

DEPARTAMENTO:	Ciencias de la computación	CARRERA:	Software		
ASIGNATURA:	Pruebas de software	NIVEL:	6to	FECHA:	08/11/2025
DOCENTE:	Ing. LUIS ALBERTO CASTILLO SALINAS	PRÁCTICA N°:	2	CALIFICACIÓN:	

Verificación y Validación

Moisés Sebastián Benalcázar Farinango

RESUMEN

En la presente práctica se implementó una API REST para gestión de usuarios utilizando Node.js, Express, Jest + Supertest y ESLint, con el objetivo de aplicar técnicas de verificación y validación de software. Se estructuró el proyecto con carpetas src, routes, controllers y test, se configuró un servidor Express con endpoints GET y POST, y se simuló una base de datos en memoria mediante un arreglo. Se realizaron pruebas unitarias y end-to-end para validar el comportamiento de la API, logrando una cobertura superior al 90%. Además, se configuró ESLint con reglas personalizadas para garantizar buenas prácticas de codificación. Los resultados demuestran la importancia de integrar herramientas de testing y linting desde las primeras etapas del desarrollo para obtener código robusto, mantenible y libre de errores comunes.

Palabras Claves: Verificación, Validación, ESLint

1. INTRODUCCIÓN:

La práctica se centró en el uso disciplinado de herramientas modernas de desarrollo para garantizar la calidad del software mediante verificación (análisis estático del código) y validación (pruebas dinámicas). Se siguieron las indicaciones del docente, se documentó cada paso con capturas de pantalla y se utilizaron datos personales en las pruebas para evidenciar la autoría del trabajo, promoviendo buenas prácticas de programación y responsabilidad técnica en el entorno de laboratorio.

2. OBJETIVO(S):

- 2.1 Aplicar técnicas de verificación sobre el código.
- 2.2 Aplicar técnicas de validación sobre el código.
- 2.3 Generación de reglas para la verificación del código.
- 2.4 Creación de pruebas unitarias para la validación del código

3. MARCO TEÓRICO:

Node.js es un entorno de ejecución que permite correr JavaScript fuera del navegador, ideal para construir servidores backend. Express es un framework minimalista que simplifica la creación de APIs REST mediante enrutamiento y middlewares. Jest es un framework de pruebas unitarias con soporte para assertions, mocks y reportes de cobertura, mientras que Supertest permite realizar pruebas HTTP sobre aplicaciones Express sin necesidad de levantar un servidor real. ESLint es una herramienta de análisis estático que detecta errores, aplica reglas de estilo y fomenta buenas prácticas mediante configuraciones personalizables (eslint.config.js). Juntos, forman un flujo de desarrollo moderno enfocado en calidad y mantenibilidad.

4. DESCRIPCIÓN DEL PROCEDIMIENTO:

PARTE 1: Establecer la estructura del proyecto y configurar el ambiente de desarrollo

Paso 1: Estructura del proyecto.

- a. Se debe crear una estructura básica para un proyecto JS teniendo en cuenta:

- Src
- Routes
- Controllers
- Test
- Archivos de configuración

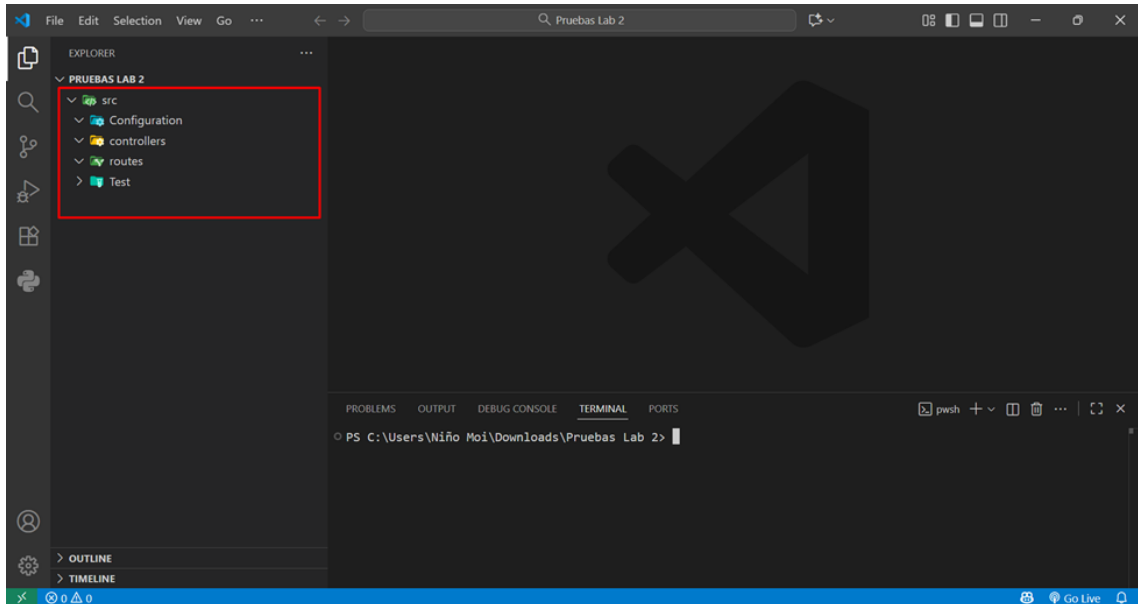


Ilustración 1: Estructura de carpetas y archivos para el laboratorio.

Paso 2: Instalar las tecnologías necesarias para el ambiente de desarrollo y pruebas.

- Crear el archivo package.json ejecutando el siguiente comando: `npm init`
- Instalar express: `npm install express`
- Instalar las librerías necesarias: `npm install --save-dev jest supertest eslint`

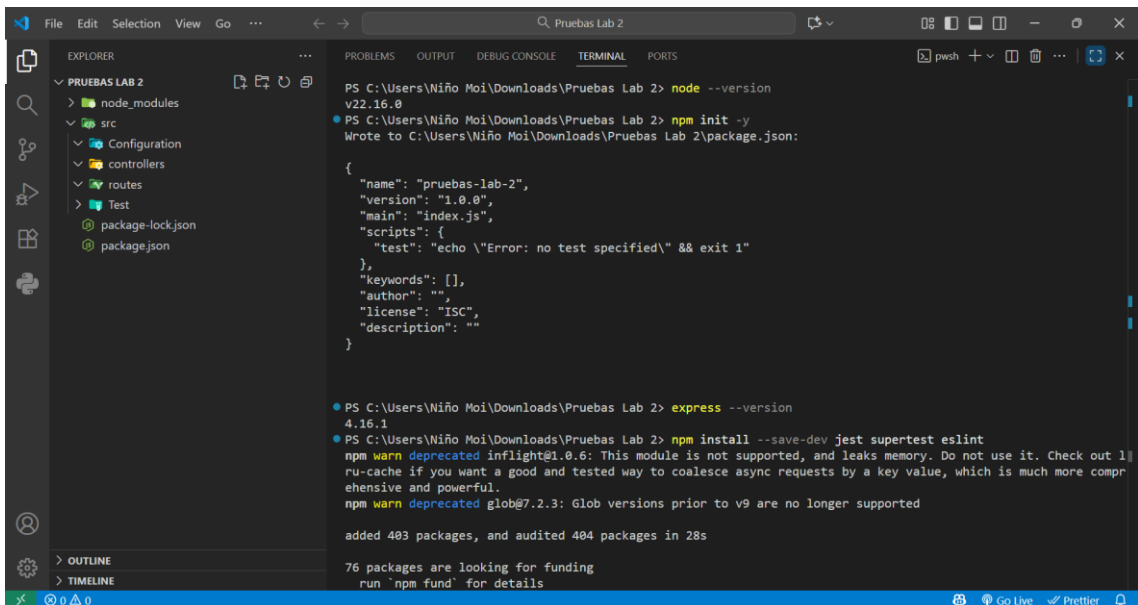


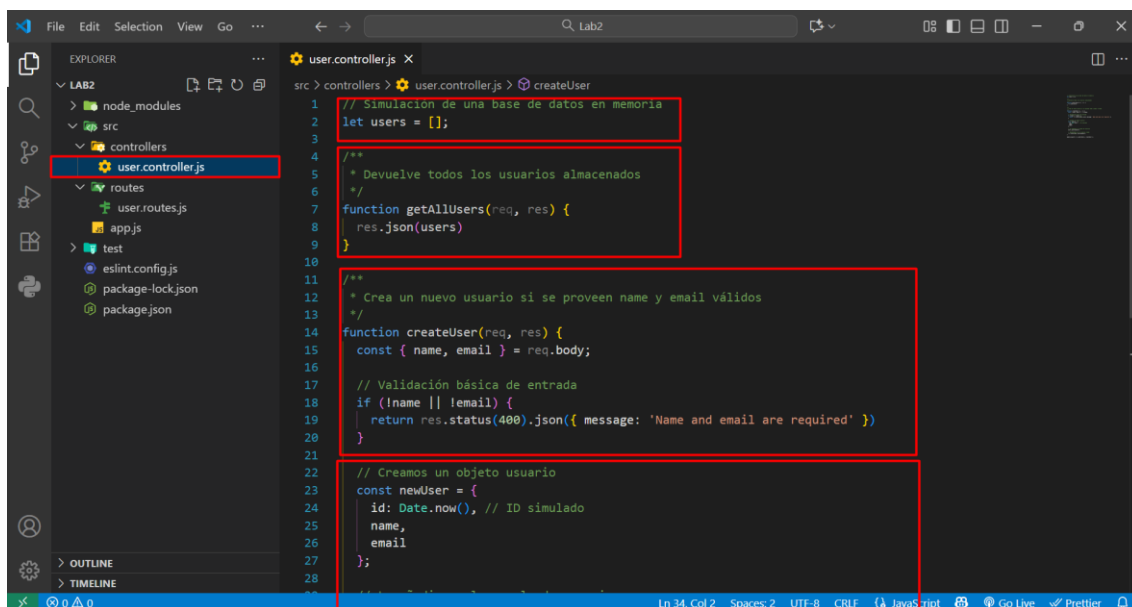
Ilustración 2: Se revisó y se instaló las herramientas necesarias para realizar el laboratorio como: express, jest, supertest y eslint.

PARTE 2: Creación de API de Gestión de Usuarios

Paso 1: Configurar el controlador.

- Crear el archivo controllers/user.controller.js.

- b. Simular de una base de datos en memoria (arreglo).
- c. Función para devolver todos los usuarios almacenados.
- d. Función para crear un nuevo usuario si se proveen nombre y correo válidos.
- e. Generar una validación básica de entrada
- f. Crear un objeto usuario y añadirlo al arreglo de usuarios
- g. Responder con el usuario creado status 201
- h. Exportar las funciones creadas con module.exports.



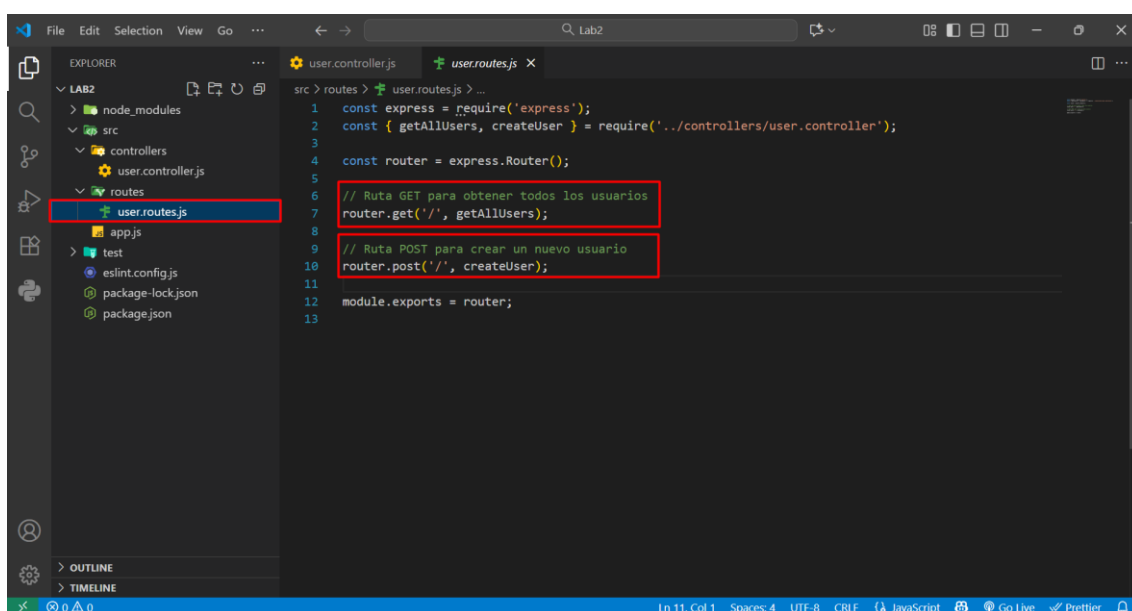
```

1 // Simulación de una base de datos en memoria
2 let users = [];
3
4 /**
5  * Devuelve todos los usuarios almacenados
6  */
7 function getAllUsers(req, res) {
8   res.json(users)
9 }
10
11 /**
12  * Crea un nuevo usuario si se proveen name y email válidos
13  */
14 function createUser(req, res) {
15   const { name, email } = req.body;
16
17   // Validación básica de entrada
18   if (!name || !email) {
19     return res.status(400).json({ message: 'Name and email are required' });
20   }
21
22   // Creamos un objeto usuario
23   const newUser = {
24     id: Date.now(), // ID simulado
25     name,
26     email
27   };
28
29   users.push(newUser);
30 }
31
32 module.exports = { getAllUsers, createUser };
  
```

Ilustración 3: configuración del controlador creando funciones para manejar usuarios, validando y almacenando en un arreglo simulado.

Paso 2: Configurar las rutas del servidor.

- a. Crear el archivo routes/user.routes.js.
- b. Importar los módulos necesarios: express y funciones del controlador.
- c. Definir ruta GET para obtener todos los usuarios.
- d. Definir ruta POST para crear un nuevo usuario.
- e. Exportar el router con module.exports.



```

1 const express = require('express');
2 const { getAllUsers, createUser } = require('../controllers/user.controller');
3
4 const router = express.Router();
5
6 // Ruta GET para obtener todos los usuarios
7 router.get('/', getAllUsers);
8
9 // Ruta POST para crear un nuevo usuario
10 router.post('/', createUser);
11
12 module.exports = router;
  
```

Ilustración 4: configuración las rutas del servidor, definir rutas GET y POST en un archivo de rutas para manejar usuarios.

Paso 3: Configurar la entrada principal.

- Crear el archivo `app.js`.
- Importar los módulos necesarios: `express` y el archivo donde se manejarán las rutas `user.routes.js`.
- Crear una instancia de la aplicación `Express`.
- Usar un `middleware` para parsear `JSON` del cuerpo de las solicitudes.
- Establecer la ruta base para los usuarios.
- Usar un manejador de rutas no encontradas (`404`).
- Exportar la instancia `app` para poder usarla en tests o en un archivo de servidor separado.

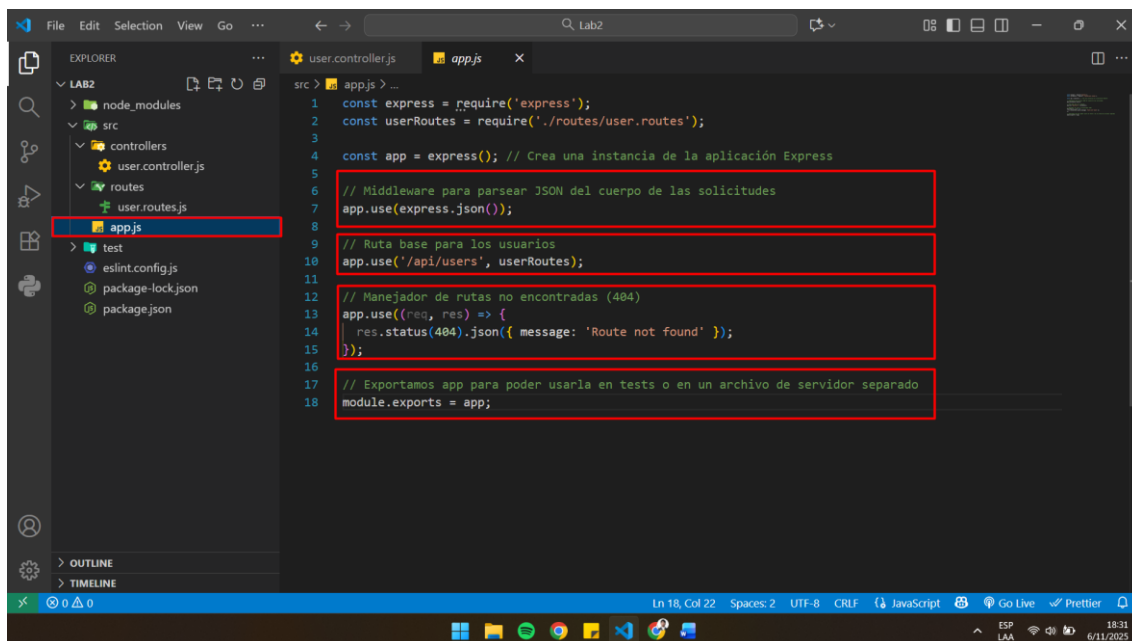
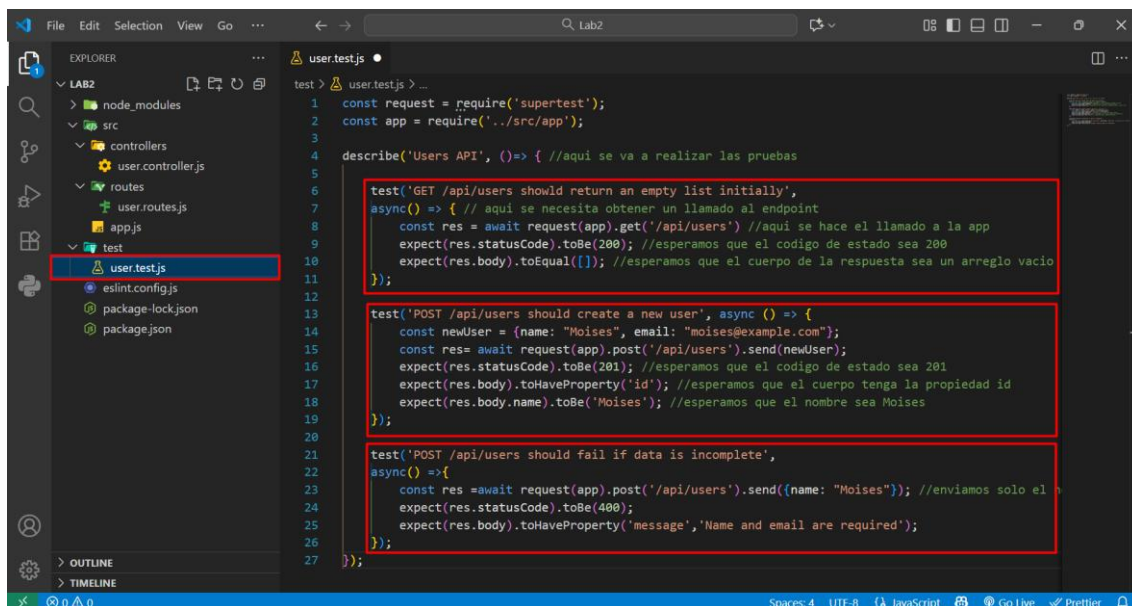


Ilustración 5: configuración la entrada principal, crear y configurar la aplicación `Express` con rutas y `middleware` para procesar solicitudes `JSON`.

PARTE 3: Verificación y validación**Paso 1:** Validación: Pruebas con `Jest` y `Supertest`.

- Crear el archivo `test/user.test.js`.
- Importar el cliente `HTTP` son `supertest` para pruebas.
- Importar `app Express`.
- Crear prueba que `GET` devuelva lista vacía inicialmente.
- Crear prueba que `POST` cree un nuevo usuario correctamente.
- Crear prueba que el endpoint rechace peticiones inválidas.
- Ejecutar las pruebas con el comando: `npm test`



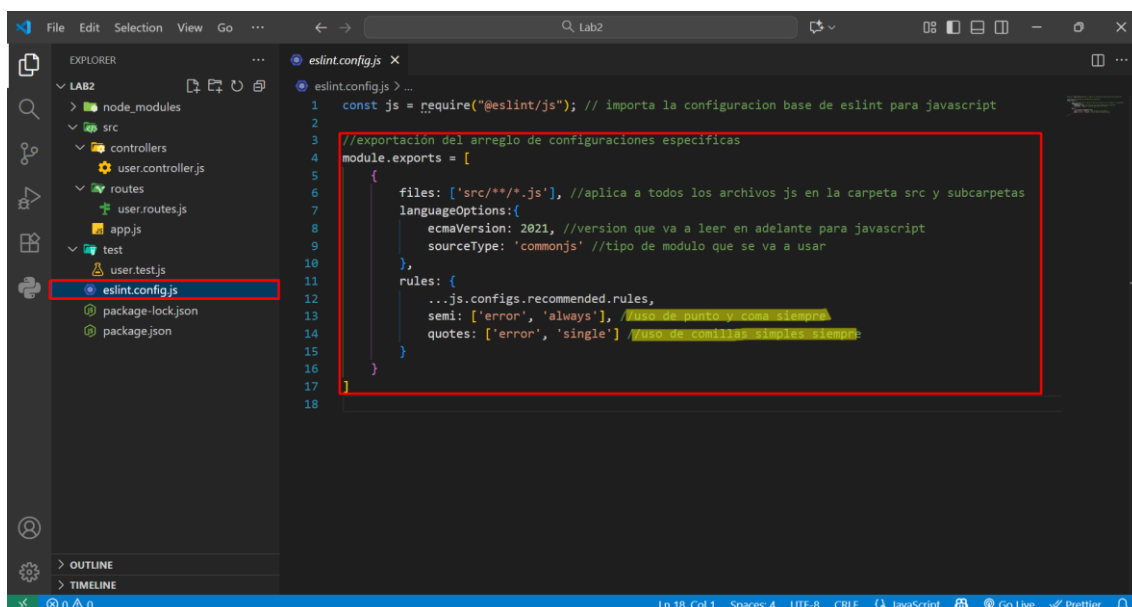
```

1  const request = require('supertest');
2  const app = require('../src/app');
3
4  describe('Users API', () => { //aquí se va a realizar las pruebas
5
6    test('GET /api/users should return an empty list initially',
7    async() => { // aquí se necesita obtener un llamado al endpoint
8      const res = await request(app).get('/api/users') //aquí se hace el llamado a la app
9      expect(res.statusCode).toBe(200); //esperamos que el código de estado sea 200
10     expect(res.body).toEqual([]); //esperamos que el cuerpo de la respuesta sea un arreglo vacío
11   });
12
13   test('POST /api/users should create a new user', async () => {
14     const newUser = {name: "Moises", email: "moises@example.com"};
15     const res= await request(app).post('/api/users').send(newUser);
16     expect(res.statusCode).toBe(201); //esperamos que el código de estado sea 201
17     expect(res.body).toHaveProperty('id'); //esperamos que el cuerpo tenga la propiedad id
18     expect(res.body.name).toBe('Moises'); //esperamos que el nombre sea Moises
19   });
20
21   test('POST /api/users should fail if data is incomplete',
22   async() =>{
23     const res =await request(app).post('/api/users').send({name: "Moises"}); //enviamos solo el
24     expect(res.statusCode).toBe(400);
25     expect(res.body).toHaveProperty('message','Name and email are required');
26   });
27 });
  
```

Ilustración 6: Creación de pruebas con Jest y Supertest para verificar el comportamiento de los endpoints GET y POST y rechazar peticiones inválidas.

Paso 2: Verificación: ESLint.

- Crear el archivo `eslint.config.js`.
- Importar la configuración base de reglas recomendadas de ESLint para JavaScript.
- Exportar un arreglo de configuraciones específicas para ESLint.
- Configurar los archivos dentro de la carpeta `src` con extensión `.js`.
- Gestionar opciones del lenguaje para estos archivos.
- Crear reglas de ESLint que se aplicarán a estos archivos.



```

1  const js = require("@eslint/js"); // importa la configuración base de eslint para javascript
2
3  //exportación del arreglo de configuraciones específicas
4  module.exports = [
5
6    {
7      files: ['src/**/*.js'], //aplica a todos los archivos js en la carpeta src y subcarpetas
8      languageOptions:{
9        ecmaVersion: 2021, //versión que va a leer en adelante para javascript
10       sourceType: 'commonjs' //tipo de módulo que se va a usar
11     },
12     rules: {
13       ...js.configs.recommended.rules,
14       semi: ['error', 'always'], //uso de punto y coma siempre
15       quotes: ['error', 'single'] //uso de comillas simples siempre
16     }
17   }
18 ];
  
```

Ilustración 7: Configuración de ESLint creando un archivo de configuración, importando reglas base y definiendo reglas específicas para los archivos `.js` en la carpeta `src`.

5. PREGUNTAS/ACTIVIDADES:

- Actividad: Implementar una ruta GET `/users` que retorne una lista de usuarios en memoria (array local).

```

1 // Simulación de una base de datos en memoria
2 let users = [
3   { name: 'Andres', email: 'andres@gmail.com' }
4 ];
5
6 /**
7  * Devuelve todos los usuarios almacenados
8  */
9 function getAllUsers(req, res) {
10   res.json(users)
11 }
12
13 /**
14  * Crea un nuevo usuario si se proveen name y email válidos
15  */
16 function createUser(req, res) {
17   const { name, email } = req.body;
18
19   // Validación básica de entrada
20   if (!name || !email) {
21     return res.status(400).json({ message: 'Name and email are required' });
22   }
23
24   // Creamos un objeto usuario
25   const newUser = {
26     id: Date.now(), // ID simulado
27     name,
28     email
29   };
30   users.push(newUser);
31   res.status(201).json(newUser);
32 }
33
34 module.exports = {
35   getAllUsers,
36   createUser
37 };

```

Ilustración 8: Creamos una lista en memoria de manera local para manejar la prueba solicitada.

```

PS C:\Users\Niño Moi\Downloads\Lab2> npm test
> lab2@1.0.0 test
> jest

FAIL test/user.test.js
  User API
    ✗ GET /api/users should return an empty list initially (68 ms)
    ✓ POST /api/users should return create a new user (44 ms)
    ✓ POST /api/users should fail if data is incomplete (9 ms)
    ✓ GET /api/ should return a list in memory (18 ms)

• User API > GET /api/users should return an empty list initially

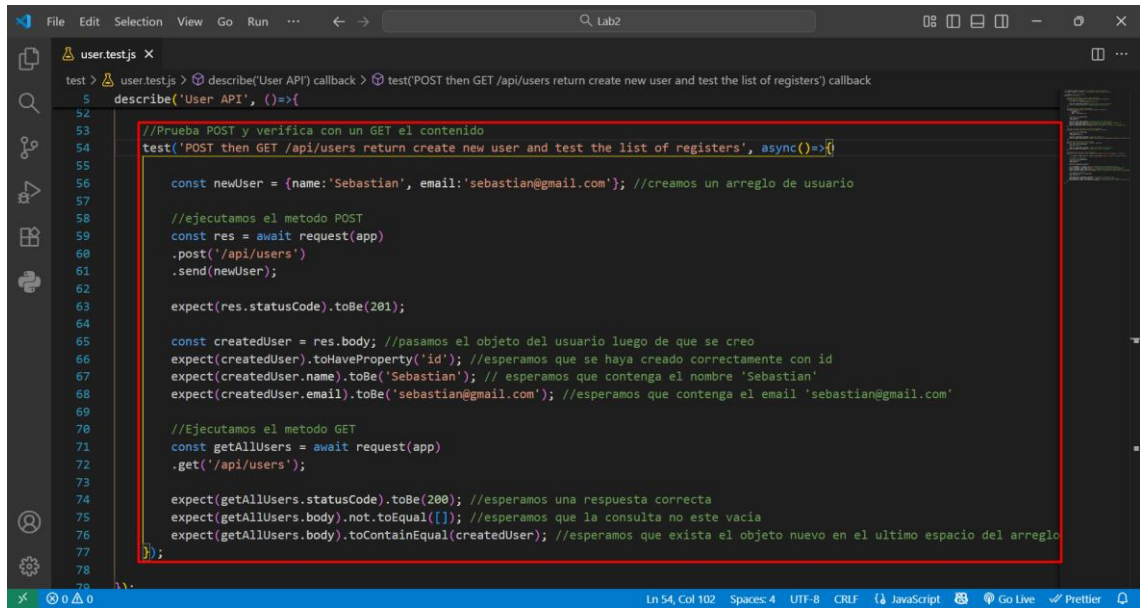
expect(received).toEqual(expected) // deep equality
- Expected   - 1
+ Received   + 6
- Array []
+ Array [
+   Object {
+     "email": "andres@gmail.com",
+     "name": "Andres",
+   },
+ ]

11 |
12 |   expect(res.statusCode).toBe(200); //esperamos que

```

Ilustración 9: Tenemos error al ejecutar la primera prueba ya que espera una lista vacía en el rectángulo rojo debido a que ya existe una lista local la cual se observa en el rectángulo azul, la prueba que se realizó y se aprobó en los rectángulos verdes.

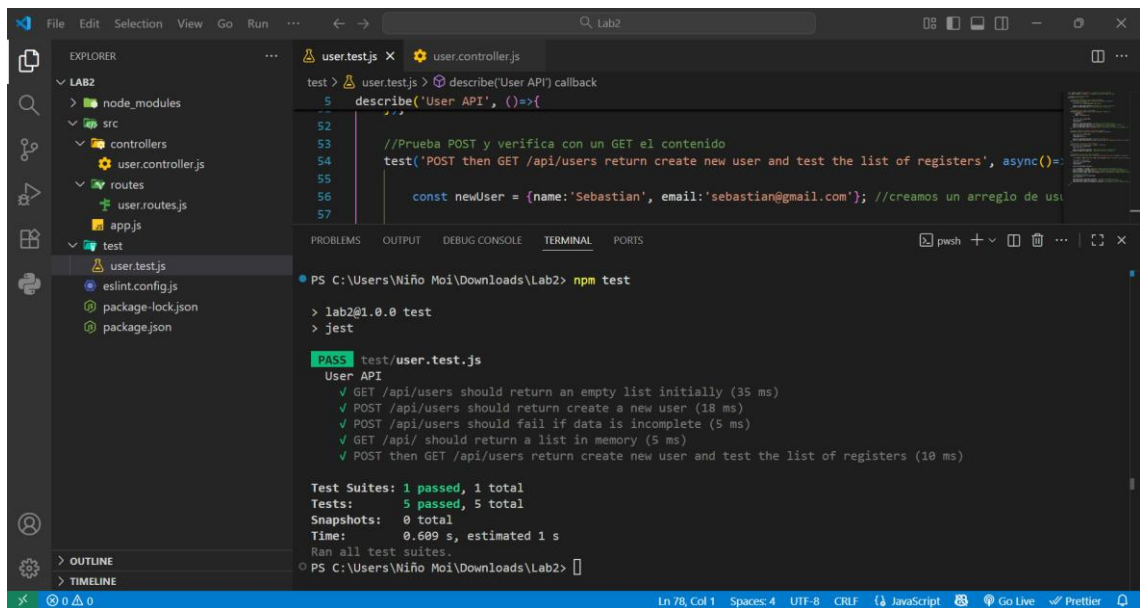
- b. Prueba: Crear usuarios con POST /users y luego verificar el contenido de GET /users.



```

test > user.test.js > describe('User API') callback > test('POST then GET /api/users return create new user and test the list of registers') callback
5 describe('User API', () => {
6
7 //Prueba POST y verifica con un GET el contenido
8 test('POST then GET /api/users return create new user and test the list of registers', async() => {
9
10 const newUser = {name: 'Sebastian', email: 'sebastian@gmail.com'}; //creamos un arreglo de usuario
11
12 //ejecutamos el metodo POST
13 const res = await request(app)
14   .post('/api/users')
15   .send(newUser);
16
17 expect(res.statusCode).toBe(201);
18
19 const createdUser = res.body; //pasamos el objeto del usuario luego de que se creo
20 expect(createdUser).toHaveProperty('id'); //esperamos que se haya creado correctamente con id
21 expect(createdUser.name).toBe('Sebastian'); // esperamos que contenga el nombre 'Sebastian'
22 expect(createdUser.email).toBe('sebastian@gmail.com'); //esperamos que contenga el email 'sebastian@gmail.com'
23
24 //Ejecutamos el metodo GET
25 const getAllUsers = await request(app)
26   .get('/api/users');
27
28 expect(getAllUsers.statusCode).toBe(200); //esperamos una respuesta correcta
29 expect(getAllUsers.body).not.toEqual([]); //esperamos que la consulta no este vacia
30 expect(getAllUsers.body).toContainEqual(createdUser); //esperamos que exista el objeto nuevo en el ultimo espacio del arreglo
31 });
32 }
  
```

Ilustración 10: Creamos el nuevo usuario y verificamos si consta como ultimo ingreso del arreglo, para esto almacenamos en una variable al usuario creado y comprobamos con el ultimo elemento de la lista con 'toContainEqual'.



```

PS C:\Users\Niño Moi\Downloads\Lab2> npm test

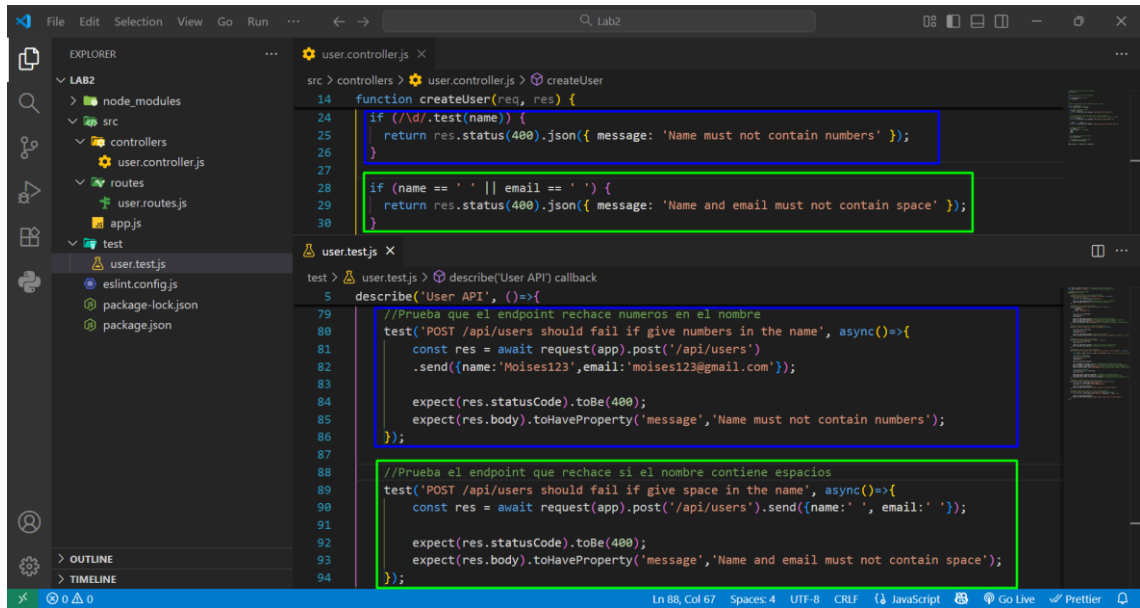
> lab2@1.0.0 test
> jest

PASS test/user.test.js
  User API
    ✓ GET /api/users should return an empty list initially (35 ms)
    ✓ POST /api/users should return create a new user (18 ms)
    ✓ POST /api/users should fail if data is incomplete (5 ms)
    ✓ GET /api/ should return a list in memory (5 ms)
    ✓ POST then GET /api/users return create new user and test the list of registers (10 ms)

Test Suites: 1 passed, 1 total
Tests: 5 passed, 5 total
Snapshots: 0 total
Time: 0.609 s, estimated 1 s
Ran all test suites.
PS C:\Users\Niño Moi\Downloads\Lab2>
  
```

Ilustración 11: Verificamos que funcione correctamente la comprobación de crear y obtener el usuario.

- c. Objetivo: Pruebas de flujo completo (end-to-end simulado).

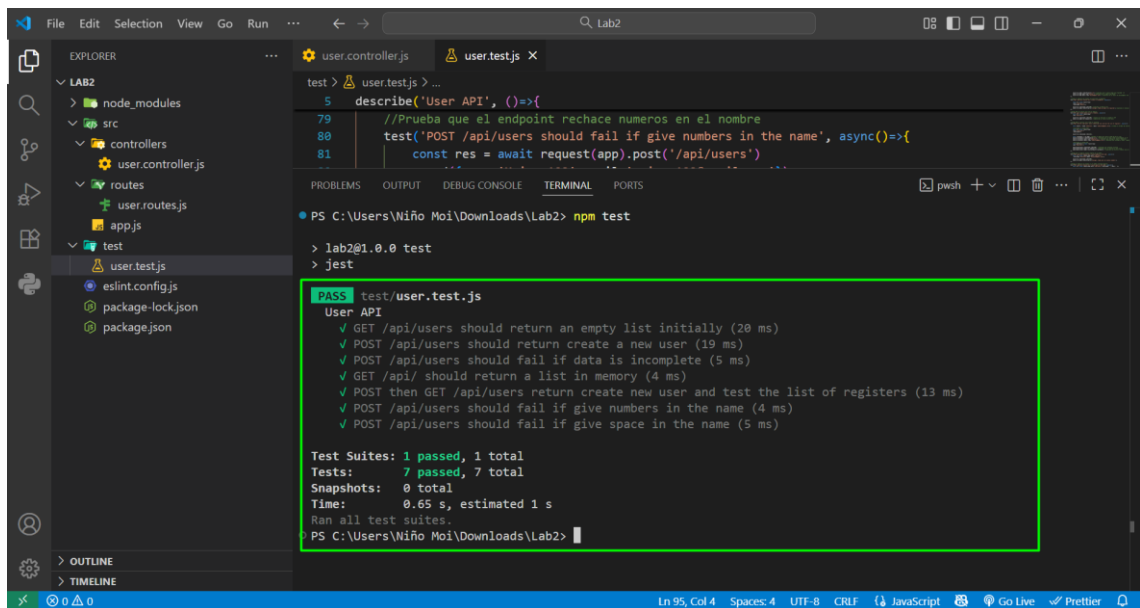


```

src > controllers > user.controller.js > createUser
14 function createUser(req, res) {
24   if (/d/.test(name)) {
25     return res.status(400).json({ message: 'Name must not contain numbers' });
26   }
27
28   if (name == ' ' || email == ' ') {
29     return res.status(400).json({ message: 'Name and email must not contain space' });
30   }
}

test > user.test.js > describe('User API', () => {
5   describe('User API', () => {
79     //Prueba que el endpoint rechace numeros en el nombre
80     test('POST /api/users should fail if give numbers in the name', async()=>{
81       const res = await request(app).post('/api/users')
82         .send({name: 'Moises123', email: 'moises123@gmail.com'});
83
84       expect(res.statusCode).toBe(400);
85       expect(res.body).toHaveProperty('message', 'Name must not contain numbers');
86     });
87
88     //Prueba el endpoint que rechace si el nombre contiene espacios
89     test('POST /api/users should fail if give space in the name', async()=>{
90       const res = await request(app).post('/api/users').send({name: ' ', email: ' '});
91
92       expect(res.statusCode).toBe(400);
93       expect(res.body).toHaveProperty('message', 'Name and email must not contain space');
94     });
95   });
}
  
```

Ilustración 12: Se crearon dos pruebas adicionales que validan si el nombre no contenga números y que el nombre o correo no acepte espacios en blanco.



```

PS C:\Users\Niño Moi\Downloads\Lab2> npm test

> lab2@1.0.0 test
> jest

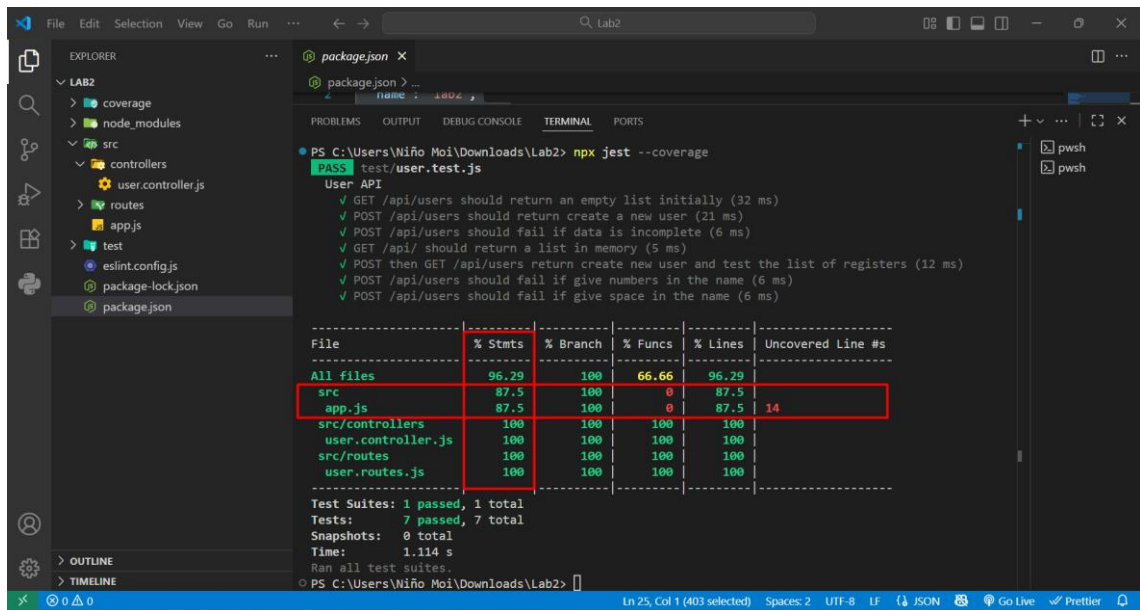
PASS test/user.test.js
  User API
    ✓ GET /api/users should return an empty list initially (20 ms)
    ✓ POST /api/users should return create a new user (19 ms)
    ✓ POST /api/users should fail if data is incomplete (5 ms)
    ✓ GET /api/ should return a list in memory (4 ms)
    ✓ POST then GET /api/users return create new user and test the list of registers (13 ms)
    ✓ POST /api/users should fail if give numbers in the name (4 ms)
    ✓ POST /api/users should fail if give space in the name (5 ms)

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:  0 total
Time:        0.65 s, estimated 1 s
Ran all test suites.
PS C:\Users\Niño Moi\Downloads\Lab2>
  
```

Ilustración 13: Se comprobó que las pruebas sean 100% válidas para continuar con el laboratorio.

2. Verificar cobertura de pruebas con Jest

- Actividad: Ejecutar `jest --coverage`, analizar el reporte y agregar pruebas hasta lograr >90% de cobertura.



package.json

```

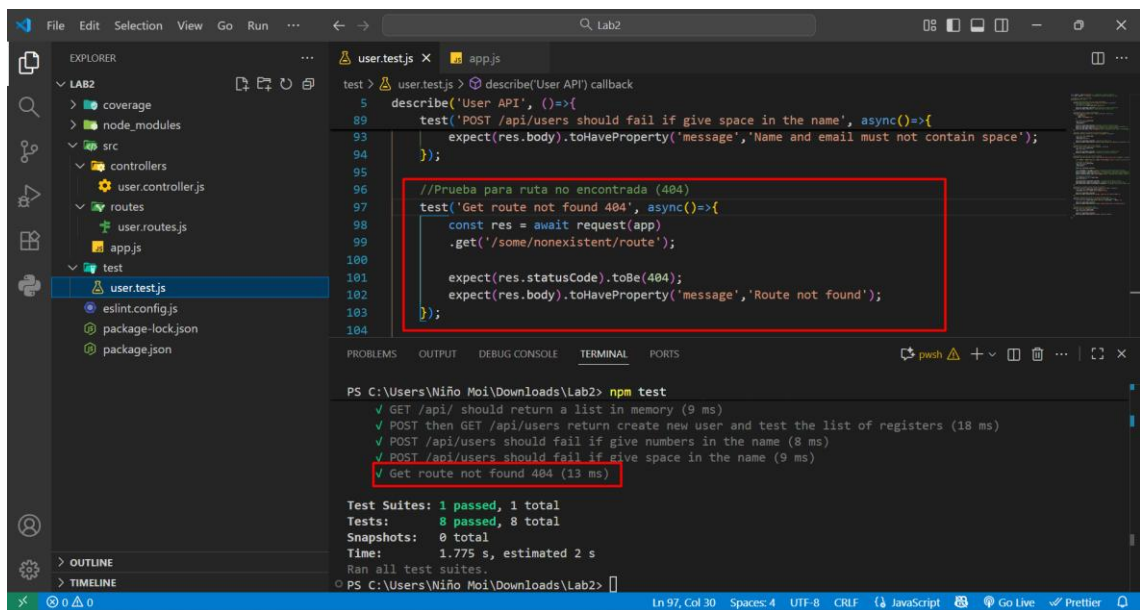
  • PS C:\Users\Niño Moi\Downloads\Lab2> npx jest --coverage
  PASS test/user.test.js
    User API
      ✓ GET /api/users should return an empty list initially (32 ms)
      ✓ POST /api/users should return create a new user (21 ms)
      ✓ POST /api/users should fail if data is incomplete (6 ms)
      ✓ GET /api/ should return a list in memory (5 ms)
      ✓ POST then GET /api/users return create new user and test the list of registers (12 ms)
      ✓ POST /api/users should fail if give numbers in the name (6 ms)
      ✓ POST /api/users should fail if give space in the name (6 ms)

  File                                     % Stmts   % Branch   % Funcs   % Lines   Uncovered Line #s
  -----
  All files                               96.29     100       66.66     96.29
  src                                     87.5      100        0      87.5
  app.js                                  87.5      100        0      87.5      14
  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:       7 passed, 7 total
  Snapshots:   0 total
  Time:        1.114 s
  Ran all test suites.
  
```

Ilustración 14: Ejecución de 'npx jest --coverage' validar el porcentaje de cobertura que se realizó mediante las pruebas, demostrando que no se ha tomado en cuenta una api de 'app.js' en la línea 14.

b. Objetivo: Comprender el impacto de la cobertura y escribir pruebas adicionales.



user.test.js

```

  test('User API', () => {
    describe('POST /api/users should fail if give space in the name', async () => {
      test('POST /api/users should fail if give space in the name', async () => {
        expect(res.body).toHaveProperty('message', 'Name and email must not contain space');
      });
    });

    //Prueba para ruta no encontrada (404)
    test('Get route not found 404', async () => {
      const res = await request(app)
        .get('/some/nonexistent/route');

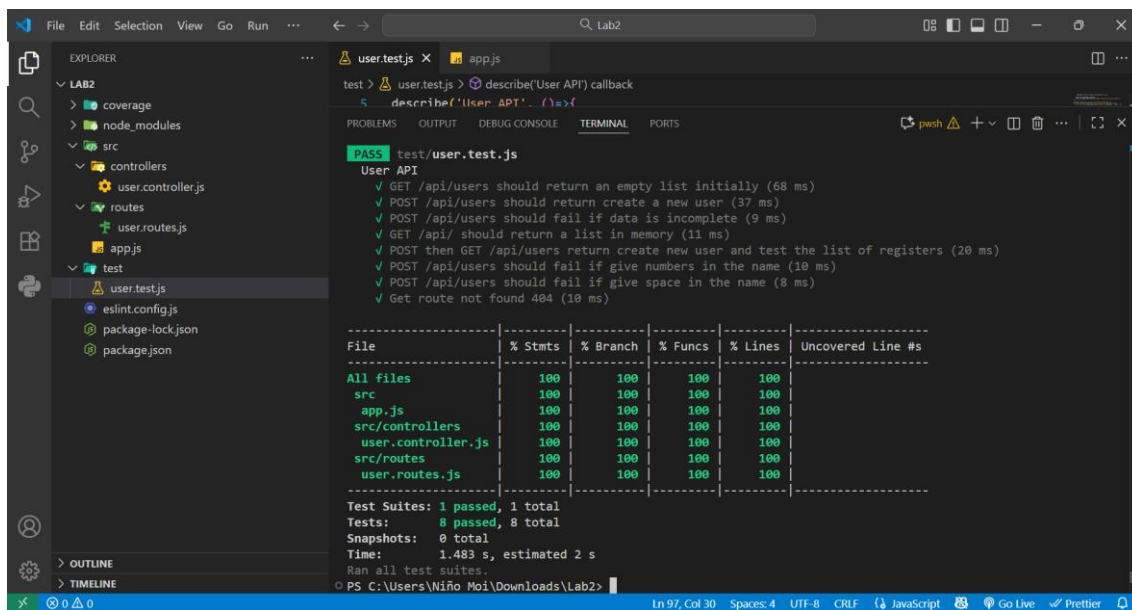
      expect(res.statusCode).toBe(404);
      expect(res.body).toHaveProperty('message', 'Route not found');
    });
  });
  
```

```

  PS C:\Users\Niño Moi\Downloads\Lab2> npm test
  ✓ GET /api/ should return a list in memory (9 ms)
  ✓ POST then GET /api/users return create new user and test the list of registers (18 ms)
  ✓ POST /api/users should fail if give numbers in the name (8 ms)
  ✓ POST /api/users should fail if give space in the name (9 ms)
  ✓ Get route not found 404 (13 ms)

  Test Suites: 1 passed, 1 total
  Tests:       8 passed, 8 total
  Snapshots:   0 total
  Time:        1.775 s, estimated 2 s
  Ran all test suites.
  
```

Ilustración 15: Se agregó una nueva prueba que lleve el error de ruta no encontrada y verificando que es una prueba exitosa, para continuar con la evaluación de la cobertura.



The screenshot shows the VS Code interface with a project named 'Lab2'. The Explorer sidebar on the left shows the file structure: src (controllers, routes, app.js), test (user.test.js), and configuration files (eslint.config.js, package-lock.json, package.json). The main editor displays the 'user.test.js' file with Jest test cases for the 'User API'. The output window at the bottom shows the test results: 'PASS test/user.test.js' and a table indicating 100% coverage for all files and code paths.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
src	100	100	100	100	
app.js	100	100	100	100	
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: 8 passed, 8 total
 Snapshots: 0 total
 Time: 1.483 s, estimated 2 s
 Ran all test suites.

Ilustración 16: Se logra obtener un 100% de cobertura demostrando que se cumple con la cobertura en todo el código.

6. CONCLUSIONES:

Las técnicas aplicadas para la verificación y validación del código fueron muy interesantes debido a que es más rápido revisar el código de esa manera, y sin necesidad de estar repitiendo tantas veces para ver si esta ejecutándose correcta o simplemente algo falla en el código.

Además, las reglas que se coloca para la verificación van de acuerdo con la API que se desea probar puesto que se puede probar la API completa o simplemente una parte de esta.

7. RECOMENDACIONES:

Desde mi criterio, recomendaría que se practique con sistemas ya levantados para probar que tan difícil es aplicar las pruebas sobre código realista en el que se suelen omitir los comentarios de los métodos o realizar métodos anidados.

Al iniciar el proceso de pruebas y luego al tener un avance significativo realizar una prueba de cobertura podría ayudar a identificar la API que nos falta probar, logrando evitar pérdidas de tiempo primero a que nos indica cuantas apis tenemos que realizar las API's y segundo que nos permite conocer si ya terminamos las pruebas cumpliendo con una cobertura >90% del código.

Creo que sería muy efectivo buscar ejemplos de pruebas unitarias aplicadas a Node.js ya que nos podríamos hacer la idea de las posibles pruebas que se pueden realizar y ver las que son recomendables para cierto tipo de API's, lo que permitirá realizar pruebas efectivas y necesarias.

8. BIBLIOGRAFÍA:

Express.js. (s.f.). *Express - Fast, unopinionated, minimalist web framework for Node.js*. Recuperado el 7 de noviembre de 2025, de <https://expressjs.com/>

Jest Team. (2025). *Jest · Delightful JavaScript Testing*. Recuperado el 7 de noviembre de 2025, de <https://jestjs.io/>

Supertest. (s.f.). *HTTP assertions made easy via superagent*. GitHub. Recuperado el 7 de noviembre de 2025, de <https://github.com/ladjs/supertest>

ESLint. (2025). *ESLint - Pluggable JavaScript Linter*. Recuperado el 7 de noviembre de 2025, de <https://eslint.org/>

Node.js Foundation. (2025). *Node.js — Run JavaScript Everywhere*. Recuperado el 7 de noviembre de 2025, de <https://nodejs.org/>

Castillo, L. (2024). *Guía para las Prácticas de Laboratorio - Práctica N° 2: Verificación y Validación*. Código: VDC-GUI-2024-V2-015. Departamento de Ciencias de la Computación, ESPE Sede Santo Domingo.

9. Anexos:

Enlace al repositorio en github:

<https://github.com/Sebas8173/UniversityRepository/tree/main/Pruebas%20de%20software/U1/Lab%202/Lab%202%20-%20practica>