UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

# Mejora del sistema de pruebas de software de una empresa de soluciones informáticas para la industria del cerramiento

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Moreno Palacio, Álex

Tutor/a: Villanueva García, Alicia

Cotutor/a externo: FRECHINA NAVARRO, FRANCISCO

CURSO ACADÉMICO: 2021/2022

# Resumen

Las pruebas de *software* o *Testing*, son las investigaciones empíricas y técnicas que tienen como objetivo proporcionar información objetiva e independiente sobre la calidad del producto, para así, verificar si el comportamiento del sistema es el deseado o no. Existen muchos tipos diferentes de pruebas de *software* con objetivos y estrategias diferentes, como las de aceptación, las de integración, las unitarias, etc.

En este trabajo se aborda la mejora de una *suite* de pruebas de interfaz de usuario para la empresa Preference S.L., con la pretensión de disminuir sus tiempos de ejecución de forma considerable, bajo una implementación libre de *anti-patterns* dentro de lo posible.

Actualmente la empresa usa el *software* de pago *Ranorex* para sus pruebas de interfaz de usuario, con un tiempo de ejecución total de 6 h 16 min 14 s, al igual que utiliza *MSTest* para las pruebas unitarias de .NET en C#, con un tiempo de ejecución de 1 min 15 s. Por ello este proyecto aborda la mejora de las pruebas de IU, ya que el tiempo de las unitarias no es tan costoso.

Todos los inconvenientes extraídos en la fase del análisis, además de la investigación sobre los *frameworks* de pruebas extremo a extremo relevantes en la actualidad, derivaron en el empleo de *Cypress* (un *framework* para pruebas de interfaz de usuario) como herramienta de ejecución de test. Posteriormente, se complementó la herramienta con una librería llamada *Testing-Library*, que añadía la posibilidad de que las pruebas se centrasen aún más en el usuario, consiguiendo testear código que antes no se testeaba.

Tras la implementación de la nueva *suite* de test, la mejora fue drástica, ya que se mejoraron en un 81% los tiempos de ejecución sin tan siquiera llegar a paralelizar. Calculando que con una paralelización de 4 equipos se podría conseguir la drástica mejora de un 90% con respecto a los tiempos iniciales de ejecución.

Es por esto por lo que el proyecto se ha considerado como un éxito para la empresa, y que, se mantendrá en período de prueba junto con la anterior *suite* de test para comprobar a largo plazo que efectivamente se llevan a cabo de forma correcta estas pruebas.

Palabras clave: automatización de pruebas, refactorización, antipatrones, pruebas de extremo a extremo y buenas prácticas.

# Abstract

Software testing, are empirical and technical investigations that aim to provide objective and independent information on the quality of the product, in order to verify whether the behavior of the system is as desired or not. There are many different types of software tests with different objectives and strategies, such as acceptance tests, integration tests, unit tests...

The intention of this project is to improve the user interface test suite for the company Preference S.L., with the aim of reducing its execution times considerably, under a free implementation of anti-patterns.

The company currently uses the paid software Ranorex for its user interface tests, with a total execution time of 6h 16min 14s, as well as using MSTest for .NET unit tests in C#, with a time of execution of 1 min 15 s. For this reason, this project addresses the improvement of UI tests, since the duration of the unit tests is not so extensive.

All the drawbacks extracted in the analysis phase, in addition to the research on the currently relevant end-to-end testing frameworks, led to the use of Cypress (a user interface testing *framework*) as a test execution tool. Subsequently, the tool was complemented with a library called Testing-Library, which added the possibility of the tests being even more focused on the user, managing to test code that was not tested before.

After the implementation of the new test suite, the improvement was drastic, since execution times were improved by 70% without even parallelizing. Calculating that with a parallelization of 4 computers, a drastic improvement of 90% could be achieved regarding the initial execution times.

This is why the project has been considered a success for the company, and that it will remain in a trial period combined with the previous test suite to verify in the long term that these tests are actually carried out successfully.



Keywords: test automation, refactor, anti-patterns, end-to-end testing and good practices.

# Tabla de contenidos

# Índice de ilustraciones

# Índice de tablas

# Glosario

- Pruebas E2E: Pruebas de extremo a extremo o *end-to-end*.
- Pruebas de extremo a extremo: aquellas pruebas que imitan el funcionamiento del *software* en la vida real mediante la ejecución de flujos de trabajo, que emulan escenarios comunes a los que se enfrentan los usuarios o clientes.
- DOM: acrónimo del término inglés *Document Object Model* o en castellano, Modelo de Objeto de Documento en castellano, que se define como una interfaz de programación para los documentos HTML y XML que facilita la representación estructurada del documento y define de qué manera los programas pueden acceder, a fin de modificar, tanto su estructura, como su estilo y contenido.
- POM: acrónimo del término inglés *Page Object Model* que se define como el patrón de diseño donde cada una de las páginas de una web se representan como clases, y los diversos elementos de la página se fijan como variables en la clase.
- *Fixtures* (Accesorios): carpeta del proyecto en la que se almacenan todos los datos respectivos a cada una de las pruebas.
- IDE: acrónimo del término inglés *Integrated Development Environment*, o lo que es lo mismo, Entorno de Desarrollo Integrado. Es el escenario digital que se ocupa de facilitar las tareas del programador añadiendo herramientas, compiladores, depuradores o bibliotecas, aumentando así la productividad de este.
- *Dashboard* o cuadro de mandos: herramienta de gestión que facilita la toma de decisiones y que recoge un conjunto coherente de indicadores que proporcionan una visión comprensible de su área de responsabilidad. La información aportada por el cuadro de mandos permite enfocar y alinear los equipos directivos, las unidades de negocio, los recursos y los procesos con las estrategias de la organización.
- Modo de ejecución *Headless*: se refiere a una ejecución de los test con el navegador lanzándose en segundo plano, sin ser visible para el usuario, es decir, solo se verían las descripciones de los test y si pasan dichos test o no.
- *Code smell*: la hediondez del código (code smell en inglés, o también conocido por código que huele o apesta) es cualquier síntoma en el código fuente de un programa que posiblemente indica un problema más profundo. Estos suelen indicar deficiencias en el diseño *software* que puede ralentizar el desarrollo o aumentan el riesgo de errores o fallos en un futuro.
- TDD: del inglés *Test-driven development*, es un proceso de desarrollo de *software* basado en que los requisitos del programa se conviertan directamente en casos de prueba, antes de que se llegue a desarrollar un ápice de este. Esto permite probar en cualquier momento el *software* contra los casos de prueba previamente establecidos hasta que se pasen todos los test. Este concepto es justo lo opuesto a programar primero y posteriormente hacer los test.

# 1. Introducción

Preference S.L. es una empresa que está enfocada en aportar soluciones informáticas para la industria del cerramiento, ventanas, puertas y fachadas desde el año 1994. Dicha empresa aporta las soluciones mediante aplicaciones de escritorio encargadas de simplificar y automatizar la cadena de suministro de productos configurables tanto en el mercado de obra nueva como en el de la distribución.

Aproximadamente en 2007, la empresa inició el proyecto PrefWeb, que se encarga de facilitar todas las herramientas proporcionadas por la aplicación de escritorio, pero, directamente en la web. El proceso hasta conseguir una funcionalidad similar a la aplicación ya existente ha sido largo y tedioso, debido a la carga computacional, entre otras dificultades.

Es por eso por lo que, ante el poco presupuesto, la escasez de personal, la poca experiencia de éste y la velocidad de adaptación de un tipo de tecnologías a otro, se dejaron de lado las pruebas de *software*. Poco a poco fueron añadiendo pruebas unitarias que verificaban el funcionamiento de ciertas funciones relevantes, pero no contaron con las pruebas de extremo a extremo hasta 2019.

A partir de 2019, se fueron automatizando pruebas centradas en la interfaz de usuario, con la pega de que estaban usando un *software* de pago (Ranorex) con ciertas limitaciones. Esto se debe a una mala toma de decisiones que repercutieron en la futura consistencia de la aplicación web.

Consecuentemente, tras ver este problema, se hizo un breve análisis para estimar los principales problemas y tener una idea de todo aquello que se podría mejorar, las tecnologías que más se adecuarían, etc. Los principales problemas que se encontraron tras el análisis inicial, algunos ya conocidos por la empresa, fueron los siguientes:

1. La duración de la *suite* de test. Su ejecución completa puede llegar a extenderse hasta las 6 h 30 min. Esto supone un coste excesivo y perjudica tanto a los programadores como al usuario final.
2. Dependencia de los test con el código de la aplicación. Teniendo en cuenta que son pruebas centradas en el usuario, éstas deberían de probar los flujos de trabajo de la misma forma que lo haría un usuario real.
3. La limitación de un *software* de pago. Ranorex es un *software* de pruebas de GUI con ciertos inconvenientes, que se verán más adelante en el punto 3: “Análisis del problema”.
4. Antipatrones. Comúnmente nos encontramos con código o acciones que nos llevan a una mala solución para el problema planteado. Esto es bastante común en el código implementado actualmente, aunque en ocasiones por razones externas al proyecto.

Todos estos problemas nos confirman el gran margen de mejora que tiene del código de pruebas, tanto en la parte de los tiempos de ejecución, como en la de las buenas prácticas, pudiendo optar a que se testeen flujos de una forma más completa y en un tiempo considerablemente menor.

## 1.1 Motivación

Los errores humanos pueden causar defectos o fallos en cualquier fase del ciclo de vida del desarrollo de *software*. El resultado podría llegar a ser catastrófico, dependiendo de la magnitud del error, además de la posibilidad de causar que el *software* se vuelva inconsistente y poco predecible.

Esto conduce directamente a una mala experiencia de usuario debido a los problemas que se podría encontrar, por ejemplo, para realizar tareas diarias que en teoría resultan sencillas. Por ende, el testing se convierte en una baza fundamental para el *software* actual, y que, si no se usa o se aplica mal, puede traer complicaciones a las empresas.

Dicho esto, la motivación estaba clara desde un primer momento. Había un gran interés en conseguir que las aplicaciones con las que se mantuviese contacto estuviesen bien estructuradas y testeadas, para así garantizar en medida de lo posible el correcto funcionamiento del *software*. Todo esto determinó el enfoque actual del proyecto, conseguir que la empresa Preference S.L. tuviese un *software* consistente y escalable, con una ejecución constante de pruebas para garantizar en medida de lo posible su correcto funcionamiento y así, evitar errores en producción (mucho más difíciles de controlar y perjudiciales para la reputación de la empresa).

Dejando de lado la motivación por mejorar la forma de programar, también está la motivación por conocer y aprender. En muchas ocasiones se ha oído hablar sobre lo importante que es testear código, sus grandes beneficios, etc. El problema es que, durante la etapa de aprendizaje, apenas se araña la superficie de los test, causando que no se preste la suficiente atención a lo que supone un gran porcentaje de la programación. Es por eso por lo que el proyecto no podía hacer más que ir enfocado a aquello que causaba tanto enredo.

## 1.2 Objetivos

Este proyecto se caracteriza por tener unos objetivos claros desde el primer momento, contando con el apoyo de la empresa para llevar a cabo y cumplir todo lo que se va a describir a continuación.

La empresa Preference S.L. cuenta con una *suite* de pruebas de interfaz de usuario que realiza test de su *software PrefWeb*, caracterizado por ser altamente personalizable, suponiendo ya de base un problema para reutilizar y unificar las pruebas.

La *suite* de pruebas se divide en siete bases de datos diferentes (cada base de datos es una réplica del estado en el que algunos clientes trabajan con la aplicación, ya que ésta es muy flexible y, dependiendo de la configuración, la web puede llegar a cambiar hasta un 35%), resultando en un tiempo de ejecución total de: 6 h 16 min 14 s.

Cada base de datos distinta implica un contenido desigual entre *PrefWebs* (así se le llama al conjunto de aplicaciones provenientes de PrefWeb), creando la necesidad de hacer pruebas comunes para las partes que se repiten y, pruebas individuales y únicas, pera testear un flujo de trabajo que solo se da en ciertas PrefWebs.

Se ha llegado a la conclusión de que no se puede evitar el hecho de utilizar distintas bases de datos, aunque solo cambien configuraciones de la aplicación que la inicializan según las preferencias del cliente.

Dado que la *suite* de test anterior es muy extensa, y la finalidad del proyecto es mostrar que se puede mejorar en gran medida mediante el uso de nuevas tecnologías, buenas prácticas en los test (por ejemplo, no testear aquello que ya se ha testeado una vez) y paralelización, tan solo se considerará una base de datos, es decir, una PrefWeb, para la ejecución y mejora de los test.

Teniendo claro el contexto, se plantean los siguientes objetivos.

- Objetivo 1: Reducir drásticamente los tiempos de ejecución totales de la *suite* de pruebas de una base de datos concreta, estimando una reducción de hasta el 70%. Con esto se pasaría de 1h 35 min 14 s a unos 16 min.

  Para ello se realizarán tareas como:

  - Estudiar y escoger entre las diferentes tecnologías actuales con las siguientes características: velocidad de ejecución, posibilidad de automatización, modo de ejecución *headless* (quiere decir que se prueba la aplicación sin llegar a ver su interfaz gráfica), etc.
  - Reestructurar las pruebas para evitar que se testee varias veces el mismo código.
  - Acelerar las pruebas al reutilizar las credenciales de inicio de sesión. (iniciar sesión una vez y testearlo para, a partir de ahí, copiar los tokens generados y usarlos en el posterior testeo o navegación de la aplicación).
  - Revisar la forma en la que se inicia sesión y en la que se cargan los datos, ya que se pierde mucho tiempo y se podría reemplazar por llamadas a la API.
  - Preparar el código para su posterior paralelización. Esta mejora puede llegar a reducir los tiempos hasta un 95% de las 1 h 35 min 14 s iniciales, es decir, dejar los tiempos en unos 3 min.
- Objetivo 2: Preparar el código para que no sea tan susceptible al cambio.

  Esto se pretende conseguir siguiendo buenas prácticas a la hora de programar las pruebas de interfaz de usuario.

  - Evitar el DOM. Esta es una de las principales razones por las que se ha optado a utilizar la librería Testing Library, la cual nos proporciona métodos para consultar semánticamente elementos del DOM y así conseguir probar la aplicación de una manera mucho más accesible.
  - Reutilizar código. Se pretende usar el patrón de diseño Page Object Models (POM) para lograr una mejor estructuración del código.
  - Separar las pruebas de la base de datos. Esto quiere decir que cualquier modificación que hagamos en la base de datos, no puede influir en el resultado del resto de pruebas, ya que llevaría a resultados condicionales y poco fiables.
- Objetivo 3: Aplicar buenas prácticas.

  Esto hará que el código sea fácil de entender, de manipular y de ampliar. Aunque sea muy general, las buenas prácticas del proyecto se han basado en un documento bastante interesante escrito por Yoni Goldberg (Goldberg, 2022).

  - Evitar *Fixtures* globales. Todos los datos usados en cada prueba deberán aislarse del resto.

- Elegir una estrategia clara de limpieza de datos. Before-All, After-All, restauración de la base de datos, etc.
- Centrarse en consultar aquellos elementos del DOM que es poco probable que sufran cambios, o de lo contrario, si cambian, se espera que falle el test (como los nombres de los inputs, botones, columnas, etc.)
- Utilizar la API en aquellos casos en los que ya se ha testeado la acción a realizar y, el principal motivo de la acción es crear algo, navegar o similares.
- Crear una prueba que navegue a través de toda la web. Esto evitará que tengamos que navegar posteriormente en los test de forma manual.

## 1.3 Convenciones

La memoria incluirá las siguientes normativas de marcado con significado adicional:

- Se optará por emplear la cursiva en el caso de los extranjerismos que no tengan traducción ni adaptación al castellano.
- El código fuente se muestra en letra Courier New. Y sólo se empleará esta tipología para este tipo de contenido

# 2. Estado del arte

Dado que este proyecto ha sido realizado en el ámbito de una práctica en empresa, con el correcto consentimiento del tutor, dedicaré el punto 2.1 a presentarla, junto con el producto que desarrolla, resaltando la importancia del trabajo realizado en el proyecto para esta.

En los siguientes puntos, también se dejarán por escrito aquellos programas existentes que desempeñan una función similar a Cypress, el *framework* empleado en este proyecto, y que se encuentran en la actualidad (año 2022), al igual que se evidenciará una crítica del estado del arte y demás.

## 2.1 Presentación de la empresa

Preference [1] es una empresa de *software* nacida en el año 1994, concebida para aportar soluciones globales, escalables y modulares de *software* específicas para la industria del cerramiento acristalado, haciendo hincapié siempre en la innovación. La empresa ha desarrollado una *suite* de programas que abarcan desde presupuestos, facturas, compras, almacén, planificación, producción, expediciones, automatización, conexión con maquinaria, monitorización, distribución, etc.

Estos serían todos los materiales productos con los que el *software* es familiar:

Materiales:

- PVC
- Aluminio
- Madera
- Vidrio

Productos:

- Ventanas
- Puertas
- Persianas
- Fachadas
- Escaparates
- Vidrio (Simple/IG/Georgian Bars)

Esta empresa, se caracteriza por tener diversos programas, cada uno de ellos especializado en una rama distinta de todas las soluciones que plantea Preference. En concreto, nos centraremos en PrefWeb, que es la solución *cloud* de la empresa.

PrefWeb aporta las siguientes características a sus clientes:

- Gráficos en 3D: los gráficos en 3D son generados y renderizados en tiempo real usando los servidores de la empresa para resolver todos los cálculos para que las figuras carguen en un instante, sin suponer una carga de datos pesada para el usuario final.
- Analíticas: aporta paneles *Power Bi* para las estadísticas de ventas, desempeño de distribuidores y tendencias de usuarios: mes a mes, año a la fecha, tasas de conversión…

[1] Vídeo sobre Preference SL: https://www.youtube.com/watch?v=dwW5xcaYUhY

- Configurador HTML5: un cliente podrá acceder al configurador de ventanas con cualquier dispositivo que tenga acceso a la web, ya sea un PC, portátil, tableta o teléfono.
- Optimización para ventas: un usuario puede actualizar dinámicamente los precios, descuentos, certificaciones y la información del producto. Todo esto mientras se está configurando junto al cliente.

Es por esto por lo que, teniendo una gran carga de clientes y una reputación que mantener, el proyecto de mejorar el sistema de pruebas de *software*, rehaciéndolo a sus necesidades y adaptándolo a las tecnologías punteras de la época, tiene sentido. Aunque, no todo son ventajas, ya que la aplicación está en marcha desde hace muchos años ya y el proyecto inicial no era el mismo que en lo que se ha convertido este *software*. Esto lleva a soluciones temporales, parches que se podrían resolver con mejores prácticas y demás, que dificultarán más adelante el proceso del testing.

## 2.2 *Software* similar al empleado

A continuación, se mencionan brevemente algunas de las herramientas de testing E2E más populares en la actualidad (Tozzi, 2022):

- WebdriverIO[2]: Es un *framework* hecho para testear aplicaciones web y móviles modernas, simplificando la interacción con el *software* a testear y otorgando diversos *plugins* que ayudan a crear una *suite* de test robusta y altamente escalable. A diferencia de Cypress, WebdriverIO no ofrece ninguna opción de pago, es decir, es completamente de código abierto.
  Algunas desventajas podrían ser:
  - Confusión en la sintaxis de las llamadas Async-await.
  - Dificultad para *debuggear* los test comparándolo con Cypress.
  - Demasiadas personalizaciones que pueden llegar a ser abrumador el tener que preparar todo el código antes de empezar.

Dicho esto, como la idea principal era obtener un *framework* capaz de ejecutarse en Chromium y, además tener un panel donde se pudiesen ver todas las estadísticas, ejecuciones en paralelo, mejoras y demás, WebdriverIO no daba la talla, así que fue descartado. Un ejemplo de cómo se vería el código escrito en WebdriverIO es el mostrado en la Ilustración 1.

[2] Página oficial: https://webdriver.io/

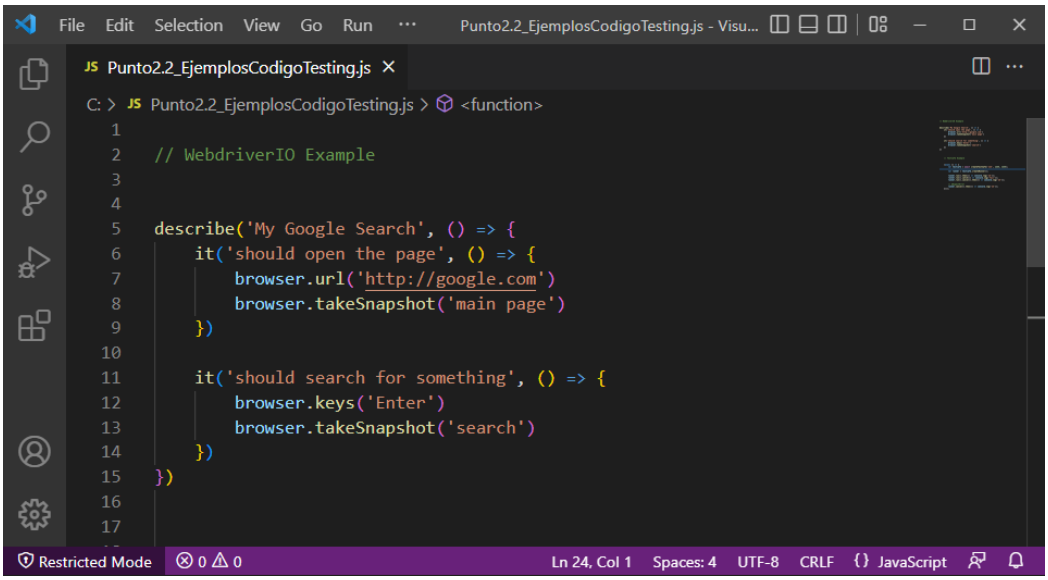



*Ilustración 1 – Ejemplo de código usando WebdriverIO*

- TestCafe[3]: *Framework* de automatización de pruebas en JavaScript que se ejecuta sobre la plataforma NodeJS que se usa principalmente para pruebas E2E, aunque también se puede usar para testear API. TestCafe no tiene una ventana propia para la depuración tal y como tiene Cypress, pero proporciona la opción modo en vivo que funciona lo suficientemente bien para la depuración (al menos para un usuario experimentado que ya conoce el entorno). Tal y como se observa en la Ilustración 2, no resulta fácil de leer o entender el funcionamiento (al menos en un primer vistazo)

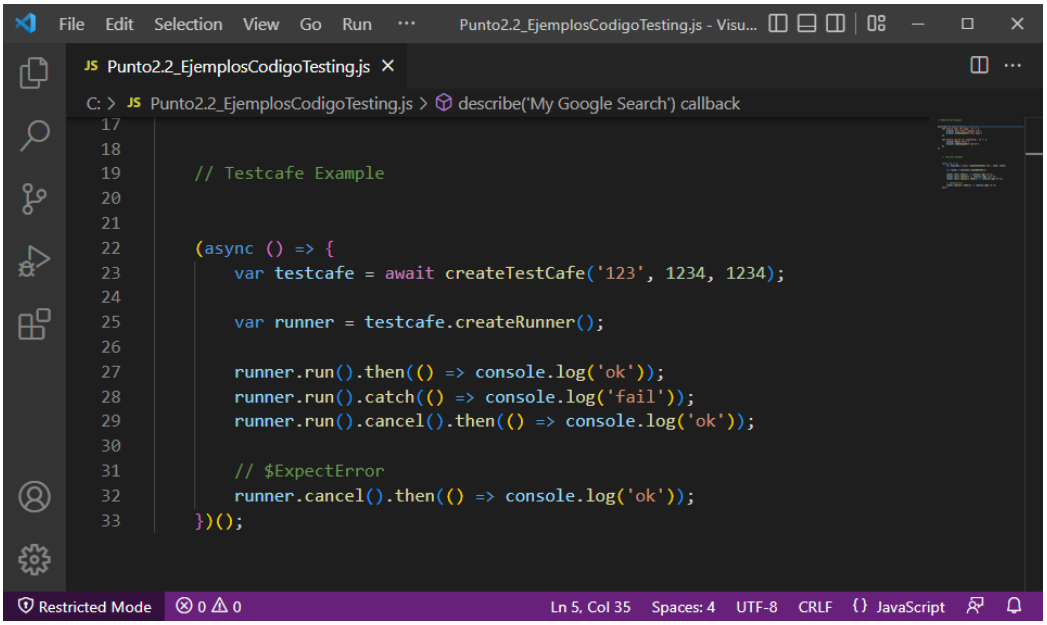



*Ilustración 2 - Ejemplo de código usando TestCafe*

[3] Página oficial: https://testcafe.io/

- Playwright[4]: es una librería NodeJS creada para la automatización de pruebas sobre navegadores que hacen uso de Chromium, WebKit y Firefox. Esta fue desarrollada por Microsoft con algunos programadores de puppeteer, otro *framework* de testing creado por Google.

  Con playwright, se pueden realizar las pruebas con C#, JavaScript, TypeScript, Python o Java, algo un tanto inusual si lo comparamos con los otros *frameworks* que hemos nombrado.

  El principal motivo por el cual no se eligió esta opción, fue principalmente por el poco tiempo que lleva en el mercado, ya que aún está en desarrollo. Dicho esto, aun así, es una muy buena alternativa a Cypress. Como se puede observar en la Ilustración 3, playwright usa un código bastante *verbose* y similar al de Cypress.

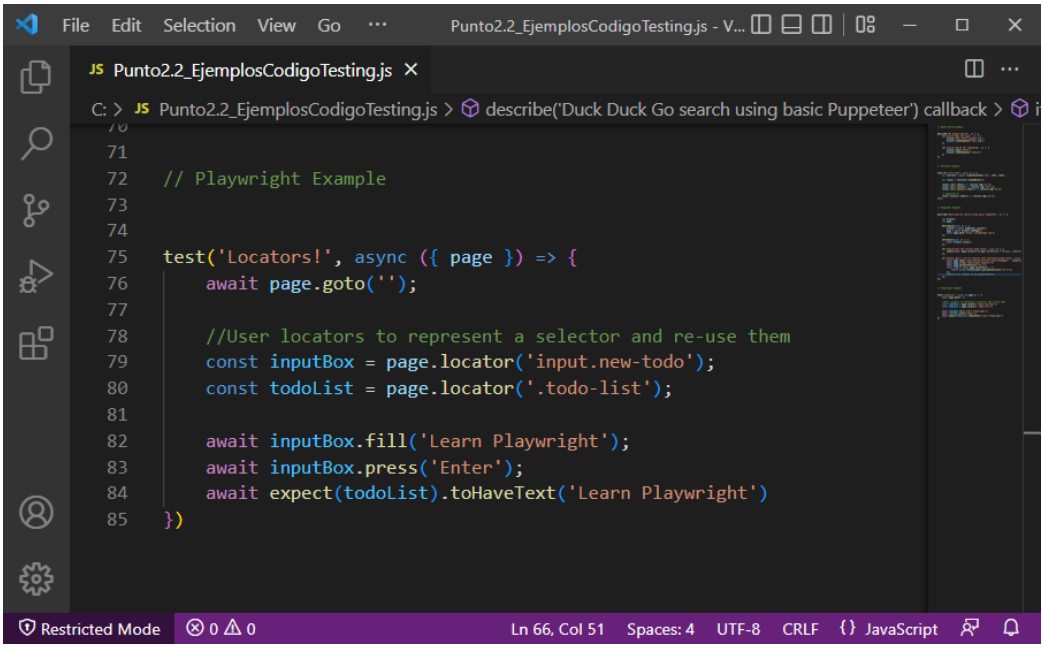



*Ilustración 3 - Ejemplo de código usando Playwright*

- Puppeteer[5]: al igual que Playwright, es una librería NodeJS que proporciona una API de alto nivel que permite automatizar acciones sobre los navegadores de Google: tanto Chrome como su versión de código abierto Chromium.

  Algunas de las ventajas que aporta esta librería son:

  - Permite acceder a la medición de tiempos de carga y renderizado que proporciona la herramienta de Performance Analysis de Chrome.
  - Permite más control sobre Chrome que Selenium WebDriver (no es el mismo que WebDriverIO)

[4] Página oficial: https://playwright.dev/
[5] Página oficial: https://pptr.dev/

- No necesita referenciar a un driver externo para ejecutar los test (esto solo es una ventaja comparándolo con Selenium, ya que los demás *frameworks* tampoco necesitan referenciar a un driver externo)

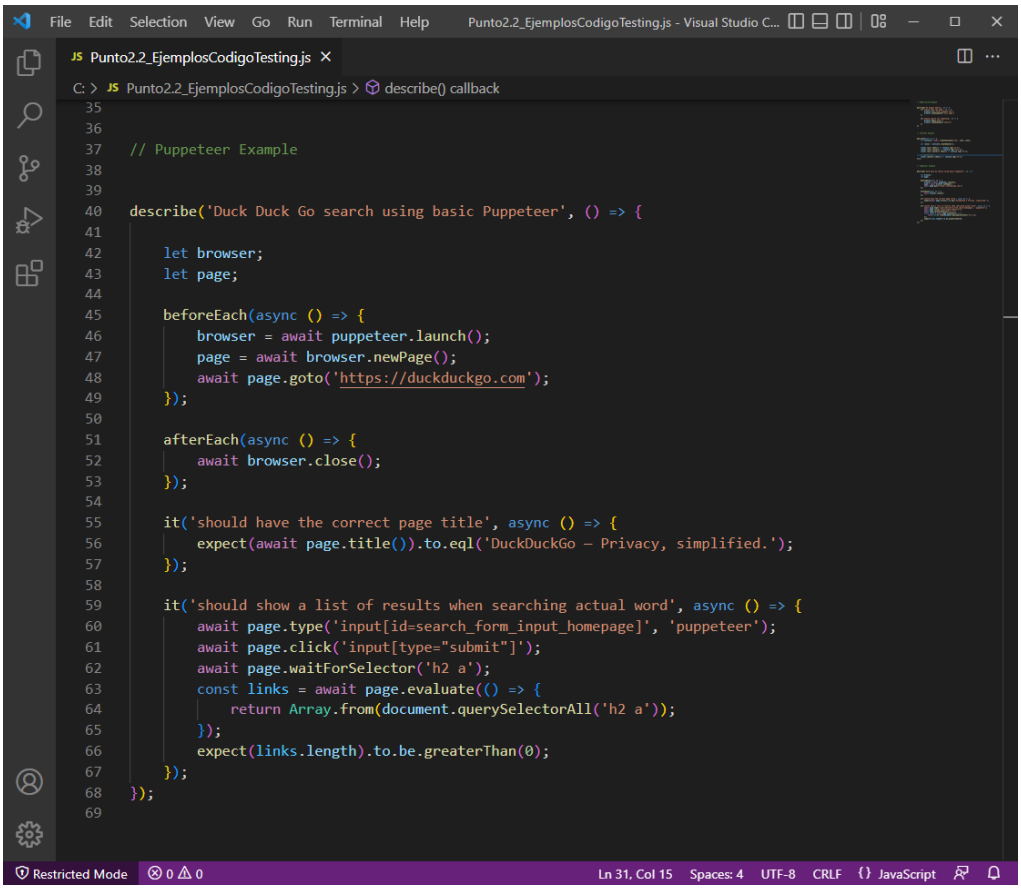



*Ilustración 4 - Ejemplo de código usando Puppeteer*

- Cypress: es una herramienta de testing construida para la web moderna que nos permite escribir todo tipo de test: End-To-End, de integración, unitarios, etc.

  Las principales funcionalidades con las que cuenta Cypress son las siguientes:
  - *Time travel*: Cypress toma *snapshots* o capturas de pantalla mientras los test están ejecutándose. Luego, en cualquier momento al terminar la ejecución, podemos volver a esos instantes y ver los valores del DOM, del *command log* y demás, simplemente clicando el evento del test que nos interese. Esto puede ser muy interesante para evolucionar las pruebas y comprobar ciertos valores de interés.
  - *Dashboard*: servicio que nos da acceso a todas las pruebas realizadas; por ejemplo, los vídeos, las estadísticas de todas las ejecuciones anteriores, capturas de pantalla, etc.
  - *Headless*: funcionalidad que nos permite correr nuestros test en un segundo plano, ocultando así el navegador.

  Nombradas ya algunas de las ventajas, las principales limitaciones serían:
  - Un soporte limitado para probar aplicaciones móviles (de poco interés para el proyecto, ya que solo se quiere testear sobre ordenadores)

- Los *iframes* (o marcos incorporados) en Cypress son difíciles de testear, con lo que si hiciese falta esta característica habría que recurrir a plugins (no es el caso, ya que no se utilizan *iframes* en la aplicación).

Todas estas ventajas y desventajas se han recopilado a través de un vídeo informativo (Cypress.io, 2022) sobre Cypress, en el que se trata todo lo que se puede llegar a hacer. A continuación, en la Ilustración 5, se muestra un ejemplo del código de Cypress donde se puede observar la facilidad con la que se pueden leer los test y, el uso de PageObjects (loginPage).

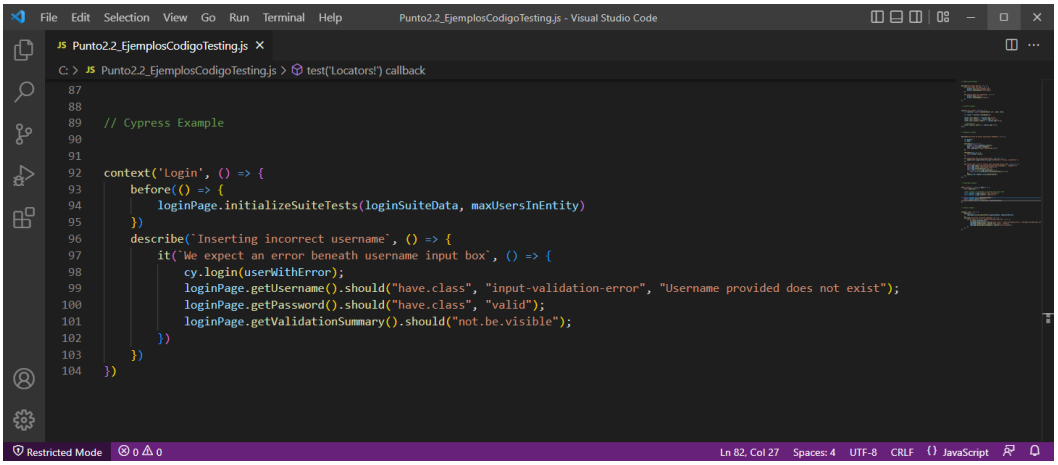



*Ilustración 5 - Ejemplo de código usando Cypress*

## 2.3 Crítica del estado del arte

La tecnología siempre se encuentra en constante desarrollo, es por ello por lo que toda aplicación debería de tener su propio *software* de pruebas que testee que su funcionalidad sigue siendo la que se esperaba.

Tras investigar otros trabajos y ver una falta de *testing* en ellos, se decidió no seguir por el mismo camino y, centrarse esta vez en las pruebas de interfaz de usuario, con el objetivo principal de lograr una *suite* de test escalable, que detecte aquellos fallos que realmente importan, centrándose siempre en la accesibilidad del *software*.

## 2.4 Propuesta

El trabajo se plantea mejorar la *suite* de test te la empresa Preference S.L. para su *software* PrefWeb. Este aportará una mejora importante tanto en tiempo como en coste y escalabilidad.

La intención del trabajo es replicar aquello que ya se está testeando mediante el antiguo *software* empleado, aunque intentando mejorar siempre el código en base a las buenas prácticas aprendidas previamente. Añadir test que deberían de lanzarse y eliminar aquellos que realmente no tienen relevancia.

# 3. Análisis del problema

Antes del desarrollo de este proyecto, las pruebas de interfaz de usuario en PrefWeb no se estaban realizando acorde a las buenas prácticas dadas en la guía “Testing Best Practices”. (Goldberg, 2022)

Esto se debía principalmente a la dificultad de llevar a cabo la mayoría de estos puntos en una aplicación altamente personalizable con una dependencia bastante alta con el contenido en la base de datos. Además de la decisión poco acertada de utilizar Ranorex para realizar unas pruebas que no van a aprovechar al máximo las funcionalidades del *framework*.

A continuación, se resumirán los principales problemas de la *suite* de pruebas anterior:

1. La duración de la *suite* de test. Su ejecución completa puede llegar a extenderse hasta las 6 h 30 min. Esto deriva en que solo se puede hacer una vez durante la jornada laboral o varias veces fuera de la jornada, arriesgándose a que, si ocurre algún problema, no se puede solucionar hasta el próximo día laboral.
   Tal y como se puede observar en la Ilustración 6, los tiempos de ejecución son excesivos si se quiere llegar a probar toda la *suite*. Incluso si solo se optase por testear una base de datos, podría llegar a extenderse a 1 h 42 min en el caso de la base de datos “Regicarp”.

**Failed User Interface Tests Development (Version 20.1.1.5290)**

| Name | Test Log | Status | Execution Time |
|---|---|---|---|
| PrefDataSourceSelector-Efco | \\altair\UiTestReports\UITest_PrefDataSourceSelector_Efco\2022-08-11\20220811_215654_UITest_PrefDataSourceSelector_Efco.html | Success | 00:01:17 |
| PrefOne-Efco | \\ALTAIR\UiTestReports\UITest_PrefOne_Efco\2022-08-11\20220811_215807_UITest_PrefOne_Efco.html | Success | 00:23:42 |
| PrefDataSourceSelector-Prevost | \\ALTAIR\UiTestReports\UITest_PrefDataSourceSelector_Prevost\2022-08-11\20220811_222149_UITest_PrefDataSourceSelector_Prevost.html | Success | 00:01:09 |
| PrefOne-Prevost | \\ALTAIR\UiTestReports\UITest_PrefOne_Prevost\2022-08-11\20220811_222258_UITest_PrefOne_Prevost.html | Success | 00:31:28 |

| Name | Browser | Test Log | Status | Execution Time |
|---|---|---|---|---|
| PrefWeb-Brisa | chrome | \\altair\UiTestReports\UITest_PrefWeb_Brisa\2022-08-11\20220811_225427_UITest_PrefWeb_Brisa.html | Failed | 00:14:15 |
| PrefWeb-ePrefExtrugasa | chrome | \\altair\UiTestReports\UITest_PrefWeb_ePrefExtrugasa\2022-08-11\20220811_230842_UITest_PrefWeb_ePrefExtrugasa.html | Success | 00:25:43 |
| PrefWeb-Global | chrome | \\ALTAIR\UiTestReports\UITest_PrefWeb_Global\2022-08-11\20220811_233426_UITest_PrefWeb_Global.html | Success | 00:08:09 |
| PrefWeb-Oknoplast | chrome | \\ALTAIR\UiTestReports\UITest_PrefWeb_Oknoplast\2022-08-11\20220811_234235_UITest_PrefWeb_Oknoplast.html | Success | 01:20:17 |
| PrefWeb-Regicarp | chrome | \\ALTAIR\UiTestReports\UITest_PrefWeb_Regicarp\2022-08-12\20220812_010253_UITest_PrefWeb_Regicarp.html | Success | 01:42:05 |
| PrefWeb-Veteco | chrome | \\altair\UiTestReports\UITest_PrefWeb_Veteco\2022-08-12\20220812_024458_UITest_PrefWeb_Veteco.html | Success | 00:15:19 |
| PrefWeb-Windoor | chrome | \\ALTAIR\UiTestReports\UITest_PrefWeb_Windoor\2022-08-12\20220812_030017_UITest_PrefWeb_Windoor.html | Success | 01:12:47 |

**Total Execution Time: 06:16:14**

Note:These tests can prove that the program is incorrect but cannot prove that the program is correct.

*Ilustración 6 - Tabla de tiempos de ejecución de la suite de test para cada base de datos.*

2. Dependencia de los test con el código de la aplicación. Teniendo en cuenta que son pruebas centradas en el usuario, éstas deberían de probar los flujos de trabajo de la misma forma que lo haría un usuario real, sin embargo, están fuertemente ligadas a la estructura del código y no a los flujos de trabajo. Este problema podría proceder de diferentes fuentes, aunque, tras el análisis inicial, se comprobó que una de las principales causas era el uso de Ids y clases como mecanismo para acceder a los elementos del DOM que se querían alterar. Éstos pueden ser perturbados fácilmente al modificar el código, como, por ejemplo, cuando algún desarrollador que desconocía que se usaba dicho atributo para las pruebas, lo eliminaba o le cambiaba el nombre (hecho bastante común por el tamaño considerable de la empresa).
3. La limitación de un *software* de pago. Ranorex es un *software* de pruebas de GUI que tiene ciertos inconvenientes, como por ejemplo el hecho de que las licencias sean

individuales y de pago (esto perjudica las ejecuciones en paralelo), la carencia de compatibilidad con macOS debido a que fue programado en .NET, y la falta de apoyo comunitario, notándose una comunidad mucho más pequeña que la competencia, añadiendo una dificultad extra a la hora de encontrar soluciones a problemas habituales.

4. Dificultad para resolver los fallos detectados por la *suite* de test. Actualmente Ranorex no tiene la posibilidad de ver el DOM en el momento exacto en el que fallan los test, con lo que en ciertas situaciones la resolución al problema suele venir acompañada de una complejidad que se puede evitar.
5. Antipatrones. Comúnmente nos encontramos en el código acciones que nos llevan a una mala solución para el problema planteado. En el caso del código de pruebas existente, se han detectado antipatrones que impiden la ejecución de distintos hilos de código en paralelo. Este antipatrón consiste en que, a la hora de implementar los test, se fuerza una dependencia entre estos que ocasiona que no puedan ejecutarse de forma aislada. Por ejemplo, si existen dos test para probar que se crea y se elimina correctamente un usuario, la prueba "Eliminar usuario" no debería de ser probada con los datos que se generan tras la ejecución del test "Crear usuario", sino que deberían de ejecutarse con datos diferentes. Este escenario se repite en reiteradas ocasiones en el código fuente de la antigua *suite* de test. Otro escenario bastante común es que, al ejecutar las pruebas con unos datos previamente insertados en la base de datos, no se está emulando el comportamiento real de un usuario, ya que el usuario siempre parte de cero.

Una vez se ha plasmado el problema de forma adecuada, se plantean los requisitos de la solución que se va a proponer. Dichos requisitos (puestos en común entre la empresa y el autor del proyecto) son los siguientes:

1. Reducir drásticamente los tiempos de ejecución totales de la *suite* de test ya existente, ya sea mediante *software* o hardware.
2. Preparar el código para que sea mantenible y no falle con facilidad debido a cambios en el código que no influyen al usuario final. Esto viene de la mano con el punto 4, de aplicar buenas prácticas.
3. Facilitar la forma en la que se solucionan los fallos. Esto se debe a que en algunos casos fallan los test, pero se exige un conocimiento previo de la *suite* de test para poder repararlos. El requisito pues consiste en obtener un *software* capaz de identificar el error con el mensaje adecuado y tener la posibilidad de *debuggear* el DOM en dicho momento.
4. Aplicar buenas prácticas para evitar conflictos en un futuro y garantizar la escalabilidad del proyecto. Las buenas prácticas se conseguirán en parte mediante el uso de la metodología de trabajo ágil Scrum. Esta es una metodología incremental que divide las tareas e itera sobre intervalos de tiempo cortos y fijos (llamados sprints) con una duración aproximada de una semana para nuestro proyecto. Las etapas de esta metodología son: planificación (del *sprint*), ejecución (*sprint*), reunión diaria y demostración de los resultados.
5. Conseguir igualar o disminuir el coste monetario de la anterior solución. Aunque no suponga un problema primordial para la empresa, se ha marcado como requisito debido a que existe un interés en que no se supere el presupuesto de la *suite* anterior.

Teniendo marcados ya los requisitos para la solución del proyecto, se estima que no se pueda completar a tiempo para la entrega del trabajo, sin embargo, será suficiente con alcanzar los objetivos, aunque no suponga la realización de todos los test que actualmente se están realizando. Todo esto siempre que se demuestre que se está yendo por el camino correcto y que es factible a largo plazo.

La forma en la que se abordará el problema será volviendo a implementar la *suite* de test de la empresa utilizando el *framework* de testing Cypress y la librería Testing-Library, entre otros plugins que facilitarán el uso de las API de PrefWeb (el *software* de la empresa) y el acceso a valores de la base de datos. Durante la validación de esta etapa, gracias a la experiencia adquirida, se extraerá una metodología que facilite el cumplimiento de los requisitos a futuros desarrolladores.

El código se implementará mediante el patrón de diseño POM (Page Object Model), que ayudará a reducir la duplicación de código y mejorará el mantenimiento y la escalabilidad de los casos de pruebas. Además de esto, ayuda drásticamente en la legibilidad del código, ya que sabes en cada momento a qué página pertenece el elemento al que se está accediendo.

## 3.1 Análisis de riesgos

En este punto se tratarán los riesgos más relevantes una vez el proyecto se use a nivel de producción. Dichos riesgos son los siguientes:

- Riesgo 1.
  - Tipo: inconsistencia
  - Consecuencias: si el resultado de los test es inconsistente y no funcionan como deberían, conducirán a una mala experiencia de usuario, además de a una impotencia por parte de los programadores que, aun sabiendas que su código funciona correctamente, no lo pueden subir a producción porque no pasa debidamente los test.
  - Solución: mantener en paralelo la *suite* de test antigua junto con la nueva durante un período de tiempo, además de cerciorarse debidamente del funcionamiento correcto de los test.
- Riesgo 2.
  - Tipo: complejidad para añadir nuevos test
  - Consecuencias: si la escalabilidad no se plantea de forma correcta, la *suite* de test se puede estancar y no conseguir testear nada más que lo que ya se estaba testeando. También puede ocasionar una dificultad innecesaria a la hora de refactorizar o añadir test.
  - Solución: mantener una estructuración clara del proyecto (en este caso mediante Page Object Models) para poder reutilizar en medida de lo posible todo el código e implementar dicho código de forma *verbose*
- Riesgo 3.
  - Tipo: prolongación excesiva del proyecto
  - Consecuencias: si no se marcan unos límites, debido a la complejidad y el tiempo requerido para la implementación de los test E2E, el proyecto podría llegar a abarcar años.

    - Solución: centrarse en los procesos más comunes para los usuarios y testear mediante test unitarios aquello que no frecuente tanto el cliente.
- Riesgo 4.
    - Tipo: Económico
    - Consecuencias: al igual que en el punto anterior, si no se marcan unos límites, debido a la complejidad y el tiempo requerido para los test E2E, el proyecto podría llegar a abarcar años, llevando de la mano al coste de implementación, incluso del *software* utilizado si este es de pago y requiere de licencias.
    - Solución: similar a la solución propuesta para el *riesgo 3*, centrarse en los procesos más comunes para los usuarios y testear mediante test unitarios aquello que no frecuente tanto el cliente, teniendo en cuenta en todo momento el coste total del proyecto hasta la fecha.

## 3.2 Identificación y análisis de soluciones posibles

De forma breve, se procederá a identificar y analizar las posibles soluciones para el problema planteado en el punto 3 del proyecto. Si se desea optar a la mejor solución, primero se deben tener claros los requisitos y las preferencias, tanto de la empresa como del autor del proyecto:

- Reducción de tiempos de ejecución: especial interés en un *software* rápido que permita ejecutarse tanto en modo *headed* como *headless*. También conviene que exista la posibilidad de ejecutar test en paralelo de una forma amigable para el usuario
- Facilitar la resolución de los errores detectados por la *suite*: este trabajo se puede facilitar o complicar dependiendo del *software* que se vaya a utilizar. Una solución planteada es que el *software* que se adquiera para desarrollar el proyecto admita el modo *debugger* para poder ver el estado en el que se encuentra nuestra aplicación en el momento del error. También es de interés poder ver las llamadas realizadas a la API, al igual que el contenido en la base de datos.
- Igualar o disminuir el coste monetario actual: dado que el *software* anterior es de pago y conlleva el uso de licencias para su utilización, la empresa ha decidido que quiere optar por un *software* libre con unas prestaciones que se adecuen en mayor medida a la aplicación de esta.
- Conseguir legibilidad en el código: el *software* a elegir debería de ser *verbose* y tener la posibilidad de aplicar el patrón de diseño (POM), que es familiar a la empresa y se considera una buena práctica para mantener una correcta estructura en el proyecto, además de que consigue evitar en gran medida código repetido.
- Documentación: que el *framework* elegido esté altamente documentado, ya que en ocasiones se han experimentado problemas con el anterior *software* (Ranorex) al intentar implementar código poco común.

Es por todos los puntos anteriores por los que se procede a analizar las posibles soluciones al problema.

La primera solución posible fue plantearse reutilizar todo el código y seguir con la herramienta Ranorex, sacrificando la utilidad de ésta por el tiempo ahorrado al no tener que cambiar de tecnologías. Esta posible solución no llegó muy lejos, ya que el proyecto consistiría en un *refactor* imposible de todo el código para llegar a aplicar buenas prácticas, sin tener apenas resultados en la bajada de tiempos, ya que uno de los principales problemas era el embudo que

ocasionaba el *software*. Descartada esta primera opción, solo quedaba plantearse qué herramienta actual sustituiría Ranorex.

En primer lugar, se investigaron las mejores alternativas y se sintetizaron sus ventajas y desventajas. En este caso los *frameworks* de interés que se encontraron con características similares a los requisitos fueron los siguientes: WebdriverIO, TestCafe, Puppeteer, Cypress y Playwright. Mientras que también hubo librerías que destacaron y podrían ser de gran utilidad, como lo son Enzyme y Testing-Library.

- WebdriverIO (Hegde, browserstack, 2022)
    - Ventajas:
        - *Page Object Pattern*: se puede configurar fácilmente para usar este patrón de diseño de gran interés.
        - Capturas de pantalla
        - Soporta *iframes*
        - Soporta paralelización
        - Soporta varias pestañas durante la ejecución de los test
    - Desventajas:
        - Sintaxis confusa
        - Documentación obtusa y confusa en algunas ocasiones.
        - Las limitaciones de Selenium (dificultad de mantenimiento y escalabilidad, inexistencia de pruebas en imágenes, incapacidad para generar informes, etc.) se aplican a este *framework*, ya que es una customización de este.
        - No tiene una ventana interactiva donde ver el resultado de los test y poder hacer pruebas
        - Bibliotecas de aserciones: Jasmine, Mocha y Cucumber
- TestCafe (Hegde, browserstack, 2022)
    - Ventajas:
        - Rápido y estable: debido a que se ejecuta dentro de un navegador, las pruebas son más rápidas
        - Soporta paralelización
        - *Headless*
        - Espera automatizada: espera a que aparezcan los elementos y no es necesario insertar esperas externas
        - Depuración: TestCafe proporciona el *Live mode*, que ayuda a visualizar acciones individuales en el navegador para facilitar su depuración
        - Capturas de pantalla
    - Desventajas:
        - Difícil depuración
        - Bibliotecas de Aserciones: solo aquellas que incorpora por defecto. Esto hace que sea menos *verbose* que el resto de *frameworks*
- Puppeteer (Vaidya, 2022)
    - Ventajas:
        - Rápido
        - Capturas de pantalla
        - *Headless*

        - Desarrollado por Google, con lo que ayuda a establecer una buena confianza con el *software*
    - Desventajas:
        - No tiene un IDE propio: se suele usar VS Code
        - Bibliotecas de Aserciones: Mocha, Jasmine y Jest
        - No tiene una estructura de *fixtures*: Carpetas donde se suelen añadir datos para los test, generalmente en archivos ". json"
        - No tiene *dashboard*
- Cypress (Monroy, 2022)
    - Ventajas:
        - *Page Object Pattern*: se puede configurar fácilmente para usar este patrón de diseño
        - "Viaje en el tiempo".
        - Espera automática: no hay que realizar esperas explicitas, sino que Cypress espera automáticamente a que acaben todas las aserciones antes de continuar.
        - Control de tráfico de red sin involucrar el servidor.
        - Soporta paralelización
        - *Headless* o *Headed*
        - IDE propio
        - *Dashboard* propio (el más completo entre todos los demás *frameworks*)
        - Rápido y estable: debido a que se ejecuta dentro de un navegador, las pruebas son más rápidas
    - Desventajas:
        - Compatibilidad con *iframes* limitada
        - Bibliotecas de Aserciones: Mocha y Chai
        - Pestañas múltiples: Cypress no admite pestañas múltiples ni permite cambiar entre ventanas Principal y Secundaria
        - *Dashboard*: aunque Cypress tiene su propio *Dashboard*, este es de pago si se quieren todas las características, aunque suele bastar con la versión gratuita
- Playwright (Tej, 2022)
    - Ventajas:
        - Permite crear escenarios abarcando varias páginas y dominios
        - Control de tráfico de red sin involucrar el servidor
        - Soporta muchos lenguajes: JavaScript, Java, Python y .Net C#
        - *Headless*
        - No soporta dispositivos reales para las pruebas en navegadores de móviles, pero soporta emuladores.
    - Desventajas:
        - Bibliotecas de Aserciones: Mocha, Jest, Jasmine
        - No tiene una comunidad grande de usuarios, con lo que cuesta encontrar información a causa de ciertos errores

Tras una síntesis de toda la información encontrada sobre los *frameworks* citados anteriormente, se procede a descartar aquellas opciones que no se considerarían válidas por no cumplir con los requisitos iniciales.

En el caso de WebdriverIO, el patrón de diseño POM es fácilmente configurable, soporta iframes (que sabemos que no se van a usar), soporta paralelización y soporta varias pestañas (que no se van a usar tampoco debido a cómo se estructura PrefWeb). Es por eso por lo que, ante las pocas ventajas coincidentes con los requisitos, se descartaría esta opción temporalmente.

En el caso de TestCafe, comparándolo con Puppeteer, tiene muchas más ventajas que nos interesan, como por ejemplo el tipo de depuración o la espera automatizada, además de ser algo más estable y de tener una comunidad significantemente más grande. Puppeteer se descartaría por ello, y TestCafe sería un candidato para la nueva *suite* de tests.

Entre Cypress y Playwright hay un poco de igualdad, aunque debido a la falta de comunidad en Playwright por su reciente implementación y aún en desarrollo, la elección entre estos es Cypress.

Quedando solo Cypress y TestCafe, la decisión por saber cuál de los dos aportaría mayores ventajas ya no depende de los requisitos, sino del enfoque que aporta cada uno y de qué requisitos queremos potenciar más. Una funcionalidad que marca la diferencia es la de "Viaje en el tiempo" de Cypress, que permite poder visitar el DOM durante cualquier comando ejecutado para ver el antes y el después, el resto del DOM, las llamadas *POST*, *GET*, etc. Además de esto, Cypress tiene un Dashboard un tanto superior a los demás *frameworks*, con lo que, a causa de ciertos detalles, la decisión final ha sido la de escoger Cypress como herramienta de testeo para la nueva *suite* de tests.

Ahora ya tenemos Cypress como la herramienta principal de testeo, pero durante la búsqueda del mejor *framework*, había una librería que destacaba y además se recomendaba su uso en combinación con Cypress, y ésta es Testing-Library. Tras buscar alternativas a Testing-Library, apareció Enzyme, una librería bastante utilizada y que le podría hacer la competencia a la anterior. Pero, lo cierto es que después de una lectura de un artículo escrito por Lucas Bernalte llamado *Por qué usar Testing Library en lugar de Enzyme* (Bernalte, 2022), no quedaban muchas opciones disponibles para Enzyme.

Las principales limitaciones de Enzyme son la necesidad de probar componentes mediante un test que requería de mantenimiento y que los test están enlazados con la implementación. Esto infiere en que el código se vuelve mucho más difícil de mantener.

De forma resumida, Testing Library promueve las buenas prácticas del testing, mediante la aplicación de la política *Behavior Driven* y, además, sirve de gran ayuda en especial para testear interfaces de usuario de forma centrada en el usuario.

Testing Library tiene como premisa evitar testear implementaciones. Esto implica que puede, entre otros:

- Permitir comprobar la existencia de elementos mediante queries directamente.
- Lanzar eventos igual que se lanzan eventos desde el DOM como por ejemplo con:

```
fireEvent.click(screen.getByRole('button'))
```

- Comprobar que existe un elemento con las queries recomendadas, para seguir las buenas prácticas, y que las encontramos como respuesta del método render o dentro de un objeto *screen*, que contiene todo lo que renderizamos en el test. Esto puede hacerse

de forma síncrona o asíncrona. El método asíncrono, en realidad funciona ejecutándose varias veces teniendo un *timeout* configurable. Por ejemplo, usando:

```
await screen.findByText('title')
```

Teniendo en cuenta toda la información anterior, se deduce que la solución adecuada al problema será el uso del *framework* Cypress junto con la librería Testing Library, que proporcionará una gran forma de conseguir buenas prácticas.

## 3.3 Plan de trabajo

Un plan de trabajo es un instrumento de planificación que ordena y sistematiza información de modo que pueda tenerse una visión global del trabajo a realizar, indicando: objetivos, metas, actividades, responsables y cronograma.

Para el proyecto actual, se usará el marco Scrum, a través de la herramienta Atlassian[6]. El marco de trabajo de Scrum es heurístico. Se basa en el aprendizaje continuo y en la adaptación a los factores fluctuantes. Reconoce que el equipo no lo sabe todo al inicio de un proyecto y evolucionará a través de la experiencia.

Para entender un poco más en profundidad cómo funciona Scrum, se empezará identificando sus tres "artefactos" o herramientas para solucionar problemas.

1. *Backlog* del producto: es la lista principal del trabajo que realiza el propietario del producto. Se trata de una lista dinámica de requisitos, mejoras y funciones que actual como entrada para el *backlog* de *sprint*. Para este caso concreto del proyecto, el propio autor del trabajo será quien manipule este *backlog* del producto, ya que será donde se añadirán las tareas en las que se ha subdividido todo el trabajo.
2. *Backlog* de *sprint*: se trata de la lista de elementos seleccionados por el equipo de desarrollo, para su implementación en el ciclo actual de *sprint*. Antes de cada *sprint*, mediante una reunión en la que intervienen todos los miembros, el equipo (en este caso el autor del proyecto) elige los elementos en los que trabajará durante el *sprint* a partir de las tareas del *backlog* inicial. El *backlog* de *sprint* suele ser flexible y puede evolucionar durante su *sprint*.
3. Incremento: es el resultado de un *sprint*. En esta fase, se muestra aquello que se ha completado durante el *sprint*.

Estos tres artefactos presentados anteriormente pueden variar dependiendo de los requisitos del equipo y sus condiciones. Por eso es importante estar abierto a la evolución en forma de mantenimiento, incluso de los artefactos.

Algunos de los componentes más conocidos de este marco de trabajo son el conjunto de eventos que se realizan de forma periódica. Dependiendo de los criterios del equipo, se consideran algunos protocolos u otros. A continuación, se muestran una lista de todos aquellos eventos Scrum en los que se participará durante el desarrollo del proyecto:

[6] Página web de Atlassian: https://www.atlassian.com/

1. Organización del *backlog*: en este evento, se anotarán todas las tareas que posteriormente se mostrarán en forma de lista, con un orden de prioridad, siendo la tarea más importante aquella que aparezca en la parte superior, y la menos importante o la más prescindible, la que aparezca en la parte inferior, tal y como se muestra en la Ilustración 7.

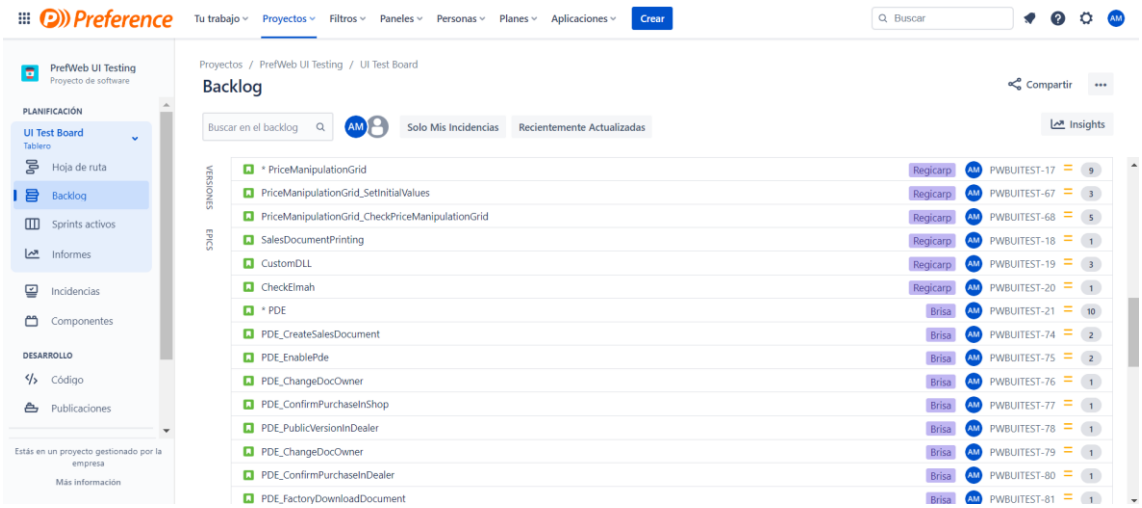



*Ilustración 7 - Backlog del producto usado en el proyecto*

2. Planificación del *sprint*: durante esta reunión, se planifica el trabajo a realizar (alcance) durante el *sprint* actual. Posteriormente, se añaden historias de usuario específicas desde el backlog el producto. Al final de la reunión, se deberá de tener claro qué se puede entregar en el *sprint* y cómo se puede entregar el incremento.
3. *Sprint*: es el período en el que el equipo trabaja de forma conjunta para realizar un incremento. Para el caso del proyecto, se establece una duración de una semana, aunque suele ser de dos. Esto se debe a que, cuanto más complejo sea el trabajo, más corto debería de ser el *sprint.*
4. Scrum diario o reunión rápida: se trata de una reunión diaria de muy corta duración que tiene lugar siempre a la misma hora (para este caso concreto a las 07:00 A.M.) y en el mismo sitio. Generalmente la duración suele ser de unos 15 o 20 minutos, dependiendo de la cantidad de progreso realizado y de las dudas planteadas el día anterior que no tuvieron solución.
5. Retrospectiva de *sprint*: la retrospectiva es donde el equipo se reúne para documentar y analizar qué ha funcionado y que no en un *sprint*.

Tras describir cómo funciona Scrum y marcar qué eventos se utilizarán en el trabajo, se procederá a estimar el esfuerzo que supondrá el proyecto, indicando las fases en las que se dividirá éste y las horas aproximadas que se necesitarán para desarrollar cada fase. Esto permitirá calcular un presupuesto aproximado en euros y horas de trabajo.

Fases de estimación del proyecto:

1. Creación de *epics* e historias: estas dos herramientas sirven para estructurar las tareas dependiendo de su finalidad. Los *epics* son grandes cuerpos de trabajo que se pueden dividir en tareas pequeñas (llamadas historias), y las historias son solicitudes escritas

desde la perspectiva del usuario final. Conocida la diferencia, se clasificarían las tareas para más adelante añadirlas en el *backlog* del producto. A modo de ejemplo, y mediante un vistazo rápido, los *epics* serían las distintas bases de datos que se testearán, mientras que las historias serán los tests de cada base de datos.

2. Creación del *backlog* del producto: para la creación de la lista principal de tareas, se partirá de todas las pruebas que se encuentran en la antigua *suite* de tests de Ranorex. Con esto se consigue una evaluación aproximada del alcance del proyecto. Tal y como se puede observar en la Ilustración 8, se han creado tareas (ya divididas en *epics* o historias) en el backlog a partir de las pruebas existentes. En aquellos casos en los que las pruebas superaban los diez minutos, se han subdividido en otras tareas, para así conseguir unos sprints más precisos.
   A estas tareas, se les asignarán un número determinado de *story points* dependiendo de la estimación que se haga. Cada *story point* equivaldrá a ciertas horas de trabajo que no nos interesan, pero que nos servirán para añadir tareas en función de estos en los sprints. Por ejemplo, en el proyecto se ha llegado a la conclusión de que cinco *story points* son los que se pueden completar cómodamente en un *sprint*. Este valor irá cambiando una vez se vayan realizando tareas, ya que se podrá saber cada *story point* cuantas horas lleva mucho mejor cuando se avance en el proyecto que nada más empezar.

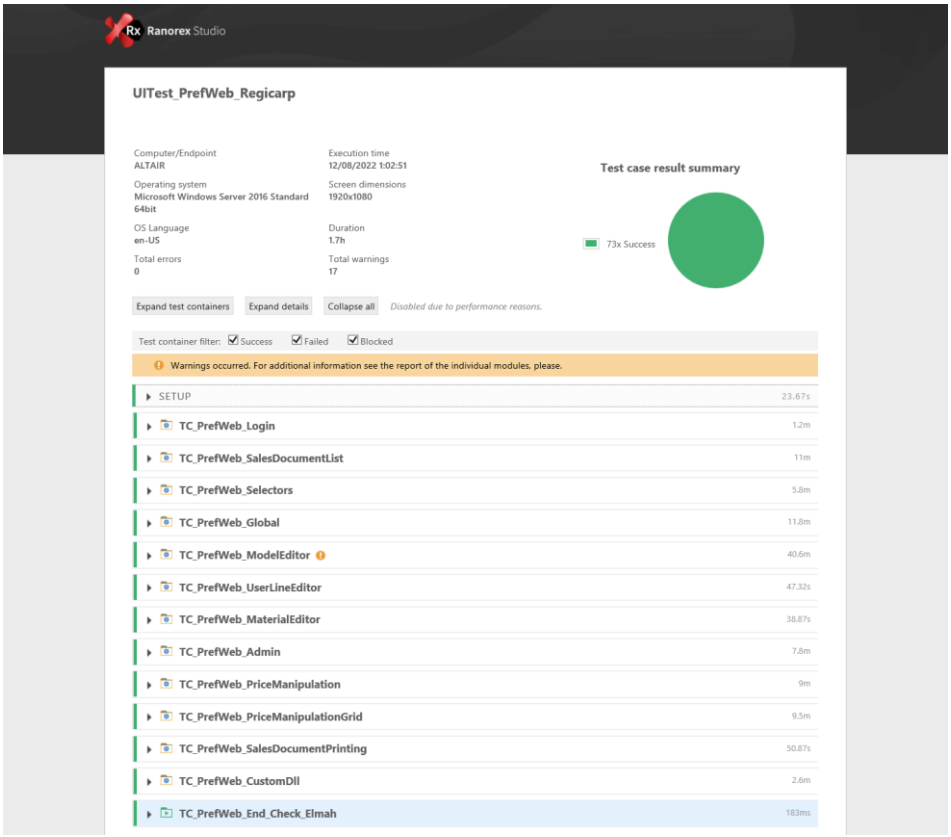



*Ilustración 8 - Casos de pruebas en Ranorex con sus respectivos tiempos de ejecución*

3. Calcular la duración estimada del proyecto: una vez se ha creado el *backlog* y se hayan asignado los *story points* a las tareas, se podrá aproximar la duración de este. Esta fase se puede hacer al inicio del trabajo, pero no suele ser realista. Es durante el segundo cuarto del proyecto cuando se pueden estimar de forma más acertada los tiempos.

Dicho esto, para el primer *epic* que se va a realizar, la suma de los *story points* de todas las historias es de 99. Si tenemos en cuenta que 5 *story points* equivalen a un *sprint* o semana de trabajo, llegamos a la conclusión de que nos llevará 19.8 semanas, de cinco días cada una y ocho horas dedicadas para cada día. Esto es solo para un *epic* de los seis existentes en el proyecto. Calculando los *story points* de todos los *epics*, el proyecto se estimaría en 288 laborales, o lo que es lo mismo, 14 meses naturales.

## 3.4 Solución propuesta

Como se ha dejado claro en el Punto 3.2, la solución propuesta para el proyecto será el desarrollo de una nueva *suite* de tests desde cero. Este desarrollo se realizará mediante la herramienta Cypress y la librería Testing Library.

El desarrollo de la nueva *suite* de tests, se llevará a cabo considerando las siguientes fases:

### 3.4.1 Documentación

Esta primera fase de la solución tiene como objetivo familiarizarse con los lenguajes de programación que se van a utilizar, al igual que con el entorno de desarrollo. Además, al tratarse de un proyecto sobre una rama de la informática en la que no se incide en exceso, también se requiere de una exhausta búsqueda sobre las mejores prácticas en base a la creación de tests.

Para ello, se pretende recopilar información sobre lo siguiente:

- Cypress: aunque es una herramienta basada en JavaScript, tiene sus propias peculiaridades y sus comandos únicos, que pueden complicarse si no se tiene cierta experiencia previa. Al ser la base de la solución del proyecto, se debería invertir un tiempo proporcional en el aprendizaje de esta herramienta. La ventaja es que Cypress dispone de suficiente documentación (tiene una página dedicada a videos tutoriales (Cypress.io, 2022)) como para tener una buena base.
- Testing-Library: esta librería aporta una forma distinta a la que propone Cypress cuando se trata de crear test. De hecho, la elección de esta librería en parte fue por el gran interés en seguir su filosofía en el testing. Por ello, aunque aporta unos pocos comandos a Cypress, es más importante entender y seguir su filosofía de testing centrado en el usuario (a un nivel diferente que Cypress).
- Visual Studio: este IDE de Microsoft es el utilizado por la empresa para el desarrollo y mantenimiento de su aplicación. Es por ello por lo que también es de interés conocer y entender cómo funciona el entorno de desarrollo que se va a usar.
- Visual Studio Code: es un editor de código de gran utilidad por la cantidad de extensiones y ayudas que aporta, agilizando el trabajo.
- Postman: es una aplicación que permite realizar pruebas API. Esto es de gran utilidad cuando se quieren probar llamadas API antes de ser implementadas, garantizando así que primero funcionen con Postman y que, posteriormente, funcionen de la misma forma tras ser implementadas.
- Lenguajes de programación, como lo son: JavaScript y C#. Ambos lenguajes son bastante populares y conocidos por gran parte de los programadores, pero no hay que descartarlos, ya que las pruebas anteriores de la empresa y el código de la empresa están escritas en C#, al igual que JavaScript también está presente en el código de PrefWeb y en la nueva *suite* que se pretende desarrollar (Cypress está basado en JavaScript).

### 3.4.2 Fijación de tareas

Cuando las tareas quedan definidas de forma clara y estratégica, se facilita la toma de decisiones. Se sabe dónde se quiere llegar, y elegir las acciones para alcanzar esa meta resulta más sencillo.

Los objetivos principales del proyecto ya se han establecido con anterioridad en el punto 1.2 de este trabajo, pero esta vez, se determinarán aquellas tareas más específicas, que permitirán una medida de tiempo para así aproximar cuando se podrían alcanzar los objetivos principales, que son el motivo primordial del desarrollo del trabajo.

Al tratarse de desarrollar una *suite* de pruebas desde cero, es necesario establecer por dónde se ha de empezar, cuál es el ritmo de progreso que se espera medido en días, etc. Para lograr esto, nos podemos centrar en las pruebas de la antigua *suite*, ya que, pese a que no se estaban usando buenas prácticas, la aplicación se estaba testeando con los flujos correctos. Es por ello por lo que se pretende iniciar un análisis de las pruebas anteriores para determinar la escala aproximada del proyecto.

Terminado lo anterior, se procedería a subdividir todos los test en tareas y subtareas para poder determinar de forma aproximada la proporción del trabajo.

Con la ayuda de Scrum, un marco que permite el trabajo colaborativo entre equipos y la distribución de tareas organizadas en *sprints*, se quedarían apuntadas todas las tareas y subtareas para su posterior asignación.

Desde este punto, es algo complicado saber con exactitud cuánto puede llevar cada tarea, con lo que la herramienta Scrum nos proporciona los *story points*. Estos *story points* se definen como una unidad de medida utilizada principalmente en la gestión de proyectos ágiles, que sirven para estimar la carga de trabajo global de los equipos, con el fin de planificar cada *sprint* o iteración.

Hecho todo lo anterior, ya tendríamos todo el proyecto organizado en tareas por hacer y, solo habría que asignar dichas tareas a *sprints* semanales e ir ajustando los *story points* hasta el punto de conocer su valor aproximado en horas y poder estimar la duración del proyecto.

### 3.4.3 Implementación

En este punto, se pretenden desarrollar todos los test que incluirá la nueva *suite* de pruebas partiendo desde cero, siguiendo una metodología ágil (realizando el trabajo en incrementos pequeños pero consumibles) con el marco de trabajo Scrum.

Antes de proceder a la implementación, es necesario marcar una serie de reuniones con el líder del equipo para poder determinar la ubicación exacta del proyecto (dentro de la solución de la aplicación PrefWeb, o como un proyecto completamente independiente), la rama de GIT que se pretende usar, las limitaciones, etc.

Una vez han sido completadas estas reuniones, se procedería al desarrollo de la *suite*, sin hacer mucho hincapié en la refactorización, ya que, a medida que se va escribiendo código, aparecen nuevas dudas y soluciones que cambian el punto de vista y permiten mejorar soluciones anteriores plasmadas en el código, y esto, puede ser un inconveniente. Por ejemplo, cuando tras

refactorizar una parte del código, se sigue con el desarrollo y aparece otra solución mejor al problema anterior (que ya había sido refactorizado otra vez). Esto solo lleva a hacer un *refactor* detrás de otro, retardando la entrega final.

La solución a este pequeño problema que puede traer bastantes dolores de cabeza es dejar la refactorización para el final del proyecto y aplicar las soluciones correctas a medida que se va creando código nuevo. La ventaja de esto es que, aunque tengamos que hacer un *refactor* bastante grande, solo se trata de uno, y directamente con las mejores soluciones que se conocen hasta el momento.

En el proceso de la implementación, se desarrollará el código fuente según las tareas divididas en el punto anterior. Éstas se agruparán en *sprints* semanales estimando el número de tareas que se podrían completar mediante el uso de los *story points*.

A medida que las tareas se vayan completando y los *sprints* avancen, se irá ajustando el valor de los *story points* para que sean más fieles a la realidad, hasta el punto en el que se asignen las tareas de tal forma que, tras finalizar el *sprint*, no sobren tareas ni sea necesario añadir más.

Tras la estimación ajustada del valor de los *story points* con respecto al tiempo, ya se puede evaluar cuánto podría extenderse el proyecto. Esto nos interesará para marcar una *deadline* fija, de gran interés para la empresa.

Una vez terminada la sección de desarrollo, el siguiente punto sería el de refactorización. Este punto es importante, ya que todas las implementaciones que en algún momento se han mejorado, pero en ciertas partes del código han permanecido igual, se cambiaran para que sigan la misma funcionalidad en todo el proyecto y mejorará la escalabilidad y la solución de errores de este.

### 3.4.4 Refactorización

Durante el proceso de desarrollo, en el código fuente se van acumulando elementos defectuosos, llamados *code smells*. Estos puntos débiles, que van aumentando a medida que avanza el proceso, ponen en peligro la funcionalidad y la compatibilidad del *software* en cuestión. Para evitar esta erosión continua del *software*, se utiliza la refactorización o *refactoring*.

En el proyecto, la refactorización se llevará a cabo en dos procesos.

- El primero, tras cada prueba. Esto se debe a que cuando se desarrollan las pruebas por primera vez, no se suelen extraer métodos de primera mano ni se suelen encapsular, ya que el objetivo principal es que funcione. Es por esto por lo que, una vez funciona el test, se deben encapsular y extraer los métodos, al igual que si es necesario, también se deben de rehacer, para así organizarlo todo en *Page Object Models* y que posteriormente se puedan reutilizar estas funciones si fuese necesario.
- El segundo, al final del epic o del proyecto, dependiendo de la duración de este. Este refactor es el más importante, ya que iguala en todo el *software* las diferentes formas de hacer algo que se repite a través de la solución. De esta forma, se garantiza que, si más adelante se quiere ampliar el código o, si cambiase el código de la aplicación que se está testeando, este se puede alterar y mejorar de una forma bastante simple y mucho menos costosa.

Tras este *refactor* final, ya debería de estar lista la solución.

### 3.4.5 Comparación entre soluciones

En esta fase se comparará la solución proporcionada por el proyecto contra la solución anterior a este. A efectos de la empresa, esta es la parte más interesante de todo el trabajo, ya que, si no se consigue solucionar aquello que ya estaba vigente, no sirve de mucho más que de quebraderos de cabeza.

En la comparativa, se deben de tener en cuenta 4 factores:

1. Tiempo de ejecución: uno de los objetivos principales de la empresa en esta nueva solución, era que se mejorasen los tiempos de ejecución de las pruebas drásticamente, con lo que, a la hora de comparar soluciones, los tiempos deberían de estar presentes.
2. Gasto: otro de los objetivos, era que la solución no fuese más costosa que la que ya había, por lo tanto, este es otro punto que no puede faltar en la comparación.
3. Buenas prácticas: el antiguo *software* se caracterizaba por tener *anti-patterns* que impedían la escalabilidad, por lo que es de interés que se comparen las buenas prácticas para tener una mejor experiencia general en los test. Esto también permitirá que se detecten más fallos mediante menos código.
4. Facilidad de implementación: este punto principalmente se da ya que el antiguo *software* no estaba siendo implementado por un equipo, sino por un único miembro. Los principales problemas que aporta lo anterior es la velocidad a la que se desarrollan pruebas a medida que se expande el *software* y, un embudo que aparece cuando se dan muchos fallos y tan solo puede solucionarlos o detectar los errores aquella persona que los ha desarrollado. El hecho de que se tuviese tan solo una licencia para la anterior *suite* también afectaba y hacía que no se pudiese solucionar el problema con facilidad.

## 3.5 Presupuesto

El presupuesto de un proyecto es un plan en el que se detallan el conjunto de costes e ingresos previstos para un determinado período de tiempo. Establecer el presupuesto antes de comenzar con el proyecto ayuda a definir el alcance del trabajo y controlar los gastos. Además, también es una buena forma de presentar el trabajo a las partes interesadas y obtener los fondos necesarios.

Para el presupuesto de este trabajo se tendrán en cuenta los siguientes recursos:

1. Miembros del equipo: Álex Moreno Palacio. Empleado de la empresa (seis meses de contrato en prácticas y el resto con contrato indefinido)
2. Adquisiciones: se necesitará un tiempo para la investigación de las posibles soluciones, productos a usar, etc.
3. Capacitación: período de adaptación para aprender las habilidades nuevas que requiere el proyecto
4. Equipamiento: ordenador, acceso a cursos de aprendizaje, *software* dedicado a la implementación, etc.
5. Espacio: se requerirá un escritorio para trabajar y una sala de reuniones.

En cuanto a la duración del proyecto, ésta depende principalmente de la cantidad de miembros que participen en él. Calculando los *story point*s de todos los *epics* (visto en el punto 3.3), el proyecto se estimaría en 288 laborales, o lo que es lo mismo, 14 meses naturales (2300 h), que no varía, ya que tan solo será desarrollada por un integrante del equipo.

Partiendo de los recursos nombrados anteriormente, que sean cuantificables en horas, se podría deducir lo siguiente:

| | **Tiempo** |
|---|---|
| **Adquisiciones** | 16 h |
| **Capacitación** | 40 h |
| **Implementación** | 2300 h |
| **Total** | **2356 h** |

*Tabla 1 - Horas estimadas en relación con los recursos*

Si estas horas se las asignamos al único miembro disponible, se deduce lo siguiente:

- Primeros 6 meses (período de prácticas): suponiendo que el empleado percibe 4,3€/h y, suponiendo que un mes está compuesto por 23 días laborables, compuestos por ocho horas diarias, las horas dedicadas al proyecto en este período equivalen a las 1104h. Si se multiplican estas horas por 4,3€/h, obtenemos un total de 4.747,2€
- Resto del proyecto: suponiendo que el empleado percibe 9€/h, las horas dedicadas al proyecto en este período equivalen a las 1252. Si se multiplican estas horas por 9€/h, obtenemos un total de 11.268€.

En total, en mano de obra se deducen 16.015,2€. A este coste no es necesario sumarle ningún otro, ya que la empresa ya contaba con el resto de los recursos anteriormente nombrados.

En conclusión, el presupuesto final será de 16.015,2€, con una duración de 288 días o, lo que es lo mismo, 1 año y 2 meses aproximadamente, debiendo haber terminado la primera entrega (primer *epic*) a los 4 meses después del inicio del proyecto.

# 4. Diseño de la solución

## 4.1 Arquitectura del sistema

En este punto se tratará la arquitectura del sistema, profundizando en el patrón de diseño POM (*Page Object Model*) y las carpetas en las que se divide la solución, que son aquellas que ya proporciona Cypress para tener una mejor estructuración del código y así conseguir unas mejores prácticas. Estos puntos se basan tanto en la documentación de Cypress (Cypress.io,

2022), como en documentación aportada por este artículo sobre POM en Cypress (lambdageeks, 2022)

### 4.1.1 Estructura POM

POM, o *Page Object Model, es un* patrón de diseño donde cada una de las páginas de una web se representan como clases, y los diversos elementos de la página se fijan como variables en la clase. A continuación, se mostrará en la Ilustración 9 cómo se utilizan los *Page Objects* en el proyecto actual:

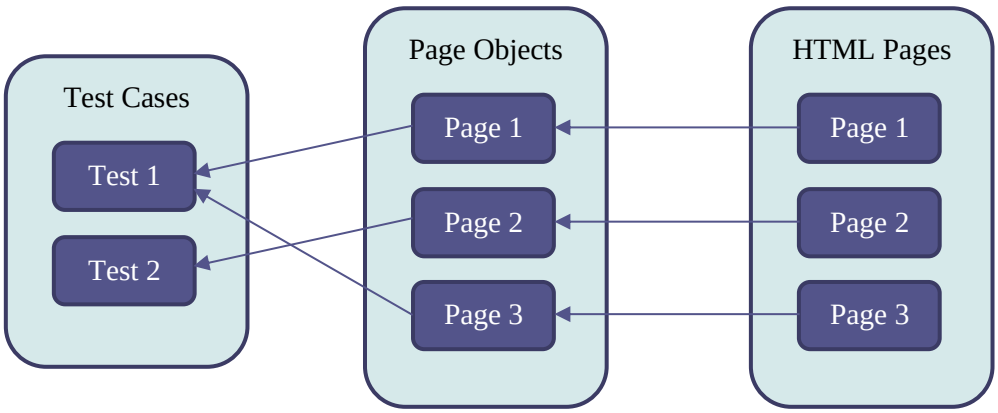



*Ilustración 9 - Diagrama UML sobre cómo funciona POM*

Ventajas:

1. De acuerdo con POM, debemos mantener las pruebas y los localizadores de elementos por separado. Esto mantendrá el código fácil de entender y de mantener, al igual que limpio.
2. Los casos de prueba se vuelven más cortos y optimizados al tener la posibilidad de reutilizar métodos de aquellos objetos de páginas en las clases POM
3. Cualquier cambio en el código fuente de la aplicación se puede implementar, actualizar y mantener en solo un lugar.
4. Evita repetir código, ya que lo estructura de tal forma que es intuitivo y fácil reutilizarlo.

Como implementar POM:

POM se tiene a aplicar cuando la aplicación a testear contiene diversas páginas, cada una de las cuales con campos a los que se puede hacer una referencia única con respecto a la página.

La forma en la que se ha implementado en el proyecto ha sido creando una carpeta llamada “PageObjects”, la cual contiene otras subcarpetas nombradas según la página a la que hacen referencia, por ejemplo, si hubiese una página “Login”, la estructura sería la siguiente:

Cypress > PageObjects > Login > LoginPage.js

En la Ilustración 10, se muestra como se ha llevado a cabo este patrón en el sistema del proyecto.

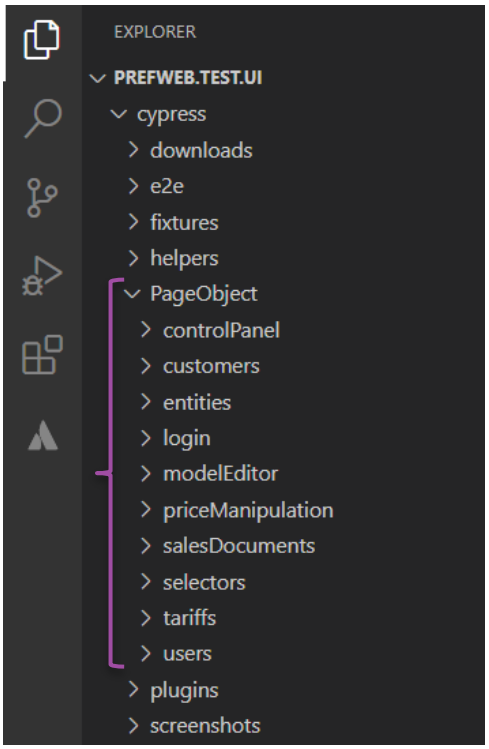



*Ilustración 10 – Estructuración de carpetas usando el patrón Page Object Model en el proyecto*

Tal y como se ha podido observar, aquellas páginas más relevantes que se usan en las pruebas se han creado dentro de la carpeta PageObjects, aunque es cierto que también se han creado otras carpetas que sirven como ayuda, pero no tienen ninguna página concreta, como por ejemplo la carpeta KendoHelper, cuya función es reutilizar los localizadores de esta librería, como tablas, filas, botones, etc.

### 4.1.2 Estructura Cypress

Tras crear un nuevo proyecto, Cypress despliega automáticamente una serie de carpetas para conseguir una estructuración del código ordenada. Por defecto se crearán las siguientes carpetas:

- E2E: dentro de esta carpeta se encontrarán todos los archivos *.spec.js, dedicados exclusivamente a la implementación de los test. Generalmente se clasifican anidando carpetas, dependiendo de la funcionalidad que esté cumpliendo la prueba
- *Fixtures*: dedicada a organizar aquellos archivos que contengan datos usados en las pruebas E2E. Aquí se pueden ubicar los datos para realizar el *setup* de nuestra *suite*, los archivos media, los *scripts* para inicializar un entorno, enlaces para la API, etc.
- *Plugins*: contiene un archivo de ejemplo “plugins/index.js” que puede usarse para cargar plugins.
- *Support*: en esta carpeta, se encuentra el archivo “support/commands.js”, usado para almacenar los métodos que queramos convertir en comandos (como si fuesen nativos de Cypress)
- *Screenshots*: dedicada a las capturas de pantalla tomadas al ejecutar las pruebas en modo *headless*.
- *Videos*: cumple la misma función que la carpeta *screenshots*, con la diferencia de que en esta se almacenan vídeos, y no capturas de pantalla.

- *Downloads*: carpeta dedicada a ubicar aquellos archivos descargados durante los test.

Además de estos valores por defecto, al proyecto se añaden ciertas carpetas que participan en la organización de este. Tal y como se ha explicado en el apartado 4.1.1, dedicado a la explicación del patrón de diseño POM, en el proyecto se crea una carpeta dedicada a este patrón de diseño.

Esta carpeta "PageObjects" contendrá todas las clases que hacen referencia a páginas reales de la web con sus métodos propios. También se añade "*helpers*", que cumple la función de almacenar aquellas clases que no tienen una página real en la web, pero se reutilizan en prácticamente toda la *suite*. Aquí podemos encontrar archivos como "globalHelper.js", que contiene métodos de guardados de formularios, de inicio de sesión, de selección de tipos de botones, etc. Otro archivo bastante utilizado es el "dataGenerator.js", cuya función es la de generar datos aleatorios para poder crear elementos de forma única. Estos elementos pueden ser usuarios administradores, usuarios estándar, entidades, clientes, tarifas, etc.

De forma simplificada, en la Ilustración 11, podemos observar la estructura o diagrama de nuestro proyecto. El *framework* proporciona los comandos mediante los cuales estableceremos los selectores (comandos Cypress como "`cy.get('input')`", "`cy.visit('https://example')`", etc.), mientras que el proyecto alberga y proporciona el resto.

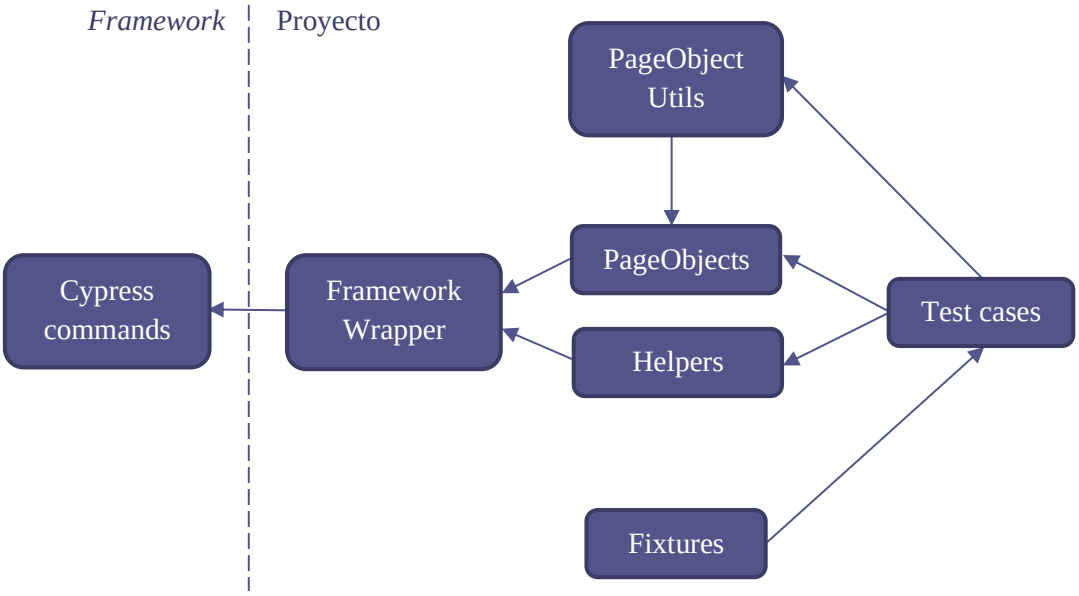
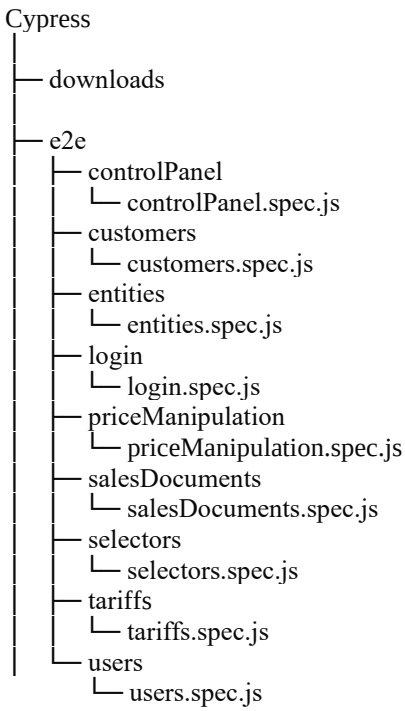



*Ilustración 11 - Diagrama/estructura del proyecto*

## 4.2 Diseño detallado

La estructuración de las pruebas es algo muy importante y para tener en cuenta. Esto nos garantizará que su futura interpretación será la correcta y, además, se logrará con facilidad. A continuación, se presentará una estructura detallada del código de la solución. Dadas las carpetas definidas en el apartado 4.1 sobre la arquitectura del sistema, se procederá a presentar cada una de las clases utilizadas en el proyecto.

En primer lugar, se verán los archivos usados para definir todos los casos de prueba, que son aquellos contenidos dentro de la carpeta Cypress > e2e:

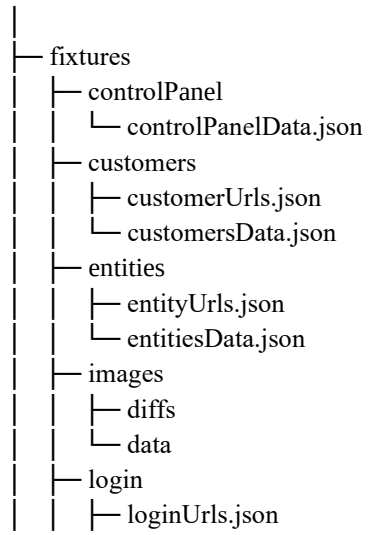



Todos los archivos anteriores (cuya extensión es *.spec.js) son usados para definir los casos de prueba, ordenados de tal forma que, su nombre, indique la página que se está testeando.

A modo de ejemplo, el archivo “login.spec.js” contiene nueve casos. Siete de ellos comprueban que, al introducir datos erróneos en el input de la página de inicio, se muestre un mensaje de error diferente para cada caso, mientras que los otros dos casos comprueban que se inicia sesión correctamente usando credenciales válidas, ya sea vía API o vía interfaz de usuario.

El resto de los archivos funciona de una forma similar, pero con test ajustados a la página que se intenta testear, y de mayor o menor extensión, dependiendo de la importancia de este.

En segundo lugar, se verán los archivos usados para definir todos los datos necesarios para realizar los test o inicializar ciertos procesos. Estos están contenidos dentro de la carpeta Cypress > fixtures:



```
│   │   └── loginData.json
│   ├── modelEditor
│   │   └── modelEditorUrls.json
│   ├── priceManipulation
│   │   ├── priceManipulation_languageData.json
│   │   └── priceManipulation_tariffsData.json
│   ├── salesDocuments
│   │   ├── salesDocumentUrls.json
│   │   └── salesDocumentsData.json
│   ├── selectors
│   │   └── selectorsData.json
│   ├── tariffs
│   │   └── tariffsData.json
│   ├── users
│   │   ├── userUrls.json
│   │   └── usersData.json
│   ├── credentials.json
│   ├── globalIds.json
│   ├── source_es.json
│   ├── source_en.json
│   ├── suiteDataGenerator.json
│   └── url.json
```

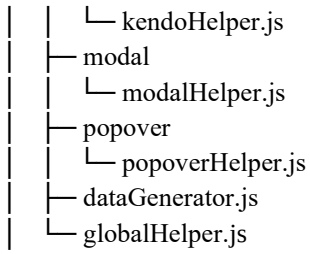
Todos los archivos *.json ubicados en la carpeta *fixtures* corresponden a los datos usados en cada caso de pruebas. En el caso de "entitiesData.json" por ejemplo, se encuentran los datos necesarios para crear una entidad con ciertos valores, para posteriormente testear que se puede crear correctamente con todos los campos disponibles.

Durante el inicio de los test era más relevante el papel de esta carpeta, pero a medida que ha ido avanzando el proyecto, se han optado por decisiones que pueden mejorar esta parte. Una solución que ha dado un gran margen de mejora (sobre todo porque los *fixtures* suelen ser bastante extensos y tediosos de crear), es crear una clase para generar todos los datos necesarios de forma aleatoria, a partir de un conjunto de nombres de diferentes tipos ubicados en el archivo "*suite*DataGenerator.json".

Esta mejora supone una reducción de hasta el 70% del tamaño de los *fixtures*, quedando solo los datos de la creación de algún elemento erróneo o muy concreto. Por ejemplo, para el test de "tariffs.spec.js", previamente se tiene que crear una entidad, un usuario administrador y un usuario estándar para dicha entidad y un cliente creado a partir del usuario estándar. Al inicio del proyecto, se requería de un archivo .json muy extenso, con los datos de cada elemento creado, sin embargo, en la actualidad, solo hacen falta los datos de la tarifa en dicho archivo, ya que el resto se generará de forma aleatoria cada vez que se ejecute el test. Esta mejora también permite que se inicie sesión cada vez con un usuario diferente, permitiendo la paralelización.

En tercer lugar, se encuentran los archivos usados para permitir la reutilización del código durante toda la implementación y para cualquier página de la web. Estos están contenidos dentro de la carpeta Cypress > helpers:

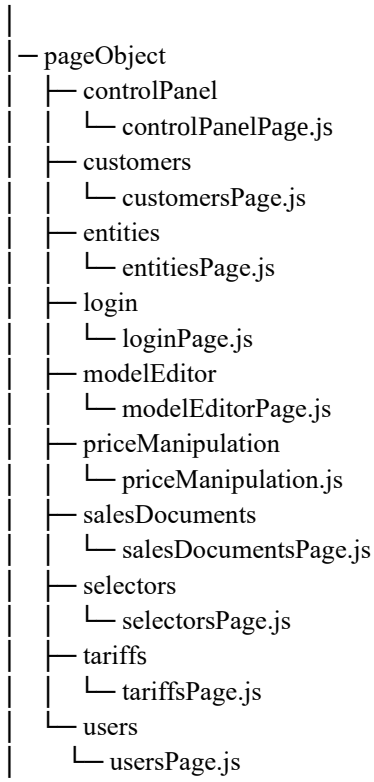
```
│
├── helpers
│   ├── Kendo
```




Como se puede observar en la estructura de archivos superior, tenemos ficheros JavaScript como lo son KendoHelper, modalHelper, dataGenerator, etc. Estos ficheros se encargan de proporcionar los métodos necesarios para poder acceder a los elementos con los que tienen relación, es decir, "KendoHelper" proporciona métodos que se usan en componentes Kendo. Esto es de gran utilidad, ya que hay tablas, *dropdowns*, *inputs*, etc., propios de Kendo, que se utilizan en toda la aplicación de la misma forma, aunque tienen una forma enrevesada de clasificar por clases, atributos únicos y demás. Es por eso que interesa crear métodos como "`getDropdown(dropdownLabel)`", "`setDropdownValue(dropdownLabel, valueToSet)`", "`getElementInsideFilteredColumn()`", y más.

En el caso del archivo "dataGenerator.js", facilita todos los métodos indispensables a la hora de generar datos aleatorios para crear usuarios, entidades, documentos de ventas, etc., utilizando datos predefinidos en un "*fixture*" y combinándolos con datos aleatorios.

En cuarto lugar, se encuentra la carpeta usada principalmente para el patrón de diseño POM. Su estructura es la siguiente:




En cuarto lugar, tenemos los PageObjects. Aquí es donde se encuentra la mayor parte del código. Estos archivos contienen los métodos cuya función consiste en obtener los elementos del DOM de su respectiva página, es decir, el archivo controlPanelPage.js se encargará de almacenar los métodos encargados de obtener los elementos de la página

…prefweb/controlPanel. Estos también se encargan de personalizar otros métodos más genéricos para que funcionen en una página específica (puede ser que algunas veces nos encontremos con inconsistencias en un código que creíamos que era igual para todo).

## 4.3 Tecnologías utilizadas

Tal y como se ha comentado en puntos anteriores, se han tenido en consideración bastantes tecnologías distintas. A continuación, se elaborará una lista con toda la tecnología utilizada y una breve descripción de esta, para lograr entender su funcionalidad, grosso modo.

### 4.3.1 ASP.NET MVC 5

ASP.NET MVC es una plataforma utilizada en el desarrollo de aplicaciones web de Microsoft basada en el patrón de diseño Modelo-Vista-Controlador (de ahí sus siglas MVC). Su arquitectura permite dividir la aplicación en tres partes diferenciadas, beneficiándose de la facilidad de mantenimiento que aporta, facilidad de test unitarios y el desarrollo TDD, la escalabilidad, etc.

Esta plataforma es la que se utiliza en la actualidad para el desarrollo de la aplicación web "PrefWeb", sobre la que se testea en el proyecto. Aunque hoy en día surgen alternativas mucho más atractivas, como lo son Ruby on Rails, Node.Js, Django, entre otros.

Para el caso del proyecto, es necesario entender cómo funciona ASP.NET con el modelo vista controlador, ya que en bastantes ocasiones habrá que modificar el código de la aplicación para adaptarlo a los test, y es crucial entender su estructura

Dado que es un proyecto vinculado a las prácticas de empresa, y este se comenzó unos meses después de la incorporación a dicha empresa, el tiempo de aprendizaje de la herramienta ASP.NET MVC 5 no se ha tenido en cuenta, principalmente porque ya se había adquirido el conocimiento necesario como para poder entender su funcionamiento.

### 4.3.2 Cypress

Cypress es una herramienta de testing E2E basada en JavaScript. Esta herramienta permite diseñar de forma sencilla diferentes test, al igual que también permite observar en tiempo real como se desenvuelven.

Su arquitectura ha sido construida desde cero, lo que implica que no está basada en Selenium. Una de sus características principales es que permite acceder a cualquier elemento de la web, ya sean objetos nativos o tráfico de la red. Es por ello por lo que logra abarcar muchos aspectos y lo convierte en una de las herramientas más completas que existen para la automatización de pruebas E2E en la web.

Un punto muy importante y que ha tenido una fuerte repercusión en el proyecto, es la cantidad de información que existe frente a posibles problemas que pueden surgir mediante el desarrollo, además de la relativa facilidad con la que se aprende el lenguaje de este *framework* gracias a que está basado en JavaScript y tiene un lenguaje de lo más *verbose* (muy parecido al lenguaje natural).Aun así, se han seguido diversos vídeos de aprendizaje proporcionados por la propia

empresa Cypress, junto con documentación relacionada con las buenas prácticas y los antipatrones.

Al inicio del proyecto, se estimó un período de adaptación a este nuevo marco de programación de aproximadamente dos semanas hasta familiarizarse con el lenguaje y su peculiar, aunque familiar funcionamiento (en especial por la falta de *asyncs/awaits*, ya que es algo esencial en JavaScript y bastante frecuente de usar). Más adelante se comprobará que aunque se trata de un lenguaje sencillo de aprender, cuyos comandos están basados en Mocha (OpenJs Foundation, 2022) y Chai (Open Source, 2022), hay ciertos puntos más específicos que son complejos de resolver y a en ocasiones también resultan poco intuitivos de resolver.

Dicho esto, se espera que el aprendizaje y la adaptación a este *framework* no supere las dos semanas desde su inicio, aunque llevará un poco más de tiempo perfeccionar y optimizar los comandos que son utilizados.

### 4.3.3 Cypress Dashboard

Cypress Dashboard es un complemento opcional basado en la versión web de la aplicación Cypress (Cypress.io, 2022). Proporciona estadísticas, tiempos, resultados y posibles mejoras en tan solo un vistazo, debido a que toda la información de los test ejecutados se resume y se analiza en esta plataforma al instante.

Aunque es una característica premium (de pago), también permite la paralelización automática y el equilibrio de carga, la optimización de la CI, entre otros. Otra característica interesante que aporta este dashboard es que permite almacenar las grabaciones de los test en la nube, consiguiendo que aquellos miembros del equipo que estén suscritos al proyecto lo pueden visualizar y estudiar en cuestión de segundos, sin la necesidad de pasar información manualmente

Esta herramienta se plantea ser usada en menor medida para la elaboración de los test, ya que no tiene tanta funcionalidad si solo participa un miembro en el proyecto. Aun así, se reserva su uso de cara al final de este, cuando se empiecen a paralelizar los test.

Su uso es claramente intuitivo y no necesita más que de unas horas para acostumbrarse a la interfaz y conocer sus características más interesantes, con lo que se estiman unas pocas horas de aprendizaje de esta herramienta, considerando incluso no dedicarle un tiempo exclusivo, sino ir conociéndola y familiarizándose con ella sobre la marcha.

### 4.3.4 Testing Library

Testing-Library en 2020, fue premiado por ser la tecnología con el porcentaje más alto de usuarios satisfechos al tener una valoración positiva en un 97% de los casos. Esto se debe principalmente a que es una librería que permite testear centrándose exclusivamente en el usuario, y no en elementos del DOM que poco tienen que ver con el usuario (Id, test id, coordenadas, etc.)

Generalmente se usarán los selectores (Dodds, 2022):

- Por rol (getByRole): el rol de un elemento es el utilizado para aumentar la accesibilidad de las pruebas. Este rol se encuentra definido en la propiedad *role*.
- Por texto (getByText): garantiza que se pueda acceder a un elemento por su texto (al final es aquello que lee el usuario y, por lo tanto, en lo que se centra
- Por etiqueta o *label* (getByLabelText): el *label* es aquel texto vinculado a un *input* (campo en el que se introducen datos). Esta funcionalidad es muy interesante, ya que simula como un usuario real accede a los *inputs* que necesita, dependiendo de su etiqueta. Por ejemplo, si tenemos un formulario en el que existe una etiqueta llamada "Name", ubicada en la parte superior de un *input*, el usuario interpreta que ese cuadro de inserción de texto sirve para introducir un nombre. Este selector devuelve el *input*, y no el propio texto de la etiqueta.

Testing-Library es una librería muy importante para este proyecto y no requiere de demasiada documentación previa, con lo que es un activo de lo más esencial y versátil. Realmente, se necesita más documentación e investigación sobre la filosofía que sigue que por los propios comandos que usa.

### 4.3.5 SQL Server

Microsoft SQL Server es un sistema de administración de bases de datos relacionales basado en SQL, un lenguaje de programación utilizado para administrar dichas bases de datos y hacer consultas a estas. (Adam Hughes, 2022)

En el caso del proyecto, es usado para administrar las bases de datos de los clientes cuya finalidad consiste en proporcionar escenarios reales a los que se enfrentan al utilizar la aplicación. Más que datos de usuarios, materiales, productos y demás, son datos utilizados para personalizar la aplicación web, ya que esta no es única y general (hecho que complica la ejecución de los test). Digamos que estos datos almacenados son configuraciones personalizadas para cada cliente (entendamos por cliente empresas de carpintería de aluminio internacionales), que permiten que existan en paralelo diversas versiones de PrefWeb.

Dicho esto, es de interés saber envolverse en este sistema de administración para poder variar entre bases de datos, comprobar ciertos valores que no se pueden comprobar mediante la web, clonar y preparar bases de datos para test, etc. Esto requiere de un proceso algo más tedioso, ya que hay bastante información y puede resultar complicado encontrar aquello que se busca.

Dado que ya se conocía esta herramienta, no será necesaria demasiada documentación para lograr aquello que se desea realizar en este proyecto. Se estima que el período de adaptación a esta herramienta sea de una semana.

### 4.3.6 Inspector de elementos

El inspector de elementos es una herramienta que permite analizar el código de un sitio web e interactuar con él (Pou, 2022).

Esta herramienta es esencial para lograr que los selectores de los test seleccionen aquellos elementos del DOM que realmente pretenden seleccionar. Con el inspector de elementos se tiene acceso a los textos, Ids, clases y demás atributos de interés. Además, permite controlar el

flujo de red de nuestra aplicación, el almacenamiento en local del navegador (también conocido como *localStorage*), las cookies, permite *debuggear* en tiempo real código JavaScript, etc.

Cypress proporciona una ventana en la que se ejecutan las pruebas y donde podemos acceder en cualquier momento al contenido del DOM mediante el inspector de elementos, siendo el único *framework* de testing capaz de lograr esta increíblemente útil función.

Dado que es una herramienta comúnmente presente en la docencia, no se requiere de un aprendizaje previo.

### 4.3.7 C#

C# es un lenguaje orientado a objetos y a componentes, fuertemente tipado. C# permite a sus usuarios crear aplicaciones sólidas y seguras, a la par que eficientes, que se ejecutan en .NET. Es similar a los lenguajes de programación: Java, C y C++ (Microsoft Corporation, 2022).

Este lenguaje es usado en la aplicación de la empresa, ya que usa la plataforma ASP.NET con el patrón de diseño MVC5.

No hay demasiado interés en manipular el *backend* del *software* PrefWeb, así que bastará con una breve lectura de la documentación en el momento en el que se requiera su manipulación, aunque, por su similitud con Java (lenguaje predominante en los estudios universitarios cursados), no debería de llevar demasiado tiempo. La estimación es que lleve dos semanas adaptarse a su uso.

### 4.3.8 Razor syntax

La sintaxis de Razor se basa en C# y HTML, con una extensión de archivos nombrada como *.cshtml. Esta sintaxis es similar a los *frameworks* que usan SPA (*Single Page Application*), como React y Vue (Microsoft Corporation, 2022).

Razor cobra un papel importante en los test cuando se trata de manipular el código de la aplicación de la empresa para adaptarlo a los test. Un caso podría ser, por ejemplo, en el que una etiqueta de un *input* no usa correctamente su atributo "*for*", causante posible de problemas cuando se trata de seleccionar algún elemento del DOM mediante Testing-Library. Como la aplicación usa el *framework* "Kendo", se emplea abundantemente Razor en los elementos del DOM, derivando así en el inevitable aprendizaje de esta sintaxis.

Aunque Razor tiene una sintaxis y unos métodos un tanto peculiares como "ViewBag" (usado para transferir datos temporales desde el controlador a la vista), ha sido implementado en tantas ocasiones en la aplicación, que casi cualquier circunstancia ya está implementada, y puede ser una gran fuente de apoyo para posibles implementaciones futuras.

### 4.3.9 JavaScript

JavaScript es un lenguaje de programación interpretado y orientado a objetos, enfocado en el desarrollo de páginas web. Es la tercera capa de las tecnologías web (siendo las dos primeras

HTML y CSS). Este lenguaje permite crear actualizaciones dinámicas, controlar multimedia, animar elementos, etc. (Mozilla Corporation, 2022)

Este lenguaje es el predominante en el proyecto, ya que Cypress está basado en JavaScript, lo que inicialmente ya lo hace el lenguaje primordial y, además, la aplicación que se desea testear es una aplicación web, que a su vez utiliza un alto porcentaje de este lenguaje

El período dedicado al aprendizaje de este lenguaje, al igual que a los demás lenguajes, será mínimo, ya que son predominantes en los estudios universitarios y, a lo sumo, aquello que no se conozca se puede buscar fácilmente en la web. Teniendo en cuenta lo descrito anteriormente, no se considerará un período de aprendizaje de este lenguaje.

#### 4.3.10 HTML 5

HTML 5 es la versión estable de HTML. Este término surge de (Lenguaje de Marcado de HiperTexto o HyperText Markup Language por sus siglas en inglés). Es un lenguaje descriptivo que da forma a la estructura de los sitios web (Mozilla Corporation, 2022).

Siendo un lenguaje tan común, no se requiere de un período de aprendizaje para lograr realizar el proyecto.

# 5. Desarrollo de la solución propuesta

Desde el inicio, la intención de este proyecto es mejorar drásticamente los medios mediante los cuales se testeaba toda la aplicación web de la empresa Preference S.L. (PrefWeb). Para lograr esto, se ha desarrollado una *suite* de test desde cero, basándose en las pruebas ya existentes, aunque poco eficientes.

Para pasar de la propuesta a la solución final, se ha llevado a cabo una larga implementación de código, con constantes cambios en el desarrollo. Es por ello por lo que a continuación se explicará en detalle el proceso seguido para llegar a la solución final.

## 5.1 Puesta a punto de la base de datos

El primer paso para empezar el desarrollo ha sido el de preparar un a base de datos con la configuración utilizada por uno de los clientes más grandes de Preference. Para ello, se ha usado una copia de seguridad antigua, se ha creado una nueva copia a partir de esta y se han dejado tan solo los datos necesarios.

Tras realizar la nueva copia de seguridad, se han insertado nuevas configuraciones esenciales en los test. Estas configuraciones se han añadido mediante variables globales y transacciones SQL sobre dicha base de datos. Una de las variables globales manipuladas, ha sido aquella que permite iniciar sesión mediante una API, y que no viene habilitado por defecto. Otra configuración previa que ha sido la de permitir que un super usuario pueda acceder también vía API (esta opción no se podía habilitar, así que se llegó a un acuerdo con el líder del equipo para poder habilitarla).

Una vez hechas las modificaciones en la nueva base de datos, se creó una copia de seguridad definitiva. Dicha copia de seguridad sería la que se utilizaría posteriormente para restaurarse después de cada ejecución de la *suite* de test, y así garantizar que los test no dependen los unos de los otros.

## 5.2 Creación de comandos personalizados

Una vez preparada la base de datos, habría que centrarse en los objetivos establecidos al inicio del proyecto, para saber en qué enfocarse. Uno de los objetivos era la mejora tiempos de ejecución, y una de las soluciones es la de crear un comando personalizado de inicio de sesión para poder hacer *login* mediante la API.

Al inicio, se creó tan solo un método para iniciar sesión mediante la API, y otro para iniciar sesión mediante la interfaz de usuario. Estos métodos eran bastante simples y solo recibían un parámetro de entrada (el usuario). Más adelante fueron apareciendo ciertos problemas a la hora de iniciar sesión, como que, al cambiar el lenguaje en algún punto, este permanecía en el mismo idioma al que se había cambiado. Esto podía provocar que cuando se intentase seleccionar un elemento del DOM mediante texto, no funcionase, debido a que se encontraba en otro idioma.

Tal y como vemos en la Ilustración 12, la solución frente a este problema es añadir otra función al login, que permita establecer el idioma en el que se intenta iniciar sesión.

```
128
129
130 Cypress.Commands.add('loginByApi', (language, user, redirectTo = url.homeUrl) => {
131     //language has to be "es" or "en", abbreviations for Español and English
132     cy.session([`${language}_session`, user.Name, user.Email], () => {
133         cy.baseLoginByApi(user)
134         cy.visit('')
135         cy.setLanguageTo(language)
136     })
137     cy.visit(redirectTo)
138     if (redirectTo !== '/home')
139         cy.url().should('equal', url.baseUrl + redirectTo)
140 });
141
142
```

*Ilustración 12 - Comando personalizado de inicio de sesión*

Para explicar un poco como funciona, se describirán las funciones de los métodos línea por línea:

- En la cabecera, se pueden observar tres parámetros: *language*, *user* y *redirectTo*. Estos hacen referencia al idioma en el que se quiere iniciar sesión, las credenciales del usuario y, opcionalmente, una url a la que desplazarse una vez se inicia sesión, respectivamente.
- En la línea 132, se puede observar el comando cy.session. Este comando es una reciente incorporación de Cypress que, tal y como se puede leer en su documentación (Cypress.io, 2022), se encuentra en fase experimental. Partiendo de la base, Cypress borra todo el *localStorage* y las *Cookies* del navegador entre cada test para prevenir malas prácticas. Un problema de esta función es que se pierden las *Cookies* de inicio de sesión. El comando cy.session aporta pues un poco de luz a este problema, proporcionando un método para guardar todos los datos (*Cookies* y *localStorage*) generados dentro de este comando para posteriormente usarlos. Por ejemplo, si iniciamos sesión con un usuario llamado Alex, cuyo correo (que es único) es 1234@correo.com, se guardarán dichos datos. Si en otra ocasión se intenta iniciar sesión con los mismos datos, en vez de ejecutar los comandos de inicio de sesión de su interior, recuperará las *Cookies* y demás para así iniciar sesión en unas centésimas de segundo.
- Líneas 133-135. En estas líneas se llama a otros métodos encargados de iniciar sesión por API y comprobar las *Cookies*, visitar PrefWeb (aunque el *"visit"* no contenga una url, esta va definida en la configuración de Cypress, y es la de la aplicación web) y establecer el idioma requerido. Posteriormente se cierra el comando *session* y se almacenan dichos datos (incluidos los del idioma).
- En el pie del método, se redirige a la url deseada, ya que una vez se cierra el comando *session*, Cypress sale de la navegación y deja una página en blanco.

Otro comando que apareció más adelante fue el de comprobar las cookies de inicio de sesión. Este comando permitía comprobar que se habían creado las cookies necesarias y que la sesión se había establecido correctamente. Posteriormente se utilizaría para comprobar que un usuario creado vía API, lo había hecho de forma satisfactoria.

## 5.3 Desarrollo de las pruebas

Bajo la recomendación de un miembro del equipo (encargado del desarrollo de los test anteriores con Ranorex), se empezaron a desarrollar los casos de prueba para la página del login, debido a su simplicidad. Seguidamente su recomendación era seguir por los test de documentos de ventas (otra página de PrefWeb) y, tras estos test, seguir libremente desarrollando, ya que se habría adquirido el conocimiento necesario para afrontar los siguientes.

Un punto para tener en cuenta a la hora de implementar los test es que no se deben de repetir las entidades, los usuarios o ningún dato que pueda depender de otro. Esto garantizará la futura ejecución de los test en paralelo sin que la manipulación de unos datos antes que otros afecten. Otro punto igual de importante y a su vez relacionado, es que se debe de iniciar sesión siempre con un usuario distinto, ya que, al ser PrefWeb un *software* de pago, actúa con licencias únicas para cada usuario que impiden que haya dos conectados en paralelo.

A continuación, se detallan todos los test implementados con sus respectivos problemas individuales.

### 5.3.1 Login

Es este caso de prueba, se procura testear la página de inicio de sesión. Esta página, al igual que las demás, tiene una alta importancia, debiendo de mostrar los mensajes adecuados en los casos que sea necesario y debiendo funcionar a la perfección. Aunque la aplicación funcione perfectamente, si un usuario no consigue iniciar sesión de forma correcta, es como si no existiese nada más allá de dicha página.

Para este test, hay que tener en consideración tres factores:

1. Se debe de probar que los mensajes de validación se muestren correctamente en caso de fallo, es decir, test de fallos.
2. Se debe de probar que se inicia sesión correctamente, tanto vía interfaz de usuario, como vía API, es decir, test de éxitos.
3. No se debe de iniciar sesión en otros test, siguiendo estas pruebas, ya que no conviene volver a testear aquello que ya se ha probado en algún momento (la solución viene dada mediante la creación de un comando personalizado que permita el inicio de sesión, comprobando tan solo lo esencial como, por ejemplo, que se hayan establecido correctamente las cookies de la sesión).

Junto con las consideraciones tomadas en los puntos anteriores, un paso muy importante en la implementación de los test es ordenarlos y describirlos de tal forma que podamos identificar el error (en caso de falla) sin un ápice de esfuerzo.

Vemos un ejemplo:

En este ejemplo vemos un buen uso de las descripciones de los test.

- Describe: Filter grid test
- It: Should return 2

En este ejemplo vemos un buen uso de las descripciones de los test.

- Describe: Filter the column named “Entity” so that it returns rows which equal "Preference" or start with "Cl".
- It: We expect the grid TO HAVE 2 rows

En este ejemplo se ha demostrado que mediante el uso del mismo número de descripciones, en el primer caso se entiende el caso general pero, si fallase, no se sabe realmente porqué podría haber fallado, mientras que en el segundo caso, se sabe en todo momento qué se está haciendo, cómo se está haciendo y qué se espera que se devuelva.

En la siguiente ilustración (Ilustración 13), se muestra el resultado de haber ejecutado los test de login, pudiendo observar los casos de prueba que se están realizando y los resultados de la prueba, al igual que los tiempos de ejecución (18 segundos)

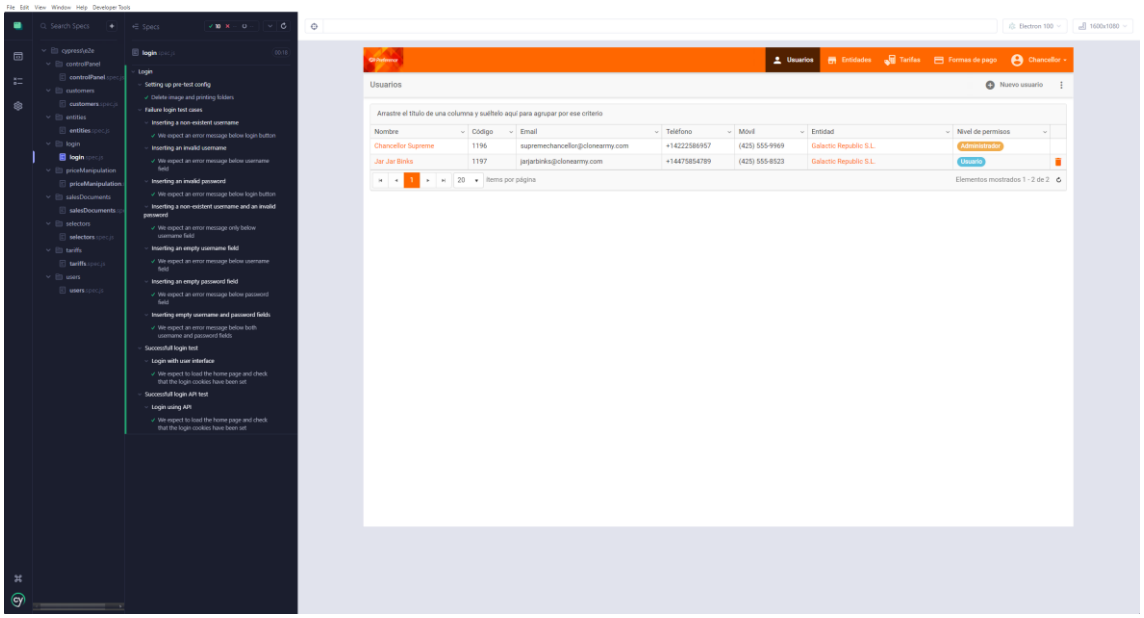


*Ilustración 13 - Ejecución de los casos de prueba de la página "Login"*

### 5.3.2 Documentos de venta

Esta página permite crear, editar, consultar, bloquear y filtrar documentos de ventas. Dentro de un documento de ventas, se pueden añadir modelos (ventanas, puertas o cualquier elemento disponible). A su vez, un documento de ventas pasa por diferentes estados, como versión aceptada, pedido creado y pedido confirmado.

Las pruebas de esta página serán bastante extensas, considerando que se van a testear todas estas funciones, desde el flujo de todos los estados posibles de un documento de ventas, hasta el filtrado de documentos entre todos los existentes.

Al concluir la implementación de los test, se obtendrá un PageObject de la página documento de ventas y del *framework* Kendo, bastante completo.

Antes de nada, se dividen los test según las diferentes funcionalidades que tiene esta página. En total, se crearán los siguientes casos de pruebas (sin entrar en detalle):

## Crear Documento

Este caso de pruebas consiste en introducir unos datos previamente estipulados en un formulario, emulando que es un usuario el que está introduciendo estos datos. Posteriormente se crea el documento de ventas y se comprueba que, tras la llamada *POST* que guarda dicho documento en la base de datos, aparezca en la tabla de documento de ventas, tal y como aparece en la Ilustración 14

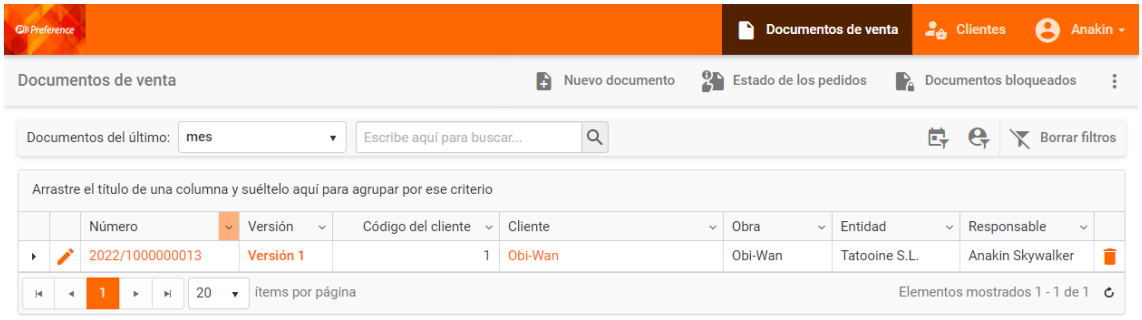



*Ilustración 14 – Documento creado durante el test "Crear Documento"*

Una peculiaridad de este test es que cuando se han creado más de veinte documentos de venta, la probabilidad de que un documento nuevo se muestre en la primera página de la tabla disminuye. Es por eso por lo que se decidió crear un método que filtrase previamente por el nombre de aquello que se iba a crear (el fondo naranja en la columna Número indica que se ha filtrado por ese campo). Esto permite que, aunque la base de datos esté inundada de información, siempre se muestre aquello, y solo aquello, de interés para el test. El nuevo método "filterColumnByName" pertenecerá al PageObject de Kendo, también llamado KendoHelper, ya que dicha tabla es una *kendoGrid* (tabla que pertenece a Kendo).

El caso de crear un documento de ventas no solo involucra dicho documento, sino que previamente hay que conseguir crear: una entidad (tienda o empresa), usuarios, tarifas, clientes y posteriormente crear el documento de ventas para todos los valores anteriores. Es un proceso largo que hasta ahora se realizaba creando los valores mediante la interfaz de usuario, y que ahora se hace mediante la API, por lo menos hasta el punto de crear el documento.

## Eliminar Documento

Previamente, este test se ejecutaba consecutivamente del test "Crear Documento". Se hacía porque así aprovechaba la creación de un nuevo, ya que no se creaba nada por API y las creaciones por interfaz usando la herramienta Ranorex eran muy lentas. Este problema se ha conseguido solucionar con la implementación de llamadas API.

Por ello, este test es bastante simple, ya que esencialmente trata de crear un usuario por API y eliminarlo mediante interfaz de usuario. Tal y como se puede observar en la Ilustración 14, a la derecha del todo del documento aparece un icono de una papelera, con el que se puede borrar

dicho documento. Simplemente se comprueba que tras la llamada *POST* que verifica que se ha lanzado la llamada de borrado, deje de existir el documento en la tabla de Kendo

Finalizar pedido

En este caso se comprueba que se realiza correctamente el flujo natural de un documento de ventas en la aplicación. Dicho flujo consiste en lo siguiente:

Abrir Documento > Añadir modelos > Aceptar versión > Crear pedido > Confirmar pedido

Tras confirmar el pedido, se envía un PDE (Preference Data Exchange) a fábrica, que es el que usará la fábrica para elaborar aquellos modelos que hay en el documento.

Los errores principales que podrían surgir en este test son que no cargue correctamente la información, que los modelos originales se acaben manipulando, que los precios no sean los mismos y que el flujo no sea el mismo (como es una aplicación configurable, cada cliente puede establecer el envío de PDEs cuando considere, pero en el caso de esta base de datos, es tras confirmar el pedido).

Imprimir Documento

Cuando se tiene un documento, pero se quiere exportar a modo de presupuesto para enviárselo manualmente a un cliente, existe la opción de imprimir un documento. Una forma de testeo de esta funcionalidad es la de comparar la impresión que queremos conseguir con la impresión que se realiza actualmente en la web. Es por ello por lo que inicialmente se requiere de un PDF correcto para poderlo comparar con el PDF generado.

La comparación entre documentos es realmente una comparación de imágenes, y se realiza con el plugin “pixelmatch”, explicado en el siguiente documento (npmjs, 2022), porque Cypress no tiene la opción de comparar imágenes.

En este caso, bastará con crear una entidad, usuario, tarifa, cliente y documento por API y seguidamente imprimir el documento para poderlo comparar con pixelmatch. Este plugin generará un archivo con la diferencia entre las imágenes (si la hay) y devolverá el porcentaje de similitud. Únicamente hay que configurar el umbral con el que se detectaría una diferencia entre documentos impresos.

Filtrar Documentos

Este caso concreto es el que más tiempo de ejecución abarca, ya que hay que ir comprobando todos los filtros posibles (igual que, contiene, mayor que, menor que, etc.) en combinación a los tipos de columna (cadenas de texto, números, fechas y elementos personalizados).

Inicialmente se estimaba aproximadamente una duración de dos semanas hasta completar todos los test de filtrado, pero no fue hasta cuatro semanas después cuando se completó. Kendo fue

uno de los principales problemas por la cantidad de scripts que se ejecutaban tras cargar la página.

Para contextualizar, Cypress es cierto que espera a que se cargue la página o los eventos que ocurren en esta hasta continuar, pero, si la página se ha cargado y quedan scripts por ejecutarse (animaciones, por ejemplo), Cypress no se espera a estos eventos. Esta pequeña diferencia causaba un quebradero de cabeza porque al usar el comando "cy.type('escribealgo')", empezaba a escribir pero tras varios milisegundos, se perdía el foco y solo se escribía o la primera o la primera y la segunda letra. Un tiempo después la solución llegó al entender el problema, y es que, cuando se ejecutaba el comando *type*, aún se estaban ejecutando ciertos scripts, y justo coincidía que se estaba estableciendo el foco en un dropdown cercano, lo que provocaba la pérdida del foco en el input, forzando que la escritura no se completase.

La solución al problema anterior consistía en esperar a que el elemento que se esperaba que tuviese el foco, efectivamente lo tuviese, para posteriormente continuar con los demás comandos. Este error sirvió sobre todo para entender cómo funcionaba Kendo y Cypress. Resuelto este problema, los test se iban implementando con mayor facilidad.

### 5.3.3 Entidades

Esta página permite crear, editar, eliminar y consultar entidades. Estas entidades, empresas o tiendas (se puede llamar de las tres formas, aunque en el proyecto se referirá siempre al concepto de entidades) tienen un máximo de usuarios establecidos muy importante para las empresas que lo contratan. Es por eso por lo que los casos de uso se dividirán en crear una entidad, editarla, y comprobar si se establecen correctamente los máximos de usuarios creados para dicha entidad, ya sea los casos en los que se resuelve con éxito, como los que se resuelven con fallos (se testea que los mensajes de validación son los correctos).

Para poder crear una entidad tan solo es necesario ser un super usuario (*super user* o *root user*). Dicho esto, los test se han dividido siguiendo las siguientes funcionalidades:

Crear Entidades

El caso de crear entidades es bastante simple en comparación a los demás test, ya que, según los datos proporcionados por su *fixture* (recordemos *fixture* como archivo en el que dejan los datos necesarios para ejecutar ciertos test), se crea una entidad y se comprueba que, tras la llamada *POST* de creación de la entidad, aparezca en la tabla de entidades de la página principal.

A causa de la realización del test de documentos de ventas en el apartado anterior, nos encontramos con que ya hay PageObjects creados para manipular datos con las tablas de Kendo, con lo que no supone mucho esfuerzo verificar que existe la entidad en la tabla. Al igual que con los documentos, antes de crearla, se aplica un filtro con el nombre de la futura entidad, para que así solo se muestre la información relevante en caso de error.

Una de las principales diferencias con respecto al test anterior de creación de documentos, es que no requiere de crear nada por API previamente, ya que la entidad se encuentra en el escalón más alto de los flujos de PrefWeb, aunque es cierto que sí se requerirá de la API para ejecutar

las próximas pruebas de eliminar entidades y de probar que el máximo de usuarios establecido para una entidad funcione.

Eliminar Entidades

Este caso es muy similar al de eliminar un documento de ventas, primordialmente porque ambos se crean por API para luego ser eliminados mediante el botón con forma de papelera, ubicado a la derecha del todo en la *grid* de Kendo, tal y como podemos volver a observar en la Ilustración 14.

Siguiendo los mismos pasos que con el documento de ventas, tras hacer clic en el botón de eliminar y tras encontrar la llamada *POST* que confirma que se ha eliminado la entidad, se comprueba que esta no aparezca más en la tabla de Kendo, de la misma forma que se había hecho antes en un test anterior.

Al llegar a este punto nos damos cuenta de la importancia de los PageObjects y de reutilizar los métodos. También, al estar estos organizados dependiendo en la página en la que se encuentran en la web, la búsqueda se estos métodos se vuelven mucho más simple.

Mediante Interfaz de usuario, comprobar máximo de usuarios permitido

En estos test, solamente se comprueba que el número máximo de usuarios que se establezca en la configuración de la entidad antes de crearla se cumple. Realmente esta funcionalidad se testea por la importancia que cumple para algunos clientes.

Dicho esto, este test se dividirá en dos partes.

La primera parte, probará que efectivamente no se permiten crear usuarios una vez superado el máximo de usuarios por defecto (el máximo por defecto es un usuario)

La segunda parte, probará que efectivamente no se permiten crear usuarios una vez superado el máximo de usuarios tras haberlo ampliado (el máximo pasa a ser cinco)

A la vez que se prueban estos test de fallo, que directamente se espera que falle el test y se muestre un mensaje de error, ya que esta es su funcionalidad correcta y esperada, también se prueba el tipo de mensaje que se muestra y el texto que contiene, ya que se espera que sea un texto que defina bien el error, con lo que se debe probar.

Mediante API, comprobar máximo de usuarios permitido

El test de comprobar el máximo de usuarios mediante la API es el mismo que el de interfaz, solo que los usuarios son creados por API, como su nombre indica. Aunque este test no se encuentra entre los test de la antigua *suite* de Ranorex, se ha añadido debido a la reciente incorporación de algunas empresas que sí que utilizan esta funcionalidad a través de la API.

Al igual que el test anterior, se divide en las mismas dos partes y se comprueba que el mensaje de error sea el que se espera.

### 5.3.4 Usuarios

Esta página permite, principalmente crear usuarios y eliminarlos. Como existen dos tipos de usuario, se debe probar que ambos tipos se pueden crear sin problema, ya que varían algunos inputs, que se encuentran disponibles o no dependiendo de si es un tipo u otro.

Para poder crear, eliminar o editar un usuario es necesario ser un super usuario (*super user* o *root user*). A continuación, se detalla la implementación de ambos test:

Crear Usuario

Para crear un usuario, se requiere de un fixture de datos del usuario y de una entidad creada previamente por API. Las creaciones por API se pueden realizar con el PageObject de DataGenerator, ya que, solo se necesita una entidad, sin importar el contenido de sus datos, mientras que, para el usuario, si generásemos los datos por el método de generación de usuario, no rellenaría todos los campos debido a que solamente genera los necesarios para poderse crear.

Una vez se ha creado la entidad y el usuario, se comprueba que, tras la llamada POST de creación de usuario, este aparezca en la tabla de Kendo de la página usuario. Una vez más, se reutilizan diversos métodos, entro otros, los métodos de la clase KendoHelper para seleccionar la fila correspondiente de la tabla.

Eliminar Usuario

Al igual que eliminar una entidad y un documento de ventas, este se hace creando previamente los datos necesarios por API (en este caso, una entidad y un usuario para dicha entidad), y se eliminará de la misma forma. Este test se repetirá y será el mismo tanto como para usuarios estándar como para usuarios administradores.

### 5.3.5 Tarifas

El funcionamiento de una tarifa es un tanto especial (entiéndese por tarifa una cuota a la que se le puede emplear un incremento o descuento), ya que esta puede existir sin necesidad de ser asignada a ninguna entidad.

Dicho esto, para poder crear una tarifa, previamente se tienen que crear: una entidad y un usuario administrador, que esté asignado a dicha entidad. A continuación, se muestran los detalles de las implementaciones de los test:

Crear Tarifa

Para crear una tarifa, es necesario crear previamente una entidad y un usuario administrador. El usuario estándar solo hará falta en el caso de que queramos usar una tarifa en un documento de ventas.

Tras crear el usuario y la entidad por API, se crea la tarifa mediante datos aleatorios, generados a partir de la clase DataGenerator previamente creada, que contiene el método "generateTariff". En esta ocasión se generan datos aleatorios, ya que una tarifa solo acepta tres campos: el nombre de la tarifa, y el incremento/descuento/ninguno de los dos. Como el incremento y el descuento son datos que influyen en el precio, estos se pasan como parámetros si fuese necesario.

Una vez se crea la tarifa, tras la llamada *POST* que verifica que se ha creado, se comprueba que esta aparezca en la tabla de Kendo. En esta ocasión surge un problema, y es que, aunque sigue siendo una tabla de Kendo, es una versión distinta con menos columnas y funcionalidades, con lo que cambia la forma con la que se accede a los elementos. Esto es fácil de solucionar, ya que añadimos un segundo tipo de tablas en el PageObject de Kendo y listo.

Mientras tanto, los PageObjects de las demás páginas siguen aumentando y cada vez, la dificultad de seleccionar los elementos del DOM disminuye, al igual que la cantidad de código escrito cada vez aumenta en menor proporción.

Asignar Tarifa predeterminada a Entidad (tarifa aplicada a los usuarios de la entidad)

Asignar una tarifa a una entidad para el disfrute de sus usuarios consiste en un proceso sencillo. Básicamente hay que crear una entidad y un usuario administrador por API, al igual que en el apartado anterior, solo que, en esta ocasión, también se crea la tarifa vía API. Esto permite que el test genere los datos en pocos segundos y que, tan solo haya que acceder vía interfaz a la página de entidades, y asignarle la nueva tarifa creada a dicha entidad.

La tarea se lleva a cabo de forma relativamente rápida debido a que ya existe un PageObject de la página entidades, hecho que permite reutilizar métodos de acceso a elementos.

Este test se comprueba volviendo a entrar en la entidad y revisando que los campos son los que se esperaban, al igual que se comprueba que la tarifa predeterminada tenga el mismo nombre que la tarifa creada exclusivamente para este test.

Asignar Tarifa a Entidad (tarifa aplicada a la entidad)

Se consigue exactamente igual que en el apartado anterior, solo que modificando un *dropdown* distinto.

Eliminar Tarifa

Para eliminar una tarifa, basta con crear una previamente, junto con la entidad y el usuario administrador (toda vía API). Se comprobará que se haya eliminado de forma correcta, comprobando que exista la llamada post de eliminar tarifa y, posteriormente, verificando en esta segunda tabla de Kendo que, efectivamente, ya no se encuentra dicha tarifa.

### 5.3.6 Clientes

En el caso de pruebas de los clientes, los tipos de test que se deben ejecutar son los mismos que para los usuarios, así que en esta ocasión solo se mostrarán los tipos de test implementarán, sin entrar en detalle.

Estos test son los de crear clientes y eliminar clientes.

La única dificultad de crear y eliminar clientes es la cantidad de elementos que hay que crear previamente para llegar a este punto. Para poder crear un cliente, previamente hay que crear una entidad, crear un usuario estándar, crear un usuario administrador, crear una tarifa con dicho usuario administrador, asignar la tarifa a la entidad para que esté disponible para el cliente y, por último, a la hora de crear el cliente por interfaz, asignarle la tarifa creada con anterioridad.

El proceso de eliminar el cliente es exactamente el mismo, solo que también se crea el cliente. Todo esto, por API.

### 5.3.7 Manipulación de precios

La manipulación de precios se encuentra tras haber creado un documento de ventas con sus respectivos modelos, tal y como se muestra en la Ilustración 15.

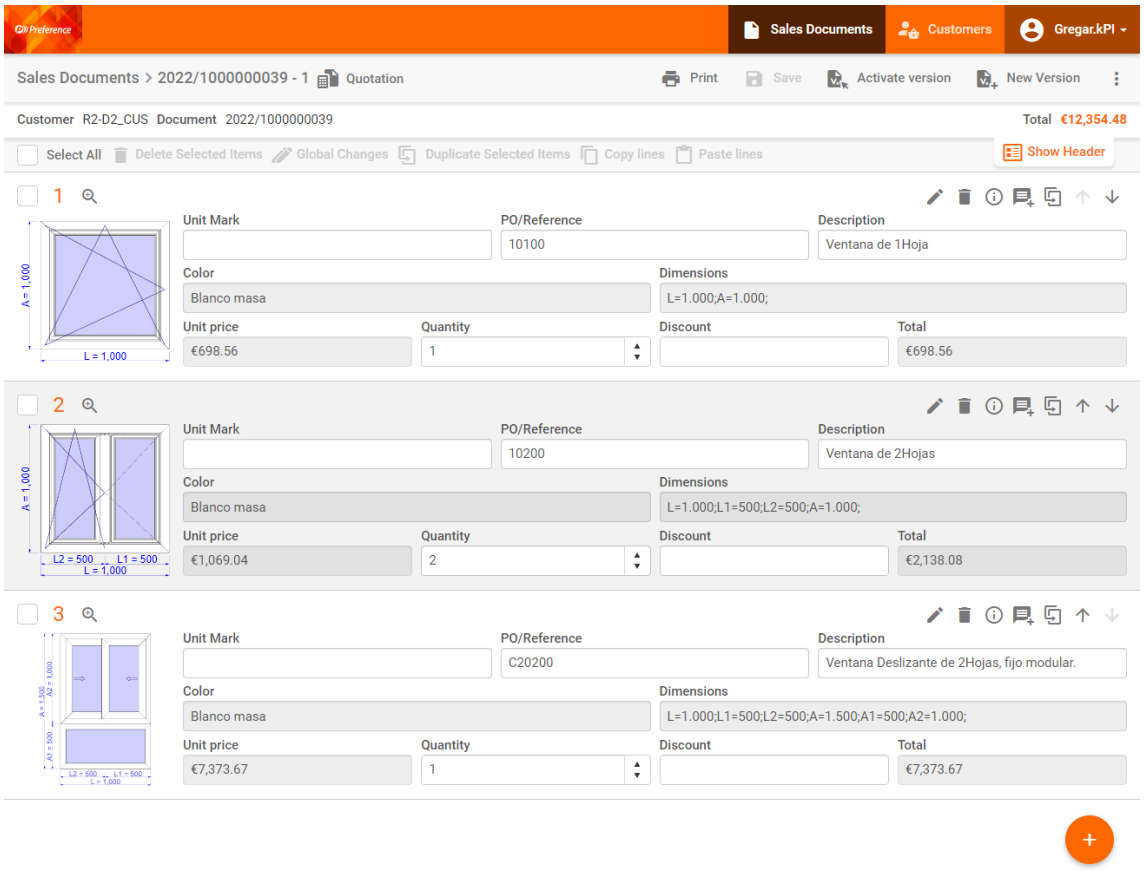



*Ilustración 15 - Modelos añadidos en un documento de ventas*

Las tres ventanas que se muestran en la Ilustración 15, se hacen llamar modelos, y los precios que aparecen son precios sin aplicar descuentos, incrementos o sobrecostes. Esta manipulación sobre el precio se puede añadir a través del menú ubicado arriba a la izquierda (característico por sus tres puntos en vertical).

Tras hacer una investigación sobre cómo crear estos modelos de la forma más rápida se dedujo lo siguiente:

1. Crear un *endpoint* para la API que permita crear documentos de ventas con modelos incluidos, ya que esta opción no se contempla en la API original.
2. Crear los modelos mediante un XML, lanzado por API también, pero que aumentaba la dificultad de este (además, había parámetros poco familiares y en una forma compleja de entender)
3. Crear los modelos vía interfaz de usuario tras haber creado el documento de ventas por API.

Finalmente, la solución empleada fue la segunda, puesto que la primera quedaba descartada debido a que los clientes tenían acceso a la API y no interesaba que estos tuviesen acceso a dicha funcionalidad, y la tercera también fue descartada por el tiempo empleado en generar los modelos.

Hay que entender, que los modelos que se generan son fieles a la realidad y llevan detrás una carga de datos importante. Esto se traduce en que emplean entre dos y cinco segundos en crearse (cada uno de ellos), sin tan siquiera contar el tiempo empleado al crearlos por interfaz.

Aclarado esto, los test implementados serán los siguientes:

Manipular un modelo

Para lograr manipular un modelo, es necesario crear todo lo descrito anteriormente. Una vez conseguido esto, basta con navegar hasta la página de manipulación de modelo y seguir los pasos hasta completar la manipulación.

En un fixture se almacenarán los datos del modelo con sus respectivos precios, para así comparar su “valorAntes" y su “valorDespués”. Si el incremento pasa de un 0% a un 15%, los precios de venta del modelo aumentarán un 15%. El coste de fabricación seguirá intacto.

Comprobando estos valores, junto con el valor del precio total de todos los modelos, se verificará si se ha manipulado correctamente o no el modelo.

Cambiar el idioma de la web al manipular un modelo

Este test se realiza por el mal funcionamiento reciente de la aplicación con los cambios de idioma y divisas en esta página.

Al igual que el test anterior de manipular el modelo, en este caso, mientras se manipula el modelo, se cambia el idioma y se guardan los cambios. La forma de cambiar el idioma se reutiliza de la forma de iniciar sesión directamente con un idioma. Durante el desarrollo del

método de inicio de sesión, se extrapoló la funcionalidad de cambiar el idioma y se llevó a una función, dentro del PageObject general.

Cambiar el idioma también supone que cambien los textos con los que se seleccionaban los elementos del DOM. Para solucionar el problema ocasionado, se ha creado un nuevo *fixture* para el idioma inglés, el cual tiene las mismas palabras que el español, solo que traducidas y adaptadas a PrefWeb.

Para comprobar que los cambios de idioma funcionan, simplemente bastará con buscar los *inputs* mediante su *label* con el idioma en inglés y verificar que el contenido de los inputs está en inglés. Esto puede ocasionar algo de confusión, ya que son números, pero realmente lo que se desea testear es que el número pase de "7.234,43 €" a "€7,234.43", ya que a los clientes estadounidenses les cuesta entender el formato usado en Europa.

Cambiar la divisa de la tarifa (de euro a dólar) al manipular un modelo

Al igual que el test anterior, se comprobarán los *inputs*, para que su valor pase de "7.234,43 €" a "7.234,43 $". En esta ocasión, no cambia el formato del número, sino el de la moneda. En caso de que se quisiese probar la divisa "dólar" en inglés, las comas y los puntos se invertirían y el símbolo del dólar pasaría a estar delante del número.

### 5.3.8 Panel de control

La página del panel de control permite establecer valores predeterminados en toda la aplicación. Para este test en concreto, únicamente se comprobarán las unidades de medida. Estas se cambiarán de métricas a imperiales. Muchas de las empresas cliente de Preference se encuentran en países donde se usan las pulgadas, mientras que otras muchas empresas usan los metros. Un fallo por la conversión de estas medidas podría ser catastrófico, así que se procederá a implementar un test que compruebe que dichos valores corresponden a su correcta conversión.

Para comprobar los cambios, basta con añadir en el panel de control la opción de pulgadas y, verificar que en ciertos puntos importantes de la aplicación como en la creación de modelos, en las entidades o en los documentos de ventas, la conversión es la adecuada.

La complicación de este test es que este valor se queda por defecto para los próximos test si no se hace nada, así que la solución a la que se ha llegado ha sido la de usar un afterEach, tal y como dice el artículo "*JavaScript Testing Best Practices*" (Goldberg, 2022) en su apartado 2.8, para así lograr que se limpie el estado.

## 5.4 Evidencias y conclusiones

En este apartado se pretenden mostrar todos los test ejecutados hasta la fecha con sus respectivos test, creados exclusivamente para este proyecto. Esto se logrará mediante la herramienta Cypress Dashboard, que hasta ahora no se ha mostrado su utilidad en el trabajo, pero sí que se han descrito sus funcionalidades.

Este Dashboard ha permitido que se almacenen los test ejecutados, con sus tiempos y con capturas de pantalla y capturas de vídeo sobre aquello que fallaba. A continuación, en la Ilustración 16, se mostrará cada una de las ejecuciones a modo de comparación entre ellas con respecto a la duración de ejecución.

*Ilustración 16 - Resumen del tiempo de ejecución de los test en Cypress Dashboard*

Aunque no se aprecien bien los valores de la imagen, podemos centrarnos en las barras horizontales de color verde y rojo. Aquello de color verde se traduce en que los test han pasado correctamente, sin ningún error. Sin embargo, aquello de color rojo se traduce en que los test sí contienen errores, aunque no es de extrema preocupación.

Estos test han salido de color rojo (han fallado) por nuevas implementaciones que cambian el funcionamiento real y, por tanto, fallan. Todas ellas excepto una, que realmente sí ha captado un error, al contrario que con los test de Ranorex, ya que sobre la misma versión de PrefWeb, estos no han fallado.

Tras inspeccionar el error, se ha deducido que se ha podido detectar mediante Testing-Library, ya que ha sido uno de sus métodos de “findByText” o “buscarPorTexto” la que ha ocasionado la detección del error. Esto confirma la importancia de guiarse por aquello que observa el usuario, y no por Ids o atributos no perceptibles para este, ya que, un texto distinto aporta una funcionalidad distinta a aquello que se desea realizar.

En la Ilustración 17, se muestran los tiempos de ejecución de todos los test, incluyendo los errores y, en la Ilustración 18, se muestran los tiempos de ejecución de todos los test de la antigua *suite* de pruebas de Ranorex.

(Run Finished)

| | Spec | | Tests | Passing | Failing | Pending | Skipped |
|---|---|---|---|---|---|---|---|
| √ | controlPanel.spec.js | 00:12 | 1 | 1 | - | - | - |
| √ | customers.spec.js | 00:27 | 2 | 2 | - | - | - |
| √ | entities.spec.js | 00:57 | 6 | 6 | - | - | - |
| √ | login.spec.js | 00:22 | 10 | 10 | - | - | - |
| √ | priceManipulation.spec.js | 03:04 | 5 | 5 | - | - | - |
| × | salesDocuments.spec.js | 03:19 | 63 | 61 | 2 | - | - |
| √ | selectors.spec.js | 01:06 | 8 | 8 | - | - | - |
| × | tariffs.spec.js | 02:55 | 13 | 9 | 4 | - | - |
| √ | users.spec.js | 00:36 | 4 | 4 | - | - | - |
| × | 2 of 9 failed (22%) | 13:03 | 112 | 106 | 6 | - | - |

*Ilustración 17 - Tabla con todos los tiempos de la suite de test. Cypress*

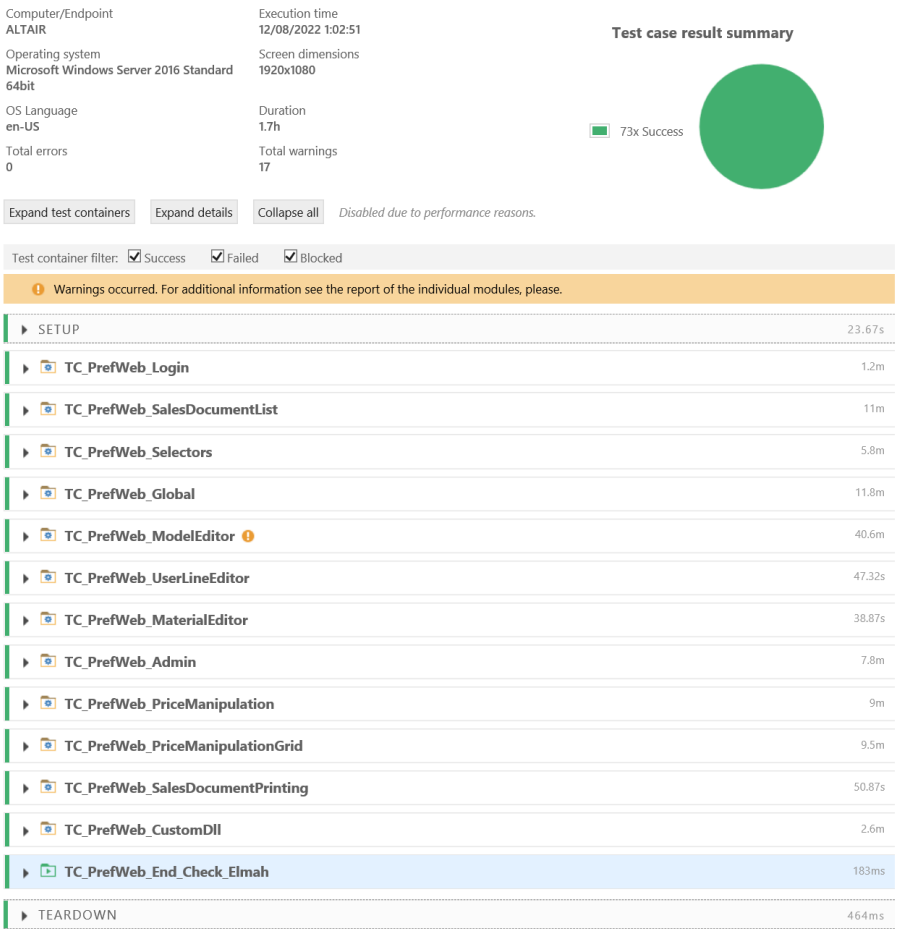



*Ilustración 18 - Tabla con todos los tiempos de la suite de test. Ranorex*

Los títulos de los test pueden variar, ya que se ha reestructurado la *suite* debido a que no seguía un orden y una estructura intuitiva. De esta forma, en la nueva *suite* de test, los archivos de testing (*.spec.js) se ordenan según la página que estén testeando. Dentro de estos archivos puede haber desde un test, en el caso de "controlPanel", hasta 63 test, en el caso de "salesDocuments".

Dado que ambas ilustraciones muestran bastante información y puede resultar confusa, en la Tabla 2, se muestra una comparativa real de los tiempos de ejecución pertenecientes a cada test.

| | **Ranorex** | **Cypress** |
|---|---|---|
| **ControlPanel** | 1 min | 12 s |
| **Customer** | 2 min | 27 s |
| **Entities** | 2 min 30 s | 57 s |
| **Login** | 1 min 15 s | 22 s |
| **PriceManipulation** | 20 min | 3 min 4 s |
| **SalesDocuments** | 20 min 30 s | 3 min 19 s |
| **Selectors** | 5 min 50 s | 1 min 6 s |
| **Tariffs** | 17 min | 2 min 55 s |
| **Users** | 4 min | 36 s |
| **Total** | **1 h 10 min 30 s** | **13 min 03 s** |

*Tabla 2 – Comparativa de los tiempos de ejecución de las diferentes suites*

Dicho esto, aunque la duración real de los test es de 1 h 42 min en el caso de Ranorex, y en la tabla aparecen 1 h 10 min 30 s, se debe a que ha habido un problema de implementación el test ModelEditor, haciendo que se retrase en medida su desarrollo. Por ello se estima que, siguiendo el mismo patrón de mejora de Cypress con respecto a Ranorex, el test ModelEditor pase de 31 min 30 s a, aproximadamente, 6 min.

Esta proporción se ha calculado teniendo en cuenta la mejora total de los tiempos. Conociendo que la duración de la ejecución total de los test de Ranorex es de 1 h 10 min 30 s, y la de Cypress es de 13 min 03 s, se calcula que la nueva *suite* es 5.31818 veces más rápida que la *suite* de Ranorex. Esto equivale a una reducción del 81% en los tiempos de ejecución.

A modo de conclusión, los resultados de la tabla verifican que, efectivamente, se han cumplido los objetivos de reducción de tiempos. Además, durante el desarrollo, se ha prestado especial atención a las buenas prácticas, a la paralelización y a los requisitos de la empresa.

# 6. Implantación

La solución del proyecto se ha desarrollado e implementado siempre dentro de la empresa Preference. Es por ello por lo que la explicación de la implantación se realizará en el contexto de la empresa.

La implantación se dividirá en los siguientes pasos:

1. Acceder al código de la aplicación PrefWeb. Para ello es necesario crear una rama a partir del proyecto de la aplicación PrefWeb con una versión estable. La solución del trabajo se encuentra dentro del proyecto de PrefWeb, es por eso por lo que es necesario tener acceso a dicho código.
2. Ejecutar Cypress. Una vez se ha creado una nueva rama, Cypress se puede ejecutar de las siguientes dos formas:
    a. Modo *headed*: en este caso, Cypress se ejecuta accediendo a un terminal (dentro de visual studio code, por ejemplo), y ejecutando la orden "npm run cy:open". Tras ejecutar la orden, se abre una ventana Cypress en la que se muestran todos los test posibles y se puede seleccionar aquel que se desee probar.
    b. Modo *headless*: en este caso, Cypress se ejecuta accediendo a un terminal y ejecutando la orden "npm run cy:run:record". El comando opcional "*record*" sirve para almacenar los datos en el Dashboard de Cypress para su posterior análisis. Tras la ejecución de este comando, instantáneamente se ejecutará toda la *suite* de test.
3. Comprobar los resultados. Para comprobar los resultados de los test, bastaría con observar la ventana Cypress (en el caso del modo *headed*) u observar el terminal (en el caso del modo *headless*). Si se ha utilizado el comando "*record*", la mejor opción es acceder a la aplicación web "Cypress Dashboard" y analizar los resultados desde esta herramienta.

# 7. Conclusiones

Las pruebas de *software* son las investigaciones empíricas y técnicas que pretenden verificar si el comportamiento del sistema es el deseado o no, y son cruciales en el desarrollo del código. Tal y como se ha comentado en la motivación del proyecto, desde el inicio había un gran interés en conseguir que las aplicaciones con las que se mantuviese contacto estuviesen bien estructuradas y testeadas, para así garantizar en medida de lo posible el correcto funcionamiento del *software*.

Todo esto ha determinado el enfoque final del proyecto, basado en conseguir que la empresa Preference S.L. obtuviese un *software* consistente y escalable, con una ejecución continua de pruebas para garantizar su correcto funcionamiento y así, evitar errores en producción.

Una vez analizado el problema, se plantearon unos objetivos acordados por la empresa que pretendían guiar este trabajo por un determinado camino. Los objetivos principales eran: conseguir una mejora de los tiempos de ejecución con respecto a la antigua *suite* de test ejecutada por el *software* Ranorex, preparar el código para que no fuese tan susceptible al cambio y desarrollar la solución empleando buenas prácticas para garantizar la futura escalabilidad de esta, además de garantizar un testeo adecuado de la aplicación.

Todos los objetivos establecidos al inicio del proyecto se han cumplido satisfactoriamente, aunque el camino no ha sido del todo sencillo. La implementación de la solución en ocasiones ha resultado costosa, ya sea por la dificultad de testear páginas que usan librerías externas poco accesibles, aunque funcionales (como Kendo), como por el tamaño y complejidad de la aplicación PrefWeb. A todo esto, se ha sumado la falta de experiencia en el testing, que obligaba a replantearse en reiteradas ocasiones cómo se estaba haciendo la implementación, y refactorizar el código, retrasando la entrega.

El desarrollo del proyecto ha derivado en la adquisición de conocimientos relativamente amplios en el mundo del testing, como en el uso de la herramienta Cypress, una profundización en el lenguaje JavaScript y C#, conocimientos de buenas prácticas en los test, etc., al igual que ha reforzado de manera drástica aquellos conceptos relacionados, adquiridos durante la etapa estudiantil (de gestión de proyectos, programación, patrones de diseño, etc.). Además, por primera vez, se han tomado decisiones de análisis, diseño, implementación y planificación para un proyecto empezado desde cero, sin el apoyo de unas transparencias guiadas o de un profesor siempre dispuesto a prestar su ayuda.

Finalmente, me gustaría recalcar la importancia del proyecto a nivel personal, y es que, el hecho de realizar un proyecto desde cero, sin saber la solución final y sin conocer el camino correcto que lleva directo a la solución, me ha proporcionado la suficiente confianza en mí mismo como para saber, que sea cual sea el proyecto o el problema, lo voy a poder implementar y terminar, aunque mis conocimientos no sean tan amplios.

## 7.1 Relación del trabajo desarrollado con los estudios cursados

El trabajo realizado está relacionado con los estudios cursados. En este, se han utilizado lenguajes presentes en la etapa estudiantil, como lo son JavaScript y C#, también se ha estado lidiando constantemente con conceptos de paralelización, patrones de diseño, metodologías, planes de trabajo, etc., que ya eran familiares debido a los estudios previamente cursados. Otras herramientas, sin embargo, no se han visto tanto, como por ejemplo metodologías ágiles de trabajo (Scrum), lenguajes como Ranorex y la implementación de test.

Cabe destacar que este trabajo ha sido un trabajo muy amplio, que a su vez ha empleado diferentes tecnologías. Esto ha derivado en un alto coste de aprendizaje y, no por una tecnología en concreto, sino por necesitar conocimientos relativamente amplios de todas aquellas que se utilizan en el proyecto para poder implementar aportando la mejor solución al problema.

Otro punto importante es que, aunque el tema del proyecto está relacionado con los estudios, durante esta etapa, no se ha llegado a entrar en detalle en el testing (seguramente debido a la gran extensión de la informática en relación con el poco tiempo empleado por los estudios). Durante estos estudios, en muchos puntos se han nombrado la importancia de los test en el *software* moderno, al igual que se han mostrado técnicas para elaborar test, como caja blanca o caja negra, pero nada relacionado con la implementación de pruebas End-To-End. Es por esto por lo que se ha necesitado en gran medida, un período de adaptación y aprendizaje sobre estas ramas de la informática.

# 8. Bibliografía

Adam Hughes, C. S. (junio de 2022). *Techtarget*. Obtenido de https://www.techtarget.com/searchdatamanagement/definition/SQL-Server

Bernalte, L. (23 de Marzo de 2022). *lucasbernalte*. Obtenido de https://lucasbernalte.com/blog/por-que-usar-testing-library-en-lugar-de-enzyme

Cypress.io. (27 de marzo de 2022). *Cypress*. Obtenido de https://docs.cypress.io/examples/examples/tutorials#Best-Practices

Cypress.io. (2022). *Cypress in a Nutshel*. Obtenido de https://www.youtube.com/watch?v=LcGHiFnBh3Y&t=1825s

Cypress.io. (Junio de 2022). *Cypress.io*. Obtenido de https://www.cypress.io/dashboard/

Dodds, K. C. (Junio de 2022). *Testing Library*. Obtenido de https://testing-library.com/

Goldberg, Y. (mayo de 2022). *github*. Obtenido de https://github.com/goldbergyoni/javascript-testing-best-practices

Hegde, G. (mayo de 2022). *browserstack*. Obtenido de https://www.browserstack.com/guide/cypress-vs-webdriverio

Hegde, G. (mayo de 2022). *browserstack*. Obtenido de https://www.browserstack.com/guide/testcafe-vs-cypress

Microsoft Corporation. (junio de 2022). *Microsoft*. Obtenido de https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

Microsoft Corporation. (junio de 2022). *Microsoft*. Obtenido de https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-6.0

Monroy, A. (mayo de 2022). *encora*. Obtenido de https://www.encora.com/es/blog/ya-utilizo-selenium-por-que-deberia-aprender-cypress

Mozilla Corporation. (junio de 2022). *Mozilla*. Obtenido de https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript

Mozilla Corporation. (junio de 2022). *Mozilla*. Obtenido de https://developer.mozilla.org/es/docs/Glossary/HTML

Open Source. (Junio de 2022). *chaijs*. Obtenido de https://www.chaijs.com/

OpenJs Foundation. (Junio de 2022). *Mocha*. Obtenido de https://mochajs.org/

Pou, I. R. (junio de 2022). *carontestudio*. Obtenido de https://carontestudio.com/blog/que-es-el-inspector-de-elementos-y-como-se-utiliza/#:~:text=El%20inspector%20de%20elementos%20de%20Google%20Chrome%20es%20una%20herramienta,la%20web%20de%20forma%20interactiva.

Tej, K. (mayo de 2022). *browserstack*. Obtenido de https://www.browserstack.com/guide/playwright-vs-cypress

Tozzi, C. (junio de 2022). *saucelabs*. Obtenido de https://saucelabs.com/blog/top-5-javascript-test-automation-frameworks-in-2021

Vaidya, N. (mayo de 2022). *browserstack*. Obtenido de https://www.browserstack.com/guide/puppeteer-vs-selenium

# ANEXO

## 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** | |

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Este trabajo, no está altamente relacionado con ningún ODS. Aun así, aquellos objetivos que pueden tener alguna relación con el proyecto realizado podrían ser los siguientes:

- Trabajo decente y crecimiento económico
- Alianzas para lograr objetivos
- Igualdad de género

Estos objetivos, bajo mi punto de vista, son bastante importantes en la sociedad actual.
En primer lugar, aunque no se nombre la igualdad de género en el trabajo, este ha sido desarrollado bajo un ámbito saludable, en el que no se ha negado ninguna oportunidad a nadie dependiendo de su género, creyendo en la igualdad entre personas.
En segundo lugar, el proyecto promueve el desarrollo del trabajo decente y el crecimiento económico, promoviendo el uso de una rama de la informática que no es tan amplia, pero que aun así es de lo más necesaria e importante. Esto hace que se esté promoviendo una rama aún por desarrollar, que no tiene el suficiente respaldo.
En tercer lugar, durante el desarrollo del trabajo, se han establecido alianzas para lograr los objetivos comunes, ya sea mediante el trabajo en equipo o mediante consultas a otras partes.

Considero pues, que, aunque no se hayan abarcado demasiados objetivos en este proyecto, estos son imprescindibles en la sociedad actual