Universidad Icesi

Computación 3: Ecosistema NestJS

Semana 5: Arquitectura Modular, Inversión de Control y Persistencia con TypeORM

Facultad de Ingeniería, Diseño y Ciencias Aplicadas — Universidad Icesi

Fundamentos y Filosofía del Framework NestJS

NestJS es un framework progresivo de Node.js diseñado para construir aplicaciones de servidor robustas y escalables. Provee una arquitectura empresarial lista para producción combinando tres paradigmas:

Orientado a Objetos (POO)

Estructura limpia basada en clases, decoradores de TypeScript, encapsulación e interfaces estrictas.

Programación Funcional (FP)

Uso de funciones puras, inmutabilidad de datos y transformaciones directas en tuberías (Pipes).

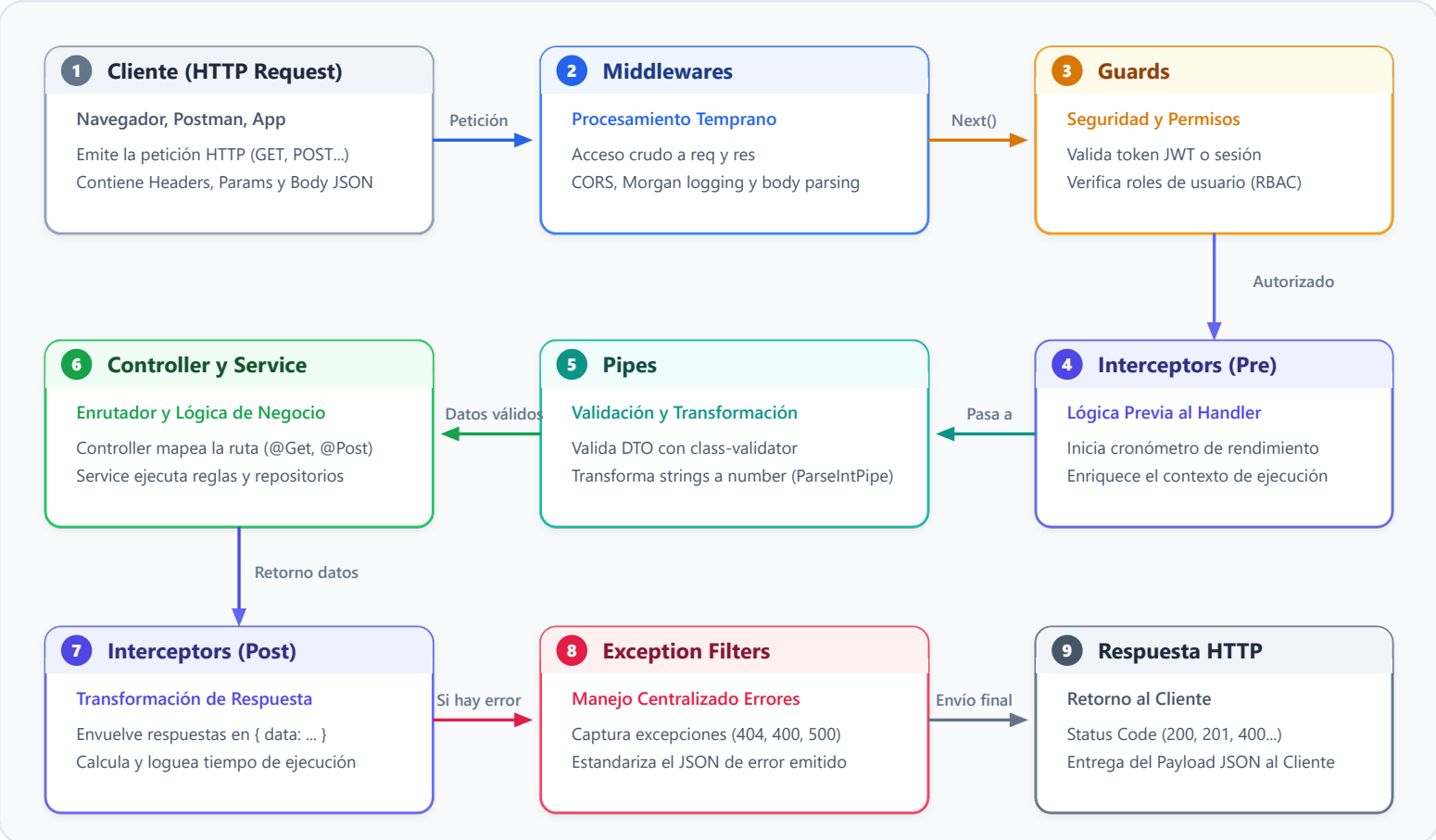
Reactivo Funcional (FRP)

Integración con RxJS y Observables para manejo reactivo de eventos, flujos asíncronos y WebSockets.

Inspiración Arquitectónica: Quienes conocen *Spring Boot* o *Angular* encuentran en NestJS una estructura familiar de controladores, servicios inyectables y módulos bien delimitados.

Ciclo de Vida de una Solicitud (Request Lifecycle)

Cada solicitud HTTP entrante recorre ordenadamente las 9 etapas antes de retornar la respuesta al cliente:



Universidad ICESI

CLI de NestJS y Primeros Pasos

Instalación, Estructura del Proyecto y Generadores

Herramientas del CLI de NestJS

Instalación y Creación

```
# 1. Instalar el CLI globalmente
npm install -g @nestjs/cli

# 2. Crear un nuevo proyecto
nest new my-nest-app

# 3. Iniciar servidor en desarrollo
cd my-nest-app
npm run start:dev
```

El CLI preconfigura TypeScript, Webpack/SWC, ESLint, Jest y la arquitectura inicial sin configuraciones manuales.

Comandos Generadores Rápidos

```
# Generación individual por capas
nest g mo users      # Genera UsersModule
nest g co users      # Genera UsersController
nest g s users       # Genera UsersService

# Generador "Todo en Uno" (CRUD Resource)
nest g resource users
# Crea Módulo, Controlador, Servicio, DTOs y Entidades
```

Tip del CLI: Los comandos de generación actualizan automáticamente el arreglo de `controllers` y `providers` en el módulo correspondiente.

Universidad ICESI

Anatomía de un Proyecto NestJS

Estructura del Directorio src/

```
my-nest-app/
├── src/
│   ├── app.controller.ts    # Controlador base (rutas HTTP '/')
│   ├── app.controller.spec  # Pruebas unitarias
│   ├── app.module.ts        # Módulo raíz de la app
│   ├── app.service.ts       # Servicio base ('Hello World!')
│   └── main.ts              # Punto de entrada HTTP
├── test/                    # Pruebas end-to-end (e2e)
├── nest-cli.json            # Configuración del CLI
├── package.json             # Dependencias y scripts
└── tsconfig.json            # Compilador TypeScript
```

El Punto de Entrada: main.ts

```ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  // Instancia Nest a partir del módulo raíz
  const app = await NestFactory.create(AppModule);

  // Inicia la escucha en el puerto 3000
  await app.listen(3000);
}
bootstrap();
```

`NestFactory.create()` inicializa el contenedor IoC, resuelve el árbol de dependencias y monta el servidor HTTP Express subyacente.

Universidad icesi

Implementación de Controlador y Servicio

src/users/users.service.ts

```ts
import { Injectable } from '@nestjs/common';

@Injectable()
export class UsersService {
  private users = [
    { id: 1, name: 'Alice', email: 'alice@icesi.edu.co' },
  ];

  findAll() {
    return this.users;
  }

  findOne(id: number) {
    return this.users.find(u => u.id === id);
  }
}
```

src/users/users.controller.ts

```ts
import { Controller, Get, Param } from '@nestjs/common';
import { UsersService } from './users.service';

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

  @Get()
  getAll() {
    return this.usersService.findAll();
  }

  @Get(':id')
  getById(@Param('id') id: string) {
    return this.usersService.findOne(Number(id));
  }
}
```

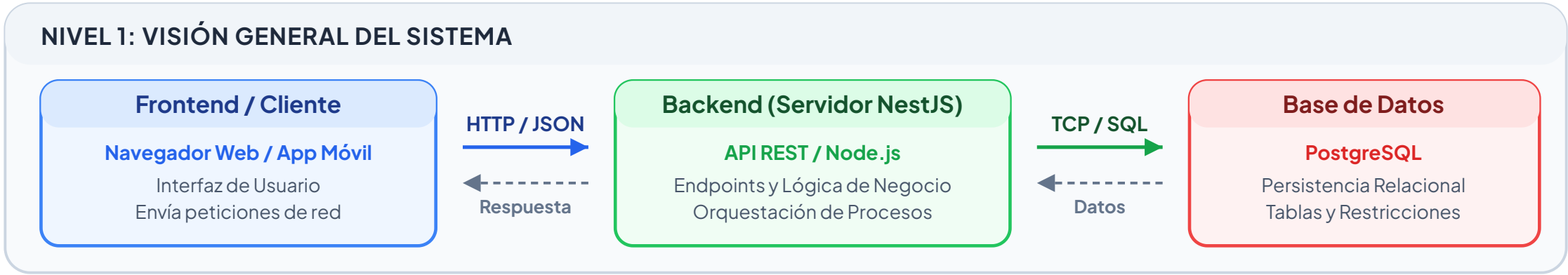
Universidad ICESI

Arquitectura por Capas e Inversión de Control

Desacoplamiento, Contenedor IoC y Providers

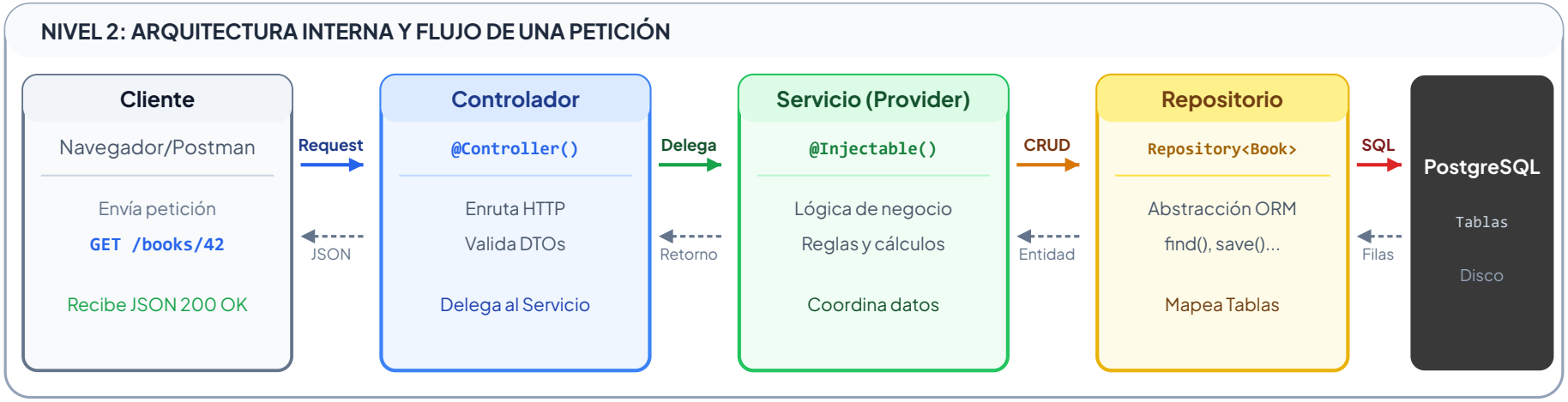
Arquitectura por Capas: Nivel 1 (Visión General)

A nivel macro, el sistema separa la interfaz de usuario de la persistencia física mediante un intermediario de servicios backend:



Arquitectura por Capas: Nivel 2 (Flujo Interno)

Ruta interna que recorre cada petición a través de los componentes especializados de NestJS:



El Problema del Acoplamiento Tradicional con "new"

Anti-patrón: Instanciación Manual

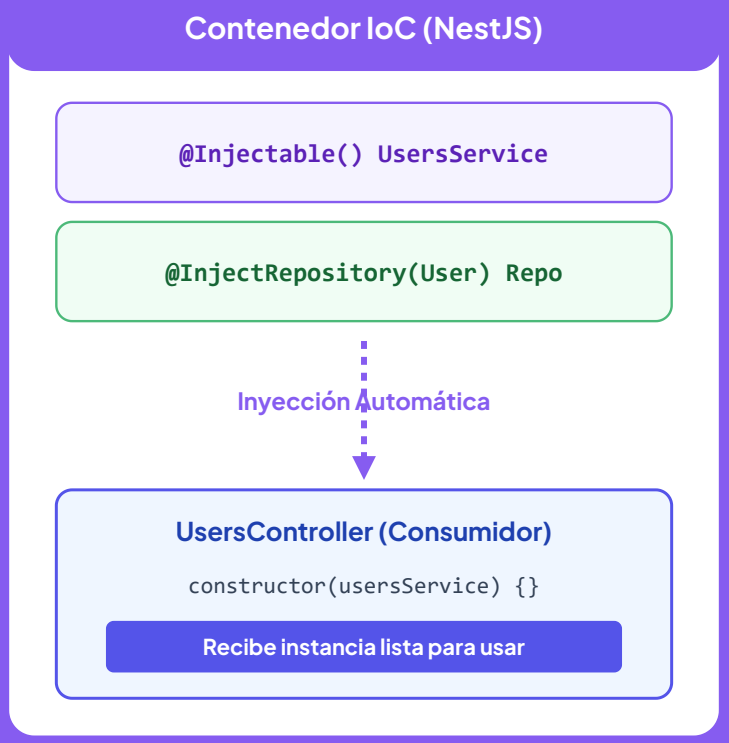
```
export class UsersController {
  private usersService: UsersService;

  constructor() {
    // Acoplamiento directo y forzoso:
    // El controlador asume la creación del servicio
    this.usersService = new UsersService();
  }

  getUsers() {
    return this.usersService.findAll();
  }
}
```

Cuando una clase usa `new` internamente, asume el control total de la instanciación de sus dependencias, creando un vínculo inflexible.

Desventajas Críticas en Producción:

- **Acoplamiento Rígido:**
  Si `UsersService` cambia su constructor para requerir un repositorio, hay que refactorizar manualmente todos los controladores del sistema.
- **Pruebas Unitarias Imposibles:**
  No se puede aislar el controlador para testearlo con un *mock* o servicio simulado en memoria.
- **Duplicación Ineficiente de Memoria:**
  Cada nueva instancia del consumidor crea de nuevo toda la cadena de objetos en cascada.

Universidad ICESI



Inversión de Control (IoC) en NestJS

En lugar de que cada componente instancie sus dependencias, **invierte el control**: la clase simplemente declara qué necesita en su constructor, y el **Contenedor IoC de NestJS** se encarga de resolverlo.

Criterio	Instanciación Manual (`new`)	Inversión de Control (IoC)
Responsabilidad	La clase consumidora crea dependencias	El Contenedor IoC las suministra
Acoplamiento	Alto (atado a implementaciones concretas)	Bajo (orientado a contratos/abstracciones)
Testing Unitario	Muy difícil (sin posibilidad de mocks)	Sencillo (se inyectan objetos simulados)
Mantenimiento	Frágil ante cambios en constructores	Centralizado y automático en módulos

Beneficio: NestJS crea una única instancia (Singleton por defecto) y la comparte eficientemente entre todos los componentes que la requieran.

Universidad ICESI

¿Qué es un Provider en NestJS?

SERVICIOS (SERVICES)

Clases decoradas con `@Injectable()` que concentran la **lógica de negocio**, reglas del dominio y orquestación de flujos de datos.

Ejemplo: UsersService

REPOSITORIOS (REPOSITORIES)

Proveedores especializados provistos por TypeORM para abstraer consultas SQL, mapear tablas a entidades y gestionar transacciones ACID.

Repository<User>

ADAPTADORES Y CLIENTES

Servicios que gestionan la comunicación con sistemas externos: pasarelas de pago (Stripe), envío de correos (SendGrid) o microservicios.

MailService, PaymentClient

HELPERS Y FACTORIES

Clases auxiliares para tareas transversales: encriptación de claves (Bcrypt), generación de tokens, formateo o fábricas de configuración dinámica.

HashHelper, ConfigService

Declaración e Inyección en TypeScript

1. Declaración del Provider

```ts
import { Injectable } from '@nestjs/common';

export interface User {
  id: number;
  name: string;
}

@Injectable() // <-- Metadatos de reflexión
export class UsersService {
  private readonly users: User[] = [
    { id: 1, name: 'Ada Lovelace' }
  ];

  findAll(): User[] {
    return this.users;
  }
}
```

`@Injectable()` le indica al compilador que adjunte metadatos para que el contenedor IoC pueda resolver la clase.

2. Inyección por Constructor

```ts
import { Controller, Get } from '@nestjs/common';
import { UsersService } from './users.service';

@Controller('users')
export class UsersController {
  // Inyección basada en constructor:
  // TypeScript analiza el tipo 'UsersService'
  constructor(
 private readonly usersService: UsersService
 ) {}

  @Get()
  getAll() {
    return this.usersService.findAll();
  }
}
```

> **Tip TypeScript:** Al usar `private readonly` en el constructor, se declara y asigna la propiedad en una sola instrucción.

Universidad ICESI

Modularidad y Encapsulación

Estructura de @Module() y Fronteras de Visibilidad

Universidad ICESI

Anatomía del Decorador @Module()

CONTROLLERS: [...]

Lista de controladores que pertenecen a este módulo. NestJS los instancia y los enlaza con el enrutador HTTP para escuchar peticiones externas.

Puntos de entrada HTTP

PROVIDERS: [...]

Servicios y clases auxiliares instanciados por el contenedor IoC. **Por defecto son PRIVADOS** al módulo y solo visibles internamente.

Encapsulación privada

IMPORTS: [...]

Lista de otros módulos cuyas capacidades y providers exportados son requeridos dentro de este módulo (ej: `TypeOrmModule`).

Dependencias externas

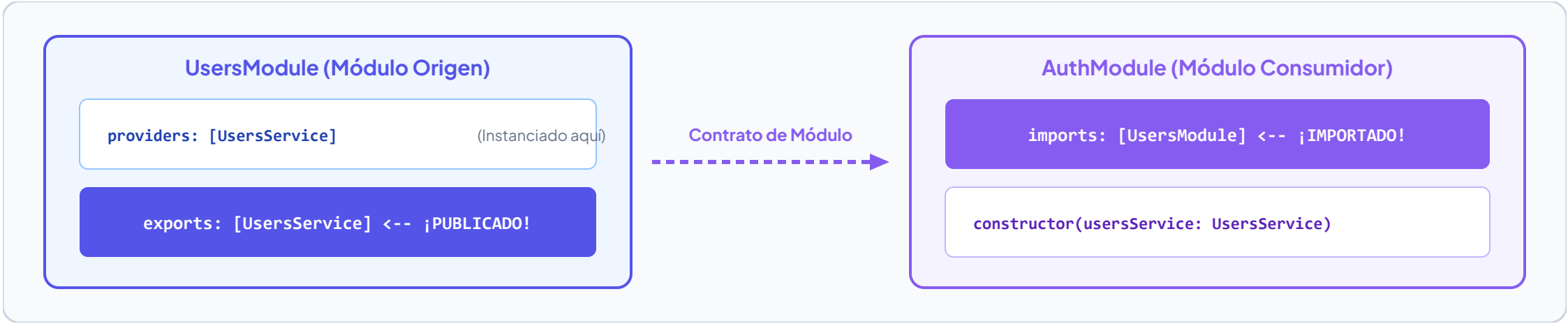
EXPORTS: [...]

Subconjunto de providers de este módulo que se hacen **PÚBLICOS** para que cualquier módulo que lo importe pueda inyectarlos.

API Pública del Módulo

Fronteras de Encapsulación y Compartición de Providers

Los módulos en NestJS aíslan su estado interno. Para que un servicio viaje entre módulos distintos, debe existir un acuerdo explícito: **exportar en el origen e importar en el destino**.



Universidad ICESI

- Missing @Injectable
- Missing in providers
- Missing in exports
- Missing in imports
- Compilación OK

Error Crítico: "Nest can't resolve dependencies"

Ocurre cuando un componente solicita un provider en su constructor pero el framework no encuentra su definición en el contexto del módulo.

```
Error: Nest can't resolve dependencies of the UsersController (?).
Please make sure that the argument UsersService at index [0]
is available in the UsersModule context.
```

Protocolo de Solución en 3 Pasos:

1. **¿Tiene decorador?** Verificar que la clase del servicio tenga `@Injectable()`.
2. **¿Está en providers?** Confirmar que esté listado en `providers: [UsersService]`.
3. **¿Módulo externo?** Si viene de otro módulo, verificar `exports` en origen e `imports` en destino.

Universidad ICESI

TypeORM, Entidades e Infraestructura Docker

PostgreSQL 18, Mapeo Objeto-Relacional y Relaciones 1:N

¿Qué es un ORM y por qué usar TypeORM?

Sin ORM (Enfoque Manual)

```
// SQL Crudo propenso a errores de tipos
const result = await db.query(
  'SELECT id, name, email FROM users WHERE id = $1',
  [userId]
);
// Requiere mapeo manual a objetos JavaScript
const user = new User(result.rows[0]);
```

- Sin validación de tipos en compilación.
- Riesgo de sintaxis SQL incorrecta.
- Mantenimiento frágil ante cambios de esquema.

Con TypeORM (Enfoque Orientado a Objetos)

```
// Métodos orientados a objetos y tipados
const user = await this.userRepo.findOne({
  where: { id: userId },
  relations: { role: true }, // JOIN tipado
});
```

- Clases TypeScript representan tablas físicas.
- Parametrización automática contra SQL Injection.
- Migraciones y control de cambios estructurado.

Infraestructura con Docker Compose y Variables .env

Variables de Entorno (.env)

```
DB_HOST=localhost
DB_PORT=5437
DB_USERNAME=postgres
DB_PASSWORD=postgres
DB_DATABASE=nest_db
```

El archivo `.env` centraliza credenciales sin exponer datos sensibles en el repositorio Git.

docker-compose.yml (PostgreSQL 18)

```yaml
services:
  db:
    image: postgres:18
    restart: always
    environment:
      POSTGRES_USER: ${DB_USERNAME}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_DB: ${DB_DATABASE}
    ports:
      - '${DB_PORT}:5432'
    volumes:
      - postgres-data:/var/lib/postgresql/data
volumes:
  postgres-data:
```

Configuración Modular con TypeORM

AppModule: TypeOrmModule.forRootAsync()

```
TypeOrmModule.forRootAsync({
  imports: [ConfigModule],
  inject: [ConfigService],
  useFactory: (cfg: ConfigService) => ({
    type: 'postgres',
    host: cfg.get<string>('DB_HOST'),
    port: cfg.get<number>('DB_PORT'),
    username: cfg.get<string>('DB_USERNAME'),
    password: cfg.get<string>('DB_PASSWORD'),
    database: cfg.get<string>('DB_DATABASE'),
    autoLoadEntities: true,
    synchronize: true, //¡Solo desarrollo!
  }),
})</string></string></string></number></string>
```

Resuelve variables de entorno de forma asíncrona y levanta el Pool de conexiones TCP global.

UsersModule: TypeOrmModule.forFeature()

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { Role } from '../roles/entities/role.entity';

@Module({
  // Registra repositorios en el contenedor local
  imports: [TypeOrmModule.forFeature([User, Role])],
  controllers: [UsersController],
  providers: [UsersService],
})
export class UsersModule {}
```

Fabrica los proveedores `Repository<User>` y `Repository<Role>` para este módulo.

Del Diagrama ER a Entidades TypeScript

Entidad ROL (Lado "Uno")

```typescript
import { Entity, PrimaryGeneratedColumn, Column, OneToMany } from 'typeorm';
import { User } from '../../users/entities/user.entity';

@Entity('roles')
export class Role {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ length: 50, unique: true })
  name: string;

  // Un rol agrupa a muchos usuarios
  @OneToMany(() => User, (user) => user.role)
  users: User[];
}
```

Correspondencia de Decoradores

Diagrama ER	TypeORM Decorator
Tabla física	`@Entity('nombre_tabla')`
Llave primaria (PK)	`@PrimaryGeneratedColumn()`
Columna estándar	`@Column({ length: 50 })`
Relación 1:N	`@OneToMany(() => Target, ...)`
Relación N:1	`@ManyToOne(() => Target, ...)`

Modelado de Relación: User y Llave Foránea

src/users/entities/user.entity.ts

```ts
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne, JoinColumn } from 'typeorm';
import { Role } from '../../roles/entities/role.entity';

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

  @Column({ length: 120, unique: true })
  email: string;

  @Column({ name: 'password_hash' })
  passwordHash: string;

  // Muchos usuarios pertenecen a un único rol
  @ManyToOne(() => Role, (r) => r.users, { onDelete: 'CASCADE' })
  @JoinColumn({ name: 'role_id' }) // Columna física
  role: Role;
}
```

La Regla de Oro de @JoinColumn:

- **Ubicación Obligatoria:** `@JoinColumn` se sitúa en la entidad del lado "muchos" (`User`), porque allí se crea la columna física `role_id` .
- **Integridad Referencial:** Previene que existan usuarios huérfanos sin rol válido en la base de datos relacional.
- **Cascada:** `onDelete: 'CASCADE'` asegura limpieza automática si se elimina el registro padre.

Universidad ICESI

Patrón Repositorio y Consultas con TypeORM

Inyección de Repositorios, Métodos CRUD y Operadores Avanzados

Universidad ICESI

undefined forFeature([User])

Token de Repositorio

Repository

undefined @InjectRepository()

Inyección Exitosa

Mecánica Interna de forFeature()

Al declarar `TypeOrmModule.forFeature([User, Role])`, NestJS orquesta tres pasos automáticos en el Contenedor IoC:

1. Fabricación del Provider

Extrae el repositorio desde la conexión activa con `dataSource.getRepository(User)`.

2. Generación del Token Único

Para resolver el type erasure de TypeScript en JS, registra el proveedor bajo el token `getRepositoryToken(User)`.

3. Habilitación de la Inyección

Permite que `@InjectRepository(User)` resuelva el token y suministre `Repository<User>` al constructor del servicio.

Universidad ICESI

Catálogo de Métodos Fundamentales de Repository

Método	Propósito	Impacto en Base de Datos (SQL)
`create(dto)`	Instancia la entidad en memoria JavaScript	**Ninguno** (no ejecuta sentencias SQL)
`save(entity)`	Persiste o actualiza el registro en la BD	Ejecuta `INSERT INTO...` o `UPDATE...`
`find(options)`	Retorna todos los registros coincidentes	Ejecuta `SELECT * FROM users WHERE...`
`findOne(options)`	Retorna el primer registro o `null`	Ejecuta `SELECT * ... LIMIT 1`
`findBy(where)`	Atajo para búsqueda simple por campos	Ejecuta `SELECT * WHERE campo = valor LIMIT 1`
`update(id, partial)`	Actualiza columnas directas por ID	Ejecuta `UPDATE users SET ... WHERE id = :id`
`delete(id)`	Elimina el registro físico por su ID	Ejecuta `DELETE FROM users WHERE id = :id`
`findAndCount(options)`	Retorna `[registros, total]` para paginación	Ejecuta consulta de datos + `COUNT(*)`

Diferencia Crucial: create() frente a save()

1. create(dto) — En Memoria

```
// 1. create() SOLO instancia en memoria
const userInstance = this.userRepo.create({
  email: 'ada@icesi.edu.co',
  passwordHash: 'hashed_password',
  role: roleInstance,
});

// ¡En este punto NO se ha ejecutado
// ningún SQL en PostgreSQL!
```

- Valida compatibilidad de tipos en TypeScript.
- Asocia entidades en memoria.
- Sincrónico (no requiere `await`).

2. save(entity) — Persistencia Física

```
// 2. save() persiste físicamente en la BD
const savedUser = await this.userRepo.save(
  userInstance
);

// Ejecuta en PostgreSQL:
// INSERT INTO users (email, password_hash, role_id)
// VALUES ('ada@...', 'hashed...', 1)
// RETURNING id;
```

- Ejecuta `INSERT` o `UPDATE` real.
- Asigna el ID autoincremental retornado.
- Asincrónico (requiere `await`).

Estrategias de Modificación: update() vs. save()

Estrategia A: update(id, partial)

```
// Ejecuta directamente UPDATE sin cargar
await this.userRepo.update(id, {
  email: 'nuevo_correo@icesi.edu.co',
});

// SQL generado:
// UPDATE users SET email = ... WHERE id = :id;
```

Ventaja: Ultra rápido para cambios puntuales. No consume memoria cargando la entidad completa.

Estrategia B: save(entity) Completo

```
// 1. Cargar la entidad primero
const user = await this.userRepo.findOneBy({ id });
if (!user) throw new NotFoundException();

// 2. Modificar propiedad y guardar
user.email = 'nuevo_correo@icesi.edu.co';
await this.userRepo.save(user);
```

Ventaja: Permite validar reglas de negocio en TypeScript, ejecutar eventos (listeners) y controlar relaciones complejas.

Consultas Declarativas con FindManyOptions y Operadores

FindManyOptions Estructurado

```
const users = await this.userRepo.find({
  select: { id: true, email: true },
  where: {
    role: { name: In(['Admin', 'Profesor']) },
    email: ILike('%@icesi.edu.co'),
  },
  relations: { role: true }, //JOIN relacional
  order: { id: 'DESC' },
  take: 10, //LIMIT
  skip: 0,  //OFFSET
});
```

Operadores Clave (typeorm)

Operador	SQL Equivalente
`ILike('%texto%')`	`ILIKE '%texto%'` (Case-Insensitive)
`Between(min, max)`	`BETWEEN min AND max`
`In([val1, val2])`	`IN ('val1', 'val2')`
`MoreThan(n) / LessThan(n)`	`> n / < n`
`IsNull() / Not(cond)`	`IS NULL / NOT (...)`