# TRABAJO FIN DE GRADO DESARROLLO DE APLICACIONES WEB

**“AUTOMATIZACION DE PRUEBAS EN APLICACIÓN WEB”**

## 

**Alumno**: Javier Díaz Fernández

**Profesor**: Jose Martínez

## RESUMEN

Este trabajo de Fin de Grado se centra en la automatización de pruebas automáticas en aplicaciones web para el correcto desarrollo de la aplicación y como ayuda para los desarrolladores de la aplicación. En este proyecto he desarrollado una serie completa de tests para comprobar la funcionalidad, diseño, análisis de rendimiento, usabilidad y evaluación de accesibilidad sobre página web de CogniFit, que es una página web especializada en entrenamientos cognitivos. Para ello he usado el framework Playwright con el lenguaje de TypeScript para tener tipadas todas las definiciones y asegurarnos del correcto funcionamiento de los tests.

He elegido este tipo de proyecto de realización porque es a lo que me quiero dedicar, es uno de los sectores de trabajo más olvidados y posiblemente de los primarios para tener una aplicación de calidad que triunfe en el sector. Hablo de QA que es Quality Assurance y se trata de testear y probar la funcionalidad, accesibilidad y usabilidad de las nuevas actualizaciones de una Aplicación. El correcto trabajo de las pruebas de test garantiza que el producto tenga la mejor calidad posible y el cliente quede totalmente satisfecho. Este proyecto no solo me ha permitido aplicar conocimiento adquiridos durante el grado, sino también profundizar en tecnologías y metodologías actuales del sector.

Los resultados obtenidos demuestran que la implementación de un sistema de pruebas automatizadas proporciona beneficios en términos de detección temprana de los errores, consistencia en las pruebas y optimización de recursos. Además, el proyecto está abierto para nuevas ampliaciones y mejoras en el proceso de QA de dicha aplicación web, favoreciendo así el trabajo en el equipo en el supuesto caso de que estuviésemos trabajando en la empresa donde otro compañero, además de entender las pruebas automatizadas podría ampliarlas sin problema alguno.

**Palabras clave**: Automatización de pruebas, Calidad de software, Playwright, TypeScript, CogniFit, Accesibilidad, Usabilidad, Funcionalidad, Layout.

Tabla de contenido

[TRABAJO FIN DE GRADO DESARROLLO DE APLICACIONES WEB 1](#_Toc199799283)

[1](#_Toc199799284)

[RESUMEN 2](#_Toc199799285)

[1. INTRODUCCIÓN 6](#_Toc199799286)

[1.1 Motivación y contexto 6](#_Toc199799287)

[1.2 Objetivos 6](#_Toc199799288)

[1.3 Estructura del documento 7](#_Toc199799289)

[2. MARCO TEÓRICO 8](#_Toc199799290)

[2.1. Fundamentos de QA (Quality Assurance) y Testing 8](#_Toc199799291)

[2.2. Automatización de pruebas 9](#_Toc199799292)

[2.3. Tipos de pruebas en aplicaciones web 10](#_Toc199799293)

[2.3.1. Pruebas de Layout 10](#_Toc199799294)

[2.3.2. Pruebas Funcionales 11](#_Toc199799295)

[2.3.3. Pruebas de Rendimiento 11](#_Toc199799296)

[2.3.4. Pruebas de accesibilidad 11](#_Toc199799297)

[2.3.5. Otros tipos de pruebas 12](#_Toc199799298)

[2.4. Herramientas y Frameworks de automatización 12](#_Toc199799299)

[2.4.1. Selenium 12](#_Toc199799300)

[2.4.2. Cypress 13](#_Toc199799301)

[2.4.3. Puppeteer 13](#_Toc199799302)

[2.4.4. TestCafe 14](#_Toc199799303)

[2.4.5. Playwright 14](#_Toc199799304)

[2.5. Playwright como herramienta de automatización 15](#_Toc199799305)

[2.5.1. Arquitectura de Playwright 15](#_Toc199799306)

[2.5.2. Características principales 16](#_Toc199799307)

[3. ANÁLISIS Y DISEÑO 16](#_Toc199799308)

[3.1. Análisis de requisitos 16](#_Toc199799309)

[3.1.1. Requisitos funcionales 16](#_Toc199799310)

[3.1.2. Requisitos no funcionales 17](#_Toc199799311)

[3.1.3. Análisis de la aplicación objetivo 17](#_Toc199799312)

[3.2. Selección de tecnologías 18](#_Toc199799313)

[3.2.1. Criterios de selección 18](#_Toc199799314)

[3.2.2. Tecnologías seleccionadas 18](#_Toc199799315)

[3.3. Arquitectura de la solución 19](#_Toc199799316)

[3.3.1. Visión general 19](#_Toc199799317)

[3.3.2. Estructura de directorios 20](#_Toc199799318)

[3.3.3. Flujo de ejecución 21](#_Toc199799319)

[3.3.4. Patrones de diseño 21](#_Toc199799320)

[3.4. Diseño de casos de prueba 21](#_Toc199799321)

[3.4.1. Metodología 22](#_Toc199799322)

[3.4.2. Casos de prueba de layout 22](#_Toc199799323)

[3.4.3. Casos de prueba funcionales 23](#_Toc199799324)

[3.4.4. Casos de prueba de rendimiento 24](#_Toc199799325)

[3.4.5. Casos de prueba de accesibilidad 25](#_Toc199799326)

[3. IMPLEMENTACIÓN 26](#_Toc199799327)

[4.1. Configuración del entorno 26](#_Toc199799328)

[4.1.1. Requisitos previos 26](#_Toc199799329)

[4.1.2. Inicialización del proyecto 26](#_Toc199799330)

[4.1.3. Instalación de dependencias 27](#_Toc199799331)

[4.1.4. Configuración de Playwright 28](#_Toc199799332)

[4.1.5. Configuración de scripts “npm” 29](#_Toc199799333)

[4.1.5. Configuración de GitHub Actions 30](#_Toc199799334)

[5. RESULTADOS Y VALIDACIÓN 30](#_Toc199799335)

[5.1. Ejecución de pruebas 30](#_Toc199799336)

[5.1.1. Ejecución local 31](#_Toc199799337)

[5.1.2. Ejecución en CI/CD 31](#_Toc199799338)

[5.2. Análisis de resultados 31](#_Toc199799339)

[5.2.1. Análisis de pruebas de layout 32](#_Toc199799340)

[5.2.2. Análisis de pruebas de rendimiento 32](#_Toc199799341)

[5.2.3. Análisis de pruebas de accesibilidad 32](#_Toc199799342)

[5.2.4 Análisis de pruebas de funcionalidad 32](#_Toc199799343)

[5.3. Problemas detectados y soluciones 32](#_Toc199799344)

[5.3.1. Problemas principales detectados 32](#_Toc199799345)

[5.4. Métricas de calidad obtenidas 33](#_Toc199799346)

[5.5 Conclusiones del análisis 33](#_Toc199799347)

[6. CONCLUSION 34](#_Toc199799348)

[7. BIBLIOGRAFÍA 34](#_Toc199799349)

[8. ANEXOS 35](#_Toc199799350)

[8.1. Código fuente 35](#_Toc199799351)

[8.3. Glosario de términos 35](#_Toc199799352)

# INTRODUCCIÓN

## Motivación y contexto

La calidad del software tiene mucha importancia en el desarrollo de aplicaciones web modernas. Como usuario, todos hemos experimentado la frustración de encontrarnos con páginas web que no funcionan correctamente: botones que no responden, formularios que pierden información, diseños que se rompen en ciertos dispositivos o tiempos de carga excesivamente largos. Estas experiencias negativas no solo afectan a la satisfacción del usuario, sino que pueden tener un impacto directo en el éxito de un negocio.

Mi interés por la automatización de pruebas surgió durante mis prácticas profesionales, donde tuve la oportunidad de trabajar como QA tester manual. Durante ese periodo, observé cómo gran parte de mi tiempo se invertía en ejecutar repetidamente las mismas pruebas cada vez que se implementaba un cambio en el código. Este proceso, además de tedioso, estaba sujeto a errores humanos y consumía recursos que podrían haberse destinado a tareas más creativas y de mayor valor añadido.

Para este Trabajo de Fin de Grado, he decidido centrarme en la automatización de pruebas para aplicaciones web, utilizando como caso práctico la plataforma CogniFit (<http://www.cognifit.com> ). La elección de esta plataforma no es casual; CogniFit es una aplicación web compleja que ofrece servicios de entrenamiento cognitivo, lo que la convierte en un excelente candidato para implementar diferentes tipos de pruebas automatizadas. Además, al ser una aplicación relacionada con la salud cognitiva, la calidad y fiabilidad del software son aspectos críticos que justifican la inversión en automatización de pruebas.

## Objetivos

El objetivo principal de este trabajo es diseñar e implementar un sistema de automatización de pruebas para aplicaciones web, utilizando CogniFit como objetivo. Este objetivo general se desglosa en los siguientes objetivos específicos:

1. **Analizar y seleccionar las herramientas y tecnologías** más adecuadas para la automatización de pruebas en aplicaciones web, considerando factores como la facilidad de uso, la compatibilidad con diferentes navegadores y la capacidad de integración con sistemas de CI/CD.

2. **Diseñar una arquitectura de pruebas escalable y mantenible** que permita la incorporación de nuevos casos de prueba de forma sencilla y que facilite la interpretación de los resultados.

3. **Implementar pruebas automatizadas para diferentes aspectos de la aplicación**:

**A**. Pruebas de **layout** para verificar la correcta visualización en diferentes dispositivos y resoluciones.

**B.** Pruebas **funcionales** para comprobar el funcionamiento de las principales características de la aplicación.

**C**. Pruebas de **rendimiento** para evaluar los tiempos de carga y la eficiencia de la aplicación.

**D.** Pruebas de **accesibilidad** para garantizar que la aplicación cumple con los estándares de accesibilidad web.

**4.** **Integrar el sistema de pruebas con una plataforma de CI/CD** (GitHub Actions) para automatizar la ejecución de las pruebas y facilitar la detección temprana de errores.

**5. Analizar los resultados obtenidos** y proponer mejoras tanto en el sistema de pruebas como en la aplicación CogniFit.

**6. Documentar todo el proceso** de forma clara y detallada, para que pueda servir como referencia para futuros proyectos de automatización de pruebas.

## Estructura del documento

Este documento se estructura en varios **puntos** que reflejan el proceso seguido durante el desarrollo del proyecto:

En el **Punto 2: Marco Teórico**, se presentan los fundamentos teóricos necesarios para comprender el ámbito de la automatización de pruebas. Cogemos conceptos básicos de QA y testing, se exploran los diferentes tipos de pruebas aplicables a aplicaciones web y se analizan las principales herramientas y frameworks disponibles en el mercado, centrándonos en Playwright.

El **Punto 3: Análisis y Diseño** se centra en el análisis de requisitos del proyecto y en la selección de las tecnologías más adecuadas para su implementación. También se describe la arquitectura de la solución propuesta y se detallan los casos de prueba diseñados.

En el **Punto 4: Implementación** se describe el proceso de desarrollo del sistema de pruebas automatizadas. Se explica la configuración del entorno, la estructura del proyecto y la implementación de los diferentes tipos de pruebas. También se detalla la integración con sistemas de CI/CD.

El **Punto 5: Resultados y Validación** presenta los resultados obtenidos tras la ejecución de las pruebas automatizadas. Se analizan las métricas de calidad obtenidas y se describen los problemas detectados y las soluciones propuestas.

Finalmente, en el **Punto 6: Conclusiones** se resumen los logros alcanzados, se discuten las limitaciones encontradas durante el desarrollo del proyecto y se proponen líneas futuras de trabajo.

En el último punto podemos ver la bibliografía consultada y una serie de anexos que incluyen el **código** **fuente** del proyecto, ejemplos de **informes** de pruebas y un **glosario** de términos técnicos.

# MARCO TEÓRICO

## 2.1. Fundamentos de QA (Quality Assurance) y Testing

La Garantía de Calidad (Quality Assurance o QA) es un conjunto de actividades planificadas y sistemáticas que tienen como objetivo asegurar que un producto o servicio cumple con los requisitos de calidad establecidos. En el contexto del desarrollo de software, el QA se centra en garantizar que el software cumple con las especificaciones funcionales y no funcionales, y que satisface las necesidades y expectativas de los usuarios.

El testing, por su parte, es una de las actividades fundamentales dentro del proceso de QA. Consiste en la ejecución de un programa o sistema con la intención de encontrar errores o defectos. El testing no puede demostrar la ausencia de errores, pero sí puede demostrar su presencia, lo que permite corregirlos antes de que el software llegue a los usuarios finales.

Durante mis años de estudio y mi experiencia profesional, he podido comprobar que el testing no es una actividad puntual que se realiza al final del ciclo de desarrollo, sino un proceso continuo que debe integrarse desde las primeras fases del proyecto. Esta visión del testing como parte integral del desarrollo es especialmente relevante en metodologías ágiles, donde las iteraciones son cortas y frecuentes.

Los principios fundamentales del testing, según el ISTQB (International Software Testing Qualifications Board), son:

**1.** **El testing muestra la presencia de defectos, no su ausencia**: Por muy exhaustivas que sean las pruebas, nunca podrán garantizar que el software está libre de errores.

**2.** **El testing exhaustivo es imposible**: No es viable probar todas las combinaciones posibles de entradas, condiciones y escenarios. Por ello, es necesario aplicar técnicas de análisis de riesgos para priorizar las pruebas.

**3.** **Pruebas tempranas**: Cuanto antes se detecte un defecto, menor será el coste de su corrección.

**4.** **Agrupación de defectos**: Los defectos tienden a concentrarse en determinados módulos o componentes del software.

**5.** **Paradoja del pesticida**: Si se ejecutan repetidamente las mismas pruebas, estas dejarán de encontrar nuevos defectos. Es necesario revisar y actualizar regularmente los casos de prueba.

**6.** **El testing depende del contexto**: Las pruebas deben adaptarse al tipo de software, al dominio de aplicación y a los riesgos asociados.

**7.** **La falacia de ausencia de errores**: El hecho de que las pruebas no encuentren defectos no significa que el software sea de calidad desde el punto de vista del usuario.

Estos principios han guiado mi enfoque durante el desarrollo de este proyecto, ayudándome a diseñar una estrategia de pruebas efectiva y eficiente.

## 2.2. Automatización de pruebas

La automatización de pruebas consiste en el uso de herramientas y scripts para ejecutar pruebas de forma automática, comparar los resultados obtenidos con los esperados y generar informes sin intervención humana. Esta aproximación ofrece muchas ventajas frente al testing manual, especialmente en grandes proyectos o con ciclos de desarrollo rápidos.

He podido constatar algunas de las principales ventajas de la automatización de pruebas leyendo diversos blogs, artículos y documentaciones:

**- Ahorro de tiempo y recursos:** Una vez implementadas, las pruebas automatizadas pueden ejecutarse repetidamente con un coste mínimo, liberando a los testers para tareas más creativas y de mayor valor añadido.

**- Consistencia y fiabilidad:** Las pruebas automatizadas se ejecutan siempre de la misma manera, eliminando la variabilidad de las pruebas manuales.

**- Mayor cobertura:** La automatización permite ejecutar un mayor número de pruebas en menos tiempo, lo que se traduce en una mayor cobertura del código y de los escenarios de uso.

**- Detección temprana de errores:** La integración de las pruebas automatizadas en sistemas de CI/CD permite detectar errores tan pronto como se introducen en el código.

**- Regresión eficiente:** Las pruebas de regresión, que verifican que los cambios no han afectado a funcionalidades existentes, son especialmente tediosas y repetitivas de realizar manualmente, pero se benefician enormemente de la automatización.

Sin embargo, la automatización de pruebas también presenta algunos desafíos y limitaciones:

**- Coste inicial elevado:** La implementación de pruebas automatizadas requiere una inversión inicial en tiempo y recursos que puede ser considerable.

**- Mantenimiento:** Los scripts de prueba deben actualizarse cuando cambia la aplicación, lo que puede suponer un esfuerzo significativo, es decir si cambia algún botón, se añade, se maqueta de otra forma, cambia el ID del componente, etc. Afectaría al script y habría que volver a invertir tiempo en volver a ponerlo adecuadamente.

**- Limitaciones técnicas:** No todos los aspectos de una aplicación pueden automatizarse fácilmente. Por ejemplo, las pruebas de usabilidad o de experiencia de usuario suelen requerir un uso humano real.

- **Falsa sensación de seguridad:** Una seriede pruebas automatizadas que pasa con éxito no garantiza que la aplicación esté libre de errores, solo que no se han encontrado errores en los escenarios probados.

Según los informes que he leido la clave para una automatización de pruebas exitosa sería encontrar el equilibrio adecuado entre pruebas manuales y automatizadas, y en seleccionar cuidadosamente qué pruebas automatizar en función de su frecuencia de ejecución, complejidad y estabilidad.

## 2.3. Tipos de pruebas en aplicaciones web

Las aplicaciones web presentan características específicas que las diferencian de otros tipos de software, como la variabilidad de entornos de ejecución, la dependencia de la red o la importancia de la interfaz de usuario. Estas hacen que sea necesario aplicar diferentes tipos de pruebas para garantizar su calidad.

A continuación vamos a ver las diferentes pruebas que pueden incluir estos tests automatizados:

### 2.3.1. Pruebas de Layout

Las pruebas de layout verifican que la interfaz de usuario se visualiza correctamente en diferentes dispositivos y resoluciones de pantalla. Estas pruebas son especialmente importantes en el contexto actual, donde los usuarios acceden a las aplicaciones web desde una amplia variedad de dispositivos, desde ordenadores de escritorio hasta smartphones y tablets.

Las pruebas de layout pueden incluir:

- Verificación de la correcta visualización de elementos en diferentes resoluciones.

- Comprobación de la adaptabilidad del diseño (responsive design).

- Verificación de la correcta carga y visualización de imágenes y otros elementos multimedia.

### 2.3.2. Pruebas Funcionales

Las pruebas funcionales verifican que la aplicación cumple con los requisitos funcionales establecidos. Se centran en comprobar que cada función de la aplicación opera de acuerdo con las especificaciones.

Algunos ejemplos de pruebas funcionales en aplicaciones web son:

- Verificación del correcto funcionamiento de formularios (validación de campos, envío de datos, etc.).

- Comprobación de la navegación entre páginas.

- Validación de procesos de negocio completos (por ejemplo, el proceso de registro o de compra).

- Verificación de la correcta integración con sistemas externos (APIs, bases de datos, etc.).

Las pruebas funcionales son las más comúnmente automatizadas, ya que suelen ser repetitivas y bien definidas, lo que las hace ideales para la automatización.

### 2.3.3. Pruebas de Rendimiento

Las pruebas de rendimiento evalúan cómo se comporta la aplicación bajo diferentes condiciones de carga y uso. Su objetivo es identificar cuellos de botella y problemas de rendimiento que podrían afectar a la experiencia del usuario.

Entre las pruebas de rendimiento más habituales se encuentran:

- Pruebas de carga: Evalúan el comportamiento de la aplicación bajo una carga esperada.

- Pruebas de estrés: Someten a la aplicación a condiciones extremas para determinar su punto de ruptura.

- Pruebas de velocidad: Miden los tiempos de respuesta de la aplicación para diferentes operaciones.

- Pruebas de escalabilidad: Evalúan la capacidad de la aplicación para manejar un crecimiento en el número de usuarios o en el volumen de datos.

En mi opinión creo que el rendimiento es un factor crítico para la satisfacción del usuario. En mi caso si un sitio web tarda en cargarme 5 segundos lo cierro directamente.

### 2.3.4. Pruebas de accesibilidad

Las pruebas de accesibilidad verifican que la aplicación puede ser utilizada por personas con diferentes capacidades y en diversos contextos. Estas pruebas son esenciales para garantizar que la aplicación cumple con estándares de accesibilidad como las WCAG (Web Content Accessibility Guidelines).

Algunos aspectos que se evalúan en las pruebas de accesibilidad son:

- Compatibilidad con lectores de pantalla y otras tecnologías de asistencia.

- Navegabilidad mediante teclado.

- Contraste de color adecuado.

- Presencia de textos alternativos para imágenes.

- Estructura semántica correcta del contenido.

Estas pruebas aseguran que la aplicación puede ser utilizada por todos los usuarios, independientemente de sus capacidades.

### 2.3.5. Otros tipos de pruebas

Además de los tipos mencionados, existen otras pruebas relevantes para aplicaciones web:

- Pruebas de seguridad: Evalúan la resistencia de la aplicación frente a diferentes tipos de ataques y vulnerabilidades.

- Pruebas de usabilidad: Verifican que la aplicación es intuitiva y fácil de usar.

- Pruebas de compatibilida: Comprueban que la aplicación funciona correctamente en diferentes entornos (navegadores, sistemas operativos, etc.).

- Pruebas de internacionalización y localización: Verifican que la aplicación puede adaptarse a diferentes idiomas y regiones.

En este proyecto, me he centrado principalmente en las pruebas de layout, funcionales, de rendimiento y de accesibilidad, por considerarlas las más relevantes para el caso de estudio seleccionado.

## 2.4. Herramientas y Frameworks de automatización

El mercado ofrece una amplia variedad de herramientas y Frameworks para la automatización de pruebas en aplicaciones web. La elección de la herramienta adecuada depende de diversos factores, como el tipo de aplicación, el lenguaje de programación utilizado, el presupuesto disponible o las habilidades del equipo/programador.

He analizado varios Frameworks y herramientas para elegir la más apropiada para este proyecto:

### 2.4.1. Selenium

Selenium es probablemente el framework de automatización de pruebas web más conocido y utilizado. Desarrollado inicialmente en 2004, Selenium permite controlar navegadores web de forma programática para simular la interacción de un usuario con la aplicación.

**Ventajas**:

- Amplio soporte para diferentes navegadores (Chrome, Firefox, Safari, Edge, etc.).

- Disponible en múltiples lenguajes de programación (Java, Python, C#, JavaScript, etc.).

- Mucha documentación a seguir.

- Gratuito y de código abierto.

**Desventajas:**

- Configuración inicial compleja.

- Pruebas relativamente lentas.

- Limitado a pruebas de interfaz de usuario.

### 2.4.2. Cypress

Cypress es un framework de automatización de pruebas más reciente (lanzado en 2014) que ha ganado popularidad rápidamente gracias a su enfoque innovador y su facilidad de uso.

**Ventajas:**

- Fácil configuración e instalación.

- Interfaz gráfica que facilita la depuración.

- Ejecución rápida de pruebas.

- Excelente documentación y ejemplos.

**Desventajas:**

- Soporte limitado para navegadores (principalmente Chrome y Firefox).

- Solo disponible en JavaScript.

- No soporta múltiples pestañas o dominios en la misma prueba.

- La versión gratuita tiene limitaciones en cuanto a paralelización y grabación de pruebas.

### 2.4.3. Puppeteer

Puppeteer es una biblioteca de Node.js desarrollada por Google que proporciona una API de alto nivel para controlar Chrome o Chromium mediante el protocolo DevTools.

**Ventajas:**

- Desarrollado y mantenido por Google.

- Excelente rendimiento.

- API sencilla y bien documentada.

- Capacidades avanzadas para manipular el navegador.

**Desventajas:**

- Solo funciona con Chrome/Chromium.

- Solo disponible en JavaScript y recientemente Python.

- No incluye funcionalidades específicas para testing (assertions, **report**ers, etc.).

- Menos maduro que otras alternativas.

### 2.4.4. TestCafe

TestCafe es un framework de automatización de pruebas end-to-end para aplicaciones web desarrollado por DevExpress.

**Ventajas:**

- No requiere drivers de navegador ni configuración adicional.

- Soporte para múltiples navegadores.

- API sencilla y expresiva.

- Buena gestión de esperas y sincronización.

**Desventajas:**

- Menos popular que otras alternativas, lo que se traduce en menos recursos y ejemplos disponibles (menos documentación y ayuda de la comunidad).

- Rendimiento inferior a otras herramientas en algunos escenarios.

- Limitaciones en la interacción con el DOM en comparación con Selenium o Puppeteer.

### 2.4.5. Playwright

Playwright es un framework de automatización de pruebas desarrollado por Microsoft que permite controlar Chrome, Firefox y WebKit con una única API.

**Ventajas:**

- Soporte nativo para múltiples navegadores (Chrome, Firefox, Safari).

- API moderna y potente.

- Excelente gestión de esperas y sincronización.

- Capacidades avanzadas como interceptación de red, emulación de dispositivos móviles, etc.

- Buena documentación y ejemplos.

**Desventajas:**

- Relativamente nuevo, lo que implica menos recursos y ejemplos disponibles que alternativas más establecidas.

- Solo disponible en JavaScript/TypeScript, Python y .NET.

- Curva de aprendizaje inicial para aprovechar todas sus capacidades.

Según la información que he encontrado sobre estas herramientas Playwright parece ser la más potente y versátil, que combina lo mejor de Selenium (soporte multi-navegador) y Puppeteer (API moderna y potente), además de tener una curva de aprendizaje no tan pronunciada. Su desarrollo activo por parte de Microsoft y su creciente comunidad lo convierten en una opción muy atractiva para proyectos nuevos de automatización de pruebas y es por ello que he utilizado este Framework.

## 2.5. Playwright como herramienta de automatización

Ya que Playwright es la herramienta seleccionada para este proyecto, merece un análisis más detallado de sus características y capacidades.

Playwright fue lanzado por Microsoft en enero de 2020 como una evolución de Puppeteer, con el objetivo de proporcionar una API única para controlar los tres principales motores de renderizado web: Chromium (Chrome, Edge), Firefox y WebKit (Safari).

### 2.5.1. Arquitectura de Playwright

Playwright utiliza una arquitectura cliente-servidor. El cliente es el script de prueba escrito en JavaScript, TypeScript, Python o .NET, mientras que el servidor es el proceso del navegador controlado por Playwright.

La comunicación entre el cliente y el servidor se realiza mediante el protocolo CDP (Chrome DevTools Protocol) para Chrome y Edge, y protocolos equivalentes para Firefox y WebKit. Esta arquitectura permite a Playwright ofrecer una API consistente para los tres motores de renderizado, a pesar de sus diferencias internas.

### 2.5.2. Características principales

Durante mi investigación y uso de Playwright, he identificado varias características que lo hacen especialmente adecuado para la automatización de pruebas en aplicaciones web:

**- Soporte multi-navegador:** Playwright permite ejecutar las mismas pruebas en Chrome, Firefox y Safari sin cambios en el código, lo que facilita la detección de problemas específicos de un navegador.

**- Auto-espera:** Playwright espera automáticamente a que los elementos estén listos para la interacción antes de realizar acciones sobre ellos, lo que reduce la necesidad de esperas explícitas y hace que las pruebas sean más robustas.

**- Aislamiento de contexto:** Cada prueba se ejecuta en un contexto aislado, lo que evita interferencias entre pruebas y permite ejecutarlas en paralelo de forma segura.

**- Capacidades avanzadas:**

- Interceptación y modificación de peticiones de red.

- Emulación de dispositivos móviles.

- Captura de screenshots y vídeos.

- Generación de trazas para depuración.

- Ejecución en modo headless o headed.

**- Playwright Test:** Además de la biblioteca principal, Playwright ofrece un framework de testing completo (Playwright Test) que incluye:

- Runner de tests con paralelización.

- Assertions específicas para web.

- Reportes visuales.

- Integración con CI/CD.

# 3. ANÁLISIS Y DISEÑO

## 3.1. Análisis de requisitos

Antes de comenzar con el diseño y la implementación del sistema de pruebas automatizadas, es fundamental realizar un análisis detallado de los requisitos del proyecto. Este análisis nos permitirá definir el alcance del proyecto, identificar las funcionalidades a probar y establecer los criterios de aceptación.

### 3.1.1. Requisitos funcionales

Los requisitos funcionales del sistema de pruebas automatizadas son:

1. El sistema debe permitir la ejecución de **pruebas de layout** para verificar la correcta visualización de la aplicación en diferentes dispositivos y resoluciones.

2. El sistema debe permitir la ejecución de **pruebas funcionales** para verificar el correcto funcionamiento de las principales características de la aplicación.

3. El sistema debe permitir la ejecución de **pruebas de rendimiento** para evaluar los tiempos de carga y la eficiencia de la aplicación.

4. El sistema debe permitir la ejecución de **pruebas de accesibilidad** para verificar que la aplicación cumple con los estándares de accesibilidad web.

5. El sistema debe **generar informes** detallados de los resultados de las pruebas, incluyendo capturas de pantalla y trazas para facilitar la depuración.

6. El sistema debe integrarse con GitHub Actions para la **ejecución automática** de las pruebas en cada push o pull request.

### 3.1.2. Requisitos no funcionales

Los requisitos no funcionales del sistema son:

1. El sistema debe ser fácilmente **extensible** para permitir la incorporación de nuevos casos de prueba.

2. El sistema debe ser **mantenible**, con un código limpio y bien documentado.

3. El sistema debe ser **eficiente**, minimizando el tiempo de ejecución de las pruebas.

4. El sistema debe ser robusto, **tolerando cambios menores** en la interfaz de la aplicación sin necesidad de actualizar los scripts de prueba.

5. El sistema debe ser portable, permitiendo su ejecución en **diferentes entornos** (desarrollo, CI/CD, etc.).

### 3.1.3. Análisis de la aplicación objetivo

CogniFit (<http://www.cognifit.com>) es una plataforma de entrenamiento cognitivo que ofrece juegos y evaluaciones diseñados por neurocientíficos para mejorar las capacidades cognitivas de los usuarios. La aplicación incluye:

- Una página principal con información sobre la plataforma y sus beneficios.

- Un sistema de registro y login.

- Un dashboard personalizado para cada usuario.

- Una biblioteca de juegos y evaluaciones cognitivas.

- Un sistema de seguimiento del progreso.

- Una tienda para la adquisición de planes premium.

Para este proyecto, me centraré en las siguientes áreas de la aplicación:

**1. Página principal:** Verificaré que se visualiza correctamente en diferentes dispositivos y que los elementos principales (menú, hero section, secciones informativas, footer) funcionan correctamente.

**2. Sistema de navegación:** Comprobaré que los enlaces del menú principal y del footer funcionan correctamente y llevan a las páginas esperadas.

**3. Rendimiento general:** Evaluaré los tiempos de carga de las principales páginas de la aplicación y la optimización de recursos.

**4. Accesibilidad:** Verificaré que la aplicación cumple con los criterios básicos de accesibilidad web, como la presencia de textos alternativos para imágenes, la navegabilidad mediante teclado o el contraste de color adecuado.

\*No incluiré en el alcance de este proyecto las pruebas de funcionalidades que requieren autenticación, como el dashboard o los juegos cognitivos, ya que esto complicaría significativamente la implementación y requeriría la creación y gestión de cuentas de prueba.

## 3.2. Selección de tecnologías

La selección de las tecnologías adecuadas es un paso crucial en el desarrollo de cualquier proyecto de software. En el caso de un sistema de automatización de pruebas, esta decisión afectará a aspectos como la facilidad de implementación, la mantenibilidad del código o la integración con otras herramientas.

### 3.2.1. Criterios de selección

Para seleccionar las tecnologías más adecuadas para este proyecto, he considerado los siguientes criterios:

**1. Facilidad de uso:** La herramienta debe tener una curva de aprendizaje razonable y una sintaxis clara y expresiva.

**2. Capacidades avanzadas:** La herramienta debe ofrecer capacidades avanzadas como la interceptación de peticiones de red, la emulación de dispositivos móviles o la captura de screenshots y vídeos.

**3. Integración con CI/CD:** La herramienta debe integrarse fácilmente con sistemas de CI/CD, especialmente con GitHub Actions.

**4. Comunidad y soporte:** La herramienta debe contar con una comunidad activa y una buena documentación.

### 3.2.2. Tecnologías seleccionadas

Tras evaluar diferentes opciones he seleccionado las siguientes tecnologías para este proyecto:

**1. Playwright:** Como framework de automatización de pruebas, por las razones detalladas en el punto 2. Playwright ofrece un excelente equilibrio entre capacidades avanzadas y facilidad de uso.

**2. TypeScript:** Como lenguaje de programación, por su tipado estático que proporciona mayor seguridad y facilita el mantenimiento del código. Además, TypeScript es el lenguaje nativo de Playwright y cuenta con un excelente soporte en términos de tipos y documentación.

**3. Playwright Test:** Como framework de testing, por su integración nativa con Playwright y sus capacidades de paralelización, assertions específicas para web y generación de reportes.

**4. GitHub Actions:** Como plataforma de CI/CD, por su integración nativa con GitHub (donde se alojará el código del proyecto) y su facilidad de configuración.

**5. Visual Studio Code:** Como entorno de desarrollo, por su excelente soporte para TypeScript y Playwright, y por contar con extensiones específicas que facilitan el desarrollo y depuración de pruebas.

## 3.3. Arquitectura de la solución

La arquitectura de un sistema de pruebas automatizadas debe diseñarse cuidadosamente para garantizar su escalabilidad, mantenibilidad y eficiencia. En este apartado, describo la arquitectura propuesta para el sistema de pruebas automatizadas para CogniFit.

### 3.3.1. Visión general

El sistema de pruebas automatizadas se estructura en torno a Playwright Test, que proporciona el framework de testing, y se organiza en diferentes módulos según el tipo de prueba y la funcionalidad a probar.

La arquitectura propuesta sigue un enfoque modular, con una clara separación de responsabilidades entre los diferentes componentes del sistema:

**1. Configuración:** Define los parámetros generales de las pruebas, como los timeouts o los reporters.

**2. Utilidades:** Proporciona funciones auxiliares que pueden ser utilizadas por diferentes pruebas, como la navegación a una página o la captura de métricas de rendimiento.

**3. Pruebas:** Implementan los casos de prueba específicos para cada tipo de prueba y funcionalidad.

**4. Reportes:** Generan informes detallados de los resultados de las pruebas.

**5. CI/CD:** Configura la ejecución automática de las pruebas en cada push o pull request.

### 3.3.2. Estructura de directorios

La estructura de directorios del proyecto refleja la arquitectura modular propuesta:

**ProyectoFinDeGrado-QA-Automation/**

**├── tests/**

**│ ├── accessibility.spec.ts**

**│ ├── functional.spec.ts**

**│ ├── layout.spec.ts**

**│ ├── performance.spec.ts**

**│ ├── simple-test.spec.ts**

**│ └── utils/**

**│ └── helpers.ts**

**├── playwright.config.ts**

**├── package.json**

**└── README.md**

Al principio quería hacer una estructura más compleja con carpetas separadas para cada tipo de prueba, pero me di cuenta de que para un proyecto de aprendizaje era mejor mantenerlo simple. Cada tipo de prueba está en su propio archivo, lo que hace que sea fácil de entender.

### 3.3.3. Flujo de ejecución

El flujo de ejecución de las pruebas es el siguiente:

**1. Configuración:** Playwright Test carga la configuración definida en “playwright.config.ts”, que especifica los timeouts, los reporters, etc.

**2. Búsqueda de pruebas:** Playwright Test busca todos los archivos con extensión “.spec.ts” en el directorio “tests” y sus subdirectorios.

**3. Generación de reportes:** Playwright Test genera informes detallados de los resultados de las pruebas, incluyendo capturas de pantalla y trazas para las pruebas fallidas.

**4. Integración con CI/CD:** Si las pruebas se ejecutan en GitHub Actions, los resultados se publican como artefactos y se notifica el estado de las pruebas en el pull request o commit correspondiente.

### 3.3.4. Patrones de diseño

Para organizar el código de las pruebas, he intentado seguir algunasbuenas prácticas que encontré en la documentación:

1. **Separar las pruebas por tipos:** Puse cada tipo de prueba en su propio archivo para que sea más fácil encontrar y modificar las pruebas.

2. **Usar funciones auxiliares**: Creé un archivo "helpers.ts" con funcionesque uso en varias pruebas, como esperar a que cargue la página o aceptar cookies.

3. **Comentarios claros**: Intenté poner comentarios que expliquen qué hace cada prueba, como si fuera para alguien que no sabe nada de testing.

Al principio todo estaba en un solo archivo, pero me di cuenta de que era muy difícil de manejar, así que lo separé poco a poco.

## 3.4. Diseño de casos de prueba

El diseño de casos de prueba es una fase crucial en el desarrollo de un sistema de pruebas automatizadas. Un buen diseño de casos de prueba garantiza una cobertura adecuada de las funcionalidades a probar y facilita la detección de errores.

### 3.4.1. Metodología

Para el diseño de los casos de prueba, he seguido una metodología basada en los siguientes principios:

**1. Enfoque en el usuario:** Los casos de prueba deben reflejar el comportamiento esperado desde el punto de vista del usuario, no desde el punto de vista técnico.

**2. Independencia:** Cada caso de prueba debe ser independiente de los demás, para evitar dependencias que dificulten el mantenimiento y la ejecución paralela.

**3. Claridad:** Los casos de prueba deben ser claros y expresivos, con nombres descriptivos que indiquen qué se está probando y qué resultado se espera.

**4. Minimalismo:** Los casos de prueba deben ser lo más simples posible, probando una única funcionalidad o aspecto en cada caso.

**5. Robustez:** Los casos de prueba deben ser robustos frente a cambios menores en la interfaz de la aplicación, utilizando selectores estables y estrategias de espera adecuadas.

### 3.4.2. Casos de prueba de layout

Los casos de prueba de layout verifican que la interfaz de usuario se visualiza correctamente en diferentes dispositivos y resoluciones de pantalla.

**TC-L01: El header debe mantener su estructura en desktop**

1. **Descripción**: Verificar que el header de la página principal se visualiza correctamente en resolución de escritorio.

2. **Pasos**:

1. Navegar a la página principal.

2. Verificar que el logo es visible.

3. Verificar que el menú de navegación es visible.

4. Verificar que los botones de acción principales son visibles.

3**. Resultado esperado**: Todos los elementos del header son visibles y se visualizan correctamente.

**TC-L02: El footer debe contener enlaces importantes y copyright**

1. **Descripción:** Verificar que el footer de la página principal contiene los enlaces importantes y el copyright.

2. **Pasos:**

1. Navegar a la página principal.

2. Verificar que el footer es visible.

3. Verificar que los enlaces de redes sociales son visibles.

4. Verificar que el texto de copyright es visible.

3. **Resultado esperado**: Todos los elementos del footer son visibles y se visualizan correctamente.

**TC-L03: La página debe ser responsive en dispositivos móviles**

1. **Descripción**: Verificar que la página principal se adapta correctamente a resoluciones móviles.

2. **Pasos:**

1. Configurar el viewport para móvil (375x667).

2. Navegar a la página principal.

3. Verificar que el menú hamburguesa es visible.

4. Verificar que el contenido principal se ajusta al ancho de la pantalla.

3. **Resultado esperado**: La página se visualiza correctamente en resolución móvil, con el menú hamburguesa visible y el contenido ajustado al ancho de la pantalla.

### 3.4.3. Casos de prueba funcionales

Los casos de prueba funcionales verifican que la aplicación cumple con los requisitos funcionales establecidos.

**TC-F01: Los enlaces del menú principal deben funcionar correctamente**

1. **Descripción**: Verificar que los enlaces del menú principal llevan a las páginas esperadas.

2. **Pasos**:

1. Navegar a la página principal.

2. Para cada enlace del menú principal:

a. Hacer clic en el enlace.

b. Verificar que la navegación lleva a la página esperada.

c. Volver a la página principal.

3. **Resultado esperado:** Todos los enlaces del menú principal llevan a las páginas esperadas.

**TC-F02: El buscador debe mostrar resultados relevantes**

1. **Descripción**: Verificar que el buscador muestra resultados relevantes para una búsqueda.

2. **Pasos**:

1. Navegar a la página principal.

2. Localizar el campo de búsqueda.

3. Introducir un término de búsqueda relevante (por ejemplo, "memoria").

4. Pulsar Enter.

5. Verificar que se muestran resultados.

3**. Resultado esperado**: Se muestran resultados relevantes para el término de búsqueda.

### 3.4.4. Casos de prueba de rendimiento

Los casos de prueba de rendimiento evalúan cómo se comporta la aplicación en términos de velocidad y eficiencia.

**TC-P01: La página principal debe cargarse en menos de 5 segundos**

1. **Descripción**: Verificar que la página principal se carga en un tiempo razonable.

2. **Pasos**:

1. Medir el tiempo de inicio.

2. Navegar a la página principal.

3. Esperar a que la página esté completamente cargada.

4. Medir el tiempo de fin.

5. Calcular el tiempo de carga.

3. **Resultado esperado**: El tiempo de carga debe ser inferior a 5 segundos.

**TC-P02: Capturar métricas de rendimiento de páginas principales**

1. **Descripción**: Capturar y verificar métricas de rendimiento para las páginas principales.

2. **Pasos**:

1. Para cada página principal (/, /login, /about, /brain-games):

a. Navegar a la página.

b. Capturar métricas de rendimiento (tiempo de carga, tiempo de renderizado, etc.).

c. Verificar que las métricas están dentro de los límites aceptables.

3. **Resultado esperado:** Todas las métricas están dentro de los límites aceptables (tiempo de carga total < 8 segundos, tiempo de renderizado < 5 segundos).

**TC-P03: Los recursos estáticos deben estar optimizados**

1. **Descripción**: Verificar que los recursos estáticos (imágenes, CSS, JavaScript) están optimizados.

2. **Pasos**:

1. Navegar a la página principal.

2. Analizar el tamaño y la optimización de las imágenes.

3. Verificar que no hay imágenes excesivamente grandes mostradas en tamaños pequeños.

3. **Resultado esperado**: Todos los recursos estáticos están optimizados, con imágenes de tamaño adecuado para su visualización.

### 

### 3.4.5. Casos de prueba de accesibilidad

Los casos de prueba de accesibilidad verifican que la aplicación puede ser utilizada por personas con diferentes capacidades y en diversos contextos.

**TC-A01: La página principal debe cumplir criterios básicos de accesibilidad**

1. **Descripción**: Verificar que la página principal cumple con criterios básicos de accesibilidad.

2. **Pasos**:

1. Navegar a la página principal.

2. Verificar que todas las imágenes tienen atributo alt.

3. Verificar que hay un solo elemento h1.

4. Verificar que los inputs tienen label asociado.

3. **Resultado esperado**: La página cumple con los criterios básicos de accesibilidad, con un máximo de 3 problemas detectados.

**TC-A02: Los formularios deben tener etiquetas y ser accesibles**

1. **Descripción**: Verificar que los formularios tienen etiquetas y son accesibles.

2. **Pasos**:

1. Navegar a la página de login.

2. Verificar que todos los inputs tienen label asociado, aria-label o placeholder.

3. **Resultado esperado**: Todos los inputs tienen al menos una forma de accesibilidad (label, aria-label o placeholder).

**TC-A03: El contraste de color debe ser adecuado para elementos principales\***

1. **Descripción**: Verificar que el contraste de color es adecuado para los elementos principales.

2. **Pasos**:

1. Navegar a la página principal.

2. Analizar el contraste de color de los textos principales.

3. **Resultado esperado**: El contraste de color es adecuado para los elementos principales, cumpliendo con los criterios WCAG 2.1 AA.

Estos casos de prueba cubren los aspectos más relevantes de la aplicación CogniFit, desde el punto de vista del layout, la funcionalidad, el rendimiento y la accesibilidad.

# IMPLEMENTACIÓN

## 4.1. Configuración del entorno

La configuración del entorno de desarrollo es el primer paso en la implementación del sistema de pruebas automatizadas. Un entorno bien configurado facilita el desarrollo, la ejecución y el mantenimiento de las pruebas.

### 4.1.1. Requisitos previos

Para la implementación del sistema de pruebas automatizadas, he utilizado los siguientes requisitos previos:

**- Node.js:** Como entorno de ejecución para JavaScript. He utilizado la versión 16.14.0.

**- npm:** Como gestor de paquetes para instalar las dependencias del proyecto. npm viene incluido con Node.js.

**- Visual Studio Code:** Como entorno de desarrollo integrado (IDE). Visual Studio Code cuenta con extensiones específicas que facilitan el desarrollo y depuración de pruebas.

**- Git:** Para gestionar el código del proyecto y facilitar la colaboración.

### 4.1.2. Inicialización del proyecto

Para inicializar el proyecto, hemos seguido estos pasos:

1. **Crear el directorio del proyecto:**

* mkdir ProyectoFinDeGrado-QA-Automation
* cd ProyectoFinDeGrado-QA-Automation

1. **Inicializar el proyecto npm**:

* npm init -y

Este comando crea un archivo “package.json” con la configuración básica del proyecto.

1. **Inicializar el repositorio Git:**

* git init

Este comando crea un repositorio Git vacío en el directorio actual.

1. **Crear un archivo “.gitignore**”:

* echo "node\_modules/
* /test-results/
* /playwright-report/
* /playwright/.cache/
* .env" > .gitignore

Este archivo especifica qué archivos y directorios deben ser ignorados por Git.

### Instalación de dependencias

Para la implementación del sistema de pruebas automatizadas, he instalado las siguientes dependencias:

1. **Playwright**:

- npm init playwright@latest

Este comando instala Playwright y crea la configuración básica del proyecto, incluyendo el archivo “playwright.config.ts” y un test de ejemplo.

Durante la instalación, he seleccionado las siguientes opciones:

1. TypeScript: Sí

2. Directorio para los tests: tests

3. Flujo de trabajo de GitHub Actions: Sí

4. Instalar navegadores de Playwright: Sí

2. **TypeScript**:

* npm install --save-dev typescript

Este comando instala TypeScript como dependencia de desarrollo.

Al principio pensé en instalar ESLint para mantener el código limpio, pero decidí centrarme primero en hacer funcionar las pruebas básicas. Para un proyecto de aprendizaje como este, preferí mantenerlo simple.

Este comando instala ESLint y los plugins necesarios para TypeScript, que ayudarán a mantener un código limpio y consistente.

### Configuración de Playwright

La configuración de Playwright se realiza en el archivo `playwright.config.ts`, que se crea automáticamente durante la instalación. He personalizado esta configuración para adaptarla a las necesidades del proyecto

Esta configuración incluye:

**- testDir:** Directorio donde se encuentran los tests.

**- timeout:** Tiempo máximo de ejecución para cada test (30 segundos).

**- fullyParallel:** Ejecutar los tests en paralelo.

**- forbidOnly:** Prohibir los tests marcados como `only` en CI.

**- retries:** Número de reintentos para tests fallidos (2 en CI, 0 en desarrollo).

**- workers:** Número de workers para la ejecución paralela (1 en CI, automático en desarrollo).

**- reporter:** Configuración de los reporters (HTML y JSON).

**- use:** Configuración global para todos los tests:

**- baseURL:** URL base para la navegación.

**- trace:** Capturar trazas solo en el primer reintento.

**- screenshot:** Capturar screenshots solo en caso de fallo.

**- video:** Grabar vídeo solo en el primer reintento.

- **projects**: Configuración de los diferentes navegadores y dispositivos a probar.

### 4.1.5. Configuración de scripts “npm”

Para facilitar la ejecución de las pruebas, he configurado varios scripts en el archivo “package.json”:

"**scripts**": {

"test": "playwright test",

"test:headed": "playwright test --headed",

"test:ui": "playwright test --ui",

"report": "playwright show-report"

}

Estos scripts me permiten:

1- **test:** Ejecutar todas las pruebas.

2- **test:headed**: Ejecutar las pruebas viendo el navegador (útil para depurar).

3- **test:ui:** Abrir la interfaz gráfica de Playwright para ver las pruebas.

4- **report**: Ver el informe de resultados después de ejecutar las pruebas.

Al principio solo usé el comando básico "test", pero después descubrí que "**test:headed**" me ayudaba mucho para entender qué estaba pasando cuando las pruebas fallaban.

### Configuración adicional

## Inicialmente pensé en configurar GitHub Actions para ejecutar las pruebas automáticamente, pero decidí centrarme primero en hacer que las pruebas funcionaran correctamente en local. Para un proyecto de aprendizaje, ejecutar las pruebas manualmente me ayuda a entender mejor lo que está pasando. En el futuro me gustaría añadir integración continua, pero por ahora prefiero ejecutar las pruebas paso a paso para aprender cómo funcionan.

# RESULTADOS Y VALIDACIÓN

## Ejecución de pruebas

Las pruebas se ejecutaron localmente usando el comando `npx playwright test`.

Se ejecutaron un total de 12 pruebas divididas en 4 categorías:

- Accesibilidad: 3 pruebas

- Funcionalidad: 3 pruebas

- Layout: 3 pruebas

- Rendimiento: 3 pruebas

### 5.1.1. Ejecución local

Para ejecutar las pruebas en un entorno local, he utilizado los scripts definidos en el archivo `package.json`:

La ejecución local de las pruebas proporciona varias ventajas:

1. **Rapidez**: La ejecución local suele ser más rápida que en CI/CD, lo que facilita el desarrollo y la depuración.

2. **Interactividad**: El modo debug permite interactuar con el navegador durante la ejecución de las pruebas, lo que facilita la depuración de problemas.

3. **Flexibilidad**: Es posible ejecutar solo un subconjunto de las pruebas o ejecutarlas en un navegador específico, lo que facilita el desarrollo y la depuración.

### 5.1.2. Ejecución manual

Para este proyecto, todas las pruebas las ejecuté manualmente desde mi ordenador usando los comandos de npm. Aunque sé que existe la posibilidad de automatizar esto con GitHub Actions, preferí hacerlo paso a paso para entender mejor cómo funcionan las pruebas.

Ejecutar las pruebas manualmente me permitió:

**1.** Ver exactamente qué estaba pasando cuando algo fallaba

**2.** Aprender a interpretar los mensajes de error

**3.** Entender mejor cómo Playwright interactúa con la página web

En el futuro me gustaría aprender a configurar la ejecución automática, pero para empezar, hacerlo manual fue la mejor opción.

## 5.2. Análisis de resultados

El análisis de los resultados de las pruebas es fundamental para identificar problemas y áreas de mejora en la aplicación. En este apartado, analizo los resultados obtenidos en cada tipo de prueba. Los resultados fueron:

- 9 pruebas exitosas (75%)

- 3 pruebas fallidas (25%)

- Tiempo total: 54.7 segundos

### 5.2.1. Análisis de pruebas de layout

- ❌ Logo no encontrado: Los selectores no localizaron el logo de CogniFit

- ✅ Responsive móvil: La página se adapta bien a pantallas pequeñas

- ❌ Elementos ocultos: Algunos elementos del menú están ocultos

### 5.2.2. Análisis de pruebas de rendimiento

- ✅ Velocidad excelente: Página carga en 957ms (muy rápido)

- ✅ Imágenes optimizadas: 38 imágenes cargan correctamente

- ✅ Métricas técnicas: Datos obtenidos en 736ms

### 5.2.3. Análisis de pruebas de accesibilidad

- ✅ Títulos bien estructurados: Se encontraron 37 títulos organizados correctamente

- ❌ Formularios sin descripciones: 34 cajas de texto no tienen etiquetas adecuadas

- ✅ Navegación con teclado: 376 elementos son accesibles por teclado

### 5.2.4 Análisis de pruebas de funcionalidad

- ✅ Menú principal: 46 enlaces funcionan correctamente

- ❌ Botón principal: No se encontró el botón "Comenzar" esperado

- ✅ Enlaces totales: 281 enlaces detectados en la página

## 5.3. Problemas detectados y soluciones

### 5.3.1. Problemas principales detectados

1. **Formularios sin etiquetas de accesibilidad**

- Problema: 34 inputs no tienen aria-label, placeholder o name

- Impacto: Dificulta el uso para personas con discapacidades

- Solución recomendada: Añadir etiquetas descriptivas

2. **Elementos de navegación ocultos**

- Problema: El menú principal tiene elementos con visibility: hidden

- Impacto: Confunde a las pruebas automatizadas

- Solución: Revisar CSS del menú de navegación

3. **Selectores de logo incorrectos**

- Problema: Los selectores usados no encuentran el logo

- Impacto: Prueba falla, aunque el logo existe

- Solución: Actualizar selectores en las pruebas

## 5.4. Métricas de calidad obtenidas

- **Tasa de éxito:** 75% (aceptable para un primer intento)

- **Velocidad de carga:** 957ms (excelente, muy por debajo del límite de 5s)

- **Elementos accesibles:** 376 elementos navegables por teclado

- **Cobertura de enlaces:** 281 enlaces verificados automáticamente

## 5.5 Conclusiones del análisis

El sitio web de CogniFit muestra un **rendimiento** **excelente** con tiempos de carga muy rápidos.

La **funcionalidad básica funciona** bien con navegación correcta entre páginas. Las principales áreas de mejora están en:

1. **Accesibilidad**: Mejorar etiquetas de formularios

2. **Layout**: Revisar elementos ocultos en el menú

3. **Pruebas**: Actualizar selectores para mayor precisión

En general, la página cumple con los requisitos básicos de funcionalidad y rendimiento, pero necesita mejoras en accesibilidad para ser completamente inclusiva.

## 5.6. Dificultades encontradas en el proyecto

Durante el desarrollo del proyecto me encontré con varios problemas que no esperaba al principio:

1. **Configuración inicial de Playwright:** Al principio me salían errores como "test.describe() not expected" que no entendía. Tardé varios días en darme cuenta de que tenía problemas con la configuración de TypeScript.

2. **Selectores que no funcionaban:** Muchas de mis primeras pruebas fallaron porque los selectores que usé no encontraban los elementos correctos de la página. Tuve que aprender a inspeccionar la página web para encontrar los selectores adecuados.

3. **Elementos ocultos en el menú:** Algunas pruebas fallaban porque había elementos del menú que estaban ocultos con CSS, pero yo no lo sabía. Las pruebas decían que no encontraban el elemento, pero yo lo veía en la página.

4. **Aprender TypeScript sobre la marcha**: Como no tenía mucha experiencia con TypeScript, tuve que aprender la sintaxis mientras hacía las pruebas. Al principio me costaba entender los tipos y las interfaces.

5. **Interpretar los errores:** Los mensajes de error de Playwright al principio me parecían muy confusos. Tuve que aprender a leer los logs y entender qué significaba cada tipo de error.

Estos problemas me ayudaron a entender que hacer pruebas automatizadas requiere mucha paciencia y práctica. No es tan fácil como pensé al principio.

# 6. CONCLUSION

El desarrollo de este proyecto me ha servido para conocer paso a paso como de importante son las pruebas de QA desde un punto de vista profesional y qué sería de una empresa sin ellas. He podido investigar y aprender los diferentes aspectos a tener en cuenta cuando desarrollamos aplicaciones y ver todo desde una vista de usuario, de la cual nos olvidamos muchas veces y por eso salen aplicaciones sin usabilidad e inútiles. Además poder trabajar con Github y ver cómo crear el proyecto desde 0 me ha motivado mucho a seguir aprendiendo a pesar de haber tenido que pasar bastantes días leyendo documentación sin parar.

# 7. BIBLIOGRAFÍA

Amodeo, E. (2015). Learning Selenium Testing Tools with Python. Packt Publishing.

Axelrod, A. (2018). Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects. Apress.

Crispin, L., & Gregory, J. (2014). Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley Professional.

Dustin, E., Garrett, T., & Gauf, B. (2009). Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality. Addison-Wesley Professional.

Huizinga, D., & Kolawa, A. (2007). Automated Defect Prevention: Best Practices in Software Management. Wiley-IEEE Computer Society Press.

International Software Testing Qualifications Board (ISTQB). (2018). Certified Tester Foundation Level Syllabus.

Kaner, C., Bach, J., & Pettichord, B. (2001). Lessons Learned in Software Testing: A Context-Driven Approach. Wiley.

Krug, S. (2014). Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability. New Riders.

Li, Y. F., Das, P. K., & Dowe, D. L. (2014). Two decades of Web application testing—A survey of recent advances. Information Systems, 43, 20-54.

Microsoft. (2023). Playwright Documentation. [https://playwright.dev/docs/intro](https://playwright.dev/docs/intro)

Patton, R. (2005). Software Testing. Sams Publishing.

Selenium Project. (2023). Selenium Documentation. [https://www.selenium.dev/documentation/en/](https://www.selenium.dev/documentation/en/)

Sommerville, I. (2015). Software Engineering. Pearson.

Web Accessibility Initiative (WAI). (2018). Web Content Accessibility Guidelines (WCAG) 2.1. W3C.

# 8. ANEXOS

## 8.1. Código fuente

El código fuente completo del proyecto está disponible en el repositorio de GitHub: [https://github.com/tu-usuario/proyectofindegrado-qa-automation](<https://github.com/tu-usuario/proyectofindegrado-qa-automation>)

## 8.3. Glosario de términos

**Automatización de pruebas**: Uso de herramientas y scripts para ejecutar pruebas de forma automática, comparar los resultados obtenidos con los esperados y generar informes sin intervención humana.

**CI/CD:** Integración Continua/Entrega Continua. Conjunto de prácticas que permiten integrar y entregar código de forma continua y automatizada.

**GitHub Actions**: Plataforma de CI/CD integrada en GitHub, que permite automatizar flujos de trabajo como la ejecución de pruebas o el despliegue de aplicaciones.

**Layout**: Disposición de los elementos en una página web, incluyendo su posición, tamaño y relación con otros elementos.

**Playwright**: Framework de automatización de pruebas desarrollado por Microsoft, que permite controlar Chrome, Firefox y WebKit con una única API.

**QA**: Quality Assurance o Garantía de Calidad. Conjunto de actividades planificadas y sistemáticas que tienen como objetivo asegurar que un producto o servicio cumple con los requisitos de calidad establecidos.

**Responsive design**: Diseño web que se adapta a diferentes tamaños de pantalla y dispositivos, proporcionando una experiencia de usuario óptima en todos ellos.

**Testing**: Proceso de ejecución de un programa o sistema con la intención de encontrar errores o defectos.

**TypeScript**: Superconjunto de JavaScript que añade tipado estático opcional y otras características a JavaScript.

**WCAG**: Web Content Accessibility Guidelines. Conjunto de recomendaciones para hacer el contenido web más accesible para personas con discapacidades.

**WebKit**: Motor de renderizado web utilizado por Safari y otros navegadores basados en él.

**Accesibilidad web**: Práctica de diseñar y desarrollar sitios web que pueden ser utilizados por personas con diferentes capacidades y en diversos contextos.

**API**: Application Programming Interface o Interfaz de Programación de Aplicaciones. Conjunto de reglas y protocolos que permiten la comunicación entre diferentes componentes de software.

**Assertions**: Declaraciones que verifican que una condición es verdadera. En el contexto de las pruebas, las assertions se utilizan para verificar que el comportamiento real del sistema coincide con el comportamiento esperado.

**BDD**: Behavior-Driven Development o Desarrollo Dirigido por Comportamiento. Enfoque de desarrollo de software que se centra en el comportamiento esperado del sistema desde el punto de vista del usuario.

**DOM**: Document Object Model o Modelo de Objetos del Documento. Representación en memoria de un documento HTML, que permite a los scripts acceder y manipular el contenido, la estructura y el estilo del documento.

**E2E**: End-to-End o Extremo a Extremo. Tipo de prueba que verifica el flujo completo de una aplicación, desde la interfaz de usuario hasta la base de datos y de vuelta.

**Headless**: Modo de ejecución de un navegador sin interfaz gráfica, útil para la automatización de pruebas en entornos de CI/CD.

**Lazy loading**: Técnica de carga diferida de recursos, que consiste en cargar solo los recursos necesarios en el momento en que se necesitan, para mejorar el rendimiento de la aplicación.

**Minificación**: Proceso de eliminación de caracteres innecesarios (espacios, saltos de línea, comentarios) de un código fuente, para reducir su tamaño y mejorar el tiempo de carga.

**Paralelización**: Ejecución simultánea de múltiples pruebas, para reducir el tiempo total de ejecución.

**Selector**: Expresión que identifica uno o más elementos en el DOM, utilizada para interactuar con elementos específicos durante las pruebas.

**TDD**: Test-Driven Development o Desarrollo Dirigido por Pruebas. Enfoque de desarrollo de software que consiste en escribir primero las pruebas y luego el código que las hace pasar.

**Viewport**: Área visible de una página web en el navegador, que puede variar según el dispositivo y la resolución de pantalla.

**WebDriver**: Protocolo que permite controlar navegadores web de forma programática, utilizado por herramientas como Selenium.

## 