Implementación de Jest con NestJS

Unit Testing y Desarrollo Dirigido por Pruebas (TDD)

Informe Técnico

Fecha: Septiembre 2025

Dirigido a: Equipo de desarrollo y docente

Equipo: Tralaleritos

Universidad Mayor de San Simón

Facultad de Ciencias y Tecnología
Carrera de Ingeniería en Informática

Índice

1 Resumen Ejecutivo

Este informe presenta la implementación de pruebas unitarias utilizando Jest en el framework NestJS, abordando dos enfoques fundamentales: la implementación tradicional de Unit Testing y la metodología de Desarrollo Dirigido por Pruebas (TDD). El documento proporciona una guía práctica con ejemplos de código, mejores prácticas y análisis comparativo entre ambas metodologías.

2 Introducción

2.1 Objetivo

Demostrar la implementación efectiva de Jest como framework de testing en aplicaciones NestJS, aplicando primero Unit Testing tradicional y posteriormente la metodología TDD para comparar beneficios, desafíos y resultados obtenidos.

2.2 Alcance

El informe cubre la configuración inicial, implementación de pruebas unitarias, aplicación de TDD, análisis de cobertura de código y recomendaciones para la adopción en proyectos empresariales.

2.3 Tecnologías Utilizadas

- **NestJS** v10.x - Framework de Node.js
- **Jest** v29.x - Framework de testing
- **TypeScript** v5.x - Lenguaje de programación
- **Supertest** - Testing de endpoints HTTP

3 Fundamentos Teóricos

3.1 Unit Testing

Las pruebas unitarias son el nivel más básico de testing en software, donde se prueban componentes individuales de manera aislada. En el contexto de NestJS, esto incluye servicios, controladores, middlewares y pipes.

Características principales:

- Aislamiento de dependencias mediante mocking
- Ejecución rápida y automatizada
- Cobertura de casos edge y escenarios de error
- Validación de comportamiento esperado

3.2 Test-Driven Development (TDD)

TDD es una metodología de desarrollo donde las pruebas se escriben antes que el código de producción, siguiendo el ciclo Red-Green-Refactor.

Ciclo TDD:

1. **Red:** Escribir una prueba que falle
2. **Green:** Escribir el mínimo código para que pase
3. **Refactor:** Mejorar el código manteniendo las pruebas

4 Configuración del Entorno

4.1 Instalación y Configuración Inicial

```
# Crear proyecto NestJS
npm i -g @nestjs/cli
nest new jest-nestjs-project

# Instalar dependencias de testing (incluidas por defecto)
npm install --save-dev @nestjs/testing jest @types/jest ts-jest
   supertest @types/supertest
```

Listing 1: Instalación de dependencias

4.2 Configuración de Jest

```
{
  "moduleFileExtensions": ["js", "json", "ts"],
  "rootDir": "src",
  "testRegex": ".*\\.spec\\.ts$",
  "transform": {
    "^.+\\.(t|j)s$": "ts-jest"
  },
  "collectCoverageFrom": [
    "**/*.(t|j)s"
  ],
  "coverageDirectory": "../coverage",
  "testEnvironment": "node",
  "coverageReporters": ["text", "lcov", "html"],
  "coverageThreshold": {
    "global": {
      "branches": 80,
      "functions": 80,
      "lines": 80,
      "statements": 80
    }
  }
}
```

Listing 2: jest.config.js

5 Implementación de Unit Testing Tradicional

5.1 Estructura del Proyecto

Implementaremos un sistema de gestión de usuarios como caso de estudio.

```
export interface User {
  id: number;
  email: string;
  name: string;
  isActive: boolean;
  createdAt: Date;
}

export interface CreateUserDto {
  email: string;
  name: string;
}

export interface UpdateUserDto {
  name?: string;
  isActive?: boolean;
}
```

Listing 3: user.entity.ts

5.2 Implementación del Servicio

```
import { Injectable, NotFoundException, BadRequestException }
   from '@nestjs/common';
import { User, CreateUserDto, UpdateUserDto } from './interfaces/
   user.interface';

@Injectable()
export class UserService {
  private users: User[] = [];
  private nextId = 1;

  async findAll(): Promise<User[]> {
    return this.users.filter(user => user.isActive);
  }

  async findById(id: number): Promise<User> {
    const user = this.users.find(u => u.id === id);
    if (!user) {
      throw new NotFoundException(`User with ID ${id} not found`)
   ;
    }
    return user;
  }

  async create(createUserDto: CreateUserDto): Promise<User> {
```

```
22      const existingUser = this.users.find(u => u.email ===
    createUserDto.email);
23      if (existingUser) {
24        throw new BadRequestException('Email already exists');
25      }
26
27      const user: User = {
28        id: this.nextId++,
29        email: createUserDto.email,
30        name: createUserDto.name,
31        isActive: true,
32        createdAt: new Date(),
33      };
34
35      this.users.push(user);
36      return user;
37    }
38
39    async update(id: number, updateUserDto: UpdateUserDto): Promise
    <User> {
40      const user = await this.findById(id);
41
42      if (updateUserDto.name) user.name = updateUserDto.name;
43      if (typeof updateUserDto.isActive !== 'undefined') {
44        user.isActive = updateUserDto.isActive;
45      }
46
47      return user;
48    }
49
50    async delete(id: number): Promise<void> {
51      const userIndex = this.users.findIndex(u => u.id === id);
52      if (userIndex === -1) {
53        throw new NotFoundException(`User with ID ${id} not found`)
    ;
54      }
55      this.users.splice(userIndex, 1);
56    }
57  }
```

Listing 4: user.service.ts

5.3 Pruebas Unitarias del Servicio

```
import { Test, TestingModule } from '@nestjs/testing';
import { NotFoundException, BadRequestException } from '@nestjs/
   common';
import { UserService } from './user.service';
import { CreateUserDto, UpdateUserDto } from './interfaces/user.
   interface';

describe('UserService', () => {
  let service: UserService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule
   ({
      providers: [UserService],
    }).compile();

    service = module.get<UserService>(UserService);
  });

  describe('create', () => {
    it('should create a new user successfully', async () => {
      const createUserDto: CreateUserDto = {
        email: 'test@example.com',
        name: 'Test User',
      };

      const result = await service.create(createUserDto);

      expect(result).toHaveProperty('id');
      expect(result.email).toBe(createUserDto.email);
      expect(result.name).toBe(createUserDto.name);
      expect(result.isActive).toBe(true);
      expect(result.createdAt).toBeInstanceOf(Date);
    });

    it('should throw BadRequestException when email already
   exists', async () => {
      const createUserDto: CreateUserDto = {
        email: 'duplicate@example.com',
        name: 'Test User',
      };

      await service.create(createUserDto);

      await expect(service.create(createUserDto))
        .rejects
        .toThrow(BadRequestException);
    });
  });
```

```
46
47  describe('findAll', () => {
48    it('should return only active users', async () => {
49      // Arrange
50      const user1 = await service.create({
51        email: 'user1@example.com',
52        name: 'User 1',
53      });
54
55      const user2 = await service.create({
56        email: 'user2@example.com',
57        name: 'User 2',
58      });
59
60      await service.update(user2.id, { isActive: false });
61
62      // Act
63      const result = await service.findAll();
64
65      // Assert
66      expect(result).toHaveLength(1);
67      expect(result[0].id).toBe(user1.id);
68    });
69  });
70
71  describe('findById', () => {
72    it('should return user when found', async () => {
73      const createdUser = await service.create({
74        email: 'findme@example.com',
75        name: 'Find Me',
76      });
77
78      const result = await service.findById(createdUser.id);
79
80      expect(result).toEqual(createdUser);
81    });
82
83    it('should throw NotFoundException when user not found',
     async () => {
84      await expect(service.findById(999))
85        .rejects
86        .toThrow(NotFoundException);
87    });
88  });
89
90  describe('update', () => {
91    it('should update user properties', async () => {
92      const user = await service.create({
93        email: 'update@example.com',
94        name: 'Original Name',
95      });
```

```
96
97        const updateDto: UpdateUserDto = {
98          name: 'Updated Name',
99          isActive: false,
100       };
101
102       const result = await service.update(user.id, updateDto);
103
104       expect(result.name).toBe(updateDto.name);
105       expect(result.isActive).toBe(updateDto.isActive);
106     });
107   });
108
109   describe('delete', () => {
110     it('should delete user successfully', async () => {
111       const user = await service.create({
112         email: 'delete@example.com',
113         name: 'Delete Me',
114       });
115
116       await service.delete(user.id);
117
118       await expect(service.findById(user.id))
119         .rejects
120         .toThrow(NotFoundException);
121     });
122   });
123 });
```

Listing 5: user.service.spec.ts

6 Implementación con TDD

Para demostrar TDD, implementaremos una funcionalidad de validación de contraseñas siguiendo estrictamente el ciclo Red-Green-Refactor.

6.1 Paso 1: Red - Escribir la Prueba que Falla

```
import { Test, TestingModule } from '@nestjs/testing';
import { PasswordValidatorService } from './password-validator.service';

describe('PasswordValidatorService - TDD Implementation', () => {
  let service: PasswordValidatorService;
  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [PasswordValidatorService],
    }).compile();
    service = module.get<PasswordValidatorService>(PasswordValidatorService);
  });

  // RED PHASE: Escribimos la primera prueba que fallara
  describe('validate', () => {
    it('should return true for valid password', () => {
      const validPassword = 'ValidPass123!';
      const result = service.validate(validPassword);
      expect(result.isValid).toBe(true);
      expect(result.errors).toHaveLength(0);
    });
  });
});
```

Listing 6: password-validator.service.spec.ts

6.2 Paso 2: Green - Mínimo Código para Pasar

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

export interface ValidationResult {
  isValid: boolean;
  errors: string[];
}
@Injectable()
export class PasswordValidatorService {
  validate(password: string): ValidationResult {
    return {
      isValid: true,
      errors: []
    };}}
```

Listing 7: password-validator.service.ts (Primera iteración)

6.3 Paso 3: Refactor y Expandir con más Pruebas

```
describe('PasswordValidatorService - TDD Implementation', () => {
  let service: PasswordValidatorService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [PasswordValidatorService],
    }).compile();

    service = module.get<PasswordValidatorService>(PasswordValidatorService);
  });

  describe('validate', () => {
    it('should return true for valid password', () => {
      const validPassword = 'ValidPass123!';
      const result = service.validate(validPassword);
      expect(result.isValid).toBe(true);
      expect(result.errors).toHaveLength(0);
    });

    RED: Nueva prueba que fallar
    it('should return false for password shorter than 8 characters', () => {
      const shortPassword = 'Short1!';
      const result = service.validate(shortPassword);
      expect(result.isValid).toBe(false);
      expect(result.errors).toContain('Password must be at least 8 characters long');
    });

    it('should return false for password without uppercase letter', () => {
      const noUpperPassword = 'lowercase123!';
      const result = service.validate(noUpperPassword);
      expect(result.isValid).toBe(false);
      expect(result.errors).toContain('Password must contain at least one uppercase letter');
    });

    it('should return false for password without lowercase letter', () => {
      const noLowerPassword = 'UPPERCASE123!';
      const result = service.validate(noLowerPassword);
      expect(result.isValid).toBe(false);
      expect(result.errors).toContain('Password must contain at least one lowercase letter');
    });

```

```
42      it('should return false for password without numbers', () =>
    {
43          const noNumberPassword = 'Password!';
44          const result = service.validate(noNumberPassword);
45          expect(result.isValid).toBe(false);
46          expect(result.errors).toContain('Password must contain at
    least one number');
47      });
48
49      it('should return false for password without special
    characters', () => {
50          const noSpecialPassword = 'Password123';
51          const result = service.validate(noSpecialPassword);
52          expect(result.isValid).toBe(false);
53          expect(result.errors).toContain('Password must contain at
    least one special character');
54      });
55
56      it('should return multiple errors for invalid password', ()
    => {
57          const invalidPassword = 'abc';
58          const result = service.validate(invalidPassword);
59          expect(result.isValid).toBe(false);
60          expect(result.errors.length).toBeGreaterThan(1);
61      });
62    });
63  });
```

Listing 8: Expandiendo las pruebas

6.4 Implementación Final del Servicio

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

export interface ValidationResult {
  isValid: boolean;
  errors: string[];
}

@Injectable()
export class PasswordValidatorService {
  private readonly MIN_LENGTH = 8;
  private readonly UPPERCASE_REGEX = /[A-Z]/;
  private readonly LOWERCASE_REGEX = /[a-z]/;
  private readonly NUMBER_REGEX = /[0-9]/;
  private readonly SPECIAL_CHAR_REGEX = /[!@#$%^&*(),.?":{}|<>]/;

  validate(password: string): ValidationResult {
    const errors: string[] = [];

    if (!password || password.length < this.MIN_LENGTH) {
      errors.push('Password must be at least 8 characters long');
    }

    if (!this.UPPERCASE_REGEX.test(password)) {
      errors.push('Password must contain at least one uppercase
   letter');
    }

    if (!this.LOWERCASE_REGEX.test(password)) {
      errors.push('Password must contain at least one lowercase
   letter');
    }

    if (!this.NUMBER_REGEX.test(password)) {
      errors.push('Password must contain at least one number');
    }

    if (!this.SPECIAL_CHAR_REGEX.test(password)) {
      errors.push('Password must contain at least one special
   character');
    }

    return {
      isValid: errors.length === 0,
      errors
    };
  }

  generateStrongPassword(length: number = 12): string {
    const uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
```

```
47      const lowercase = 'abcdefghijklmnopqrstuvwxyz';
48      const numbers = '0123456789';
49      const symbols = '!@#$%^&*(),.?":{}|<>';
50
51      let password = '';
52
53      // Garantizar al menos un car cter de cada tipo
54      password += uppercase[Math.floor(Math.random() * uppercase.
    length)];
55      password += lowercase[Math.floor(Math.random() * lowercase.
    length)];
56      password += numbers[Math.floor(Math.random() * numbers.length
    )];
57      password += symbols[Math.floor(Math.random() * symbols.length
    )];
58
59      // Completar con caracteres aleatorios
60      const allChars = uppercase + lowercase + numbers + symbols;
61      for (let i = password.length; i < length; i++) {
62        password += allChars[Math.floor(Math.random() * allChars.
    length)];
63      }
64
65      // Mezclar el password
66      return password.split('').sort(() => 0.5 - Math.random()).
    join('');
67    }
68  }
```

Listing 9: password-validator.service.ts (Implementación completa)

7 Análisis Comparativo: Unit Testing vs TDD

7.1 Ventajas del Unit Testing Tradicional

Pros:

- Flexibilidad en el diseño inicial
- Menor tiempo inicial de desarrollo
- Facilita la refactorización de código existente
- Permite enfoques pragmáticos

Contras:

- Posible sesgo hacia casos de éxito
- Menor cobertura de casos edge
- Diseño menos orientado a la testabilidad

7.2 Ventajas de TDD

Pros:

- Mejor diseño de interfaces y APIs
- Mayor cobertura de casos edge
- Código más modular y testeable
- Documentación viva del comportamiento esperado
- Menor cantidad de bugs en producción

Contras:

- Curva de aprendizaje inicial
- Mayor tiempo de desarrollo inicial
- Puede generar sobre-ingeniería en casos simples
- Requiere disciplina del equipo

8 Mejores Prácticas

8.1 Naming Conventions

```
// Estructura de describe/it descriptiva
describe('UserService', () => {
  describe('create', () => {
    it('should create user when valid data provided', () => {
      // Given - When - Then pattern
    });

    it('should throw BadRequestException when email already exists', () => {
      // Arrange - Act - Assert pattern
    });
  });
});

// Nombres descriptivos para tests
describe('PasswordValidatorService', () => {
  describe('validate', () => {
    it('should accept password with all required criteria', () => {});
    it('should reject password shorter than 8 characters', () => {});
    it('should reject password without uppercase letters', () => {});
  });
});
```

Listing 10: Convenciones de nomenclatura

8.2 Organización de Tests

```
it('should update user when valid data provided', async () => {
  // Arrange
  const user = await service.create({
    email: 'test@example.com',
    name: 'Test User'
  });

  const updateDto: UpdateUserDto = {
    name: 'Updated Name',
    isActive: false
  };

  // Act
  const result = await service.update(user.id, updateDto);

  // Assert
  expect(result.name).toBe(updateDto.name);
```

```
  expect(result.isActive).toBe(updateDto.isActive);
  expect(result.id).toBe(user.id);
});
```

Listing 11: Estructura AAA (Arrange-Act-Assert)

8.3 Mock Strategies

```
describe('UserController', () => {
  let controller: UserController;
  let userService: jest.Mocked<UserService>;

  beforeEach(async () => {
    const mockUserService = {
      findAll: jest.fn(),
      findById: jest.fn(),
      create: jest.fn(),
      update: jest.fn(),
      delete: jest.fn(),
    };

    const module: TestingModule = await Test.createTestingModule
    ({
      controllers: [UserController],
      providers: [
        {
          provide: UserService,
          useValue: mockUserService,
        },
      ],
    }).compile();

    controller = module.get<UserController>(UserController);
    userService = module.get<UserService>(UserService) as jest.
    Mocked<UserService>;
  });

  it('should call userService.create with correct parameters',
   async () => {
    const createDto = { email: 'test@example.com', name: 'Test'
   };
    const expectedUser = { id: 1, ...createDto, isActive: true,
   createdAt: new Date() };

    userService.create.mockResolvedValue(expectedUser);

    const result = await controller.create(createDto);

    expect(userService.create).toHaveBeenCalledWith(createDto);
    expect(userService.create).toHaveBeenCalledTimes(1);
    expect(result).toEqual(expectedUser);
```

```
  });
});
```

Listing 12: Estrategias de mocking efectivas

8.4 Test Data Builders

```
export class UserTestDataBuilder {
  private user: Partial<User> = {
    id: 1,
    email: 'default@example.com',
    name: 'Default User',
    isActive: true,
    createdAt: new Date('2023-01-01'),
  };

  withId(id: number): UserTestDataBuilder {
    this.user.id = id;
    return this;
  }

  withEmail(email: string): UserTestDataBuilder {
    this.user.email = email;
    return this;
  }

  withName(name: string): UserTestDataBuilder {
    this.user.name = name;
    return this;
  }

  inactive(): UserTestDataBuilder {
    this.user.isActive = false;
    return this;
  }

  build(): User {
    return this.user as User;
  }
}

// Uso en tests
it('should return only active users', async () => {
  const activeUser = new UserTestDataBuilder()
    .withId(1)
    .withEmail('active@example.com')
    .build();

  const inactiveUser = new UserTestDataBuilder()
    .withId(2)
    .withEmail('inactive@example.com')
```

```
45      .inactive()
46      .build();
47
48    // Resto del test...
49  });
50  \end{lstlisting>
51
52  \section{Implementaci n de CI/CD}
53
54  \subsection{GitHub Actions Workflow}
55
56  \begin{lstlisting}[language=yaml, caption=.github/workflows/test.
       yml]
57  name: Test Suite
58
59  on:
60    push:
61      branches: [ main, develop ]
62    pull_request:
63      branches: [ main ]
64
65  jobs:
66    test:
67      runs-on: ubuntu-latest
68
69      strategy:
70        matrix:
71          node-version: [18.x, 20.x]
72
73      steps:
74      - uses: actions/checkout@v4
75
76      - name: Use Node.js ${{ matrix.node-version }}
77        uses: actions/setup-node@v4
78        with:
79          node-version: ${{ matrix.node-version }}
80          cache: 'npm'
81
82      - name: Install dependencies
83        run: npm ci
84
85      - name: Run linter
86        run: npm run lint
87
88      - name: Run unit tests
89        run: npm run test:cov
90
91      - name: Run e2e tests
92        run: npm run test:e2e
93
94      - name: Upload coverage to Codecov
```

```
95        uses: codecov/codecov-action@v3
96        with:
97          file: ./coverage/lcov.info
98          flags: unittests
99          name: codecov-umbrella
100
101     - name: SonarCloud Scan
102       uses: SonarSource/sonarcloud-github-action@master
103       env:
104         GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
105         SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

Listing 13: Builders para datos de prueba

9 Recomendaciones

9.1 Para Equipos Principiantes

- Comenzar con Unit Testing tradicional
- Establecer umbrales de cobertura progresivos (60 % → 80 % → 90 %)
- Implementar tests en código legacy gradualmente
- Capacitar al equipo en conceptos fundamentales
- Usar pair programming para tests complejos

9.2 Para Equipos Avanzados

- Adoptar TDD para funcionalidades críticas
- Implementar mutation testing para validar calidad de tests
- Establecer architectural decision records (ADR) para estrategias de testing
- Integrar property-based testing para casos complejos
- Implementar testing de contratos para microservicios

Referencias

[1] NestJS Documentation. *Testing - NestJS Official Documentation*. `https://docs.nestjs.com/fundamentals/testing`

[2] Jest Documentation. *Jest JavaScript Testing Framework*. `https://jestjs.io/docs/getting-started`

[3] Beck, Kent. *Test-Driven Development by Example*. Addison-Wesley Professional, 2003.