Universidad Icesi

Repositorios y Providers en NestJS

Semana 5: Arquitectura por Capas, Inversión de Control y Persistencia Relacional con TypeORM

Facultad de Ingeniería, Diseño y Ciencias Aplicadas — Universidad Icesi

Universidad ICESI

- Arquitectura e IoC
- Módulos y Providers
- Módulos TypeORM
- TypeORM Queries
- Buenas Prácticas

Ruta de Aprendizaje — Semana 5

Arquitectura por Capas e IoC

Niveles 1 y 2 de abstracción, acoplamiento rígido con `new` vs. IoC y decorador `@Injectable()`.

Modularidad y Encapsulación

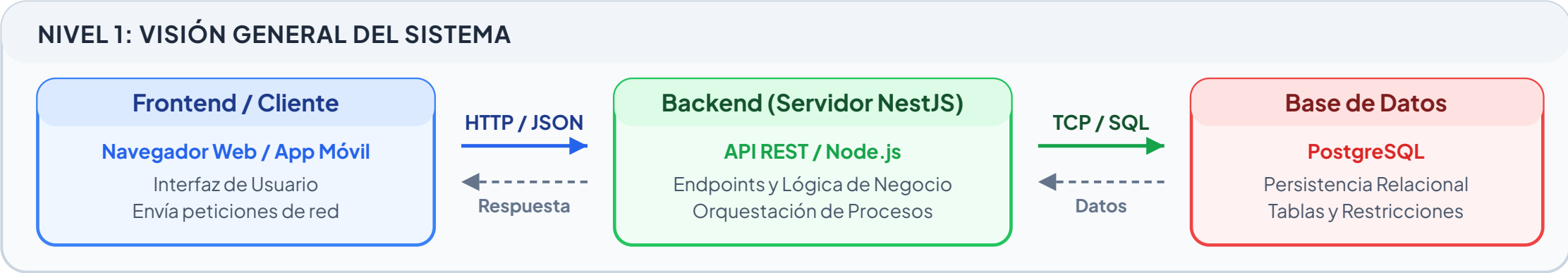
Estructura de `@Module()`, fronteras de visibilidad privadas y depuración de errores.

Módulos TypeORM y Repositorios

Configuración `forRoot` vs `forFeature`, modelo 1:N Autores-Libros e inyección con `@InjectRepository()`.

Arquitectura por Capas: Nivel 1 (Visión General)

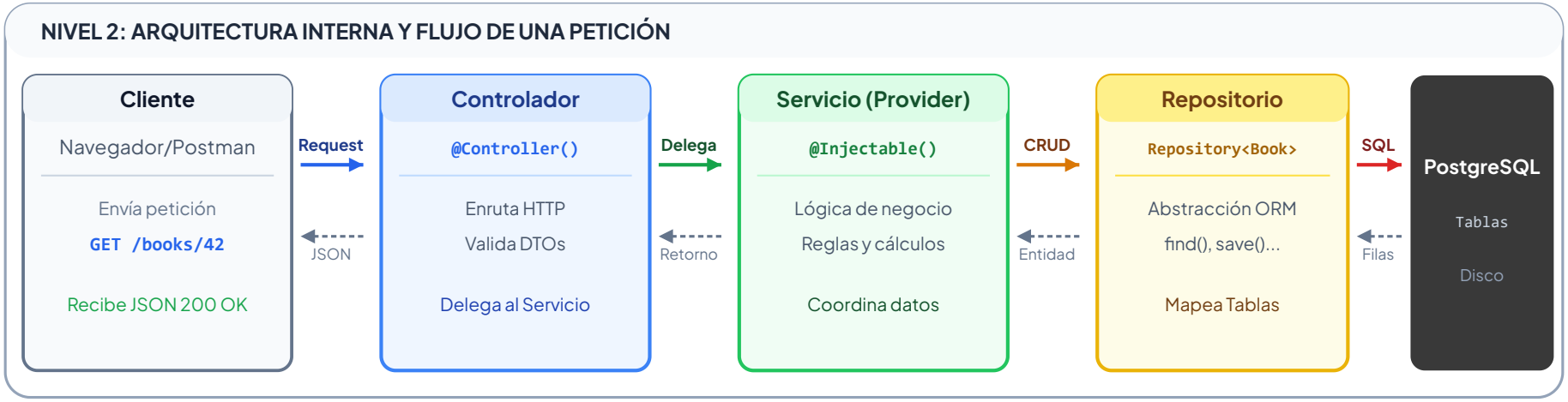
A nivel macro, el sistema separa la experiencia de usuario de la persistencia física mediante un intermediario de servicios backend:



Conclusión Nivel 1: El cliente nunca se conecta a PostgreSQL de forma directa; el backend actúa como guardián de seguridad, aplicando autenticación, validaciones y reglas de negocio.

Arquitectura por Capas: Nivel 2 (Flujo Interno en NestJS)

Al hacer zoom dentro del servidor NestJS, observamos la ruta precisa que recorre cada petición a través de las capas internas:



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.

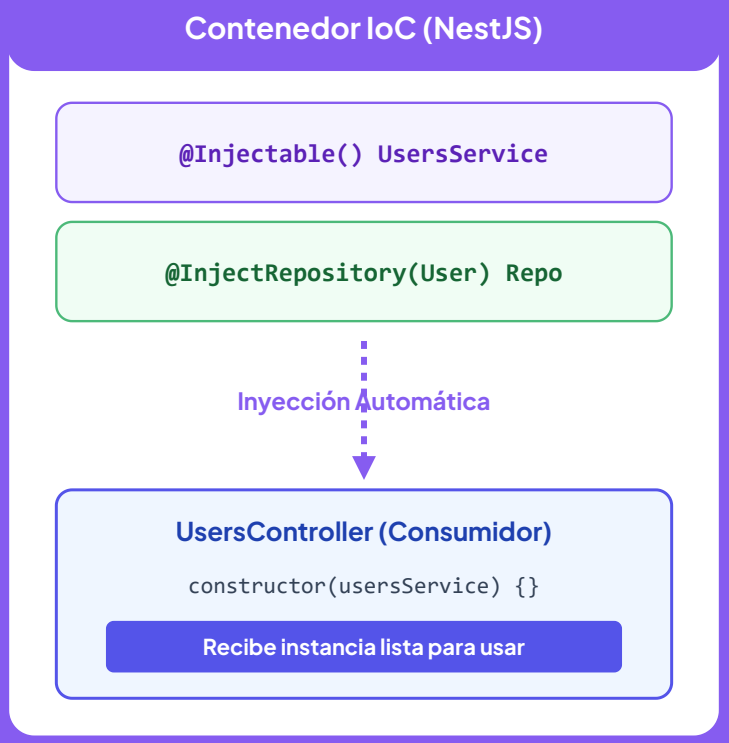
Desventajas Críticas en Producción:

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

Universidad icesi

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

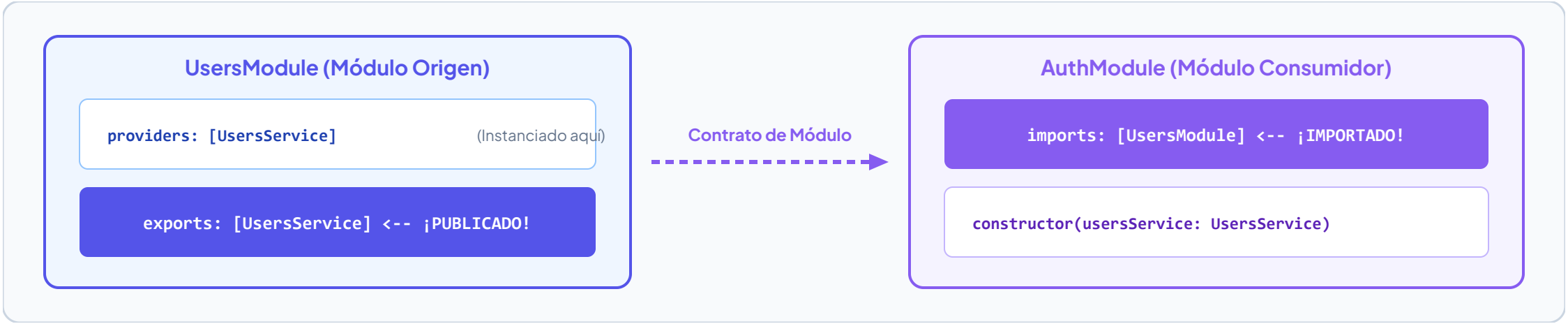
EXPORTS: [...]

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**.



Regla de Oro: Si `UsersModule` no lista `UsersService` en su arreglo `exports`, ningún otro módulo podrá inyectarlo, generando un error fatal en el arranque.

Universidad ICESI

Implementación Modular en NestJS

src/users/users.module.ts

```
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
  // Módulos que aportan dependencias
  imports: [],

  // Controladores que escuchan rutas HTTP
  controllers: [UsersController],

  // Providers gestionados por el contenedor
  providers: [UsersService],

  // Hace visible a UsersService para otros módulos
  exports: [UsersService],
})
export class UsersModule {}
```

Principios de Diseño Modular:

- **Alta Cohesión:** Cada módulo agrupa únicamente los archivos correspondientes a un mismo dominio de negocio (usuarios, libros, pedidos).
- **Bajo Acoplamiento:** Los módulos solo exponen lo estrictamente necesario a través de `exports`, manteniendo su lógica interna privada.
- **Reutilización Limpia:** Cualquier módulo del sistema puede reutilizar `UsersService` sin duplicar instancias en memoria.

Universidad ICESI

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

Error Crítico: "Nest can't resolve dependencies"

Es el error más frecuente en NestJS. Ocurre cuando un componente solicita un provider en su constructor pero el framework no sabe cómo construirlo.

```
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

Patrón Repositorio y TypeORM

Abstracción de Datos y Modelado Relacional

Configuración de Módulos con TypeORM

1. Global: TypeOrmModule.forRoot()

```ts
// src/app.module.ts
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: 'localhost',
      port: 5432,
      username: 'icesi_user',
      password: 'secret_password',
      database: 'icesi_db',
      autoLoadEntities: true,
      synchronize: true, // ¡Solo en desarrollo!
    }),
  ],
})
export class AppModule {}
```

Establece el **Pool de Conexiones TCP** con PostgreSQL a nivel de toda la aplicación. Se invoca una sola vez en el módulo raíz.

2. Por Dominio: TypeOrmModule.forFeature()

```ts
// src/books/books.module.ts
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Book } from './entities/book.entity';
import { Author } from '../authors/entities/author.entity';
import { BooksService } from './books.service';

@Module({
  // Registra los repositorios en este módulo
  imports: [TypeOrmModule.forFeature([Book, Author])],
  controllers: [BooksController],
  providers: [BooksService],
  exports: [BooksService],
})
export class BooksModule {}
```

Indica qué entidades necesita este módulo específico y **fabrica dinámicamente los Repositorios** en el contenedor IoC.

Universidad ICESI

forFeature([Book])

Token de Repositorio

Repository

@InjectRepository()

Inyección Exitosa

¿Cómo Funciona forFeature() por Debajo?

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

1. Fabricación del Provider

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

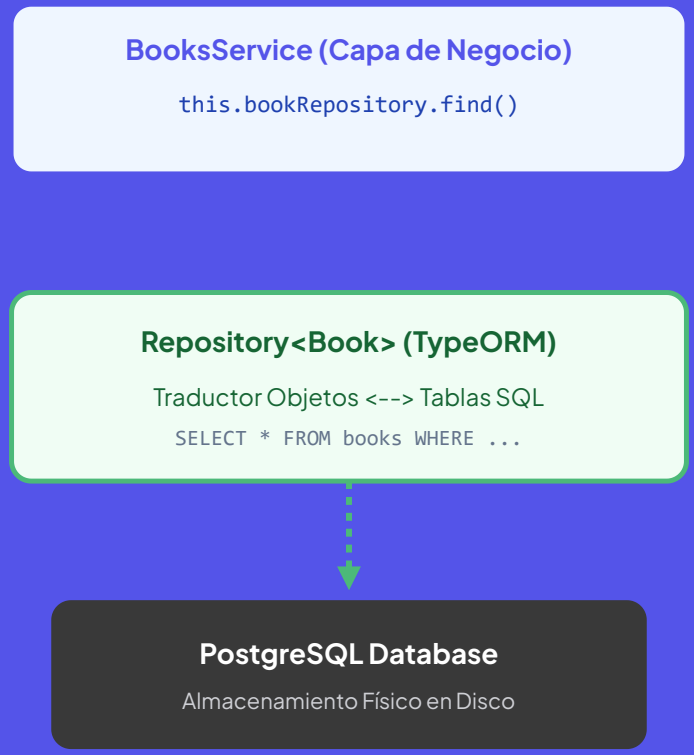
2. Generación del Token Único

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

3. Habilitación de la Inyección

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

Universidad ICESI



El Patrón Repositorio en NestJS

El **Patrón Repositorio** actúa como un mediador entre la lógica de negocio y la base de datos relacional. En lugar de escribir SQL en los servicios, interactuamos con métodos fuertemente tipados.

Aislamiento Tecnológico:

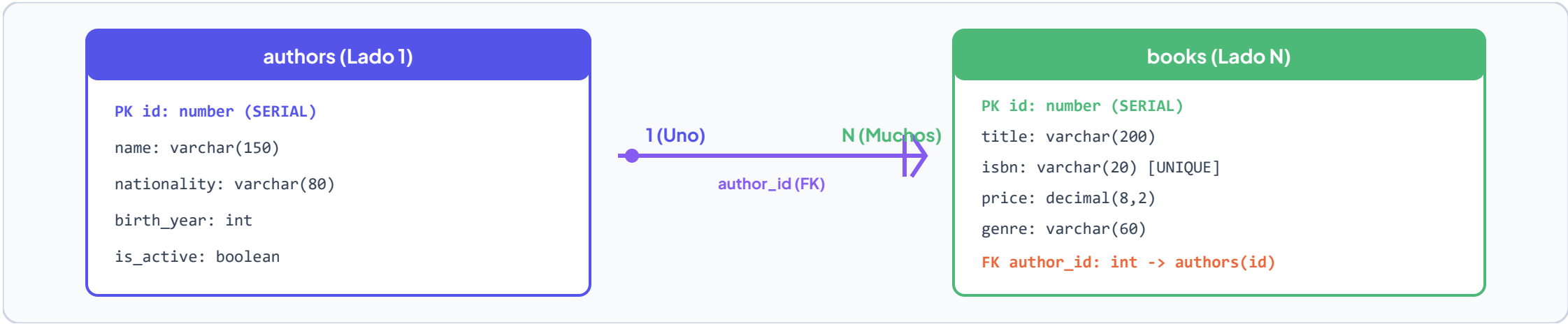
El servicio desconoce el motor físico (PostgreSQL, MySQL, SQLite); solo interactúa con contratos tipados de TypeORM.

Seguridad contra SQL Injection:

TypeORM parametriza automáticamente todas las consultas y valores antes de enviarlos a la base de datos.

Modelo de Dominio Relacional: Autores y Libros (1:N)

Un **Autor** puede registrar muchos libros en el catálogo, pero cada **Libro** pertenece a un único autor. La clave foránea `author_id` reside físicamente en la tabla `books`.



Modelado de Entidad: Author (1:N)

src/authors/entities/author.entity.ts

```typescript
import { Entity, PrimaryGeneratedColumn, Column, OneToMany } from 'typeorm';
import { Book } from '../../books/entities/book.entity';

@Entity('authors')
export class Author {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ length: 150 })
  name: string;

  @Column({ length: 80, nullable: true })
  nationality: string;

  @Column({ name: 'birth_year', type: 'int', nullable: true })
  birthYear: number;

  @Column({ name: 'is_active', default: true })
  isActive: boolean;

  // Relación: Un autor tiene muchos libros
  @OneToMany(() => Book, (book) => book.author)
  books: Book[];
}
```

Análisis de Decoradores Relacionales:

- **@Entity('authors'):** Mapea la clase TypeScript a la tabla física `authors` de PostgreSQL.
- **@OneToMany(() => Book, ...):** Declara el extremo "uno a muchos". Especifica cómo llegar desde el autor hacia sus libros.
- **Sin @JoinColumn aquí:** En el lado `@OneToMany` no se crea ninguna columna física en la tabla; es solo una relación virtual en TypeScript.

Modelado de Entidad: Book y Llave Foránea

src/books/entities/book.entity.ts

```typescript
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne, JoinColumn } from 'typeorm';
import { Author } from '../../authors/entities/author.entity';

@Entity('books')
export class Book {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ length: 200 })
  title: string;

  @Column({ unique: true, length: 20 })
  isbn: string;

  @Column({ type: 'decimal', precision: 8, scale: 2 })
  price: number;

  // Relación: Muchos libros pertenecen a un autor
  @ManyToOne(() => Author, (a) => a.books, { onDelete: 'CASCADE' })
  @JoinColumn({ name: 'author_id' }) // Columna física
  author: Author;
}
```

La Regla de Oro de @JoinColumn:

- **Ubicación Obligatoria:** `@JoinColumn` va **SIEMPRE** en la entidad del lado "muchos" (`Book`), porque allí es donde reside físicamente la columna `author_id` .
- **Borrado en Cascada:** Al configurar `{ onDelete: 'CASCADE' }` , si se elimina un autor en la base de datos, todos sus libros asociados se eliminarán automáticamente.
- **Integridad Referencial:** Previene registros huérfanos sin autor válido en la base de datos relacional.

Inyección del Repositorio en el Servicio

src/books/books.service.ts

```ts
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Book } from './entities/book.entity';
import { Author } from '../authors/entities/author.entity';

@Injectable()
export class BooksService {
  constructor(
  @InjectRepository(Book)
  private readonly bookRepo: Repository<Book>,

  @InjectRepository(Author)
  private readonly authorRepo: Repository<Author>,
 ) {}
}
```

¿Por qué se requiere @InjectRepository()?

- **Type Erasure en TypeScript:** En tiempo de ejecución JS, `Repository<Book>` pierde su genérico y solo queda `Repository`.
- **Resolución de Token:** `@InjectRepository(Book)` le indica a NestJS que busque el token específico generado por `forFeature()`.
- **Propiedad readonly:** Garantiza que la referencia al repositorio no pueda ser sobreescrita accidentalmente dentro de la clase.

Universidad ICESI

Consultas Avanzadas y Operadores

Filtros Declarativos, Relaciones y Paginación

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 books WHERE...`
`findOne(options)`	Retorna el primer registro o `null`	Ejecuta `SELECT * ... LIMIT 1`
`findOneBy(where)`	Atajo para búsqueda simple por campos	Ejecuta `SELECT * WHERE campo = valor LIMIT 1`
`update(id, partial)`	Actualiza columnas directas por ID	Ejecuta `UPDATE books SET ... WHERE id = :id`
`delete(id)`	Elimina el registro físico por su ID	Ejecuta `DELETE FROM books 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 bookInstance = this.bookRepo.create({
  title: 'Cien años de soledad',
  isbn: '978-0307474728',
  price: 45000,
  author: authorInstance,
});

// ¡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 savedBook = await this.bookRepo.save(
  bookInstance
);

// Ejecuta en PostgreSQL:
// INSERT INTO books (title, isbn, price, author_id)
// VALUES ('Cien años...', '978...', 45000, 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.bookRepo.update(id, {
  price: 49900,
});

// SQL generado:
// UPDATE books SET price = 49900 WHERE id = 1;
```

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

Estrategia B: save(entity) Completo

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

// 2. Modificar propiedad y guardar
book.price = 49900;
await this.bookRepo.save(book);
```

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

Consultas Declarativas con FindManyOptions y Operadores

FindManyOptions Estructurado

```
const books = await this.bookRepo.find({
  select: { id: true, title: true, price: true },
  where: {
    genre: In(['Ciencia Ficción', 'Novela']),
    price: Between(20, 70),
    title: ILike('%guía%'),
  },
  relations: { author: true }, // JOIN relacional
  order: { price: 'ASC' },
  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 (...)`

Universidad icesi

Principios Clave para Backend en NestJS

INVERSIÓN DE CONTROL

Jamás instancies servicios con `new`. Declara las dependencias en el constructor y deja que NestJS resuelva el grafo singleton.

Desacoplamiento total

ENCAPSULACIÓN MODULAR

Los providers son privados por defecto. Expón solo lo indispensable mediante el arreglo `exports` de cada módulo.

Módulos herméticos

REPOSITORIOS TIPADOS

Registra con `TypeOrmModule.forFeature()` e inyecta con `@InjectRepository()` para consultas seguras y libres de SQL manual.

TypeORM seguro

MANEJO DE EXCEPCIONES

Comunica errores de negocio mediante excepciones HTTP nativas (`NotFoundException` , `ConflictException`) desde los servicios.

REST Estándar