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:**	6to	**FECHA:**	30/12/2025
DOCENTE:	Ing. Luis Castillo, Mgs	**PRÁCTICA N°:**	6	**CALIFICACIÓN:**	

CI/CD usando GitHub Actions

Christian Marcelo Acuña Gamboa

ÍNDICE GENERAL

Contenido

RESUMEN

En el presente laboratorio se llevó a cabo la implementación de un flujo de Integración Continua (CI) utilizando la plataforma GitHub Actions para una aplicación backend desarrollada en Node.js. El proceso abarcó desde la configuración inicial del entorno y la instalación de dependencias clave como Express, hasta la integración de herramientas de aseguramiento de calidad: Jest para la ejecución de pruebas unitarias y ESLint para el análisis estático del código. Se configuró un workflow automatizado capaz de detectar errores de lógica y sintaxis cada vez que se realizan cambios en el repositorio, validando su funcionamiento mediante la implementación de funciones matemáticas y la simulación de errores intencionales para verificar la robustez del sistema.
Palabras Claves: Integración Continua (CI), GitHub Actions, Automatización, Node.js, Jest, ESLint, Testing.

1. INTRODUCCIÓN:

La integración continua (CI) es una práctica fundamental del desarrollo de software moderno. Este laboratorio tiene como propósito familiarizar con la automatización de tareas esenciales como la instalación de dependencias, la ejecución de pruebas unitarias y la verificación de calidad del código mediante ESLint, todo ello gestionado a través de GitHub Actions. A través de una aplicación sencilla en Node.js, se experimentará el poder de los flujos automatizados y se comprenderá la importancia de detectar errores temprano en el ciclo de vida del desarrollo.
.

2. OBJETIVO(S):

1.1 Configurar un flujo de integración continua (CI) en GitHub Actions que se active automáticamente con cada push o pull request a la rama principal del repositorio.
1.2 Implementar pruebas unitarias usando Jest, garantizando que la lógica del sistema funcione correctamente en cada actualización del código.
1.3 Aplicar análisis estático de código con ESLint, reforzando buenas prácticas de programación y detección temprana de errores o inconsistencias.
1.4 Simular un proceso de despliegue automatizado, demostrando cómo se automatizan las etapas previas al paso final de entrega continua (CD), aún sin depender de un proveedor de hosting.

3. MARCO TEÓRICO:

3.1 Fundamentos de DevOps y CI/CD

La ingeniería de software moderna se apoya en la cultura **DevOps**, la cual busca unificar el desarrollo de software (Dev) y la operación del software (Ops). Dentro de este paradigma, la **Integración Continua (CI)** y la **Entrega Continua (CD)** son pilares fundamentales.

La **Integración Continua (CI)** consiste en la práctica de fusionar copias de trabajo de todos los desarrolladores en una línea principal compartida varias veces al día. El objetivo es prevenir problemas de integración masivos ("integration hell") al verificar cada cambio mediante una compilación y ejecución de pruebas automatizadas. Esto permite detectar errores en etapas tempranas (*fail fast*), reduciendo drásticamente el costo y tiempo de corrección.

3.2 Plataforma de Automatización: GitHub Actions

GitHub Actions es un motor de CI/CD integrado directamente en el repositorio de código, lo que elimina la necesidad de servidores de compilación externos complejos. Su arquitectura se basa en los siguientes componentes clave:

- **Workflows (Flujos de trabajo):** Procesos automatizados configurables definidos en archivos con formato **YAML** (.yml). Un repositorio puede tener múltiples flujos para tareas distintas (ej. pruebas, despliegue, etiquetado).
- **Events (Eventos):** Desencadenantes específicos que inician un flujo de trabajo. Pueden ser eventos internos de GitHub (como un push, pull_request o la creación de un release), eventos programados (cron jobs) o eventos externos.
- **Runners (Ejecutores):** Son los servidores que ejecutan los trabajos definidos en el workflow. GitHub ofrece *hosted runners* (máquinas virtuales con Linux, Windows o macOS mantenidas por GitHub) y *self-hosted runners* (servidores propios del usuario).
- **Jobs (Trabajos) y Steps (Pasos):** Un *job* es un conjunto de pasos (*steps*) que se ejecutan secuencialmente en el mismo ejecutor. Los trabajos pueden configurarse para ejecutarse en paralelo o depender unos de otros.

3.3 Entorno de Ejecución: Node.js y Express

Node.js es un entorno de tiempo de ejecución para JavaScript construido sobre el motor V8 de Chrome. Su modelo de E/S sin bloqueo y orientado a eventos lo hace ideal para aplicaciones escalables y ligeras. En este laboratorio, se utiliza **Express**, un marco de aplicación web minimalista y flexible para Node.js, que facilita la gestión de rutas y peticiones HTTP, sirviendo como la base de la aplicación sobre la cual se aplicarán las pruebas.

3.4 Aseguramiento de Calidad Estático: ESLint

El análisis estático de código es una técnica de depuración que se realiza sin ejecutar el programa. **ESLint** es la herramienta estándar de facto para este propósito en el ecosistema JavaScript. Funciona analizando el código fuente para transformarlo en un Árbol de Sintaxis Abstracta (AST) y evaluando patrones que no cumplen con reglas predefinidas.

Su uso en un entorno de CI es crítico para:

1. **Higiene del código:** Asegurar que todo el equipo siga las mismas guías de estilo (indentación, uso de comillas, etc.).
2. **Prevención de errores:** Detectar variables no utilizadas, bucles infinitos potenciales o código inalcanzable antes de la fase de pruebas dinámicas.

3.5 Pruebas Unitarias Automatizadas: Jest

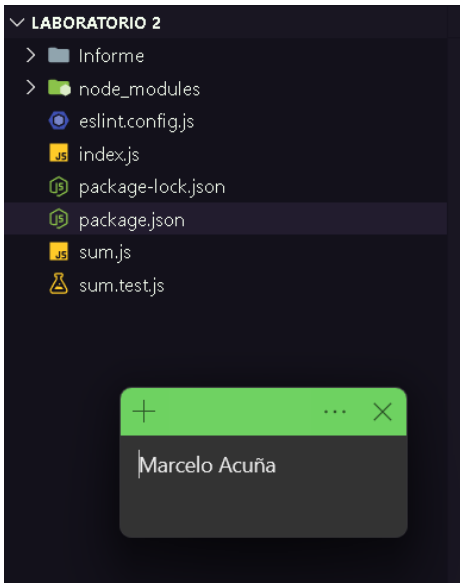
A diferencia del análisis estático, las pruebas dinámicas requieren la ejecución del código. **Jest** es un marco de pruebas completo que actúa como:

- **Test Runner:** Busca y ejecuta archivos de prueba (usualmente terminados en .test.js o .spec.js).
- **Librería de Aserciones:** Proporciona funciones como expect() y toBe() para validar que el resultado obtenido coincida con el esperado.
- **Herramienta de Mocking:** Permite simular dependencias externas para aislar la unidad de código que se está probando.

4. DESCRIPCIÓN DEL PROCEDIMIENTO:

PARTE 1: Establecimiento de la estructura del proyecto base .

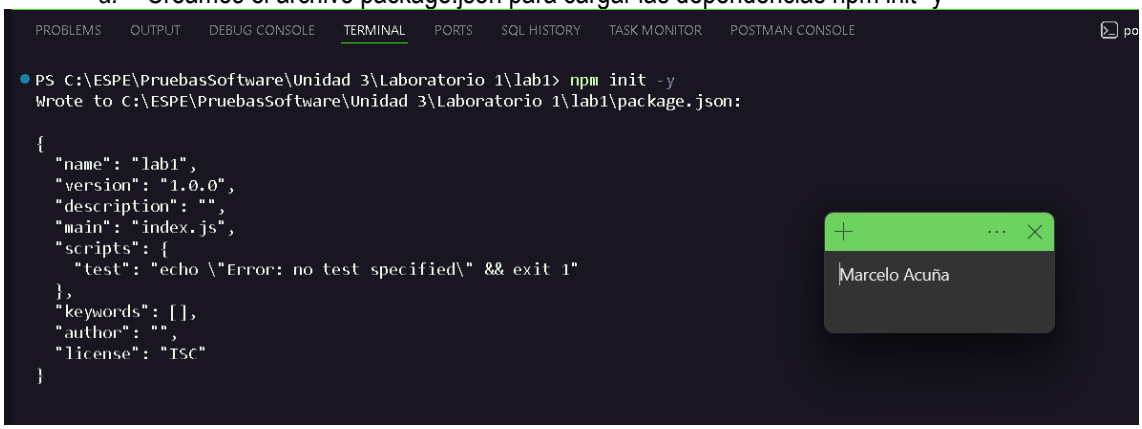
Paso 1: Creación de la estructura básica.



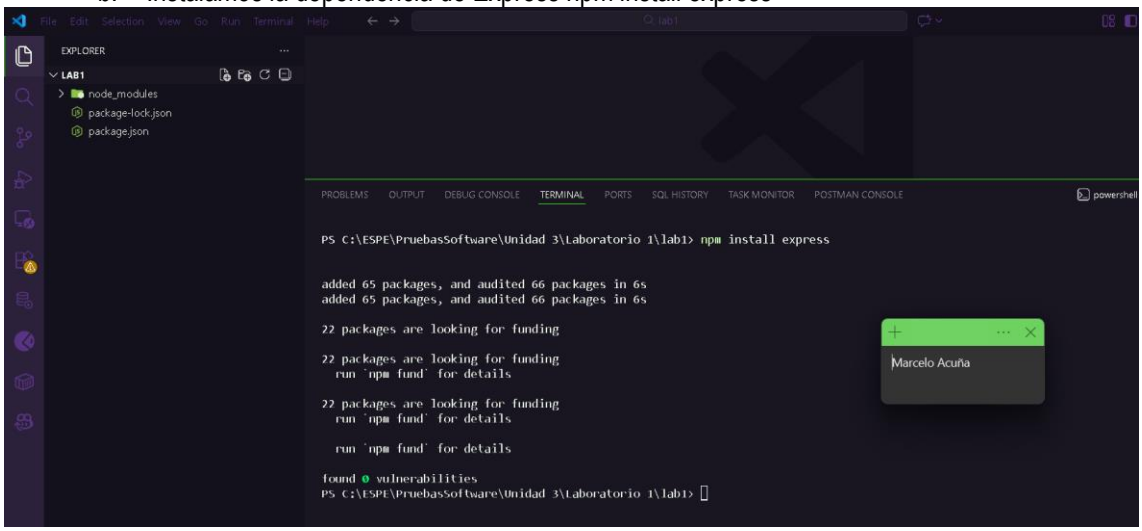
`

Paso 2: Instalación de dependencias necesarias

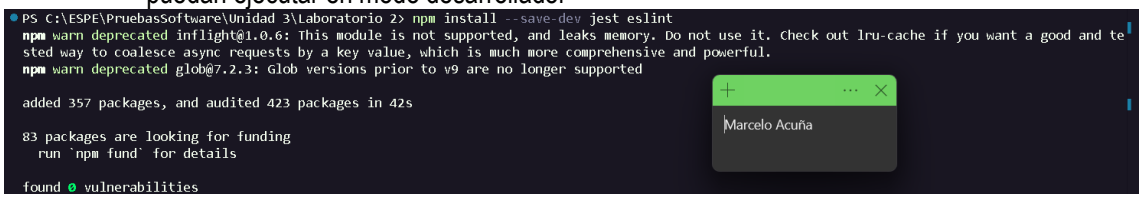
a. Creamos el archivo package.json para cargar las dependencias npm init -y



b. Instalamos la dependencia de Express npm install express



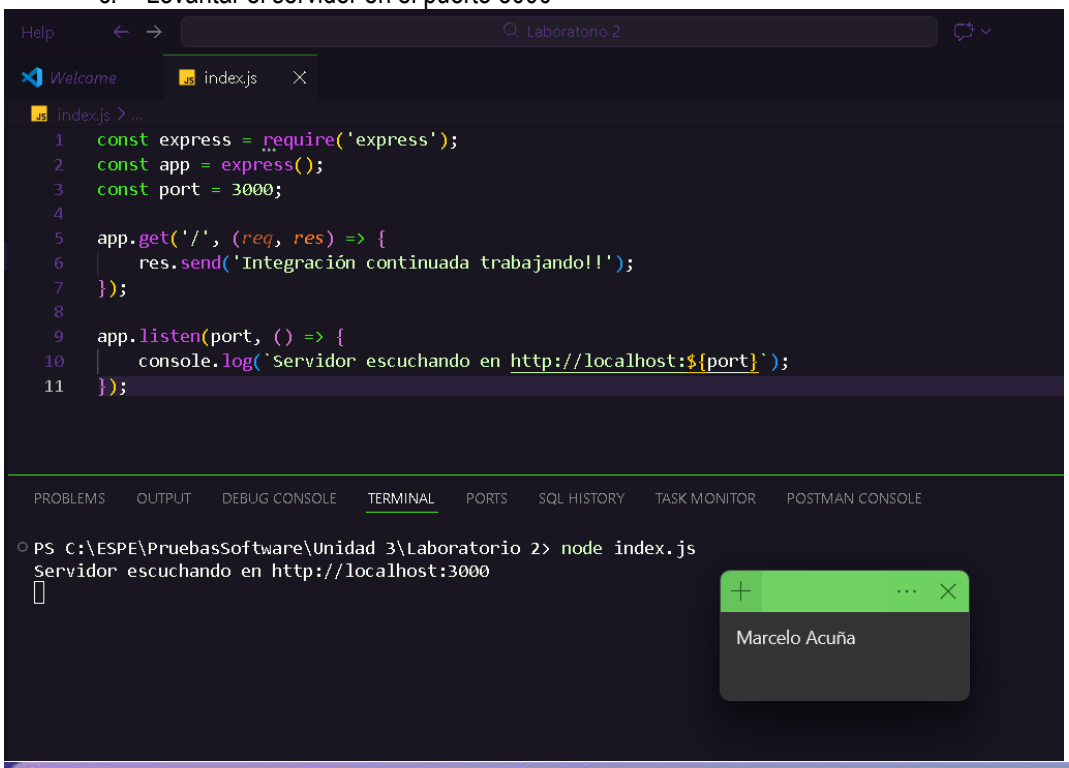
c. Instalamos las dependencias de Jest y ESLint npm install --save-dev jest eslint para que se puedan ejecutar en modo desarrollador

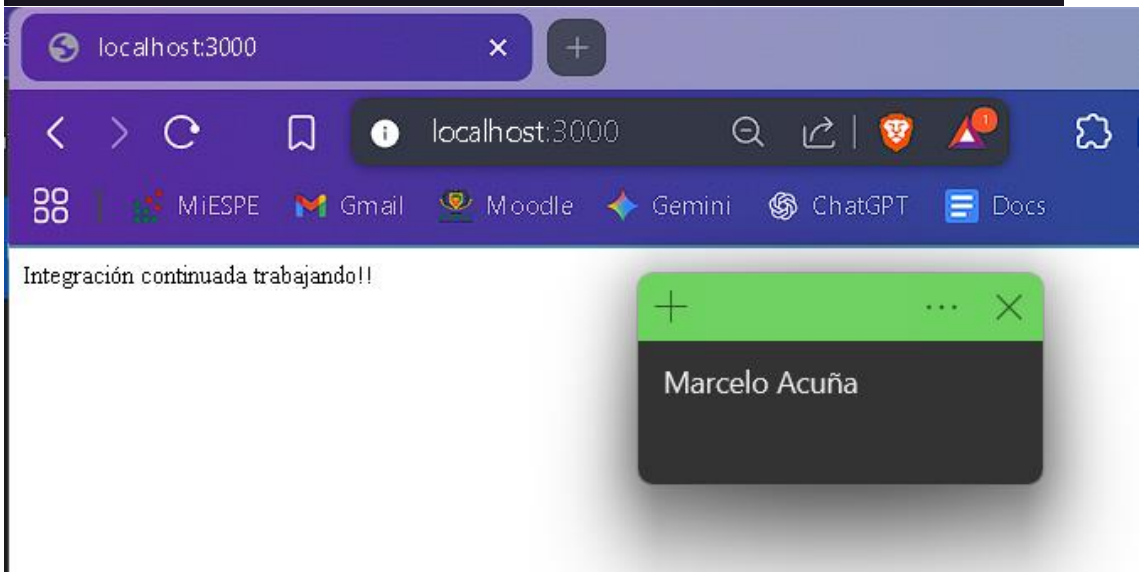


PARTE 2: Creación de archivos base

Paso 1: Crear archivo index.js.

a. Usar el servidor express
b. Implementar un endpoint sencillo que responda con un mensaje
c. Levantar el servidor en el puerto 3000



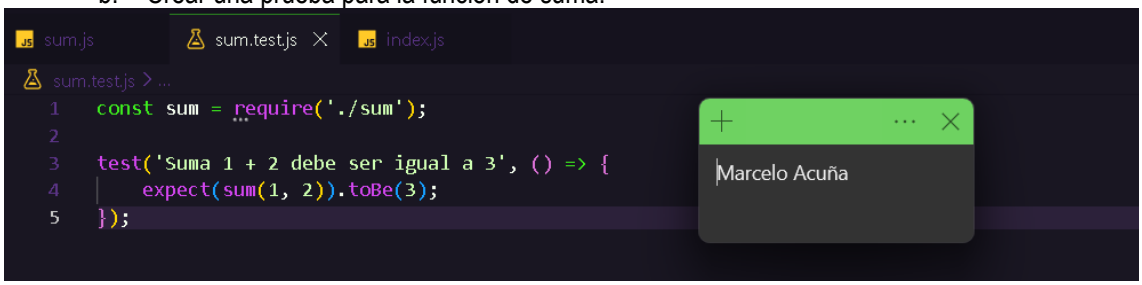


Paso 2: Crear archivo sum.js.

a. Crear una función que sume dos números pasados como parámetros
b. Exportar la función.

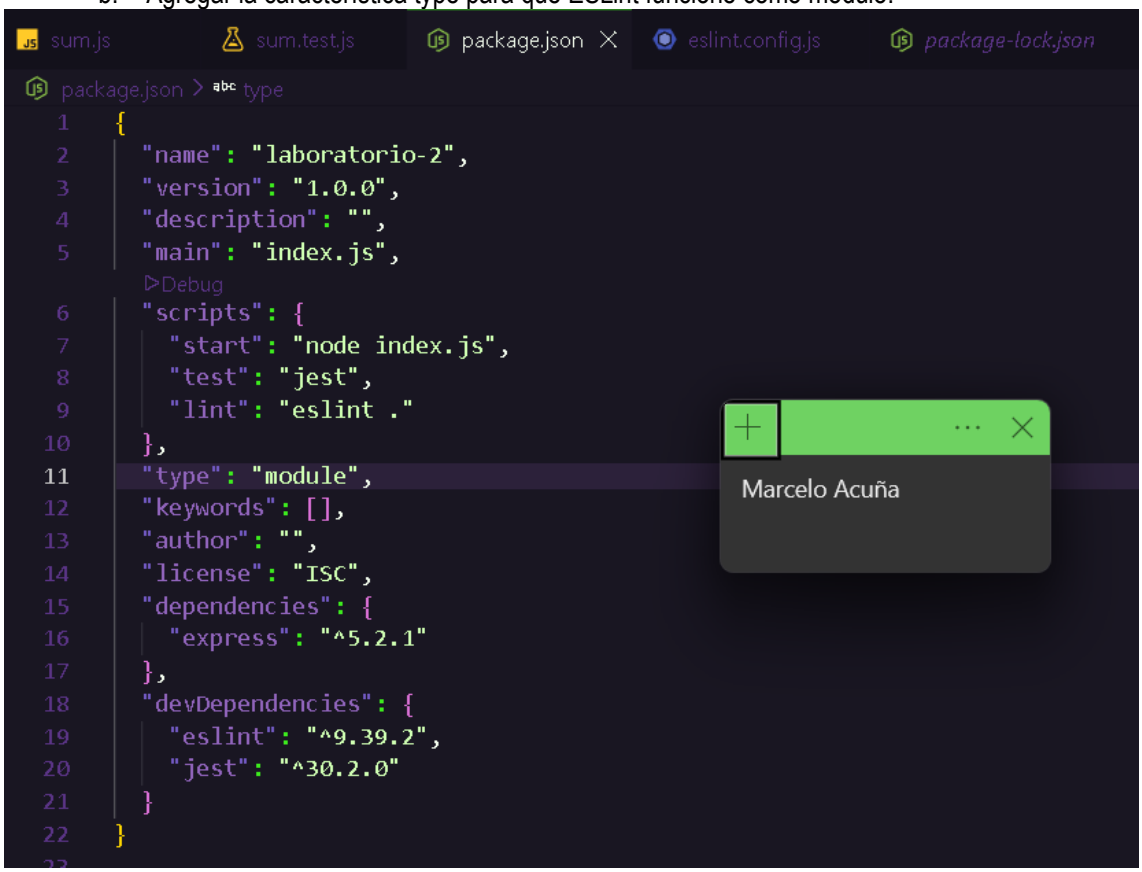
Paso 3: Crear archivo sum.test.js

a. Usar el archivo con la función de suma
b. Crear una prueba para la función de suma.



Paso 4: Configurar package.json.

a. Agregar o editar los scripts para start, test y lint
b. Agregar la característica type para que ESLint funcione como módulo.

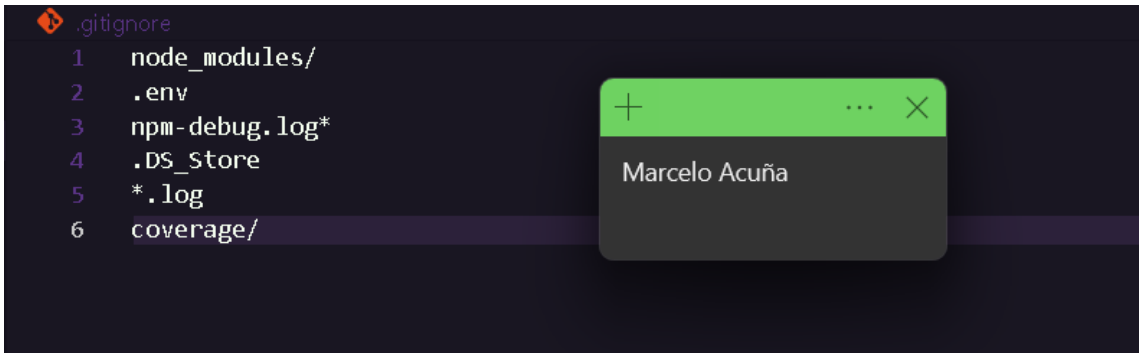


Paso 5: Crear el archivo ESLint.

a. Trabajar con reglas sencillas.

Paso 6: Cambio en la configuración del test

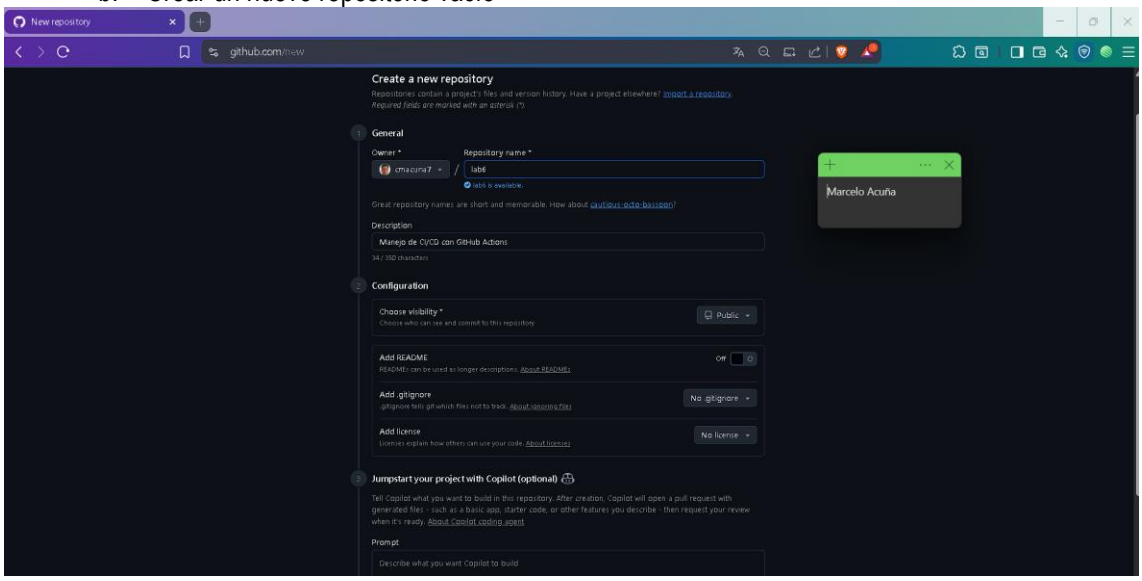
a. En el archivo .gitignore ignorar todos los archivos que puedan causar conflictos para un proyecto NodeJS.



PARTE 3: Configuración de Git

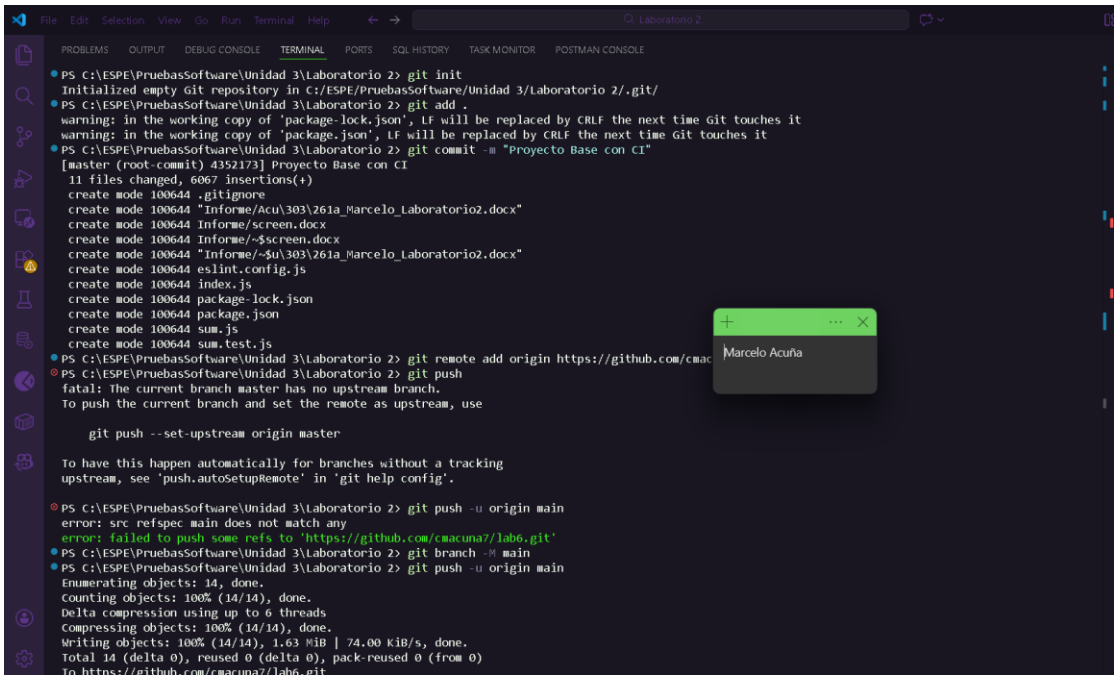
Paso 1: Crear repositorio en la cuenta de Git.

a. Abrir la cuenta de Git en el navegador
b. Crear un nuevo repositorio vacío



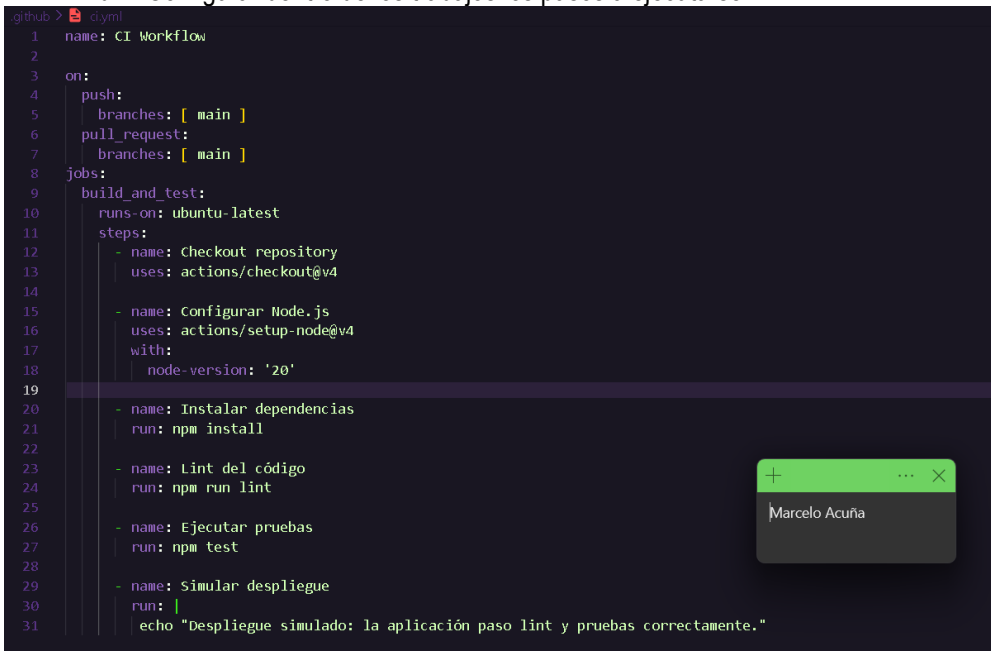
Paso 2: Ejecución de comandos para clonar al repositorio.

a. git init
b. git add .
c. git commit -m "Proyecto base con CI"
d. git branch -M main
e. git remote add origin https://github.com/TU_USUARIO/nombreRepositorio.git
f. git push -u origin main



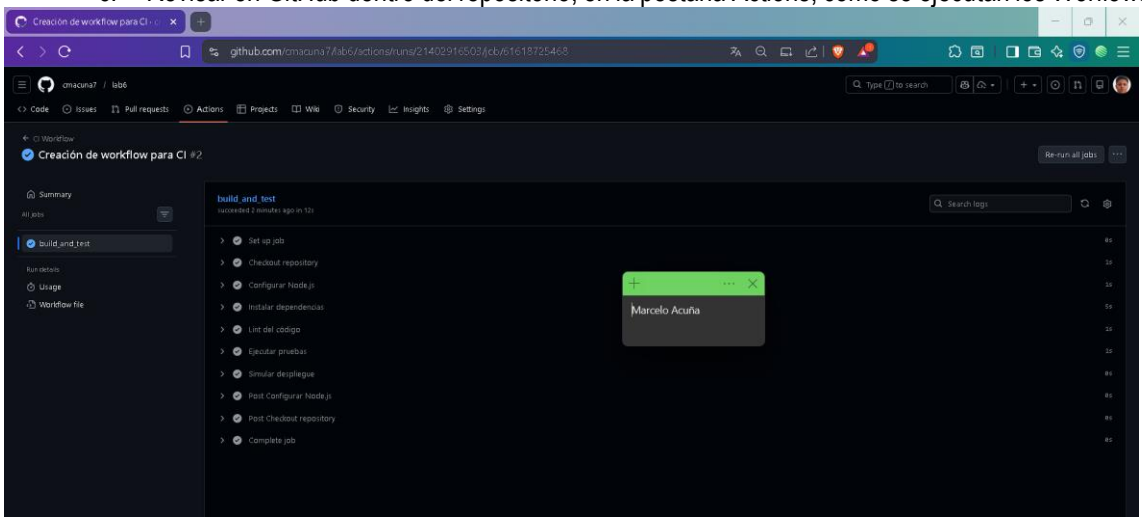
Paso 3: Crear el workflow de GitHub Actions

a. Crear un archivo nuevo para el workflow .github/workflows/ci.yml.
b. Configurar los triggers.
c. Configurar los trabajos a realizar
d. Configurar dentro de los trabajos los pasos a ejecutarse.



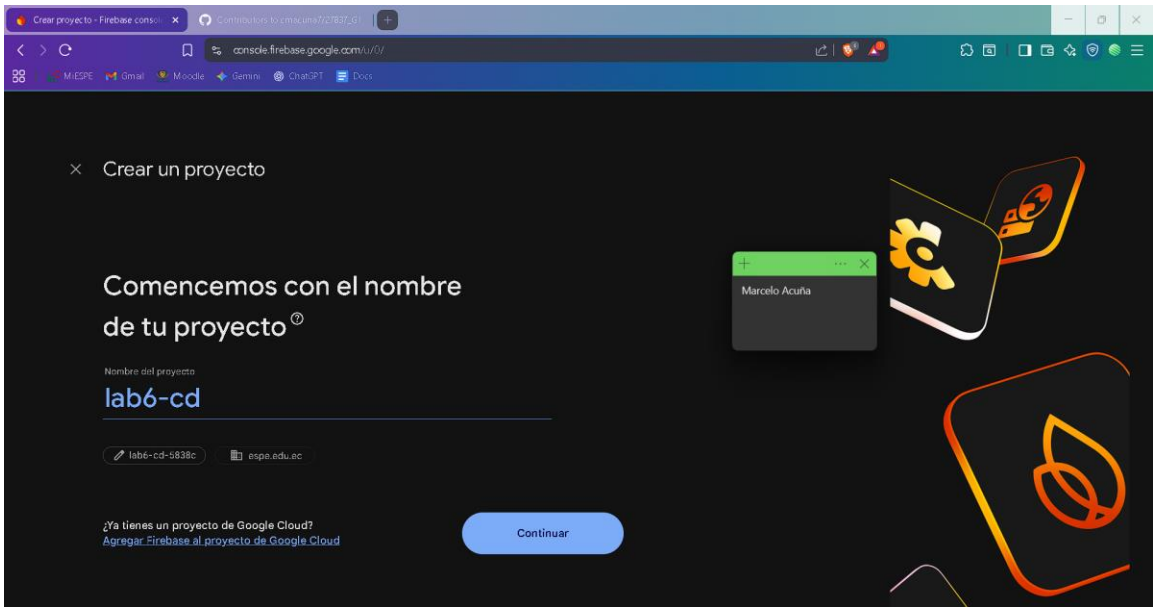
Paso 4: Probar la CI

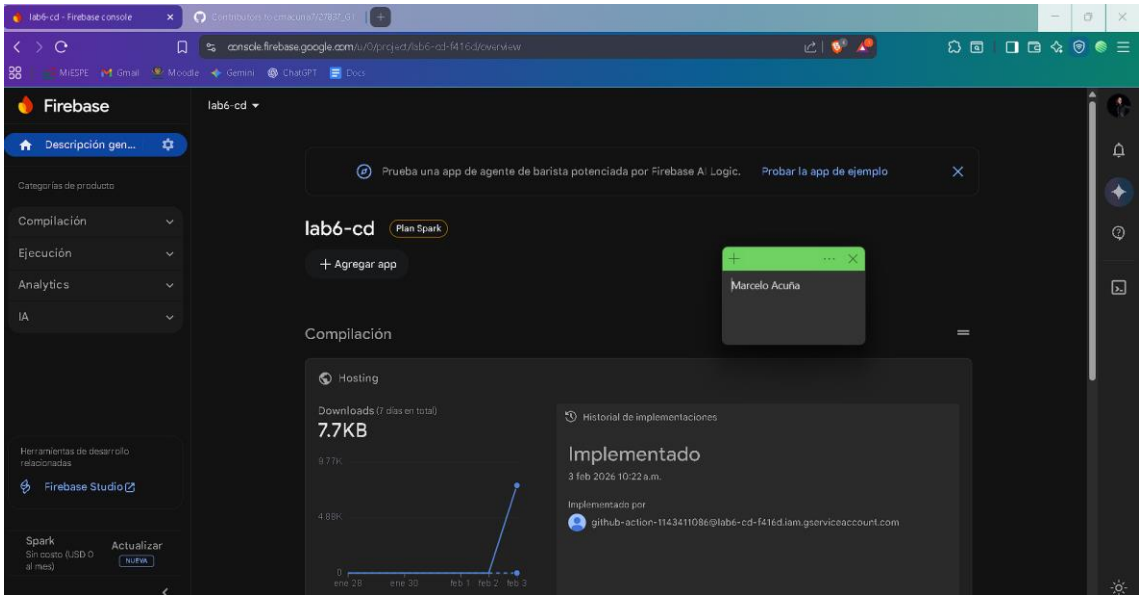
a. Realizar un cambio al código.
b. Ejecutar de nuevo los comandos para realizar un nuevo push.
c. Revisar en GitHub dentro del repositorio, en la pestaña Actions, como se ejecutan los Worflows.



PARTE 4: FireBase

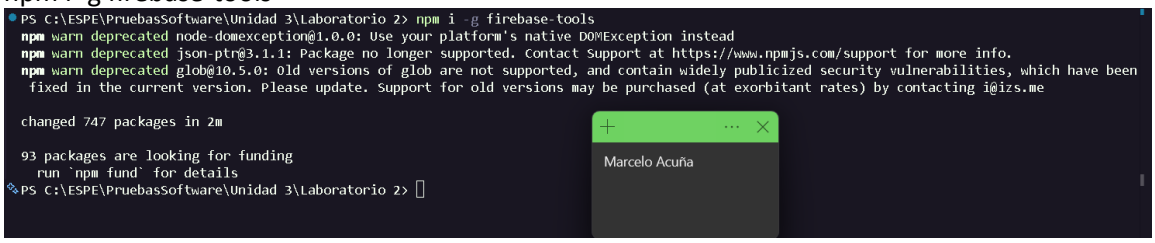
Paso 1: Crear repositorio en Firebase



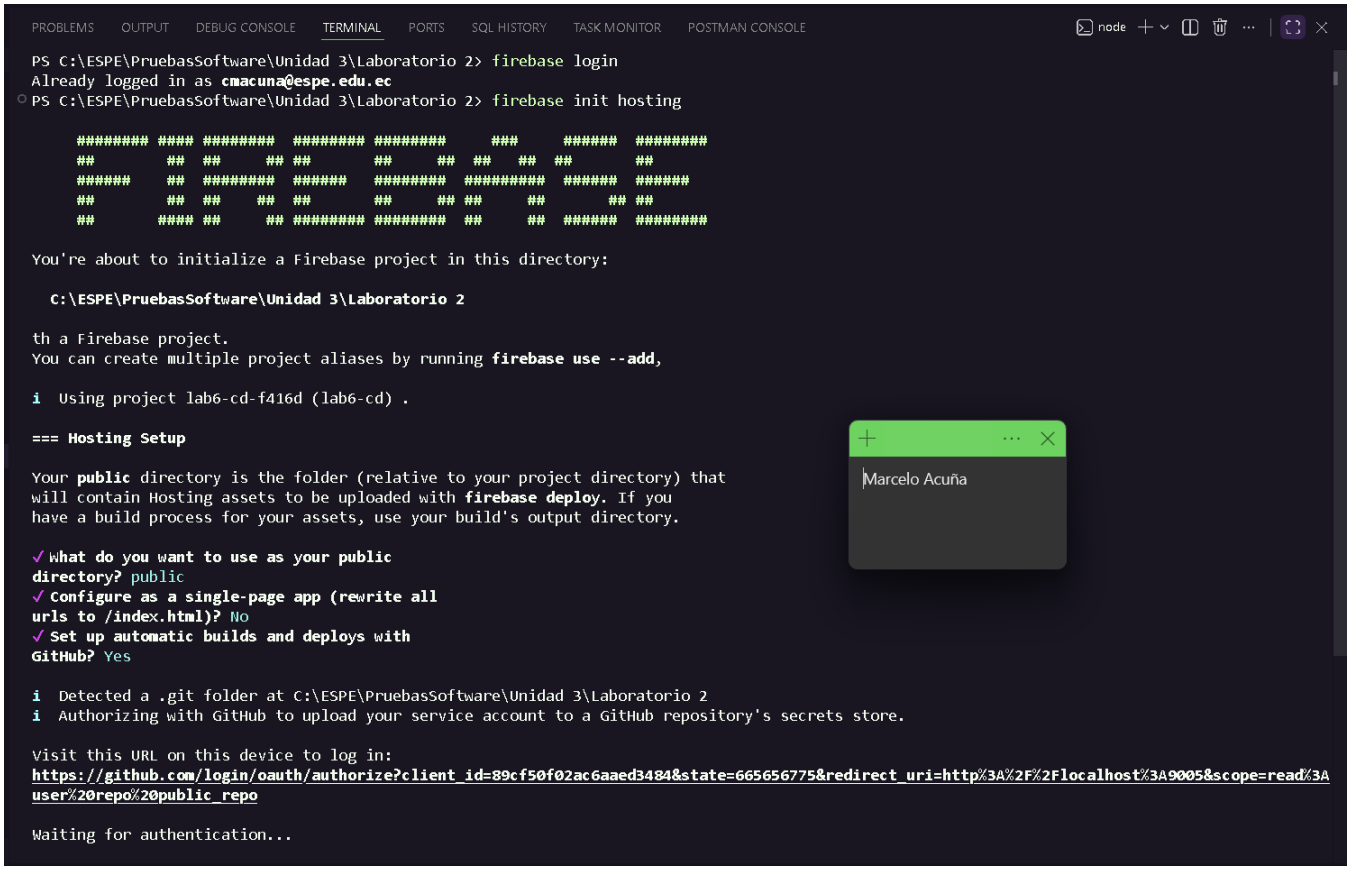


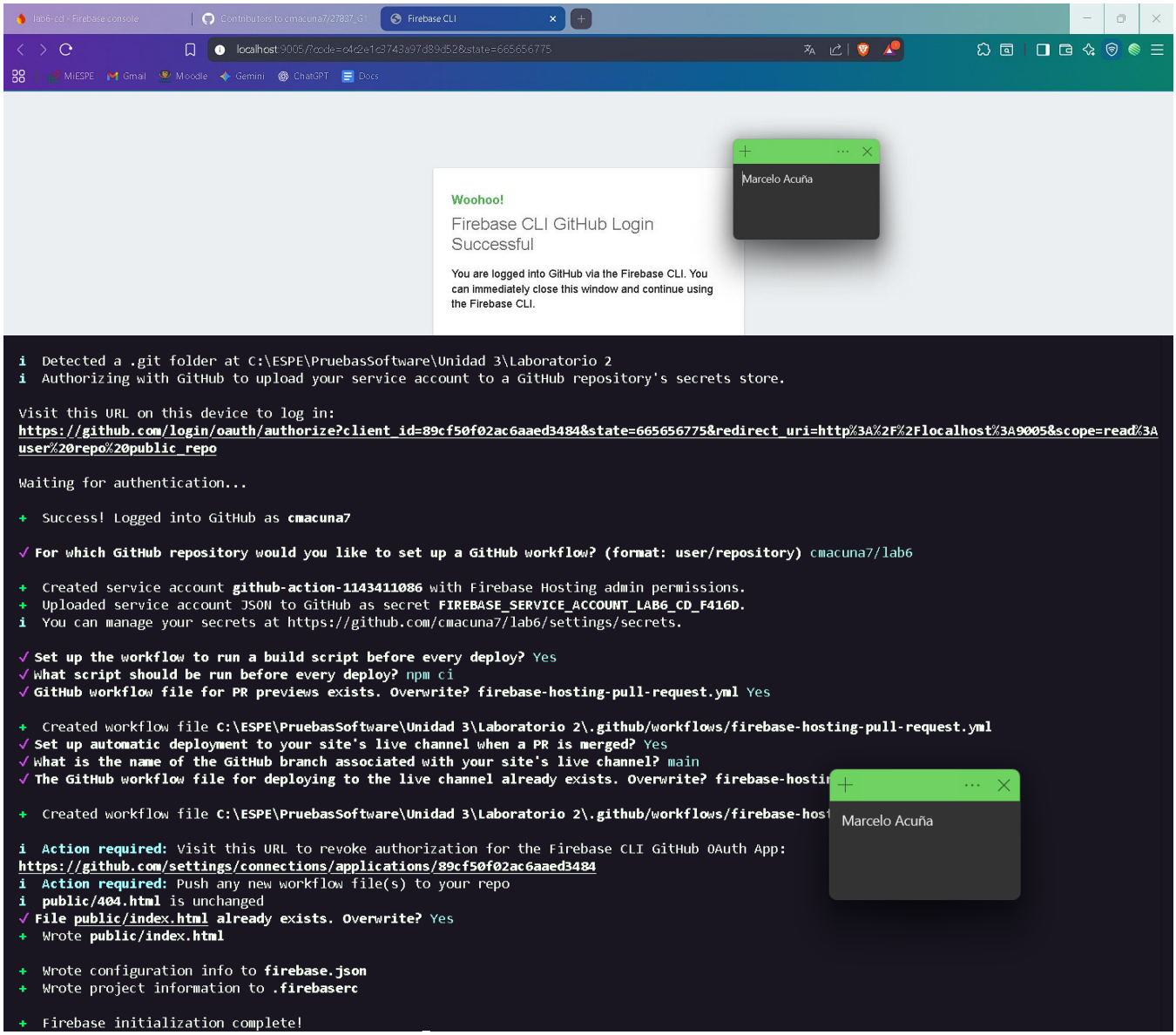
Paso 2: Instalar dependencias

npm i -g firebase-tools



Paso 3: Iniciar sesión

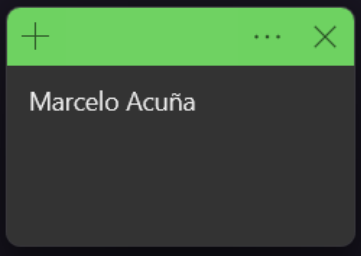




Paso 4: Firebase.json

firebase.json > { } hosting > [ ] rewrites

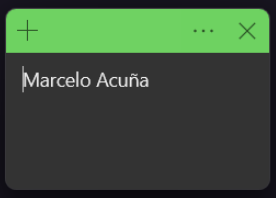
```
{
  "hosting": {
    "public": "public",
    "ignore": [
      "firebase.json",
      "**/.*",
      "**/node_modules/**"
    ],
    "rewrites":
    [
      {
        "source": "**",
        "destination": "/index.html"
      }
    ]
  }
}
```



Paso 5: ci.yaml

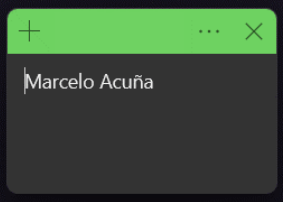
.github > workflows > ci.yml

```
name: CI Workflow

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
jobs:
  build_and_test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Configurar Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'

      - name: Instalar dependencias
        run: npm install

      - name: Lint del código
        run: npm run lint

      - name: Ejecutar pruebas
        run: npm test

      - name: Build
        run: npm ci && npm run build
```

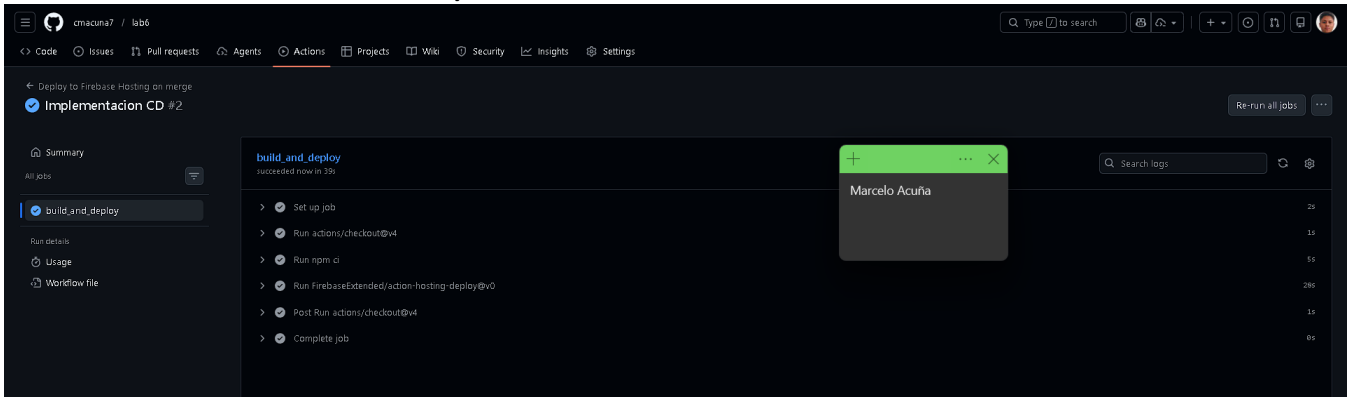


Paso 6: Subir al GIT

```
PS C:\ESPE\PruebasSoftware\Unidad 3\Laboratorio 2> git add .
warning: in the working copy of '.firebaserc', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of '.github/workflows/firebase-hosting-merge.yml', LF will be replaced by CRLF the nex
t time Git touches it
warning: in the working copy of '.github/workflows/firebase-hosting-pull-request.yml', LF will be replaced by CRLF
the next time Git touches it
warning: in the working copy of 'firebase.json', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of 'public/index.html', LF will be replaced by CRLF the next time Git touches it
PS C:\ESPE\PruebasSoftware\Unidad 3\Laboratorio 2> git commit -m "Implementacion CD"
[main 90136de] Implementacion CD
 5 files changed, 98 insertions(+), 19 deletions(-)
 create mode 100644 "Informe/~$u\303\261a_Marcelo_Laboratorio2.docx"
PS C:\ESPE\PruebasSoftware\Unidad 3\Laboratorio 2> git push
Enumerating objects: 20, done.
Counting objects: 100% (20/20), done.
Delta compression using up to 6 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (11/11), 2.42 KiB | 412.00 KiB/s, done.
Total 11 (delta 7), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (7/7), completed with 7 local objects.
To https://github.com/cmacuna7/lab6.git
   e5eba10..90136de  main -> main
PS C:\ESPE\PruebasSoftware\Unidad 3\Laboratorio 2>
```

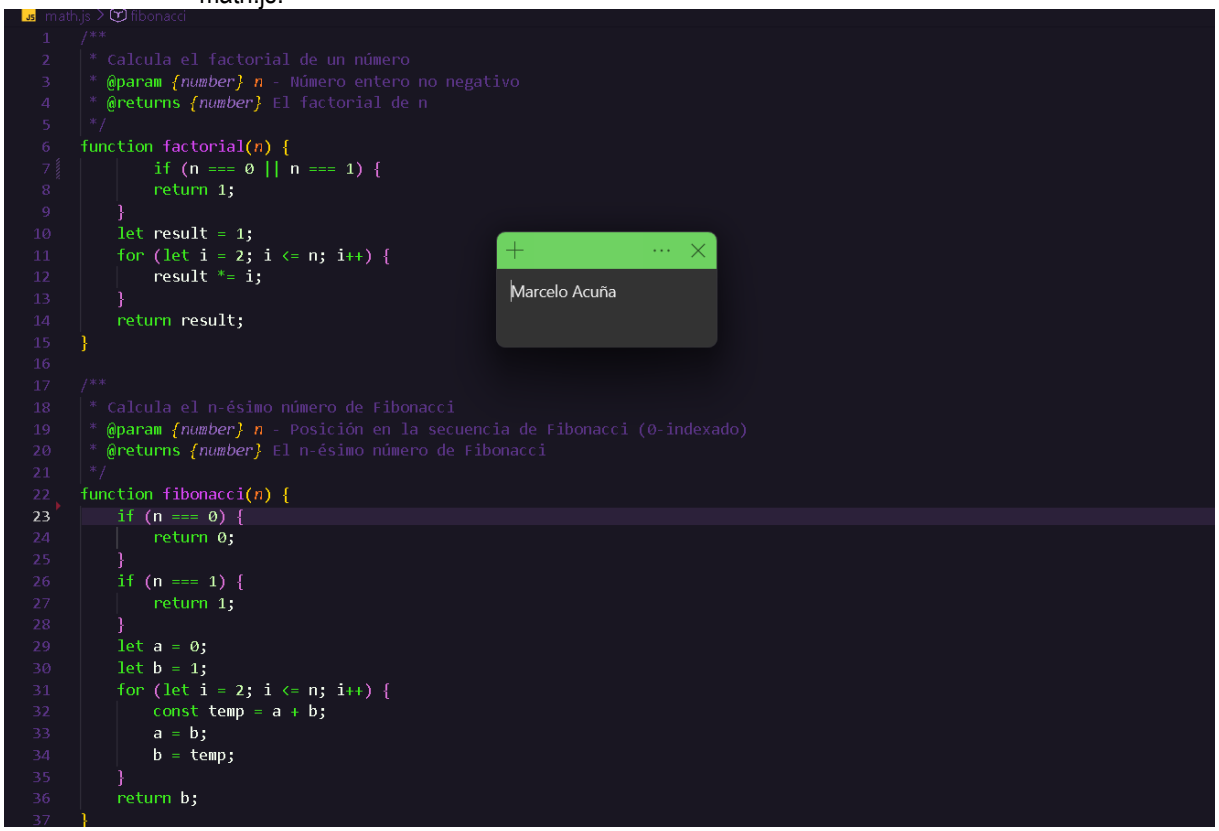


Paso 7: Revisar en Git/Actions



PREGUNTAS/ACTIVIDADES:

- Agregar más pruebas unitarias
  - Agregar al menos 2 funciones nuevas (por ejemplo, factorial, fibonacci) en un archivo math.js.



```
math.js > fibonacci
/**
 * Calcula el factorial de un número
 * @param {number} n - Número entero no negativo
 * @returns {number} El factorial de n
 */
function factorial(n) {
        if (n === 0 || n === 1) {
        return 1;
    }
    let result = 1;
    for (let i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}

/**
 * Calcula el n-ésimo número de Fibonacci
 * @param {number} n - Posición en la secuencia de Fibonacci (0-indexado)
 * @returns {number} El n-ésimo número de Fibonacci
 */
function fibonacci(n) {
    if (n === 0) {
        return 0;
    }
    if (n === 1) {
        return 1;
    }
    let a = 0;
    let b = 1;
    for (let i = 2; i <= n; i++) {
        const temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}
```

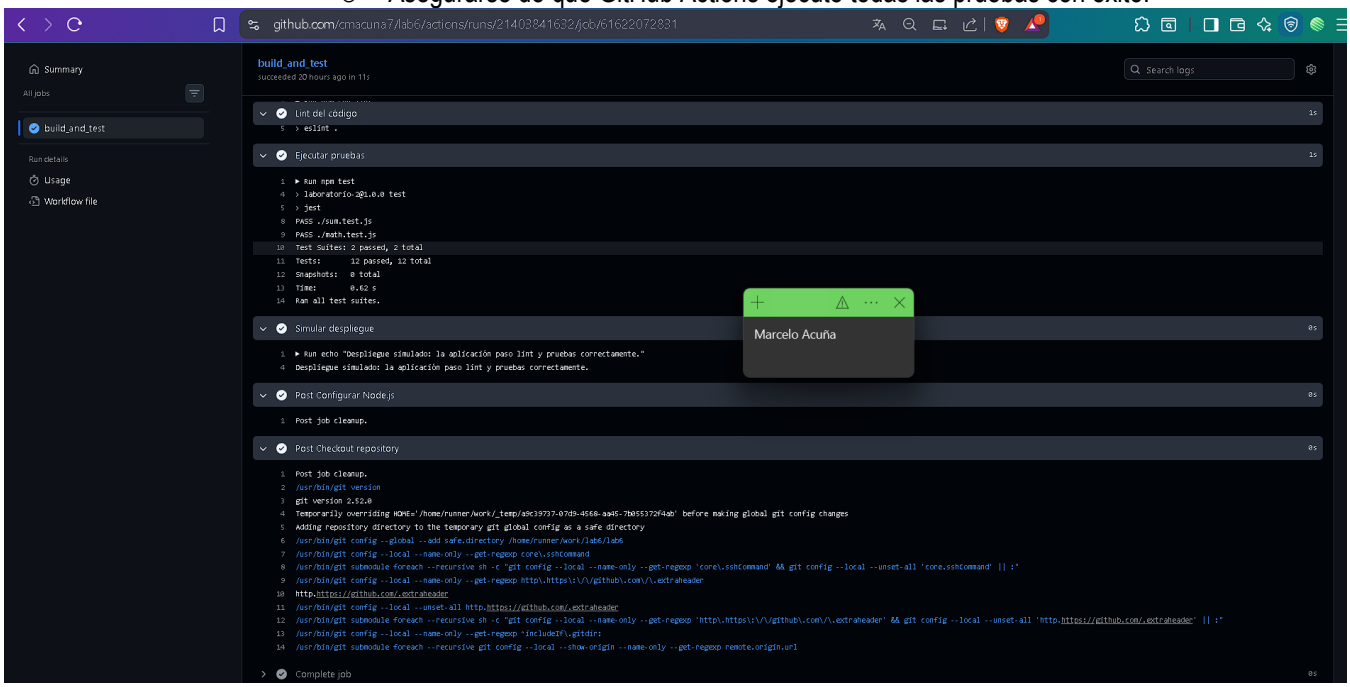
Marcelo Acuña

  - Crear su correspondiente archivo math.test.js con pruebas Jest.

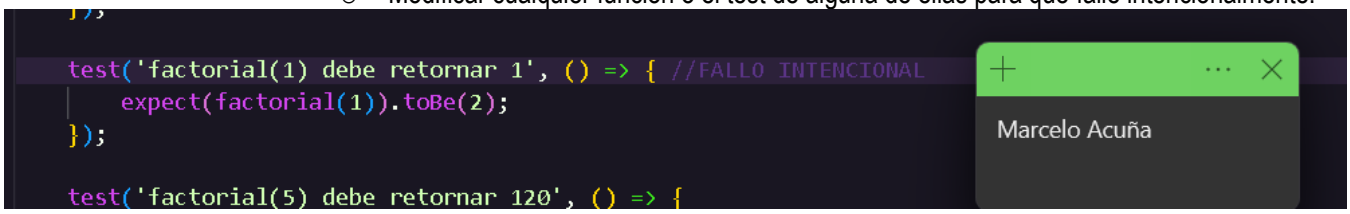
math.test.js > ...

```
const { factorial, fibonacci } = require('./math.js');

describe('Función factorial', () => {
    test('factorial(0) debe retornar 1', () => {
        expect(factorial(0)).toBe(1);
    });

    test('factorial(1) debe retornar 1', () => {
        expect(factorial(1)).toBe(1);
    });

    test('factorial(5) debe retornar 120', () => {
        expect(factorial(5)).toBe(120);
    });

    test('factorial(6) debe retornar 720', () => {
        expect(factorial(6)).toBe(720);
    });

    test('factorial(10) debe retornar 3628800', () => {
        expect(factorial(10)).toBe(3628800);
    });

});

describe('Función fibonacci', () => {
    test('fibonacci(0) debe retornar 0', () => {
        expect(fibonacci(0)).toBe(0);
    });

    test('fibonacci(1) debe retornar 1', () => {
        expect(fibonacci(1)).toBe(1);
    });

    test('fibonacci(2) debe retornar 1', () => {
        expect(fibonacci(2)).toBe(1);
    });
```

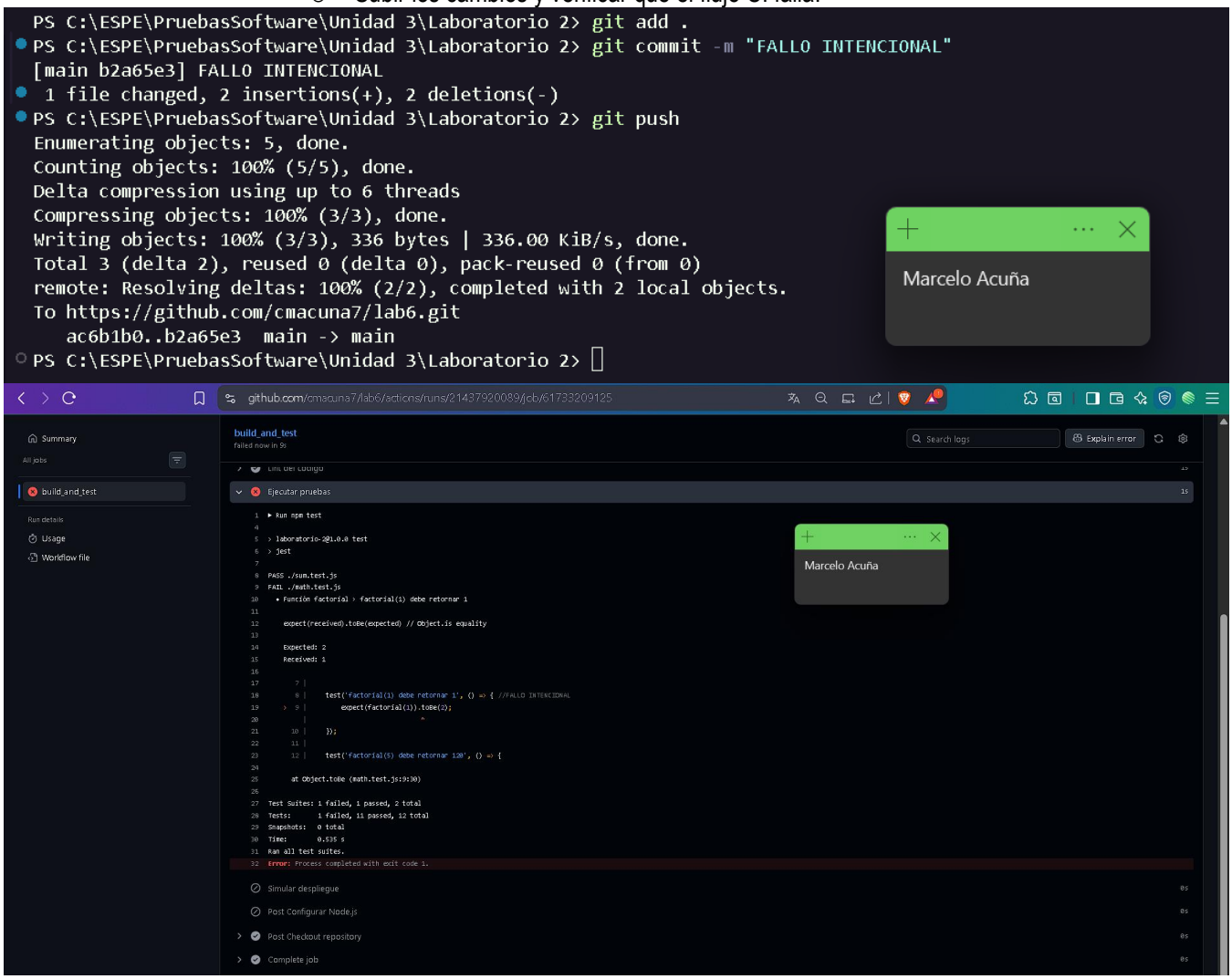

- Asegurarse de que GitHub Actions ejecute todas las pruebas con éxito.



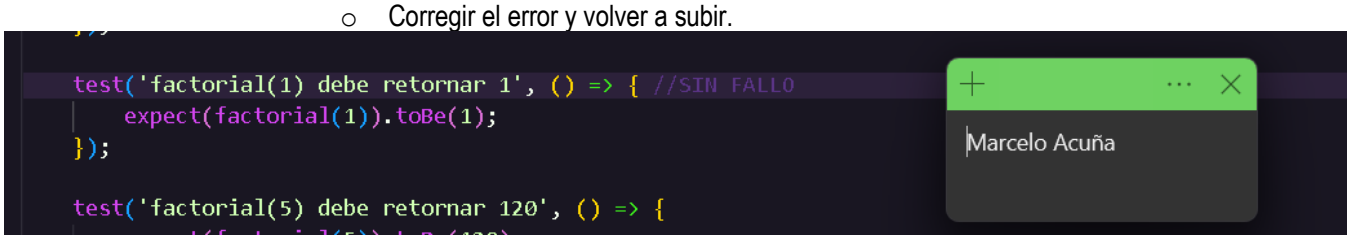
- Provocar un error intencional y corregirlo
  - Modificar cualquier función o el test de alguna de ellas para que falle intencionalmente.

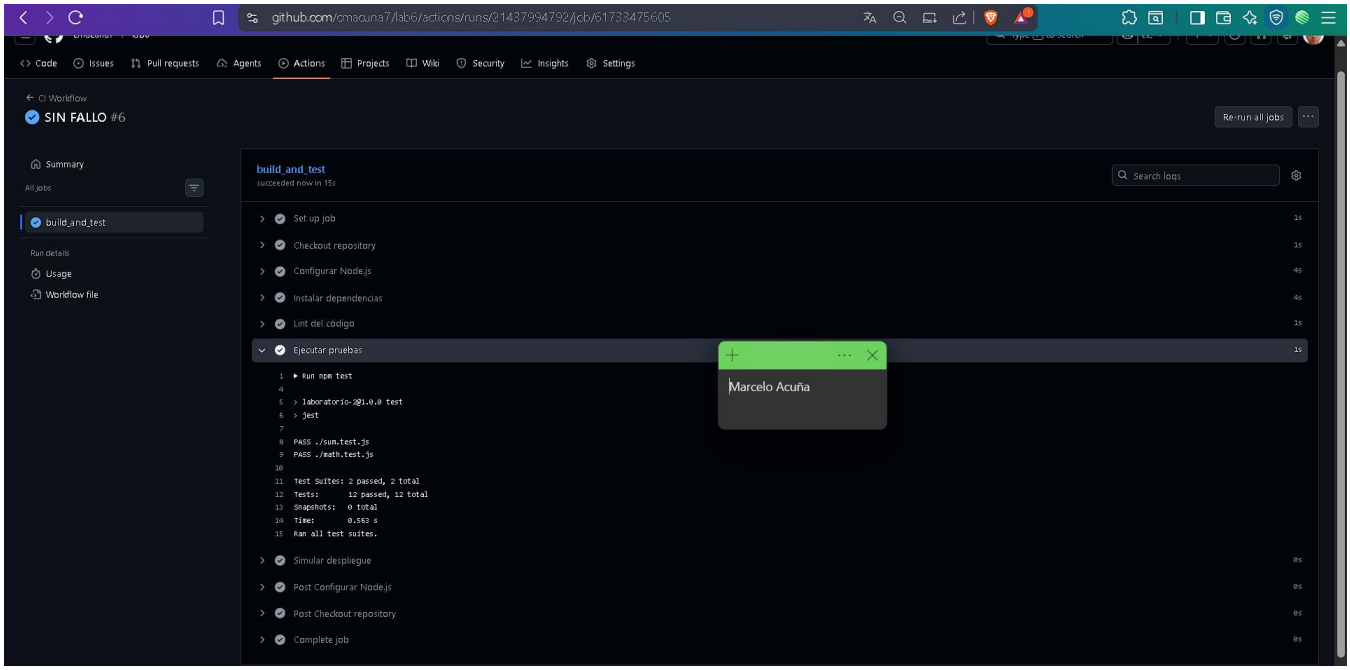


  - Subir los cambios y verificar que el flujo CI falla.



  - Corregir el error y volver a subir.





5. CONCLUSIONES:

- Se logró configurar exitosamente un entorno de Integración Continua (CI) utilizando GitHub Actions, automatizando tareas críticas como la instalación de dependencias, el análisis de código (Linting) y la ejecución de pruebas cada vez que se realiza un push a la rama principal.
- Aunque no se desplegó la aplicación en un servidor de producción real, el paso de "Simular despliegue" en el archivo YAML demostró cómo la Integración Continua es el paso previo necesario para la Entrega Continua (CD), garantizando que solo el software que ha pasado todos los controles de calidad llegue a la fase de despliegue.

6. RECOMENDACIONES:

- Se recomienda ejecutar los comandos npm run lint y npm test localmente antes de realizar un git push, esto ahorra tiempo de ejecución en la nube y permite al desarrollador corregir errores básicos inmediatamente sin esperar al feedback de GitHub Actions.
- Para entornos profesionales, se sugiere configurar reglas de protección en la rama main en GitHub, impidiendo que se pueda hacer merge de código si el workflow de CI ha fallado, garantizando así que la rama principal siempre sea estable.

7. BIBLIOGRAFÍA:

ExpressJS. (2025). Express - Fast, unopinionated, minimalist web framework for Node.js. Recuperado de https://expressjs.com/

GitHub Docs. (2025). GitHub Actions Documentation. Recuperado de https://docs.github.com/es/actions

Jestjs.io. (2025). Jest · Delightful JavaScript Testing. Recuperado de https://jestjs.io/