ESPE UNIVERSIDAD DE LAS FUERZAS ARMADAS INNOVACIÓN PARA LA EXCELENCIA	INFORME DE PRÁCTICA DE LABORATORIO

DEPARTAMENTO:	**Ciencias de la Computación**	**CARRERA:**	**Ingeniería de Software**		
ASIGNATURA:	Pruebas de Software	**NIVEL:**	Sexto	**FECHA:**	29/05/2025
DOCENTE:	Ing. Luis Castillo, Mgt	**PRÁCTICA N°:**	2	**CALIFICACIÓN:**	

Verificación y Validación

Allan Vinicio Panchi Pillajo

RESUMEN

Este informe detalla la implementación y prueba de una API RESTful básica para la gestión de usuarios utilizando Node.js y Express.js. El objetivo principal fue desarrollar una API funcional capaz de crear y consultar usuarios, adhiriéndose a las mejores prácticas de diseño de APIs y manteniendo una alta disciplina en el desarrollo.

Se exploraron y corrigieron problemas críticos relacionados con la configuración de rutas y controladores, lo que garantizó que la API respondiera con los códigos de estado HTTP semánticamente correctos La verificación de la API se realizó exhaustivamente mediante pruebas unitarias implementadas con Jest y Supertest, destacando la importancia de resetear el estado entre pruebas para asegurar su fiabilidad.

Asimismo, se configuró ESLint para mantener la consistencia y calidad del código, resolviendo un error de tipografía en su configuración. La práctica subrayó que la precisión en la configuración de frameworks y el uso disciplinado de herramientas de análisis estático son tan cruciales como la lógica de negocio para construir sistemas robustos y mantenibles.

Palabras Claves: API RESTful, Node.js, Express.js, Controladores, Enrutamiento, Validación, Verificación

1. INTRODUCCIÓN:

Este informe detalla las actividades realizadas en el laboratorio, enfocadas en la implementación y verificación de una API RESTful básica para la gestión de usuarios. El objetivo principal fue desarrollar una API funcional que permitiera la creación y consulta de usuarios, siguiendo las mejores prácticas de diseño de APIs y aplicando una disciplina rigurosa en el manejo del código y la configuración de herramientas. Se enfatizó la importancia de la correcta configuración de rutas y controladores para asegurar que la API respondiera con los códigos de estado HTTP esperados (200 OK y 201 Created), así como la integración de pruebas unitarias para validar su comportamiento. Además, se abordó la configuración de un linter (ESLint) para mantener la consistencia y calidad del código, un aspecto fundamental en el desarrollo de software profesional.

2. OBJETIVO(S):

- Diseñar e implementar una API RESTful para la gestión de usuarios utilizando tecnologías como nodejs y express, asegurándonos que incluya las operaciones de creación de usuarios y consulta de los mismos.
- Asegurar que la parte del controlador nos mande las correctas respuestas de las API's con los códigos de estado HTTP apropiados para cada tipo de solicitud, definidos en la clase controlador.
- Integrar y ejecutar pruebas unitarias utilizando supertest y Jest para validar el comportamiento de la API en diferentes escenarios, haciendo que retornen los códigos esperados tanto en casos válidos como inválidos.
- Configurar un entorno de desarrollo disciplinado mediante la inclusión de un linter para aplicar reglas de estilo y buenas prácticas de codificación para de esta forma realizar las verificaciones correspondientes a la configuración antes hecha.

3. MARCO TEÓRICO:

Para la realización de este laboratorio, se necesitan saber varios conceptos y herramientas fundamentales en el desarrollo backend con Node.js y la arquitectura REST, también se debe saber conceptos como validación y verificación para comprobar si se están viendo los mismos dentro del laboratorio:

Las API RESTful son una interfaz de programación de aplicaciones (API) que sigue los principios de la arquitectura REST, estas se basan en recursos identificables por URIs (Richardson & Ruby, 2007), y las operaciones sobre estos recursos se realizan mediante métodos HTTP estándar como lo son el GET para leer, POST para crear, PUT/PATCH para actualizar y DELETE para eliminar. Un aspecto crucial dentro de las API REST es el uso de códigos de estado HTTP para indicar el resultado de una operación. (Fielding, 2000)

Dentro del desarrollo del backend, se usan tecnologías como Node.js el cual es un entorno de ejecución de JavaScript de código abierto, multiplataforma, que permite construir aplicaciones del lado del servidor de alto rendimiento, útil para realizar aplicaciones que tengan que ver con API's. (Tilkov & Vinoski, 2010), dentro del mismo node.js, express.js se presenta como un framework de aplicación web que proporciona un conjunto robusto de características para desarrollar APIs y aplicaciones web. Facilita el enrutamiento HTTP, el manejo de middlewares y la gestión de solicitudes/respuestas. (Holmes, 2013).

Dentro del desarrollo del laboratorio también se manejan los controladores, las cuales son funciones que encapsulan la lógica de negocio para manejar las solicitudes HTTP entrantes (Gamma et al., 1994). Son responsables de interactuar con el modelo de datos y de enviar la respuesta adecuada al cliente, el manejo de rutas también es un concepto a entender para este laboratorio, el cual es el proceso de determinar cómo una aplicación responde a una solicitud de cliente a una endpoint particular, que es una URI y un método HTTP específico. En Express, esto se logra a través de objetos Router que definen rutas como router.get('/') o router.post('/'). (Express.js Documentation, n.d.)

El uso de middlewares, es esencial para realizar el laboratorio ya que son funciones que tienen acceso al objeto de solicitud, al objeto de respuesta, y a la siguiente función middleware en el ciclo de solicitud-respuesta de la aplicación. Se utilizan para realizar tareas como el análisis del cuerpo de la solicitud o el manejo de errores. (Express.js Documentation, n.d.)

Aspectos que se van a ver en este laboratorio y que se espera que se evidencien, es la validación el cual es el proceso de asegurar que el sistema cumple con las necesidades del usuario o del negocio. Responde a la pregunta: "¿Estamos construyendo el producto correcto?". (IEEE, 1990), verificación el cual es el proceso de asegurar que el sistema cumple con sus especificaciones técnicas y requisitos funcionales. Responde a la pregunta: "¿Estamos construyendo el producto correctamente?" (IEEE, 1990). En este laboratorio, la verificación se llevó a cabo exhaustivamente a través de las pruebas unitarias usando Jest y Supertest. Al escribir tests que esperan un statusCode: 200 para GET, un statusCode: 201 para POST exitoso, y un statusCode: 400 para datos inválidos, se verificó que la implementación de la API se comportara exactamente según lo especificado. La corrección de errores de enrutamiento y asignación de controladores también fue parte del proceso de verificación, asegurando que el código ejecutara lo que se pretendía. Las pruebas unitarias son un nivel de prueba de software donde las "unidades" más pequeñas de una aplicación son probadas de forma aislada. En este laboratorio, se probaron las endpoints de la API con el framework Jest el cual es muy usado para javascript, desarrollado por Facebook, que es simple y ofrece un conjunto completo de características para probar código JavaScript, también se usa Supertest es una librería que facilita la prueba de aplicaciones HTTP. Permite simular solicitudes HTTP a la aplicación Express exportada y hacer aserciones sobre las respuestas y por último, se ve el Linting es el proceso de analizar el código fuente para detectar patrones que no cumplen con un conjunto de reglas, ya sea de estilo o de errores potenciales. ESLint es una herramienta configurable que ayuda a los desarrolladores a mantener la calidad y consistencia del código. Las "Flat Configs" (ESLint 9+) representan un nuevo formato de configuración que ofrece mayor flexibilidad y claridad al definir reglas y su aplicación a diferentes conjuntos de archivos. (Ammann & Offutt, 2017)

4. DESCRIPCIÓN DEL PROCEDIMIENTO:

4.1 Definición de Controladores (user.controller.js):

- 4.1.1 Se creó un arreglo global users para simular el almacenamiento de datos (sin una base de datos real).
- 4.1.2 Se implementó la función getAllUsers que devuelve el arreglo users con un código de estado 200 OK.
- 4.1.3 Se implementó la función createUser que recibe name y email del cuerpo de la solicitud, valida su presencia (devolviendo 400 Bad Request si faltan), crea un nuevo objeto usuario con un ID, lo añade al arreglo users, y devuelve el nuevo usuario con un código de estado 201 Created.
- 4.1.4 Se añadió una función resetUsers para limpiar el estado del arreglo users, crucial para asegurar la independencia de las pruebas.

4.2 Definición de Rutas (routes/user.routes.js):

- 4.2.1 • Se importaron los controladores createUser y getAllUsers.
- 4.2.2 • Se configuró el enrutador de Express (express.Router()).
- 4.2.3 • Se definieron las endpoints (corregidas):
  - 4.2.3.1 router.get('/', getAllUsers); para manejar las solicitudes GET y obtener todos los usuarios.
  - 4.2.3.2 router.post('/', createUser); para manejar las solicitudes POST y crear un nuevo usuario.

4.3 Configuración de la Aplicación Express (app.js):

- 4.3.1 Se inicializó la aplicación Express.
- 4.3.2 Se incluyó el middleware app.use(express.json()); para parsear los cuerpos de las solicitudes en formato JSON, permitiendo que req.body contenga los datos enviados en solicitudes POST.

4.3.3 Punto clave de corrección: Se montaron las rutas de usuario utilizando app.use('/api/users', routes);. Esto asegura que las solicitudes a /api/users sean manejadas por el Router definido en routes.js.

4.3.4 Se añadió un middleware de manejo de rutas no encontradas (404 Not Found) al final de la cadena.

4.4 Implementación de Pruebas Unitarias (test/user.test.js):

4.4.1 Se importaron supertest y la instancia de la aplicación app.

4.4.2 Se importó el userController para poder llamar a la función resetUsers.

4.4.3 Test GET /api/users: Se envió una solicitud GET a /api/users y se verificó que el statusCode fuera 200 y el body fuera un arreglo vacío.

4.4.4 Test POST /api/users: Se envió una solicitud POST a /api/users con un objeto newUser y se verificó que el statusCode fuera 201, y que el body de la respuesta tuviera una propiedad id, name, y email coincidentes.

4.4.5 Test GET /api/users: Se envió una solicitud GET a /api/users y se verificó que el statusCode fuera 200 y el body fuera con un usuario.

4.4.6 Test POST /api/users (manejo de errores): Se añadió un test para verificar que una solicitud POST con datos faltantes (name o email) retornara un 400 Bad Request.

4.5 **Configuración de ESLint (eslint.config.js):**

4.5.1 Se creó un archivo de configuración en formato "flat config".

4.5.2 Se especificó que las reglas se aplicaran a los archivos .js dentro de src/**.

4.5.3 Se configuró languageOptions para ecmaVersion: 2021 y sourceType: 'commonjs'.

4.5.4 Se incluyeron las reglas recomendadas por ESLint (...js.configs.recommended.rules).

4.5.5 Se añadieron reglas personalizadas para forzar el uso de punto y coma (semi: ['error', 'always']) y comillas simples (quotes: ['error', 'single']).

4.5.6 Punto clave de corrección: Se corrigió un error tipográfico en la clave de configuración de ESLint: languajeOptions fue cambiado a languageOptions.

5. ANÁLISIS DE RESULTADOS:

Prueba / Solicitud	Expectativa Inicial	Resultado Observado	Causa del Problema	Solución Implementada	Resultado Final
GET /api/users	statusCode: 200, body: []	statusCode: 500	Error de enrutamiento en app.js y asignación de controlador en routes.js: La combinación inicial de app.use('/api/users', routes); y router.get('/', createUser); no coincidía con el test GET /api/users.	Corrección en routes.js: Se asignó router.get('/', getAllUsers);. La estructura de enrutamiento en app.js (app.use('/api/users', routes);) fue clave para mapear el path correcto.	statusCode: 200, body: []
POST /api/users	statusCode: 201, body: { id, name, email }	statusCode: 200	Error de asignación de controlador en routes.js: router.post('/', getAllUsers); estaba asignado incorrectamente.	Corrección en routes.js: Se corrigió la asignación a router.post('/', createUser);	statusCode: 201, body: { id, name, email }
GET /api/users	statusCode: 200, body: { name: Allan }	statusCode: 200	No existen problemas al llamar al endpoint	No se realizaron correcciones	statusCode: 200, body: { name: Allan }
npm run lint	Linting exitoso	ConfigError: Unexpected key "languajeOptions" found.	Error tipográfico en eslint.config.js: La clave de configuración estaba	Corrección en eslint.config.js: Se corrigió el error de escritura a languageOptions.	Linting exitoso

			mal escrita como languajeOptions.		

6. GRÁFICOS O FOTOGRAFÍAS:

```
C:\RESPALDOS\D\Allan\UNIVERSIDAD\Septimo Semestre\Pruebas\Parcial1\Laboratorio2>mkdir src routes controllers test

C:\RESPALDOS\D\Allan\UNIVERSIDAD\Septimo Semestre\Pruebas\Parcial1\Laboratorio2>code .

C:\RESPALDOS\D\Allan\UNIVERSIDAD\Septimo Semestre\Pruebas\Parcial1\Laboratorio2>
```

Fig1. Creación de la estructura del proyecto.

```
C:\RESPALDOS\D\Allan\UNIVERSIDAD\Septimo Semestre\Pruebas\Parcial1\Laboratorio2>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (laboratorio2) laboratorio2
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\RESPALDOS\D\Allan\UNIVERSIDAD\Septimo Semestre\Pruebas\Parcial1\Laboratorio2\package.json:

{
  "name": "laboratorio2",
  "version": "1.0.0",
  "main": "index.js",
  "directories": {
```

Fig2. Instalación y Configuración del proyecto

```
C:\RESPALDOS\D\Allan\UNIVERSIDAD\Septimo Semestre\Pruebas\Parcial1\Laboratorio2>npm install --save-dev jest supertest eslint
npm warn deprecated inflight@1.0.6: This module is not supported, and leaks memory. Do not use it. Check out lru-cache if you want
nd tested way to coalesce async requests by a key value, which is much more comprehensive and powerful.
npm warn deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported

added 344 packages, and audited 411 packages in 2m

67 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Fig3. Instalación de paquetes.

user.controller.js × user.routes.js

controllers > user.controller.js > createUser

```
let user = [];
let id = 1;

const getAllUsers = (req, res) => {
  res.status(200).json({
    status: 'success',
    data: {
      user,
    },
  });
}

const createUser = (req, res) => {
    const {name, email} = req.body;
    if (!name || !email) {
        return res.status(400).json({
            status: 'fail',
            message: 'Please provide name and email',
        });
    }

    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    if (!emailRegex.test(email)) {
        return res.status(400).json({ message: 'Invalid email format.' });
    }
    if (name.length < 3) {
        return res.status(400).json({ message: 'Name must be at least 3 characters long.' });
    }
    const existingUser = users.find(user => user.email === email);
    if (existingUser) {
        return res.status(409).json({ message: 'User with this email already exists.' });
    }
```

Fig4. Funciones para crear y obtener usuarios en el controller

user.controller.js user.routes.js ×

routes > user.routes.js > ...

```
import {express} from 'express';
import { getAllUsers, createUser } from '../controllers/user.controller';

const router = express.Router();
// Define the routes for user operations
router.get('/', getAllUsers); // Get all users
router.post('/', createUser); // Create a new user

module.exports = router;
```

Fig5. Definición de rutas

app.js > ...

```
import {express} from 'express';
import {routes} from './routes/user.routes.js';

const app = express();

app.use(express.json()); // Middleware to parse JSON request bodies

app.use('/api/users', routes);

app.use((req, res, next) => {
    res.status(404).json({ message: 'Route not found' });
});

module.exports = app;
```

Fig6. Archivo App.js

user.test.js > ...

```
import {request} from 'supertest';
const app = require('../app');


beforeEach(() => {
    // Accedemos directamente al arreglo 'users' del controlador para resetearlo.
    // En una aplicación real, esto implicaría limpiar la base de datos.
    const userController = require('../controllers/user.controller');
    userController.users = []; // Resetear el arreglo de usuarios
    userController.nextId = 1; // Resetear el contador de IDs
});

describe('User API', () => {
    // d. Crear prueba que GET devuelva lista vacía inicialmente.
    test('GET /api/users should return an empty array initially', async () => {
        const response = await request(app).get('/api/users');

        expect(response.statusCode).toBe(200);
        expect(response.body).toEqual([]);
    });

    // e. Crear prueba que POST cree un nuevo usuario correctamente
    test('POST /api/users should create a new user successfully', async () => {
        const newUser = {
            name: 'John Doe',
            email: 'john.doe@example.com'
        };

        const response = await request(app)
            .post('/api/users')
            .send(newUser)
            .set('Accept', 'application/json');
```

Fig7. Codificación de las pruebas en el archivo .test.js

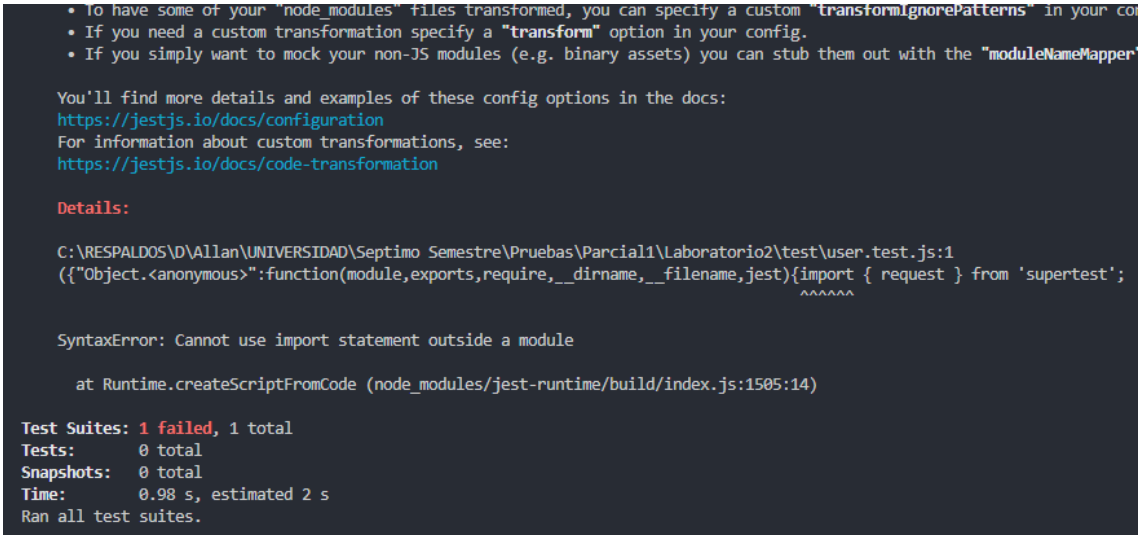

Fig8. Prueba no pasada por error en la importación de archivos

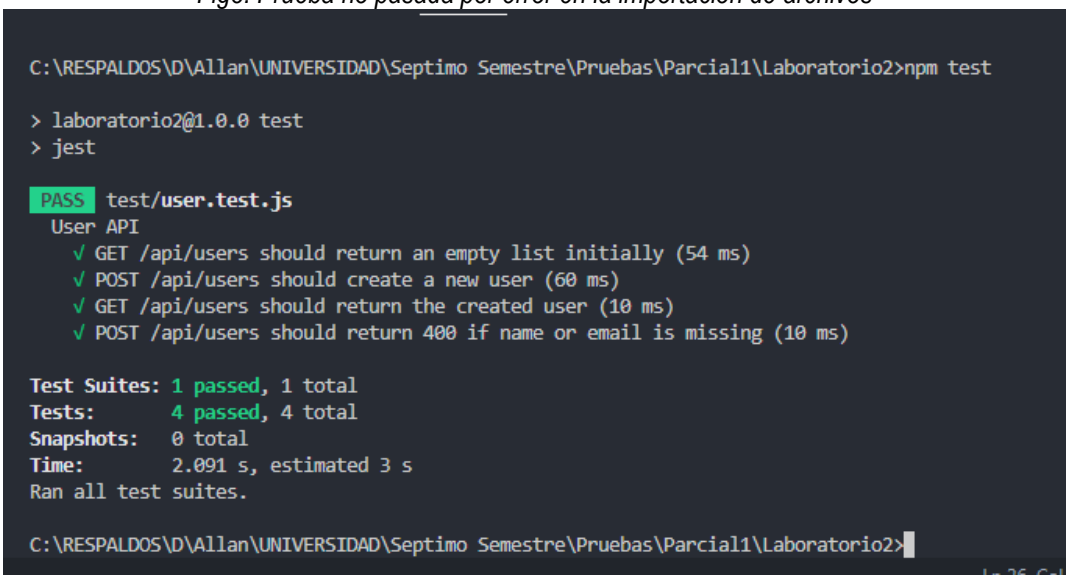

Fig9. Todas las pruebas aprobadas

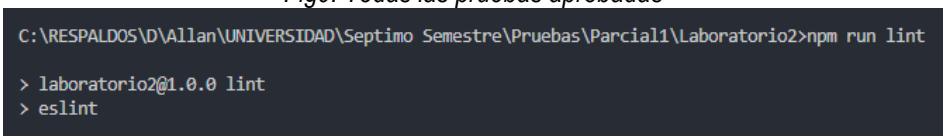

Fig10. Ejecución del comando run lint para aplicar las configuraciones del archivo eslint.js

7. DISCUSIÓN:

La implementación y prueba de la API RESTful para la gestión de usuarios, aunque conceptualmente sencilla, reveló la importancia crítica de la precisión en la configuración y la asignación de responsabilidades entre los diferentes componentes de una aplicación Node.js con Express. Los errores iniciales, manifestados en códigos de estado HTTP incorrectos (500 en lugar de 200 para GET, y 200 en lugar de 201 para POST), no se debieron a fallos lógicos complejos dentro de los controladores getAllUsers o createUser. En cambio, la causa principal fue una desalineación en el enrutamiento dentro de app.js y una inversión en la asignación de controladores a las rutas HTTP específicas en routes/user.routes.js.

La corrección de app.use('/api/users', routes); en app.js junto con la correcta asignación de router.get('/', getAllUsers); y router.post('/', createUser); en routes.js fue fundamental. Este ajuste aseguró que las solicitudes de supertest llegaran al controlador esperado, permitiendo que la lógica de la API (incluida la validación de name y email en el POST) se ejecutara y respondiera con los códigos de estado semánticamente correctos (200, 201, 400). Este proceso resaltó que, incluso con controladores lógicamente correctos, un enrutamiento mal configurado puede hacer que una API parezca fallar o comportarse de manera impredecible desde la perspectiva del cliente o del tester.

Otro aspecto clave de la discusión es el manejo del estado en las pruebas. Utilizar una variable global users para simular la base de datos es conveniente para una práctica sencilla, pero introduce el riesgo de "tests sucios", donde el resultado de una prueba influye en la siguiente

Finalmente, la experiencia con ESLint subrayó la relevancia de las herramientas de calidad de código. El error languajeOptions fue un simple error tipográfico, pero su detección por parte de ESLint evitó problemas futuros y reforzó la necesidad de un análisis estático riguroso. La configuración de reglas para forzar el uso de punto y coma y comillas simples ejemplifica cómo ESLint no solo detecta errores, sino que también impone un estilo de código consistente, lo que mejora la legibilidad y mantenibilidad del proyecto, especialmente en entornos de desarrollo colaborativo.

8. CONCLUSIONES:

- La precisión en la configuración del enrutamiento y la asignación de controladores es crucial para el correcto funcionamiento de una API RESTful. Un pequeño error en la definición de rutas en app.js o en la asignación de funciones a los métodos HTTP en routes.js puede llevar a comportamientos inesperados, incluso si la lógica de negocio dentro de los controladores es correcta. Esto subraya la importancia de una verificación exhaustiva a través de pruebas unitarias que cubran los diferentes escenarios de respuesta.

- La adopción de prácticas de desarrollo disciplinadas, como la implementación de pruebas unitarias robustas con reseteo de estado (beforeEach en Jest) y la integración de linters como ESLint, es fundamental para garantizar la calidad, fiabilidad y mantenibilidad del código. Las pruebas permiten verificar el comportamiento esperado de la API y detectar regresiones, mientras que el linting fuerza la consistencia estilística y ayuda a prevenir errores comunes, contribuyendo significativamente a la robustez del software desarrollado.

9. BIBLIOGRAFÍA:

Ammann, P., & Offutt, J. (2017). Introduction to Software Testing. Cambridge University Press.

ESLint Documentation. (n.d.). About ESLint. Recuperado de https://eslint.org/docs/latest/use/about/

Express.js Documentation. (n.d.). Routing. Recuperado de https://expressjs.com/en/guide/routing.html

Express.js Documentation. (n.d.). Using middleware. Recuperado de https://expressjs.com/en/guide/using-middleware.html

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Tesis doctoral, University of California, Irvine. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Holmes, J. (2013). Beginning Node.js. Apress.

IEEE (Institute of Electrical and Electronics Engineers). (1990). IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990).

Richardson, L., & Ruby, S. (2007). RESTful Web Services. O'Reilly Media.

Tilkov, S., & Vinoski, S. (2010). Node.js: Using JavaScript for Server-Side Programming. IEEE Internet Computing, 14(6), 80-83.