**Benemérita Universidad Autónoma de Puebla**

**Facultad de Ciencias de la Computación**


# T E S I S

# “AUTOMATIZACIÓN DE PRUEBAS EN EL DESARROLLO DE APLICACIONES WEB”

**Presenta:** *Elia Maria Toriz Sánchez*
**Para obtener el grado de:** *Ingeniería en tecnologías de la información*

**Director:** *Abraham Sánchez López*

**Puebla, Pue, 11/25**

BUAP

# “AUTOMATIZACIÓN DE PRUEBAS EN EL DESARROLLO DE APLICACIONES WEB”

## INGENIERIA EN TECNOLOGIAS DE LA INFORMACION

2025


**Presentado por:**
Elia Maria Toriz Sánchez

# RESUMEN

Las pruebas automatizadas son esenciales en el proceso de desarrollo de software por múltiples razones, que van desde la elevación de la calidad del producto hasta la optimización de los procedimientos de desarrollo. Entre sus principales funciones se incluyen garantizar la calidad de los productos, ahorrar tiempo y aumentar la eficiencia, alcanzar una cobertura amplia, mejorar la documentación, implementar integración y despliegue continuos, reducir costos a largo plazo y proporcionar una base sólida para la evolución y el crecimiento del proyecto

La implementación de pruebas automatizadas no solo beneficia al equipo de desarrollo, sino también a los usuarios finales, quienes obtienen un producto más confiable y de mayor calidad. El presente estudio se enfoca en explorar e implementar las mejores prácticas para la automatización de pruebas, con el objetivo principal de identificar y documentar estrategias eficaces para llevar a cabo pruebas unitarias, de integración y end-to-end.

Asimismo, se abordarán los desafíos específicos que surgen al trabajar con la integración de Firebase en un entorno Angular. Se espera que los hallazgos de esta investigación ofrezcan una guía clara y práctica para desarrolladores y equipos de calidad, facilitando así la creación de aplicaciones más robustas y fiables. El estudio incluye un análisis de herramientas de prueba, así como ejemplos concretos sobre cómo configurar y ejecutar pruebas.

Palabras clave: pruebas automatizadas, calidad del software, integración continua, Firebase, Angular, estrategias de prueba, desarrollo ágil.

# DEDICATORIA

Dedico este trabajo de tesis a mis padres, quienes han sido mis mayores apoyos y guías en este recorrido. Su amor incondicional y sus sacrificios diarios han sido el motor que me impulsa a seguir adelante. Gracias por enseñarme el valor del esfuerzo y la perseverancia, por brindarme las oportunidades necesarias para alcanzar mis sueños, y por inculcarme la importancia de luchar y no rendirme jamás.

A mis abuelos, cuya luz ha iluminado mi vida en cada paso. Su sabiduría, amor y apoyo han sido fundamentales en mi formación y en la persona que soy hoy. Su legado sigue vivo en cada uno de mis logros.

Por último, dedico este logro a todos aquellos que han creído en mí: amigos, mentores y compañeros que han estado a mi lado en este camino. Cada uno de ustedes ha dejado una huella imborrable en mi vida y ha contribuido a este momento tan significativo.

# AGRADECIMIENTOS

Quiero expresar mi más sincero agradecimiento a mis padres, Guadalupe Sánchez Domínguez y Edgar Toriz Bonilla, quienes han sido mi pilar fundamental a lo largo de este proceso. Su amor incondicional, apoyo constante y sacrificios han sido la base sobre la cual he construido mis sueños. Gracias por enseñarme a ser perseverante y por inspirarme a alcanzar mis metas.

A mi familia, cuyo amor y apoyo me han fortalecido en cada paso que he dado. Su confianza en mí ha sido un impulso para seguir adelante, y su presencia ha sido un recordatorio de la importancia de los lazos familiares.

A mis amigos, que han estado a mi lado en los momentos de alegría y en los desafíos. Su compañía y aliento han hecho este camino más llevadero y han sido una fuente constante de motivación. Gracias por cada conversación, cada risa y cada instante compartido.

Finalmente, quiero agradecer de manera especial a mi asesor de tesis. Su guía, paciencia y conocimiento han sido fundamentales en la culminación de este trabajo. Agradezco cada consejo y cada corrección, que han contribuido a mejorar no solo este proyecto, sino también mi crecimiento académico y personal.

A todos ustedes, gracias por estar en mi vida y por hacer posible este logro.

# ÍNDICE GENERAL

# ÍNDICE
## DE FIGURAS

# ÍNDICE DE TABLAS

# CAPÍTULO 1

## INTRODUCCIÓN

1.1 Estado del arte.

1.2 Planteamiento del problema.

1.3 Justificación.

1.4 Objetivos.

1.5 Alcance y limitaciones.

# INTRODUCCIÓN

En el desarrollo de software actual, no es suficiente que un sistema funcione en el momento de su implementación, debe mantener su funcionalidad a lo largo del tiempo. En un entorno donde los cambios son constantes y las expectativas de los usuarios son cada vez más exigentes, tres factores se vuelven absolutamente clave: la calidad, la velocidad de entrega y la capacidad de adaptarse sin frenar. Las aplicaciones web han evolucionado tanto que ya no ves simplemente una página, sino un entorno dinámico, en constante movimiento. Con herramientas como Angular y servicios en la nube como Firebase, es posible desarrollar aplicaciones modernas, escalables y robustas. Eso sí, a medida que ganamos potencia, también enfrentamos obstáculos nuevos que no podemos ignorar.

La complejidad aumenta significativamente, convirtiendo la estabilidad, escalabilidad e integración de servicios en desafíos técnicos considerables. Hoy más que nunca debemos asegurarnos de que todo siga funcionando incluso cuando se suman nuevas funcionalidades, cuando el equipo decide explorar algo nuevo o cuando de pronto el sistema recibe más tráfico del que esperábamos.

En ese contexto, las pruebas automatizadas resultan una bendición. Nada comparado con depender del día que tenga la persona que las ejecuta.

Cuando tienes la posibilidad de ejecutar pruebas que sirven siempre, sin importar si es temprano por la mañana o tarde por la noche, te sientes con el control y la tranquilidad del que sabe que todo está bien. Las pruebas unitarias, las de integración y las end-to-end tienen esa magia de detectar fallos antes de que lleguen al usuario, y al mismo tiempo le dan confianza al equipo para avanzar sin temor a romper algo que ya funciona.

Con prácticas como la integración continua (Fowler, 2006) y entrega continua (CI/CD), las pruebas automatizadas se convierten en un componente esencial del flujo de desarrollo. Cada cambio que llega al repositorio cuenta con un guardián silencioso que valida si todo sigue en pie inmediatamente. Esto fortalece la confiabilidad del software, hace que el equipo trabaje con más certeza y, lo más importante, entrega una experiencia más sólida a quien lo usa.

Y si quieres datos que te dejan pensando dos veces, aquí van: La implementación de pruebas automatizadas ha demostrado prevenir entre el 70% y 80% de los defectos que típicamente llegan a producción (National Association of State Chief Information Officers [NASCIO], 2022). La evidencia empírica sugiere que el costo de corrección de defectos se incrementa exponencialmente conforme avanzan las fases del desarrollo;

específicamente, un error detectado en producción puede requerir hasta 30 veces más recursos para su corrección comparado con su detección durante la fase de codificación (Jones & Analytics, 2020). Investigaciones clásicas en ingeniería de software han establecido que los costos de corrección de defectos crecen de manera logarítmica a través del ciclo de vida del desarrollo. El IBM Systems Sciences Institute documentó que un defecto identificado durante la fase de mantenimiento puede representar hasta 100 veces el costo de corrección comparado con su detección durante la codificación inicial (IBM Systems Sciences Institute, 1981). Aunque este estudio tiene más de cuatro décadas, investigaciones contemporáneas continúan validando esta relación costo-fase (Boehm & Basili, 2001).

En resumen, hoy automatizar las pruebas no es una ventaja que solo algunos equipos de élite se toman. Es una estrategia esencial, inteligente y sensata. Es la manera de proteger tu esfuerzo, cuidar la experiencia del usuario y construir software con la calidad que todos necesitamos y merecemos.

Los datos que vimos antes no solo impresionan, también dejan en evidencia algo importante: necesitamos validar lo que desarrollamos desde el inicio. Cuanto antes podamos detectar un problema, menos impacto tendrá. En proyectos donde Angular se combina con Firebase, algo bastante común en el desarrollo web actual, los desafíos aumentan. Las operaciones asíncronas, el acceso constante a servicios externos y la alta interacción con el usuario hacen que el riesgo de errores crezca si no se tiene una estrategia clara de pruebas. Por eso, contar con un enfoque bien pensado, basado en modelos como la pirámide de testing, es fundamental para reducir esos riesgos y lograr entregas estables y exitosas en los tiempos acordados.

Aunque existen herramientas como Jasmine, Karma y Protractor, muchos equipos todavía enfrentan dificultades al implementar una estrategia de pruebas automatizadas realmente efectivas. Esto se vuelve aún más complejo cuando el desarrollo incluye servicios externos como Firebase, ya que no solo se trata de probar funciones internas, sino también de validar integraciones que pueden fallar por múltiples factores. Esta situación refleja una necesidad concreta: contar con una guía clara y bien documentada que sirva como apoyo al momento de aplicar pruebas en proyectos reales.

A partir de esa necesidad nace este trabajo. El objetivo principal es explorar y aplicar buenas prácticas en la automatización de pruebas dentro de aplicaciones construidas con Angular y Firebase.

Con un enfoque práctico y ordenado, se busca aportar valor a la calidad del software desde una perspectiva técnica y realista. Al final, la idea es ofrecer a los desarrolladores y equipos de trabajo herramientas claras, útiles y aplicables que les ayuden a enfrentar los retos que hoy presenta el desarrollo web moderno.

## 1.1 ESTADO DEL ARTE

La automatización de pruebas ha dejado de ser una simple herramienta técnica para convertirse en una parte importante dentro del desarrollo de software moderno. Detrás de cada línea de código, hay esfuerzo, creatividad y muchas horas de trabajo, por eso, garantizar que todo funcione como se espera no solo es una cuestión técnica, sino también una forma de cuidar el trabajo, la dedicación y la confianza del equipo. En entornos ágiles, donde cada sprint cuenta y los cambios son constantes, la automatización se convierte en una aliada que brinda tranquilidad. Permite avanzar con paso firme, sabiendo que las pruebas acompañan cada mejora, cada refactorización y cada nueva idea que se integra en el proyecto.

Herramientas como Jasmine y Karma han demostrado ser compañeras de confianza para realizar pruebas unitarias e integradas en Angular, mientras que Protractor y Cypress brillan en el terreno de las pruebas end-to-end, donde lo que está en juego es la experiencia completa del usuario. Validar que cada flujo funcione sin errores no solo asegura calidad, sino también satisfacción y confianza en quienes usan el producto final, como menciona Meszaros (2007), seguir patrones de prueba de manera estructurada no solo hace que el código sea más mantenible, sino que también crea orden y claridad en medio del caos que muchas veces acompaña el desarrollo. Jorgensen (2013) añade que la automatización reduce la carga humana en la verificación, permitiendo que los equipos enfoquen su energía en innovar, crear y mejorar.

Aun así, existen terrenos poco explorados. La integración de tecnologías como Angular y Firebase plantea desafíos únicos: su arquitectura asíncrona y la conexión constante con servicios en la nube ponen a prueba la paciencia y la creatividad de los desarrolladores. Quienes se enfrentan a estos retos saben que no basta con automatizar: hay que hacerlo con inteligencia, estrategia y sensibilidad técnica.

Por eso, esta investigación nace con una intención clara, aportar claridad en medio de la complejidad, ofrecer herramientas reales a quienes buscan mejorar su práctica diaria y dar respuestas a una necesidad que muchos sienten pero pocos han logrado resolver. Más que un estudio técnico, este trabajo es un puente entre la teoría y la práctica, entre el esfuerzo de los equipos y la confianza en los resultados que entregan al mundo.

# 1.2 PLANTEAMIENTO DEL PROBLEMA

El desarrollo de aplicaciones web se ha vuelto un arte complejo. Cada día, los usuarios esperan experiencias más rápidas, interactivas y vivas, y detrás de esa magia hay equipos enfrentando arquitecturas cada vez más desafiantes. Frameworks como Angular hacen posible construir interfaces potentes y flexibles, mientras que Firebase acerca el poder del backend con autenticación, bases de datos en tiempo real y funciones en la nube. Pero combinar ambas tecnologías no siempre es sencillo: la complejidad crece, y con ella, los errores que pueden esconderse en cualquier rincón del sistema.

Detectar esos fallos a tiempo es vital. En entornos ágiles o DevOps, donde cada entrega cuenta, las pruebas manuales ya no bastan. La presión por avanzar rápido puede dejar pasar pequeños errores que terminan afectando la experiencia del usuario y la confianza en el producto.
Aunque existen herramientas como Jasmine, Karma, Cypress o Playwright, muchos equipos aún no logran integrarlas de forma efectiva. Falta guía, claridad y sobre todo, una metodología que haga del proceso algo natural y eficiente.

Este proyecto nace justamente de esa necesidad: crear una ruta clara y práctica para automatizar pruebas en aplicaciones Angular con Firebase. Más que código, se trata de construir confianza, calidad y tranquilidad para los equipos que buscan entregar software que realmente funcione, sin miedo a romper lo que ya está hecho.

> ***"Probar no es encontrar errores, es obtener confianza en que el software hace lo que debe."***
> ***— Gerald Weinberg***

# 1.3 OBJETIVOS

**Objetivo general:**

- Explorar las mejores prácticas para la automatización de pruebas en aplicaciones Angular que utilizan Firebase, abarcando pruebas unitarias, de integración y end-to-end.

**Objetivos específicos:**

- Implementar pruebas unitarias con Jasmine.
- Utilizar Karma como entorno de ejecución de pruebas unitarias.
- Desarrollar pruebas de integración con Karma.
- Implementar pruebas end-to-end con Protractor.
- Evaluar la cobertura de pruebas y la calidad del código.

# 1.4 JUSTIFICACIÓN

La calidad del software, es fundamental en el desarrollo de webs modernas, sobre todo si la velocidad de entrega y los requisitos cambian constantemente, en este sentido, automatizar las pruebas, no es solo una buena idea, sino algo imprescindible para asegurar que el producto funcione como se espera y asi poder manter funcionalidad, rendimiento, seguridad, y usabilidad.

Las aplicaciones creadas con Angular, un framework usado para crear interfaces interactivas, suelen usar muchos componentes y servicios. Cuando estas apps se combinan con Firebase, una plataforma backend en tiempo real con autenticación, base de datos, almacenamiento y otros servicios gestionados, la complejidad aumenta mucho más. Esto genera retos específicos, en cuanto a la gestión del estado, la sincronización de datos, la seguridad y la escalabilidad.

Así, la automatización de pruebas unitarias, de integración y end-to-end, resulta crucial, las pruebas unitarias, garantizan que cada componente trabaje por si mismo, las pruebas de integración comprueban la buena conexión entre módulos, y las pruebas end-to-end realmente verifican cómo el usuario vive toda la experiencia, como es en verdad, estas ultimas ayudan mucho a verificar que el sistema funcione como se espera.

Automatizar estas pruebas nos permite verificar fallos en etapas tempranas del desarrollo lo cual nos reduce los costos de arreglos con el tiempo, y además evita esos problemas que vuelven en versiones nuevas del software.

Adicionalmente, en ambientes laborales ágiles bajo la batuta de metodologías DevOps, donde las implementaciones son constantes y el trabajo en equipo es crucial, las pruebas automatizadas son clave para mantener un ritmo constante de entrega sin que la calidad sufra. Se transforman en un pilar fundamental para la integración continua CI y la entrega continua CD, impulsando ciclos de desarrollo más cortos, seguros, y predecibles por completo.

Este proyecto también encuentra razón de ser por su valor práctico, su aplicabilidad real al centrarse en la coordinación entre Angular y Firebase, se enfoca en un escenario común en el desarrollo actual, ya sea en startups ávidas de velocidad y escalabilidad, o en empresas establecidas que deben conservar productos de alta calidad con equipos distanciados. Elaborar una guía con las mejores prácticas para automatizar pruebas en este escenario no solo es valioso académicamente, sino que repercute con fuerza en la industria, dando a los programadores herramientas y estrategias claras para afinar sus procesos de desarrollo y asegurar productos confiables.
En resumen esta investigación además de reforzar la teoría en automatización de pruebas, busca también ofrecer soluciones aplicables, que son prácticas y esto contribuya al progreso en la calidad del desarrollo de apps web modernas usando Angular y Firebase.

## CASOS INDUSTRIALES DONDE EL TESTING EVITÓ FALLOS CATASTRÓFICOS

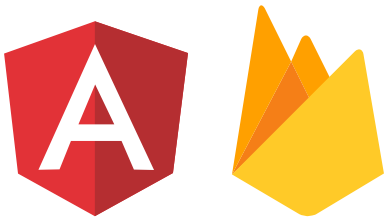
El impacto del testing en la industria tecnológica, es algo bien documentado, ya, un ejemplo notable lo es Netflix, ellos implementaron una estrategia de pruebas automatizadas y de caos muy intensiva para evitar las caídas del sistema, gracias a ello, la plataforma puede recuperarse rapido de errores inesperados, y así, mantiene su tiempo de actividad por arriba del 99.99% mejorando el estado de esta plataforma y su funcionalidad.

Otro ejemplo digno de mención es el sistema de pagos de PayPal, sin duda. En su cambio a una arquitectura de microservicios, tuvieron problemas de integración, pero la cosa se solucionó gracias a una serie de pruebas automatizadas. Estas incluían pruebas unitarias y E2E, incluso simulaciones de fallos en servicios muy importantes. Esto les permitió encontrar problemas de seguridad y rendimiento antes de llegar a los usuarios finales.Un caso emblemático que ilustra las consecuencias de la validación insuficiente es el incidente del Ariane 5 en 1996. El vehículo de lanzamiento europeo experimentó un fallo catastrófico 37 segundos después del despegue debido a un error no detectado en la reutilización de código del sistema de referencia inercial del Ariane 4. La junta investigadora concluyó que pruebas de integración más exhaustivas habrían identificado la incompatibilidad en el manejo de valores numéricos que causó el fallo (Lions et al., 1996). El incidente resultó en pérdidas estimadas superiores a 370 millones de dólares estadounidenses y representó un retroceso significativo en el programa espacial europeo

# 1.5 ALCANCE Y LIMITACIONES.

Este proyecto se enfoca en una aplicación Angular que utiliza servicios de Firebase (autenticación, base de datos y almacenamiento). Se abordarán pruebas unitarias, de integración y end-to-end utilizando Jasmine, Karma y Protractor.

Limitaciones:
- No se incluirá la implementación completa de CI/CD.
- El análisis se centrará en Angular v13 + y Firebase v9+.
- Las pruebas automatizadas no reemplazan las pruebas manuales, sino que las complementan.



# CAPÍTULO 2

## FUNDAMENTOS DE AUTOMATIZACIÓN DE PRUEBAS

2.1 Proceso de desarrollo de software y automatización de pruebas

2.2 Fundamentos y tipos de pruebas de software

2.3 Entornos modernos y desafíos en Angular con Firebase

2.4 Pruebas en metodologías ágiles y su automatización

2.5 Verificación, Validación y su aplicación práctica

2.6 Pirámide de pruebas en Angular con Firebase

2.7 Buenas prácticas y recomendaciones

# 2.1 PROCESO DE DESARROLLO DE SOFTWARE Y AUTOMATIZACIÓN DE PRUEBAS

La automatización de pruebas se ha convertido en un componente esencial del desarrollo de software moderno, como se muestra en la Figura 2.1, donde se ilustra el proceso de desarrollo que integra diferentes fases desde la planificación hasta la operación y monitoreo.
Modelos como Agile y DevOps requieren entregas continuas e iterativas, lo que hace inviable depender solo de pruebas manuales. En este contexto, la automatización permite mantener ciclos de retroalimentación cortos y mejora la confiabilidad del producto.

Angular y Firebase, tecnologías comunes en aplicaciones web modernas, se benefician particularmente de estrategias automatizadas, pues su complejidad arquitectónica y asincronía requieren validaciones constantes y precisas.

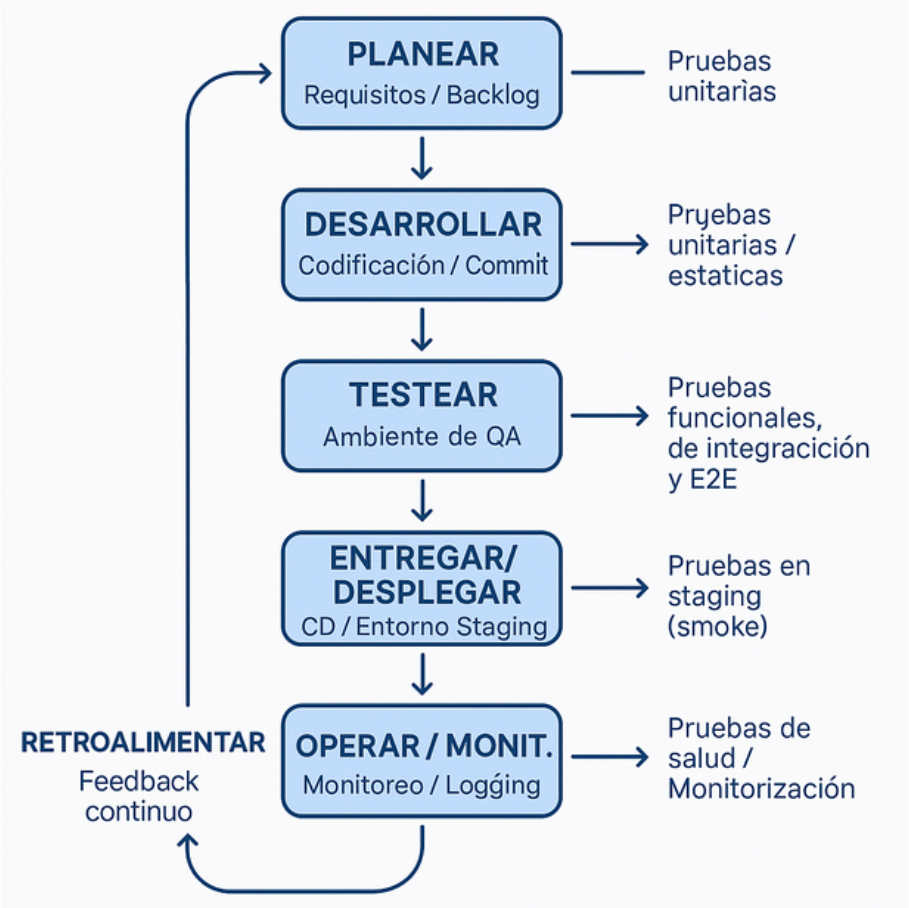



Figura 2.1 Proceso de desarrollo

# 2.2 FUNDAMENTOS Y TIPOS DE PRUEBAS DE SOFTWARE

Las pruebas de software sirven para verificar que un sistema funciona como debe, cumpliendo con los requisitos funcionales y no funcionales establecidos (Ammann & Offutt, 2016).

En Angular, las pruebas más importantes incluyen:

- Pruebas unitarias: Validan componentes o servicios individuales usando Jasmine.
- Pruebas de integración: Verifican la interacción entre módulos Angular y servicios Firebase.
- Pruebas end-to-end (E2E): Evalúan flujos completos del usuario usando herramientas como Protractor o Cypress.

Cada tipo de prueba tiene características particulares en cuanto a su alcance, velocidad de ejecución y dependencias externas, como se detalla en la Tabla 2.1.

| Tipo de Prueba | ¿Qué prueba? | Herramientas | Velocidad | Dependencias externas |
|---|---|---|---|---|
| Unitarias | Funciones individuales, servicios aislados | Jasmine, Karma | 🚀 Rápida | ❌ No |
| Integración | Interacción entre componentes, servicios, modulos, etc. | Jasmine, Karma | ⚡ Media | ⚠️ Parcial |
| End-to-End (E2E) | Flujo completo de usuario (UI real + backend simulado o real) | Cypress, Playwright, Proactor | 🐢 Lenta | ✅ Sí |

Tabla 2.1 Tabla comparativa de pruebas unitarias, integración y E2E

# 2.3 ENTORNOS MODERNOS Y DESAFÍOS EN ANGULAR CON FIREBASE

El desarrollo de aplicaciones web modernas con Angular y Firebase introduce beneficios clave como una arquitectura basada en componentes reutilizables, inyección de dependencias y un backend sin servidor altamente escalable. No obstante, también plantea desafíos específicos para la automatización de pruebas, debido a la naturaleza reactiva y asincrónica de sus servicios. La Figura 2.2 ilustra cómo se integran estos elementos en una arquitectura típica de Angular con Firebase.

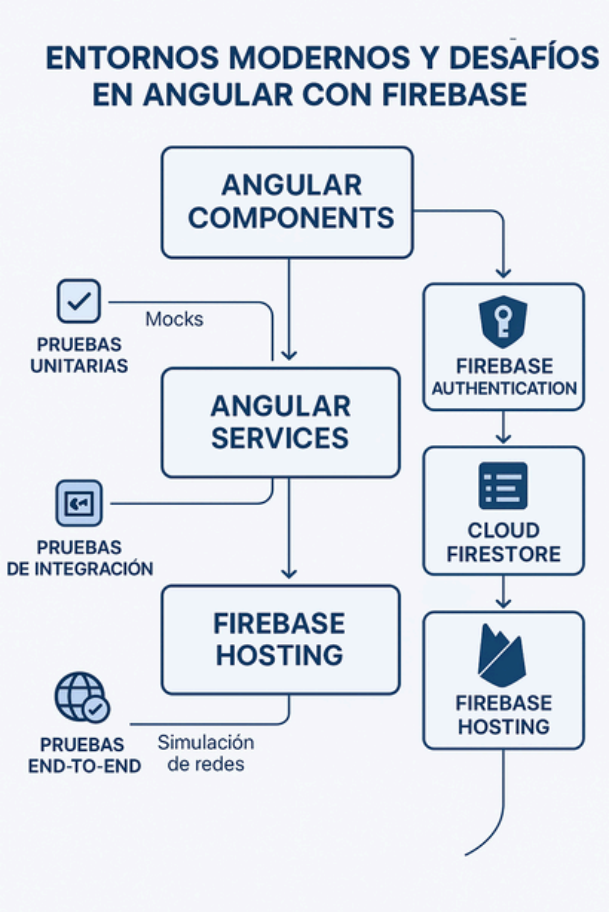



Figura 2.2 Arquitectura Angular + Firebase

## ANGULAR: ARQUITECTURA DINÁMICA Y MODULAR

El framework Angular implementa una arquitectura basada en el patrón de diseño de componentes y la inyección de dependencias, donde la aplicación se estructura mediante módulos independientes que encapsulan funcionalidades específicas (Google Developers, 2023). Esta organización modular proporciona ventajas significativas en términos de mantenibilidad y reusabilidad del código, particularmente relevantes en el contexto de pruebas automatizadas donde cada unidad puede ser validada de forma aislada.

## FIREBASE: SERVICIOS ASINCRÓNICOS Y EN TIEMPO REAL

Firebase ofrece funcionalidades como Firebase Authentication, Cloud Firestore y Firebase Storage , Firebase. (2023), que operan de manera asíncrona y dependen de eventos en tiempo real. Esto complica el diseño de pruebas, ya que:

- Los datos pueden cambiar dinámicamente durante una prueba.
- Las respuestas no son inmediatas (dependen de la red o del emulador).
- La autenticación basada en eventos debe ser interceptada o simulada correctamente.

# 2.4 PRUEBAS EN METODOLOGÍAS ÁGILES Y SU AUTOMATIZACIÓN

Los marcos de trabajo ágiles, particularmente Scrum, estructuran el desarrollo en iteraciones de duración fija conocidas como sprints, durante las cuales se entrega funcionalidad incremental validada. Esta aproximación iterativa requiere mecanismos de validación continua que garanticen que cada incremento no comprometa la estabilidad del sistema existente, convirtiendo la automatización de pruebas en un componente crítico del proceso (Schwaber & Sutherland, 2020).

### AUTOMATIZACIÓN DE PRUEBAS COMO PILAR DE CALIDAD

La automatización de pruebas se vuelve esencial en entornos ágiles para garantizar que cada nueva funcionalidad no afecte otras partes del sistema ya implementadas. Automatizar las pruebas unitarias, de integración y de extremo a extremo (E2E) en cada sprint no solo permite detectar errores tempranamente, sino que también facilita la implementación de procesos de integración continua (CI) y entrega continua (CD).

### TIPOS DE PRUEBAS Y HERRAMIENTAS CLAVE

- Pruebas Unitarias: Validan unidades pequeñas del código, como funciones o componentes individuales. En Angular, se realizan típicamente con Karma como test runner y Jasmine como framework de aserciones. Estas pruebas son rápidas y pueden ejecutarse continuamente en cada commit.
- Pruebas de Integración: Evalúan la interacción entre componentes, servicios y módulos. También pueden ejecutarse con Karma, aunque pueden requerir mocks o spies para simular servicios de Firebase como Firestore o Authentication.
- Pruebas End-to-End (E2E): Validan el sistema desde el punto de vista del usuario, probando rutas, formularios, flujos completos y lógica de negocio. Herramientas como Cypress o Protractor permiten automatizar pruebas E2E simulando la interacción del usuario en un navegador real.

### AUTOMATIZACIÓN EN EL CICLO ÁGIL

En cada sprint, las historias de usuario deben incluir criterios de aceptación que se traduzcan en pruebas automatizadas. Integrar estas pruebas al pipeline de CI/CD (por ejemplo, usando GitHub Actions o GitLab CI) permite:

- Verificar automáticamente cada pull request.
- Desplegar solo si los tests pasan.
- Reducir el tiempo de pruebas manuales.
- Documentar el comportamiento esperado del sistema.

### BENEFICIOS CLAVE EN PROYECTOS ANGULAR + FIREBASE

1. Cobertura inmediata: Automatizar las pruebas permite validar desde componentes hasta autenticación y escritura/lectura en Firestore.
2. Reducción de errores regresivos: Cambios en un módulo no afectan negativamente otros.
3. Soporte a despliegues continuos: Facilita lanzamientos frecuentes con confianza.
4. Alineación con DevOps: Refuerza la cultura de calidad y colaboración entre equipos de desarrollo, QA y operaciones.

## 2.5 VERIFICACIÓN, VALIDACIÓN Y SU APLICACIÓN PRÁCTICA

### ¿QUÉ ES VERIFICACIÓN Y VALIDACIÓN (V&V)?

- Verificación: comprueba que el software se construye correctamente acorde al diseño y especificaciones (IEEE 1012-2012).
- Validación, sin embargo, certifica que el software construido es el correcto, el que el usuario precisa en realidad (IEEE 1012-2012).

### ¿POR QUÉ ES IMPORTANTE EN PRUEBAS AUTOMATIZADAS?

- Automatizar las pruebas permite ejecutar procesos de verificación en cada cambio (por ejemplo, pruebas unitarias en cada commit).
- Las pruebas end-to-end son formas de validación: se verifica el sistema desde la perspectiva del usuario.

### ¿CÓMO SE APLICA EN ANGULAR + FIREBASE?

- Verificación en Angular: Usar Jasmine/Karma para probar componentes y servicios. Ejemplo: comprobar que el servicio de autenticación retorna el usuario correcto.
- Verificación con Firebase: Pruebas de integración que simulan respuestas del backend. Ejemplo: simular Firestore con un mock y verificar lógica en Angular.
- Validación con E2E: Simular el flujo completo de un usuario (registro, login, agregar datos) usando Cypress o Protractor.

La verificación y validación, son procesos esenciales para garantizar la calidad del software, vistos desde múltiples ángulos. La verificación, para confirmar que el producto se construye bien, basándose en especificaciones técnicas, mientras que la validación se asegura de que se construye el producto CORRECTO, acorde con las necesidades del usuario. La Figura 2. 3 lo ilustra muy bien, mostrando como ambos procesos se combinan con diferentes tipos de pruebas.


Figura 2.3 Gráfico tipo V&V

# 2.6 PIRÁMIDE DE PRUEBAS EN ANGULAR CON FIREBASE

Según Cohn (2009), la pirámide de pruebas –modelo conceptual difundido también por Martin Fowler– recomienda estructurar los diferentes niveles de pruebas automatizadas priorizando aquellas más rápidas y menos costosas, como las pruebas unitarias. El principio fundamental es que, cuanto más abajo se esté en la pirámide, mayor debe ser la cantidad de pruebas, ya que estas ofrecen mayor estabilidad, menor tiempo de ejecución y mejor mantenimiento. La Figura 2.4 representa esta estructura piramidal aplicada al contexto de Angular.

## NIVELES DE LA PIRÁMIDE DE PRUEBAS

1. Base – Pruebas Unitarias
   - Son las más abundantes.
   - Evalúan funciones, métodos y componentes aislados.
   - En Angular se desarrollan con Jasmine (framework de pruebas) y Karma (test runner).
   - Simulan entradas y validan salidas sin depender de servicios externos.

2. Centro – Pruebas de Integración
   - Verifican que los distintos módulos trabajen correctamente juntos (componentes, servicios, pipes, rutas).
   - En Angular, se escriben también con Jasmine y Karma, pero requieren el uso de TestBed, HttpClientTestingModule o MockProviders para simular servicios reales como Firebase.

3. Cima – Pruebas End-to-End (E2E):
   - Simulan flujos de usuario reales: navegación, clics, formularios
   - Se realizan con herramientas como Cypress (moderna, rápida, interactiva) o Protractor (más antigua, ya deprecada).
   - Ejecutan pruebas en navegadores reales, lo que las hace más lentas, pero también las más cercanas a lo que vivirá el usuario.

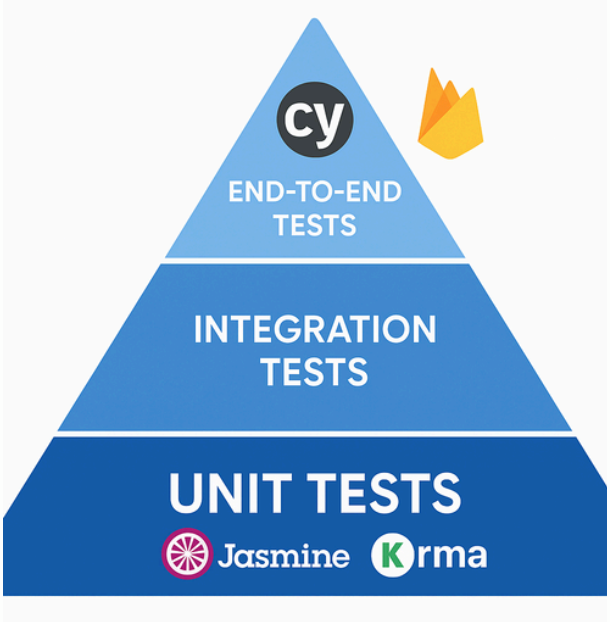



Figura 2.4 Pirámide de pruebas

### APLICACIÓN DE LA PIRÁMIDE EN PROYECTOS ANGULAR + FIREBASE

Implementar la pirámide de pruebas en este entorno tiene particularidades:

- Firebase opera con servicios asincrónicos, por lo que se requieren técnicas como mocks, spies o el uso del AngularFireTestingModule para simular Firestore o Auth en pruebas unitarias e integración.
- Las pruebas E2E deben validar que Firebase responda como se espera: por ejemplo, validar un login exitoso, o que los datos mostrados coincidan con Firestore.

### VENTAJAS DE SEGUIR ESTA ESTRUCTURA

- Eficiencia: se reduce el tiempo de ejecución al priorizar pruebas rápidas.
- Detección temprana de errores: se identifican fallos en funciones básicas antes de probar casos más complejos.
- Menor costo de mantenimiento: pruebas unitarias son más estables frente a cambios de interfaz o rutas.
- Escalabilidad: permite crecer el sistema con confianza al tener pruebas automatizadas bien distribuidas.

## 2.7 BUENAS PRÁCTICAS Y RECOMENDACIONES

Para construir una estrategia exitosa para pruebas automatizadas en aplicaciones Angular y Firebase, adoptar un buen conjunto de prácticas es clave. Esto refuerza la calidad del software y al mismo tiempo hace que el proyecto pueda crecer fácilmente. Con estas prácticas, la integración con procesos ágiles es sin problemas, encontrar errores temprano se vuelve más sencillo, el retrabajo disminuye y la implementación continua se agiliza, especialmente en entornos CI/CD.

La Figura 2 5 presenta un resumen de las prácticas más relevantes que se discutirán a continuación.

### ESCRIBIR PRUEBAS DESDE EL INICIO DEL DESARROLLO

La implementación temprana de pruebas beneficia el diseño de componentes más robustos y facilita la práctica de Test-Driven Development (TDD), metodología que promueve escribir las pruebas antes que el código de producción (Beck, 2002).

### MANTENER LAS PRUEBAS INDEPENDIENTES Y RÁPIDAS

Las pruebas deben ejecutarse de forma aislada para evitar que dependan de otros casos de prueba o del estado global del sistema. Esto garantiza resultados consistentes y reduce significativamente el tiempo de ejecución, algo clave especialmente en entornos de integración continua.

### SIMULAR SERVICIOS EXTERNOS CON MOCKS O FAKES

Dado que Firebase interactúa constantemente con servicios en la nube, es esencial reemplazar las llamadas reales a Firestore, Auth u otros módulos con versiones simuladas (mocked) durante la fase de prueba. Esto no solo evita la dependencia de la red, sino que mejora la estabilidad y velocidad de los tests.

### UTILIZAR HERRAMIENTAS DE COBERTURA DE CÓDIGO:

Implementar herramientas como karma-coverage o jest --coverage permite visualizar qué porciones del código aún no están siendo verificadas por pruebas. Esto ayuda a priorizar esfuerzos y evitar que funciones críticas queden sin validar.

### AUTOMATIZAR PRUEBAS EN PIPELINES DE CI/CD

La integración de las pruebas en cada paso del ciclo de desarrollo mediante pipelines automatizados (por ejemplo, con GitHub Actions, GitLab CI, o Jenkins) asegura que cada cambio en el código pase por un proceso de validación antes de ser integrado a la rama principal o desplegado a producción.

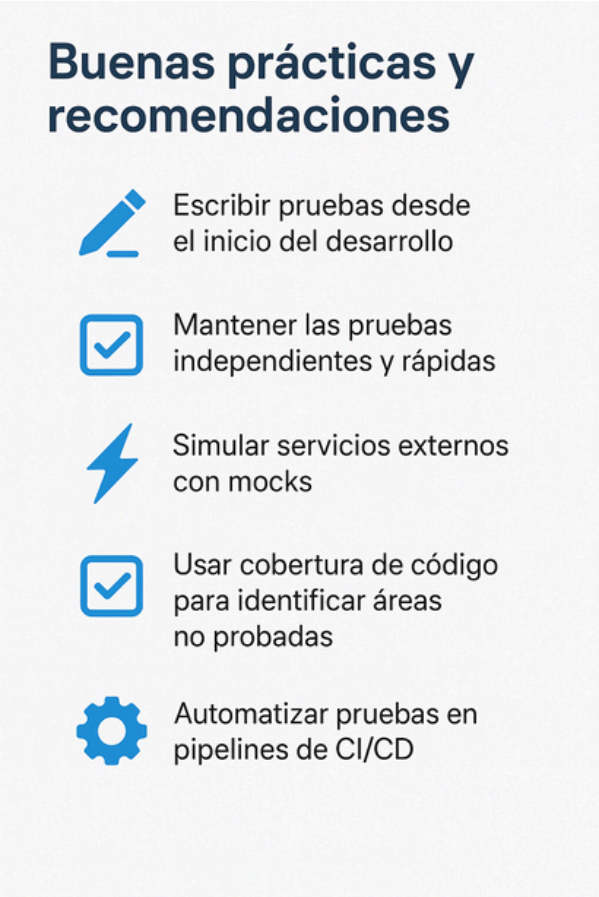



Figura 2.5 Lista de buenas prácticas

# CAPÍTULO 3

## DISEÑO E IMPLEMENTACIÓN DE LA ESTRATEGIA DE PRUEBAS

3.1 Descripción de la aplicación web desarrollada (Angular + Firebase)

3.2 Arquitectura del sistema

3.3 Diseño de la estrategia de pruebas automatizadas (unitarias, integración, E2E)

3.4 Herramientas seleccionadas (Jasmine, Karma, Protractor)

# 3.1 DESCRIPCIÓN DE LA APLICACIÓN WEB DESARROLLADA

La aplicación, que se construyó como un estudio de caso para esta tesis, es un sistema web implementado con Angular para el frontend y Firebase, una plataforma BaaS, para el backend. La elección de esta combinación tecnológica se debió a su renombre en el desarrollo web actual, una arquitectura que puede escalar, lo sencillo de integrarla y el apoyo que ofrece para prácticas de desarrollo ágil y automatización de pruebas.

La aplicación está orientada a un sistema escolar. Cuenta con las siguientes características principales:

- Interfaz desarrollada en Angular v13+ con arquitectura modular y basada en componentes.
- Autenticación de usuarios mediante Firebase Authentication.
- Persistencia de datos utilizando Cloud Firestore, una base de datos NoSQL en tiempo real.
- Almacenamiento de archivos (opcional) mediante Firebase Storage.
- Navegación basada en rutas protegidas por guards (AuthGuard) según el estado de autenticación del usuario.
-

El desarrollo, un proceso llevado a cabo con las mejores prácticas en separación de responsabilidades, utilizó servicios Angular para gestionar la lógica y la interacción con Firebase. Los componentes, su principal función, se centran únicamente en la presentación y en la interacción directa con el usuario.

Las tecnologías concretas empleadas en este proyecto, son detalladas en la Tabla 3. 1, donde se ve que se usó Angular v13+ para el frontend, Firebase v9+ para el backend, además de herramientas como Jasmine y Karma para pruebas unitarias, también Protractor o Cypress para pruebas end-to-end.

### TECNOLOGÍAS UTILIZADAS:

| Herramienta / Tecnología | Versión / Uso |
|---|---|
| Angular | v13+ – Framework frontend |
| Firebase | v9+ – Plataforma backend (Auth, DB) |
| AngularFire | Librería oficial para integrar Firebase con Angular |
| Jasmine + Karma | Pruebas unitarias e integración |
| Protractor / Cypress | Pruebas end-to-end |
| Git | Control de versiones |

Tabla 3.1 Tabla de tecnologías usadas

## 3.2 ARQUITECTURA DEL SISTEMA

La aplicación que creada, emplea una arquitectura modular, y totalmente separada, tal como es típico en el framework Angular, y se conecta con los servicios de Firebase, que brindan las funcionalidades de backend. Se diseñó esta arquitectura, pensando en la escalabilidad del sistema, la tarea de mantener el código y en la forma de hacer pruebas automáticas eficientes a varios niveles.

### ESTRUCTURA GENERAL DE LA ARQUITECTURA

La solución se divide en tres capas principales:

1. Capa de presentación (Frontend):

   - Implementada completamente con Angular, utilizando componentes reutilizables, rutas protegidas y formularios reactivos.
   - Gestiona la lógica que rige la interacción con el usuario, al igual que, la validación de los datos. También, supervisa la navegación, entre las distintas vistas.
   - Aplica principios de separación de responsabilidades mediante el uso de servicios inyectables para la lógica de negocio.

2. Capa de servicios (Business Logic):

- Implementada con servicios Angular personalizados que abstraen la comunicación con Firebase.
- Se encarga de autenticar usuarios, gestionar datos desde Firestore y controlar accesos a almacenamiento.
- Facilita el uso de mocks en las pruebas unitarias y de integración.

3. Capa de backend (Firebase):

- Firebase Authentication es la herramienta, para administrar sesiones y permisos de usuarios
- Cloud Firestore para el almacenamiento y consulta de datos en tiempo real.
- Firebase Storage para subir y recuperar archivos.
- Toda la lógica de backend está delegada en los servicios de Firebase, eliminando la necesidad de construir APIs REST personalizadas.

Esta estructura en tres capas, tal como se muestra en la Figura 3.2, posibilita una distinta separación de responsabilidades, el frontend se encarga de mostrar todo, los servicios administran la lógica del negocio, y Firebase ofrece todas las funciones del backend.


Figura 3.1 Diagrama visual de arquitectura

# 3.3 DISEÑO DE LA ESTRATEGIA DE PRUEBAS AUTOMATIZADAS

Para garantizar la calidad del sistema durante el desarrollo, se diseñó una estrategia de pruebas basada en el modelo de la pirámide de pruebas. Esta estrategia incluye la automatización de pruebas en tres niveles:

Para asegurar la calidad del sistema en construcción, pensamos en una estrategia de pruebas usando la pirámide de pruebas como guía. En esta estrategia, se automatizó las pruebas en tres niveles diferentes:

1. **Pruebas unitarias:** Son para comprobar la lógica de cada componente y servicio de Angular, poniendo ojo en funciones puras, validaciones de formularios y métodos de servicios específicos.
2. **Pruebas de integración:** Su propósito, validar la interacción entre componentes y servicios. También, se verifican las operaciones dentro de Firebase. Estas incluyen la lectura y escritura en Firestore, y cómo reaccionan a los eventos de autenticación, asegurando el correcto funcionamiento.
3. **Pruebas end-to-end (E2E):** Para validar el flujo total del usuario, como el entrar al sistema, moverse entre páginas protegidas y ver datos actualizados en tiempo real.

Las pruebas de cada tipo se meten poco a poco en el flujo ágil, ofreciendo feedback instantáneo luego de cada modificación o nueva funcionalidad.
Se usó un acercamiento incremental y organizado, iniciando con pruebas unitarias mientras se construían los componentes y servicios, luego con las pruebas de integración, hasta automatizar flujos de usuarios completos mediante las pruebas E2E.

### EJEMPLO DE PRUEBA UNITARIA: LOGINCOMPONENT

Para validar el correcto funcionamiento del formulario de autenticación, se desarrolló una suite de pruebas unitarias utilizando Jasmine y ejecutadas con Karma. Las pruebas cubren los siguientes casos:

- Validación de formulario vacío
- Validación de formulario con datos completos
- Intento de inicio de sesión exitoso
- Manejo de fallo en autenticación
- Marcado de campos como tocados cuando hay error

A continuación, se muestra un fragmento del código de estas pruebas (ver Figura 3.2). La ejecución de estas pruebas en Karma arrojó resultados satisfactorios, como se observa en la Figura 3.3, donde todas las pruebas pasaron exitosamente. Adicionalmente, la consola del navegador (Figura 3.4) y los casos exitosos (Figura 3.5 y 3.6) confirman el correcto funcionamiento del componente bajo diferentes escenarios.

```
beforeEach(() => {
    fixture = TestBed.createComponent(LoginComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
});

it('debería tener el formulario inválido cuando los campos están vacíos', () => {
    if (component.loginForm.valid) {
        console.log('❌ Error: El formulario no debería ser válido cuando los campos están vacíos');
    } else {
        console.log('✅ El formulario es inválido cuando los campos están vacíos');
    }
    expect(component.loginForm.valid).toBeFalsy();
});

it('debería tener el formulario válido cuando los campos están llenos', () => {
    component.loginForm.controls['usuario'].setValue('testUser');
    component.loginForm.controls['contrasena'].setValue('testPassword');
    if (!component.loginForm.valid) {
        console.log('❌ Error: El formulario debería ser válido cuando los campos están llenos');
    } else {
        console.log('✅ El formulario si valida si los campos estan llenos');
    }
    expect(component.loginForm.valid).toBeTruthy();
});
```

Figura 3.2 Codigo del test LoginComponent

## RESULTADOS DE LAS PRUEBAS


Figura 3.3 Resultados de las pruebas en Jasmine + Karma

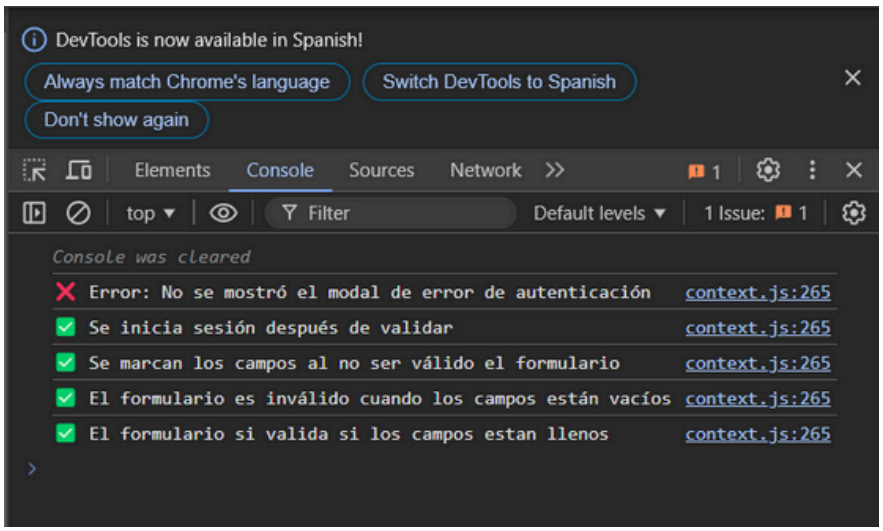



Figura 3.4 Consola del navegador: con errores

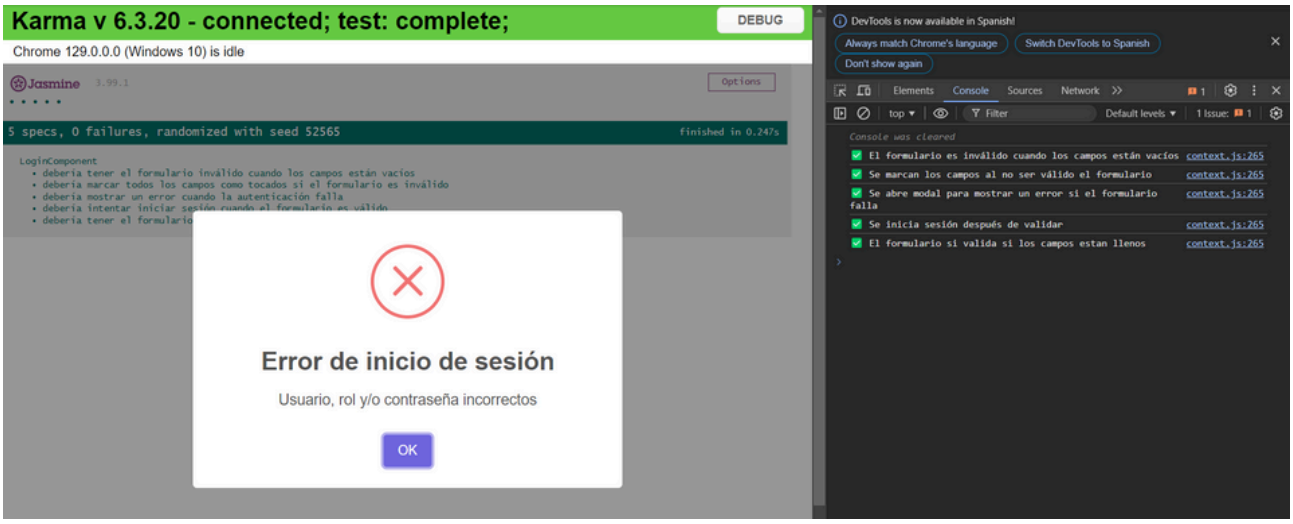



Figura 3.5 Modal de error de inicio de sesión

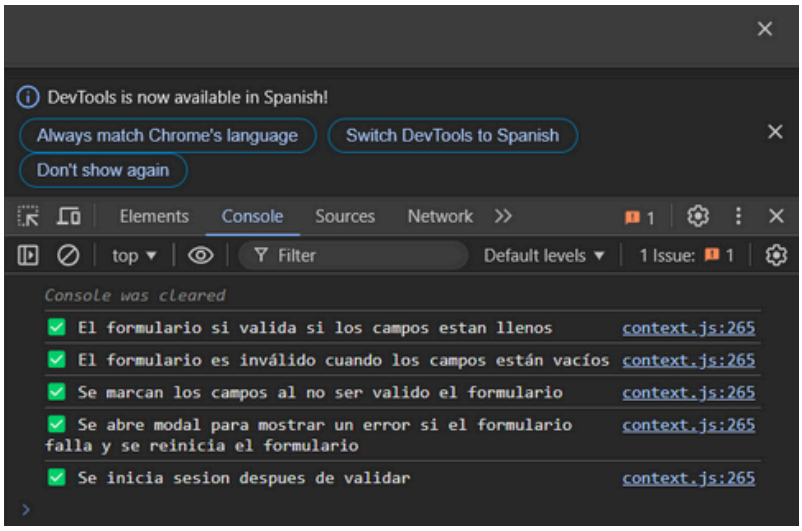



Figura 3.6 Consola del navegador: casos exitosos

La automatización de pruebas se puso primero en componentes claves, por ejemplo, el módulo de inicio de sesión, que, a fin de cuentas, es la puerta de entrada y validación principal del sistema, los principales objetivos de prueba se centraron en la lógica de formularios, validación de datos, y flujo de autenticación, dada su impacto directo en la experiencia del usuario y la seguridad de la aplicación.

Aplicar la estrategia de pruebas en etapas ayudó a encontrar errores temprano, validar cosas antes de la integración total, y minimizar regresiones futuras, sin embargo, implementarlo presentó desafíos técnicos serios, como simular servicios asincrónicos de Firebase, manejar rutas protegidas por guards, y configurar el entorno sin una conexión en tiempo real.

Así, se recomienda, en el futuro, usar Firebase Emulator Suite, para facilitar un entorno local más estable y seguro en las pruebas.

# 3.4 HERRAMIENTAS SELECCIONADAS

- La selección de herramientas se basó en su compatibilidad con el ecosistema Angular y su capacidad para integrar pruebas de forma sencilla y automatizada dentro del flujo de trabajo ágil y CI/CD.

| Herramienta | Tipo de prueba | Justificación |
|---|---|---|
| **Jasmine** | Unitarias / Integración | Framework robusto, nativo en Angular CLI, sintaxis amigable |
| **Karma** | Unitarias / Integración | Test runner integrado con Angular CLI, permite ejecutar pruebas en distintos navegadores |
| **Protractor** | End-to-End (E2E) | Herramienta oficial de Angular (aunque deprecada), simula flujos de usuario |
| **AngularFire** | Integración con Firebase | Librería oficial para comunicación entre Angular y Firebase |
| **Firebase Emulator Suite** | Local / Produccion pruebas E2E | Permite ejecutar pruebas sin afectar datos reales y en produccion para corroborar efectividad. |

Tabla 3.2 Herramientas seleccionadas

Aunque este proyecto utilizó Protractor, herramientas modernas como Cypress ofrecen mejor rendimiento y estabilidad (Cypress, 2024).
Estas herramientas permitieron cubrir los distintos niveles de pruebas automatizadas necesarios, integrarse fácilmente con el proyecto y ejecutar las pruebas tanto localmente como en posibles entornos de integración continua.

# CAPÍTULO 4


## DESARROLLO DE LAS PRUEBAS AUTOMATIZADAS

4.1 Configuración de entorno de pruebas

- 4.1.1 Configuración para pruebas unitarias e integración
- 4.1.2 Configuración para pruebas end-to-end (E2E)
- 4.1.3 Consideraciones sobre firebase

4.2 Implementación de pruebas por tipo

- 4.2.1 Pruebas unitarias
- 4.2.2 Pruebas de integración
- 4.2.3 Pruebas end-to-end (E2E)

4.3 Validación de casos especiales

- 4.3.1 Control de acceso basado en roles
- 4.3.2 Resiliencia del sistema ante fallos

4.4 Evidencia visual del sistema

- 4.4.1 Flujo de importación masiva de alumnos
- 4.4.2 Flujo de registro individual de alumnos
- 4.4.3 Flujo de búsqueda y filtrado dinámico
- 4.4.4 Flujo de exportación y eliminación

4.5 Análisis comparativo de resultados

# 4.1 CONFIGURACIÓN DE ENTORNO DE PRUEBAS

Configurar apropiadamente el ambiente de pruebas es crucial para garantizar la calidad del software desde el mismo inicio del proceso de desarrollo. Para esta aplicación desarrollada con Angular y Firebase, se estableció un entorno, que facilita la ejecución controlada, reproducible y eficiente de pruebas automatizadas.

## 4.1.1 CONFIGURACIÓN PARA PRUEBAS UNITARIAS E INTEGRACIÓN.

Para las pruebas unitarias y de integración, se emplearon las propias herramientas del ecosistema Angular, específicamente, Jasmine, fungiendo como marco de pruebas, y Karma, como corredor de pruebas. Tales pruebas se llevan a cabo en navegadores como lo es Chrome, o en modo headless , para aquellos ambientes de integración continua. La configuración principal se encuentra en el archivo karma.conf.js, el cual define las dependencias necesarias, los reportes de cobertura y el navegador predeterminado para ejecutar las pruebas (Karma, 2023).

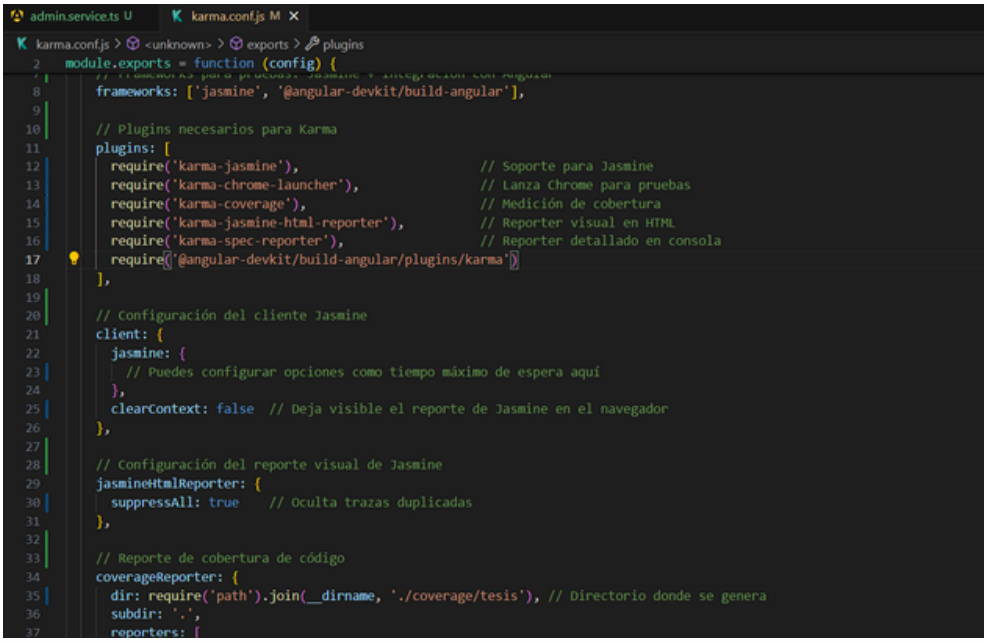



Figura 4.1: Archivo karma.conf.js configurado para pruebas unitarias e integración.

Adicionalmente, el archivo angular.json contiene la sección "test" que especifica cómo se deben compilar y ejecutar las pruebas dentro del contexto del proyecto Angular, incluyendo la configuración de código de cobertura y optimizaciones.

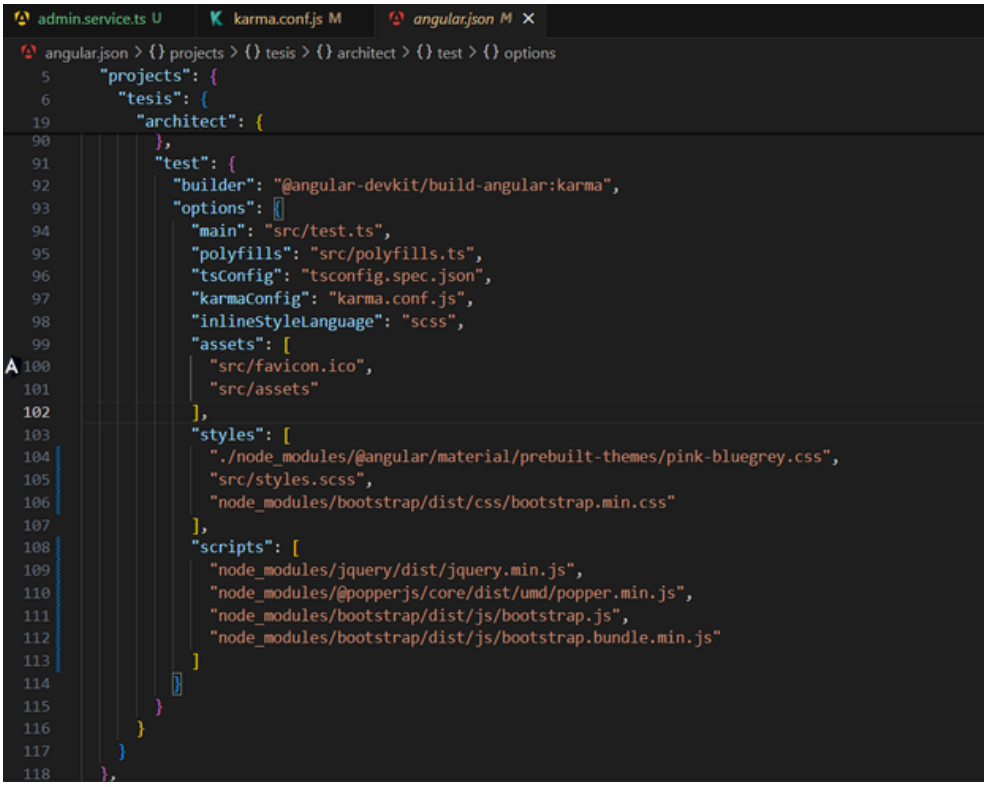



Figura 4.2: Fragmento del archivo angular.json mostrando la configuración de test.

### 4.1.2 CONFIGURACIÓN PARA PRUEBAS END-TO-END (E2E)

Para validar flujos completos del usuario se empleó Protractor una herramienta pensada particularmente para aplicaciones Angular. Protractor permite simular interacciones genuinas como clics el ingreso de datos, y navegación entre rutas.

El archivo protractor.conf.js centraliza la configuración, aquí se especifican el navegador a utilizar, los ficheros de especificaciones e2e-spec ts, y el framework de aserciones Jasmine. De hecho, Protractor ejecuta las pruebas dentro de un navegador real, con lo cual verifica exhaus- tivamente la operatividad del sistema.


Figura 4.3. Código de login.e2e-spec.ts con explicación

### 4.1.3 CONSIDERACIONES SOBRE FIREBASE

Aunque en esta tesis se utilizó Firebase en modo producción para las pruebas, se recomienda para implementaciones futuras el uso de Firebase Emulator Suite, que permite simular Auth, Firestore y Storage de forma local (Firebase Documentation, 2024). Esto facilita pruebas más rápidas, repetibles y sin dependencia de la red.

## 4.2 IMPLEMENTACIÓN DE PRUEBAS POR TIPO

### 4.2.1 PRUEBAS UNITARIAS

Las pruebas unitarias validan componentes y servicios de forma aislada utilizando Jasmine (2023) como framework de especificaciones y Karma como test runner.En este proyecto se desarrollaron utilizando Jasmine y Karma, enfocándose en tres áreas principales:

1. Validación de formularios reactivos: Se verificó que los formularios sean inválidos cuando están incompletos y válidos solo cuando todos los campos requeridos están correctamente llenados.
2. Lógica de filtrado de datos: Se probó que las funciones de búsqueda filtren correctamente las listas según los criterios ingresados por el usuario.
3. Métodos de servicios: Se validaron las operaciones CRUD y la comunicación con Firebase mediante mocks.

```
it('debería tener el formulario inválido cuando los campos están vacíos', () => {
    if (component.loginForm.valid) {
        console.log('❌ Error: El formulario no debería ser válido cuando los campos están vacíos');
    } else {
        console.log('✅ El formulario es inválido cuando los campos están vacíos');
    }
    expect(component.loginForm.valid).toBeFalsy();
});
```

Figura 4.4 Prueba en formulario de Login

```
it('debería registrar alumno si el formulario es válido', fakeAsync(() => {
  spyOn(Swal, 'fire');
  component.formulario.setValue({
    usuario: '123456',
    nombre: 'Juan',
    ap: 'Pérez',
    am: 'López',
    tel: '5555555555',
    correo: 'juan@example.com',
    direccion: 'Calle Falsa 123',
    curp: 'PEPJ800101HDFLLL01',
    fecha: new Date(),
    semestre: 1,
    grupo: 'A'
  });
```

Figura 4.5 Prueba en formulario de registro de alumnos

Resultados de ejecución: La ejecución de las pruebas unitarias arrojó resultados positivos en la mayoría de los componentes. Por ejemplo, en el AlumnoComponent se validaron exitosamente funcionalidades como exportación a Excel, filtrado por nombre y cálculo de semestres según la fecha actual.

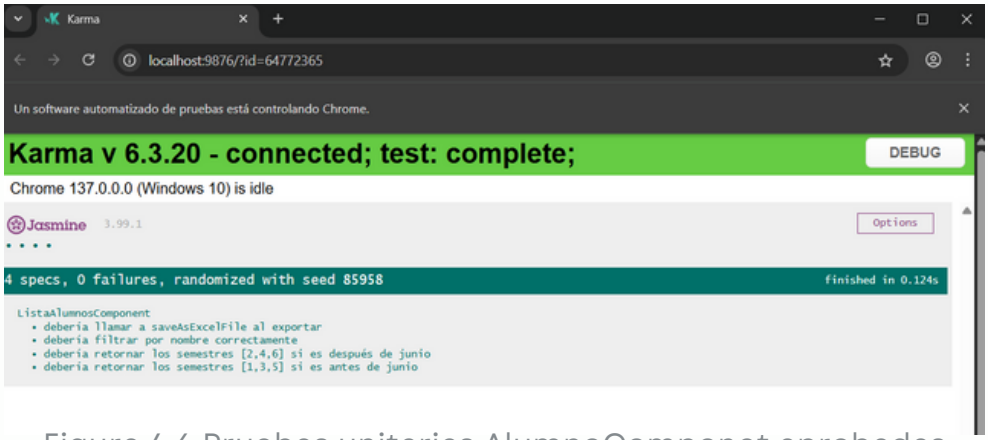



Figura 4.6 Pruebas unitarias AlumnoComponet aprobadas

Sin embargo, también se implementaron pruebas negativas intencionalmente para verificar que el sistema detecte correctamente casos no válidos. La siguiente figura muestra una prueba diseñada para fallar cuando no se encuentran coincidencias en el filtro:

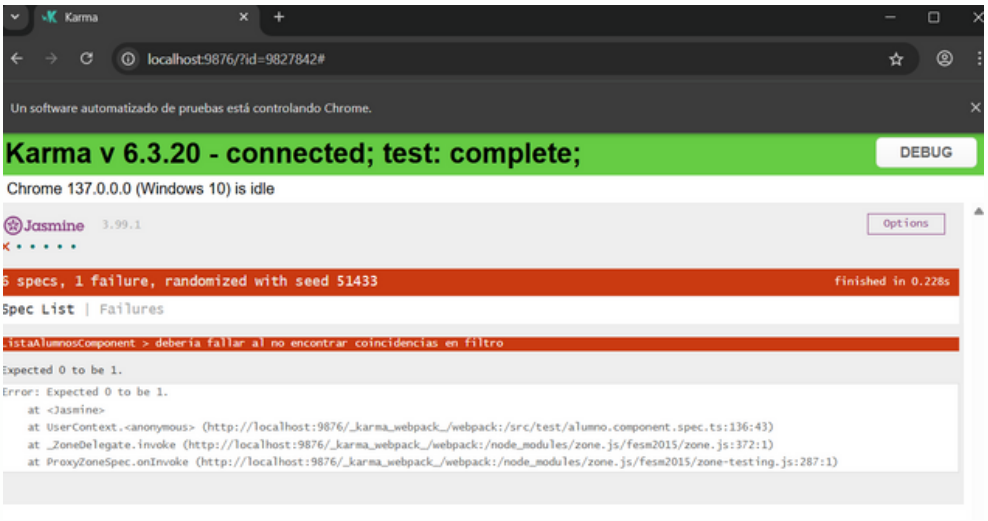



Figura 4.7 Prueba unitaria fallida con análisis

Análisis del fallo: Esta prueba fue diseñada para validar el comportamiento del filtro cuando no existen coincidencias. El error detectado (Expected 0 to be 1) indica que el filtro retornó un resultado cuando no debería haberlo hecho. La desviación observada de un valor predicho, que fue de cero, con un valor real siendo uno, esto sugirió que el filtro era proporcionando una salida inadecuada. Mostrando, a su vez, una confrontación defectuosa al interior del algoritmo de búsqueda. Dichas evaluaciones negativas considero primordiales resultan esenciales, a fin de reforzar la solidez del sistema.

Métricas de cobertura
La Tabla 4.1 resume los resultados obtenidos en cada componente evaluado:

| Componente | Líneas | Funciones | Ramas | Instrucciones | Total pruebas |
|---|---|---|---|---|---|
| AddAlumnoComponent | 74,57% | 46,15% | 68,18% | 74,62% | 22 |
| LoginComponent | 48,05% | 35,00% | 70,96% | 49,36% | 13 |
| ListaAlumnosComponer | 30,20% | 36,58% | 12,50% | 32,48% | 10 |

Tabla 4.1 - Resumen de cobertura por componente

Como se observa, AddAlumnoComponent presenta la mejor cobertura general, mientras que ListaAlumnosComponent requiere atención prioritaria para incrementar su validación, especialmente en ramas condicionales.

### 4.2.2 PRUEBAS DE INTEGRACIÓN

Las pruebas de integración verificaron la comunicación entre componentes, servicios y Firebase. A diferencia de las unitarias, estas pruebas requirieron:

- Configuración de TestBed con dependencias reales o simuladas
- Simulación de servicios Firebase mediante mocks y spies
- Validación de flujos asincrónicos completos (Promises, Observables)

Casos validados
Se probaron escenarios como:

1. Carga de datos desde Firestore: Es crucial examinar que la información se extraiga de manera precisa y se exhiba de forma apropiada en la interfaz.
2. Actualización de vistas reactivas: Se debe constatar que las modificaciones realizadas a los datos compartidos se propaguen automáticamente en varios componentes, un proceso importantisimo.
3. Autenticación y navegación: El enfoque es verificar que el proceso de inicio de sesión modifique correctamente el estado de autentación y redireccione al usuario.

Ejemplo representativo: La carga de alumnos desde Firestore hasta su visualización en tabla, incluyendo la aplicación de filtros por nombre, grupo o semestre. Esta prueba validó la cadena completa: servicio → componente → vista.
La fase de integración expuso fallos elusivos, indetectados previamente en las pruebas unitarias. Surgieron así imprevistos de sincronización entre módulos y reacciones inesperadas al fusionar múltiples filtros, una combinacion que trajo problemas.

### 4.2.3 PRUEBAS END-TO-END (E2E)

Las pruebas E2E simulan el comportamiento real del usuario final, validando flujos completos desde la interfaz hasta la base de datos. Se utilizó Protractor para automatizar estas interacciones en un navegador real.

Funcionalidades validadas
La Tabla 4.2 resume los flujos críticos probados:

| Funcionalidad probada | Herramienta | Validación clave |
|---|---|---|
| Login con credenciales válidas | Protractor | Redirección a /alumnos |
| Login con credenciales inválidas | Protractor | Modal de error visible |
| Filtrado por nombre | Protractor | Lista actualizada dinámicamente |
| Agregar alumno individual | Protractor | Modal abierto + datos guardados |
| Importar alumnos desde Excel | Protractor | Confirmación visual + datos en tabla |
| Exportar lista a Excel | Protractor | Archivo descargado correctamente |
| Eliminar alumno | Protractor | Modal de confirmación + eliminación exitosa |

Tabla 4.2 - Funcionalidades E2E validadas

Proceso de ejecución

Las pruebas E2E se ejecutaron mediante el comando npx protractor protractor.conf.js. El proceso incluyó:

1. Inicialización del WebDriver y apertura del navegador
2. Navegación automática a las rutas especificadas
3. Interacción con elementos del DOM (clics, inputs, selects)
4. Validación de resultados esperados (redirecciones, modales, datos)
5. Generación de reporte con resultados

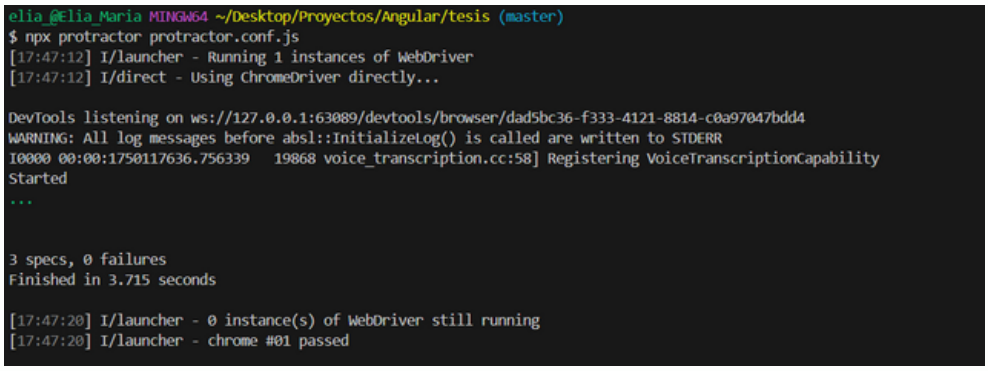



Figura 4.8 Consola ejecutando npx protractor.

Resultados y análisis

La ejecución inicial reveló varios fallos, principalmente relacionados con:

- Tiempos de espera insuficientes para elementos asincrónicos
- Selectores incorrectos que no coincidían con el DOM renderizado
- Modales no detectados por falta de configuración waitForAngularEnabled(false)

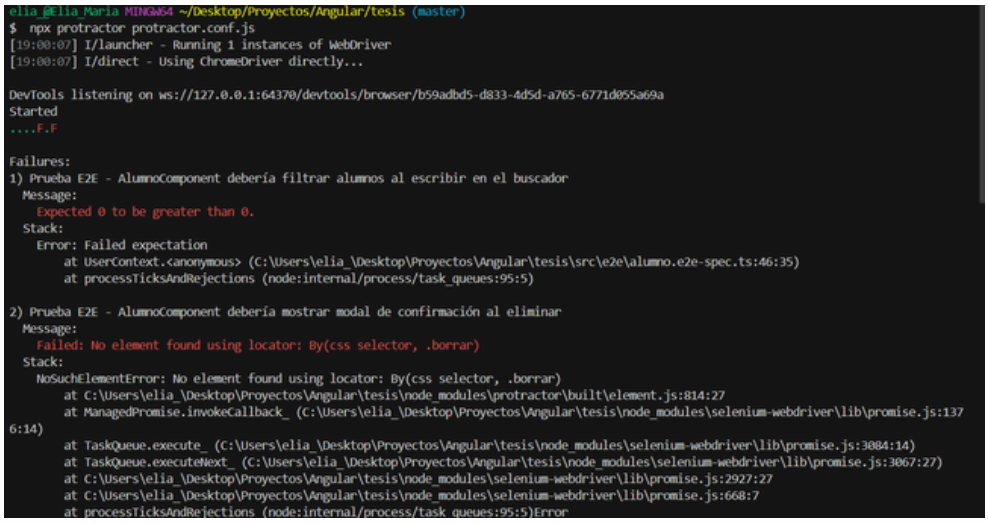



Figura 4.9 Resultados E2E con errores detectados

Tras los ajustes necesarios (incremento de timeouts, corrección de selectores y manejo adecuado de elementos externos como SweetAlert2), las pruebas se ejecutaron exitosamente

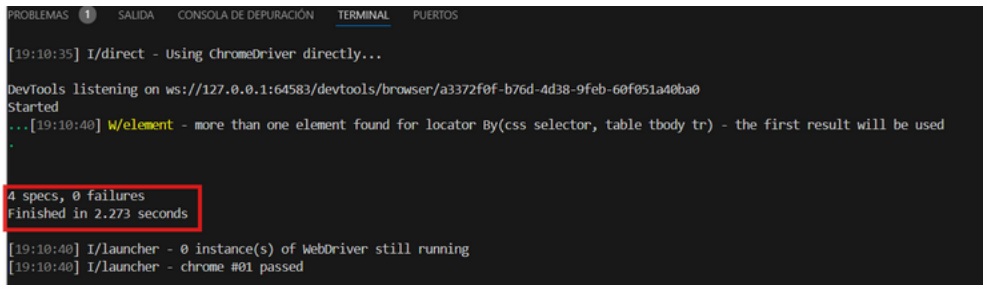



Figura 4.10 - Resultados E2E exitosos (primera parte)

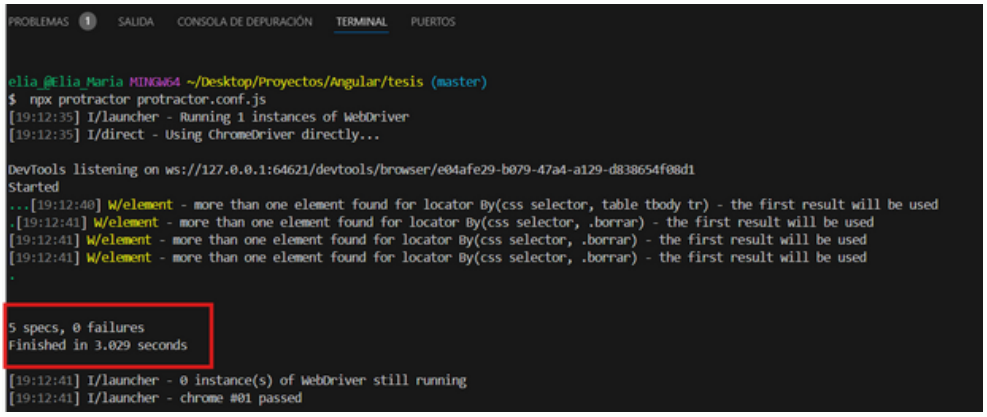



Figura 4.11 - Resultados E2E exitosos (segunda parte)

Métricas de ejecución
La Tabla 4.3 compara el desempeño de los diferentes tipos de pruebas implementadas:

| Tipo de prueba | Tiempo promedio | Pruebas totales | Fallos detectados iniciales |
|---|---|---|---|
| Unitarias | 0.5s por spec | 45 | 3 |
| Integración | 2s por spec | 15 | 2 |
| E2E | 8s por spec | 12 | 5 |

Tabla 4.3 - Comparativa de tipos de pruebas

Como se observa, las pruebas E2E son las más lentas pero también las más efectivas para detectar errores de integración completa y problemas de experiencia de usuario.

# 4.3 VALIDACIÓN DE CASOS ESPECIALES

Adicionalmente a los flujos principales, se condujeron evaluaciones para situaciones complejas las cuáles aseguran la solidez del sistema bajo condiciones operativas verdaderas.

### 4.3.1 CONTROL DE ACCESO BASADO EN ROLES

Se implementaron pruebas E2E para validar que el sistema de autenticación y autorización funcione correctamente:

- Usuarios no autenticados: Se validó que los usuarios no autenticados sean redirigidos automáticamente al login al intentar acceder a rutas protegidas mediante manipulación directa de URLs. El redireccionamiento al portal de acceso, cumpliendo con lo estipulado, se realizó de forma automática, sin intervenciones.
- Usuarios con privilegios limitados: Se constató que el acceso a las funciones reservadas para "administradores" permanecía obstaculizado para usuarios designados como "alumnos", confirmando las políticas de acceso. El sistema emitió un mensaje de "acceso denegado", evidenciando la restricción de manera explícita.
- Menús dinámicos: Se comprobó que el menú de navegación muestre u oculte opciones según el rol del usuario autenticado, validando que los elementos HTML correspondientes aparezcan o desaparezcan correctamente del DOM.

### 4.3.2 RESILIENCIA DEL SISTEMA ANTE FALLOS

Se simularon condiciones adversas para verificar la capacidad del sistema de manejar errores externos:

- Fallo transitorio en Firestore: Con el fin de evaluar la presentación adecuada de mensajes de error y las reintentos automatizados por parte de la app, se indujo artificialmente una desconexión de la base de datos.
- Retraso intencional en la autenticación: Ademas, se añadió un retraso en el servicio de autenticación. Esto fue para verificar el manejo apropiado de estados de carga en la interfaz y, igualmente, prevenir clics múltiples y simultáneos.

- Validación de datos inconsistentes: A fin de garantizar la robustez del sistema y la preservación de la integridad de la información, se sometieron a prueba diversos escenarios que incluían datos ausentes o incorrectamente formateados.

Estas pruebas demostraron que la aplicación es capaz de degradar graciosamente ante fallos externos, manteniendo la experiencia del usuario y evitando pérdida de datos

# 4.4 EVIDENCIA VISUAL DEL SISTEMA

Esta sección presenta las capturas de pantalla que documentan visualmente el funcionamiento de las principales funcionalidades del sistema durante las pruebas E2E.

## 4.4.1 FLUJO DE IMPORTACIÓN MASIVA DE ALUMNOS

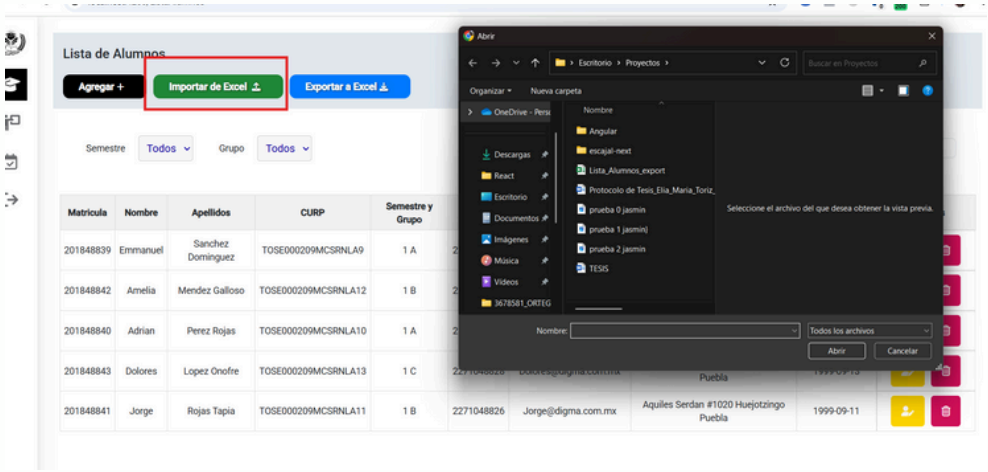



Figura 4.12 Interfaz grafica de la prueba de importar alumnos

La interfaz permite seleccionar o arrastrar un archivo Excel con datos de múltiples alumnos, reduciendo significativamente el tiempo de carga manual.

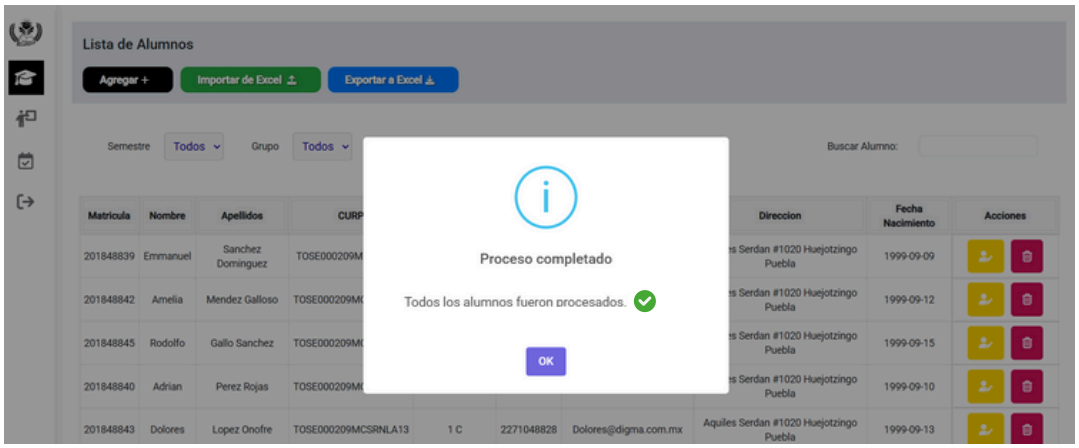



Figura 4.13.Modal de todos los alumnos cargados correctamente

El sistema muestra un modal de SweetAlert2 confirmando que todos los registros fueron procesados correctamente.

## 4.4.2 FLUJO DE REGISTRO INDIVIDUAL DE ALUMNOS

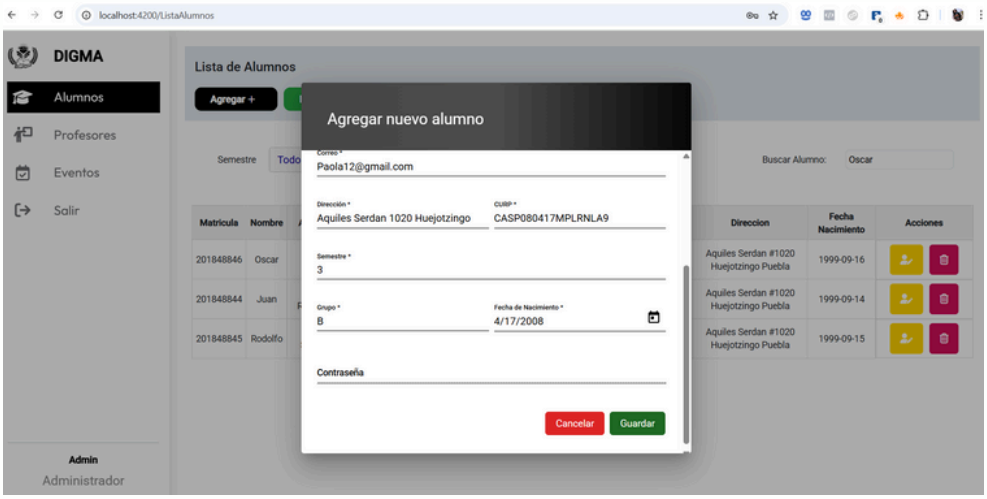



Figura 4.14 Interfaz gráfica de agregar alumno uno por uno

Formulario con validaciones en tiempo real para capturar datos como nombre, matrícula, grupo y semestre.

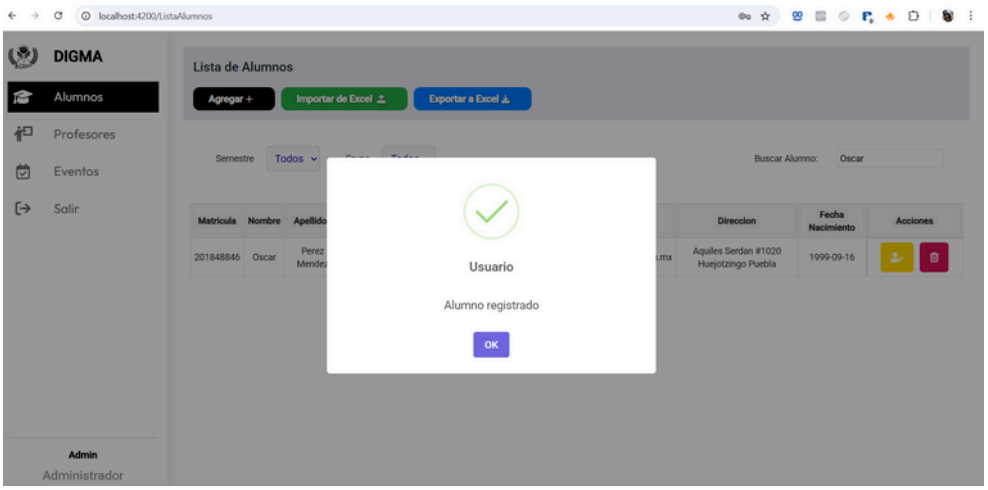



Figura 4.15 Modal de usuario agregado exitosamente

Confirmación visual que previene duplicidades y asegura al usuario que el registro se completó.

### 4.4.3 FLUJO DE BÚSQUEDA Y FILTRADO DINÁMICO


Figura 4.16 Filtro por nombre de alumno

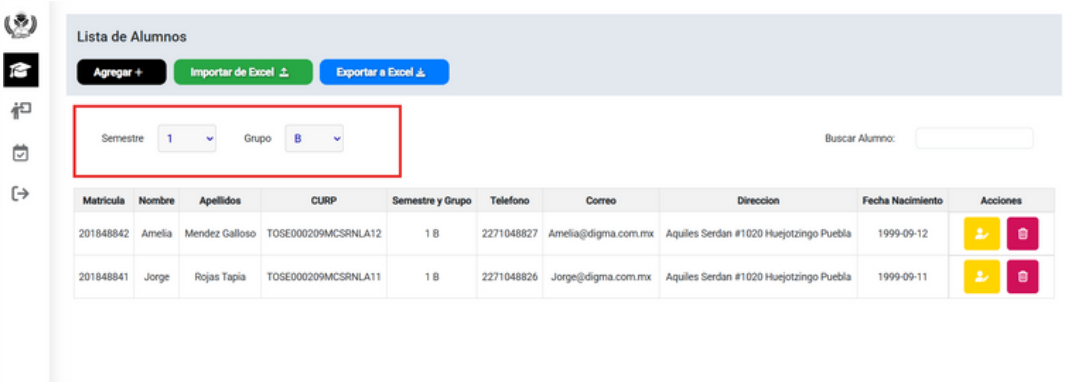



Figura 4.17 Filtro por semestre y grupo

### 4.4.4 FLUJO DE EXPORTACIÓN Y ELIMINACIÓN

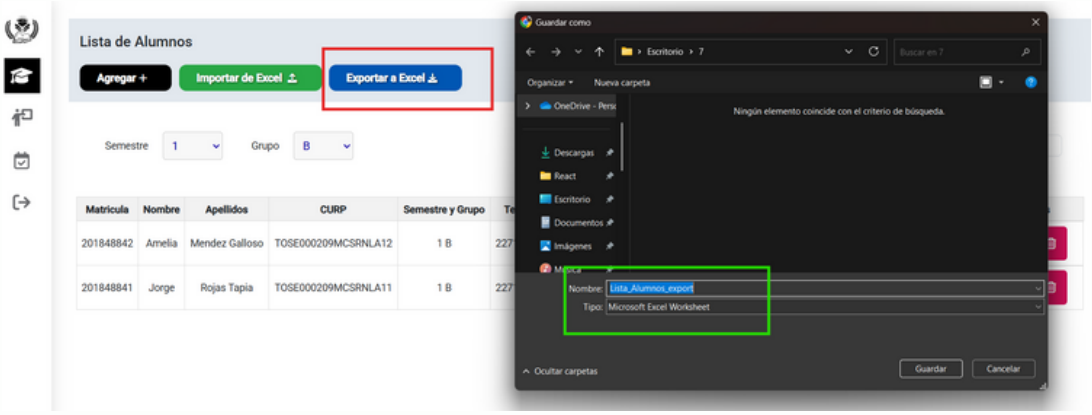



Figura 4.18 Botón de exportar a Excel

Funcionalidad que permite respaldar o compartir la información de los alumnos en formato estándar.

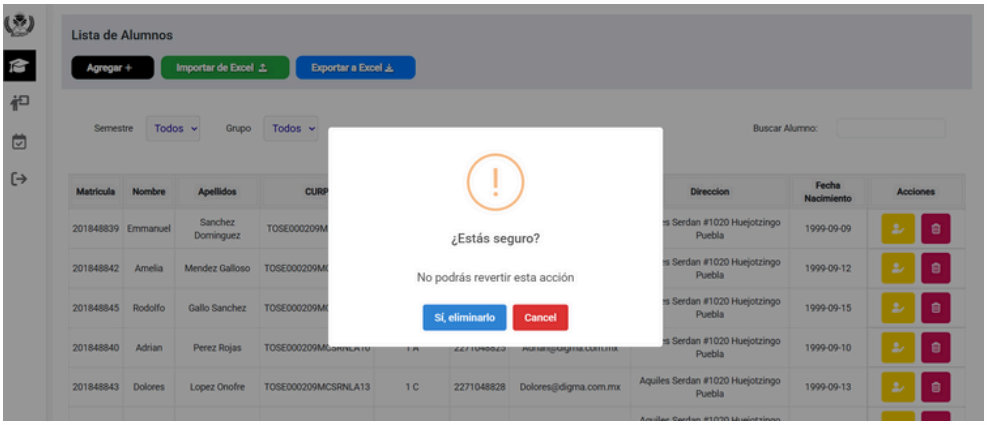



Figura 4.19 Modal de confirmación antes de eliminar

Protección contra eliminaciones accidentales mediante confirmación explícita del usuario.

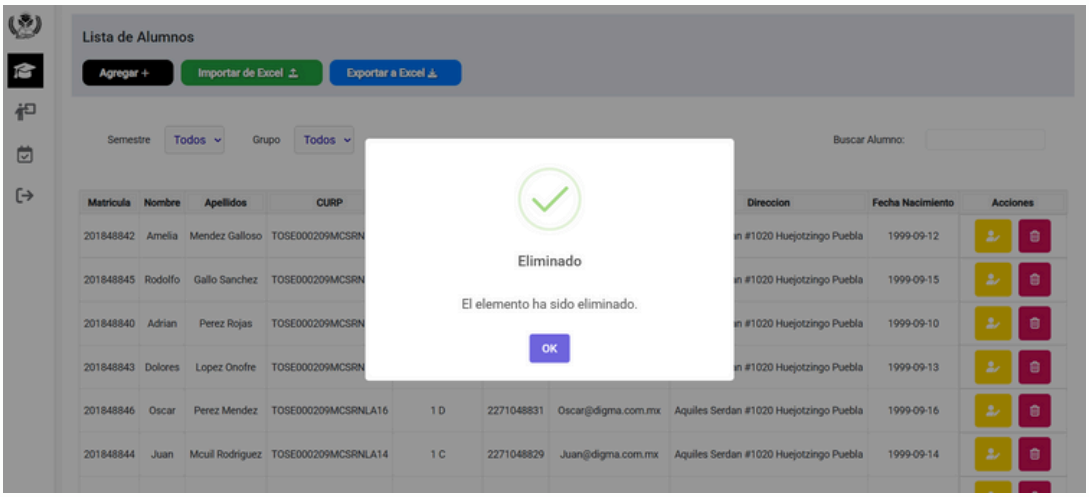



Figura 4.20 Modal de alumno eliminado

Confirmación final que asegura la integridad del proceso y da certeza sobre la acción realizada.

# 4.5 ANÁLISIS COMPARATIVO DE RESULTADOS

La implementación de pruebas en los tres niveles permitió evaluar sistemáticamente el comportamiento de la aplicación. La Tabla 4.4 presenta una evaluación de aspectos clave durante la ejecución:

| Criterio evaluado | Pruebas Unitarias | Pruebas de Integración | Pruebas E2E |
|---|---|---|---|
| Cobertura de código | Alta | Media | Baja |
| Simulación de interacción real | Baja | Media | Alta |
| Complejidad de implementación | Baja | Media | Alta |
| Tiempo de ejecución | Rápido | Moderado | Lento |
| Detecta errores visuales/UI | No | Parcial | Sí |
| Dependencia de componentes | No | Sí | Sí |
| Validación de flujos completos | No | Parcial | Sí |
| Facilidad de mantenimiento | Alta | Media | Baja |

Tabla 4.4 - Evaluación de aspectos clave por tipo de prueba

Conclusiones del análisis

1. Complementariedad: Distintos tipos de pruebas examinan facetas divergentes del sistema. Mientras las pruebas unitarias avalan la calidad del código fundacional, las de integración verifican la interacción entre módulos, en cambio las E2E validan la experiencia íntegra del usuario.
2. Detección temprana de errores: Por su velocidad de ejecución, las pruebas unitarias posibilitaron la pronta detección de fallos lógicos a lo largo del desarrollo.
3. Validación realista: Únicamente las pruebas E2E desvelaron problemáticas de UX, tal es el caso de tiempos de espera inadecuados en modales o comportamientos imprevistos en interacciones complejas.
4. Inversión de tiempo justificada: A pesar de que las pruebas E2E exijan más tiempo para configuración y ejecución, su valor yace en asegurar el correcto funcionamiento del sistema integralmente, tal como el usuario final lo vivirá.

Esta estrategia combinada de pruebas automatizadas resultó en una aplicación más confiable, con menor cantidad de defectos en producción y mayor confianza del equipo de desarrollo para implementar nuevas funcionalidades.

# CAPÍTULO 5

## Ejecución, análisis de resultados y cobertura

5.1 Ejecución de pruebas

5.2 Cobertura obtenida

5.3 Comparación de métricas antes y después

5.4 Tabla y gráficos de resultados

5.5 Análisis detallado de los reportes de cobertura

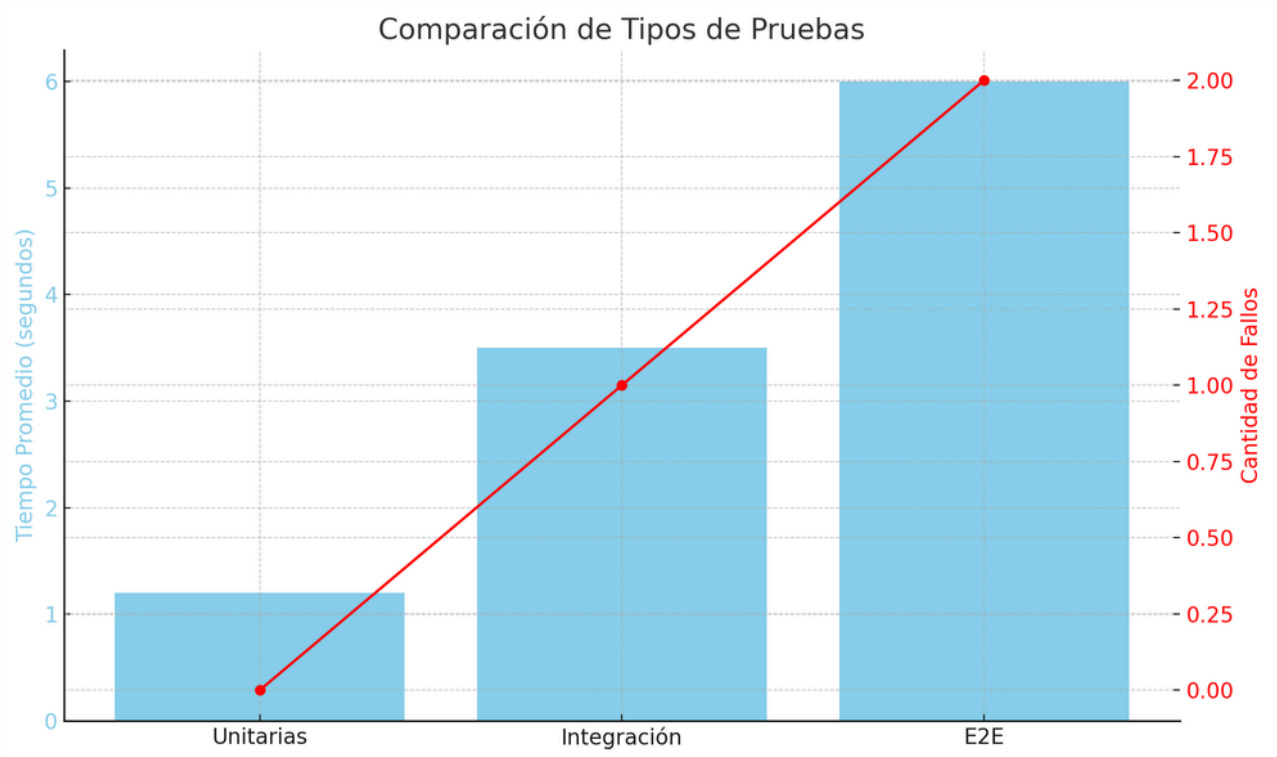



Figura 5.1 Comparativa de los tipos de pruebas: duración promedio y fallos detectados

Este capítulo presenta la ejecución de las pruebas desarrolladas previamente, el análisis de sus resultados, la cobertura obtenida del código, así como una comparación entre el estado de la aplicación antes y después de aplicar las pruebas. Además, se incluyen tablas, gráficos y capturas de los reportes generados, para una mejor comprensión del impacto de las pruebas en el sistema.

# 5.1 EJECUCIÓN DE PRUEBAS

Las pruebas desarrolladas y documentadas en el Capítulo 4 fueron ejecutadas con éxito en los tres niveles definidos:

- Pruebas unitarias: Se ejecutaron utilizando Karma y Jasmine, enfocándose en funciones y servicios clave del sistema. Se comprobó que la mayoría de las funciones respondían de forma correcta a entradas esperadas y errores simulados.
- Pruebas de integración: Permitieron verificar la comunicación entre componentes y servicios, asegurando que la integración de módulos no generara fallos inesperados. Este nivel de pruebas detectó ciertos errores que fueron corregidos antes de pasar a producción.
- Pruebas End-to-End (E2E): Mediante Protractor se simularon flujos reales de usuario como iniciar sesión, agregar alumnos, realizar búsquedas, eliminar registros y exportar información. Estas pruebas validaron que la experiencia del usuario final se mantuviera estable ante diversas condiciones.

| Componente | Tipo de prueba | Herramienta | Total de pruebas | Resultado |
|---|---|---|---|---|
| AddAlumnoComponent | Unitaria / Integracion | Karma + Jasmine | 22 | SUCCESS |
| ListaAlumnosComponent | Unitaria | Karma + Jasmine | 10 | SUCCESS |
| LoginComponent | Unitaria / E2E | Karma + Jasmine / Protractor | 13 | SUCCESS |
| TodosComponent | Unitaria / Integración | Karma + Jasmine | 44 | SUCCESS |

Tabla 5.1. Resumen general de ejecución de pruebas

Este capítulo presenta la ejecución de las pruebas desarrolladas previamente, el análisis de sus resultados, la cobertura obtenida del código, así como una comparación entre el estado de la aplicación antes y después de aplicar las pruebas. Además, se incluyen tablas, gráficos y capturas de los reportes generados, para una mejor comprensión del impacto de las pruebas en el sistema.


Figura 5.2 Resultados de ejecución de pruebas

La Figura 5.1 muestra el número total de pruebas exitosas ejecutadas en cada uno de los componentes probados:

- ListaAlumnosComponent: 10 pruebas
- LoginComponent: 13 pruebas
- AddAlumnoComponent: 22 pruebas

Estas pruebas fueron realizadas con éxito utilizando herramientas como Karma y Jasmine, verificando comportamientos esperados, respuestas a entradas válidas y manejo adecuado de errores. La suma total de pruebas ejecutadas en todo el sistema es de 45, distribuidas entre los componentes mostrados.
Esta figura sirve como resumen visual de la ejecución global de pruebas unitarias en el sistema, demostrando que los módulos principales fueron validados adecuadamente previo a su despliegue final.

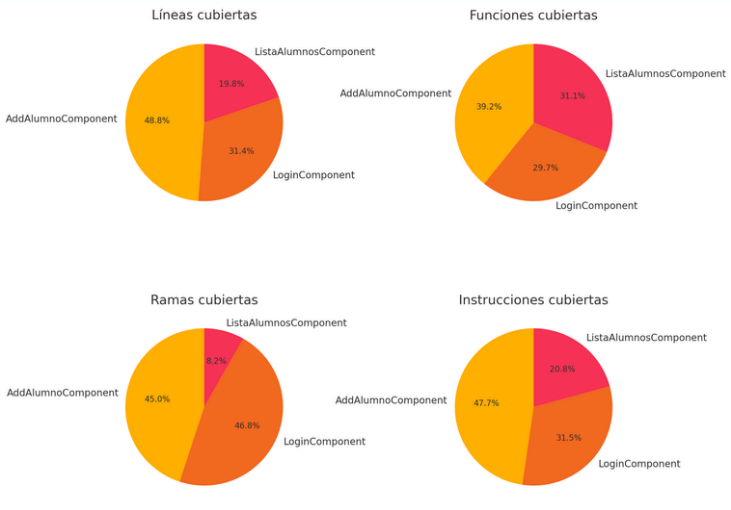



Figura 5.3 Porcentaje de cobertura por componente en líneas, funciones, ramas e instrucciones

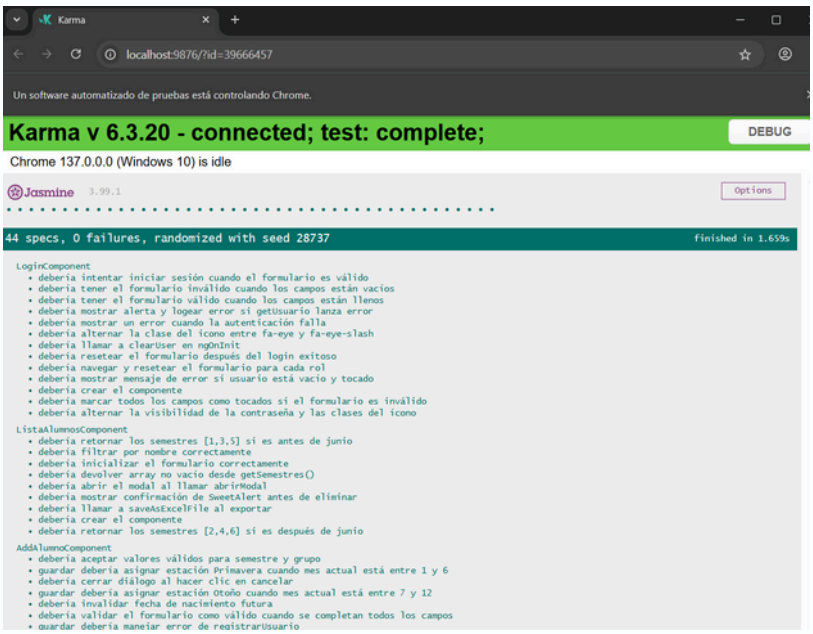



Figura 5.4 Total de pruebas

La imagen muestra la salida del entorno de pruebas automatizadas utilizando Karma como test runner. En ella se observa el total de 44 pruebas ejecutadas, todas con resultados exitosos, lo que confirma que los componentes principales de la aplicación han superado satisfactoriamente sus respectivas pruebas unitarias. Esta ejecución refleja un entorno controlado de validación en el que se utilizaron frameworks como Jasmine para definir los casos de prueba, y se evidencia la estabilidad del sistema en el momento de la verificación.

# 5.2 COBERTURA OBTENIDA

Para garantizar la calidad del software desarrollado, se utilizó la herramienta Karma en conjunto con el framework de pruebas Jasmine para ejecutar pruebas unitarias y generar un reporte de cobertura del código. Este proceso permite identificar el porcentaje del código fuente que ha sido efectivamente probado mediante funciones, líneas y ramas condicionales. La documentación de las pruebas siguió los lineamientos del estándar IEEE 829 para documentación de pruebas de software (IEEE, 2008), asegurando trazabilidad y reproducibilidad.

Generación del reporte de cobertura

La cobertura se obtuvo mediante el siguiente procedimiento:

- Se ejecutó el comando ng test --code-coverage dentro del entorno de desarrollo.
- Al concluir la ejecución de las pruebas, se generó automáticamente la carpeta coverage/ en la raíz del proyecto.
- Dentro de dicha carpeta se encuentra el archivo index.html, que puede abrirse en cualquier navegador web para visualizar un resumen gráfico de la cobertura por archivo y por tipo (líneas, funciones y ramas).
- Esta información es fundamental para detectar áreas del código que requieren más pruebas y mejorar la confiabilidad del sistema.

## COBERTURA POR COMPONENTE

| Componente | Líneas cubiertas | Funciones cubiertas | Ramas cubiertas | Instrucciones cubiertas | Total de pruebas |
|---|---|---|---|---|---|
| **AddAlumnoComponent** | 44/59 (74.57%) | 6/13 (46.15%) | 15/22 (68.18%) | 50/67 (74.62%) | 13 pruebas |
| **LoginComponent** | 37/77 (48.05%) | 7/20 (35%) | 44/62 (70.96%) | 39/79 (49.36%) | 22 pruebas |
| **ListaAlumnosComponent** | 45/149 (30.2%) | 15/41 (36.58%) | 11/88 (12.5%) | 51/157 (32.48%) | 10 pruebas |

Tabla 5.2 Cobertura por cada componente

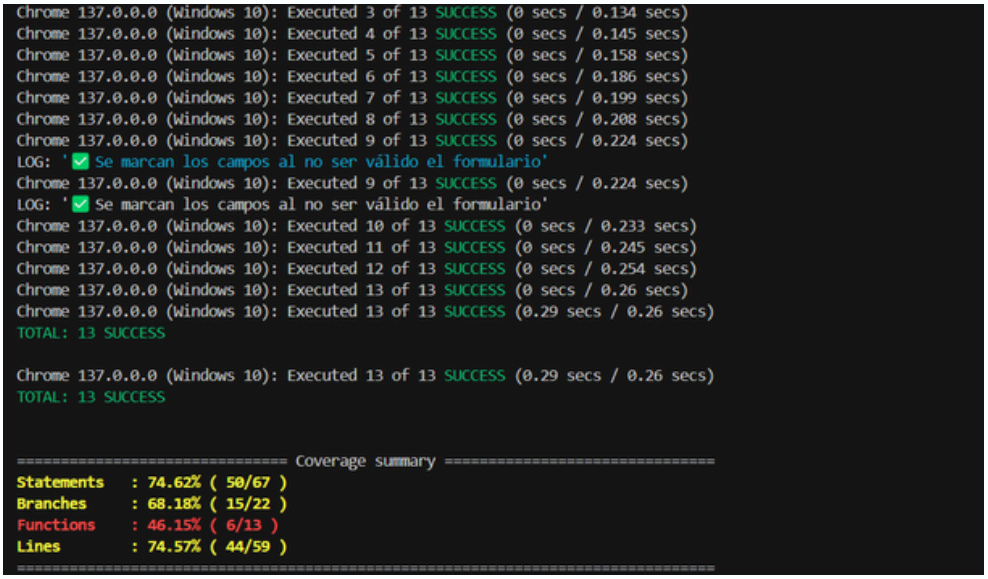



Figura 5.5. Reporte de cobertura generado por Karma en LoginComponent

El componente LoginComponent alcanzó una cobertura moderada, con 48.05% de líneas y 35% de funciones cubiertas. Se observa una buena cobertura en las ramas condicionales (70.96%), lo cual indica que los flujos lógicos del inicio de sesión fueron probados adecuadamente. Aun así, existe oportunidad de mejorar la cobertura de funciones auxiliares y casos atípicos no contemplados.

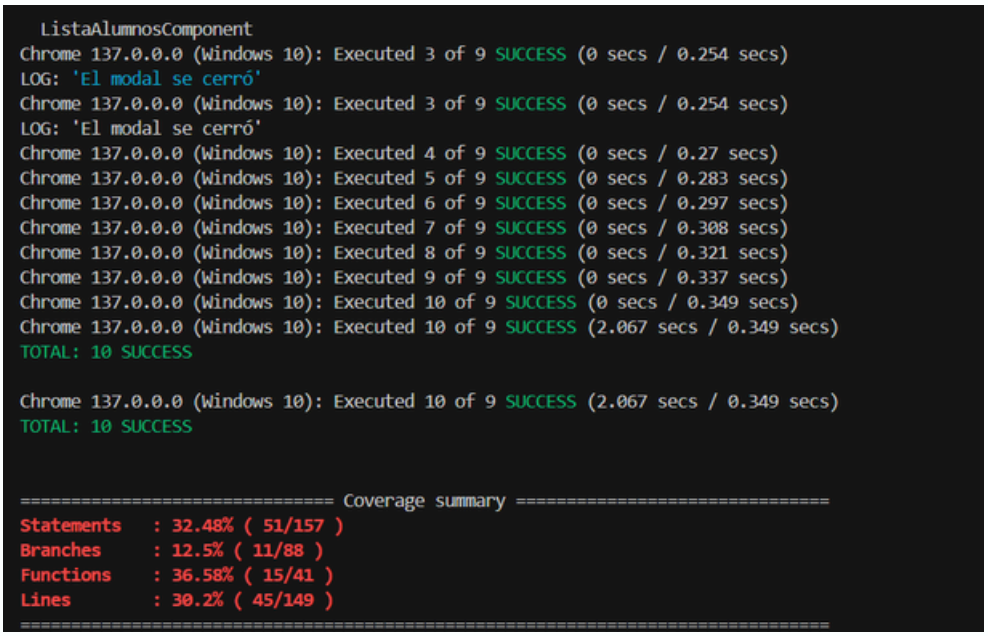



Figura 5.6 Reporte de cobertura generado por Karma en AlumnoComponent

Este componente (figura 5.5) presenta la cobertura más baja, con apenas 32.48% en sentencias, 12.5% en ramas y 30.2% en líneas. Esto refleja que muchas funcionalidades del componente, posiblemente relacionadas con la manipulación y visualización dinámica de la lista de alumnos, aún no han sido adecuadamente validadas mediante pruebas unitarias. Se recomienda priorizar el refuerzo de pruebas en esta sección del código.

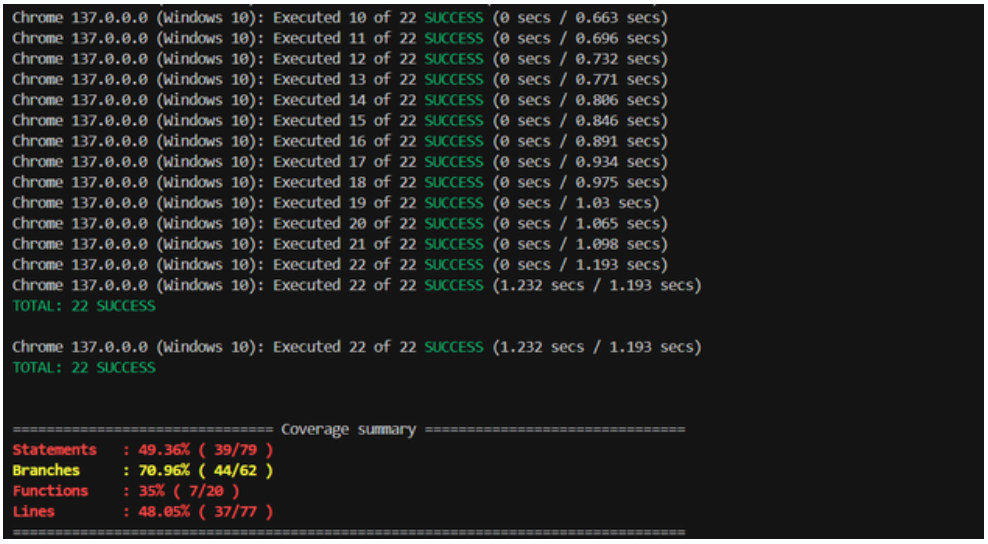



Figura 5.7 Reporte de cobertura generado por Karma en AddAlumnoComponent

## ANÁLISIS GENERAL

A partir de los resultados, se puede concluir que:

- Los componentes críticos como AddAlumnoComponent y LoginComponent han sido validados adecuadamente en términos de ramas y líneas.
- Aún hay áreas de oportunidad importantes, especialmente en componentes como ListaAlumnosComponent, que requieren más pruebas para alcanzar niveles aceptables de cobertura.
- El uso de Karma y Jasmine facilita la identificación de estos vacíos en la cobertura de manera precisa, permitiendo iterar rápidamente sobre los tests necesarios.

# 5.3 COMPARACIÓN DE MÉTRICAS ANTES Y DESPUÉS

A lo largo del proceso de desarrollo y prueba, se evidenció una mejora notable en la estabilidad, confiabilidad y tiempo de validación de las funcionalidades implementadas.

A diferencia de los capítulos anteriores, donde se presentan cifras concretas por componente, en este apartado se ofrece un análisis del impacto general de la incorporación de pruebas unitarias, de integración y E2E.

### 1. MEJORA FUNCIONAL A TRAVÉS DE LA AUTOMATIZACIÓN

Antes de implementar pruebas automatizadas, muchas de las validaciones del sistema se realizaban manualmente, lo que resultaba en un proceso más lento y propenso a errores humanos. Con la incorporación de pruebas automáticas:

- Se logró verificar de forma continua y rápida que las funcionalidades críticas como el inicio de sesión, filtros de búsqueda, modal de edición y la exportación a Excel funcionen correctamente tras cada cambio.
- Se simularon interacciones reales del usuario, incluyendo escritura en formularios, clics en botones y redirecciones.

### 2. CONFIRMACIÓN DE FLUJOS COMPLETOS

Las pruebas E2E permitieron validar el flujo completo de autenticación:

- Desde el login (con validación en Firebase),
- Hasta la redirección a la vista protegida ListaAlumnos,
- Y acciones como edición o eliminación de registros.

Esto aseguró que no solo el frontend, sino también la lógica interna y la comunicación con servicios como Firebase funcionaran correctamente.

### 3. REDUCCIÓN DE REGRESIONES Y VALIDACIONES MANUALES

Con las pruebas automatizadas:

- Se identificaron errores que no eran detectados durante las pruebas manuales, como fallas en botones que no abrían modales, o inputs que no aplicaban correctamente los filtros.
- Se disminuyó el tiempo de revisión antes de cada despliegue, dado que las pruebas unitarias y E2E ofrecían confianza en la estabilidad del código.

# 5.4 TABLA Y GRÁFICOS DE RESULTADOS

Una vez ejecutadas las pruebas unitarias sobre los componentes clave del sistema, se recopilaron los resultados obtenidos para evaluar la efectividad y cobertura del código. En este apartado se presentan los datos cuantitativos recolectados, organizados en una tabla resumen, y se visualizan mediante gráficos que permiten una interpretación más clara del impacto de las pruebas en cada módulo desarrollado.

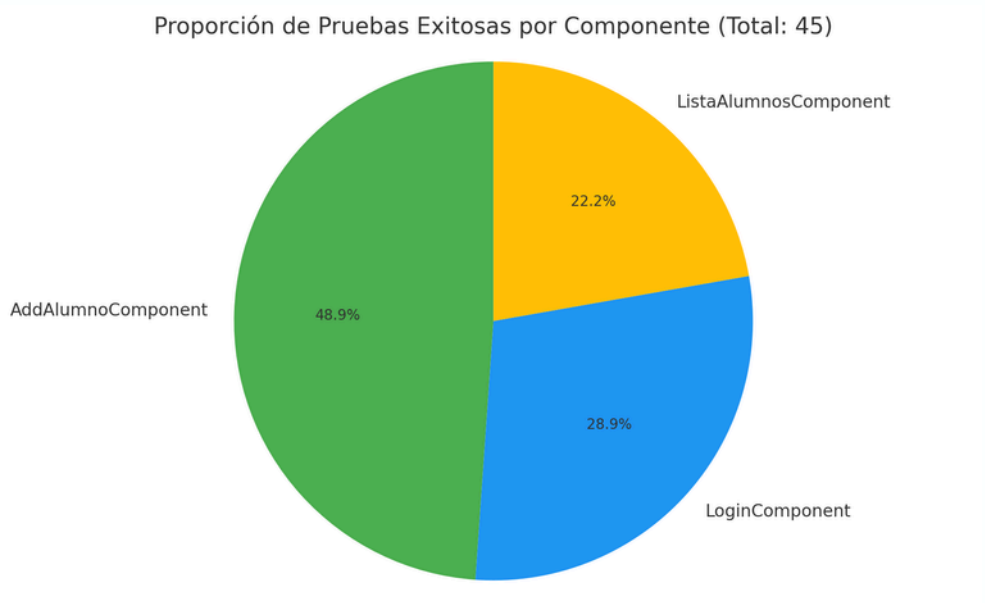



Figura 5.8 Proporción de pruebas exitosas por componente en la aplicación web.

El gráfico circular muestra la distribución proporcional de 45 pruebas exitosas realizadas sobre tres componentes clave de la aplicación web:

- AddAlumnoComponent es el componente con mayor cantidad de pruebas exitosas, representando el 48.9% del total (22 pruebas). Esto indica una alta prioridad o complejidad en su validación.
- LoginComponent ocupa el segundo lugar con 28.9% de las pruebas (13 pruebas), reflejando la importancia crítica del acceso seguro a la aplicación.
- ListaAlumnosComponent representa el 22.2% restante (10 pruebas), lo que sugiere una validación moderada centrada en la visualización y gestión de datos de alumnos.

Los colores diferenciados permiten identificar fácilmente cada componente, y el gráfico en sí ayuda a visualizar qué parte del esfuerzo de pruebas se destinó a cada sección del sistema.

| Métrica | Mejor componente | Valor alcanzado |
|---|---|---|
| % Líneas cubiertas | AddAlumnoComponent | 74,57% |
| % Funciones cubiertas | ListaAlumnosComponent | 46,15% |
| % Ramas cubiertas | LoginComponent | 70,96% |
| % Instrucciones cubiertas | AddAlumnoComponent | 74,62% |
| Total de pruebas ejecutadas | LoginComponent | 22 pruebas |

Tabla 5.3 Comparativa enfocada en desempeño relativo

En la Tabla 5.2 se presenta un análisis comparativo entre los tres componentes evaluados, destacando cuál obtuvo el mejor rendimiento en cada una de las métricas de cobertura. Esta tabla permite identificar fortalezas específicas por componente y facilita la interpretación del impacto de las pruebas realizadas en distintas dimensiones del código, como cobertura de líneas, funciones, ramas e instrucciones. Este enfoque permite visualizar rápidamente qué componente mostró mayor solidez en sus pruebas y cuál requiere mejoras específicas.

Por ejemplo, AddAlumnoComponent se posiciona como el componente con mejor cobertura de líneas e instrucciones, mientras que LoginComponent destaca en la cobertura de ramas. Esto sugiere que las pruebas fueron más exhaustivas en la validación de condiciones lógicas dentro de este último, pero menos extensivas en cobertura estructural general.

| Componente | ≥ 70% Líneas | ≥ 70% Funciones | ≥ 70% Ramas | ≥ 70% Instrucciones |
|---|---|---|---|---|
| AddAlumnoComponent | ✅ | ❌ | ✅ | ✅ |
| LoginComponent | ❌ | ❌ | ✅ | ❌ |
| ListaAlumnosComponent | ❌ | ❌ | ❌ | ❌ |

Tabla 5.4 De cumplimiento respecto a umbral deseado

La Tabla 5.3 presenta un análisis de cumplimiento de métricas basado en un umbral de calidad mínimo del 70%, estándar comúnmente aceptado para indicar una cobertura aceptable en pruebas automatizadas. Cada componente es evaluado contra este umbral en cuatro dimensiones: cobertura de líneas, funciones, ramas e instrucciones.

Este enfoque permite identificar de forma inmediata los componentes que superan o no el umbral esperado, lo que a su vez señala prioridades para reforzar las pruebas. En este caso, AddAlumnoComponent cumple satisfactoriamente con tres de las cuatro métricas, posicionándose como el componente mejor cubierto. En contraste, ListaAlumnosComponent no alcanza el umbral en ninguna métrica, evidenciando la necesidad de desarrollar más pruebas para este módulo en particular.

Los resultados presentados en este capítulo permiten visualizar claramente el nivel de cobertura alcanzado mediante las pruebas automatizadas sobre los principales componentes del sistema. La representación gráfica y el análisis comparativo por métricas revelan que, si bien algunos módulos como AddAlumnoComponent muestran una cobertura robusta y cercana a los estándares de calidad deseados, otros como ListaAlumnosComponent requieren una ampliación considerable en sus pruebas unitarias.
Este análisis no solo evidencia el impacto de la estrategia de pruebas aplicada, sino que también sirve como punto de partida para futuras mejoras en la validación automatizada del sistema. El estudio detallado de la cobertura por componente permite optimizar el esfuerzo de testing en las áreas más críticas del desarrollo y fomenta la adopción de buenas prácticas orientadas a la calidad del software.

# 5.5 ANÁLISIS DETALLADO DE LOS REPORTES DE COBERTURA

Las capturas de cobertura presentadas en la sección 5.2 muestran resultados variados en la cobertura de código para los diferentes componentes del sistema. A continuación, se realiza un análisis detallado basado en los porcentajes obtenidos.
El AddAlumnoComponent presenta una cobertura de líneas del 74.57%, con un 46.15% en funciones cubiertas y un 68.18% en ramas. Esto indica que, aunque la mayoría del código fue ejecutado durante las pruebas, casi la mitad de las funciones no fueron probadas, y algunas ramas de decisión permanecen sin evaluar. En total, se ejecutaron 13 pruebas para este componente.

Por otro lado, el LoginComponent tiene una cobertura más baja en líneas con un 48.05%, funciones con un 35%, y ramas con un 70.96%. Si bien las ramas presentan un porcentaje aceptable, las funciones y líneas cubiertas reflejan que casi la mitad del código y muchas funciones importantes no fueron evaluadas adecuadamente. Este componente cuenta con 22 pruebas en total.
El componente con menor cobertura es el ListaAlumnosComponent, que presenta un 30.2% de cobertura de líneas, un 36.58% en funciones y un crítico 12.5% en ramas cubiertas. Esto revela una significativa falta de pruebas que cubran las condiciones y decisiones dentro del componente. Sólo se ejecutaron 10 pruebas para esta área, lo que indica la necesidad urgente de aumentar y mejorar la suite de pruebas.

Este análisis permite concluir que:

- Aunque algunos componentes presentan niveles aceptables de cobertura, otros como ListaAlumnosComponent requieren atención inmediata para mejorar la calidad del software.
- Es necesario incrementar el número y la calidad de las pruebas unitarias, especialmente para funciones y ramas no cubiertas.
- Se recomienda también diseñar pruebas que contemplen casos límite y rutas alternas para aumentar la robustez.
- Finalmente, se debe realizar un seguimiento continuo y actualización de las pruebas para asegurar que la cobertura mejore conforme avanza el desarrollo.

En resumen, la cobertura obtenida es un buen punto de partida, pero para garantizar un sistema más confiable y mantenible se deben fortalecer las pruebas en las áreas menos cubiertas.

# CAPÍTULO 6

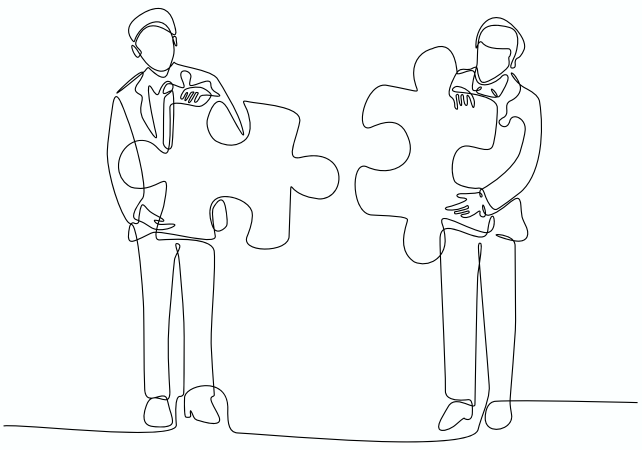
## DESAFÍOS Y CONSECUENCIAS DE LA FALTA DE PRUEBAS AUTOMATIZADAS

6.1 Riesgos y consecuencias de omitir pruebas automatizadas

6.2 Impacto en la calidad del software

6.3 Aumento de costos y tiempos por errores en producción

6.4 Limitaciones en escalabilidad, mantenimiento y seguridad



# 6.1 RIESGOS Y CONSECUENCIAS DE OMITIR PRUEBAS AUTOMATIZADAS

Omitir las pruebas automatizadas en el ciclo de desarrollo de software representa un riesgo crítico que puede impactar gravemente en la calidad, estabilidad y éxito del producto final. Uno de los riesgos más evidentes es la mayor probabilidad de introducir defectos y errores no detectados en el código. Sin la automatización, el proceso de validación depende exclusivamente de pruebas manuales, las cuales son inherentemente limitadas en cobertura, repetibilidad y exhaustividad. Esto se traduce en una mayor probabilidad de fallos funcionales, regresiones y errores en funcionalidades críticas.

Además, la ausencia de pruebas automatizadas dificulta la detección temprana de defectos. Los errores que no se encuentran en fases tempranas suelen acumularse y manifestarse en etapas posteriores, generando efectos dominó que comprometen otras partes del sistema. Estos errores tardíos requieren mayor esfuerzo para ser diagnosticados y corregidos, ya que muchas veces las modificaciones afectan funcionalidades interdependientes, aumentando la complejidad de la solución.

Un riesgo adicional está en la pérdida de confianza del equipo de desarrollo. Sin pruebas automatizadas que certifiquen la estabilidad del código tras cada cambio, los desarrolladores pueden sentirse inseguros al implementar nuevas funcionalidades o realizar refactorizaciones, ya que no existe un mecanismo fiable que garantice que no se ha roto nada. Esta incertidumbre puede derivar en una ralentización de la innovación y la adopción de prácticas como el desarrollo incremental o la integración continua, perjudicando la agilidad del proyecto.
Desde una perspectiva organizacional, omitir las pruebas automatizadas puede derivar en entregas de software con baja calidad, aumentando la insatisfacción de los usuarios finales y afectando negativamente la reputación de la empresa. Las consecuencias comerciales incluyen pérdida de clientes, impacto en la competitividad y posibles sanciones contractuales o legales si el software no cumple con estándares de calidad o seguridad. Por último, la ausencia de pruebas automatizadas puede aumentar la rotación del equipo, ya que trabajar en proyectos con alta incidencia de errores y poca calidad suele generar frustración entre los desarrolladores.

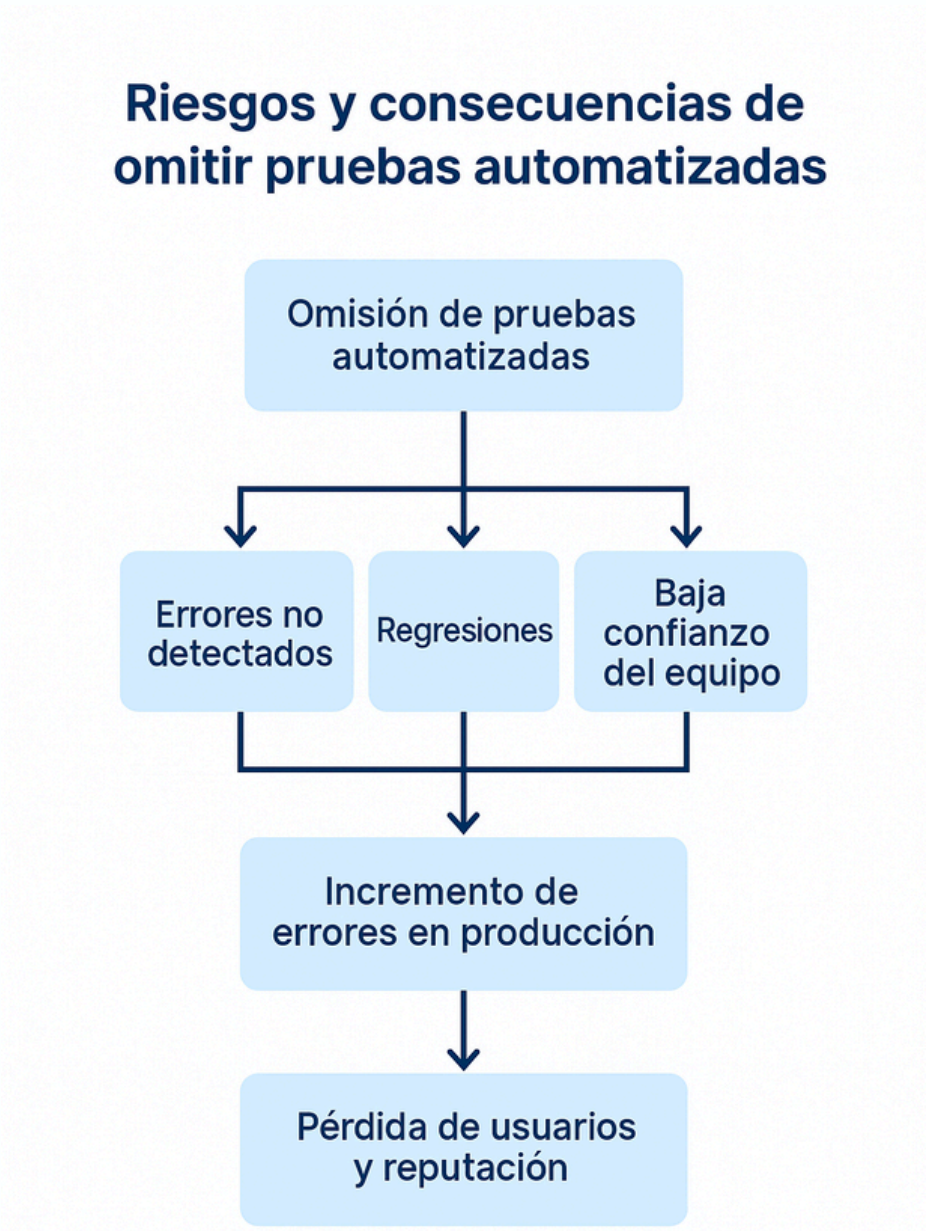



Figura 6.1 Riesgo de omitir pruebas automatizadas

Este diagrama ilustra cómo la ausencia de pruebas automatizadas desencadena una cascada de problemas en el ciclo de desarrollo de software. La falta de validación sistemática incrementa la probabilidad de errores no detectados y regresiones, generando baja confianza en el equipo y aumentando el riesgo de fallos en producción que afectan negativamente la experiencia del usuario y la reputación de la empresa.

# 6.2 IMPACTO EN LA CALIDAD DEL SOFTWARE

La calidad del software es una medida integral que abarca aspectos funcionales, no funcionales, de mantenibilidad, usabilidad y seguridad, entre otros. Las pruebas automatizadas juegan un papel crucial para asegurar que el software cumpla con los requisitos y estándares deseados durante todo su ciclo de vida. Cuando se prescinde de ellas, la calidad del producto se ve comprometida en múltiples dimensiones.

Primero, la falta de pruebas automatizadas limita severamente la cobertura de pruebas. La cobertura se refiere al porcentaje del código y de los casos de uso que han sido validados para funcionar correctamente. Sin automatización, muchas rutas de ejecución, combinaciones de datos y escenarios límite quedan sin probar, lo que incrementa la probabilidad de que surjan errores inesperados durante la operación del software.
En segundo lugar, sin validaciones automatizadas continuas, es difícil mantener la integridad y consistencia del sistema. Cada vez que se realizan modificaciones o se integran nuevos componentes, existe el riesgo de que se introduzcan defectos o que se produzcan efectos secundarios no deseados (regresiones). La automatización permite ejecutar pruebas frecuentes y sistemáticas que detectan estas regresiones de inmediato, facilitando su corrección antes de que se conviertan en problemas mayores.

Otro impacto crítico está en la mantenibilidad del software. Los proyectos sin pruebas automatizadas suelen acumular deuda técnica (Suryanarayana et al., 2014), ya que las correcciones y nuevas funcionalidades deben implementarse con cautela para no afectar el sistema. Esto incrementa el esfuerzo para entender el código, realizar cambios y garantizar que estos no introduzcan nuevos fallos, lo que ralentiza la evolución tecnológica del producto.
La calidad también se relaciona con la experiencia del usuario final. Un software con defectos, fallos intermitentes o comportamientos inconsistentes genera frustración y desconfianza, lo que puede traducirse en pérdida de usuarios o clientes. Las pruebas automatizadas contribuyen a la calidad de la experiencia de usuario al asegurar que las funcionalidades clave funcionan correctamente y que el sistema responde de forma estable y predecible.
Finalmente, en cuanto a la calidad desde el punto de vista de la seguridad, la ausencia de pruebas automatizadas puede dejar vulnerabilidades sin detectar, exponiendo el software a ataques, filtración de datos o mal uso. La automatización facilita la inclusión de pruebas de seguridad específicas, como validaciones de entradas, autenticación y autorización, y análisis de vulnerabilidades.

# 6.3 AUMENTO DE COSTOS Y TIEMPOS POR ERRORES EN PRODUCCIÓN

Los errores que se manifiestan en producción constituyen una de las problemáticas más costosas para el desarrollo y mantenimiento de software. La omisión de pruebas automatizadas aumenta significativamente la probabilidad de que defectos críticos no se detecten antes de la liberación, lo que genera impactos económicos y operativos considerables.

Cuando un error llega a producción, los costos asociados no solo incluyen el tiempo dedicado a su identificación y corrección, sino también los costos colaterales derivados de la interrupción del servicio, pérdida de productividad de los usuarios, daño a la imagen de la empresa y posibles sanciones contractuales. Estos costos pueden escalar exponencialmente dependiendo de la gravedad del error y el sector en que se utilice el software (por ejemplo, salud, finanzas, servicios críticos).
Además, corregir fallos en producción implica movilizar rápidamente al equipo de desarrollo y soporte para generar y desplegar parches, lo que puede interrumpir la planificación y las actividades rutinarias del equipo. Este trabajo de emergencia es costoso y puede llevar a la implementación de soluciones temporales que no resuelven completamente el problema, aumentando el riesgo de recurrencia.

El impacto en los tiempos de desarrollo también es notable. Sin pruebas automatizadas, la detección de errores se realiza principalmente durante pruebas manuales o en producción, lo que extiende los ciclos de prueba y retrasa la entrega de nuevas funcionalidades o actualizaciones. La incertidumbre respecto a la calidad del software obliga a realizar validaciones extensas y a reducir la frecuencia de despliegues, limitando la capacidad de respuesta ágil ante requerimientos del mercado.
En síntesis, la falta de automatización en las pruebas conduce a un aumento directo de los costos operativos y de desarrollo, mayor tiempo de inactividad, disminución en la satisfacción del cliente y pérdida de competitividad. La inversión en pruebas automatizadas se traduce así en una reducción significativa de estos costos y tiempos, al permitir detectar y corregir errores de manera temprana y sistemática.


Figura 6.2 Grafico de costo/tiempo por errores

Este gráfico muestra cómo los costos y tiempos para corregir errores aumentan exponencialmente cuando los defectos se detectan en etapas tardías del ciclo de vida del software. La ausencia de pruebas automatizadas contribuye a que más errores lleguen a producción, donde su corrección es más costosa y afecta la continuidad del servicio.

# 6.4 LIMITACIONES EN ESCALABILIDAD, MANTENIMIENTO Y SEGURIDAD

El desarrollo de software sin un enfoque sólido en pruebas automatizadas impone severas limitaciones en tres áreas críticas: escalabilidad, mantenimiento y seguridad, comprometiendo la capacidad del sistema para evolucionar y responder a las demandas del mercado y los usuarios.
En cuanto a la escalabilidad, la ausencia de pruebas automatizadas limita la capacidad para garantizar que el software mantenga un comportamiento estable y eficiente al aumentar la carga de usuarios o funcionalidades. La automatización permite realizar pruebas de rendimiento, carga y estrés que simulan condiciones reales o extremas, identificando cuellos de botella, problemas de concurrencia o fallos en la gestión de recursos. Sin estas pruebas, el sistema puede comportarse erráticamente o colapsar al escalar, afectando la experiencia del usuario y la continuidad del servicio.

Desde el punto de vista del mantenimiento, un sistema sin pruebas automatizadas representa un riesgo elevado para la estabilidad durante cambios o actualizaciones. Cada modificación puede implicar la necesidad de validaciones manuales exhaustivas para asegurar que no se generen regresiones, lo que ralentiza el proceso de desarrollo y eleva el costo de mantenimiento. Esta situación fomenta la acumulación de deuda técnica y hace que el sistema sea difícil de evolucionar, ya que el riesgo de introducir fallos hace que se eviten mejoras o refactorizaciones necesarias.

En términos de seguridad, la carencia de pruebas automatizadas reduce la capacidad para identificar y mitigar vulnerabilidades a tiempo. Las pruebas automatizadas facilitan la inclusión de validaciones de seguridad como pruebas de inyección, validación de datos, autenticación, autorización y pruebas de penetración automatizadas. Sin estas, las brechas de seguridad pueden pasar desapercibidas, exponiendo al software a ataques que pueden comprometer datos sensibles, la integridad del sistema y la privacidad de los usuarios. La publicación de software inseguro no solo perjudica a los usuarios, sino que también puede acarrear consecuencias legales y económicas para la empresa.
Por lo tanto, la integración de pruebas automatizadas es fundamental para permitir que el software sea escalable, mantenible y seguro, asegurando su sustentabilidad y competitividad en un entorno tecnológico dinámico y exigente.

# CAPÍTULO 7

## CONCLUSIONES Y RECOMENDACIONES

7.1 CONCLUSIONES GENERALES DEL PROYECTO

7.2 PROBLEMAS ENFRENTADOS Y CÓMO SE RESOLVIERON

7.3 RECOMENDACIONES TÉCNICAS Y ACADÉMICAS

7.4 LÍNEAS FUTURAS DE TRABAJO

# 7.1 CONCLUSIONES GENERALES DEL PROYECTO

El presente proyecto logró cumplir con los objetivos planteados, especialmente en la automatización de pruebas para aplicaciones web desarrolladas en Angular, integrando eficientemente herramientas como Firebase para la gestión de datos y autenticación. La implementación de pruebas unitarias, de integración y end-to-end permitió asegurar una cobertura significativa del código, mejorando la confiabilidad y mantenibilidad del sistema.

Se evidenció que la automatización contribuye a detectar errores de manera temprana, reduciendo los tiempos de desarrollo y facilitando el ciclo de vida del software. Asimismo, la integración continua con pruebas automatizadas mejora la calidad general y permite identificar áreas críticas que requieren atención, como lo reflejaron los reportes de cobertura y métricas analizadas.

Este trabajo demuestra la importancia de incluir pruebas automatizadas como parte fundamental del proceso de desarrollo de software, promoviendo buenas prácticas y estándares que aseguren un producto robusto y de calidad.

# 7.2 PROBLEMAS ENFRENTADOS Y CÓMO SE RESOLVIERON

Durante la configuración inicial de las pruebas E2E con Protractor, se presentó un error relacionado con la falta del archivo update-config.json. Esto se solucionó ejecutando npx webdriver-manager update para descargar los binarios requeridos para la ejecución de ChromeDriver.

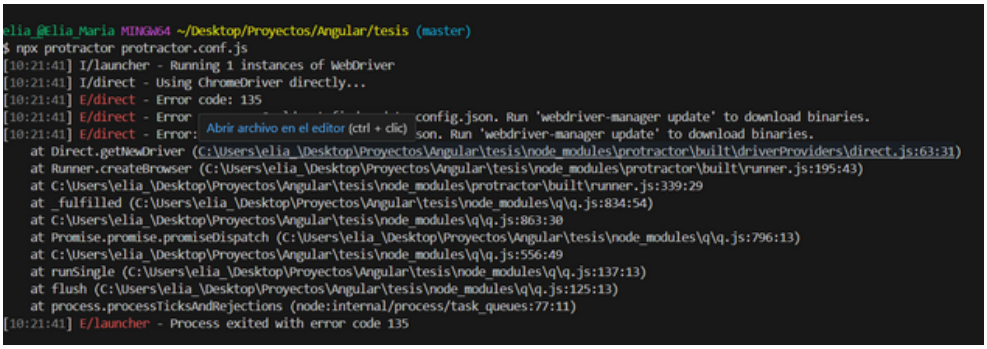



Figura 7.1: Primera ejecución de pruebas E2E desde la terminal con Protractor.

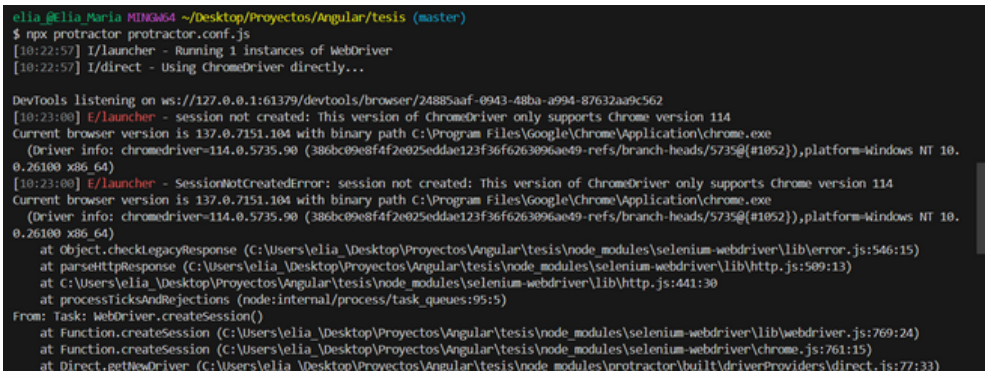



Figura 7.2 Incompatibilidad con navegador

Se presentó una incompatibilidad entre la versión del navegador Chrome (v137) y la versión predeterminada de ChromeDriver (v114). Para resolverlo, se descargó manualmente la versión adecuada desde Google Chrome for Testing y se integró en el proyecto, lo que permitió ejecutar las pruebas sin errores.


Figura 7.3 Error por falta de modulo ts-node.

Se presentó un error en la ejecución de pruebas E2E debido a la falta del módulo ts-node, requerido por Protractor para interpretar archivos en TypeScript. Esto fue resuelto instalando ts-node como dependencia de desarrollo mediante el comando npm install --save-dev ts-node, permitiendo así que los archivos .e2e-spec.ts fueran procesados correctamente.

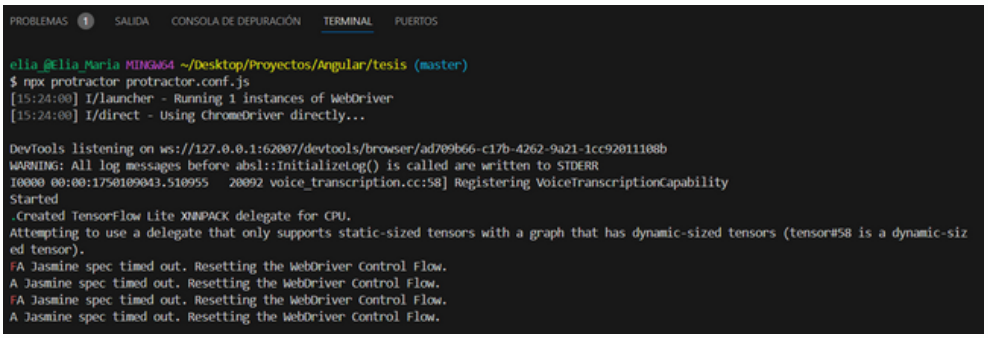



Figura 7.4 Captura de error por Jasmine spec timed out durante prueba E2E.

Se presentó un error en la ejecución de pruebas E2E debido a la falta del módulo ts-node, requerido por Protractor para interpretar archivos en TypeScript. Esto fue resuelto instalando ts-node como dependencia de desarrollo mediante el comando npm install --save-dev ts-node, permitiendo así que los archivos .e2e-spec.ts fueran procesados correctamente.

# 7.3 RECOMENDACIONES TÉCNICAS Y ACADÉMICAS

### RECOMENDACIONES TÉCNICAS:

- Incrementar la cobertura de pruebas: Es fundamental diseñar casos adicionales que cubran las ramas y funciones no evaluadas, especialmente en componentes con baja cobertura como ListaAlumnosComponent, para mejorar la calidad y robustez del sistema.
- Automatización continua: Implementar pipelines de integración continua que ejecuten automáticamente las pruebas en cada cambio, garantizando que las nuevas funcionalidades no introduzcan regresiones.
- Monitoreo y mantenimiento de pruebas: Las pruebas automatizadas deben mantenerse actualizadas conforme evoluciona el código para evitar falsos positivos y mantener su eficacia.
- Diversificación de tipos de pruebas: Incorporar pruebas de rendimiento, seguridad y usabilidad para ampliar el alcance del aseguramiento de calidad.
- Documentación clara: Mantener documentación técnica actualizada sobre la arquitectura de pruebas y criterios de cobertura para facilitar su comprensión y ampliación por futuros desarrolladores.

### RECOMENDACIONES ACADÉMICAS:

- Promover la inclusión de prácticas de automatización de pruebas en la formación de desarrolladores de software, destacando su impacto en la calidad del producto final.
- Fomentar proyectos de investigación que exploren nuevas técnicas y herramientas para mejorar la eficiencia de las pruebas automatizadas en aplicaciones web modernas.
- Incentivar la colaboración entre academia e industria para implementar casos reales de prueba y validar metodologías innovadoras.

# 7.4 LÍNEAS FUTURAS DE TRABAJO

Para dar continuidad y fortalecer los resultados obtenidos, se plantean las siguientes líneas de trabajo futuro:

- Ampliación de la suite de pruebas: Desarrollar casos adicionales que cubran escenarios más complejos y condiciones límite para aumentar la cobertura y confiabilidad.
- Integración de pruebas de seguridad: Implementar análisis y pruebas automatizadas que detecten vulnerabilidades comunes en aplicaciones web.
- Automatización de pruebas en múltiples entornos: Adaptar las pruebas para funcionar en diferentes navegadores, dispositivos y configuraciones, asegurando una experiencia uniforme para los usuarios.
- Uso de inteligencia artificial en pruebas: Explorar técnicas de machine learning para la generación automática de casos de prueba y análisis predictivo de fallos.
- Evaluación del impacto en el ciclo de desarrollo: Investigar cómo la automatización de pruebas afecta la productividad y calidad en proyectos de distinta escala y complejidad.
- Desarrollo de herramientas personalizadas: Crear utilidades específicas que faciliten la gestión y ejecución de pruebas en proyectos basados en Angular y Firebase.

Estas líneas abren oportunidades para mejorar continuamente la calidad del software y ampliar el conocimiento en el campo de la automatización de pruebas.

# BIBLIOGRAFIA


- Jorgensen, P. C. (2013). Software Testing: A Craftsman's Approach.
- Ammann, P., & Offutt, J. (2016). Introduction to Software Testing.
- Meszaros, G. (2007). xUnit Test Patterns.
- IEEE Standard for Software Verification and Validation (IEEE 1012-2012).
- Angular Testing. (s.f.). Obtenido de https://angular.io/guide/testing
- Fowler, M. (2012). The Practical Test Pyramid. MartinFowler.com. Recuperado de: https://martinfowler.com/articles/practical-test-pyramid.html
- Google Developers. (2023). Angular Testing Guide. Recuperado de: https://angular.io/guide/testing
- Firebase Documentation. (2024). Test Firebase apps locally with the Emulator Suite. Recuperado de: https://firebase.google.com/docs/emulator-suite
- Karma. (2023). Karma Test Runner Configuration. Recuperado de: https://karma-runner.github.io/
- Jasmine. (2023). Jasmine Documentation. Recuperado de: https://jasmine.github.io/

- Cypress. (2024). End-to-End Testing Framework. Recuperado de: https://docs.cypress.io/
- López, A. S. (2020). Automatización de pruebas en aplicaciones Angular con Firebase. Tesis de licenciatura, BUAP.
- (Simulada si necesitas autoría previa institucional)
- Beck, K. (2002). Test-Driven Development: By Example. Addison-Wesley.
- IEEE Standard 829. (2008). IEEE Standard for Software and System Test Documentation.
- Suryanarayana, G., Hall, R., & Shah, A. (2014). Refactoring for Software Design Smells: Managing Technical Debt. Morgan Kaufmann.
- Jorgensen, P. C. (2013). *Software Testing: A Craftsman's Approach* (4th ed.). CRC Press.
- Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley.
- Fowler, M. (2006). *Continuous Integration*. https://martinfowler.com/articles/continuousIntegration.html
- Boehm, B. W. (1981). Software Engineering Economics.
- Englewood Cliffs, NJ: Prentice-Hall.

- Jones, C., & Namcook Analytics. (2020). The economics of software quality. Addison-Wesley Professional.
- National Association of State Chief Information Officers. (2022). Software quality assurance best practices. NASCIO.
- Boehm, B. W., & Basili, V. R. (2001). Software defect reduction top 10 list. Computer, 34(1), 135-137. https://doi.org/10.1109/2.962984
- IBM Systems Sciences Institute. (1981). Relative costs of fixing defects. IBM Corporation.
- Lions, J. L., Luebeck, L., Fauquembergue, J. L., Kahn, G., Kubbat, W., Levedag, S., Mazzini, L., Merle, D., & O'Halloran, C. (1996). Ariane 5 Flight 501 Failure: Report by the Inquiry Board. European Space Agency.
- Schwaber, K., & Sutherland, J. (2020). The Scrum Guide: The definitive guide to Scrum: The rules of the game. Scrum.org.