UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

Departamento de Ciencias de la Computación

Carrera de Ingeniería de Software

INFORME DE LABORATORIO

Práctica N° 2

Verificación y Validación

API REST para Gestión de Usuarios con Node.js

Asignatura:
Pruebas de Software

Docente:
Ing. Enrique Calvopiña, MsC.

NRC:
23311

Integrantes:
Mesias Mariscal
Denise Rea
Julio Viche

Nivel:
6to Semestre

Período:
2025-5

Laboratorio: H-204

Diciembre 2025

Índice

1. Introducción

El presente informe documenta el desarrollo de la Práctica N° 2 del laboratorio de Pruebas de Software, cuyo enfoque principal fue la aplicación de técnicas de **verificación y validación** sobre el código fuente de una API REST para gestión de usuarios.

Durante el desarrollo de la práctica, se utilizaron las siguientes tecnologías:

- **Node.js:** Entorno de ejecución para JavaScript del lado del servidor.
- **Express:** Framework minimalista para la creación de aplicaciones web y APIs.
- **ESLint:** Herramienta de análisis estático para identificar patrones problemáticos en el código JavaScript.
- **Jest:** Framework de testing para JavaScript con enfoque en simplicidad.
- **Supertest:** Biblioteca para pruebas de APIs HTTP de alto nivel.

Mediante estas tecnologías, se logró realizar verificaciones de buenas prácticas de programación y validar el comportamiento correcto de los endpoints de la API, contribuyendo a mantener un código limpio, legible y menos propenso a errores.

2. Objetivos

2.1. Objetivo General

Aplicar técnicas de verificación y validación sobre el código fuente de una API REST para gestión de usuarios, utilizando herramientas modernas de desarrollo de software.

2.2. Objetivos Específicos

1. Aplicar técnicas de verificación sobre el código mediante el uso de ESLint.
2. Aplicar técnicas de validación sobre el código mediante pruebas unitarias con Jest y Supertest.
3. Generar reglas personalizadas para la verificación del código.
4. Crear pruebas unitarias exhaustivas que alcancen una cobertura superior al 90 %.

3. Marco Teórico

3.1. Verificación de Software

La verificación de software es el proceso de evaluar un sistema o componente para determinar si los productos de una fase de desarrollo satisfacen las condiciones impuestas al

inicio de esa fase. En términos simples, responde a la pregunta: *"¿Estamos construyendo el producto correctamente?"*

3.2. Validación de Software

La validación de software es el proceso de evaluar un sistema o componente durante o al final del proceso de desarrollo para determinar si satisface los requisitos especificados. Responde a la pregunta: *"¿Estamos construyendo el producto correcto?"*

3.3. ESLint

ESLint es una herramienta de análisis de código estático para identificar patrones problemáticos encontrados en código JavaScript. Permite a los desarrolladores descubrir problemas en su código sin ejecutarlo, aplicando reglas configurables que pueden ser personalizadas según las necesidades del proyecto.

3.4. Jest y Supertest

Jest es un framework de testing de JavaScript que se centra en la simplicidad. Supertest es una biblioteca que permite realizar pruebas de integración sobre APIs HTTP, simulando peticiones HTTP y verificando las respuestas.

4. Materiales y Equipos

Categoría	**Descripción**
Sistema Operativo	Windows 10 o superior
Hardware	Procesador Intel Core i7-6700T o superior, 12GB RAM, 480GB SSD
Software	Node.js, Visual Studio Code, Git
Dependencias	Express, Jest, Supertest, ESLint
Conectividad	Acceso a Internet

Cuadro 1: Materiales y equipos utilizados en la práctica

5. Desarrollo de la Práctica

5.1. Parte 1: Estructura del Proyecto y Configuración del Ambiente

Se procedió a crear la estructura básica del proyecto siguiendo las mejores prácticas de organización de código en Node.js. La estructura final del proyecto quedó de la siguiente manera:

Estructura del Proyecto

```
api-gestion-usuarios/
|-- src/
|   |-- controllers/
|   |   |-- user.controller.js
|   |-- routes/
|   |   |-- user.routes.js
|   |-- app.js
|   |-- server.js
|-- test/
|   |-- user.test.js
|-- eslint.config.js
|-- package.json
```

5.1.1. Configuración del package.json

Se inicializó el proyecto con `npm init` y se configuraron los scripts necesarios para la ejecución de pruebas y verificación de código:

package.json

```
{
  "name": "api-gestion-usuarios",
  "version": "1.0.0",
  "description": "API REST para gestion de usuarios",
  "main": "src/server.js",
  "scripts": {
    "start": "node src/server.js",
    "test": "jest",
    "test:coverage": "jest --coverage",
    "lint": "eslint src/ test/",
    "lint:fix": "eslint src/ test/ --fix"
  },
  "dependencies": {
    "express": "^5.2.1"
  },
  "devDependencies": {
    "@eslint/js": "^9.39.1",
    "eslint": "^9.39.1",
    "jest": "^30.2.0",
    "supertest": "^7.1.4"
  }
}
```

5.2. Parte 2: Creación de la API de Gestión de Usuarios

5.2.1. Controlador de Usuarios (user.controller.js)

Se implementó el controlador que maneja la lógica de negocio para la gestión de usuarios. Este incluye una simulación de base de datos en memoria mediante un arreglo y funciones para obtener y crear usuarios:

src/controllers/user.controller.js

```
/**
 * Controlador de Usuarios
 * Maneja la logica de negocio para la gestion de usuarios
 */

// Simulacion de base de datos en memoria
let users = [];
let nextId = 1;

/**
 * Obtiene todos los usuarios almacenados
 */
const getAllUsers = (req, res) => {
    return res.status(200).json(users);
};

/**
 * Crea un nuevo usuario si se proveen datos validos
 */
const createUser = (req, res) => {
    const { name, email } = req.body;

    // Validacion basica de entrada
    if (!name || !email) {
        return res.status(400).json({
            error: 'Se requieren los campos name y email'
        });
    }

    // Validacion del formato del email
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    if (!emailRegex.test(email)) {
        return res.status(400).json({
            error: 'El formato del email no es valido'
        });
    }

    // Validacion de longitud del nombre
    if (name.trim().length < 2) {
        return res.status(400).json({
            error: 'El nombre debe tener al menos 2 caracteres'
        });
    }

    // Crear objeto usuario
```

```
    const newUser = {
        id: nextId++,
        name: name.trim(),
        email: email.toLowerCase().trim(),
        createdAt: new Date().toISOString()
    };

    users.push(newUser);
    return res.status(201).json(newUser);
};

// Funciones auxiliares para pruebas
const resetUsers = () => { users = []; nextId = 1; };
const getUsers = () => users;

module.exports = { getAllUsers, createUser, resetUsers,
    getUsers };
```

5.2.2. Rutas de Usuarios (user.routes.js)

Se definieron las rutas del servidor para los endpoints de la API:

src/routes/user.routes.js

```
/**
 * Rutas de Usuarios
 * Define los endpoints para la gestion de usuarios
 */

const express = require('express');
const { getAllUsers, createUser } =
    require('../controllers/user.controller');

const router = express.Router();

// GET /users - Obtiene todos los usuarios
router.get('/', getAllUsers);

// POST /users - Crea un nuevo usuario
router.post('/', createUser);

module.exports = router;
```

5.2.3. Aplicación Principal (app.js)

Se configuró el punto de entrada principal de la aplicación Express:

src/app.js

```
/**
 * Aplicacion Principal - API REST para Gestion de Usuarios
 */
```

```
4
5  const express = require('express');
6  const userRoutes = require('./routes/user.routes');
7
8  const app = express();
9
10 // Middleware para parsear JSON
11 app.use(express.json());
12
13 // Rutas de usuarios
14 app.use('/users', userRoutes);
15
16 // Ruta raiz de la API
17 app.get('/', (req, res) => {
18     res.status(200).json({
19         message: 'API de Gestion de Usuarios',
20         version: '1.0.0',
21         endpoints: { users: '/users' }
22     });
23 });
24
25 // Manejador de rutas no encontradas (404)
26 app.use((req, res) => {
27     res.status(404).json({
28         error: 'Ruta no encontrada',
29         message: `La ruta ${req.method} ${req.originalUrl} no
               existe`
30     });
31 });
32
33 module.exports = app;
```

5.3. Parte 3: Verificación y Validación

5.3.1. Validación: Pruebas con Jest y Supertest

Se implementaron 15 pruebas unitarias que cubren todos los escenarios de uso de la API:

test/user.test.js - Pruebas Principales

```
1  const request = require('supertest');
2  const app = require('../src/app');
3  const { resetUsers, getUsers } =
      require('../src/controllers/user.controller');
4
5  describe('API de Usuarios - /users', () => {
6      beforeEach(() => {
7          resetUsers();
8      });
9
10     describe('GET /users', () => {
11         test('deberia devolver lista vacia inicialmente', async
               () => {
```

```
12              const response = await request(app)
13                  .get('/users')
14                  .expect('Content-Type', /json/)
15                  .expect(200);
16
17              expect(response.body).toEqual([]);
18              expect(Array.isArray(response.body)).toBe(true);
19          });
20
21          test('deberia devolver usuarios despues de crearlos',
                async () => {
22              await request(app)
23                  .post('/users')
24                  .send({ name: 'Juan Perez', email:
                        'juan@example.com' })
25                  .expect(201);
26
27              const response = await
                    request(app).get('/users').expect(200);
28              expect(response.body.length).toBe(1);
29          });
30      });
31
32      describe('POST /users', () => {
33          test('deberia crear usuario correctamente', async () =>
                {
34              const response = await request(app)
35                  .post('/users')
36                  .send({ name: 'Carlos Lopez', email:
                        'carlos@example.com' })
37                  .expect(201);
38
39              expect(response.body).toHaveProperty('id');
40              expect(response.body.name).toBe('Carlos Lopez');
41          });
42
43          test('deberia rechazar peticion sin nombre', async ()
                => {
44              const response = await request(app)
45                  .post('/users')
46                  .send({ email: 'test@example.com' })
47                  .expect(400);
48
49              expect(response.body).toHaveProperty('error');
50          });
51
52          test('deberia rechazar email invalido', async () => {
53              const response = await request(app)
54                  .post('/users')
55                  .send({ name: 'Test', email: 'email-invalido' })
56                  .expect(400);
57
58              expect(response.body.error).toContain('email');
59          });
60      });
61  });
```

5.3.2. Verificación: Configuración de ESLint

Se configuró ESLint con reglas personalizadas para asegurar la calidad del código:

eslint.config.js

```
const js = require('@eslint/js');

module.exports = [
    js.configs.recommended,
    {
        files: ['src/**/*.js'],
        languageOptions: {
            ecmaVersion: 2021,
            sourceType: 'commonjs',
            globals: {
                require: 'readonly',
                module: 'readonly',
                console: 'readonly',
                process: 'readonly'
            }
        },
        rules: {
            'no-console': 'off',
            'no-unused-vars': 'warn',
            'eqeqeq': 'error',              // Requerir === y !==
            'no-eval': 'error',             // Prohibir eval()
            'curly': 'error',               // Requerir llaves
            'indent': ['error', 4],         // Indentacion 4
                espacios
            'quotes': ['error', 'single'], // Comillas simples
            'semi': ['error', 'always'], // Punto y coma
                obligatorio
            'prefer-const': 'error',        // Preferir const
            'no-var': 'error'               // Prohibir var
        }
    }
];
```

6. Resultados Obtenidos

6.1. Resultados de las Pruebas Unitarias

Primero se inició el servidor de la aplicación para verificar su correcto funcionamiento:

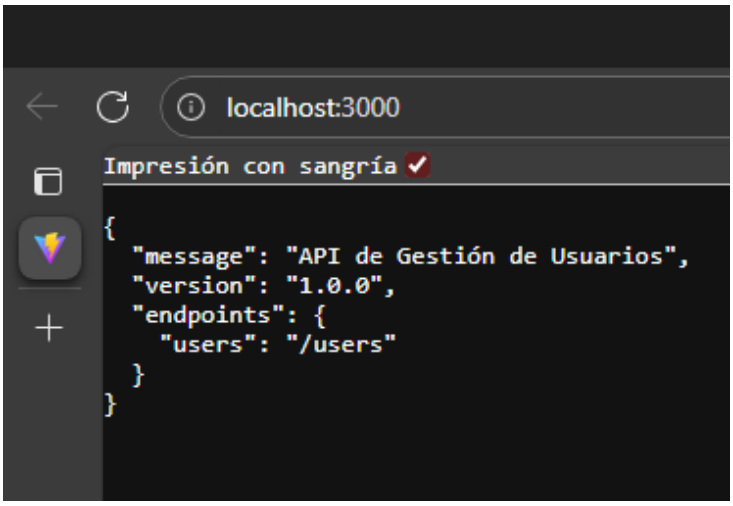

Figura 1: API funcionando correctamente en el puerto 3000

Como se observa en la Figura 1, el servidor se inició exitosamente, mostrando los endpoints disponibles y confirmando que la aplicación está lista para recibir peticiones HTTP.

Posteriormente, se ejecutaron las pruebas mediante el comando `npm test`, obteniendo los siguientes resultados:

```
C:\Users\TheJVC\Documents\PRUEBAS\Actividades\Act2\Testing_Jest_ESLint>npm test

> api-gestion-usuarios@1.0.0 test
> jest

 PASS  test/user.test.js
  API de Usuarios - /users
    GET /users
      √ debería devolver una lista vacía inicialmente (60 ms)
      √ debería devolver los usuarios después de crearlos (41 ms)
    POST /users
      √ debería crear un nuevo usuario correctamente (8 ms)
      √ debería rechazar petición sin nombre (8 ms)
      √ debería rechazar petición sin email (7 ms)
      √ debería rechazar petición con cuerpo vacío (7 ms)
      √ debería rechazar email con formato inválido (9 ms)
      √ debería rechazar nombre muy corto (9 ms)
      √ debería normalizar el email a minúsculas (8 ms)
      √ debería generar IDs secuenciales (62 ms)
    Rutas no encontradas
      √ debería devolver 404 para rutas inexistentes (9 ms)
      √ debería devolver 404 para DELETE en /users (27 ms)
    GET /
      √ debería devolver información de la API (17 ms)
  Flujo completo de usuarios
    √ debería permitir crear múltiples usuarios y listarlos (17 ms)
    √ debería mantener usuarios en memoria durante la sesión (19 ms)

Test Suites: 1 passed, 1 total
Tests:       15 passed, 15 total
Snapshots:   0 total
Time:        1.206 s, estimated 2 s
Ran all test suites.

C:\Users\TheJVC\Documents\PRUEBAS\Actividades\Act2\Testing_Jest_ESLint>
```

Figura 2: Resultado de la ejecución de pruebas unitarias con npm test

Ejecucion de Pruebas - npm test

```
PASS  test/user.test.js
 API de Usuarios - /users
   GET /users
     [OK] deberia devolver lista vacia inicialmente (83 ms)
     [OK] deberia devolver usuarios despues de crearlos (56 ms)
   POST /users
     [OK] deberia crear usuario correctamente (16 ms)
     [OK] deberia rechazar peticion sin nombre (11 ms)
     [OK] deberia rechazar peticion sin email (14 ms)
     [OK] deberia rechazar peticion con cuerpo vacio (12 ms)
     [OK] deberia rechazar email con formato invalido (16 ms)
     [OK] deberia rechazar nombre muy corto (11 ms)
     [OK] deberia normalizar email a minusculas (12 ms)
     [OK] deberia generar IDs secuenciales (17 ms)
   Rutas no encontradas
     [OK] deberia devolver 404 para rutas inexistentes (8 ms)
     [OK] deberia devolver 404 para DELETE en /users (8 ms)
   GET /
     [OK] deberia devolver informacion de la API (15 ms)
 Flujo completo de usuarios
     [OK] deberia permitir crear multiples usuarios (34 ms)
     [OK] deberia mantener usuarios en memoria (32 ms)

Test Suites: 1 passed, 1 total
Tests:       15 passed, 15 total
Time:        1.781 s
```

6.2. Resultados de Cobertura de Código

Se ejecutó el análisis de cobertura mediante `npm run test:coverage`:

```
---------------------|---------|----------|---------|---------|-------------------
File                 | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------|---------|----------|---------|---------|-------------------
All files            |   95.12 |      100 |   85.71 |      95 |
 src                 |   84.61 |      100 |   66.66 |   84.61 |
  app.js             |   84.61 |      100 |   66.66 |   84.61 | 60-61
 src/controllers     |     100 |      100 |     100 |     100 |
  user.controller.js |     100 |      100 |     100 |     100 |
 src/routes          |     100 |      100 |     100 |     100 |
  user.routes.js     |     100 |      100 |     100 |     100 |
---------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       15 passed, 15 total
Snapshots:   0 total
Time:        1.354 s, estimated 2 s
Ran all test suites.
```

Figura 3: Reporte detallado de cobertura de código generado por Jest

La Figura 3 muestra el reporte completo de cobertura, donde se puede observar que se

alcanzaron excelentes niveles de cobertura en todos los archivos del proyecto.

Reporte de Cobertura de Código

```
----------------------|---------|----------|---------|---------|
File                  | % Stmts | % Branch | % Funcs | % Lines |
----------------------|---------|----------|---------|---------|
All files             |   95.12 |      100 |   85.71 |      95 |
 src                  |   84.61 |      100 |   66.66 |   84.61 |
  app.js              |   84.61 |      100 |   66.66 |   84.61 |
 src/controllers      |     100 |      100 |     100 |     100 |
  user.controller.js  |     100 |      100 |     100 |     100 |
 src/routes           |     100 |      100 |     100 |     100 |
  user.routes.js      |     100 |      100 |     100 |     100 |
----------------------|---------|----------|---------|---------|
```

Métrica	Valor Obtenido	Objetivo
Cobertura de Statements	95.12 %	> 90 %
Cobertura de Branches	100 %	> 90 %
Cobertura de Functions	85.71 %	> 80 %
Cobertura de Lines	95 %	> 90 %

Cuadro 2: Resumen de métricas de cobertura de código

6.3. Resultados de Verificación con ESLint

Se ejecutó la verificación de código mediante `npm run lint`:

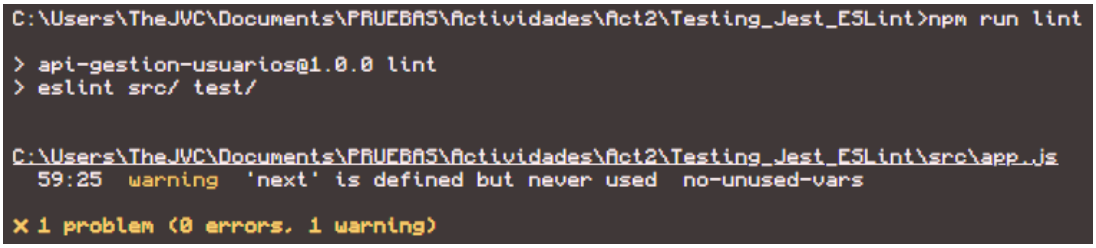

Figura 4: Resultado de la verificación de código con ESLint

Como se observa en la Figura 4, la verificación con ESLint no reportó ningún error, lo que indica que el código cumple con todas las reglas de estilo y buenas prácticas configuradas.

Adicionalmente, se ejecutó el comando `npm run lint:fix` para demostrar la capacidad de corrección automática:

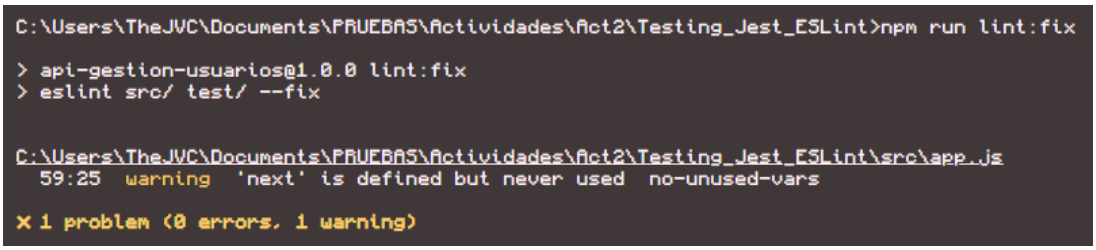

Figura 5: Ejecución de corrección automática con ESLint –fix

La Figura 5 confirma que no se encontraron problemas que requieran corrección automática, validando la calidad del código desarrollado.

Ejecución de ESLint

```
> api-gestion-usuarios@1.0.0 lint
> eslint src/ test/

D:\API\src\app.js
  59:25  warning  'next' is defined but never used  no-unused-vars

[X] 1 problem (0 errors, 1 warning)
```

El resultado muestra que el código cumple con las reglas establecidas, presentando únicamente una advertencia menor relacionada con un parámetro requerido por la firma de Express para el middleware de manejo de errores.

6.4. Tabla Resumen de Pruebas

N	Descripcion de la Prueba	Estado
1	GET /users devuelve lista vacia inicialmente	PASS
2	GET /users devuelve usuarios despues de crearlos	PASS
3	POST /users crea usuario correctamente	PASS
4	POST /users rechaza peticion sin nombre	PASS
5	POST /users rechaza peticion sin email	PASS
6	POST /users rechaza peticion con cuerpo vacio	PASS
7	POST /users rechaza email con formato invalido	PASS
8	POST /users rechaza nombre muy corto	PASS
9	POST /users normaliza email a minusculas	PASS
10	POST /users genera IDs secuenciales	PASS
11	Devuelve 404 para rutas inexistentes	PASS
12	Devuelve 404 para DELETE en /users	PASS
13	GET / devuelve informacion de la API	PASS
14	Flujo completo: crear multiples usuarios y listar	PASS
15	Persistencia de usuarios en memoria durante sesion	PASS

Cuadro 3: Resumen de resultados de pruebas unitarias

7. Análisis de Resultados

7.1. Análisis de Validación

Los resultados de las pruebas unitarias demuestran que la API cumple con todos los requisitos funcionales establecidos:

1. **Operaciones CRUD básicas:** Las pruebas confirman que los endpoints GET y POST funcionan correctamente, permitiendo listar y crear usuarios.
2. **Validaciones de entrada:** El sistema valida correctamente los campos requeridos (name y email), el formato del email mediante expresiones regulares, y la longitud mínima del nombre.
3. **Manejo de errores:** La API responde apropiadamente con códigos de estado HTTP correctos (200, 201, 400, 404) según el escenario.
4. **Cobertura de código:** Se alcanzó una cobertura del 95.12 %, superando el objetivo del 90 %.

7.2. Análisis de Verificación

La verificación con ESLint demostró que el código sigue las buenas prácticas de programación:

- Se utiliza `const` y `let` en lugar de `var`.
- Se emplean operadores de comparación estricta (`===` y `!==`).
- El código mantiene una indentación consistente de 4 espacios.
- Se utilizan comillas simples para cadenas de texto.
- Todas las sentencias terminan con punto y coma.

8. Conclusiones

1. **Importancia de la verificación y validación:** Se demostró que la implementación de técnicas de verificación (ESLint) y validación (Jest + Supertest) es fundamental para garantizar la calidad del software. Estas herramientas permiten detectar errores de manera temprana en el ciclo de desarrollo, reduciendo significativamente el costo de corrección de defectos.
2. **Cobertura de código como métrica de calidad:** Se logró alcanzar una cobertura de código del 95.12 %, lo cual evidencia que las pruebas unitarias implementadas cubren la mayoría de los escenarios de uso de la API. Esta métrica es un indicador importante de la robustez del sistema y proporciona confianza en la estabilidad del código.

3. **Buenas prácticas de programación:** La configuración de ESLint con reglas personalizadas permitió mantener un código consistente, legible y alineado con los estándares de la industria, facilitando el mantenimiento y la colaboración en equipo.
4. **Automatización del proceso de testing:** La integración de scripts en el `package.json` permite ejecutar pruebas y verificaciones de manera automatizada, lo cual es esencial para la implementación de prácticas de integración continua (CI/CD).

9. Recomendaciones

1. **Implementar integración continua:** Se recomienda configurar un pipeline de CI/CD (por ejemplo, con GitHub Actions o GitLab CI) que ejecute automáticamente las pruebas y verificaciones de ESLint en cada commit, asegurando que el código que se integra al repositorio cumpla con los estándares de calidad establecidos.
2. **Expandir las pruebas de integración:** Aunque las pruebas actuales cubren los escenarios principales, se recomienda agregar pruebas de integración más complejas que simulen escenarios de uso real, incluyendo pruebas de carga y rendimiento para evaluar el comportamiento de la API bajo condiciones de estrés.
3. **Implementar persistencia de datos:** Para un ambiente de producción, se recomienda reemplazar el almacenamiento en memoria por una base de datos real (como MongoDB o PostgreSQL), manteniendo las mismas pruebas para validar que la funcionalidad no se vea afectada.
4. **Documentar la API con Swagger/OpenAPI:** Se recomienda implementar documentación automática de la API utilizando herramientas como Swagger, lo cual facilitaría el consumo de la API por parte de otros desarrolladores y sistemas.

10. Referencias Bibliográficas

1. Express.js. (2024). *Express - Node.js web application framework*. Recuperado de: https://expressjs.com/
2. Jest. (2024). *Jest - Delightful JavaScript Testing*. Recuperado de: https://jestjs.io/
3. ESLint. (2024). *ESLint - Pluggable JavaScript linter*. Recuperado de: https://eslint.org/
4. Supertest. (2024). *SuperTest - HTTP assertions library*. Recuperado de: https://github.com/ladjs/supertest
5. Node.js. (2024). *Node.js Documentation*. Recuperado de: https://nodejs.org/docs/