nunca puede revertirse.

Hashing Criptográfico con bcrypt

bcrypt es una función de derivación de claves basada en el cifrado Blowfish diseñada específicamente para la protección de contraseñas:

- **Unidireccional:** No existe algoritmo matemático viable para revertir el hash al texto plano original.
- **Salting Automático:** Genera un valor aleatorio (*salt*) único por usuario, neutralizando los ataques con **Rainbow Tables** (tablas precomputadas).
- **Costo Computacional Adaptable:** El parámetro de costo k ejecuta 2^k rondas internas. Un costo de 10 representa 1024 iteraciones, ralentizando la fuerza bruta en GPU.

FLUJO DE TRANSFORMACIÓN Y PERSISTENCIA



bcrypt en NestJS: Paquetes y Configuración

01. PAQUETES A INSTALAR

Instalar la librería nativa y sus definiciones de tipos TypeScript:

```
npm i bcrypt
```

```
npm i -D @types/bcrypt
```

02. VARIABLES DE ENTORNO

Configurar el factor de costo en el archivo `.env`:

```
SALT_ROUNDS=10
```

10–12 rondas: equilibrio ideal entre seguridad y latencia HTTP.

03. ARCHIVO A MODIFICAR

Ruta del archivo:

`src/users/users.service.ts`

Inyectar `ConfigService` para leer las rondas antes de persistir la entidad.

04. OPERACIONES CLAVE

- **Registro:** `bcrypt.hash(pass, rounds)`
- **Autenticación:** `bcrypt.compare(pass, hash)`

Autenticación vs. Autorización: La Gran Distinción

Comprender la frontera operativa entre **AuthN** y **AuthZ** es la base del diseño de seguridad en APIs RESTful:

AuthN **¿Quién es el usuario?**

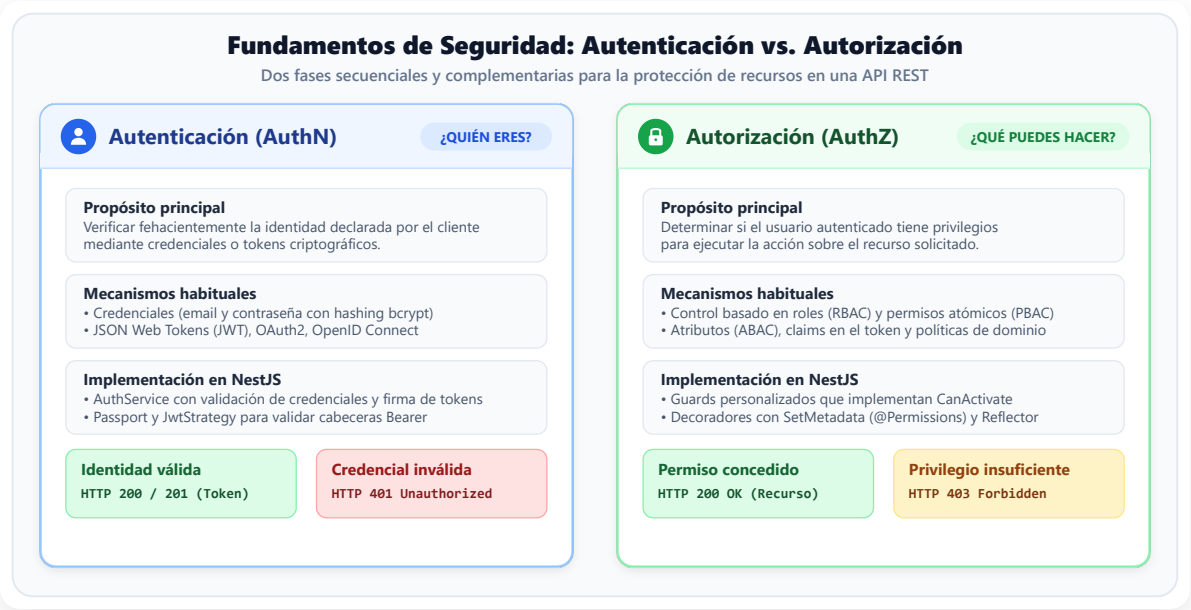
Verifica la **identidad** declarada mediante credenciales (email/password) o tokens JWT.

Fallo: **HTTP 401 Unauthorized** (reintentable con credenciales correctas).

AuthZ **¿Qué permisos tiene?**

Regula las **acciones permitidas** sobre un recurso según roles y privilegios.

Fallo: **HTTP 403 Forbidden** (identidad confirmada, pero acceso denegado).



Universidad ICESI

02. Passport y JSON Web Tokens (JWT)

Autenticación Stateless en el ciclo de vida de NestJS mediante el Patrón Strategy y el estándar RFC 7519.

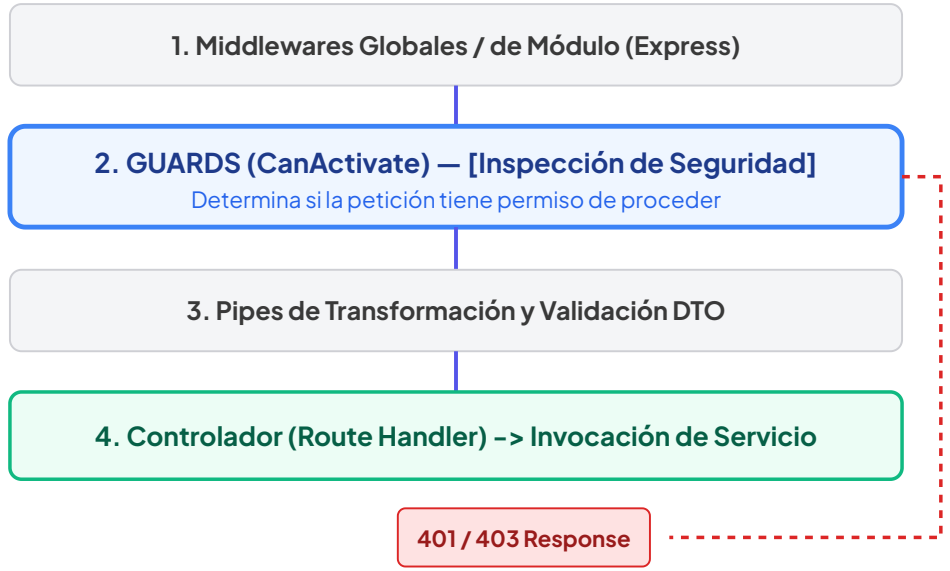
¿Qué es un Guard en NestJS?

Un **Guard** es una clase decorada con `@Injectable()` que implementa la interfaz `CanActivate`:

```
export interface CanActivate {
  canActivate(context: ExecutionContext):
    boolean | Promise<boolean>;
}
```

- Si retorna `true`, la petición continúa hacia pipes y el controlador.
- Si retorna `false`, NestJS cancela y emite **HTTP 403 Forbidden**.
- Tiene acceso a `ExecutionContext` y metadatos con `Reflector`.

CICLO DE VIDA DE UNA PETICIÓN EN NESTJS



Universidad ICESI

Passport y el Patrón Strategy

Passport desacopla el *mecanismo de autenticación* de la lógica de enrutamiento mediante el patrón de diseño **Strategy**:

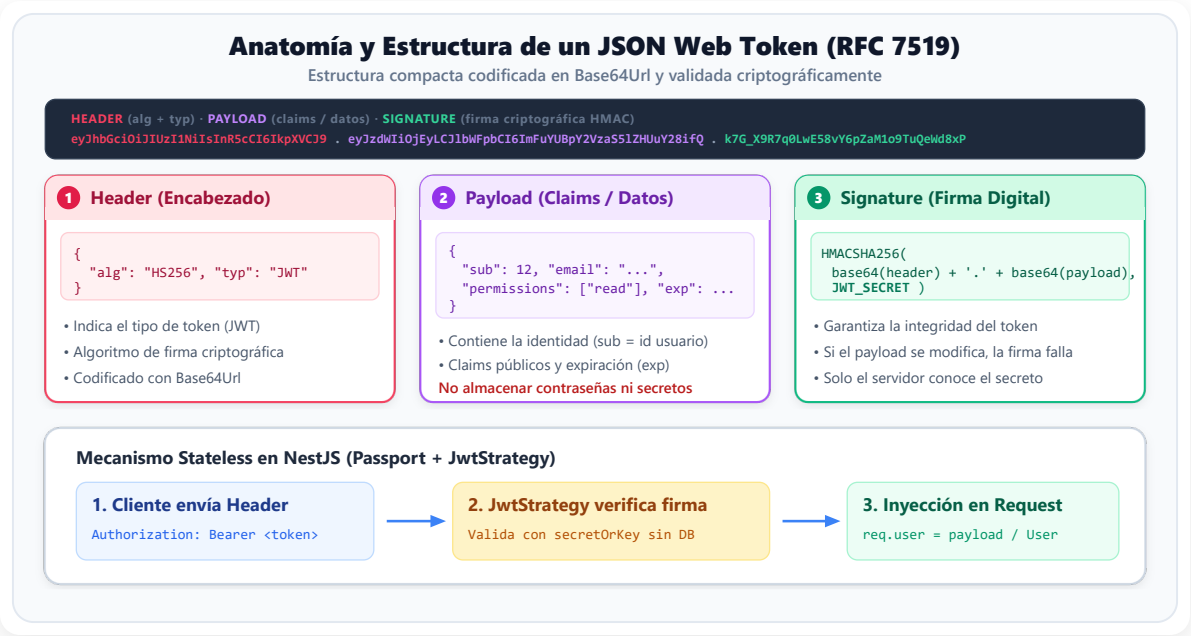
- **Encapsulamiento:** Cada estrategia (`JwtStrategy`, `LocalStrategy`) se implementa en una clase aislada.
- **Inyección en NestJS:** Mediante `@nestjs/passport`, las estrategias se convierten en *Providers* inyectables.
- **Controladores Agnósticos:** El controlador solo invoca `@UseGuards(AuthGuard('jwt'))` sin preocuparse por la extracción del token ni la validación criptográfica.

Anatomía de un JSON Web Token (RFC 7519)

Un **JWT** es un estándar abierto para transmitir declaraciones seguras entre cliente y servidor estructuradas en 3 partes:

- **1. Header:** Metadatos del token (tipo `JWT` y algoritmo de firma `HS256`).
- **2. Payload:** Declaraciones (*claims*) del usuario: `sub` (ID), `email`, `iat`, `exp`.
- **3. Signature:** Firma criptográfica calculada con la clave secreta del servidor.

> **¡Advertencia Crítica!** El Payload solo está codificado en Base64Url, *no cifrado*. Nunca guardes contraseñas ni datos bancarios dentro de un JWT.

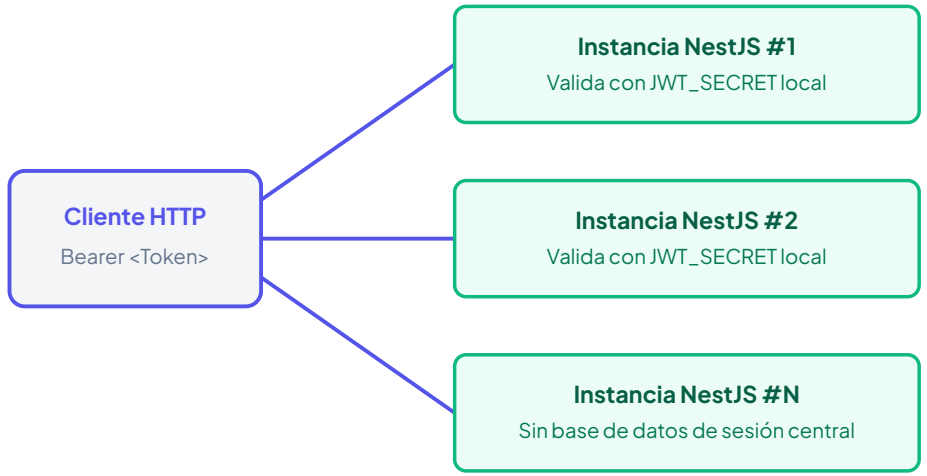


Stateful (Sesiones) vs. Stateless (JWT)

Elegir entre sesiones tradicionales y JWT transforma radicalmente la escalabilidad del backend:

Criterio	Sesiones (Stateful)	JWT (Stateless)
Estado	En memoria / DB / Redis	En el cliente
Escalabilidad	Requiere *Sticky Sessions*	Horizontal inmediata
RAM Servidor	Crece con usuarios $O(N)$	Casi nulo $O(1)$
Revocación	Instantánea (borrar en DB)	Por expiración / Blacklist

ESCALABILIDAD HORIZONTAL CON JWT



Ecosistema de Autenticación: Paquetes NestJS

@NESTJS/PASSPORT

IoC Bridge

Módulo oficial que integra Passport con el contenedor de inyección de dependencias de NestJS.

Provee `PassportStrategy` y `AuthGuard`.

PASSPORT & PASSPORT-JWT

Auth Engine

`passport` : Motor central de middleware.

`passport-jwt` : Estrategia que extrae y valida cabeceras `Authorization: Bearer <token>` .

@NESTJS/JWT

Token Utility

Envuelve la librería `jsonwebtoken` en un servicio inyectable (`JwtService`) para firmar (`sign`) y verificar tokens.

COMANDOS DE INSTALACIÓN

En la terminal del proyecto:

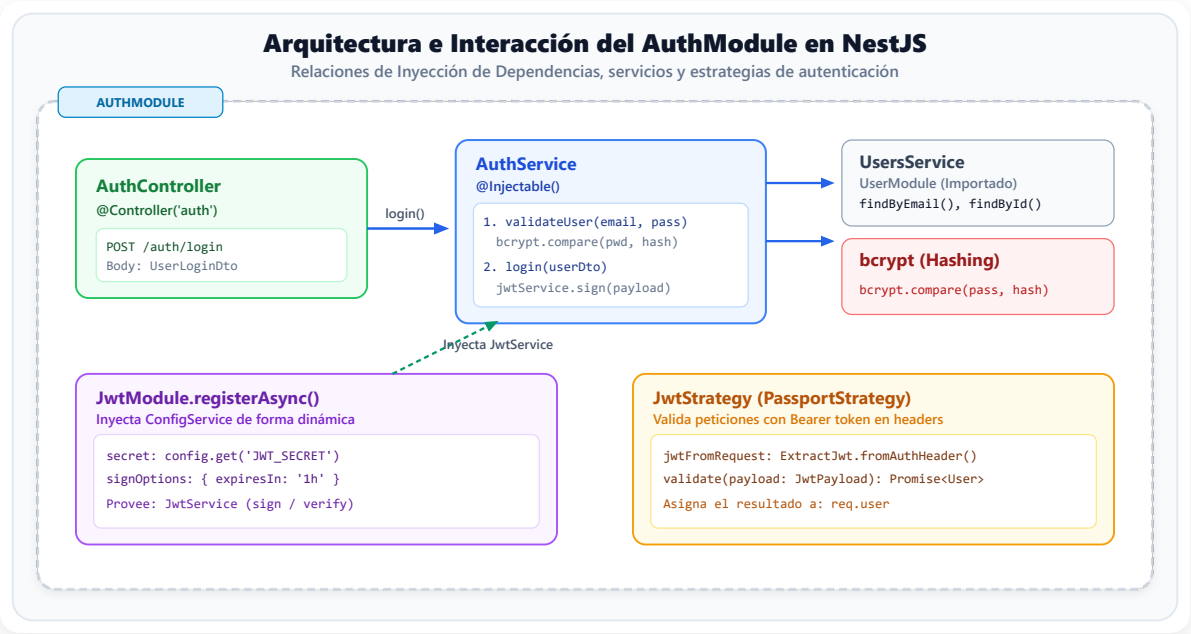
```
npm i @nestjs/passport passport passport-jwt @nestjs/jwt
```

```
npm i -D @types/passport-jwt
```

Arquitectura de Componentes: AuthModule

El **AuthModule** organiza las dependencias de seguridad y establece el flujo de autenticación:

- **AuthController:** Expone el endpoint público `POST /auth/login`.
- **AuthService:** Valida credenciales contra la DB con `bcrypt.compare` y firma el token con `JwtService`.
- **JwtStrategy:** Intercepta peticiones protegidas y coloca al usuario autenticado en `req.user`.



Guía de Implementación: AuthModule en 4 Pasos

PASO 1: VARIABLES .ENV

Definir en el archivo `.env` :

```
JWT_SECRET=super_secreto_icesi
```

```
JWT_EXPIRES_IN=1h
```

PASO 2: DTOS & PAYLOAD

Archivos a crear:

- `src/auth/dto/user-login.dto.ts` (validación con class-validator)
- `src/auth/interfaces/jwt-payload.interface.ts` (sub, email, permissions)

PASO 3: AUTHSERVICE & STRATEGY

Lógica de servicio:

- `src/auth/auth.service.ts` (validateUser + login)
- `src/auth/strategies/jwt.strategy.ts` (extiende PassportStrategy)

PASO 4: MÓDULO & CONTROLLER

Ensamble final:

- `src/auth/auth.module.ts` (JwtModule.registerAsync)
- `src/auth/auth.controller.ts` (POST /auth/login)

Universidad ICESI

03. Autorización Granular con Guards

De Roles a Permisos Granulares (PBAC), decorador @Permissions y construcción del PermissionsGuard con Reflector.

De Roles a Permisos Granulares: RBAC vs. PBAC

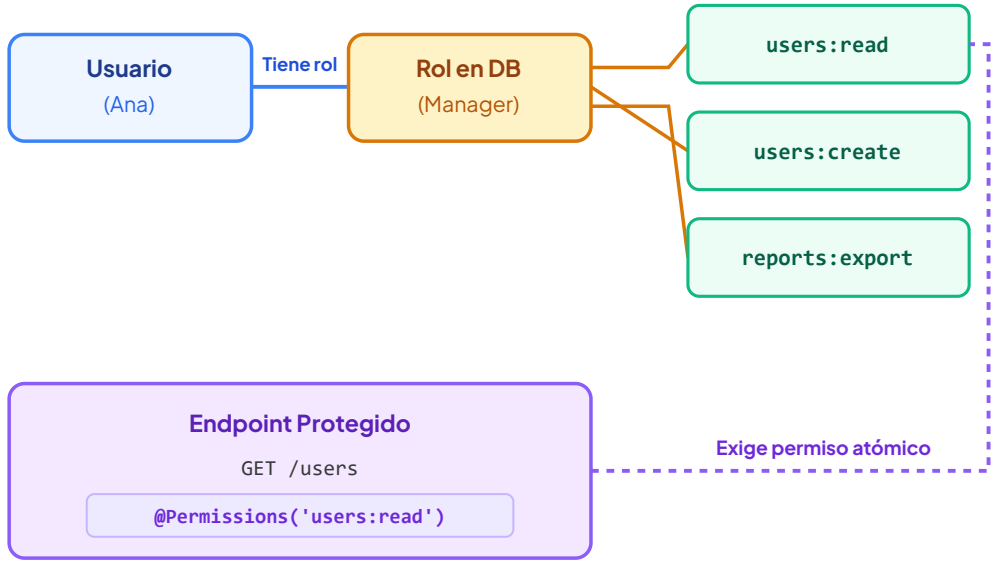
LIMITACIONES DE RBAC TRADICIONAL

- **Explosión de Roles:** Surgen roles artificiales (*editor_sin_borrado*, *auditor_temporal*).
- **Acoplamiento:** Reconfigurar permisos de un rol exige modificar y recompilar el backend.

LA SOLUCIÓN: PERMISOS ATÓMICOS (PBAC)

- **Rutas Declarativas:** Los endpoints solo exigen permisos atómicos (ej. `users:read`).
- **Roles Dinámicos:** Los roles son simples agrupaciones configurables en base de datos.

ARQUITECTURA DE AUTORIZACIÓN PBAC



Universidad ICESI

@Permissions('users:read')
Decorador en el Controlador

SetMetadata(KEY, values)
Almacena ['users:read'] en Reflect

Reflector.get() en Guard
Lee metadatos y verifica permisos

Creación del Decorador @Permissions

NestJS permite adjuntar metadatos personalizados a cualquier controlador mediante `SetMetadata`:

```
import { SetMetadata } from '@nestjs/common';

export const PERMISSIONS_KEY = 'permissions';

export const Permissions = (...permissions: string[]) =>
 SetMetadata(PERMISSIONS_KEY, permissions);
```

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

En tiempo de ejecución: Guarda los strings en la tabla de reflexión de TypeScript para ser leídos por los guards.

El Guardia PermissionsGuard con Reflector

El `PermissionsGuard` implementa `CanActivate` y ejecuta la validación granular:

- **1. Extrae Requisitos:** Consulta los permisos con `this.reflector.get(PERMISSIONS_KEY, context.getHandler())`.
- **2. Lee Usuario:** Obtiene el usuario autenticado desde `req.user`.
- **3. Aserción de Permisos:** Valida que el rol contenga *todos* los permisos requeridos (`every(...)`).
- **4. Bloqueo 403:** Lanza `ForbiddenException` si carece de al menos un permiso.

Protección de Controladores y Regla de Orden

En `src/users/users.controller.ts` aplicamos los guards en cascada:

```
@Controller('users')
// Regla de orden estricta:
@UseGuards(AuthGuard('jwt'), PermissionsGuard)
export class UsersController {

 @Get()
 @Permissions('users:read')
 findAll() { ... }

 @Post()
 @Permissions('users:create')
 create() { ... }
}
```

¡REGLA DE ORO DEL ORDEN EN @USEGUARDS!

El orden de ejecución de los guards es **estrictamente secuencial de izquierda a derecha**:

1. **1° AuthGuard('jwt'):** Valida el token Bearer y puebla el objeto `req.user`.
2. **2° PermissionsGuard:** Lee `req.user` para validar sus permisos.

Si se invierte el orden, PermissionsGuard fallará siempre con error 401 porque req.user aún no existe.

Diagnóstico de Respuestas HTTP en Seguridad

HTTP 200 / 201 OK

Acceso Concedido

- Token Bearer presente y firma criptográfica válida.
- Fecha de expiración vigente.
- El usuario posee el permiso atómico requerido en la ruta.

HTTP 401 Unauthorized

Fallo de Autenticación

- Cabecera `Authorization` ausente.
- Token adulterado (firma no coincide con `JWT_SECRET`).
- El token superó su fecha límite de expiración.

HTTP 403 Forbidden

Fallo de Autorización

- Identidad verificada con éxito (sabemos quién es).
- El rol del usuario **no posee** el permiso requerido.
- Reintentar con las mismas credenciales fallará igual.

Solución a Errores Comunes y Buenas Prácticas

ERROR 500: RELACIONES EN TYPEORM

¡Cuidado con Relations Undefined!

Si `user.role` o `rolePermissions` no se cargan, el Guard arroja error 500.

Asegurar `relations: ['role', 'role.rolePermissions', ...]` en el UsersService.

GESTIÓN DE SECRETOS

Nunca Hardcodear Claves

`JWT_SECRET` jamás debe subirse a Git. Inyectar en producción vía variables de entorno.

Longitud mínima recomendada: 256 bits aleatorios.

DATOS PROHIBIDOS EN JWT

El Payload es Público

Cualquiera puede decodificar Base64Url sin la clave secreta.

Nunca incluir passwords, hashes, tokens de tarjetas ni datos privados en claims.

TIEMPOS DE EXPIRACIÓN

Ventana de Exposición Corta

Tokens JWT con expiración de **15m a 1h**.

En caso de robo del token en tránsito, la ventana de ataque queda acotada.

Resumen de Paquetes y Archivos de la Semana 7

PAQUETES Y COMANDOS INSTALADOS

Paquete	Propósito
`bcrypt`	Hashing criptográfico y salting
`@nestjs/passport`	Enlace IoC para Passport en NestJS
`passport-jwt`	Estrategia para cabeceras Bearer
`@nestjs/jwt`	Firma y verificación reactiva de tokens

```
npm i bcrypt @nestjs/passport passport passport-jwt @nestjs/jwt
npm i -D @types/bcrypt @types/passport-jwt
```

ARCHIVOS CLAVE EN EL PROYECTO

```
.env [SALT_ROUNDS, JWT_SECRET, JWT_EXPIRES_IN]
src/
  ├── users/
  │   ├── users.service.ts (bcrypt.hash)
  │   └── users.controller.ts (@UseGuards)
  └── auth/
      ├── auth.service.ts (validateUser, login)
      ├── auth.controller.ts (POST /login)
      ├── auth.module.ts (JwtModule.registerAsync)
      ├── strategies/jwt.strategy.ts
      ├── decorators/permissions.decorator.ts
      └── guards/permissions.guard.ts
```

Conclusiones: Cadena de Defensa en Profundidad

La seguridad robusta en NestJS no depende de un solo componente, sino de una **cadena de defensa en profundidad** articulada en cuatro niveles:

1. Persistencia — **Hashing bcrypt**

Contraseñas protegidas con salt dinámico y factor de costo adaptable contra filtraciones.

2. Identidad — **Tokens Stateless JWT**

Firma criptográfica HMAC-SHA256 permitiendo escalabilidad horizontal lineal sin sesiones centralizadas.

3. Perímetro — **Guards en el Ciclo de Vida**

Interceptores de seguridad previos a pipes y controladores que cortan accesos no autorizados (401).

4. Autorización — **Control Granular PBAC**

Permisos atómicos declarados con `@Permissions` y validados dinámicamente con `Reflector` (403).