|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **DEPARTAMENTO:** | **Ciencias de la Computación** | **CARRERA:** | **Ingenieria de Software** | | |
| **ASIGNATURA:** | Pruebas de software | **NIVEL:** | 6to | **ASIGNATURA:** | Pruebas de software |
| **DOCENTE:** | Ing. Luis Castillo | **PRÁCTICA N°:** | 1 | **DOCENTE:** | Ing. Luis Castillo |

**Manejo de Pruebas Unitarias y CI/CD en Node.js con Express y GitHub Actions**

**Bryan Roberto Quispe Romero**

**RESUMEN**

Durante el laboratorio se desarrolló un proyecto base en Node.js utilizando Express para crear un servidor web simple. Se configuraron las dependencias necesarias, incluyendo Jest para la ejecución de pruebas unitarias y ESLint para el análisis estático del código. Se implementaron funciones básicas como sum, factorial y fibonacci, junto con sus pruebas automatizadas usando Jest.  
Además, se configuró un flujo de Integración Continua (CI) con GitHub Actions, que ejecuta automáticamente las pruebas y validaciones de código en cada cambio subido al repositorio. Como parte de la práctica, se introdujo un error intencional en una función para observar la falla del flujo CI y posteriormente se corrigió para verificar la ejecución exitosa.  
**Palabras Claves:** Node.js, Express, Jest, ESLint, GitHub Actions, CI/CD, Pruebas Unitarias.

1. **INTRODUCCIÓN:**

El objetivo de esta práctica fue fortalecer el manejo de entornos Node.js y aplicar el enfoque de desarrollo guiado por pruebas (TDD) junto con técnicas de integración continua. Se buscó fomentar la disciplina en la construcción de aplicaciones modernas mediante la correcta configuración de un servidor Express, la implementación de pruebas unitarias con Jest y el uso de GitHub Actions para automatizar la validación del código.  
El laboratorio permitió al estudiante comprender la importancia de implementar pruebas desde las primeras fases del desarrollo y mantener un flujo de CI/CD que garantice la calidad del software antes de llegar a producción.

1. **OBJETIVO(S):**

**Objetivo General:**

* Implementar un proyecto Node.js con pruebas unitarias y un flujo de CI/CD usando GitHub Actions.

**Objetivos Específicos:**

* Crear la estructura base del proyecto con Node.js y Express.
* Configurar Jest para ejecutar pruebas unitarias.
* Configurar ESLint para aplicar buenas prácticas de código.
* Implementar funciones y sus pruebas (sum, factorial, fibonacci).
* Configurar un flujo de CI/CD con GitHub Actions para automatizar pruebas y validaciones.
* Simular fallos en el flujo CI y corregirlos para asegurar la estabilidad del código.

1. **MARCO TEÓRICO:**

* **Node.js:** Entorno de ejecución para JavaScript que permite desarrollar aplicaciones del lado del servidor.
* **Express:** Framework minimalista de Node.js para crear servidores y manejar rutas HTTP.
* **Jest:** Framework de pruebas de JavaScript que permite definir, ejecutar y verificar pruebas unitarias.
* **ESLint:** Herramienta de análisis estático que ayuda a encontrar y corregir problemas en el código.
* **GitHub Actions:** Plataforma de automatización que permite crear flujos de trabajo (workflows) para ejecutar tareas como pruebas, compilaciones y despliegues automáticamente.
* **CI/CD:** Práctica que integra cambios de código de manera continua y despliega aplicaciones con frecuencia, asegurando calidad y rapidez.

1. **DESCRIPCIÓN DEL PROCEDIMIENTO:**

**PARTE 1: Establecimiento de la estructura del proyecto base**

**Crear la carpeta del proyecto y abrirla en el terminal.**

**Inicializar package.json:**

****

**Instalar Express:**

****

**Instalar Jest y ESLint como dependencias de desarrollo:**

****

**PARTE 2: Creación de archivos base**

**index.js**

* + **Se creó un servidor Express con un endpoint / que devuelve "Servidor funcionando".**
  + **Se levantó en el puerto 3000.**

**sum.js**

* + **Función que suma dos números y devuelve el resultado.**
  + **Exportada para su uso en pruebas.**

**sum.test.js**

* + **Prueba unitaria con Jest verificando la suma de dos números.**

**math.js**

* + **Funciones adicionales factorial y fibonacci.**

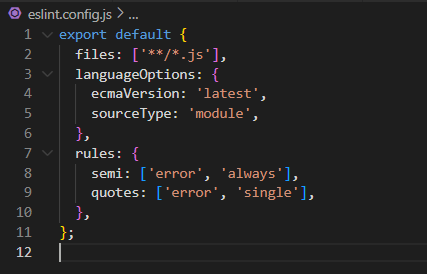
**math.test.js**

* + **Pruebas unitarias para factorial y fibonacci.**

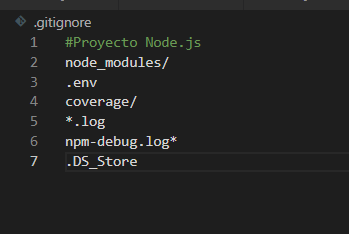
**package.json configurado con:**

****

**eslint.config.js con reglas básicas.**

****

**.gitignore ignorando node\_modules/ y archivos temporales.**

****

**PARTE 3: Configuración de Git y subida al repositorio**

1. **CONCLUSIONES:**

 La API desarrollada respondió correctamente incluso en condiciones de alta demanda (hasta 300 VUs), sin sobrepasar los límites críticos de latencia ni errores.

 El umbral de duración (p95 < 500 ms) se cumplió en todos los escenarios, demostrando una buena capacidad de procesamiento concurrente.

 El uso de K6 como herramienta de pruebas de carga es altamente efectivo para identificar puntos críticos en un sistema backend desde etapas tempranas del desarrollo

.

1. **RECOMENDACIONES:**

 Automatizar **estas pruebas** dentro del pipeline CI/CD, permitiendo detectar regresiones de rendimiento tras cada cambio de código.

 Realizar pruebas adicionales con escenarios de spike testing y soak testing para simular picos súbitos o cargas prolongadas.

 Escalar horizontalmente los servicios en producción si se espera crecimiento del tráfico, usando contenedores o instancias paralelas.

1. **BIBLIOGRAFÍA:**

 Grafana K6 Docs: https://k6.io/docs

 Express.js Documentation: https://expressjs.com

 Guía del laboratorio: VDC-GUI-2024-V2-015