UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

Departamento de Ciencias de la Computación

Sede Sangolquí

Práctica Integral

Metodología de Pruebas de Software

Sistema de Gestión de Reservas

Carrera:
Ingeniería de Software

Docente:
Ing. Enrique Calvopiña E, Mgtr.

Asignatura:
Pruebas de Software

NRC:
27873

Nivel:
6to Nivel

Período:
202555

Estudiante:

Mesias Orlando Mariscal Oña

momariscal@espe.edu.ec

Sangolquí, Febrero 2026

Resumen Ejecutivo

Este informe documenta la ejecución de un programa integral de pruebas de software aplicando 5 fases metodológicas al Sistema de Gestión de Reservas. Se utilizaron las siguientes herramientas especializadas: ESLint para análisis estático, Jest y Supertest para pruebas unitarias e integración, Postman para pruebas funcionales de API, k6 para pruebas de carga y rendimiento, y GitHub Actions para integración continua.

Resultados principales:

- **23 pruebas automatizadas** implementadas (unitarias + integración)
- **98.61 % de cobertura de código** (superando la meta del 80 %)
- **100 % de pruebas pasadas** exitosamente
- **0 vulnerabilidades críticas** detectadas
- **Pipeline CI/CD** completamente automatizado

Palabras clave: Pruebas de Software, Jest, ESLint, k6, CI/CD, API REST, JWT

Índice

1 Introducción

1.1 Presentación del Proyecto

El presente informe documenta la aplicación sistemática de técnicas de pruebas de software al **Sistema de Gestión de Reservas**, una API REST desarrollada con Node.js, Express y MongoDB que implementa autenticación basada en JWT.

1.2 Objetivo General

Aplicar un conjunto integral de técnicas de pruebas de software mediante un enfoque metodológico estructurado, integrando pruebas estáticas, dinámicas, unitarias, de integración, de sistema, de seguridad, de rendimiento y automatizadas en un proyecto web de producción.

1.3 Objetivos Específicos

1. Ejecutar análisis estático del código fuente para identificar bugs, vulnerabilidades y code smells
2. Implementar pruebas unitarias con Jest alcanzando cobertura superior al 80 %
3. Realizar pruebas de integración de la API REST con Supertest
4. Ejecutar pruebas de carga y rendimiento con k6
5. Configurar un pipeline de CI/CD con GitHub Actions
6. Documentar todos los hallazgos y resultados obtenidos

2 Marco Teórico

2.1 Pruebas Estáticas

Las pruebas estáticas analizan el código sin ejecutarlo. Incluyen revisión de código, análisis estático automatizado (linting) y verificación de estándares de codificación.

2.2 Pruebas Unitarias

Verifican el comportamiento de unidades individuales de código (funciones, métodos) de forma aislada, utilizando mocks para simular dependencias externas.

2.3 Pruebas de Integración

Evalúan la interacción entre múltiples componentes del sistema, verificando que trabajen correctamente en conjunto.

2.4 Pruebas de Carga

Miden el rendimiento del sistema bajo diferentes niveles de carga, identificando cuellos de botella y límites de capacidad.

2.5 Integración Continua (CI/CD)

Práctica donde los desarrolladores integran código frecuentemente, verificado por construcciones automatizadas para detectar errores tempranamente.

3 Materiales y Herramientas

Herramienta	Versión	Propósito
Node.js	v20+	Entorno de ejecución
npm	v10+	Gestor de paquetes
ESLint	9.x	Análisis estático
Jest	30.x	Pruebas unitarias
Supertest	7.x	Pruebas de integración HTTP
k6	Latest	Pruebas de carga
MongoDB	8.x	Base de datos
GitHub Actions	-	CI/CD

Cuadro 1: Herramientas utilizadas en el laboratorio

4 Arquitectura del Sistema

El sistema implementa una arquitectura de tres capas con separación clara de responsabilidades:

- **Capa de Rutas:** Define los endpoints de la API REST
- **Capa de Controladores:** Implementa la lógica de negocio
- **Capa de Modelos:** Define los esquemas de datos con Mongoose
- **Middleware:** Maneja la autenticación JWT

4.1 Estructura del Proyecto

```
EvaluacionPractica1-reservas/
          src/
                app.js                  # Configuracion
    Express
                server.js               # Punto de entrada
                controllers/
                      authController.js  # Logica de
    autenticacion
                      reservaController.js
```

```
                middlewares/
                      auth.js              # Middleware JWT
                models/
                      User.js
                      Reserva.js
                routes/
                    auth.js
                    reserva.js
          tests/
                unit/
                integration/
          k6-tests/
          .github/workflows/
```

5 FASE 1: Análisis Estático

5.1 Configuración de ESLint

Se configuró ESLint con el plugin de seguridad para detectar vulnerabilidades potenciales:

```
// eslint.config.mjs
import js from "@eslint/js";
import security from "eslint-plugin-security";

export default [
  js.configs.recommended,
  {
    plugins: { security },
    rules: {
      "no-unused-vars": "warn",
      "security/detect-object-injection": "warn",
      "security/detect-unsafe-regex": "error",
      "security/detect-eval-with-expression": "error",
      "eqeqeq": "error",
      "no-eval": "error"
    }
  }
];
```

5.2 Resultados del Análisis

Categoría	Encontrados	Severidad
Errores	0	-
Advertencias	0	-
Vulnerabilidades de Seguridad	0	-

Cuadro 2: Resultados del análisis estático con ESLint

El código pasó todas las verificaciones de análisis estático sin errores ni advertencias críticas.

6 FASE 2: Pruebas Unitarias con Jest

6.1 Configuración de Jest

```
// jest.config.js
module.exports = {
  testEnvironment: 'node',
  coverageDirectory: 'coverage',
  collectCoverageFrom: ['src/**/*.js', '!src/server.js'],
  coverageThreshold: {
    global: {
      branches: 80, functions: 80, lines: 80, statements: 80
    }
  },
  testMatch: ['**/tests/**/*.test.js'],
  verbose: true
};
```

6.2 Pruebas del Controlador de Autenticación

```
// tests/unit/auth.test.js
describe('AuthController - Pruebas Unitarias', () => {
  describe('register()', () => {
    test('CP-01: Debe registrar un usuario nuevo', async ()
       => {
      mockReq.body = { email: 'test@test.com', password: '
         pass123' };
      mockUser.findOne.mockResolvedValue(null);
      mockUser.save.mockResolvedValue({});
      bcrypt.hash.mockResolvedValue('hashedPassword');

      await register(mockReq, mockRes);

      expect(mockRes.status).toHaveBeenCalledWith(201);
```

```
        expect(mockRes.json).toHaveBeenCalledWith({ msg: '
            Usuario creado' });
      });

      test('CP-02: Debe rechazar usuario duplicado', async ()
          => {
        mockReq.body = { email: 'existing@test.com', password:
            'pass123' };
        mockUser.findOne.mockResolvedValue({ email: '
            existing@test.com' });

        await register(mockReq, mockRes);

        expect(mockRes.status).toHaveBeenCalledWith(400);
      });
    });
});
```

6.3 Casos de Prueba Unitarios Implementados

ID	Descripción	Resultado Esperado	Estado
CP-01	Registrar usuario nuevo	201 Created	✓
CP-02	Rechazar usuario duplicado	400 Error	✓
CP-03	Manejar error interno registro	500 Error	✓
CP-04	Login con credenciales válidas	Token JWT	✓
CP-05	Login usuario inexistente	400 Error	✓
CP-06	Login contraseña incorrecta	400 Error	✓
CP-07	Error de BD en login	500 Error	✓
CP-08	Crear reserva válida	201 Created	✓
CP-09	Error al crear reserva	500 Error	✓
CP-10	Asociar reserva a usuario	userId correcto	✓
CP-11	Acceso sin token	401 Unauthorized	✓
CP-12	Acceso con token válido	next() llamado	✓
CP-13	Token inválido	400 Error	✓
CP-14	Token expirado	400 Error	✓
CP-15	Token sin Bearer	Procesado	✓

Cuadro 3: Casos de prueba unitarios - 15 pruebas pasadas

7 FASE 3: Pruebas de Integración

7.1 Configuración con Supertest y MongoDB Memory Server

```
// tests/integration/api.test.js
const request = require('supertest');
const { MongoMemoryServer } = require('mongodb-memory-server
    ');
```

```
beforeAll(async () => {
  mongoServer = await MongoMemoryServer.create();
  process.env.MONGO_URI = mongoServer.getUri();
  app = require('../../src/app');
});

describe('POST /api/auth/register', () => {
  test('CP-INT-01: Registro exitoso', async () => {
    const response = await request(app)
      .post('/api/auth/register')
      .send({ email: 'nuevo@usuario.com', password: '
        password123' });

    expect(response.status).toBe(201);
    expect(response.body.msg).toBe('Usuario creado');
  });
});
```

7.2 Casos de Prueba de Integración

ID	Endpoint	Descripción	Estado
CP-INT-01	POST /api/auth/register	Registro exitoso	✓
CP-INT-02	POST /api/auth/register	Rechaza duplicado	✓
CP-INT-03	POST /api/auth/login	Login exitoso	✓
CP-INT-04	POST /api/auth/login	Contraseña incorrecta	✓
CP-INT-05	POST /api/auth/login	Usuario inexistente	✓
CP-INT-06	POST /api/reservas	Crear con token válido	✓
CP-INT-07	POST /api/reservas	Rechaza sin token (401)	✓
CP-INT-08	POST /api/reservas	Rechaza token inválido	✓

Cuadro 4: Casos de prueba de integración - 8 pruebas pasadas

8 FASE 4: Pruebas de Carga con k6

8.1 Script de Prueba de Carga

```
// k6-tests/simple-load.js
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
  vus: 50,
  duration: '30s',
  thresholds: {
    http_req_duration: ['p(95)<2000'],
    http_req_failed: ['rate<0.1'],
```

```
  },
};

export default function() {
  const user = {
    email: `load_${Date.now()}@test.com`,
    password: 'LoadTest123!'
  };

  // Registro
  const registerRes = http.post(`${BASE_URL}/api/auth/
     register`,
    JSON.stringify(user), { headers: { 'Content-Type': '
       application/json' }});
  check(registerRes, { 'registro OK': (r) => r.status === 201
      });

  // Login y Reserva...
  sleep(0.5);
}
```

8.2 Resultados de Pruebas de Carga

Métrica	Valor	Umbral
Usuarios Virtuales (VUs)	50	-
Requests Totales	127	-
Tiempo Respuesta (avg)	1.2s	-
Tiempo Respuesta (p95)	4.33s	$<$2s
Tasa de Errores	0.00 %	$<$10 %
Throughput	18.8 req/s	-

Cuadro 5: Métricas de pruebas de carga con k6

Análisis: El sistema maneja correctamente la carga con 0 % de errores, aunque el p95 excede el umbral de 2s bajo alta concurrencia.

9 FASE 5: Automatización CI/CD

9.1 Pipeline de GitHub Actions

```
# .github/workflows/ci.yml
name: CI Pipeline

on:
  push:
    branches: [ main, develop ]
```

```yaml
  pull_request:
    branches: [ main ]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: '20' }
      - run: npm ci
      - run: npm run lint

  unit-tests:
    runs-on: ubuntu-latest
    needs: lint
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
      - run: npm ci
      - run: npm run test:unit

  integration-tests:
    runs-on: ubuntu-latest
    needs: unit-tests
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
      - run: npm ci
      - run: npm run test:integration
```

9.2 Jobs del Pipeline

1. **lint** - Análisis estático con ESLint
2. **unit-tests** - Pruebas unitarias con Jest
3. **integration-tests** - Pruebas de integración
4. **security** - Auditoría de dependencias (npm audit)
5. **build** - Verificación de inicio del servidor
6. **quality-gate** - Resumen final de calidad

10 Resultados Consolidados

10.1 Métricas de Calidad

Métrica	Meta	Resultado	Estado
Cobertura de código	≥80 %	**98.61 %**	✓
Pruebas pasadas	100 %	**100 %**	✓
Errores de linting	0	**0**	✓
Vulnerabilidades críticas	0	**0**	✓
Tasa de errores (carga)	<10 %	**0 %**	✓

Cuadro 6: Métricas de calidad del software

10.2 Cobertura por Archivo

Archivo	Stmts	Branch	Funcs	Lines
authController.js	100 %	100 %	100 %	100 %
reservaController.js	100 %	100 %	100 %	100 %
auth.js (middleware)	100 %	100 %	100 %	100 %
User.js	100 %	100 %	100 %	100 %
Reserva.js	100 %	100 %	100 %	100 %
auth.js (routes)	100 %	100 %	100 %	100 %
reserva.js (routes)	100 %	100 %	100 %	100 %
app.js	90.9 %	100 %	50 %	90.9 %
TOTAL	**98.61 %**	**100 %**	**83.33 %**	**98.52 %**

Cuadro 7: Cobertura de código por archivo

10.3 Resumen de Pruebas

Tipo de Prueba	Total	Pasadas	Fallidas
Pruebas Unitarias	15	15	0
Pruebas de Integración	8	8	0
TOTAL	**23**	**23**	**0**

Cuadro 8: Resumen de ejecución de pruebas

11 Matriz de Trazabilidad

Requisito	Unitaria	Integración	Carga	Seguridad
REQ-01: Crear reserva	CP-08,09	CP-INT-06,07,08	✓	✓
REQ-02: Autenticación	CP-04,05,06	CP-INT-03,04,05	✓	✓
REQ-03: Registro usuarios	CP-01,02,03	CP-INT-01,02	✓	✓
REQ-04: Validación tokens	CP-11,12,13,14	CP-INT-07,08	-	✓

Cuadro 9: Matriz de trazabilidad: Requisitos vs Pruebas

12 Conclusiones

1. **Alta Cobertura de Código:** Se logró una cobertura del 98.61 %, superando significativamente la meta del 80 %. Esto garantiza que prácticamente todo el código está respaldado por pruebas automatizadas, reduciendo el riesgo de regresiones.
2. **Robustez del Sistema de Autenticación:** Las pruebas demuestran que el sistema de autenticación basado en JWT maneja correctamente todos los escenarios: tokens válidos, inválidos, expirados, y solicitudes sin autenticación.
3. **Estabilidad bajo Carga:** Las pruebas con k6 mostraron que el sistema mantiene una tasa de errores del 0 % bajo carga de 50 usuarios concurrentes, demostrando buena estabilidad operacional.
4. **Automatización Efectiva:** El pipeline de CI/CD implementado con GitHub Actions asegura que cada cambio en el código pasa por análisis estático, pruebas unitarias y de integración antes de ser integrado, garantizando la calidad continua del software.

13 Recomendaciones

1. **Optimizar Tiempo de Respuesta:** El p95 de 4.33s bajo carga excede el umbral de 2s. Se recomienda implementar caching, optimizar queries y considerar escalado horizontal.
2. **Implementar Rate Limiting:** Agregar limitación de peticiones para prevenir ataques de fuerza bruta en endpoints de autenticación.
3. **Expandir Funcionalidades:** Implementar endpoints GET, PUT y DELETE para reservas, junto con sus respectivas pruebas.
4. **Pruebas E2E:** Considerar implementar pruebas end-to-end con herramientas como Playwright para flujos completos de usuario.
5. **Monitoreo en Producción:** Implementar APM (Application Performance Monitoring) para detectar problemas en tiempo real.

Repositorio

El código fuente completo del proyecto está disponible en:
`https://github.com/AMVMesias/-Examen-Pr-ctico-Parcial-3-Pruebas`

Anexo: Comandos de Ejecución

```
# Instalar dependencias
npm install

# Ejecutar todas las pruebas con cobertura
```

```
npm test

# Ejecutar solo pruebas unitarias
npm run test:unit

# Ejecutar solo pruebas de integracion
npm run test:integration

# Analisis estatico
npm run lint

# Pruebas de carga con k6
k6 run k6-tests/simple-load.js
```

Referencias

[1] Jest, "Jest - Delightful JavaScript Testing," Jest Documentation, 2026. [Online]. Available: https://jestjs.io/

[2] ESLint, "ESLint - Find and fix problems in your JavaScript code," ESLint Documentation, 2026. [Online]. Available: https://eslint.org/

[3] k6, "k6 - Modern load testing for developers," k6 Documentation, 2026. [Online]. Available: https://k6.io/

[4] GitHub, "GitHub Actions Documentation," GitHub Docs, 2026. [Online]. Available: https://docs.github.com/en/actions

[5] Supertest, "Supertest - Super-agent driven library for testing HTTP servers," npm, 2026. [Online]. Available: https://www.npmjs.com/package/supertest