ESPE	INFORME DE PRÁCTICA DE LABORATORIO

DEPARTAMENTO:	Ciencias de la Computación	CARRERA:	Ing. de Software		
ASIGNATURA:	Pruebas de Software	NIVEL:	Séptimo	FECHA:	08/10/2025
DOCENTE:	Ing. Luis Castillo	PRÁCTICA N°:	1	CALIFICACIÓN:	

CI/CD usando GitHub Actions

Allan Vinicio Panchi Pillajo

RESUMEN

Este laboratorio introduce la integración continua (CI) y entrega continua (CD) mediante GitHub Actions, usando una aplicación sencilla en Node.js. Se automatizan tareas como instalación de dependencias con npm, ejecución de pruebas unitarias con Jest y análisis estático con ESLint. El flujo se activa automáticamente en cada push o pull request hacia la rama principal, permitiendo detectar errores de forma temprana, reforzar buenas prácticas de codificación y simular procesos de despliegue automatizado.

Palabras clave: CI/CD, GitHub Actions, Node.js, npm, Jest, ESLint, automatización, integración continua, pruebas unitarias, análisis estático.

1. INTRODUCCIÓN:

En el desarrollo de software moderno, la automatización se ha convertido en un elemento clave para garantizar la calidad, consistencia y velocidad en la entrega de productos. La práctica de Integración Continua (CI) y Entrega Continua (CD) permite que los equipos integren cambios de código de manera frecuente y confiable, reduciendo errores y optimizando el ciclo de vida del desarrollo.

En este laboratorio, se implementa un flujo de CI/CD mediante GitHub Actions utilizando una aplicación sencilla en Node.js. El flujo automatiza tareas críticas como la instalación de dependencias con npm, la ejecución de pruebas unitarias con Jest y el análisis estático de código con ESLint. Además, se simula un proceso de despliegue automatizado, evidenciando el potencial de los pipelines para mejorar la productividad y asegurar la calidad del software antes de su entrega.

Esta experiencia permite comprender de manera práctica cómo los flujos de trabajo automatizados ayudan a detectar errores de forma temprana, reforzar buenas prácticas de programación y facilitar la colaboración en proyectos de desarrollo distribuidos.

2. OBJETIVO(S):

- Configurar un flujo de integración continua en GitHub Actions que ejecute automáticamente pruebas y análisis estático al detectar cambios en la rama principal.
- Implementar pruebas unitarias con Jest para validar la funcionalidad del sistema en cada actualización del código.
- Aplicar ESLint para reforzar buenas prácticas y detectar inconsistencias o errores de codificación de forma temprana.

3. MARCO TEÓRICO:

Node.js

Node.js es un entorno de ejecución de JavaScript basado en el motor V8 de Google Chrome. Permite ejecutar código JavaScript fuera del navegador y es ampliamente utilizado para crear aplicaciones del lado del servidor. Su modelo de E/S no bloqueante y orientado a eventos lo hace eficiente y escalable.

npm (Node Package Manager)

npm es el gestor de paquetes oficial de Node.js. Facilita la instalación, actualización y administración de librerías y dependencias necesarias para el desarrollo de aplicaciones. También permite definir scripts para automatizar tareas de desarrollo, pruebas y despliegue.

GitHub Actions

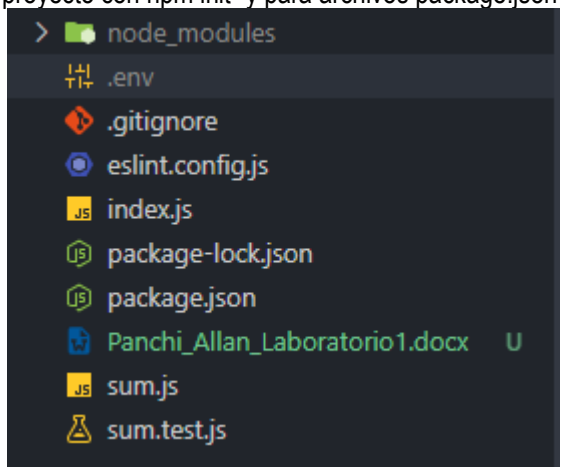
GitHub Actions es una herramienta de automatización integrada en GitHub que permite definir flujos de trabajo (workflows) mediante archivos YAML. Estos flujos se activan por eventos como commits, pull requests o despliegues. Dentro de cada flujo se ejecutan jobs y steps en runners, que pueden ser entornos Linux, Windows o macOS.

CI/CD (Integración Continua / Entrega Continua)

- Integración Continua (CI): Práctica de desarrollo que integra de forma frecuente los cambios de código en un repositorio central, ejecutando automáticamente compilaciones y pruebas para detectar errores lo antes posible.
- Entrega Continua (CD): Extiende la CI, automatizando la entrega del software a entornos de staging o producción, reduciendo el tiempo de liberación y minimizando riesgos.

4. DESCRIPCIÓN DEL PROCEDIMIENTO:

Creación de la estructura básica del proyecto, archivo .gitignore, index.js, sum.js, sum.test.js, archivo de configuración eslint, e iniciar un proyecto con npm init -y para archivos package.json



Codificación de las funciones a probar, sum.js, sum.test.js y esling.config.js

```
function suma(a, b){
    return a + b;
}

module.exports = suma;
```

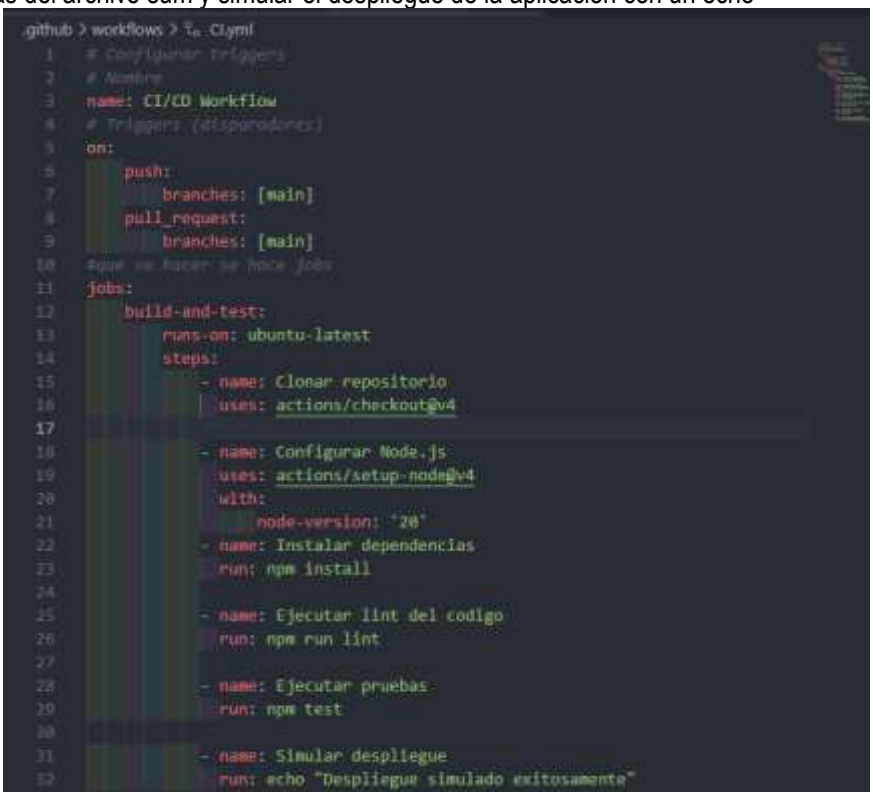
```
const add = require('./sum');

test('Suma 1 + 2 = 3', () => {
    expect(add(1, 2)).toBe(3);
});
```

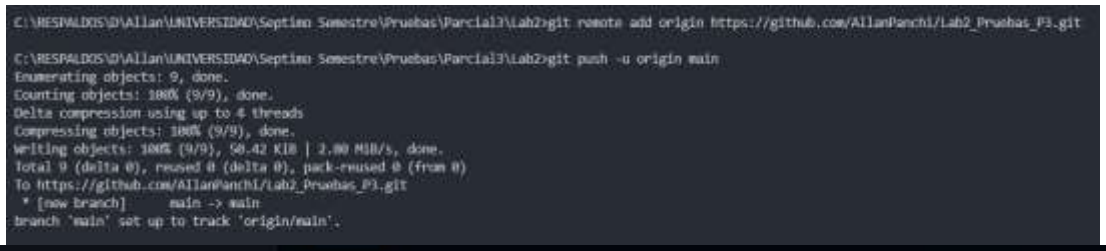
eslint.config.js > default

```
export default [
    {
        files: ['**/*.js'],
        languageOptions: {
            ecmaVersion: 2020,
            sourceType: 'module'
        },
        rules: [
            {
                'semi': ['error', 'always'],
                'quotes': ['error', 'single']
            }
        ]
    }
];
```

Creación del archivo YAML en la carpeta .github/workflow en donde se va a hacer lo siguiente, clonar un repositorio, va a configurar las dependencias, instalar las dependencias, ejecutar el archivo de configuración eslint, correr el archivo de pruebas del archivo sum y simular el despliegue de la aplicación con un echo



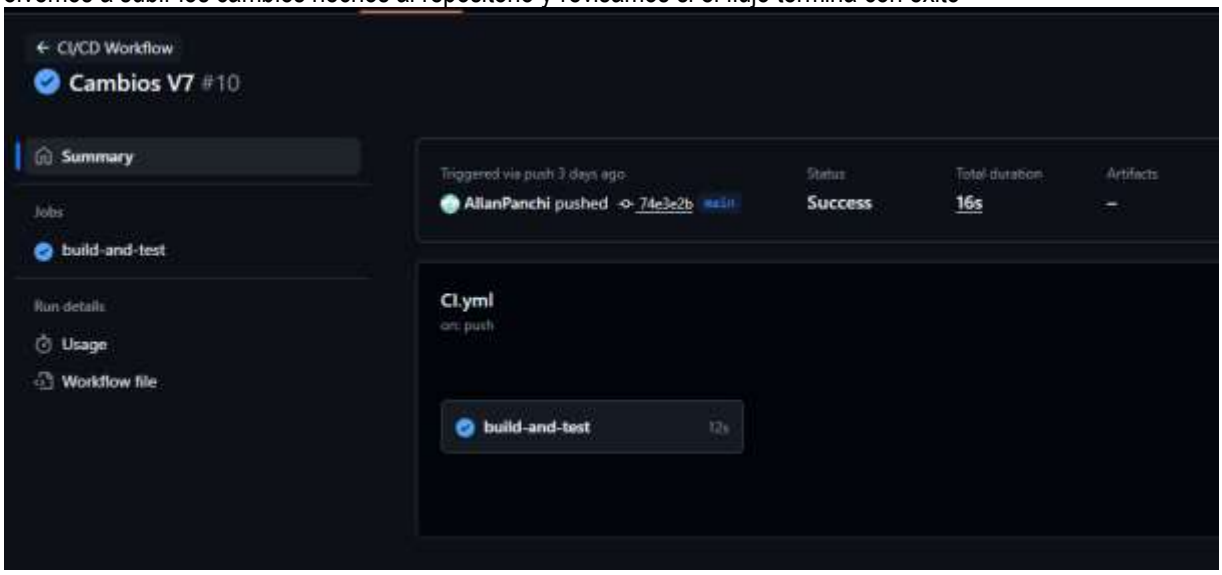
Una vez realizado el archivo YAML, subimos todos los archivos al repositorio y revisamos el flujo en el apartado de actions.



Como se observa en la imagen, se ve que existen algunos errores en el archivo eslint cuando se lo quiso ejecutar, cambiamos el archivo eslint para que se ejecute el flujo correctamente

```
eslint.config.js > ...
export default [
    {
        files: ['**/*.js'],
        languageOptions: {
            ecmaVersion: 2020,
            sourceType: 'module'
        },
        rules: {
            'semi': ['error', 'always'],
            'quotes': ['error', 'single']
        }
    }
];
```

Volvemos a subir los cambios hechos al repositorio y revisamos si el flujo termina con éxito



Se muestra que el flujo termino con éxito, es decir, que los archivos cumplen tanto con las condiciones del eslint que se definieron en el archivo de configuración, y las pruebas que se definieron en el archivo sum.test.js.

A continuación se realizarán las actividades extras, se hicieron las funciones de Fibonacci y la de factorial en un archivo math.js

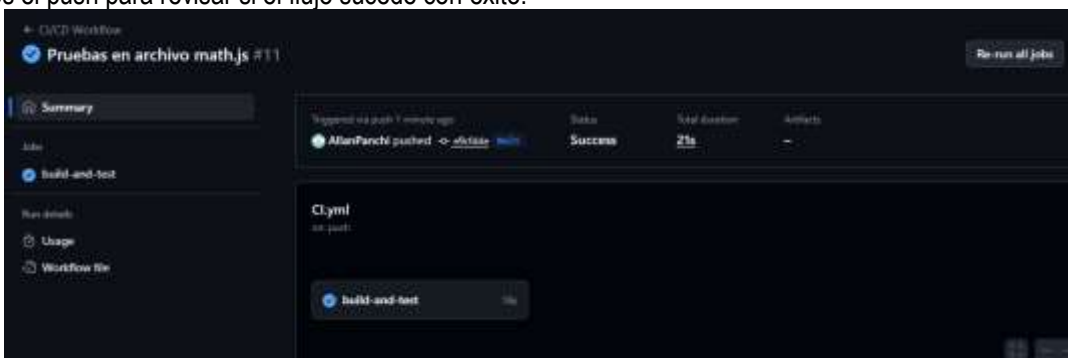
```
math.js > factorial
1  function fibonacci(n) {
2      if (n <= 1) return n;
3      return fibonacci(n - 1) + fibonacci(n - 2);
4  }
5
6  function factorial(n) {
7      if ((n === 0) || (n === 1)) return 1;
8      return n * factorial(n - 1);
9  }
```

Se programan las pruebas en archivos distintos, uno para la serie de Fibonacci y otra para el factorial

```
fibonacci.test.js > ...
1  const fib = require('./math.js');
2
3  test(
4      'fibonacci',
5      () => {
6          expect(fib.fibonacci(0)).toBe(1);
7          expect(fib.fibonacci(1)).toBe(1);
8      }
9  );
```

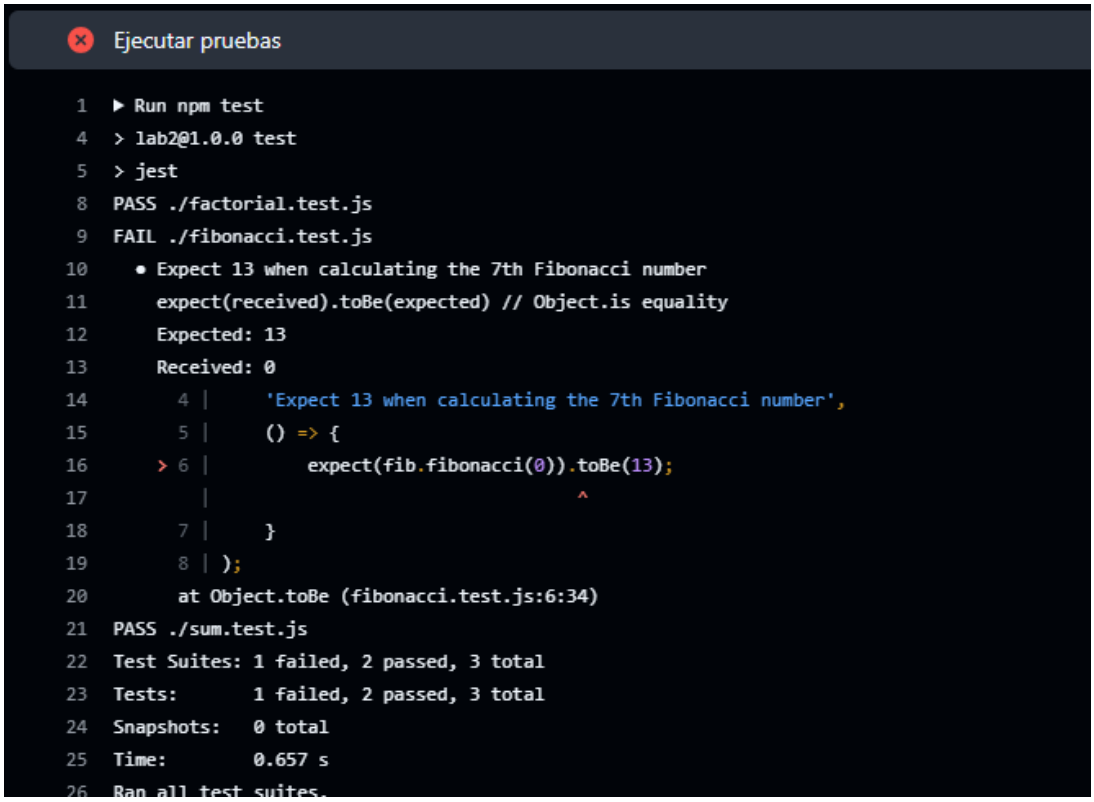
```
factorial.test.js > ...
1  const fac = require('./math.js');
2
3  test('Factorial of 0 is 1', () => {
4      expect(fac.factorial(0)).toBe(1);
5  });
6
```

Hacemos el push para revisar si el flujo sucede con éxito.



Comprobamos que el flujo terminó con éxito, ahora vamos a realizar un cambio en las pruebas para que fallen y que eso se muestren en el flujo del github actions.

```
fibonacci.test.js > ...
1  const fib = require('./math.js');
2
3  test(
4      'Expect 13 when calculating the 7th Fibonacci number',
5      () => {
6          expect(fib.fibonacci(0)).toBe(13);
7      }
8  );
```



Como vemos, la prueba ha fallado, como se ha esperado, una vez hechos estos cambios, volvemos a cambiar para que las pruebas pasen de forma exitosa.

Con esto ya hemos revisado las funciones básicas de github actions.

5. CONCLUSIONES:

- La implementación de un flujo de CI/CD con GitHub Actions permite detectar errores y fallos de calidad de forma temprana, optimizando el ciclo de desarrollo.
- El uso combinado de Jest y ESLint garantiza no solo la corrección funcional, sino también el cumplimiento de estándares de codificación.
- Automatizar procesos reduce la intervención manual, mejora la consistencia de las entregas y aumenta la productividad del equipo de desarrollo.

6. RECOMENDACIONES:

- Mantener actualizadas las dependencias y herramientas de análisis para evitar vulnerabilidades y problemas de compatibilidad.

- Integrar etapas adicionales en el flujo, como pruebas de integración y despliegues reales a entornos de prueba o producción.

- Configurar notificaciones automáticas para que el equipo reciba alertas inmediatas ante fallos en el pipeline.

7. BIBLIOGRAFÍA:

[1] GitHub, “GitHub Actions Documentation,” 2025. [En línea]. Disponible en: https://docs.github.com/actions
[2] Node.js Foundation, “Node.js v22 Documentation,” 2025. [En línea]. Disponible en: https://nodejs.org/en/docs
[3] npm, “npm Documentation,” 2025. [En línea]. Disponible en: https://docs.npmjs.com
[4] M. Fowler, “Continuous Integration,” martinfowler.com, 2024. [En línea]. Disponible en: https://martinfowler.com/articles/continuousIntegration.html