UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Automatización de pruebas para aplicaciones web con Cypress

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: de Martinis Pacifico, Mario

Tutor/a: Villanueva García, Alicia

CURSO ACADÉMICO: 2021/2022

Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Automatización de pruebas para aplicaciones web con Cypress

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Mario De Martinis Pacifico

Tutor: Alicia Villanueva García

Curso 2021-2022

Resumen

En la actualidad los proyectos crecen con gran velocidad, muchas veces se ve comprometida la calidad del producto. La necesidad de cumplir con los requisitos, estándares de calidad y las fechas de entrega nos obliga a buscar una alternativa a las pruebas manuales. Debido a esto se opta por automatizar las pruebas, de manera que nos permita ahorrar tiempo y ser más eficientes. Así pues, en este TFG se busca definir una metodología para la integración de la automatización de pruebas en desarrollos de aplicaciones web en proyectos que sigan una metodología ágil. Se pretende desarrollar una metodología capaz de detectar los defectos tanto en funcionalidades antiguas como en nuevas funcionalidades. Se analizará los beneficios de la automatización mediante la implementación de esta metodología con Cypress como herramienta para automatizar las pruebas.

Palabras clave: Automatización del testing; Cypress; Aplicación web; Desarrollo ágil.

Abstract

Projects are currently growing rapidly, and the product quality is often compromised. The need to meet requirements, quality standards and delivery dates forces us to look for an alternative to manual testing. Due to this, it is decided to automate tests, which would allow us to save time and be more efficient. Thus, this TFG seeks to define a methodology for the integration of test automation in web application development in projects that follow an agile methodology. The aim is to develop a methodology capable of detecting defects in both old and new features. The benefits of automation will be analyzed by implementing this methodology with Cypress as a tool to automate tests.

Keywords: Test automation, Cypress, Web Application, Agile development.

Resum

En l'actualitat els projectes creixen amb gran velocitat, moltes vegades es veu compromesa la qualitat del producte. La necessitat de complir amb els requisits, estàndards de qualitat i les dates de lliurament ens obliga a buscar una alternativa a les proves manuals. A causa d'això s'opta per automatitzar les proves, de manera que ens permeta estalviar temps i ser més eficients. Així doncs, en aquest TFG es busca definir una metodologia per a la integració de l'automatització de proves en desenvolupaments d'aplicacions web en projectes que seguisquen una metodologia àgil. Es pretén desenvolupar una metodologia capaç de detectar els defectes tant en funcionalitats antigues com en noves funcionalitats. S'analitzarà els beneficis de l'automatització mitjançant la implementació d'aquesta metodologia amb Cypress com a eina per a automatitzar les proves.

Paraules clau: Automatització del testing; Cypress; Aplicació web; Desenvolupament àgil.

Prólogo

En el TFG presentado a continuación titulado “Automatización de pruebas para aplicaciones web”, se estudia y analiza el ciclo de vida del software para incorporar el diseño e implementación de pruebas automatizadas de una aplicación web, mediante una metodología que se acople al desarrollo ágil.

El proyecto se llevó a cabo durante el período en el cual realicé mis prácticas en el departamento de calidad de NTTDATA. En este período fui responsable de investigar las distintas herramientas que nos permitirían llevar a cabo este proceso, así como también de mejorar la metodología que se adaptara a un proyecto ágil. En el TFG se describe el proceso de investigación de los distintos tipos de pruebas que existen, además de distintas herramientas que nos permiten automatizar las pruebas de un producto software.

Tabla de contenidos

Contenido

1. Introducción 10
1.1 Motivación 10
1.2 Objetivos 10
1.3 Impacto esperado 11
1.4 Estructura del proyecto 11
2. Pruebas de software 13
2.1 Ciclo de vida del software 13
2.2 Pruebas de software 14
3. Estudio estratégico 15
3.1 Comparativa entre herramientas 15
3.1.1 Selenium 15
3.1.2 Cypress 17
3.1.3 Conclusiones de la comparación 19
4. Diseño de la solución 21
4.1 Metodología 21
4.1.1 Pruebas de humo 21
4.1.2 Pruebas de regresión 22
4.1.3 Desarrollo de las pruebas 22
4.1.4 Escenarios de ejecución 23
4.2 Tecnologías utilizadas 23
4.2.1 GitHub 24
4.2.2 Jira 24
4.3 Estructura de Cypress 24
4.4 Estructura de una prueba 27
5. Automatización de pruebas con Cypress 29
5.1 Diseño de las pruebas 30
5.2 Implementación de las pruebas 31
5.3 Buenas prácticas 39
6. Ejecución de pruebas 41
6.1 Tipos de ejecución 41

6.2 Resultados obtenidos ... 46
6.3 Dashboard de Cypress ... 47
7. Conclusiones ... 49
Trabajo futuro ... 50
Relación del trabajo con los estudios cursados ... 51
8. Referencias ... 53
9. Anexo I – Figuras complementarias ... 54
10. Anexo II - Objetivos de desarrollo sostenible ... 57
11. Glosario ... 59

Tabla de figuras

Figura 1: fases del ciclo de vida del software. Obtenida en [2]......14
Figura 2: Resumen de todo lo que incluye Cypress. Obtenida en [6]......17
Figura 3 Relación entre cantidad de Github Stars en el tiempo. Obtenida en [7]......20
Figura 4: Cantidad de descargas semanales de Cypress y Selenium webdriver. Obtenida el día 15/06/2022......20
Figura 5: Ejemplo de tablero scrum. Obtenida en [10]......24
Figura 6: Estructura de Cypress......25
Figura 7: Archivo e2e.js......26
Figura 8: Archivo de configuración cypress.config.js......26
Figura 9: Estructura de una prueba en Cypress......27
Figura 10: Cláusulas útiles para desarrollar pruebas......28
Figura 11: Identificadores de tipo de prueba......32
Figura 12: Archivo e2e.js......32
Figura 13: Cláusula “before()” del CU-01......33
Figura 14: Cláusula beforeEach() del CU-01......33
Figura 15: Prueba de humo CU-01 parte 1......34
Figura 16: Prueba de humo CU-01 parte 2......35
Figura 17: Pruebas de regresión CU-01 parte 1......35
Figura 18: Pruebas de regresión CU-01 parte 2......36
Figura 19: Cláusula “after()” CU-01......36
Figura 20: Prueba de humo CU-02 Añadir dirección......37
Figura 21: Prueba de humo CU-02 Eliminar una dirección......38
Figura 22: Mensaje de error campo obligatorio......38
Figura 23: Prueba de regresión CU-02 Error en la API......39
Figura 24: Scripts para la ejecución de las pruebas......41
Figura 25: Elección de navegador para ejecutar las pruebas.......42
Figura 26: Menú interfaz de usuario de Cypress.......42
Figura 27: Interfaz de usuario durante la ejecución de una prueba......43
Figura 28: Llamada a la api para obtener un producto CU-01......43
Figura 29: Resumen de ejecución mediante terminal......44
Figura 30: Resultado de un banco de prueba ejecutado por terminal......44
Figura 31: Prueba fallida durante ejecución por terminal......45
Figura 32: Resultados de ejecución por terminal......45
Figura 33: Analíticas del Dashboard de Cypress......47
Figura 34: Resultado ejecución pruebas Dashboard parte 1......47
Figura 35: Histórico de ejecución......48
Figura 36: metodología durante la fase de Pruebas......49
Figura 37: Páginas para implementar el patrón Page Object Model......51
Figura 38: Archivo directions-page.js......52
Figura 39: Cláusula before() utilizando el patrón Page Object Model......52

Tabla de tablas

TABLA 1: RESUMEN COMPARATIVA DE HERRAMIENTAS 19
TABLA 2: TIPO DE PRUEBA SEGÚN EL ESCENARIO 23
TABLA 3 CU-01 AGREGAR PRODUCTOS AL CARRITO 30
TABLA 4: CU-02 AÑADIR/BORRAR DIRECCIÓN DE ENTREGA 31

1. Introducción

1.1 Motivación

Actualmente la mayoría de las empresas ofrecen servicios a través de internet, lo que provoca que las aplicaciones web estén en constante desarrollo y/o crecimiento. Las aplicaciones web son aquellas a las que los usuarios pueden acceder a través de un servidor web o un navegador. Con el fin de poder asegurar que las aplicaciones tengan la menor cantidad de defectos posible y que cumplan con los requisitos acordados con el cliente, nacen las pruebas del software que nos permiten garantizar la calidad de las aplicaciones.

En la actualidad los proyectos crecen con mucha velocidad, lo que ocasiona que la calidad se vea comprometida en muchos casos. Debido a esto nace la figura del ingeniero de QA, el cual se encarga de monitorear cada fase del proceso software y es el que debe asegurar que el diseño y el software cumplen con los estándares de la compañía.

La gran mayoría de las aplicaciones están sujetas a constantes cambios y mejoras. Pueden incluir desde la mejora de un proceso hasta el desarrollo de nuevas funcionalidades. Debido a los constantes cambios todos los proyectos son susceptibles a la introducción de defectos en funcionalidades ya probadas. Para reducir este problema existen distintos tipos de pruebas que se pueden realizar.

Cuando los proyectos son muy extensos los ingenieros de QA suelen invertir mucho tiempo en realizar las pruebas manuales repetitivas, lo cual es muy costoso ya que consume mucho tiempo y recursos. La incorporación de la automatización en el proceso y gestión de las pruebas nos permite realizar los distintos tipos de pruebas de una manera mucho más eficiente y rápida.

1.2 Objetivos

El objetivo principal de este proyecto es definir una metodología la cual se pueda aplicar en cualquier proyecto software que esté desarrollando una aplicación web. Abarcará todo el ciclo de las pruebas, desde el diseño, la automatización y la ejecución de las pruebas junto con su análisis posterior.

Además, otro objetivo es determinar un conjunto de herramientas que nos permitan implementar esta metodología. Tras analizar distintas herramientas se opta por la automatización utilizando Cypress[1], la cual se explicará en capítulos posteriores.

[1] Cypress – Página oficial: https://www.cypress.io/

1.3 Impacto esperado

Para cuantificar el impacto que puede generar la automatización de las pruebas debemos comprender lo que ocurre cuando no se realiza. El *testing* manual o prueba manual es aquel tipo de prueba en el cual los *testers* ejecutan manualmente los casos de uso sin el uso de ningún tipo de herramientas.

Este tipo de pruebas requieren de mucho tiempo y esfuerzo, además de que se requiere contar con un profesional encargado para esta labor. Este tipo de pruebas están sujetas a los errores que un humano puede cometer bien sea errores tipográficos u omitir pasos durante la prueba.

Otro factor que resalta la importancia de pruebas automatizadas es el proceso de cambio de una metodología tradicional a una metodología ágil. Esta última se ha encargado de reducir el ciclo de desarrollo software, lo que antes podía durar meses ahora solo dura días, esto ocasiona la necesidad de poder automatizar las pruebas para ejecutarlas cuando sean necesarias y reducir el tiempo y el esfuerzo.

Hay muchos beneficios de automatizar las pruebas de los cuales destacan reducir el tiempo y esfuerzo requerido en pruebas de regresión. Estas pruebas que suponen una gran cantidad de horas invertidas por parte del *tester*. Además, suelen ser muy repetitivas, y deben realizarse tras cambios en el sistema. Otro beneficio es que minimiza los errores: las pruebas automatizadas ejecutan los mismos pasos una y otra vez con la misma precisión, lo que evita que los *testers* pasen por alto errores debido a la monotonía de las pruebas manuales.

Teniendo en cuenta lo mencionado anteriormente se espera que esta metodología junto con los tipos de pruebas que propondremos en este TFG reduzca significativamente los costes del *testing* y los errores no detectados.

1.4 Estructura del proyecto

El trabajo se dividirá en los siguientes capítulos:

- **Capítulo 2:** en este se hace una breve introducción al ciclo de vida del software y a los distintos tipos de prueba existentes.

- **Capítulo 3:** se presentarán las herramientas que se tomaron en cuenta durante el proceso de investigación, además de las ventajas, desventajas y diferencias entre ellas.

- **Capítulo 4:** se introducirá la metodología propuesta, así como también las herramientas utilizadas durante el proyecto y la estructura que debe seguir una prueba en Cypress.

- **Capítulo 5:** se realiza la automatización de los distintos tipos de prueba que se implementarán debido a la metodología propuesta.

- **Capítulo 6:** se introducen las distintas opciones para ejecutar nuestras pruebas automatizadas y las herramientas disponibles para analizar sus resultados.

- **Capítulo 7:** se recopilan las conclusiones obtenidas de la investigación y de la implementación de la metodología durante las prácticas en empresa. Se presenta el trabajo a realizar en el futuro. Además, se desglosan las asignaturas que guardan relación directa con este trabajo.

- **Bibliografía**.

- **Anexo I:** en este se introducen todas las figuras complementarias mencionadas en el TFG.

- **Anexo II:** se indaga en la relación que tiene este proyecto con los objetivos de desarrollo sostenible.

2. Pruebas de software

Para introducir el proceso de automatización de pruebas primero debemos entender ciertos conceptos básicos, que nos permiten comprender cómo se integran las pruebas dentro del proceso de desarrollo de software, qué son las pruebas de software y los tipos de prueba más relevantes para este TFG.

2.1 Ciclo de vida del software

El ciclo de vida del software es el proceso que contiene las tareas relacionadas con el desarrollo y el mantenimiento de un producto software *[1]*. En éste se abarca todo el ciclo de vida del sistema, desde la definición de requisitos hasta la finalización de su uso. Este ciclo contiene distintas actividades que nos permiten garantizar que el software cumple con las funcionalidades requeridas por el cliente además de asegurarse que cumple ciertos estándares de calidad.

El ciclo de vida ágil es iterativo como se muestra en la Figura 1 y sus etapas son las siguientes:

- **Determinar el alcance del proyecto (Plan):** en esta fase se definen los sistemas a diseñar y determinan el alcance, lo que permite establecer la cantidad de recursos necesarios.

- **Diseño (Design):** esta fase implica tomar los requisitos para diseñar una solución conceptual y posteriormente un diseño técnico.

- **Desarrollo (Develop):** en esta fase se desarrolla el producto en base a las fases realizadas anteriormente.

- **Pruebas (Test):** mediante pruebas de software se determina si el diseño cumple con el conjunto de requisitos.

- **Entrega (Release):** esta fase realiza la puesta en producción del sistema una vez desarrollado y probado.

- **Retroalimentación (Feedback):** el cliente realiza comentarios sobre el producto.

Figura 1: fases del ciclo de vida del software. Obtenida en [2]

2.2 Pruebas de software

Las pruebas de software son el proceso en el cual se evalúa y verifica un producto o aplicación software para saber si éste hace lo que se supone que debe hacer [3]. Entre los beneficios principales de las pruebas se encuentran la prevención de errores, la reducción de costes de desarrollo y la mejora del rendimiento.

Existen muchos tipos diferentes de pruebas de software, los siguientes son relevantes para aplicaciones web ya que cada uno es capaz de determinar ciertos requisitos ideales en páginas web:

- **Pruebas de aceptación:** verifican si el sistema funciona según lo previsto.
- **Pruebas de integración:** se asegura que los componentes del software operen juntos correctamente.
- **Pruebas de rendimiento:** comprueba el rendimiento del software con distintas cargas de trabajo.
- **Pruebas de regresión:** verifica si las nuevas funcionalidades rompen o degradan las funcionalidades antiguas.
- **Pruebas de estrés:** comprueba la cantidad de tensión que puede comprobar el sistema antes de fallar.
- **Pruebas de humo o *Smoke test:*** buscan realizar una serie de validaciones que aseguren las funcionalidades más críticas.
- **Pruebas exploratorias:** descubrir escenarios y situaciones difíciles de predecir que pueden conducir a errores.

3. Estudio estratégico

En la actualidad existen una gran cantidad de herramientas que nos permiten automatizar nuestras pruebas. Para poder entender qué herramienta beneficia más al momento de automatizar las pruebas de una página web, se debe definir un conjunto de características deseables o requeridas, no solo en cuanto a la herramienta en sí misma, sino también en cuanto a la persona que desarrolle las pruebas.

Algunas características deseables que debe poseer la herramienta para automatizar pruebas de una página web serían las siguientes:

- **Ejecución en múltiples navegadores:** se puede acceder a una página web desde distintos navegadores por lo tanto se requiere poder ejecutar las pruebas en los navegadores más utilizados en el mercado.

- **Pruebas consistentes:** las pruebas no deben fallar a menos que el sistema tenga un error por lo que debe ser capaz de ejecutar las pruebas la cantidad de veces que se requiera sin que el resultado varíe de una ejecución a otra.

- **Curva de aprendizaje plana:** el proceso de aprendizaje del uso de la herramienta debe ser fácil y eficiente, para evitar que se invierta una cantidad de tiempo mayor al beneficio que pueda generar.

- **Facilidad de integración:** el proceso de instalar la herramienta e integrarlo en el proyecto debe ser sencillo ya que si requiere un trabajo extra para poder incorporar lo puede evitar que se realice.

3.1 Comparativa entre herramientas

En esta sección definiremos las dos herramientas más utilizadas actualmente para la automatización de pruebas de páginas web (Selenium [4] y Cypress [5]). En esta comparativa se tomarán cuenta características descritas anteriormente, además de los beneficios que posee cada una de las herramientas.

3.1.1 Selenium

Es un conjunto de herramientas que permiten la automatización de pruebas mediante el control de los navegadores. Se trata de posiblemente la más utilizada para la automatización de pruebas. Además, es utilizada por otras herramientas de más alto nivel como base. Está compuesto por distintas herramientas como, por ejemplo:

- **IDE (Integrated Development Environment):** esta herramienta se utiliza para el desarrollo de casos de prueba. Registra las acciones de los usuarios en el navegador utilizando los comandos de Selenium.

- **Grid:** permite ejecutar las pruebas en distintas máquinas con distintas plataformas.

- **WebDriver:** es una API de automatización del navegador proporcionadas por los proveedores de los navegadores para controlar el navegador.

Selenium tiene una serie de ventajas sobre otras herramientas como, por ejemplo:

- **Amplia gama de lenguajes de programación:** Selenium soporta distintos lenguajes de programación como Java, Ruby, Python, C#, entre otros.

- **Integrado con distintas herramientas de desarrollo:** se puede integrar fácilmente con las siguientes plataformas: Jenkins, Maven, DevOps, entre otras.

- **Gran comunidad:** se estima que unas 56.000 empresas utilizan Selenium.

- **Amplia gama de complementos:** se puede ampliar más allá de su funcionalidad estándar.

Pero también cuenta con las siguientes desventajas:

- **Curva de aprendizaje empinada:** una página puede tener llamadas asíncronas lo que puede ocasionar que si tarda mucho en cargar la prueba falle, por lo que se deben tener amplios conocimientos de programación para poder manejar los problemas de sincronía.

- **Sin capacidad de informes:** no cuenta con un sistema de generación de informe por lo tanto se deben utilizar herramientas de terceros.

- **Configuración del entorno:** esta tarea puede tomar un tiempo considerable, ya que para cada navegador se tiene un WebDriver distinto y en caso de querer ejecutarla en múltiples navegadores requiere mayor configuración.

- **Robustez de las pruebas:** al desarrollar tus pruebas puedes ejecutarlas una vez y creer que funcionan, pero debido a problemas de sincronía pueden fallar la próxima vez.

3.1.2 Cypress

Es un framework de pruebas que nos permite simular la interacción con el navegador. Esta herramienta incluye librerías y pruebas E2E automatizadas que a diferencia de otras herramientas como Appium[2], Protractor[3] no utiliza Selenium como base.

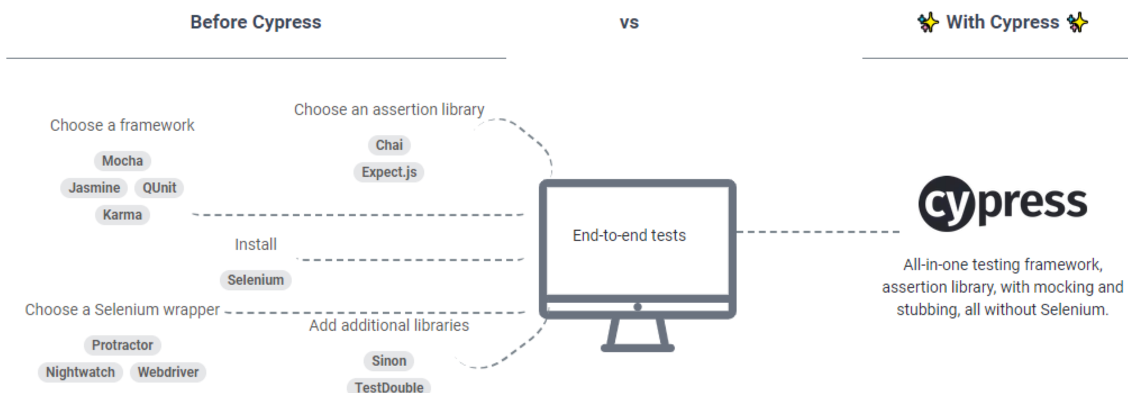


Figura 2: Resumen de todo lo que incluye Cypress. Obtenida en [6].

Como se observa en la Figura 2, cuando no existía Cypress era necesario utilizar una amplia gama de herramientas, mientras que Cypress ya incluye distintos frameworks, librerías de aserciones entre otras que facilitan el trabajo, ya que no requiere la instalación de distintas herramientas para poder utilizar sus funciones.

Cypress se ejecuta dentro del navegador controlándolo desde un proceso *backend* en node.js [6]. Además, al ejecutarse dentro del navegador permite controlar todo el tráfico de red entrante y saliente de la aplicación. Otra ventaja de ejecutarse dentro del navegador es que es mucho más rápido.

A continuación, describiremos las ventajas que tiene Cypress sobre otras herramientas:

- **Viaje en el tiempo:** toma instantáneas mientras se ejecutan las pruebas por lo tanto se podrá acceder los distintos estados por los que pasó la prueba en caso de que se desee revisar un paso anterior.

- **Capacidad de depuración:** muestra específicamente en qué momento falló la prueba y en qué comando, lo que facilita el proceso de depuración.

- **Control de tráfico de red:** permite simular el tráfico de red deseado en nuestra prueba.

[2] Appium – Página oficial: https://appium.io/
[3] Protractor – Página oficial: https://www.protractortest.org/#/

- **Resultados consistentes:** no varían los resultados entre una ejecución y otra, ya que automáticamente espera a los elementos, lo que evita que las pruebas sean inestables.

- **Capacidad de recargar en tiempo real:** se recarga automáticamente una vez realizado un cambio en las pruebas.

- **Curva de aprendizaje plana:** definir las pruebas es muy intuitivo, ya presenta una arquitectura muy simple y además existe mucha documentación sobre la herramienta.

- **Configuración del entorno:** una vez instalado ya puedes empezar a desarrollar tus pruebas, no es necesario hacer más configuraciones.

También tiene sus desventajas:

- **Solo utiliza JavaScript y TypeScript:** por lo tanto, se deberá tener conocimiento en JS para poder desarrollar pruebas complejas.

- **Ciertas funcionalidades son de pago:** desarrollar las pruebas es gratis, pero Cypress tiene un *dashboard* que nos muestra distintas métricas que es de pago.

- **Soporta solo algunos navegadores:** no soporta todos los navegadores del mercado, como Internet Explorer y Safari.

- **Múltiples pestañas:** solo permite interactuar con una pestaña del navegador.

3.1.3 Conclusiones de la comparación

Después de definir y recopilar información sobre Selenium y Cypress resumiremos las características de ambos:

Características	**Selenium**	**Cypress**
Ejecución en múltiples navegadores.	Sí: Chrome, Edge, Firefox, Internet Explorer, Opera y Safari	Sí: Chrome, Electron, Edge, Firefox y Brave
Resultados consistentes.	No	Sí
Curva de aprendizaje plana.	No	Sí
Facilidad de integración.	No	Sí
Múltiples lenguajes de programación	Sí	No, solo soporta JavaScript y TypeScript
Velocidad de ejecución	Retrasos de red	No tiene retrasos ya que se ejecuta dentro del navegador
Espera de elementos	Se debe programar manualmente	Espera automáticamente a que exista el elemento
Gratis	Sí	No, tiene funcionalidades pagas
Control de tráfico de red	No	Sí, ya que se ejecuta dentro del navegador
Múltiples pestañas	Sí	No

Tabla 1: Resumen comparativa de herramientas

Como podemos observar en la tabla, Cypress se adapta un poco más a las características deseadas, además de poseer ciertas funcionalidades que nos ayudarían bastante a la hora de desarrollar y ejecutar las pruebas.

Junto con lo mencionado anteriormente, otro factor importante a la hora de decantarse por una herramienta es que Cypress se está convirtiendo en una de las herramientas más populares para diseñar pruebas automatizadas. Una métrica que nos puede servir de referencia para determinar la popularidad de este framework es la cantidad de *GitHub Stars* que tiene un proyecto en su repositorio Git. Esto le permite a

un usuario marcar con una estrella el repositorio que le guste. En la siguiente gráfica se puede apreciar la cantidad de *GitHub Stars* a lo largo del tiempo.

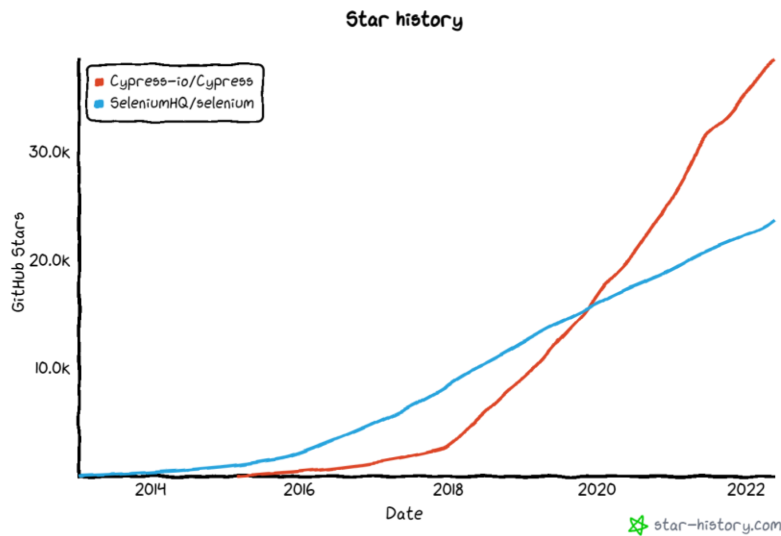


Figura 3 Relación entre cantidad de Github Stars en el tiempo. Obtenida en [7]

Otra métrica que podemos utilizar es la cantidad de descargas que tiene Cypress contra la cantidad que tiene Selenium, para esto podemos acceder a la página de NPM[4] y buscar cada uno de los paquetes para obtener esta información. Como se puede observar en la Figura 4, Cypress tiene casi el doble de descargas semanales.

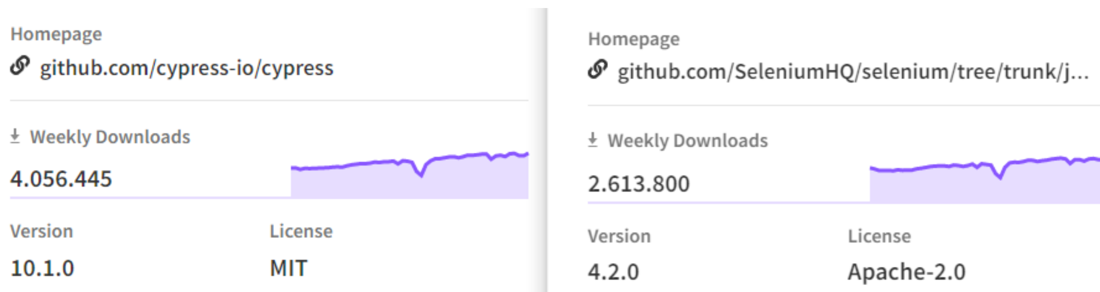


Figura 4: Cantidad de descargas semanales de Cypress y Selenium webdriver. Obtenida el día 15/06/2022

[4] NPM – Página oficial: https://www.npmjs.com/

4. Diseño de la solución

En este capítulo describiremos la metodología propuesta, las tecnologías utilizadas además de otras herramientas que pueden ser útiles en el contexto del desarrollo de las pruebas en un entorno ágil.

4.1 Metodología

Tras analizar los distintos tipos de pruebas de software que existen, la propuesta metodológica que veremos a continuación está basada en el desarrollo de dos tipos de pruebas principalmente; pruebas de humo o *smoke test* y pruebas de regresión.

La razón por la cual se opta por estos tipos de prueba es por la necesidad de cubrir cualquier tipo de cambio realizado en nuestro producto, es decir, en caso de realizar un simple cambio no es necesario ejecutar toda la batería de pruebas sino comprobar que las funcionalidades críticas siguen funcionando.

En el caso que se incorpore una nueva funcionalidad tras realizar las pruebas manuales y la implementación de las pruebas automatizadas, utilizaremos las pruebas de humo para comprobar que no se introducen errores. Por otro lado, si se introducen cambios que pueden afectar a distintas funcionalidades sí es necesario ejecutar una batería de pruebas más exhaustivas.

Se descartan algunos tipos de prueba por distintas razones: en el caso de las pruebas de estrés y las pruebas de rendimiento no aseguran que las funcionalidades críticas estén funcionando, sino que el sistema soporta una carga determinada y que cumple con tiempos determinados respectivamente, por otro lado, las pruebas exploratorias no siguen un guion en particular por lo que no se pueden automatizar.

Una vez entendido por qué elegimos este tipo de pruebas, pasamos a ver un poco más en profundidad los tipos de pruebas elegidas, para posteriormente definir cuándo se ejecuta cada tipo de prueba basándonos en el escenario que nos encontremos.

4.1.1 Pruebas de humo

Las pruebas de humo o también conocidas como pruebas de confianza son un conjunto de pruebas que se ejecutan para comprobar que las funcionalidades claves siguen funcionando [8]. Son unas pruebas simples que determinan si el producto es estable.

Este tipo de prueba tiene ventajas importantes con respecto a otros tipos:

- Proceso rápido y simple, ya que es una menor cantidad de pruebas.

- Posibilidad de detectar errores en una etapa temprana, ya que se pueden realizar continuamente.
- Facilita el trabajo del equipo de QA, solo se revisan las funciones críticas.

Este tipo de pruebas deben ejecutarse cuando se desarrollan e integran nuevas funcionalidades en el software, para asegurar que éstas no introducen un error en nuestras funcionalidades críticas.

Las pruebas de humo no sustituyen unas pruebas de regresión completas, incluso después de pasar estas pruebas podemos encontrar errores en nuestro software, pero nos da una noción general de que nuestro sistema cumple con los requerimientos mínimos para poder completarse la entrega de las nuevas funcionalidades.

4.1.2 Pruebas de regresión

Las pruebas de regresión son un conjunto de prueba que se ejecutan para comprobar que todas las funcionalidades de nuestra aplicación siguen funcionando conforme están definidas [9]. Son unas pruebas más complejas/exhaustivas.

Estas pruebas deben ejecutarse cuando se introducen cambios en funcionalidades antiguas o cambios que puedan afectar a distintas funcionalidades del software. Estos cambios pueden ser:

- **Cambio de entorno del producto:** cuando se desea entregar una versión del producto se debe comprobar que no se pierde ninguna funcionalidad en el proceso.
- **Subida de versión del *framework*:** cuando se desea realizar una subida a versión de algunos de los componentes del software bien sea la parte *frontend, api o batch.*

4.1.3 Desarrollo de las pruebas

Para el desarrollo de las pruebas se debe tener en cuenta la metodología seguida en el proyecto. La metodología ágil consiste en el desarrollo de pequeñas piezas de software en periodos cortos de tiempo, por esta razón, cuando se pase de la fase de desarrollo a la fase de pruebas, como se comentó en el capítulo 2. Debemos realizar el diseño de las pruebas y posteriormente su desarrollo, de esta manera podemos acoplar la implementación de estas pruebas a la metodología del proyecto.

Debemos empezar por el desarrollo de las pruebas de humo, ya que con éstas podemos comprobar que nuestras funcionalidades más críticas están funcionando. Posteriormente debemos desarrollar las pruebas de regresión. Este último tipo de pruebas, deben ir aumentado con el paso del tiempo, es decir, en un principio se diseñaron e implementaron una serie de pruebas, pero con el paso del tiempo nos

damos cuenta de que se pueden completar con otras pruebas que no se encontraban en el enfoque inicial.

4.1.4 Escenarios de ejecución

Tras analizar los dos tipos de pruebas y el proceso de desarrollo en la cual se basa esta metodología. Debemos comprender qué tipo de pruebas debemos ejecutar, según el desarrollo realizado en esta iteración del proyecto. Para esto definimos un conjunto de escenarios en los que se puede encontrar el proyecto.

En caso de que en esta iteración se incorporen solo nuevas funcionalidades a la página web, tendremos que ejecutar solamente nuestras pruebas de humo, ya que estas funcionalidades no modifican el código de nuestras funcionalidades anteriores.

Por otro lado, cuando se realizan cambios importantes en nuestras funcionalidades anteriores es necesario ejecutar un conjunto de pruebas que sea capaz de abarcar todas las funcionalidades, sean críticas o secundarias.

Para esto definimos un conjunto de escenarios específicos junto con el tipo de pruebas que se deben ejecutar:

Escenarios	**Pruebas de humo**	**Pruebas de regresión**
Incorporación de nuevas funcionalidades	X	
Cambios o mejoras en funcionalidades antiguas		X
Subida de versión del *framework*		X
Cambios de entorno del software		X

Tabla 2: Tipo de prueba según el escenario

4.2 Tecnologías utilizadas

Como se menciona en el capítulo 3 utilizaremos Cypress como nuestra herramienta para desarrollar los casos de prueba, pero además de ésta utilizamos distintas herramientas como GitHub[5] y Jira[6], para gestionar las versiones del proyecto y para organizar las tareas a realizar respectivamente.

[5] GitHub – Página oficial: https://github.com/
[6] Jira – Página oficial: https://www.atlassian.com/es/software/jira

4.2.1 GitHub

Es un repositorio online gratuito que permite gestionar proyectos y controlar versiones que permite a los desarrolladores administrar cambios a la vez que evoluciona el proyecto. Es uno de los repositorios más usados a nivel mundial. Además, esta herramienta permite generar flujos de trabajo, los cuales facilitan el desarrollo y la mantenibilidad.

4.2.2 Jira

En este caso se decide optar por Jira ya que proporciona herramientas para la planificación de equipos. De esta herramienta utilizaremos tableros de scrum (Figura 5) y Kanban para la gestión del desarrollo de las pruebas en un equipo de QA con múltiples personas.

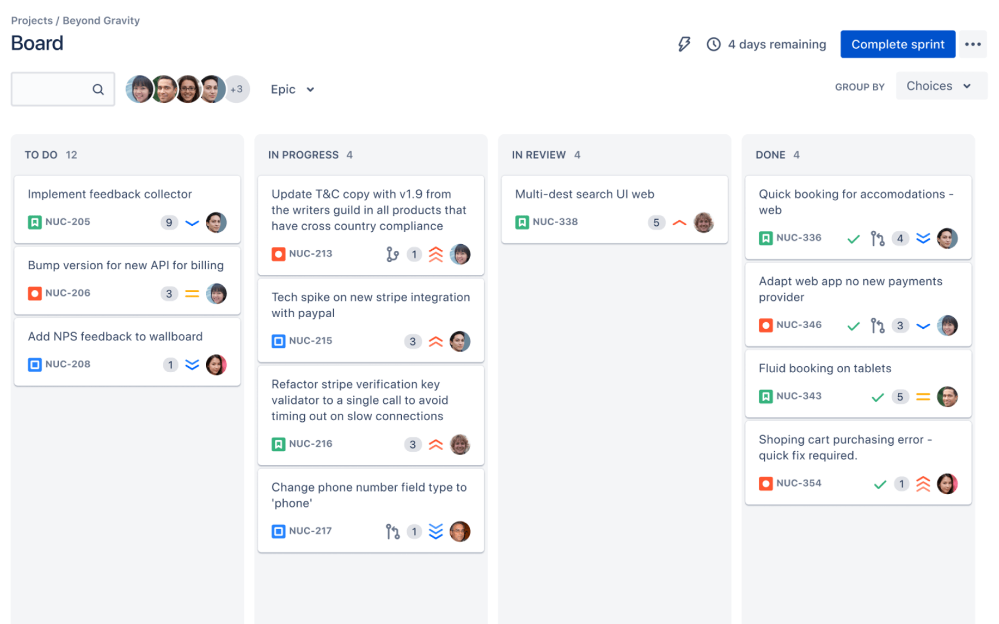


Figura 5: Ejemplo de tablero scrum. Obtenida en [10]

4.3 Estructura de Cypress

El siguiente paso sería entender la estructura interna del framework. Por esta razón en este apartado se desglosa la estructura de carpetas (Figura 6) que tiene el framework tras de su instalación.

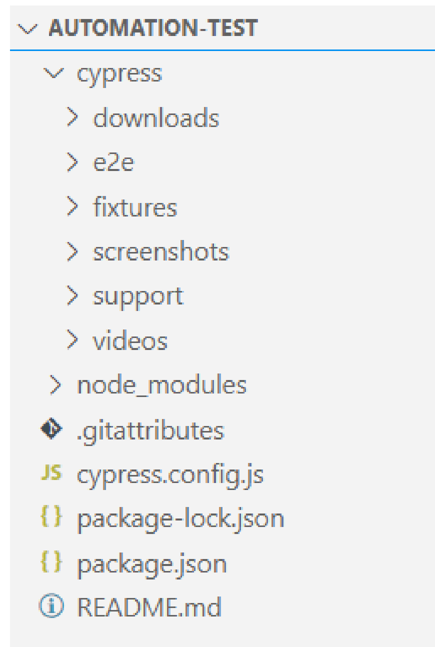


Figura 6: Estructura de Cypress

- **Carpeta e2e:** aquí se encuentran los casos de prueba desarrollados. Lo casos de prueba pueden estar desarrollados en los siguientes formatos:
    - .js
    - .jsx
    - .ts
    - .tsx
    - .coffee
    - .cjsx

- **Fixtures:** en esta carpeta se encuentran los archivos accesorios, es decir, archivos que nos proporcionan datos para utilizar en nuestros casos de prueba.

- **Downloads:** en esta encontraremos todos los archivos descargados durante la ejecución de los casos de prueba.

- **Screenshots:** se almacenarán las capturas de pantallas realizadas durante el desarrollo de las pruebas.

- **Videos:** en esta carpeta se almacenan los videos de la ejecución de los casos de prueba.

- **Support:** en esta carpeta encontramos dos archivos:

    - **Commands.js:** en el cual podemos crear nuestros propios comandos para poder reutilizar en múltiples pruebas y ahorrar tiempo.

- **E2E.js:** en este archivo (Figura 7) podemos agregar configuraciones que modificarán cómo se ejecuta Cypress posteriormente. Como se observa Figura 7 se tiene una cláusula que evalúa el nombre de la prueba para determinar si se debe ejecutar.

```
// Import commands.js using ES2015 syntax:
import './commands'

// Alternatively you can use CommonJS syntax:
// require('./commands')

beforeEach(function() {
    let testSuite = Cypress.env('SUITE');
    if (!testSuite) {
      return;
    }

    const testName = Cypress.mocha.getRunner().test.fullTitle();
    testSuite = "<"+testSuite+">"
    if (!testName.includes(testSuite)) {
      this.skip();
    }
  })
```

Figura 7: Archivo e2e.js

Luego tenemos el archivo **cypress.config.js** en el cual podemos almacenar cualquier información específica que deseemos para Cypress. En el ejemplo mostrado en la Figura 8 solo definiremos en las líneas 9 y 10 el tamaño de pantalla en la cual queremos ejecutar las pruebas y en la línea 12 la cantidad de reintentos que deseamos para las pruebas en caso de que fallen.

```
JS cypress.config.js > ...
1   const { defineConfig } = require("cypress");
2
3   module.exports = defineConfig({
4     projectId: 'h5d41y',
5     e2e: {
6       setupNodeEvents(on, config) {
7         // implement node event listeners here
8       },
9       viewportHeight:768,
10      viewportWidth:1360,
11      retries:{
12        runMode:1
13      }
14    },
15  });
```

Figura 8: Archivo de configuración cypress.config.js

4.4 Estructura de una prueba

Las pruebas en Cypress tienen una estructura (Figura 9) la cual le permite definir un conjunto de casos de prueba relacionados dentro de un mismo banco de pruebas. La estructura es la siguiente:

```
cypress > e2e > SmokeTest > JS EstructuraPrueba.cy.js > ...
1  describe('Nombre del banco de pruebas', function () {
2
3      it('Descripción de la prueba 1', function () {
4          //comandos a ejecutar
5      })
6
7      it('Descripción de la prueba 2', function () {
8          //comandos a ejecutar
9      })
10 })
```

Figura 9: Estructura de una prueba en Cypress

La primera parte de la prueba "*describe*" define el banco de pruebas, puede ser el nombre de la pantalla o el caso de uso que engloba las pruebas. Luego tenemos los "*it*" que serían cada una de las pruebas que se quieren desarrollar.

Cypress tiene otra nomenclatura que se puede usar, "*context*" como equivalente a "*describe*" y "*specify*" corresponde con "*it*".

Además, proporciona ciertas cláusulas (Figura 10) que son útiles a la hora de implementar una prueba. Éstas nos permiten reutilizar código antes o después de cada prueba sin necesidad de escribirlo dentro de cada prueba.

```
cypress > e2e > SmokeTest > JS EstructuraPrueba.cy.js > ...
1   describe('Funciones utiles', function () {
2
3       before(() => {
4           // se ejecuta antes de todos los test una sola vez
5         })
6
7         beforeEach(() => {
8           // se ejecuta antes de cada test
9         })
10
11        afterEach(() => {
12          // se ejecuta despues de cada test
13        })
14
15        after(() => {
16          // Se ejecuta despues de todos los test
17        })
18
19  })
```

Figura 10: Cláusulas útiles para desarrollar pruebas

Las cláusulas deben ir dentro del mismo banco de pruebas, es decir, dentro del *"describe"* junto con los *"it"*. La cláusula que se encuentra en la línea 3 se ejecutará antes de la primera prueba solamente, mientras que la cláusula que se encuentra en la línea 7 se ejecutará antes de todas las pruebas. Por otro lado, la cláusula que se encuentra en la línea 11 se ejecutará después de cada prueba, mientras que la cláusula que está en la línea 15 se ejecutará luego de la última prueba.

5. Automatización de pruebas con Cypress

En este capítulo se describirá el proceso de automatización del tipo de pruebas indicado en los capítulos anteriores. Para ello se debe decidir la forma en la que dividiremos nuestras pruebas, bien sea por pantallas, por casos de uso o de cualquier manera que la empresa divida las funcionalidades o el tester considere conveniente. Es recomendable utilizar los casos de uso, a menos que sean muy similares entre sí, es decir, que un caso de uso se realice en la misma pantalla que otro y que la diferencia sea un botón. En este último caso es recomendable organizarlo por pantallas.

En la memoria vamos a mostrar el proceso para el caso de organizar las pruebas por funcionalidades. Se ha elegido como caso el diseño e implementación de las pruebas de la tienda online de Consum[7]. Al no poseer los casos de uso originales, se diseñarán en el siguiente apartado para posteriormente implementarlos.

A continuación desglosaremos los pasos que se deben seguir para cubrir todo el ciclo de las pruebas en la metodología:

1) Diseño de las pruebas: en este primer paso debemos definir las pruebas que se implementarán y se ejecutarán posteriormente. Este paso tiene dos aspectos a considerar; el primero es definir el caso de prueba crítico, el cual será el primero que debemos implementar en nuestra prueba de humo y el segundo aspecto incluye definir los flujos alternativos o ciertas funcionalidades que no son críticas. Se puede observar en el apartado 5.1 de este capítulo.

2) Implementación de las pruebas: este proceso estará en constante evolución ya que debemos implementar las pruebas que se definieron en el paso anterior, pero también se deberán implementar pruebas que surjan a lo largo del desarrollo del producto. Lo descubriremos en el apartado 5.2 de este capítulo.

3) Ejecución de las pruebas: luego se encuentra la fase en la cual debemos ejecutar nuestras pruebas para que cumplan el propósito por el cual fueron diseñadas, detectar errores que se introduzcan a nuestro proyecto. Lo encontramos en el capítulo 6.

4) Análisis de los resultados: por último, debemos analizar los resultados obtenidos en las pruebas y en caso de detectar errores reportarlos para que sean solucionados por el equipo de desarrollo.

[7] Tienda online CONSUM – Página oficial: https://tienda.consum.es/

5.1 Diseño de las pruebas

El proceso empieza con la definición de los casos de prueba, en este caso se eligen ciertas funcionalidades para realizar los dos tipos de prueba, de esta manera se pueden observar las diferencias entre ellas. Pasamos a definir los casos de uso:

CU-01 Agregar productos al carrito de compras

Caso de Uso	CU-01 Agregar productos al carrito
Actor	Cliente
Descripción	El cliente busca un producto, tras encontrar el producto agrega la cantidad que desea incluir en el carrito.
Flujo básico	**1. Introducir el nombre del producto.** El cliente escribe en el buscador el producto deseado. **2. Selección del producto desde el menú flotante.** El cliente selecciona el producto y la cantidad que desea.
Flujos Alternos	**2da opción de búsqueda:** **1. Introducir el nombre del producto y presionar buscar.** El cliente escribe en el buscador el producto deseado y presiona buscar. **2. Selección del producto desde el menú fijo.** El cliente selecciona el producto y la cantidad que desea. **Agregar/Eliminar cantidades del producto desde el carrito** **1.Entrar al carrito** El cliente selecciona el botón del carrito **2.Agregar/Eliminar producto** El cliente selecciona en + o – según sea el caso
Pre-condiciones	**1. Acceder a la tienda online.** El cliente debe iniciar sesión. **2. Cliente registrado.** El cliente debe iniciar sesión.
Post-condiciones	**1. Productos deben estar en el carrito.** Se comprueba que los productos se encuentran en el carrito.

Tabla 3 CU-01 Agregar productos al carrito

La prueba de humo de este caso de uso vendría dada por su funcionalidad principal que es agregar productos al carrito y comprobar que éstos se agregan posteriormente en el mismo. Mientras que las pruebas de regresión pueden implementar los flujos alternativos.

CU-02 Añadir/Borrar dirección de entrega

Caso de Uso	CU-02 Añadir/Borrar dirección de entrega
Actor	Cliente
Descripción	El cliente añade/elimina dirección de entrega.
Flujo básico	**1. Acceder a “Mi cuenta”.** El cliente selecciona en el menú “Mi cuenta”. **2. Selecciona “Mis Direcciones”.**

	El cliente selecciona en el submenú “Mis Direcciones”. En el caso de añadir una dirección: 3**. Selecciona “Añadir dirección”.** Seleccionar el botón “Añadir dirección”. 4**. Completar formulario.** Se completan los campos obligatorios del formulario. En el caso de eliminar una dirección: 3**. Selecciona la dirección.** Seleccionar el recuadro con la dirección que deseamos eliminar. 4**.Seleccionar el icono de basura.** Seleccionar el botón con icono de basura y confirmar la eliminación.
Flujos alternos	**Intentar dar de alta una dirección con campos obligatorios faltantes.** **Realizar los pasos 1 y 2 del flujo básico.** **3. Selecciona “Añadir dirección”** Seleccionar el botón “Añadir dirección”. **4. Completar el formulario de forma parcial** Se completan parte de los campos obligatorios del formulario. **5. Intentar guardar** Se intenta guardar seleccionando el botón, pero se muestran mensajes de error en todos los campos obligatorios y al final del formulario, **En caso de que esté fallando la API** **Tras completar correctamente el formulario y seleccionar Guardar si el proceso falla en algún momento no se vuelve a la página anterior si no que se mantiene con los datos del formulario rellenos**
Pre-condiciones	**1. Acceder a la tienda online.** El cliente debe iniciar sesión. **2. Cliente registrado.** El cliente debe iniciar sesión.
Post-condiciones	**1.Comprobar los datos añadidos o eliminados.** Se comprueba que los datos que aparecen en “Mis Direcciones” corresponden con los datos introducidos.

Tabla 4: CU-02 Añadir/Borrar dirección de entrega

La prueba de humo para este caso de uso sería añadir una dirección y eliminarla, mientras que las pruebas de aceptación pueden ir desde comprobar el formato de los campos requeridos hasta lo que pasaría si falla el proceso de añadir o eliminar.

5.2 Implementación de las pruebas

En esta sección desarrollaremos ambos tipos de pruebas para cada uno de los casos de uso descritos en el apartado anterior. Para esto seguiremos la estructura comentada en el capítulo 4.

Ambos tipos se encontrarán dentro del mismo banco de pruebas, para diferenciarlos añadiremos en el nombre de cada prueba “<smokeTest>” o “<regresion>” según sea el caso. Se pudiera realizar en dos proyectos separados, pero esto no es recomendable

debido a que requiere el doble de mantenimiento y mucho código repetido ya que un tipo de pruebas esta englobado por el otro.

Como se observa en la Figura 11 así quedaría la cabecera de ambos tipos de prueba. Dentro de éstos se deberá desarrollar la prueba a implementar.

```
it('Caso de prueba de ambos tipos <smokeTest> <regresion>',function(){
    // código de la prueba
    // .....
    // .....
})

it('Caso de prueba solo de regresion <regresion>',function(){
    // código de la prueba
    // .....
    // .....
})
```

Figura 11: Identificadores de tipo de prueba

En la Figura 12 se puede observar el código implementado en el archivo **e2e.js** mencionado en el capítulo 4, esta función la utilizamos para decidir si la prueba se ejecuta o no según su tipo. Para esto obtenemos el nombre de cada prueba y si incluye el tipo se ejecuta si no se salta.

```
beforeEach(function () {
  let testSuite = Cypress.env('SUITE');
  if (!testSuite) {
    return;
  }

  const testName = Cypress.mocha.getRunner().test.fullTitle();
  testSuite = "<" + testSuite + ">"
  if (!testName.includes(testSuite)) {
    this.skip();
  }
})
```

Figura 12: Archivo e2e.js

En el siguiente capítulo veremos cómo ejecutar ambos tipos de prueba, además de las diferentes formas de ejecutarlas.

CU-01 Agregar productos al carrito de compras

Para el banco de pruebas es necesario implementar las precondiciones indicadas en el caso de uso, además de leer los datos necesarios para las pruebas. Para realizarlo lo dividiremos en dos cláusulas; la primera será un "*before()*" en la cual accederemos a la página web y posteriormente iniciaremos sesión.

```
before('Acceder a la pagina e iniciar sesion', function () {
    // Visitar pagina web
    cy.visit('https://tienda.consum.es/')
    // Aceptar todas las cookies
    cy.get('#onetrust-accept-btn-handler', { timeout: 100000 }).click()
    // Iniciar sesion
    cy.fixture('cuenta/datos-acceso').then(function (datosAcceso) {
        cy.iniciarSesion(datosAcceso.user, datosAcceso.password);
    })
})
```

Figura 13: Cláusula "before()" del CU-01

En la Figura 13 podemos observar que primero visitamos la web, posteriormente aceptaremos las cookies y para finalizar iniciaremos sesión mediante un comando de Cypress personalizado, se opta por realizar un comando debido a que se deberá iniciar sesión para cada uno de los bancos de prueba y no sería correcto copiar y pegar código. En caso de querer ver su implementación se encuentra en el Anexo I -3.

La segunda cláusula (Figura 14) "*beforeEach()*" la utilizaremos para leer los datos, que en este caso de uso serán un conjunto de productos (Anexo I - 1), esto se debe a que después de cada caso de prueba se vacían las variables utilizadas por lo tanto deberemos leer los datos para cada uno de los casos de prueba.

```
beforeEach('Cargar datos para cada prueba', function () {
    // Obtenemos los datos a utilizar durante las pruebas
    cy.fixture('datos/cu-01.json').then(function (datos) {
        this.datos = datos;
    })
})
```

Figura 14: Cláusula beforeEach() del CU-01

Pruebas de humo

Para este tipo de prueba y caso de uso se desarrollará un caso de prueba para que se agreguen múltiples productos al carrito y posteriormente se comprobará que éstos se encuentran dentro del carrito.

Para facilitar la comprensión dividiremos la explicación en dos partes, en la Figura 15 se encuentra la primera parte y se puede observar que dado una lista de productos (*this.datos.productos*) se procede a buscarlos a través del menú flotante de la página web, para esto debemos introducir el nombre del producto, como se puede observar en la línea 32, luego esperar a que carguen los productos encontrados con el nombre introducido, para posteriormente agregarlo al carrito seleccionando sobre él, como se indica en la línea 38.

```
23    it('Agregar productos al carrito a traves del menu flotante <smokeTest> <regresion>', function () {
24        // Hacemos click en la barra de busqueda, escribimos el producto y pulsamos enter
25        cy.get('#txtSearch').click()
26        // A partir de una lista de productos indefinida
27        this.datos.productos.forEach(producto => {
28            // Interceptamos las llamadas para obtener productos
29            cy.intercept('GET', '/api/rest/V1.0/catalog/searcher/**').as('getProduct')
30            cy.intercept('GET', 'https://api.empathybroker.com/tagging/v1/track/consum/add2cart?**')
31                .as('add2cart')
32            cy.get('.smart-searcher-header__container__input input')
33                .clear().type(producto, { force: true })
34            cy.get('#smart-searcher--container-results-title span').click()
35            // Esperamos a que cargue el producto
36            cy.wait('@getProduct', { timeout: 10000 })
37            // Lo agregamos al carrito
38            cy.wait(500).get('.smart-searcher-grid--prod > cmp-widget-product:first() .btn')
39                .should('be.visible').click()
40            cy.wait('@add2cart', { timeout: 10000 })
41            cy.get('.triple-element-block__right--remove').click()
42        });
43        cy.get('.iconAngular-cancel').click()
```

Figura 15: Prueba de humo CU-01 parte 1

En la siguiente parte (Figura 16) accederemos al carrito y comprobaremos que se agregaron los productos en el orden en el que fueron seleccionados. Para esto se invierte la lista de productos original (línea 48) y comparamos que el nombre de la lista original es el mismo que el que se encuentra en el carrito de compras, como se observa en la línea 51.

```
45        // Accedemos al carrito
46        cy.get('.header__main .header__cart .iconAngular-cart').click()
47        // Invertimos la lista de productos para coincida con la introducida
48        var productosInvert = this.datos.productos.reverse()
49        // Comparamos la lista introducida con la existente en el carrito
50        cy.get('.cart-product').each(function ($el, index) {
51            cy.wrap($el).should('contain.text', productosInvert[index])
52        })
53        cy.get('.iconAngular-cancel').click()
54    })
```

Figura 16: Prueba de humo CU-01 parte 2

Pruebas de regresión

Para las pruebas de regresión incorporaremos más pruebas, entre las cuales se encuentran los flujos alternos indicados en el caso de uso. El primero que se menciona es otra manera de agregar productos al carrito, en este caso agregaremos un producto desde el menú fijo.

```
it('Agregar producto desde el menu fijo <regresion>', function () {
    // Interceptamos las llamadas para obtener productos
    cy.intercept('GET', '/api/rest/V1.0/catalog/product?**').as('getProduct')
    cy.intercept('GET', 'https://api.empathybroker.com/tagging/v1/track/consum/add2cart?**')
        .as('add2cart')
    // Hacemos click en la barra de busqueda, escribimos el producto y pulsamos enter
    cy.get('#txtSearch').click()
    cy.get('.smart-searcher-header__container__input input').clear()
        .type(this.datos.productoExtra + '{enter}')
    // Esperamos a que cargue el producto
    cy.wait('@getProduct', { timeout: 10000 })
    // Lo agregamos al carrito
    cy.wait(500).get('cmp-widget-product :first() .btn.ng-star-inserted').wait(500).click()
    cy.wait('@add2cart', { timeout: 10000 })
})
```

Figura 17: Pruebas de regresión CU-01 parte 1

Como se observa en la Figura 17 este caso de prueba es muy similar al anterior con la diferencia de que al momento de buscar el producto presionamos en buscar en vez de seleccionar el producto directamente del menú flotante, como se observa en la línea 9.

En el siguiente caso de prueba será el otro flujo alterno del caso de uso en el cual se debe aumentar y disminuir la cantidad de un producto y comprobar que se suma y se resta correctamente.

```
it('Aumentar y disminuir cantidad de un producto desde el carrito <regresion>', function () {
    // Accedemos al carrito
    cy.get('.header__main .header__cart .iconAngular-cart').click()
    // Obtenemos el elemento
    cy.get('cmp-cart-product:has(:contains("' + this.datos.productoExtra + '"))')
        .then(function (element) {
            var cantidad = parseInt(element[0].textContent
                .substring(element[0].textContent.lastIndexOf('€') + 1))
            cy.wrap(element).find('.iconAngular-plus').click()
            cy.wrap(element).then(function (product) {
                assert.equal(parseInt(product[0].textContent
                    .substring(product[0].textContent.lastIndexOf('€') + 1)), cantidad + 1)
            })
            cy.wrap(element).find('.iconAngular-less').click()
            cy.wrap(element).then(function (product) {
                assert.equal(parseInt(product[0].textContent
                    .substring(product[0].textContent.lastIndexOf('€') + 1)), cantidad)
            })
            cy.wrap(element).find('.iconAngular-less').click()
        })
    cy.get('.iconAngular-cancel').click()
})
```

Figura 18: Pruebas de regresión CU-01 parte 2

En la Figura 18 se observa que accedemos al carrito localizamos el producto del cual deseamos aumentar/disminuir la cantidad. Posteriormente obtenemos la cantidad actual del producto y la aumentamos para luego comparar la cifra original + 1 con la cifra actual. A continuación, hacemos lo mismo, pero disminuyendo la cantidad del producto.

Para finalizar tanto las pruebas de humo como las pruebas de regresión insertamos una última cláusula "*after()*" en la cual vaciaremos el carrito para devolver al estado inicial nuestro entorno de pruebas.

```
after('Borrar productos del carrito <smokeTest> <regresion>', function () {
    // Accedemos al carrito
    cy.get('.header__main .header__cart .iconAngular-cart').click()
    // Vaciamos a lista de compras
    cy.intercept('DELETE', '/api/rest/V2.0/shopping/cart?**').as('deleteCarrito')
    cy.get('.cart-products-list-component__header .iconAngular-remove').click()
    cy.get('.btn-primary-reverse').click()
    cy.wait('@deleteCarrito', { timeout: 10000 })
})
```

Figura 19: Cláusula "after()" CU-01

En la Figura 19 observamos que se accede al carrito y luego se selecciona el botón para eliminar todos los productos del carrito.

Todas las pruebas y cláusulas se encontrarán dentro un solo archivo llamado "*cu-01-cy.js*" el cual se encuentra en la carpeta "*e2e*" mencionada en el capítulo 4.

CU-02 Añadir/Borrar dirección de entrega

Las precondiciones para este caso de uso serán muy similares al anterior que se encuentra en la Figura 13 pero se incorporará el acceso a "Mi cuenta" y posteriormente a "Mis direcciones". El archivo del que se extraerán los datos para la prueba será cu-02.json.

Pruebas de humo

Para este tipo de prueba y caso de uso tendremos dos casos de prueba: en el primero añadiremos una dirección y comprobaremos que se añade correctamente y en el segundo en el cual eliminaremos una dirección.

Para el primer caso de prueba debemos definir un conjunto de datos necesarios para introducir la dirección, como: el código postal, la provincia, la población, el tipo de vía, el nombre de la vía y el número (Anexo I - 2), en este caso se puede añadir la misma dirección múltiples veces por lo que un solo conjunto de datos nos permitirá cubrir las pruebas sin necesidad de modificarlos.

```
it('Añadir direccion <smokeTest> <regresion>', function () {
    cy.intercept('GET', '**/deliveryaddress').as('getAddress')
    cy.intercept('GET', '/api/rest/V1.0/commons/country/**').as('getRegions')
    cy.intercept('GET', '/api/rest/V1.0/commons/city/**').as('getCities')
    cy.intercept('POST', '**/deliveryaddress').as('postAddress')
    // Selecionamos Agregar direccion
    cy.get('.button-address-left').click()
    cy.wait('@getRegions', { timeout: 10000 })
    // Llamamos a nuestro comando para agregar una direccion
    cy.addDirection(this.datos)
    cy.wait('@postAddress', { timeout: 5000 }).then(function (intercept) {
        assert.equal(intercept.response.statusCode, 201)
    })
})
```

Figura 20: Prueba de humo CU-02 Añadir dirección

Como se observa en la Figura 20, primero seleccionamos el botón para agregar dirección y luego ejecutamos nuestro comando "*addDirection()*" definido en el fichero *commands* mencionado en el capítulo 4 (Anexo I -4), creado debido a que este procedimiento se realizará en otra prueba. Posteriormente nos aseguramos de que el procedimiento se realizó correctamente ya que envía como respuesta un 201.

El segundo caso de prueba elimina una dirección de las que se encuentren disponibles en nuestro perfil. Para esto busca a la primera disponible y la elimina, seleccionando el icono de la basura y confirmando el procedimiento como se observa en la Figura 21.

```
it('Eliminar direccion de entrega <smokeTest> <regresion>', function () {
    cy.intercept('DELETE', '/api/rest/V1.0/user/**').as('deleteAddress')
    // Seleccionamos la primera direccion
    cy.get('.card.address-delivery:first()').as('card')
        .find(' cmp-icon .iconAngular-up ').click()
    // Selecionamos el icono de la papelera
    cy.get('@card').find('.iconAngular-remove').click()
    // Confirmamos el borrado
    cy.get('.action-button-yes').click()
    cy.wait('@deleteAddress', { timeout: 5000 })
})
```

Figura 21: Prueba de humo CU-02 Eliminar una dirección

Pruebas de regresión

Para este tipo de prueba se desarrollarán dos casos de prueba. El primero comprueba los campos obligatorios. Para esto se opta seleccionar el campo, pero no rellenarlo, lo que debería ocasionar que salte un mensaje de error indicando que es obligatorio en cada uno de los campos que lo sean, como se observa en la Figura 22.

Añadir dirección

Código Postal*

El campo es obligatorio

Figura 22: Mensaje de error campo obligatorio

Para la segunda prueba se utilizará nuevamente el comando "*addDirection()*" pero esta vez se modificará la llamada realizada a la API para el alta de la dirección, intervendremos la llamada y devolveremos un error para comprar que el sistema reacciona como se indica en el flujo alterno indicado en el caso de uso.

```
it('Fallo al intentar añadir una direccion <regresion>', function () {
    cy.intercept('GET', '**/deliveryaddress').as('getAddress')
    cy.intercept('GET', '/api/rest/V1.0/commons/country/**').as('getRegions')
    cy.intercept('GET', '/api/rest/V1.0/commons/city/**').as('getCities')
    cy.intercept('POST', '**/deliveryaddress', (req) => {
        req.destroy()
    }).as('postAddress')
    // Selecionamos Agregar direccion
    cy.get('.button-address-left').click()
    cy.wait('@getRegions', { timeout: 10000 })
    // Llamamos a nuestro comando para agregar una direccion
    cy.addDirection(this.datos)
    cy.wait('@postAddress', { timeout: 5000 })
    cy.get('#purchase-address-adduseraddress-content-button--cancel').click()
    cy.get('#purchase-address-adduseraddress-content-button--cancel')
        .should('contain.text','No se ha podido completar la operación. Inténtelo de nuevo más tarde.')
})
```

Figura 23: Prueba de regresión CU-02 Error en la API

Para realizar esto debemos indicarle a través de "*cy.intercept()*" que cuando detecte la llamada la API la destruya con "*req.destroy()*". Esto ocasiona que nunca se realice la petición y se devuelva un error al *front*, lo que ocasiona que éste reaccione como si hubiera fallado realmente. Al seleccionar "Cancelar" nos debe mostrar un mensaje de error debido a que no se pudo completar la operación, como se observa en la Figura 23.

Tras desarrollar ambos tipos de prueba podemos observar la diferencia del enfoque utilizado para los tipos de prueba, en la que las pruebas de humo cubren el camino crítico y las pruebas de regresión cubren flujos alternativos o las funcionalidades secundarias.

5.3 Buenas prácticas

Como en cualquier producto software siempre debemos seguir buenas prácticas a la hora de implementarlo. En este caso como Cypress se desarrolla en JavaScript, es recomendable seguir las buenas prácticas del lenguaje, además de las indicadas por Cypress.

Buenas prácticas para cualquier producto software [11]:

- Utilizar nombres de variables que nos permitan entender el código.
- Extraer código a una función.
- Reutilizar código.

Algunas recomendaciones para buenas prácticas en JavaScript son las siguientes [12]:

- Debemos tener nuestro código indentado, para facilitar la lectura del mismo.
- No superar los 80 caracteres por línea.

- A pesar de no ser necesario en JavaScript siempre deberíamos declarar las variables.
- Seguir la convención de nombres camelCase. Por ejemplo, el nombre de una variable: datosDePrueba.

Algunas recomendaciones de Cypress para desarrollar nuestras pruebas [4]:

- Intentar no utilizar selectores CSS como identificadores y clases a menos que sea absolutamente necesario, debido a que los selectores CSS pueden variar lo que conllevaría mayor mantenimiento.
- Crear nuestros propios comandos para evitar copiar y pegar código innecesario.
- Evitar añadir esperas innecesarias, ya que puede ocasionar que nuestras pruebas tarden más de lo necesario.
- Durante el desarrollo, ejecutar las pruebas varias veces para asegurar que funcionan correctamente.

Para finalizar este apartado, se recomienda utilizar kebab-case para los nombres de los archivos y carpetas que se encuentren en nuestro proyecto ya que de no seguirlo se puede tener problemas con la mayoría de los proveedores de integración continua. Por ejemplo, si deseamos crear una carpeta para nuestros datos de prueba debe ser datos-de-prueba o un archivo dentro de esa carpeta datos-caso-uso.

Una situación que puede vivirse en cualquier empresa y que se vivió en la empresa en la que se realizaron las practicas es la de incorporación de nuevos miembros a un equipo, los cuales pueden carecer de los conocimientos sobre las tecnologías utilizadas por lo que deben pasar por un proceso de aprendizaje.

Como en todo proceso de aprendizaje se cometen errores, por lo que al principio es normal que las pruebas sean un poco inestables. Para evitar esto es necesario apoyarnos en la documentación que proporciona la herramienta, en la cual se tiene una guía para las buenas prácticas, para que las pruebas desarrolladas tengan la menor cantidad de errores posibles.

Además, debemos entender que debe ser considerado un proyecto software en sí mismo por lo que debemos desarrollarlo teniendo en cuenta la mantenibilidad de este. Así como también debemos entender que como todo proyecto puede sufrir cambios por lo que debemos prevenir este tipo de situaciones y utilizar los comandos de Cypress, de esta manera si hay un cambio solo lo realizamos en un solo sitio y no debemos recorrer prueba a prueba para cambiar lo necesario.

6. Ejecución de pruebas

En este capítulo veremos las distintas formas que nos brinda Cypress para ejecutar nuestras pruebas y los beneficios mencionados en el capítulo 3.1.2. También mostraremos los resultados obtenidos según la ejecución de los tipos de prueba.

6.1 Tipos de ejecución

Cypress nos brinda dos maneras de ejecutar las pruebas, una a través de su interfaz de usuario y otra mediante el terminal del navegador. Para esto definimos tres *scripts* en el archivo **package.json** (Figura 24) del proyecto.

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "cypress:open": "cypress open",
  "cypress:run": "cypress run",
  "cypress:record:run": "cypress run --record --key 7f5d4c68-f9e3-4ebd-a1b2-4fc174eb85ad"
},
```

Figura 24: Scripts para la ejecución de las pruebas

Dividiremos en dos los *scripts*. El primero es el que utilizaremos para desarrollar las pruebas debido a que tras realizar un cambio y guardarlo se ejecuta de nuevo automáticamente, lo que nos facilita el desarrollo. En cambio, el segundo y el tercero los utilizaremos cuando deseamos ejecutar nuestra batería de pruebas. Esto se debe a que con estos scripts se graban automáticamente cada una de las pruebas y se realiza una captura de pantalla justo en el momento que falla. La diferencia entre estos últimos es que en el tercero se guardarán los resultados en la nube de Cypress, los cuales podremos analizar posteriormente

El primero "*cypress:open*" abre la interfaz de usuario de Cypress, en la cual podremos seleccionar distintas configuraciones para nuestras pruebas.

El primer paso sería elegir el navegador en la cual deseamos que se ejecuten las pruebas. Para esto debemos tener instalado el navegador en cual deseemos ejecutar nuestras pruebas. Como se observa en la Figura 25, Cypress nos mostrará el conjunto de navegadores que tengamos instalados que soporte.

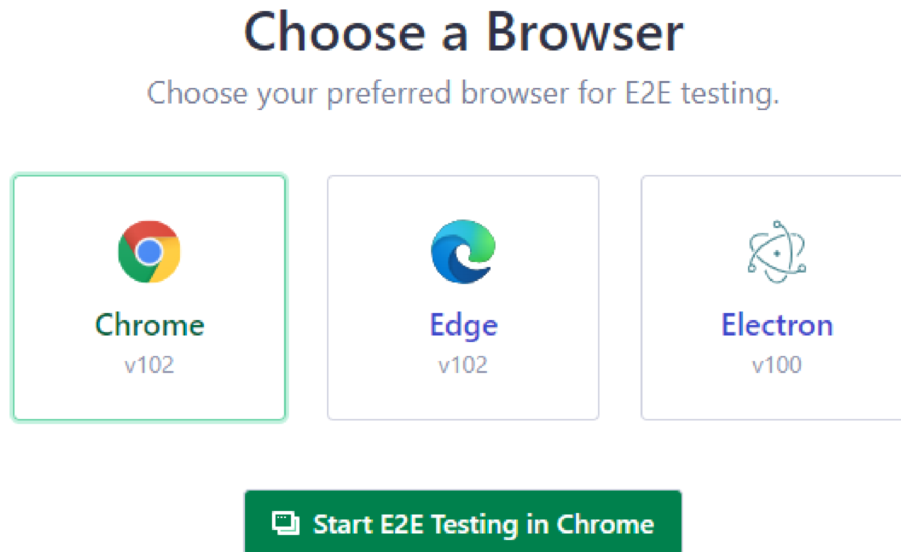


Figura 25: Elección de navegador para ejecutar las pruebas.

Una vez seleccionado el navegador, nos muestra una interfaz con distintas pestañas (Figura 26) que podemos elegir: las pruebas (*specs*), las ejecuciones anteriores o *runs* (cuando se ejecutan con el tercer *script*) y configuraciones (*settings*).

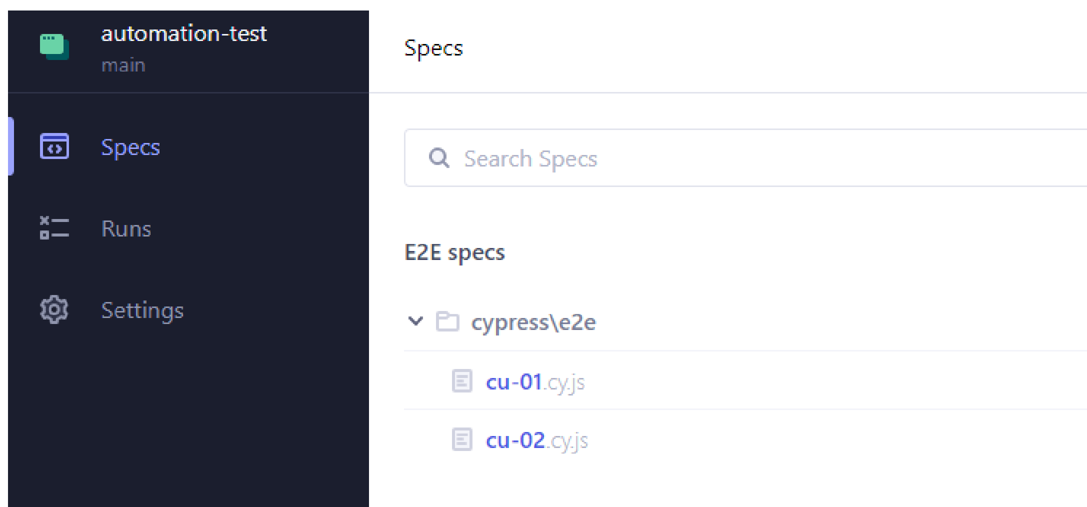


Figura 26: Menú interfaz de usuario de Cypress.

Podemos buscar una prueba con el buscador que se observa en la Figura 24 o podemos seleccionar la prueba que queremos que se ejecute. Posteriormente se empezará a ejecutar la prueba que seleccionemos. En este tipo de ejecución no se puede seleccionar más de un banco de pruebas ya que no se puede asegurar la independencia de estados entre los distintos bancos de prueba.

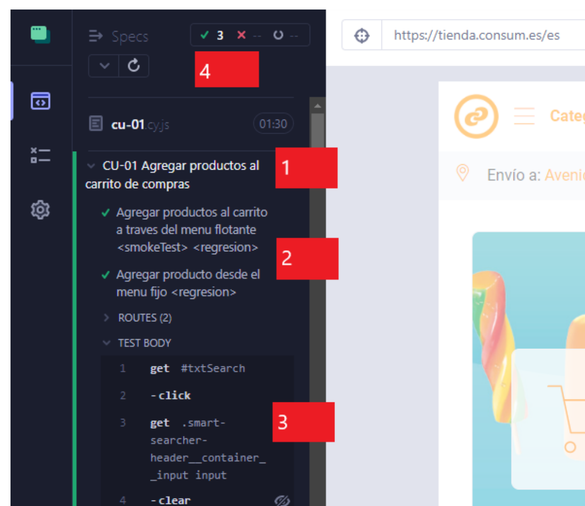


Figura 27: Interfaz de usuario durante la ejecución de una prueba

En la Figura 27 podemos observar distintos números, los cuales describiremos a continuación:

1. Nombre del banco de pruebas indicado en el “*describe()*” .
2. Nombre del caso de prueba indicado en el “*it()*”.
3. Pasos realizados durante la ejecución y el tráfico de llamadas de red.
4. Resultados de los casos de prueba.

Si seleccionamos en uno de los pasos de ejecución nos devolverá al momento donde se realiza ese paso en concreto lo que facilita la depuración de las pruebas, además de si seleccionamos una de las llamadas realizadas nos mostrará por la consola del navegador la respuesta obtenida por la misma, como se observa en la Figura 28.

```
Event:                     request
Resource type:             xhr
Method:                    GET
Url:                       https://tienda.consum.es/api/rest/V1.0/catalog/product?q=Bloc%20de%20Cuadros%20Tapa%20Dura%2080%20Hojas
Request went to origin?:   yes
Matched `cy.intercept()`:  ▶{RouteMatcher: {…}, RouteHandler Type: 'Spy', RouteHandler: undefined, Request: {…}, Response: {…}, …}
Request headers:
▶{sec-ch-ua: '" Not A;Brand";v="99", "Chromium";v="102", "Google Chrome";v="102"', X-TOL-ZONE: '163', X-TOL-CHANNEL: '1', sec-ch-ua-
Response status code:      200
Response headers:          ▶{date: 'Sun, 12 Jun 2022 18:44:21 GMT', content-encoding: 'gzip', x-content-type-options: 'nosniff', x-c
Response body:             ▼{totalCount: 1, hasMore: false, products: Array(1)}
                              hasMore: false
                             ▶products: [{…}]
                              totalCount: 1
                             ▶[[Prototype]]: Object
```

Figura 28: Llamada a la api para obtener un producto CU-01

Como se mencionó anteriormente, el segundo y el tercer *script* nos permiten ejecutar la batería de pruebas completa, se deben ejecutar por consola para esto se pueden indicar distintas opciones, pero entre las más utilizadas están:

- --browser: le indicamos el navegador con el cual queremos que se ejecuten las pruebas

- --headed: podemos indicarle que se muestre el navegador en lugar de ejecutarse solo por el terminal. Para esto se abriría y cerraría un navegador para cada prueba.
- --spec: ruta de los bancos de prueba a ejecutar
- --tag: indicar una etiqueta para la ejecución que se mostrará posteriormente en el dashboard de resultados

La diferencia principal entre el *script “cypress:run”* y *“cypress:record:run”* es la opción –record –key el cual le indica que queremos que la ejecución se guarde en la nube de Cypress para posteriormente ver sus resultados en el dashboard.

Tras indicar todo lo que deseemos para ejecutar nuestras pruebas y ejecutarlas nos mostrará un resumen de la configuración de la ejecución, como el navegador, la cantidad de bancos de prueba y el nombre de cada uno de ellos, como se observa en la Figura 29.

```
(Run Starting)

Cypress:        10.0.2
Browser:        Electron 100 (headless)
Node Version:   v16.15.1 (C:\Program Files\nodejs\node.exe)
Specs:          2 found (cu-01.cy.js, cu-02.cy.js)
Searched:       C:\Users\Mario\Desktop\automation-test\cypress\e2e\**
```

Figura 29: Resumen de ejecución mediante terminal

Posterior a esto nos irá mostrando una tabla de resultados de cada uno de los bancos de prueba ejecutados, en la cual nos indicará para cada caso de prueba si pasó o falló y el tiempo que tarda en ejecutarlo, como se observa en la Figura 30. Además, se guardará un video de la ejecución en la carpeta “video” mencionada en el capítulo 4.3.

```
Running:  cu-01.cy.js                                                   (1 of 2)

CU-01 Agregar productos al carrito de compras
  √ Agregar productos al carrito a traves del menu flotante <smokeTest> <regresion> (54130ms)
  √ Agregar producto desde el menu fijo <regresion> (9712ms)
  √ Aumentar y disminuir cantidad de un producto desde el carrito <regresion> (5006ms)

3 passing (1m)

(Results)

Tests:        3
Passing:      3
Failing:      0
Pending:      0
Skipped:      0
Screenshots:  0
Video:        true
Duration:     1 minute, 16 seconds
Spec Ran:     cu-01.cy.js
```

Figura 30: Resultado de un banco de prueba ejecutado por terminal

En caso de que una prueba falle, lo reintentará según la política de reintentos definida en nuestro archivo **cypress.config.js.** Es recomendable establecer un reintento para este tipo de ejecución, ya que, si la prueba falla por un motivo externo, como por ejemplo la conexión, se pueda reintentar la misma.

Si se realiza el reintento, previo a este se guardará una captura de pantalla del momento en que falló en la carpeta "*screenshots*", se puede observar una captura de ejemplo en el anexo I - 5. Además, se mostrará por la consola el detalle de las pruebas ejecutadas de cada banco de prueba, como se observa en la Figura 31.

```
Running:  cu-02.cy.js                                                    (2 of 2)


CU-02 Añadir/Borrar direccion de entrega
  √ Añadir direccion <smokeTest> <regresion> (51964ms)
  √ Fallo al intentar añadir una direccion <regresion> (16431ms)
  (Attempt 1 of 2) Comprobar mensajes de error <regresion>
  1) Comprobar mensajes de error <regresion>
  (Attempt 1 of 2) Eliminar direccion de entrega <smokeTest> <regresion>
  2) Eliminar direccion de entrega <smokeTest> <regresion>


2 passing (2m)
2 failing
```

Figura 31: Prueba fallida durante ejecución por terminal

En caso de que el primer intento falle, pero el segundo pase, se deberá analizar posteriormente para determinar cuál fue la razón por la que el primero falló y se realicen las modificaciones en la prueba de ser necesario.

Una vez ejecutados todos los casos nos mostrará una tabla resumen con todos los bancos de prueba ejecutados y los resultados de cada uno de ellos, como se observa en la Figura 32.

(Run Finished)

	Spec		Tests	Passing	Failing	Pending	Skipped
√	cu-01.cy.js	01:16	3	3	-	-	-
×	cu-02.cy.js	01:34	4	2	2	-	-
×	1 of 2 failed (50%)	02:50	7	5	2	-	-

Figura 32: Resultados de ejecución por terminal

Para los tres *scripts* es necesario indicar el tipo de prueba a ejecutar. Para hacer esto debemos agregar la siguiente opción al momento de ejecutar las pruebas:

- En caso de ejecutar las pruebas de regresión: --env SUITE=regresion
- En caso de ejecutar las pruebas de humo: --env SUITE=smokeTest

6.2 Resultados obtenidos

Tras la ejecución de las pruebas de los CU definidos en el capítulo anterior podemos observar que el tiempo total de ejecución de las pruebas estaba alrededor de los 2 minutos, mientras que si decidimos realizar este procedimiento manualmente nos llevaría aproximadamente 15 minutos.

Además, pudimos encontrar dos tipos de error principalmente:

- Problemas de rendimiento: tras múltiples ejecuciones se puede observar que la misma prueba podía fallar o pasar esto dependía del tiempo de carga de ciertas pantallas. Para los casos de prueba se definieron unos tiempos aceptables y muchas veces la página no lo cumplía, esto generaba un *timeout* o un fallo por acabarse el tiempo.

- Error en la automatización: esto puede deberse a dos factores, el primero que desde un principio esté mal y el otro es el cambio en el funcionamiento de un componente por lo tanto se deben adaptar las pruebas al nuevo funcionamiento.

El primer tipo de error descrito anteriormente supone un gran problema de calidad en el producto, ya que puede generar que un cliente que considera que la página web es muy inestable decida no utilizarla y opte por otra opción. El segundo tipo de error descrito ocurre durante el proceso de implementación de las pruebas, ya que pueden pasarse por alto ciertos aspectos que en un principio no habían sido considerados, pero tras múltiples ejecuciones nos damos cuenta de que son relevantes, lo que conlleva un pequeño retrabajo para mejorar las pruebas.

Tras aplicar esta metodología en la empresa, se implementaron (hasta el momento) 87 bancos de prueba aproximadamente, en la cual cada uno de estos incluye 4 pruebas aproximadamente. Esto se traduce en 348 pruebas que no deberán realizarse manualmente.

El tiempo estimado de unas pruebas de regresión manuales en el proyecto era aproximadamente de 180 horas, esto ocasionaba que el equipo de QA estuviera un par semanas enfocado solamente en las pruebas de regresión. Una vez implementadas las pruebas automatizadas estos tiempos se redujeron significativamente, pasamos de 180 horas de pruebas manuales a 5 horas de pruebas automatizadas.

En cuanto al tipo y cantidad de errores que se detectan varía según el escenario en que se encuentre el proyecto. Cuando debían ejecutarse las pruebas de regresión por un cambio de versión de framework se detectaban aproximadamente 80 bugs, mientras que, si era una incorporación de nuevas funcionalidades, es decir, que se ejecutaban las pruebas de humo se detectaban entre 10 y 15 bugs aproximadamente.

6.3 Dashboard de Cypress

Es una herramienta de gestión de información, en este caso gestión de los resultados obtenidos en nuestras pruebas. Ésta monitoriza y analiza ciertos indicadores o métricas como, por ejemplo: duración de las pruebas, el estado de las ejecuciones actuales o antiguas, entre otras (Figura 33). Esta es una herramienta que tiene funciones de pago.

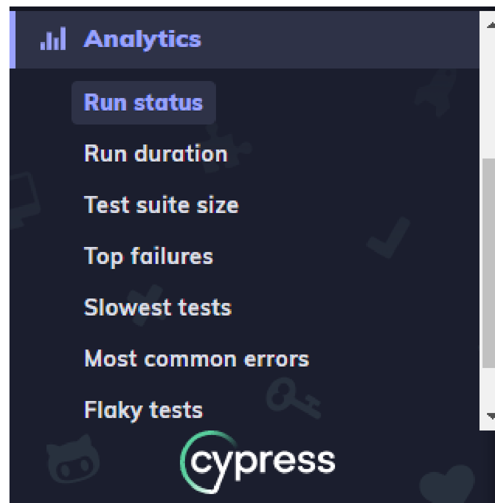


Figura 33: Analíticas del Dashboard de Cypress

De cada ejecución se pueden ver distintas métricas, entre los cuales están la cantidad de pruebas aprobadas, fallidas y saltadas. Además de poder acceder al desglose de cada una de las pruebas (Anexo I - 6), un video de su ejecución y en caso de fallo una captura de pantalla del momento preciso en el cual falló la prueba.

Figura 34: Resultado ejecución pruebas Dashboard parte 1

Como se puede observar en la Figura 34 cada ejecución tendrá asociado un resultado según los resultados de la ejecución (1), un nombre de *commit* el cual se mostrará como nombre de ejecución (2), la persona que ejecuta la prueba (3), además como se mencionó en el apartado 6.1 puede tener un tag asociado (4) y entre otras cosas la duración de la ejecución de la batería de pruebas.

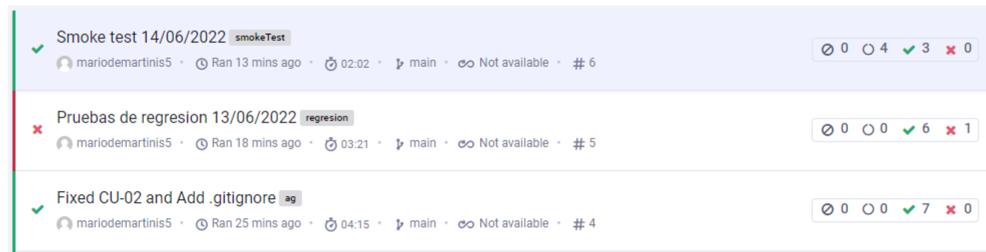


Figura 35: Histórico de ejecución

Tener acceso al histórico de resultados (Figura 35: Histórico de ejecuciónFigura 35) nos permitirá ver que pruebas suelen fallar y necesitan una corrección, además al tener asociado un nombre de commit específico nos permite saber qué cambios introdujeron los fallos en caso de tenerlos.

Tras analizar múltiples ejecuciones en las pruebas automatizadas para la empresa, nos encontrábamos con que existía un conjunto de pruebas que solía fallar debido a la forma en que se implementaron las mismas. Esto se debía principalmente a no se tomaban en cuenta llamadas asíncronas que realizaba la página web y se introducían esperas síncronas, es decir, se establecía una duración de 5 segundos en una prueba para esperar a un elemento, pero éste podía tardar más o menos, lo que ocasionaba que en ciertas ejecuciones fallara y en otras pasara.

Para solucionar lo anterior, se introducían esperas a las llamadas de las APIs de esta forma se establece un tiempo máximo superior, pero una vez recibía la respuesta de la llamada dejaba de esperar. Por lo tanto, si nuestra espera máxima era 10 segundos, si se ejecutaba en 2 nos permitiría ahorrar esos 8 segundos. Esto llevado a las múltiples esperas que hay durante una prueba nos permitiría ahorrar bastante tiempo.

7. Conclusiones

Para concluir esta memoria debemos recordar que el objetivo principal de este proyecto era definir una metodología de automatización de pruebas la cual se pudiera aplicar a cualquier proyecto software que esté desarrollando una aplicación web, con el fin de reducir el tiempo y coste del proceso de pruebas manuales. A pesar de que es necesario una persona con conocimientos técnicos para poder realizar esta automatización, la curva de aprendizaje es muy leve lo que supone un costo inicial muy bajo.

Se diseñó con el objetivo que se acople a la metodología ágil de un proyecto software. Específicamente está diseñada para realizarse durante la fase de Pruebas vista en el capítulo 2. Una vez el producto sale de la fase de Desarrollo, se debe evaluar si es una nueva funcionalidad, este caso se deben implementar ambos tipos de pruebas. Por el contrario, si es una funcionalidad antigua la cual ya se tiene automatizada, se procede directamente a la fase de ejecución.

Según el escenario en que nos encontremos se deberá ejecutar la batería de pruebas críticas o toda la batería de pruebas, esto dependerá de lo desarrollado durante esa iteración. Posterior a la ejecución de estas pruebas se deberá analizar los resultados obtenidos y en caso de que se detecten errores deberán ser reportados para el equipo de desarrollo los resuelva antes de que el producto pase a la fase de Entrega.

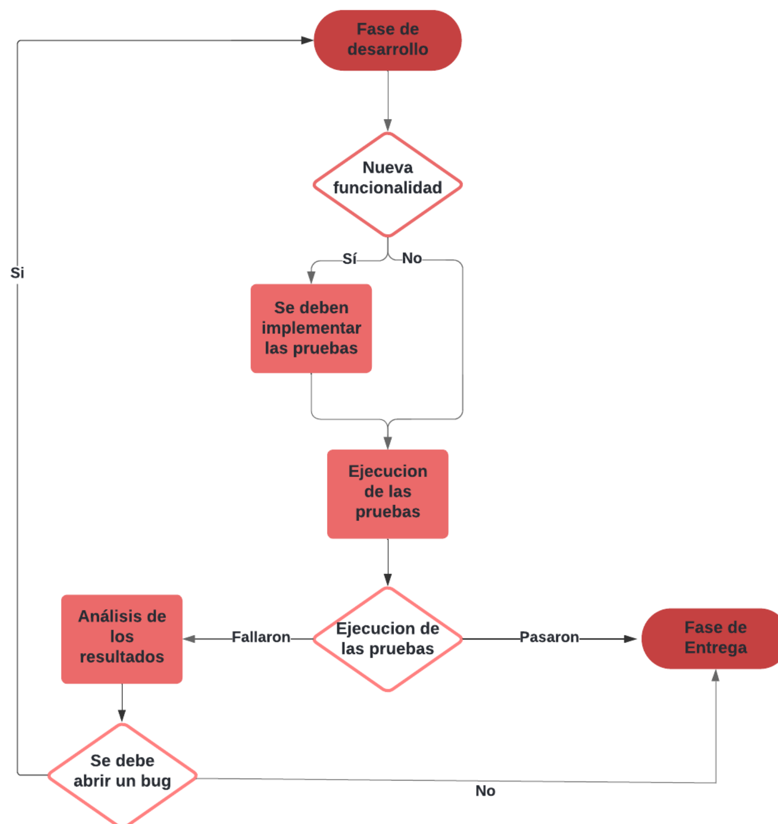


Figura 36: metodología durante la fase de Pruebas

Para lograr esto, primero se debió investigar el conjunto de herramientas disponibles en el mercado la cual presente beneficios para páginas web. Como se mencionó en el capítulo 3, Cypress tiene múltiples beneficios a la hora de realizar pruebas en este tipo de proyectos al ejecutarse sobre el propio navegador.

Luego se propuso definir una batería de pruebas críticas, las cuales debemos asegurar su funcionamiento en todo momento y otra batería de pruebas mucho más extensa que incluya las primeras y que además compruebe ciertas funcionalidades que no son prioritarias. Luego de implementar estas baterías de pruebas ejecutar una u otra según la situación en que se encuentre. Para esto definimos cuatro escenarios:

- Incorporación de nuevas funcionalidades.
- Cambios en funcionalidades antiguas.
- Subida de versión de producto.
- Cambio del entorno.

Como se menciona en el Prólogo este proyecto fue realizado durante el desarrollo de mis prácticas en empresa, en la cual se implementó esta metodología y se desarrollaron las pruebas automatizadas para el proyecto en el cual me encontraba. Se decidió implementar en un principio debido a la gran cantidad de trabajo manual que se debía realizar cuando se quería hacer una entrega al cliente. Esto ocasionaba que el equipo de QA invirtiera aproximadamente una semana y media en caso de tener que ejecutar la batería de pruebas completas o tres o cuatro días en caso de que fueran las pruebas más críticas.

Una vez implementadas las pruebas se pasó de estar una semana y media realizando las pruebas a un par de horas ejecutándose y una hora extra analizado los errores en caso de que existieran. Además, se detectaban errores que a lo mejor a una persona se le podían pasar, se realizaron pruebas en caso de que una API estuviera caída sin necesidad de tumbarla.

Debido a esto podemos resaltar la gran importancia que tiene la automatización de pruebas sobre todo en un proyecto largo en el cual se deban realizar múltiples entregas y requieran realizar muchas pruebas de regresión.

Trabajo futuro

La integración continua o CI es el proceso que se ejecuta cada vez que un desarrollador realiza un *commit*, sobre éste se realizan un conjunto de pruebas. Si se pasan todas las pruebas el código se añade, pero si algo llega a fallar se rechaza [7]. Normalmente se cree que en este proceso solo se realizan pruebas unitarias, pero esto no es correcto. Por lo tanto, el paso siguiente sería incorporar nuestra batería de pruebas al proceso de integración continua, de esta manera nos aseguramos de que solo se introduce código que funciona.

Relación del trabajo con los estudios cursados

Este trabajo guarda relación con múltiples asignaturas de la rama de Ingeniería del software, las cuales desglosaremos a continuación:

- **Calidad del software:** directamente relacionada con el control y la garantía de la calidad de un producto software, además de la definición de métricas y estándares para nuestro producto.
- **Mantenimiento y evolución del software:** la automatización de pruebas es un proyecto de software por lo tanto requiere de un mantenimiento y evolución del producto.
- **Proceso del software:** la implantación de la metodología de diseño para que pueda acoplarse al desarrollo ágil de un proyecto software.

Las asignaturas que guardan mayor relación con el proyecto son:

Análisis, validación y depuración de software ya que esta asignatura se analizan las distintas técnicas que existen para evaluar si un producto cumple los requisitos de calidad, estándares, además de ser capaz de utilizar herramientas para el análisis, validación y depuración del código.

Diseño del software en la cual analizamos las distintas arquitecturas software, a diseñar software de calidad siguiendo buenas prácticas. Además, de que en la automatización se pueden aplicar distintos patrones del diseño. A continuación, se implementará el patrón *Page Object Model*, el cual es un patrón el cual nos permite tratar páginas como objetos [13]. Este patrón tiene distintas ventajas:

- Nos facilita tanto el desarrollo como la comprensión del código implementado.
- Los casos de prueba se hacen más cortos.
- Podemos reutilizar los objetos de páginas.
- Cualquier cambio puede ser implementado fácilmente en un único punto.

Implementaremos el CU-02 para comprobar lo mencionado anteriormente, para esto debemos crear una clase para cada pantalla involucrada: página de inicio, página del perfil y la página de las direcciones, como se observa en la Figura 37.

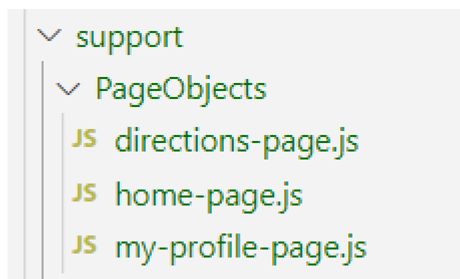


Figura 37: Páginas para implementar el patrón Page Object Model

Dentro de cada una de estas páginas definiremos los componentes que interactúan en nuestra prueba. Como, por ejemplo, los elementos definidos en la Figura 38: Archivo directions-page.js.

```
class DirectionsPage {
    getAddDirectionButton(){
        return cy.get('.button-address-left').click()
    }

    getFirstDirectionsCard(){
        return cy.get('.card.address-delivery:first()')
    }

    getConfirmButton(){
        return cy.get('.action-button-yes')
    }
}
export default DirectionsPage
```

Figura 38: Archivo directions-page.js

De esta manera solo debemos crear un objeto *DirectionsPage* en nuestra prueba y de esta manera ya tendremos acceso a los elementos de nuestra página, lo que hace que nuestra prueba sea más legible, como se observa en la Figura 39: Cláusula before() utilizando el patrón Page Object Model.

```
describe('CU-02 Añadir/Borrar direccion de entrega - Page Object Model', function () {

    before('Acceder a la pagina e iniciar sesion', function () {
        // Visitar pagina web
        cy.visit('https://tienda.consum.es/')
        // Aceptar todas las cookies
        homePage.getAcceptCookiesButton().click()
        // Iniciar sesion
        cy.fixture('cuenta/datos-acceso').then(function (datosAcceso) {
            cy.iniciarSesion(datosAcceso.user, datosAcceso.password);
        })
        cy.wait('@load', { timeout: 10000 })
        homePage.getMyProfileButton().click({ force: true })
        homePage.getMyAccountButton().click({ force: true })
        myProfilePage.getDirectionsButton().click({ force: true })
    })
```

Figura 39: Cláusula before() utilizando el patrón Page Object Model

En el anexo I - 7 y el anexo I - 8, se pueden observar el código utilizado para implementar el CU-02 con el patrón *Page Object Model*. Por último, en el anexo I - 9 se puede observar cómo se crean las instancias de cada una de las páginas.

8. Referencias

[1] I. Sommerville, Ingenieria del software, Madrid: Pearson, 2005.

[2] E. Larequi, «Asociacion de la Industria Navarra,» [En línea]. Available: https://www.ain.es/archivo-proyectos/metodologia-agile-tcar/. [Último acceso: Junio 2022].

[3] «IBM,» Abril 2022. [En línea]. Available: https://www.ibm.com/cl-es/topics/software-testing. [Último acceso: Abril 2022].

[4] «Selenium,» [En línea]. [Último acceso: Abril 2022].

[5] «Cypress.io,» [En línea]. Available: https://www.cypress.io/. [Último acceso: 10 Marzo 2022].

[6] Cypress, «Cypress,» [En línea]. Available: https://www.cypress.io/how-it-works. [Último acceso: Marzo 2022].

[7] «Star history,» [En línea]. Available: https://star-history.com/. [Último acceso: Marzo 2022].

[8] «QALOVERS,» [En línea]. Available: https://www.qalovers.com/2017/12/smoke-test.html. [Último acceso: Abril 2022].

[9] A. Axelrod, Complete Guide to Test Automation, APRESS, 2018.

[10] «Atlassian,» [En línea]. Available: https://www.atlassian.com/es/software/jira/features/scrum-boards. [Último acceso: Abril 2022].

[11] Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 2018.

[12] J. M. a. P. Wilton, Beginning JavaScript, Indiana: Wrox: Indianapolis, 2015.

[13] S. J. Millet, G. S. Lindhorst, J. P. Shewchuk, D. C. Johnson y J. M. Buehler, «Page object model». U.S. Patent Patente 7,284,193, 2007.

9. Anexo I – Figuras complementarias

1. Datos utilizados para el CU-01. Archivo cu-01.json.

```json
{
  "productos": [
    "Jamón Gran Reserva 18 Meses Duroc",
    "Leche Calcio sin Lactosa Desnatada Brik"
  ],
  "productoExtra": "Bloc de Cuadros Tapa Dura 80 Hojas"
}
```

2. Datos utilizados para el CU-02. Archivo cu-02.json.

```json
{
  "codigoPostal": "46015",
  "provincia": "valencia",
  "poblacion": "Valencia",
  "tipoVia": "Avenida",
  "via": "maestro rodrigo",
  "numero": "84"
}
```

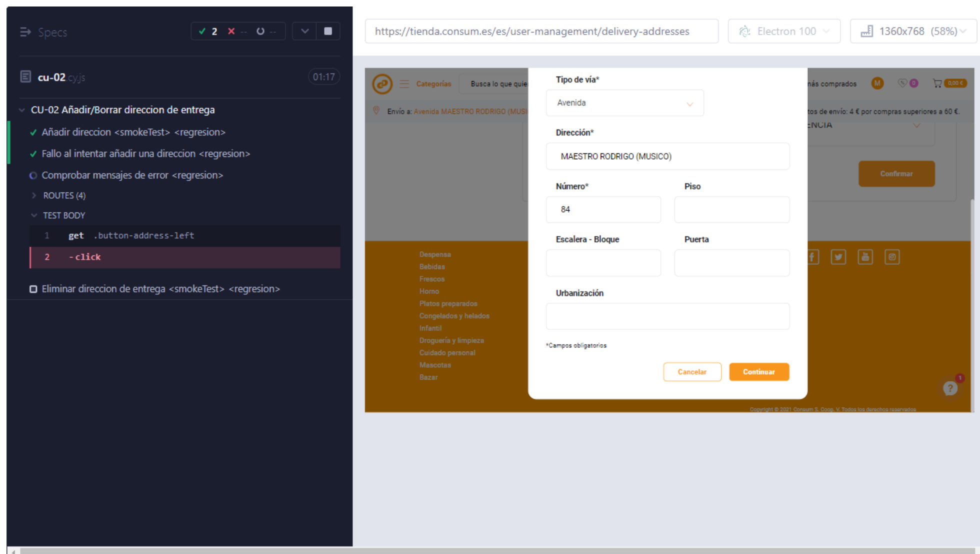
3. Comando Cypress para iniciar sesión.

```javascript
Cypress.Commands.add('iniciarSesion', (user, password) => {
    cy.intercept('GET', '/api/rest/V1.0/catalog/state/**').as('load')
    cy.intercept('POST', '/api/rest/V2.0/account/token').as('postToken')
    cy.get('.header__main .header__user [title=Usuario]').click()
    cy.contains('Acceder').click()
    cy.get('#userName').type(user)
    cy.get('#password').type(password)
    cy.get('#form-login--logo').click()
    cy.get('#form-login--btn-login').click()
    cy.wait('@postToken', { timeout: 10000 })
    cy.wait('@load', { timeout: 10000 })
})
```

4. Comando Cypress para añadir dirección.

```
Cypress.Commands.add('addDirection', (datos) => {
    // Completamos los campos obligatorios de los campos
    cy.get('[formcontrolname="zipCode"]').type(datos.codigoPostal)
    cy.wait(500).get('[formcontrolname="region"]+').click({ force: true })
    cy.get('[formcontrolname="region"] option:contains("' + datos.provincia.toUpperCase() + '")')
        .click({ force: true })
    cy.wait('@getCities', { timeout: 10000 })
    cy.get('[formcontrolname="city"]').select(datos.poblacion)
    cy.wait('@getRegions', { timeout: 10000 })
    cy.get('[formcontrolname="streetType"]').select(datos.tipoVia)
    cy.get('[formcontrolname="street"]').type(datos.via)
    cy.wait(500).get('.dropdown-item span:contains("' + datos.via.toUpperCase() + '")').click()
    cy.get('[formcontrolname="number"]').type(datos.numero)
    cy.get('.required-fields--label').click()
    // Seleccionamos Continuar
    cy.get('#purchase-address-adduseraddress-content-button--continue').click()
})
```

5. Captura de pantalla del momento en que falla una prueba.



6. Desglose de una ejecución

7. CU-02 Añadir dirección de entrega utilizando el patrón *Page Object Model*

```
it('Añadir direccion <pattern>', function () {
    cy.intercept('GET', '**/deliveryaddress').as('getAddress')
    cy.intercept('GET', '/api/rest/V1.0/commons/country/**').as('getRegions')
    cy.intercept('GET', '/api/rest/V1.0/commons/city/**').as('getCities')
    cy.intercept('POST', '**/deliveryaddress').as('postAddress')
    // Selecionamos Agregar direccion
    directionsPage.getAddDirectionButton().click()
    cy.wait('@getRegions', { timeout: 10000 })
    // Llamamos a nuestro comando para agregar una direccion
    cy.addDirection(this.datos)
    cy.wait('@postAddress', { timeout: 5000 }).then(function (intercept) {
        assert.equal(intercept.response.statusCode, 201)
    })
})
```

8. CU-02 Borrar dirección de entrega utilizando el patrón *Page Object Model*

```
it('Eliminar direccion de entrega <pattern>', function () {
    cy.intercept('DELETE', '/api/rest/V1.0/user/**').as('deleteAddress')
    // Seleccionamos la primera direccion
    directionsPage.getFirstDirectionsCard()
        .find(' cmp-icon .iconAngular-up ').click()
    // Selecionamos el icono de la papelera
    directionsPage.getFirstDirectionsCard()
        .find('.iconAngular-remove').click()
    // Confirmamos el borrado
    directionsPage.getConfirmButton().click()
    cy.wait('@deleteAddress', { timeout: 5000 })
})
```

9. Crear instancias de las páginas

```
import MyProfilePage from '../support/PageObjects/my-profile-page'
import DirectionsPage from '../support/PageObjects/directions-page'
import HomePage from '../support/PageObjects/home-page'
const myProfilePage = new MyProfilePage()
const directionsPage = new DirectionsPage()
const homePage = new HomePage()
```

10. Anexo II - Objetivos de desarrollo sostenible

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**No Procede**
ODS 1. **Fin de la pobreza.**				**X**
ODS 2. **Hambre cero.**				**X**
ODS 3. **Salud y bienestar.**				**X**
ODS 4. **Educación de calidad.**				**X**
ODS 5. **Igualdad de género.**				**X**
ODS 6. **Agua limpia y saneamiento.**				**X**
ODS 7. **Energía asequible y no contaminante.**				**X**
ODS 8. **Trabajo decente y crecimiento económico.**		**X**		
ODS 9. **Industria, innovación e infraestructuras.**				**X**
ODS 10. **Reducción de las desigualdades**.				**X**
ODS 11. **Ciudades y comunidades sostenibles**.				**X**
ODS 12. **Producción y consumo responsables.**		**X**		
ODS 13. **Acción por el clima.**	**X**			
ODS 14. **Vida submarina.**				**X**
ODS 15. **Vida de ecosistemas terrestres.**				**X**
ODS 16. **Paz, justicia e instituciones sólidas.**				**X**
ODS 17. **Alianzas para lograr objetivos.**				**X**

A continuación, se explica la relación que guarda este TFG con los ODS listados anteriormente:

- **Trabajo decente y crecimiento económico:**
    - La meta 8.2 consiste en "Lograr niveles más elevados de productividad económica mediante la diversificación, la modernización tecnológica y la innovación." Por lo que considero personalmente que el proyecto promueve esta meta ya que se busca mejorar la productividad, mediante la automatización de pruebas.
- **Producción y consumo responsable:**
    - Considero que guarda relación con la meta 12.6 "Alentar a las empresas, en especial las grandes empresas y las empresas transnacionales, a que adopten prácticas sostenibles" ya que al automatizar procesos que realizados de manera manual conllevarían el uso de una mayor cantidad de recursos.
- **Acción por el clima:**
    - Considero que tiene mucha relación con este punto ya que al automatizar pruebas nos permite ahorrar tiempo que se traduce en ahorro de energía y recursos que es precisamente lo que promueve este objetivo de desarrollo sostenible.

11. Glosario

API
Application programming interfaces o Interfaz de programación de aplicaciones.....15
Backend
Logica que se encarga de que una Página web funcione 16
Batch
También conocido como procesamiento por lotes, consiste en la ejecución de un programa sin supervisión directa del usuario 21
CamelCase
Estilo de escritura para palabras compuestas. En el cual la primera letra de cada una de las palabras va en mayúscula con excepción de la primera 36
Commit
Es una captura instantánea de los cambios, se consideran versiones estables.......... 43
Dashboard
Es una herramienta de gestion de informacion 39
Framework
Un esquema o marco de trabajo que ofrece una estructura base para elaborar un proyecto....................16
Frontend
Es la parte de la aplicación con la que interactúa y ve el usuario 21
Kebab-case
Estilo de escritura para oraciones. En el cual se substituyen los espacios " " por guiones "-".................... 36
Librerías
Conjunto de archivos que proporcionan diversas funcionalidades 16
Pruebas E2E
Tipo de prueba que consiste en probar la aplicación desde el punto de vista del usuario final 16
Pruebas unitarias
Son pequeñas pruebas de porciones de código, diseñado para comprobar que el código principal funciona 45
QA
Quality assurance o Aseguramiento de la calidad.................... 9
Repositorio
Sitio web donde se almacena informacion digital....................18
Script
Fragmento de código, con el objetivo de realizar funciones 37
Software
Un programa o conjuntos de programas 10
Tablero Kanban
Nos permite vizualizar el flujo de trabajo.................... 22
Tablero scrum
Nos permite ver el trabajo a realizar por el equipo..................... 22
Tester
Persona encargada de probar software....................10