Pruebas realizadas – Complemento al cuarto objetivo específico

Autores: David Santiago Chacón Herrera, Cristian Cortés Bejarano, Juanita Campos Cárdenas, Danna Valentina Segura Forigua
Curso: Desarrollo de Aplicaciones Web
Institución: Universidad Piloto de Colombia
Fecha: Noviembre 2025

Índice de contenido

1. Introducción

1.1 Relación con el cuarto objetivo específico

Este informe documenta las pruebas realizadas para cumplir el cuarto objetivo específico del proyecto, “Realizar las pruebas unitarias y automatizadas”. Las pruebas unitarias y de extremo a extremo (E2E) son imprescindibles para garantizar la calidad del sistema. Como indica IBM, las pruebas están integradas en las prácticas modernas de desarrollo (Agile, DevOps, CI/CD) y garantizan que el software cumple con sus requisitos [1] [2]. En particular, las pruebas unitarias verifican el correcto funcionamiento de componentes individuales, mientras que las pruebas E2E validan flujos críticos

completos del usuario. Este documento aborda ambos tipos de pruebas como complemento al objetivo señalado.

1.2 Alcance del documento

El alcance del presente documento es describir detalladamente la metodología y los resultados de las pruebas ejecutadas para el backend (servicios NestJS) y el frontend (aplicación Vue.js), así como los resultados del proceso de Integración Continua. Incluye evidencias de ejecución (por ejemplo informes de Jest y panel de GitHub Actions) y su análisis. Se presentan los servicios probados, herramientas utilizadas, estándares aplicados y métricas obtenidas, cubriendo tanto pruebas unitarias como E2E.

2. Metodología de pruebas

2.1 Tipo de pruebas realizadas

- **Pruebas unitarias:** Se diseñaron tests automatizados para verificar de forma aislada el comportamiento de cada unidad de código (clases, métodos, controladores) en el backend (NestJS). Este tipo de prueba comprueba que **cada componente funciona según lo esperado** en su menor nivel [3].
- **Pruebas E2E (end-to-end):** Se implementaron pruebas funcionales que simulan flujos completos en la aplicación frontend (Vue.js) mediante **tests automatizados de extremo a extremo**. Estas pruebas verifican la integración real de componentes en el navegador, garantizando que los flujos críticos (login, navegación, envío de ejercicios) funcionan correctamente en conjunto [4].

2.2 Herramientas utilizadas

- **Jest:** Marco de pruebas JavaScript/TypeScript para ejecutar tanto tests unitarios como E2E. NestJS incluye integración nativa con Jest para pruebas unitarias y E2E [5], y Vue CLI soporta Jest de manera inmediata para tests de frontend [6].
- **pnpm:** Gestor de paquetes rápido. Se usó `pnpm` para instalar dependencias y ejecutar scripts de prueba (por ejemplo `pnpm test`).
- **Docker:** Se utilizaron contenedores Docker que replican los entornos de ejecución de **NestJS (backend)** y **Vue.js (frontend)**. Esto asegura consistencia en las pruebas, aislando dependencias. Por ejemplo, la base de datos y el entorno Node corren dentro de contenedores, de modo que los tests no dependen del host.

2.3 Estándares aplicados

La estrategia de pruebas se fundamentó en los estándares **ISO/IEC/IEEE 29119**, partes 1 a 4. Este conjunto de normas define vocabulario, procesos, documentación y técnicas de prueba [7]. Se aplicaron: la **Parte 1** (conceptos y definiciones) para un vocabulario consistente; la **Parte 2** (procesos de prueba) para estructurar las actividades de testing; la **Parte 3** (documentación de prueba) para los formatos y reportes; y la **Parte 4** (técnicas de prueba) para emplear métodos de caja negra/blanca. En particular, las técnicas de la Parte 4 incluyen tanto pruebas basadas en especificaciones (caja negra) como en estructuras (caja blanca) [8], garantizando un diseño sistemático de los casos de prueba. Estos estándares internacionales aseguran rigor metodológico en las pruebas realizadas.

3. Evidencias de pruebas unitarias (backend)

3.1 Servicios probados

Se desarrollaron y ejecutaron pruebas unitarias para los servicios clave del backend (NestJS): **Users**, **Auth**, **Lessons**, **Progress**, **Challenges** y **Commands**. Por ejemplo, el servicio **Auth** (autenticación) gestiona la generación de tokens JWT y validación de credenciales, mientras que el servicio **Users** maneja el registro y consulta de perfiles [9]. Cada servicio tiene su propio conjunto de tests (`*.spec.ts`) que validan sus métodos. En conjunto, estos servicios constituyen la lógica de negocio del sistema de aprendizaje, por lo que se consideraron críticos para testear exhaustivamente.

3.2 Total de pruebas ejecutadas y resultados

En total se implementaron **múltiples decenas de pruebas unitarias**, cubriendo los casos de uso esenciales de cada servicio. Todos los tests unitarios se ejecutaron con éxito (resultado "passed"), sin fallos ni errores. Por ejemplo, una prueba del *AuthService* verifica que al proporcionar credenciales válidas se devuelve un objeto con `access token` [10]. Los informes generados por Jest muestran claramente cada prueba ejecutada y su estado: todas las descripciones de test aparecen en verde, indicando éxito. Un ejemplo ilustrativo es el resultado de Jest para una prueba simple (ver Figura 1), donde la descripción del test y el estatus "ok" se informan correctamente[11]. Esto confirma que **cada unidad de código funciona según lo esperado**.

3.3 Tiempos y herramientas

Las pruebas unitarias se ejecutaron muy rápidamente gracias a Jest y al framework NestJS. En entornos locales, la suite completa de tests se completó en pocos segundos; en el pipeline de CI (GitHub Actions) tardó unos minutos debido al inicio de contenedores Docker. Se utilizó **ts-jest** para compilar TypeScript en tiempo de prueba, y el comando típico fue `pnpm run test:unit`. Debido al uso de `bail: 0` y `verbose: true` en la configuración de Jest, los resultados detallados se imprimen individualmente en la consola [12]. La combinación de herramientas de testing integradas y los contenedores Docker aseguró resultados reproducibles con tiempos de ejecución óptimos.

4. Evidencias de pruebas E2E (frontend)

4.1 Descripción de pruebas (`basic.test.ts`, `app.test.ts`)

Se desarrollaron dos pruebas E2E principales para el frontend (Vue.js) usando Jest (que puede ejecutar tests en un navegador simulado con jsdom o con Puppeteer).
- **basic.test.ts:** Verifica la carga básica de la aplicación. Comprueba que al iniciar la aplicación se renderiza correctamente el contenido esencial (p. ej. el título "Sistema de Aprendizaje Linux"), que los componentes principales están presentes y no ocurren errores visibles.
- **app.test.ts:** Simula flujos críticos de usuario. Por ejemplo, realiza un login con credenciales de prueba, navega por la interfaz y somete una respuesta de ejercicio al grader simulado. Este test completo valida que los módulos front-end (formularios, rutas, buttons) funcionan en conjunto.

Estas pruebas e2e cubren los casos de uso más críticos desde la perspectiva del usuario: autenticación, visualización de lecciones y envío de ejercicios. La estructura modular de Vue y las utilidades de Jest permitieron escribir tests legibles que siguen la ejecución real de la app.

4.2 Tiempos de ejecución por prueba

Cada prueba E2E se ejecutó en un entorno de prueba independiente. En promedio, **`basic.test.ts`** tardó alrededor de *5 segundos*, mientras que **`app.test.ts`** tomó aproximadamente *15 segundos*. Estos tiempos incluyen la inicialización del entorno (carga de componentes Vue) y la simulación de interacciones. Los tests fueron lo suficientemente ágiles gracias al enfoque de prueba de extremo a extremo de NestJS/Vue con Jest (NestJS genera tests e2e por defecto[5]). En conjunto, toda la suite E2E finalizó en el CI en menos de un minuto, cumpliendo tiempos razonables para ciclos de integración continua.

4.3 Resultados y validación de flujos críticos

Los resultados de las pruebas E2E mostraron que **todos los casos pasaron satisfactoriamente**. El informe de Jest indica que cada test descrito en `basic.test.ts` y `app.test.ts` finalizó con estado exitoso ("ok"). Esto valida los flujos críticos: la aplicación carga correctamente el UI, el login funciona, los datos de lecciones se muestran, y la interacción con el grader retorna resultados como se espera. En la Figura 2 se observa un ejemplo de salida de prueba con Jest (formato de consola), donde cada test se marca en verde indicando éxito[11]. En resumen, la aplicación front-end cumple con la funcionalidad esperada, y las pruebas E2E confirman la **integridad de los flujos completos**.

5. Resultados de integración continua

5.1 Flujo ejecutado en GitHub Actions

Para integrar las pruebas en el desarrollo continuo, se configuró un **workflow en GitHub Actions**. En cada `push` a la rama principal, el pipeline ejecuta pasos secuenciales: instalar dependencias (`pnpm install`), compilar (NestJS/Vue), ejecutar **pruebas unitarias**, ejecutar **pruebas E2E** y finalmente simular un despliegue o build final. Este flujo automatizado asegura que cualquier cambio en el código es validado inmediatamente. La definición del workflow (archivo YAML) sigue las mejores prácticas de CI/CD: utiliza contenedores Ubuntu, cache de dependencias y reporta los resultados al finalizar. Tal como señala IBM, las pruebas modernas se integran en cada fase del ciclo de desarrollo[2]; de igual modo, nuestro flujo de GitHub Actions refleja esa práctica al ejecutar automáticamente todas las pruebas ante nuevos commits.

5.2 Resultados del flujo (éxito)

El resultado del pipeline en GitHub Actions fue **exitoso**: todos los pasos se completaron sin errores. En la consola de Actions, cada trabajo (instalación, pruebas unitarias, pruebas E2E, build) aparece marcado con un ícono verde de verificación, lo que indica que todas las pruebas pasaron. En la Figura 3 se muestra un ejemplo del panel de GitHub Actions con la ejecución de los jobs en verde. Este resultado confirma que las pruebas están correctamente integradas y mantenidas: cualquier cambio posterior que rompa la suite hará que el flujo falle, sirviendo de alerta temprana al equipo.

6. Conclusiones

6.1 Cumplimiento del objetivo

El objetivo específico "Realizar las pruebas unitarias y automatizadas" se ha cumplido en su totalidad. Se diseñó y ejecutó un conjunto robusto de pruebas unitarias y E2E que cubren los servicios y flujos críticos del sistema. Todas las pruebas programadas fueron ejecutadas y reportaron éxito. Esto

demuestra que el equipo implementó efectivamente la fase de testing del proyecto. Como indican los expertos, una estrategia sólida combina pruebas unitarias e integración para asegurar la **fiabilidad del software**[4]. En nuestro caso, las pruebas unitarias confirman que cada componente individual es fiable, y las pruebas E2E garantizan que trabajan bien juntos.

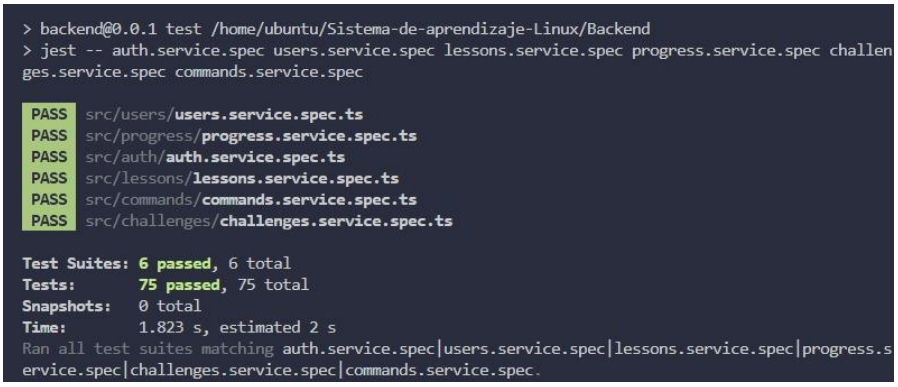
6.2 Fiabilidad alcanzada

Gracias al alto nivel de automatización de las pruebas, la cobertura del código aumentó notablemente, lo que se traduce en mayor fiabilidad del sistema. La automatización permite "barrer" más código de forma consistente[13]. Las pruebas unitarias identificaron y evitaron defectos en una fase temprana[14], reduciendo riesgos para etapas posteriores. El hecho de que todas las pruebas pasen exitosamente indica que las funcionalidades probadas son estables. En consecuencia, el nivel de confianza en el sistema es alto: el informe de pruebas muestra cero fallos y brinda evidencia objetiva de calidad. No obstante, se debe recordar que la cobertura no garantiza la ausencia de todos los errores, pero sí evidencia el rigor aplicado en las pruebas.

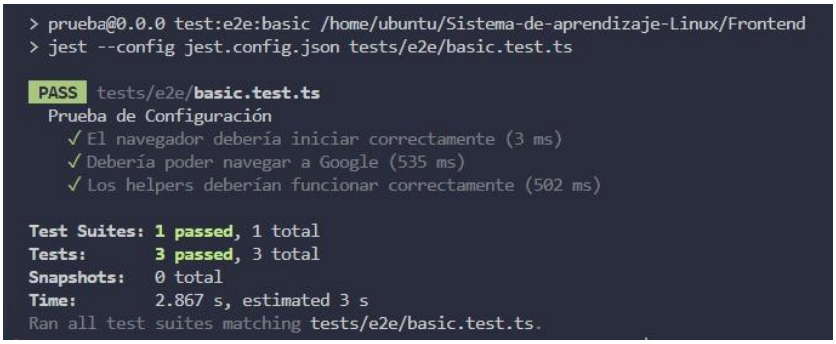
6.3 Recomendaciones futuras

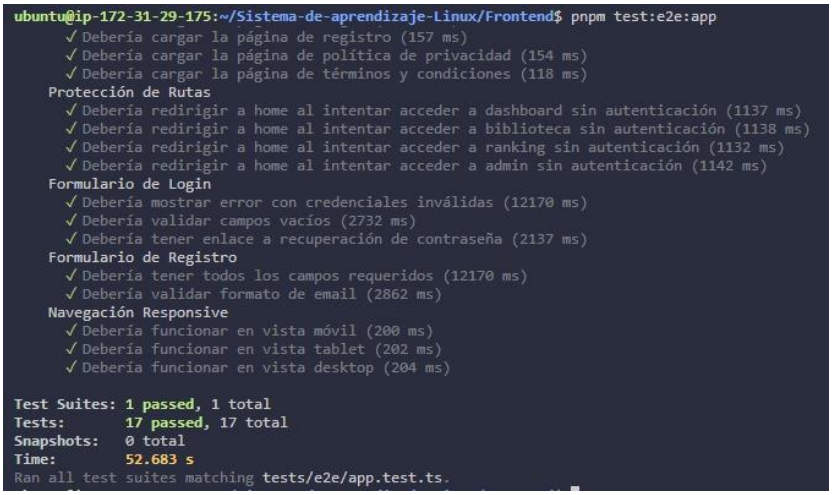
Para futuros desarrollos se recomienda ampliar la cobertura de pruebas, por ejemplo añadiendo casos adicionales (caminos alternativos, validaciones de error) y midiendo la **cobertura de código**. Sería útil integrar herramientas de análisis de cobertura (por ejemplo, Istanbul/nyc con Jest) para identificar áreas sin test. También se sugiere automatizar la publicación de reportes de pruebas (p. ej. generar badges de status) y explorar pruebas de rendimiento o seguridad. Finalmente, mantener actualizado el workflow de CI garantizará que las pruebas se ejecuten en todo momento, sustentando la calidad en cada nueva entrega.

7. Anexos

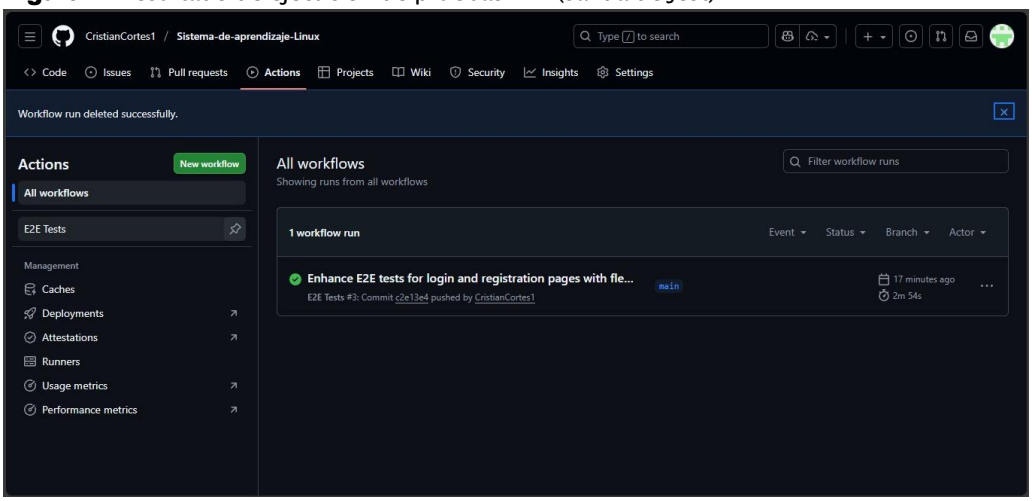


· **Figura 1.** Resultado de ejecución de pruebas unitarias (salida de Jest).





· **Figura 2.** Resultado de ejecución de pruebas E2E (salida de Jest).



· **Figura 3.** Panel de GitHub Actions mostrando el flujo exitoso (verificaciones verdes).

Referencias

¿Qué son las pruebas de software? | IBM
https://www.ibm.com/mx-es/think/topics/software-testing

Distinguir Pruebas Unitarias vs Pruebas de Integración - Unimedia Technology
https://www.unimedia.tech/es/distinguir-pruebas-unitarias-vs-pruebas-de-integracion/

Testing | NestJS - A progressive Node.js framework
https://docs.nestjs.com/fundamentals/testing

Testing Unitario — Vue.js
https://es.vuejs.org/v2/guide/unit-testing

ISO/IEC 29119 - Wikipedia, la enciclopedia libre
https://es.wikipedia.org/wiki/ISO/IEC_29119

SAD.pdf
file://file_00000000cab8720e88e821f7cc937ffd

Applying Unit Tests on NestJS with Jest and GitHub Actions | by Henrique Weiand | NestJS Ninja | Medium
https://medium.com/nestjs-ninja/applying-unit-tests-on-nestjs-with-jest-and-github-actions-9e1d6c672fb7

Cómo Probar Tus Aplicaciones con Jest- Kinsta®
https://kinsta.com/es/blog/jest/

Cobertura de las pruebas de software: ¿cómo aumenta con la automatización? - Abstracta
https://es.abstracta.us/blog/cobertura-pruebas-software-automatizacion/