Universidad Icesi

Computación en Internet 3 — Semana 6

Controladores REST, Pipes de Validación & Testing con Jest

Arquitectura Backend Modular con NestJS

Facultad de Ingeniería — Universidad Icesi

Universidad ICESI

Agenda y Tres Pilares de la Semana 6

1. Controladores REST

Capa de Entrada y Enrutamiento

- Enrutamiento declarativo con `@Controller()`.
- Mapeo con `@Param`, `@Query` y `@Body`.
- Manejo de excepciones con Exception Filters.

2. Pipes y Validadores

Transformación y Sanitización

- Pipes integrados (`ParseIntPipe`, `ParseUUID`).
- Validación declarativa con `class-validator`.
- Implementación de Custom Pipes (`PositiveInt`).

3. Testing Automatizado

JUnit/Jest & Supertest

- Módulo de prueba con `createTestingModule`.
- Mocks aislados de TypeORM con `jest.fn()`.
- Pruebas End-to-End simulando HTTP con Supertest.

Universidad ICESI

01. Controladores en NestJS

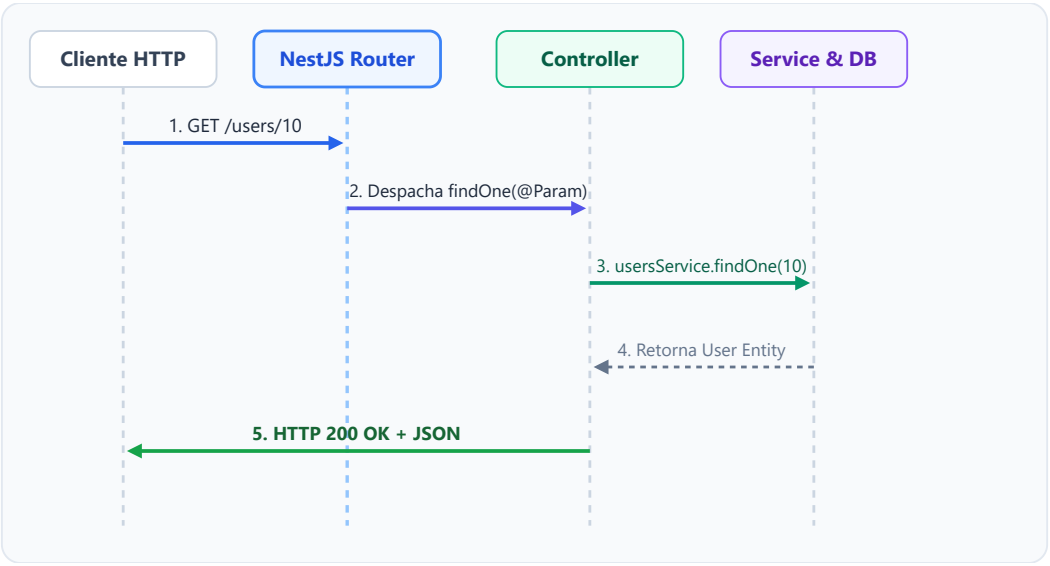
Capa de entrada, despacho de rutas y ciclo de vida de peticiones HTTP

¿Qué es un Controlador en NestJS?

Punto de Entrada al Sistema

En la arquitectura en capas de NestJS, el **Controlador** es la frontera externa que conecta clientes HTTP con la lógica de negocio interna:

- **Enrutamiento Declarativo:** La clase se anota con `@Controller('prefix')` para agrupar rutas bajo una misma URL base.
- **Extracción y Mapeo:** Interpreta parámetros, query strings y cuerpos JSON mediante decoradores específicos.
- **Delegación:** Invoca métodos en los Servicios inyectados y retorna respuestas tipadas al cliente.



Mapeo de Parámetros y Decoradores HTTP

@PARAM('ID')

Captura parámetros de segmento en la URL:

```
@Get(':id')
findOne(@Param('id') id: string)
```

Ejemplo: `/users/42` -> `id = "42"`.

@QUERY('FILTER')

Captura query strings opcionales o filtros:

```
@Get()
findAll(@Query('role') role: string)
```

Ejemplo: `/users?role=admin`.

@BODY()

Extrae el payload enviado en la petición:

```
@Post()
create(@Body() dto: CreateUserDto)
```

Asocia el cuerpo JSON a una clase DTO tipada.

@HEADERS('AUTH')

Lee cabeceras HTTP de autenticación:

```
@Get('me')
getMe(@Headers('authorization') token)
```

Captura tokens Bearer JWT o API keys.

Universidad ICESI

@Controller Prefix

@HttpCode Custom

Verbos REST

Rutas, Verbos HTTP y Códigos de Respuesta

NestJS asocia métodos a verbos REST estándar y permite personalizar los códigos de estado devueltos:

```
@Controller('users')
export class UsersController {
 @Post()
 @HttpCode(HttpStatus.CREATED) // 201
 create(@Body() dto: CreateUserDto) {
  return this.usersService.create(dto);
 }

 @Delete(':id')
 @HttpCode(HttpStatus.NO_CONTENT) // 204
 remove(@Param('id') id: string) {
  return this.usersService.remove(+id);
 }
}
```

Rutas con Comodín (Wildcards)

Usa `@Get('*')` o `@Get('download/*/files')` para capturar patrones dinámicos en subrutas.

Principio: Controladores Delgados (Thin Controllers)

Separación Estricta de Responsabilidades

Un controlador **nunca** debe contener lógica de negocio compleja ni ejecutar consultas a la base de datos:

- ⚠ **Controlador Gordo (Antipatrón):** Cálculos, transacciones y consultas SQL mezcladas en el manejador. Imposible de testear.
- ✓ **Controlador Delgado (Recomendado):** Solo extrae datos, valida, delega al servicio y retorna el resultado.
- ✓ **Reutilización:** La lógica en los servicios puede ser invocada por controladores, tareas programadas (cron) o microservicios.

CONTROLADOR DELGADO Y LIMPIO

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

 @Post()
 async create(@Body() dto: CreateUserDto) {
  // ✅ Delegación limpia en una línea
  return await this.usersService.create(dto);
 }

 @Get(':id')
 async findOne(@Param('id', ParseIntPipe) id: number) {
  // ✅ El pipe valida y el servicio busca
  return await this.usersService.findOne(id);
 }
}
```

Universidad ICESI

Manejo de Respuestas y Excepciones HTTP

Exception Filters Integrados

NestJS intercepta automáticamente cualquier excepción que herede de `HttpException` y responde con un JSON estándar:

Excepción	Status	Uso Habitual
`BadRequestException`	400	Fallo de validación en DTO
`UnauthorizedException`	401	Falta token o credenciales
`ForbiddenException`	403	Permisos insuficientes
`NotFoundException`	404	Recurso no encontrado
`ConflictException`	409	Email/clave única duplicada

LANZAMIENTO Y RESPUESTA JSON

```
// Lanzada en el servicio
throw new NotFoundException(
  'El usuario no existe',
  { description: 'ID_NOT_FOUND' }
);
```

RESPUESTA AUTOMÁTICA DEL SERVIDOR

```
{
  "statusCode": 404,
  "message": "El usuario no existe",
  "error": "Not Found"
}
```

Excepciones de Dominio Personalizadas

Estandarización de Errores

Para no repetir mensajes ni estructuras de error en múltiples servicios, se crean clases personalizadas que extiendan de excepciones HTTP base:

- **Mensajes Centralizados:** Se define el texto de error en un único lugar del código.
- **Códigos Internos:** Permite incluir un código semántico para el frontend (ej. `USER_NOT_FOUND`).
- **Barrel Files:** Se exportan desde `src/common/exceptions/index.ts`.

USERNOTFOUNDEXCEPTION.TS

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

export class UserNotFoundException extends NotFoundException {
 constructor(userId: number, code = 'USER_NOT_FOUND') {
  super({
   statusCode: 404,
   error: 'User Not Found',
   message: `El usuario con ID ${userId} no fue encontrado.`,
   code,
  });
 }
}
```

```ts
// Uso en el servicio:
throw new UserNotFoundException(id);
```

Implementación de UsersController Completo

Operaciones CRUD con NestJS

El controlador implementa los 5 métodos estándar REST inyectando el servicio por constructor:

- ✓ `POST /users`: Crea un nuevo usuario validando `CreateUserDto`.
- ✓ `GET /users`: Lista usuarios con filtros opcionales `@Query()`.
- ✓ `GET /users/:id`: Busca por ID usando `ParseIntPipe`.
- ✓ `PATCH /users/:id`: Modifica parcialmente con `UpdateUserDto`.
- ✓ `DELETE /users/:id`: Elimina el registro por ID.

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

 @Post()
 create(@Body() createUserDto: CreateUserDto) {
  return this.usersService.create(createUserDto);
 }

 @Get()
 findAll(@Query('role') role?: string) {
  return this.usersService.findAll(role);
 }

 @Get(':id')
 findOne(@Param('id', ParseIntPipe) id: number) {
  return this.usersService.findOne(id);
 }

 @Patch(':id')
 update(@Param('id', ParseIntPipe) id: number, @Body() dto: UpdateUserDto) {
  return this.usersService.update(id, dto);
 }

 @Delete(':id')
 remove(@Param('id', ParseIntPipe) id: number) {
  return this.usersService.remove(id);
 }
}
```

Universidad ICESI

Buenas Prácticas en Controladores

TIPADO FUERTE CON DTOS

Nunca usar `any` en entradas:

Define clases DTO para cada verbo (`CreateUserDto` , `UpdateUserDto`) garantizando autocompletado y validación.

EVITAR @RES() NATIVO

Preservar la abstracción:

Usar `@Res() res: Response` rompe la compatibilidad con Fastify y complica las pruebas unitarias. Retorna objetos puros.

RUTAS SEMÁNTICAS

Convención REST estándar:

Usa sustantivos en plural (`/users` , `/orders`) en lugar de verbos (`/getUsers` , `/createOrder`).

MÉTODOS ASÍNCRONOS

Gestión de Promises:

Retorna directamente la Promesa del servicio o usa `async/await` para que Nest resuelva los valores automáticamente.

Universidad ICESI

02. Pipes y Validación de Datos

Transformación de tipos, sanitización y validación declarativa con DTOs

¿Qué es un Pipe? Interfaz PipeTransform

Propósito Dual del Pipe

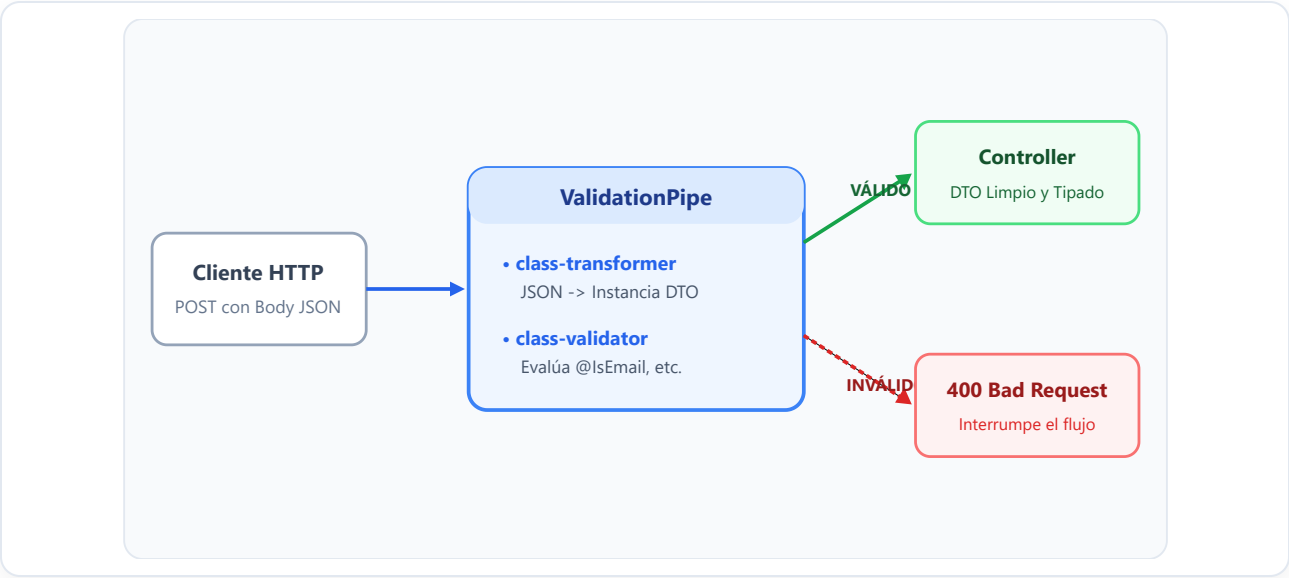
Un Pipe es una clase `@Injectable()` que intercepta los argumentos de la solicitud antes de que el controlador los reciba:

1. Transformación de Datos

Convierte strings en enteros (`"10" -> 10`), parsea cadenas en booleanos o transforma objetos JSON en instancias DTO.

2. Validación de Reglas

Comprueba si los datos cumplen restricciones. Si fallan, lanza `BadRequestException` (400) e interrumpe la petición de inmediato.



Pipes Integrados en NestJS

PARSEINTPIPE & FLOAT

Conversión a números primitivos:

```
@Get(':id')
findOne(@Param('id', ParseIntPipe) id)
```

Garantiza que `typeof id === 'number'`.

PARSEUUIDPIPE

Validación de identificadores:

```
@Param('uuid', new ParseUUIDPipe({version:'4'}))
```

Rechaza con 400 si no tiene formato UUID v4.

PARSEENUMPIPE

Validación contra Enums:

```
@Param('role', new ParseEnumPipe(RoleEnum))
```

Comprueba pertenencia al enum de TypeScript.

DEFAULTVALUEPIPE

Valores por omisión:

```
@Query('page', new DefaultValuePipe(1), ParseIntPipe
```

Asigna `1` si el cliente no envió el query param.

Ámbitos de Aplicación de Pipes

4 Niveles de Granularidad

- **1. Nivel de Parámetro:** Se aplica a un argumento específico: `@Param('id', ParseIntPipe)`.
- **2. Nivel de Método:** Se aplica con `@UsePipes(new ValidationPipe())` sobre un `@Post()`.
- **3. Nivel de Controlador:** Aplica a todas las rutas de la clase: `@UsePipes(...)` sobre `@Controller()`.
- **4. Nivel Global:** Aplica a **toda la aplicación** en `main.ts`. Es el estándar recomendado para validación.

CONFIGURACIÓN GLOBAL EN MAIN.TS

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

async function bootstrap() {
 const app = await NestFactory.create(AppModule);

 // Valida automáticamente el 100% de los DTOs entrantes
 app.useGlobalPipes(
  new ValidationPipe({
   whitelist: true,
   forbidNonWhitelisted: true,
   transform: true,
  }),
 );

 await app.listen(3000);
}
```

Universidad ICESI

@IsEmail() Check

@MinLength & Regex

class-validator

Validación Declarativa con class-validator

Con `class-validator` cada propiedad del DTO se decora con reglas declarativas legibles:

```
export class CreateUserDto {
 @IsString({ message: 'El nombre debe ser texto' })
 @MinLength(3, { message: 'Mínimo 3 letras' })
 username: string;

 @IsEmail({}, { message: 'Correo inválido' })
 email: string;

 @MinLength(8, { message: 'Contraseña de 8+ chars' })
 @Matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d).*$/, {
  message: 'Debe tener mayúscula, minúscula y número',
 })
 password: string;

 @IsOptional()
 @MaxLength(200)
 bio?: string;
}
```

Configuración Segura de ValidationPipe

Protección contra Inyecciones

Un `ValidationPipe` sin opciones puede ser vulnerable a inyecciones de parámetros no deseados (*Mass Assignment*):

whitelist: true

Descarta silenciosamente cualquier propiedad en el JSON que no esté explícitamente declarada en el DTO.

forbidNonWhitelisted: true

Lanza un error 400 inmediato si el cliente envía propiedades desconocidas (ej. `isAdmin: true`).

TRANSFORM: TRUE

Convierte el objeto plano de JavaScript (POJO) en una verdadera instancia de la clase DTO mediante `class-transformer`:

```
app.useGlobalPipes(
 new ValidationPipe({
  whitelist: true,
  forbidNonWhitelisted: true,
  transform: true,
  transformOptions: {
   enableImplicitConversion: true,
  },
 }),
);
```

Resultado: Los métodos del DTO y decoradores como `@Type()` funcionan en tiempo de ejecución.

Creación de un Pipe Personalizado: PositiveIntPipe

Implementar PipeTransform

Cuando los pipes nativos no satisfacen una regla de negocio específica, creamos un pipe propio implementando `PipeTransform<T, R>`:

- **ArgumentMetadata:** Permite conocer el origen (`param`) y nombre del campo.
- **Validación Estricta:** Verifica que sea un número entero y además estrictamente positivo (`val > 0`).
- **Lanzamiento 400:** Dispara `BadRequestException` si es menor o igual a cero.

```
@Injectable()
export class PositiveIntPipe
  implements PipeTransform<string, number> {

 transform(value: string, metadata: ArgumentMetadata): number {
  const val = parseInt(value, 10);

  if (isNaN(val)) {
   throw new BadRequestException(
    `El parámetro '${metadata.data}' debe ser un entero válido.`
   );
  }

  if (val <= 0) {
   throw new BadRequestException(
    `El parámetro '${metadata.data}' debe ser mayor a 0.`
   );
  }

  return val;
 }
}
```

Universidad ICESI

Beneficios de la Validación con Pipes

PRINCIPIO FAIL-FAST

Rechazo Inmediato

Las peticiones con datos corruptos se cancelan en la frontera, evitando consumir CPU, memoria o conexiones a base de datos.

CONTROLADORES LIMPIOS

Cero if/else Manuales

Elimina la necesidad de comprobar manualmente si cada campo está vacío o mal formateado dentro de la función del controlador.

SEGURIDAD DE DOMINIO

Protección Anti-Inyección

`forbidNonWhitelisted` impide que usuarios maliciosos inyecten propiedades de rol o permisos en payloads JSON.

REUTILIZACIÓN MODULAR

Componentes Portables

Un pipe como `PositiveIntPipe` o `ValidationPipe` puede aplicarse en cualquier módulo del backend.

Universidad ICESI

03. Testing Automatizado en NestJS

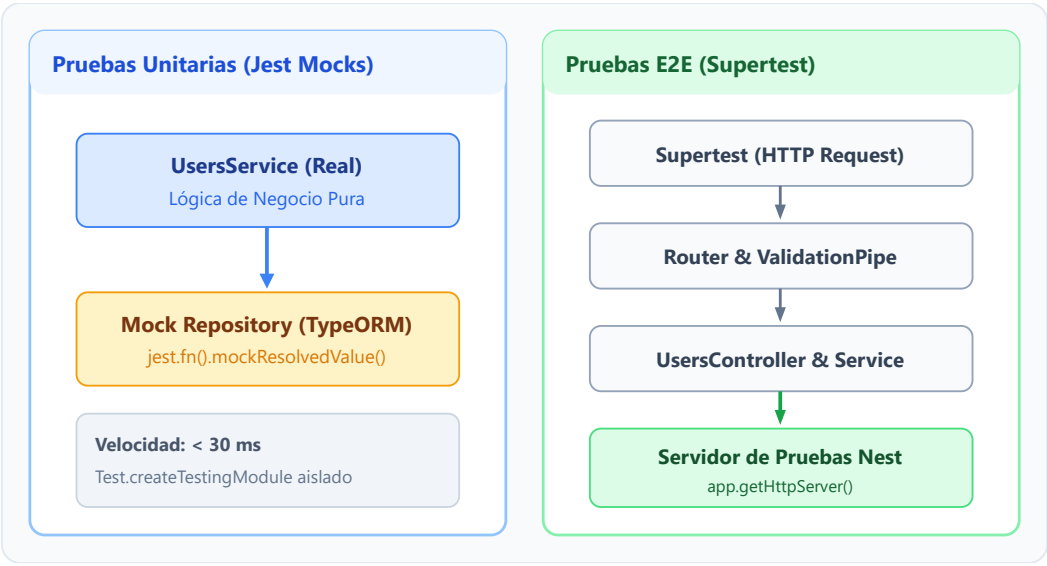
Pruebas unitarias con Jest, simulación con Test.createTestingModule y pruebas E2E

Pruebas Unitarias vs. Pruebas E2E

Estrategia de Pruebas en NestJS

Criterio	Pruebas Unitarias	Pruebas E2E
Alcance	1 clase aislada	Flujo HTTP completo
Entorno	Node.js puro con Jest	`app.getHttpServer()`
Dependencias	Mocks con `jest.fn()`	Módulos + BD de pruebas
Velocidad	Milisegundos (< 30 ms)	Segundos por suite
Objetivo	Lógica y excepciones	Rutas, pipes e integración

Inyección de Dependencias: La arquitectura de NestJS facilita el reemplazo de cualquier provider por un mock.



Universidad ICESI

Jest Test Runner

npm run test:cov

Supertest E2E

Herramientas y Comandos con Jest

NestJS incluye Jest preconfigurado con scripts de terminal en el archivo `package.json`:

```
# 1. Ejecutar todas las pruebas unitarias
npm run test

# 2. Modo observador (Watch Mode ante cambios)
npm run test:watch

# 3. Ejecutar un archivo específico
npm test users.service.spec.ts

# 4. Ejecutar pruebas de integración End-to-End
npm run test:e2e

# 5. Generar reporte visual de cobertura
npm run test:cov
```

Los reportes HTML se generan automáticamente en la carpeta `coverage/`.

Métricas de Cobertura y coverageThreshold

% STATEMENTS

Instrucciones

Porcentaje de sentencias de código JavaScript/TypeScript visitadas y ejecutadas por los tests.

% BRANCHES

Ramas Condicionales

Evalúa si se probaron ambos caminos en bifurcaciones (`if` , `else` , ternarios `?:` y `switch`).

% FUNCTIONS

Métodos y Funciones

Comprueba qué proporción de métodos de las clases fueron invocados al menos una vez.

COVERAGETHRESHOLD

Umbrales en CI/CD

Configuración en `package.json` que hace fallar el build si la cobertura global cae por debajo de un umbral (ej. 85%).

El Módulo de Pruebas: Test.createTestingModule

Contenedor IoC en Memoria

`@nestjs/testing` provee la clase `Test` para construir un módulo aislado donde inyectamos mocks en lugar de dependencias reales:

- **getRepositoryToken(User):** Resuelve el token de inyección interno que usa `@InjectRepository(User)`.
- **useValue:** Provee el objeto mock con las funciones simuladas.
- **module.get:** Extrae las instancias compiladas listas para usar en cada `it()`.

CONFIGURACIÓN EN BEFOREEACH()

```
beforeEach(async () => {
 const module: TestingModule = await Test.createTestingModule({
  providers: [
   UsersService,
   {
    provide: getRepositoryToken(User),
    useValue: mockRepository, // Mock del repositorio TypeORM
   },
   {
    provide: RolesService,
    useValue: mockRolesService, // Mock de servicio externo
   },
  ],
 }).compile();

 service = module.get<UsersService>(UsersService);
 repo = module.get(getRepositoryToken(User));
});
```

Simulación con jest.fn() en Mocks

Programar Respuestas Ficticias

En pruebas unitarias, los mocks responden inmediatamente sin tocar el disco ni la red:

- ✓ `mockReturnValue(val)` : Para métodos síncronos (ej. `repo.create()`).
- ✓ `mockResolvedValue(val)` : Para métodos asíncronos que retornan Promesas (ej. `repo.save()`).
- ✓ `toHaveBeenCalledWith(...)` : Verifica que los argumentos enviados coincidan exactamente.

DEFINICIÓN DEL OBJETO MOCK

```
const mockRepository = {
 create: jest.fn(),
 save: jest.fn(),
 find: jest.fn(),
 findOne: jest.fn(),
 delete: jest.fn(),
};

// Programación de comportamiento en el test:
mockRepository.create.mockReturnValue(dto);
mockRepository.save.mockResolvedValue({ id: 1, ...dto });

// Verificación de interacciones:
expect(mockRepository.save).toHaveBeenCalledTimes(1);
```

Prueba Unitaria Completa: UsersService.spec.ts

CASO 1: CREACIÓN EXITOSA

```
it('debe crear y persistir un usuario', async () => {
 const dto = { username: 'ana', roleName: 'ADMIN' };
 mockRolesService.findByName.mockResolvedValue({ id: 1, name: 'ADMIN' });
 mockRepository.create.mockReturnValue(dto);
 mockRepository.save.mockResolvedValue({ id: 10, ...dto });

 const result = await service.create(dto as any);

 expect(result).toHaveProperty('id', 10);
 expect(mockRepository.save).toHaveBeenCalled();
});
```

CASO 2: EXCEPCIÓN ANTE ROL INEXISTENTE

```
it('debe lanzar RoleNotFoundException', async () => {
 const dto = { username: 'ana', roleName: 'SUPER_ADMIN' };
 mockRolesService.findByName.mockResolvedValue(null);

 // Comprueba que la Promesa sea rechazada con el tipo esperado
 await expect(service.create(dto as any))
  .rejects.toThrow(RoleNotFoundException);

 // Verifica que NUNCA se intentó guardar en la BD
 expect(mockRepository.save).not.toHaveBeenCalled();
});
```

Pruebas End-to-End (E2E) con Supertest

Validar el Stack Completo

Las pruebas E2E emulan peticiones HTTP reales enviadas al servidor NestJS, validando rutas, pipes y respuestas:

- **INestApplication:** Se inicializa la app de pruebas con `await app.init()`.
- **Supertest:** Envía solicitudes a `app.getHttpServer()`.
- **Aserciones HTTP:** Comprueba `.expect(200)` o `.expect(201)` y valida la estructura JSON de salida.

USERS.E2E-SPEC.TS

```
import * as request from 'supertest';

describe('UsersController (e2e)', () => {
 let app: INestApplication;

 beforeAll(async () => {
  const moduleFixture = await Test.createTestingModule({
   imports: [AppModule],
  }).compile();

  app = moduleFixture.createNestApplication();
  app.useGlobalPipes(new ValidationPipe({ whitelist: true }));
  await app.init();
 });

 it('/users (POST) -> 201 Created', () => {
  return request(app.getHttpServer())
   .post('/users')
   .send({ username: 'carlos', email: 'carlos@icesi.edu.co' })
   .expect(201)
   .expect((res) => {
    expect(res.body).toHaveProperty('id');
   });
 });
});
```

Universidad ICESI

Cheatsheet de Testing en NestJS

ASERCIONES DE VALOR

- `expect(a).toBe(b)`
- `expect(a).toEqual(b)`
- `expect(x).toBeDefined()`
- `expect(x).toHaveProperty('id')`

PROMESAS Y ERRORES

- `expect(p).resolves.toBe(v)`
- `expect(p).rejects.toThrow(Error)`
- Valida excepciones asíncronas en servicios.

VERIFICACIÓN DE MOCKS

- `expect(fn).toHaveBeenCalled()`
- `expect(fn).toHaveBeenCalledTimes(1)`
- `expect(fn).toHaveBeenCalledWith(x)`

CICLO DE VIDA DE PRUEBAS

- `beforeEach()`: Limpia estado.
- `beforeAll()` / `afterAll()`
- `jest.clearAllMocks()`