Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de herramientas para mejorar el mantenimiento y análisis de resultados de pruebas automatizadas usando Protractor

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: David Olmos Garrido

Tutor: Patricio Orlando Letelier Torres

2019-2020

Resumen

En este trabajo se explora la necesidad de la mejora en las herramientas de apoyo a pruebas de sistema basadas en Protractor para aplicaciones web. Para ello, lleva a cabo un desarrollo para mejorar la trazabilidad y facilitar la aplicación de los principios de desarrollo SOLID y DRY con la creación de una herramienta específica y un framework ad-hoc. Todo esto, combinado, pretende mejorar la mantenibilidad de las suites de pruebas basadas en Protractor.

Palabras clave: Protractor, pruebas automatizadas, aplicaciones web, mantenimiento, SOLID, DRY, herramientas.

Abstract

This paper explores the need of better test support tools based on Protractor for web applications. It performs a development to enhance the tests traceability and to ease the application of the DRY and SOLID development principles with the creation of a specific tool and an ad-hoc framework. All of this, together, aims to enhance the maintainability of Protractor-based testing suites.

Keywords: Protractor, automated tests, web applications, maintenance, SOLID, DRY, tools.

Tabla de contenidos

Tabla de contenido

Capítulo 1. Introducción 1
1.1. Motivación 2
1.2. Objetivos 2
1.3. Impacto esperado en los desarrolladores de las pruebas 2
1.4. Estructura del trabajo 3
Capítulo 2. Estado del arte 5
2.1. Contexto del SUT 5
2.2. Tipos de pruebas 5
2.2.1. Según el conocimiento sobre el código del SUT 5
2.2.1.1. Caja blanca 5
2.2.1.2. Caja negra 5
2.2.2. Según la granularidad de la prueba 6
2.2.2.1. Pruebas unitarias 6
2.2.2.2. Pruebas del sistema 6
2.2.2.3. Pruebas de integración 6
2.2.3. Según el objetivo de la prueba 7
2.2.3.1. Pruebas de validación o funcionales 7
2.2.3.2. Pruebas de aceptación 7
2.2.3.3. Pruebas de regresión 7
2.3. Contextualización de la suite existente 8
2.4. Herramientas de automatización de pruebas 8
2.4.1. Herramientas para probar código 8
2.4.2. Herramientas para probar API HTTP 8
2.4.3. Herramientas para probar GUI 9
2.4.3.1. Herramientas de 2ª generación 9
2.4.3.1.1. Selenium Web Driver 9
2.4.3.1.2. Protractor 9
2.4.3.1.3. SerenityJS 10

2.4.3.1.4. Cypress.io 10
2.4.3.2. Herramientas de 3ª generación 10
2.4.3.2.1. SikuliX 10
2.4.3.2.2. EyeAutomate 10
2.5. Principios de desarrollo de software mantenible 11
2.5.1. SOLID 11
2.5.1.1. Single Responsibility Principle 11
2.5.1.2. Open/Closed Principle 11
2.5.1.3. Liskov Substitution Principle 11
2.5.1.4. Interface Segregation Principle 11
2.5.1.5. Dependency Inversion Principle 11
2.5.2. DRY 12
2.6. Crítica al estado del arte 12
2.7. Propuesta 12
Capítulo 3. Análisis 13
3.1. Requisitos 13
3.1.1. Específicos para pruebas GUI de sistema 13
3.1.1.1. Ejecución determinista 13
3.1.1.2. Trazabilidad de fallos 13
3.1.1.3. Nuevas abstracciones 13
3.1.1.4. Persistencia de los resultados 13
3.1.2. Específicos del SUT 13
3.1.2.1. Pruebas fácilmente parametrizables 14
3.2. Restricciones técnicas 14
3.2.1. Identificación del fin de la carga de la web del SUT 14
3.2.2. Difícil automatización del despliegue del SUT 14
3.3. Soluciones propuestas 14
3.3.1. Respecto a la ejecución determinista 14
3.3.2. Respecto a la trazabilidad 14
3.3.3. Respecto a la persistencia de resultados 15
3.3.4. Respecto a las nuevas abstracciones. Aplicación de los principios SOLID y DRY 15
3.3.5. Respecto a la parametrización de las pruebas 16
3.4. Marco legal 16
3.4.1. Análisis de la protección de datos 16
3.4.2. Propiedad intelectual 16

Capítulo 4. Diseño de la aplicación 17
4.1. Punto de partida 17
4.2. Arquitectura de la aplicación 17
4.2.1. Arquitectura inicial 17
4.2.2. Arquitectura final 18
4.2.2.1. Base de datos 19
4.2.2.2. Servidor 19
4.2.2.3. Cliente web 19
4.3. Diseño de la UI de la aplicación de apoyo 22
4.3.1. Lista de ejecuciones 23
4.3.2. Registro de llamadas HTTP 23
4.4. Nueva funcionalidad del *plugin*. Grabación de pantalla. 24
4.5. Tecnología utilizada 24
4.5.1. TypeScript 24
4.5.2. JavaScript 25
4.5.3. AngularJS 25
4.5.4. NodeJS 25
4.5.5. ESLint 25
4.5.6. NestJS 25
4.5.7. MongoDB 26
4.5.8. FFmpeg 26
4.5.9. Git 26
Capítulo 5. Desarrollo de la aplicación 27
5.1. Refactorización de la aplicación inicial 27
5.2. Separación entre *frontend* y *backend.* Integración con una base de datos.. 27
5.2.1. Implementando el servidor 27
5.2.2. Modificando el *plugin* 27
5.2.3. Modificando el cliente web 28
5.3. Funcionalidad con cambios en el *frontend* 28
5.3.1. Lista de ejecuciones realizadas 28
5.3.2. Trazabilidad mediante el registro de llamadas HTTP durante la prueba 29
5.4. Grabación de la ejecución de las pruebas 29
Capítulo 6. SOLID y DRY con un *framework* adhoc 31
6.1. Patrón *Page object* vs *Componente*. DRY 31
6.1.1. Page object 31
6.1.2. Los problemas con *Page object* 32

6.1.3. Componente 32
6.2. Diseño del *framework* 34
6.2.1. Componente 34
6.2.2. Encapsulación de *Requisito* y *CasoDePrueba* 34
6.2.3. Un DSL para la composición y parametrización de flujos de usuario y su integración con el *framework* 35
6.2.3.1. Parametrización compleja usando el patrón Builder 35
6.2.4. Autoconfiguración mediante un bus de eventos 36
Capítulo 7. Conclusiones 39
7.1. Conclusiones 39
7.2. Problemas encontrados 39
7.3. Errores cometidos 39
7.4. Conocimiento adquirido 39
7.5. Relación con los estudios cursados 40
Capítulo 8. Trabajos futuros 41
8.1. Desarrollo de funcionalidad deseable en la aplicación 41
8.1.1. Marcar falsos positivos 41
8.1.2. Gráficas y estadísticas que exploten mejor los datos recogidos 41
8.1.3. Reproductor de las pruebas GUI en video 41
8.1.4. Usar SQL en lugar de NoSQL 41
8.1.5. Mejorar la integración con otros *framework* de pruebas 41
8.2. Mejoras en el *framework* ad-hoc 41
8.2.1. Mejorar la grabación de pruebas dentro de Docker 41
8.2.2. Integración con otros *frameworks* 42
Capítulo 9. Referencias 43
Capítulo 10. Anexo 45
10.1. Código de ejemplo 45

Tabla de figuras

Figura 1: Vista del reporte creado con el plugin inicial. Fuente (7) 18
Figura 2: Arquitectura base de datos, servidor y cliente web. 18
Figura 3: Entidades en base de datos 19
Figura 4: Arquitectura del servidor de la aplicación 19
Figura 5: Diagrama Model-View-Controller. Fuente (8) 20
Figura 6: La fila de la prueba 20
Figura 7: Los datos de la suite 21
Figura 8: La lista de suites 21
Figura 9: El filtro de búsqueda de las suites 21
Figura 10: Visor del log del navegador y de las trazas de error 21
Figura 11: Vista de capturas de pantalla 22
Figura 12: Barra de recuento de pruebas pasadas y falladas 22
Figura 13: Componente de detalles de la ejecución 22
Figura 14: Lista de ejecuciones 23
Figura 15: Registro de llamadas HTTP 24
Figura 16: Modificación de la pantalla original de detalles de la ejecución 24
Figura 17: Demo de una prueba Protractor. Fuente (11) 31
Figura 18: Ejemplo de Page object para la página de AngularJS. Fuente (11) 32
Figura 19: Uso del Page object de ejemplo. Fuente (11) 32
Figura 20: Ejemplo de un Componente con las interacciones de un calendario 33
Figura 21: Ejemplo de prueba que usa el Componente 33
Figura 22: Caso de prueba de reservar mesa implementando la abstracción CasoDePrueba 34
Figura 23: Mismo CasoDePrueba definido con diferentes argumentos 34
Figura 24: Implementación del requisito de tener un usuario creado 35
Figura 25: Forma normal de encadenar un Requisito con un CasoDePrueba 35
Figura 26: Definición del caso de prueba de ReservarMesa y sus requisitos mediante el DSL 35
Figura 27: Ejemplo de un Builder para el caso de prueba ReservaMesa 36
Figura 28: Interfaces de configuración 37

Glosario

Backend: Parte de un sistema software que se encarga de la lógica de negocio y del modelo de datos.

Clase: En un contexto de lenguajes de programación, es una plantilla que define el comportamiento y las propiedades de una estructura lógica. *Véase* Objeto.

DSL: *Domain Specific Language* o Lenguaje de Dominio Especifico. Es un lenguaje gráfico o textual creado con el objetivo de representar estructuras pertenecientes a un dominio más específico.

Framework: Paquete de código con el objetivo de ser usado para facilitar uno o varios desarrollos.

Frontend: Parte de un sistema software que tiene el objetivo de pintar información en su GUI.

GUI: *Graphical User Interface* o Interfaz Gráfica de Usuario. Se refiere a la parte visible de una aplicación software por parte del usuario.

HIS: *Hospital Information System* o Sistema de Información Hospitalario.

Interfaz: En un contexto de lenguajes de programación, es una abstracción que define un contrato de las propiedades y/o comportamientos que debe implementar una clase.

Objeto: En un contexto de lenguajes de programación, es la instancia de una clase.

Plugin: Software externo a un *framework* que se crea con la finalidad de ampliar su funcionalidad.

Suite: Conjunto de pruebas de software. Una *suite* puede contener otras *suites.*

SUT: *System Under Test* o Sistema Bajo Prueba. Hace referencia a la aplicación que se está probando. En nuestro contexto es la aplicación HIS.

UI: *User Interface* o interfaz de usuario. *Véase* GUI.

Capítulo 1. Introducción

Hoy en día existen dos visiones acerca de desarrollar software opuestas pero complementarias, a saber, como una actividad artesanal o como un proceso de ingeniería. Esto se debe, posiblemente, a que el desarrollo de software como actividad productiva es muy nuevo en comparación con otros procesos productivos.

En las últimas décadas ha habido un esfuerzo enorme tanto por parte de la academia como de la industria por automatizar la mayor parte del trabajo que involucra el ciclo de desarrollo de un producto software, desde la extracción de los requisitos hasta las tareas de mantenimiento. Para ello, han ido apareciendo nuevas herramientas, metodologías de desarrollo aplicables a todo el ciclo de vida del software, desde la extracción de requisitos hasta el mantenimiento del producto pasando por las pruebas de verificación y validación.

Desde que el software se introdujo en procesos críticos como podría ser el control de un cohete o una máquina de radiología, existe la necesidad de probar a fondo los desarrollos con el fin de evitar imprevistos con consecuencias posiblemente fatales. Aun así, la actividad de pruebas es, a menudo, lo primero que se reduce en un proyecto cuando la fecha de entrega se echa encima, por lo que la calidad del producto final se puede ver seriamente comprometida. En contraposición a esta manera de pensar, nace de las metodologías ágiles la idea de centrar la fase de desarrollo en torno a la automatización de las pruebas, el *Test Driven Development* o TDD. Para apoyar esta forma de producir software han sido desarrolladas muchas tecnologías y herramientas orientadas a todas las partes que componen una aplicación software.

Además, a la hora de mantener un software no es igual de eficiente modificar un código con una suite de pruebas automatizadas que uno que solo se prueba a mano.

En un programa medianamente complejo es sencillo que lo que se modifica en una parte del programa impacte indirectamente en otra parte de este de manera imprevista con consecuencias inesperadas. Sabiendo esto no podemos conformarnos con probar tan solo en lo que impacta directamente el cambio, sino que tenemos que volver a probar el programa entero. Esto en un programa medio es una tarea de dimensiones considerables si ha de realizarse por un equipo de personas, por eso es normal dejar estas pruebas para antes del lanzamiento del producto, o de una versión de este. El problema con esto es que para cuando se realiza la prueba y se detecta un posible error, el desarrollador ha cambiado de contexto de trabajo produciendo en él una sobrecarga mental que no tendría si el error se hubiese detectado cuando aún estaba en ese contexto de desarrollo. Por ello, para captar el mayor número de errores nuevos durante el desarrollo de nueva funcionalidad que afecte a la antigua o de correctivos/modificaciones, se pueden implementar pruebas automatizadas con las que el desarrollador puede comprobar con cierto grado de confianza que no está rompiendo funcionalidad previamente existente.

Además, las pruebas automatizadas son mucho más deterministas que las pruebas manuales. Aunque en las pruebas manuales es más sencillo detectar problemas inesperados, son poco eficientes y nada deterministas, es decir, las personas suelen

cometer errores en tareas repetitivas y rara vez hacen un mismo proceso exactamente de la misma manera. Esto puede llevar a que hoy encuentre un error y mañana, ejecutando la misma prueba, no lo vea porque no se ha fijado bien mientras probaba. En cambio, un ordenador una vez programado es muy bueno repitiendo siempre las mismas acciones.

1.1. Motivación

Durante la etapa de mantenimiento de un producto software con una suite de pruebas automatizadas es normal que las propias pruebas deban ser ejecutadas y modificadas con cada correctivo llevado a cabo.

Aunque la automatización de las pruebas sea una manera de mejorar la mantenibilidad del software legado, las propias pruebas automatizadas no dejan de ser también software y por tanto pueden ser más o menos fáciles de mantener.

Si bien es cierto que muchas prácticas usadas en el desarrollo de software para mejorar la mantenibilidad de un producto son aplicables, la finalidad del código de las pruebas hace necesaria también otro tipo de herramientas que reciben menos atención por parte de la comunidad de desarrolladores en comparación a las herramientas de desarrollo genéricas.

Existe un desarrollo considerable en las herramientas de apoyo a las pruebas automatizadas que tienen como objetivo probar el código, pero hay evidentes carencias en las herramientas de apoyo a pruebas automatizadas que tienen como objetivo probar GUI, sobre todo sobre aplicaciones web que en estos últimos 10 años han crecido considerablemente en complejidad.

La confección de dicho trabajo se ha realizado en un contexto de prácticas de empresa desarrollando y manteniendo un Sistema de Información Hospitalario (HIS en inglés) que dispone de una limitada suite de pruebas automatizadas mediante el uso de la herramienta para pruebas web Protractor[1].

1.2. Objetivos

En este trabajo se pretende mejorar la mantenibilidad de la suite de pruebas de una aplicación web hecha con AngularJS[2].

Para ello, se tratará de aplicar estrategias para organizar el código usadas comúnmente en aplicaciones, pero en el código de las pruebas, y se analizará si en ellas tienen sentido los principios SOLID y DRY.

Además, se analizarán los requisitos de una posible aplicación que ayude al análisis de los resultados de las pruebas para facilitar la distinción entre un falso positivo que requeriría la modificación de la prueba y un error en la aplicación probada. Se propondrán soluciones a esos requisitos, se definirá el alcance del desarrollo de la aplicación, se diseñará y se implementará.

1.3. Impacto esperado en los desarrolladores de las pruebas

[1] Protractor - https://www.protractortest.org/#/
[2] AngularJS - https://angularjs.org/

Lo tratado en este trabajo afecta principalmente a los desarrolladores de las pruebas automatizadas, tanto a quien las desarrolle inicialmente como a quien las mantenga.

Quien las desarrolle inicialmente debe tener en cuenta las estrategias propuestas e implementadas en la suite para que proliferen dentro del código de las pruebas y quien las mantenga se debería ver beneficiado de ellas y tendría la obligación de seguir cumpliéndolas.

También es probable que afecte al equipo de sistemas/devops/despliegues pues la aplicación a desarrollar podría centralizarse en un servidor común del que sacar estadísticas y explotar los datos de la aplicación de apoyo a las pruebas.

1.4. Estructura del trabajo

En el capítulo 2 – *Estado del arte,* se describe el *System Under Test* o SUT para el cual está desarrollada nuestra suite de pruebas y se da un repaso a la teoría de pruebas de software para tratar de enmarcarla dentro de ella. Además, trata de sondear las herramientas de desarrollo de pruebas automatizadas y como tratan de abordar la mantenibilidad de éstas.

En el capítulo 3 – *Análisis,* se introducen los principios SOLID y DRY y se proponen soluciones para hacer más fácil su aplicación en el código de pruebas, se extraen los requisitos de la aplicación de apoyo a las pruebas, se explican las restricciones debidas al SUT y propuestas que traten de cumplir con los requisitos identificados a la vez que tienen en cuenta las restricciones.

En el capítulo 4 – *Diseño de la aplicación,* se concretan soluciones a las propuestas menos relacionadas con el código de las pruebas mediante el diseño de una aplicación de apoyo a la *suite* de pruebas y se describe la tecnología usada para conseguirlo.

En el capítulo 5 – *Desarrollo de la aplicación,* se pretende narrar los pasos seguidos para completar el desarrollo de la aplicación de apoyo a las pruebas y la necesaria refactorización del *plugin* de integración con Protractor.

En el capítulo 6 – *SOLID y DRY mediante un framework ad-hoc*, se detalla el diseño de un *framework* creado con la finalidad de reutilizar todo el código posible y facilitar la aplicación de los principios SOLID en el desarrollo de pruebas automáticas.

En el capítulo 7 – *Conclusiones,* se repasan los objetivos propuestos en el capítulo 1: se comprueba si se han cumplido, no se han cumplido o se han cumplido parcialmente. Además, se reflexiona sobre los problemas encontrados en la realización del trabajo, sobre el conocimiento adquirido y se relaciona este trabajo con lo aprendido durante el grado en Ingeniería Informática.

En el capítulo 8 – *Trabajos futuros,* se proponen mejoras y nuevos desarrollos que continúen con lo implementado durante este trabajo.

Capítulo 2. Estado del arte

2.1. Contexto del SUT

El HIS es una aplicación web con una estructura cliente-servidor típica dividida en dos partes donde la aplicación servidor o *backend* es un servicio web hecho en Java que accede a una base de datos SQL Server e intercambia datos mediante el protocolo HTTP/1.1 con la aplicación cliente o *frontend* que es una *Single Page Application o* SPA hecha con Angular[3] y AngularJS.

2.2. Tipos de pruebas

Antes de entrar en materia, conviene dar un repaso a grandes rasgos sobre qué tipos de pruebas se pueden realizar sobre los sistemas software y tratar de localizar nuestra *suite* de pruebas dentro de este marco.

2.2.1. Según el conocimiento sobre el código del SUT

A la hora de abordar el diseño de las pruebas de un sistema hemos de plantearnos primero el conocimiento que tenemos de este. En función de si conocemos la implementación de un sistema a fondo o si solo tenemos disponible la especificación de su interfaz, podemos dividir los tipos de pruebas en dos.

2.2.1.1. Caja blanca

“Estas pruebas se centran en probar el comportamiento interno y la estructura del programa examinando la lógica interna “(1)

Por las características de este tipo de pruebas, el código de la aplicación software a probar debe estar disponible.

Para el diseño de estas pruebas se realiza el grafo del flujo de control o CFG con el fin de analizar todos los caminos posibles que puede atravesar el código dependiendo de sus entradas. La finalidad de este tipo de pruebas es comprobar la ejecución de cada una de las sentencias del programa, recorrer todos los caminos independientes de su ejecución, comprobar la corrección de todas las decisiones lógicas y la ejecución de todos los bucles del flujo. (1)

2.2.1.2. Caja negra

“Las pruebas de caja negra están conducidas por los datos de entrada y salida. Así, los datos de entrada deben generar una salida en concordancia con las especificaciones considerando el software como una caja negra sin tener en cuenta los detalles procedimentales de los programas“(1)

Aunque en (1) definen este tipo de pruebas dentro de un contexto de programación y se proponen técnicas de diseño orientadas a probar funciones o campos concretos, la definición sirve en muchos otros contextos de pruebas como puede ser a la hora de probar GUI. Se puede considerar la GUI y el sistema que hay detrás como una caja negra en la que se introducen datos, que pueden ser desde cadenas de texto hasta clics

[3] Angular - https://angular.io/

del ratón, y la GUI devuelve una imagen o vista que debe coincidir con lo que pone en las especificaciones o requisitos del sistema.

A la hora de diseñar casos de prueba para este tipo de sistemas, como no se puede probar todos y cada uno de los valores de entrada, conviene elegir casos representativos de esos valores en función de la especificación de estos para tratar de cubrir el mayor número de estados finales del sistema.

2.2.2. Según la granularidad de la prueba

Cuando revisamos los tipos de pruebas de software en base a la granularidad del sistema probado, podemos clasificarlas en tres tipos de pruebas.

2.2.2.1. Pruebas unitarias

Son aquellas de granularidad más fina.

“Las pruebas unitarias se corresponden con la prueba de cada uno de los módulos o clases del programa de forma independiente y es realizada por el programador en su entorno de trabajo. En definitiva, consiste en probar los bloques más pequeños con identidad propia presentes dentro del programa. De esta forma, si una prueba descubre un nuevo error, este está más localizado. “(1)

Por sus características, este tipo de pruebas se realizan sobre el código de la aplicación. Si están bien diseñadas y desarrolladas, son rápidas y se pueden combinar diferentes pruebas para probar varios módulos a la vez.

Son, posiblemente, las pruebas que producen un *feedback* más inmediato a los desarrolladores puesto que suelen ser rápidas de ejecutar y localizadas en módulos puntuales de la aplicación.

2.2.2.2. Pruebas del sistema

Son aquellas de granularidad más gruesa.

“Es aquí donde se prueba el sistema integrado en su entorno hardware y software para verificar que se cumplen los requisitos especificados. “(1)

Es decir, son pruebas que tratan de probar todo el sistema en conjunto y verificar la corrección de los requisitos implementados en nuestra aplicación una vez puesto en marcha.

2.2.2.3. Pruebas de integración

La granularidad de estas pruebas se sitúa entre las pruebas unitarias y las de sistema.

“Consiste en integrar los módulos o clases, ya probados de forma independiente en las pruebas unitarias centrándose en probar sus interfaces. Habitualmente se utiliza el enfoque de caja negra.“(1)

El objetivo de estas pruebas es comprobar que los diferentes módulos o piezas que componen la aplicación interactúan correctamente entre sí.

Otra vez, en (1), parecen centrar la definición de interfaces en aquellas internas del código como puede ser la definición de una función, pero existen otras interfaces

externas al código que pueden ser probadas siendo un ejemplo evidente la interfaz entre un *backend* y un *frontend*. Esta interfaz puede ser una estructura JSON enviada por HTTP por el servidor, es decir, la prueba de integración de esa interfaz comprobaría mediante un enfoque de caja negra que el servidor devuelve unos datos en JSON correctos en función de los parámetros enviados en la petición.

2.2.3. Según el objetivo de la prueba

Existen diversos motivos por los que queremos ejecutar pruebas, ya sean manuales o automáticas, a nuestro software.

2.2.3.1. Pruebas de validación o funcionales

“El objetivo de las pruebas de validación o pruebas funcionales es comprobar que se cumplen los requisitos del usuario. Es por tanto una fase en la que interviene tanto el usuario como el desarrollador y se realiza en el entorno de desarrollo. Los criterios de validación que se utilizan son aquellos que se acordaron al inicio del proyecto en la fase de definición de requisitos y que deben constar en el correspondiente documento de especificación de requisitos. Dado que se está probando el sistema completo y que lo relevante es la salida producida, la técnica que se utiliza es la de caja negra, comprobando los resultados obtenidos con los resultados esperados. “(1)

Como se realiza en el entorno de desarrollo y no es necesario que las realice el usuario, se puede acordar el diseño de la prueba con el usuario y que este esté presente durante su ejecución sin que llegue a usar él mismo el software. Esto permite que sea posible automatizar la prueba al mismo tiempo que se graba su ejecución.

2.2.3.2. Pruebas de aceptación

“Es el último paso antes de la entrega formal del software al cliente y se realiza, normalmente, en el entorno del usuario. Consiste en la aceptación por parte del cliente del software desarrollado. El cliente comprueba que el sistema está listo para su uso operativo y que satisface sus expectativas. Habitualmente es el usuario quien aporta los casos de prueba. “(1)

A diferencia de las pruebas de validación o funcionales, las pruebas de aceptación, por su naturaleza, han de ser ejecutadas manualmente por el usuario.

2.2.3.3. Pruebas de regresión

“Cuando se realiza un cambio en el software, es necesario volver a probar comportamientos y aspectos ya probados de forma satisfactoria anteriormente y volver a utilizar algunos del conjunto de casos de prueba diseñados para asegurar que el cambio efectuado en el software no ha interferido en el correcto comportamiento del sistema software y no se han producido efectos colaterales. Este tipo de pruebas son, por tanto, muy comunes durante la fase de mantenimiento de un sistema software. “(1)

En resumen, las pruebas de regresión son aquellas diseñadas con la finalidad de detectar el efecto ola producido por las actividades de mantenimiento y constatar que una funcionalidad desarrollada o corregida no falla al llevar a cabo un correctivo en el software. Por ejemplo, si corregimos un error o defecto del software y queremos asegurarnos de que en versiones posteriores del software no vuelve aparecer, podemos diseñar una prueba de regresión que lo verifique.

2.3. Contextualización de la suite existente

Así pues, podemos posicionar nuestra *suite* de pruebas existente dentro de esta clasificación.

Siendo una *suite* de pruebas que se ejecuta sobre un navegador, podemos concluir que no tienen consciencia del comportamiento interno de la aplicación y por tanto son pruebas de caja negra.

La suite ha sido diseñada para probar el sistema en su totalidad y, por tanto, según la granularidad de las pruebas las podemos clasificar como pruebas del sistema.

En un principio, el objetivo de la *suite* usada era servir como pruebas de validación puesto que están basadas en el plan de pruebas funcionales del producto, pero pasado algún tiempo y ante la falta de inversión en aumentar los casos de prueba se empezaron a usar como pruebas de regresión.

2.4. Herramientas de automatización de pruebas

La informática tiene como objetivo primordial la automatización de todos los procesos que realizan los seres humanos, desde la escritura e impresión de este trabajo hasta la gestión de la información de millones de personas. Las pruebas de software no son una excepción y por ello han surgido muchas herramientas que tratan de automatizar todos los tipos de pruebas posibles tanto a nivel de código como a nivel de sistema.

A continuación se ha categorizado en tres tipos el resultado de la búsqueda en Google de las herramientas más completas y populares o mejor posicionadas.

2.4.1. Herramientas para probar código

Son aquellas destinadas a ejecutarse sobre el código fuente de la aplicación. Estas herramientas habilitan la realización de pruebas tanto de caja negra como de caja blanca. Su objetivo consiste en facilitar la automatización de pruebas unitarias y de integración sobre módulos e interfaces internas del código fuente.

En este apartado, cada lenguaje de programación tiene herramientas específicas de su paradigma. Algunas de estas herramientas son, por ejemplo, Junit[4] y TestNG[5] para Java o VSTest[6] y NUnit[7] para .NET.

2.4.2. Herramientas para probar API HTTP

Desde el lanzamiento de la primera especificación Swagger[8], hoy renombrada como especificación OpenAPI y dentro de la *OpenAPI Initiative*[9], han aparecido varias herramientas para la automatización de pruebas de integración de caja negra para interfaces HTTP.

[4] JUnit 5 - https://junit.org/junit5/
[5] TestNG - https://testng.org/doc/
[6] VSTest - https://github.com/microsoft/vstest
[7] NUnit - https://nunit.org/
[8] Swagger - https://swagger.io/
[9] OpenAPI Initiative - https://www.openapis.org/

Tanto SoapUI[10] como Postman[11] son herramientas con este objetivo.

2.4.3. Herramientas para probar GUI

El mundo de las GUI está en rápida y constante evolución. Por ello, las herramientas para este tipo de pruebas tienen poco tiempo para madurar antes de quedarse desactualizadas. En esta sección se comentarán algunas herramientas nacidas o popularizadas en la última década.

Esta clase de pruebas normalmente son de sistema, pero según la herramienta es posible falsear parte del sistema por lo que si eso ocurre se podrían considerar pruebas de integración. Este tipo de pruebas se diseñan como prueba de caja negra porque el código, con toda seguridad, no está disponible.

Dentro de las herramientas actuales de automatización de pruebas en GUI podemos diferenciar tres generaciones aunque excluiremos la primera por estar demasiado obsoleta.(2)

2.4.3.1. Herramientas de 2ª generación

Son aquellas herramientas que localizan los componentes de la GUI mediante las propiedades de éstos. Por ejemplo, en una GUI definida en el lenguaje de marcado HTML se podría encontrar el botón sobre el que se quiere clicar buscando por el nombre de la etiqueta *button* o por algún otro atributo de esta como las clases CSS del botón en cuestión.

En esta generación podemos encontrar muchas herramientas, aunque algunas están basadas en la que detallamos a continuación.

2.4.3.1.1. Selenium Web Driver

Selenium, nacida en 2004 y mantenida hasta ahora, es una herramienta cuya función es automatizar las posibles interacciones de los usuarios con el navegador. Aunque su principal objetivo es probar aplicaciones web, también se puede destinar a otros objetivos que requieran de la automatización de interacciones con un navegador como puede ser la creación de demos técnicas de aplicaciones web o usarlo para crear *web scrappers*.

Es una herramienta políglota por lo que está disponible para varios lenguajes de programación como son Ruby, JavaScript[12], Java, Python y C#.

2.4.3.1.2. Protractor

Un proyecto nacido para mejorar la interacción de pruebas de aplicaciones web hechas con los frameworks AngularJS y Angular de Google. Usa JavaScript como lenguaje de programación e interactúa con el navegador usando Selenium Web Driver.

Su principal ventaja, en comparación con otras herramientas alternativas, es que detecta automáticamente si la página web ha terminado de cargar antes de ejecutar el

[10] SoapUI - https://www.soapui.org/
[11] Postman - https://www.getpostman.com/
[12] JavaScript - https://developer.mozilla.org/es/docs/Web/JavaScript

siguiente evento simulado de navegador y permite un acceso sencillo a los datos guardados por la aplicación web.

2.4.3.1.3. SerenityJS

Pese a que desarrollar pruebas automatizadas con Protractor es más cómodo que usar solo Selenium, sigue sin proporcionar suficientes abstracciones que faciliten la traducción de las especificaciones de la prueba a código que las automatice.

Por esto nació SerenityJS[13]. Esta herramienta implementada sobre Protractor (y por tanto Selenium) provee de abstracciones comunes en el diseño de las pruebas como son los actores, habilidades, interacciones, tareas y preguntas.

Cabe destacar que esta herramienta lleva sin lanzar una actualización estable desde hace dos años.

2.4.3.1.4. Cypress.io

Al igual que las anteriores, Cypress.io[14] es una herramienta que se centra en el desarrollo de pruebas de GUI web. Como punto diferenciador con la mayoría de herramientas de este estilo destaca el rechazo por Selenium como plataforma sobre la que desarrollarse.

2.4.3.2. Herramientas de 3ª generación

Aunque las herramientas de 2ª generación eran un avance importante en flexibilidad respecto a las herramientas de 1ª generación, siguen siendo fáciles de romper si se cambia el modelo subyacente a la vista que, en el caso de la web, es el DOM generado a partir de la combinación de HTML, CSS y código JavaScript, aunque el usuario siga viendo el mismo resultado en pantalla.(2)

Las herramientas de 3ª generación pretenden acercarse más a cómo los humanos probamos el software mediante técnicas de reconocimiento y comparación de imágenes.

2.4.3.2.1. SikuliX

Creada en 2009 y aunque con un equipo pequeño se mantiene todavía en desarrollo. SikuliX[15] es una herramienta para la automatización de interacciones con navegador hecha en Java que usa la librería de reconocimiento de imágenes mediante *machine learning* OpenCV para la localización de los componentes de la aplicación.

Aunque se ejecuta sobre la máquina virtual de Java o JVM, es políglota pues ofrece soporte para las versiones de Python y Ruby implementadas sobre la JVM, Jython y JRuby respectivamente. También, al estar desarrollada en Java, ofrece una API para programar los scripts con la lógica de prueba también en este lenguaje.

SikuliX es un producto gratuito y el proyecto está publicado bajo la licencia permisiva MIT.

2.4.3.2.2. EyeAutomate

[13] SerenityJS - https://serenity-js.org/
[14] Cypress.io - https://www.cypress.io/
[15] SikuliX - http://sikulix.com/

EyeAutomate[16] es un conjunto de productos para la automatización de pruebas mediante reconocimiento de imagen guiado por machine learning.

2.5. Principios de desarrollo de software mantenible

2.5.1. SOLID

SOLID es un acrónimo mnemotécnico para referirse a los que en (3) se declara como los cinco principios fundamentales para el diseño de clases en el diseño en lenguajes orientados a objetos con el fin de conseguir código fácil de extender y de mantener. Aunque es aconsejable seguir estos principios a la hora de desarrollar cualquier tipo de aplicación, como comenta el propio autor, estos principios no son leyes ni reglas sino consejos o heurísticas para producir mejor software.(4)

2.5.1.1. Single Responsibility Principle

Una clase solo puede tener un motivo para cambiar. (5)

Se entiende que si una clase cambia por dos o más motivos diferentes tiene más de una responsabilidad.

2.5.1.2. Open/Closed Principle

Las clases deben poder extenderse sin que se tengan que modificar.(5)

La motivación de este principio es hacer el código más reusable y mantener el código que ya funciona sin cambios.

2.5.1.3. Liskov Substitution Principle

Las clases derivadas deben ser sustituibles por su clase base.(5)

Las clases derivadas no deberían romper los contratos implícitos definidos por la clase de la que extienden. Por ejemplo, si creamos una clase que represente un pájaro y suponemos cuando la usamos que puede volar y más tarde creamos una clase que representa un pingüino, no podemos hacer que la clase del pingüino derive de la clase de pájaro porque los pingüinos no vuelan.

2.5.1.4. Interface Segregation Principle

Las interfaces, cuanto más específicas sean, mejor.(5)

Este principio busca que los métodos definidos en las interfaces tengan la mayor cohesión posible y que las clases que las implementen no tengan métodos propios de la interfaz que no necesiten implementar. Si por algún motivo esto es necesario, que la clase no implemente la interfaz directamente, sino que dependa de una clase abstracta que le obligue a implementar solamente lo necesario.

2.5.1.5. Dependency Inversion Principle

Depende de abstracciones en lugar de implementaciones.(5)

Este principio busca aumentar la flexibilidad del código haciendo fácil la sustitución de una implementación de una abstracción por otra implementación diferente de esa misma abstracción.

[16] EyeAutomate - https://eyeautomate.com/

2.5.2. DRY

Don't Repeat Yourself o DRY es un principio de desarrollo de software acuñado por primera vez por Andrew Hunt y David Thomas que consiste en que toda pieza de información debe tener una representación única, autoritaria y precisa dentro del sistema. (6)

Existen 4 fuentes de duplicidad en cualquier sistema software.

- ***Imposed duplication.*** Las duplicidades ocasionadas por necesidad del entorno.
- ***Inadvertent duplication.*** Aquellas aparecidas por despiste de los desarrolladores.
- ***Impatient duplication.*** Información duplicada que aparece porque a veces parece más rápido duplicar y modificar ligeramente un sistema que reutilizarlo.
- ***Interdeveloper duplication.*** Cuando varios trabajadores trabajan en el mismo sistema pueden introducir información que ya había introducido otro previamente.

2.6. Crítica al estado del arte

Existe bastante literatura acerca de las pruebas de software y sobre mantenimiento de software individualmente, pero existe una laguna en lo referente al mantenimiento del software de pruebas.

2.7. Propuesta

La finalidad de este TFG es dar solución a problemas frecuentes en el mantenimiento de pruebas de sistema para una aplicación web. Más en concreto a aquellos problemas surgidos en el mantenimiento de una *suite* desarrollada con la herramienta Protractor. Parte de esos problemas podrían haberse abordado migrando a la herramienta SerenityJS, pero como lleva sin lanzar una actualización estable dos años y tampoco solucionaba muchos de los demás problemas se decidió hacer un desarrollo propio.

Capítulo 3. Análisis

3.1. Requisitos

A continuación analizaremos los diferentes requisitos que nos gustaría que cumplieran nuestras pruebas. Como estamos ejecutando una *suite* concreta sobre un SUT en concreto, nos pararemos a hacer una diferencia entre los requisitos deseables en este tipo de pruebas que se hacen en la *suite* y los que lo son por la naturaleza del SUT.

3.1.1. Específicos para pruebas GUI de sistema

3.1.1.1. Ejecución determinista

Con esto nos referimos a que la ejecución de cada uno de los casos de prueba produzca el mismo resultado para la misma versión de la *suite*, la misma versión del SUT y la misma parametrización de datos.

De esta manera, el desarrollador que analice la prueba podrá tratar de reproducir el error para intentar localizarlo.

3.1.1.2. Trazabilidad de fallos

Las pruebas de software tienen como objetivo encontrar errores o defectos y corregirlos de manera eficiente. Para conseguirlo, uno de los requisitos más importantes, si no el que más, es la trazabilidad del error a corregir. Acotar el fallo a una parte de la aplicación o incluso hasta el nivel de módulo o clase es el primer paso para desarrollar la corrección del error.

Además, como consecuencia de una buena trazabilidad del fallo, se debería poder diferenciar un error en el SUT de un falso positivo en la prueba pues si la traza acaba en el SUT debería ser un error en éste y si no, en la lógica de las pruebas.

3.1.1.3. Nuevas abstracciones

Con el fin de encapsular comportamientos y reusarlos en diferentes pruebas para seguir el principio DRY se proponen nuevas abstracciones para los conceptos de *CasoDePrueba, Acción* y *Componente*.

El *CasoDePrueba* representa una prueba de nuestra *suite*.

La *Acción* representa algo que se tiene que realizar antes de un *CasoDePrueba*.

Un *Componente* encapsula la descripción y las interacciones necesarias para realizar una *Acción* o *CasoDePrueba*. Se tratará más en detalle en el capítulo 6 - SOLID y DRY mediante un framework ad-hoc.

3.1.1.4. Persistencia de los resultados

Es conveniente obtener datos de más de una ejecución para poder extraer conclusiones acerca de los posibles problemas en la *suite* de pruebas o del propio SUT.

3.1.2. Específicos del SUT

A la hora de analizar los requisitos conviene pararse a pensar en qué tipo de pruebas podrían implementarse en un futuro para nuestro SUT.

3.1.2.1. Pruebas fácilmente parametrizables

En nuestro caso, el SUT es un HIS con un modelo de datos y una configuración de alta complejidad. La casuística de los casos de prueba puede ser complicada de manejar y sería deseable que los casos de prueba fuesen sencillos de parametrizar.

3.2. Restricciones técnicas

3.2.1. Identificación del fin de la carga de la web del SUT

En la transición de AngularJS 1.7 a Angular 2, Google cambió mucho la manera en que funcionaba su framework. En un principio el SUT usaba solamente AngularJS, pero se comenzó a utilizar junto con código hecho con Angular 8 lo que provocó que Protractor dejara de comprobar correctamente cuándo la página web del SUT había terminado de cargar. Esto provocó en la *suite* la necesidad de añadir esperas a mano cada vez que se le pedía a Protractor que interactuase con el navegador pues ya no era capaz de detectarlas automáticamente.

3.2.2. Difícil automatización del despliegue del SUT

La base de datos del SUT, que como se ha comentado anteriormente es una base de datos SQL, no posee scripts de inicialización y configuración por defecto por lo que, cada vez que se crea un entorno nuevo, la nueva base de datos se inicializa con un *backup* de otro entorno ya en funcionamiento. Esto es una limitación a la hora de automatizar un despliegue.

3.3. Soluciones propuestas

3.3.1. Respecto a la ejecución determinista

Aunque no es común que ocurra y se puede solucionar añadiendo esperas manuales dentro de la lógica de las pruebas, al no detectar automáticamente cuando la página web del SUT ha terminado de cargar, las interacciones con el navegador pueden resultar inconsistentes y devolver datos vacíos que no lo estarían si ese evento se hubiese ejecutado al finalizar la carga.

La solución más sencilla a corto plazo es ralentizar la ejecución de las pruebas añadiendo esperas manuales o código extra para intentar ejecutar de nuevo la parte fallida en la lógica de las pruebas, pero puede ocasionar a medio-largo plazo problemas de mantenimiento.

Otra solución menos sencilla pero quizás más correcta sería desarrollar un sistema que vuelva a ejecutar las pruebas fallidas tantas veces como se configure hasta que la prueba pase o se agoten los reintentos.

3.3.2. Respecto a la trazabilidad

Existen varias maneras de mejorar la trazabilidad de los errores detectados durante las pruebas.

Un forma puede ser mejorar las trazas de error resultantes cuando se produce un error en el código de las pruebas haciéndolas más descriptivas y especificas según el tipo de error. Esto ayudaría a localizar más rápido el código fallido dentro de nuestra *suite*.

Relacionado con esta sugerencia anterior, cuanto antes falle el programa, más cerca estará la traza del código fallido del error real. Podría darse el caso, por ejemplo, de que una parametrización de la prueba no sea opcional pero no se configure. En este caso, si el programa falla cuando intenta usar el dato que no existe puede que falle más tarde. En cambio, si lo hacemos fallar en cuanto se evalúe la parametrización sabremos qué parametrización falta y habremos perdido menos tiempo.

Otra medida, esta vez más involucrada con el SUT en cuestión, es registrar las llamadas HTTP que realiza para, en caso de fallo, comprobar si el error ha sido detectado por el servidor del SUT o ha pasado desapercibido. Si el error ha sido detectado, es decir, nos devuelve un código de error alguna llamada HTTP indispensable, nos puede facilitar la trazabilidad indicándonos en qué modulo y clase del servidor se ha producido el error.

Para acabar con las mejoras realizables a la trazabilidad de las pruebas, sería de utilidad, ya que son pruebas de GUI, poder grabarlas para tener más información de las acciones que se han realizado en la ejecución y cómo se han realizado por si hubiese algún fallo. El motivo de la existencia de esta característica es el mismo que el de las capturas de pantalla, pero con más detalle.

3.3.3. Respecto a la persistencia de resultados

Para este requisito debemos considerar varias soluciones.

Primero, ¿guardamos los resultados de las diferentes ejecuciones en un sistema de ficheros o en una base de datos? Quizás la solución más sencilla sea en ficheros, pero guardarlos como texto es poco eficiente y después de muchas ejecuciones puede que toque borrar ejecuciones y si lo guardamos como archivos binarios tendríamos que desarrollar código específico para su serialización y deserialización. Además, eso iría en contra del objetivo del requisito, que es poder sacar conclusiones de esos datos. Para esto y por su capacidad de integración en un futuro es mejor persistir los resultados en una base de datos.

Segundo, ¿una base de datos en servidores propios o de terceros? Como este trabajo está realizado en un contexto de prácticas de empresa, para evitar filtrar material con derechos de autor a terceros, se ha preferido optar por una base de datos propia.

Como primera forma de explotación de esta persistencia se propone una pantalla con un histórico que muestre el resumen del resultado de cada una de las ejecuciónes de la *suite* que enlace a una vista más detallada de ésta.

3.3.4. Respecto a las nuevas abstracciones. Aplicación de los principios SOLID y DRY

Protractor y Selenium no proporcionan abstracciones mediante el uso de clases. En ese sentido son *frameworks* que proporciona una API imperativa pese a que tanto JavaScript como TypeScript[17] implementan el paradigma orientado a objetos desde hace bastantes versiones. Por este motivo, es difícil aplicar directamente los principios

[17] TypeScript - http://www.typescriptlang.org/

SOLID en el diseño de los scripts de prueba y será necesario definir una serie de clases como abstracción.

Respecto al principio DRY, Protractor propone un método para encapsular funcionalidad de pruebas al que llaman *Page object*. Más adelante veremos porqué este método de encapsulación no encaja bien en todas las situaciones y más concretamente en la nuestra. Se propondrá como alternativa la abstracción de *Componente*. Al implementar estas nuevas abstracciones habrá que cuidar de que intenten seguir los principios SOLID.

Además, otra mejora de la trazabilidad consistiría en guardar la URL en la que finaliza cada prueba y hacerla accesible desde el reporte final de la ejecución. Esto se podría implementar como un hiperenlace que lleve a la URL de la prueba si se clica en la descripción de ésta.

3.3.5. Respecto a la parametrización de las pruebas

Se propone un DSL que abstraiga al desarrollador de los detalles de la ejecución del flujo y que solo deje su estructura más sencilla de manera declarativa. De esta forma, en ese DSL se especificará qué tareas de alto nivel se ejecutan antes que otras y con qué parámetros lo hacen.

Para simplificar la visualización del flujo en ese DSL, la implementación debe ser capaz de autoconfigurar los datos todo lo que pueda. Para ello, en una primera fase de configuración, las diferentes tareas deben comunicar entre sí sus requisitos para una correcta ejecución.

Este DSL sería fácil de integrar con el *framework* propuesto en el requisito sobre las nuevas abstracciones.

3.4. Marco legal

3.4.1. Análisis de la protección de datos

Los únicos datos personales recogidos de las pruebas y almacenados podrían ser los nombres de los usuarios de red que hayan ejecutado las pruebas. El resto de los datos son generados y no van asociados a ningún tipo de persona física.

3.4.2. Propiedad intelectual

El código desarrollado como solución a este trabajo de fin de grado se ha escrito en un contexto de prácticas de empresa que tiene su licencia. Por ello, el código que pueda aparecer será de fuentes externas o creado exprofeso para esta memoria.

Capítulo 4. Diseño de la aplicación

4.1. Punto de partida

Por economía de trabajo, a la hora de diseñar la vista y la integración con la *suite* de pruebas se toma como base el proyecto, licenciado bajo la licencia permisiva MIT, *protractor-beautiful-reporter.* Este proyecto es un *plugin* para el *framework* de pruebas Protractor que para cada ejecución genera una página HTML con los resultados de las pruebas y se puede configurar para que haga una captura en el momento del fallo de la prueba y lo asocie en el reporte a ésta.(7)

4.2. Arquitectura de la aplicación

4.2.1. Arquitectura inicial

La herramienta inicial consiste en un *plugin* que genera una página estática para cada ejecución, sobrescribiendo los datos que pudiese haber de ejecuciones anteriores.

En cuanto al código y la vista de la web generada, la plantilla hace uso de AngularJS para segregar la vista de la lógica, pero incluye toda la vista en un archivo *index.html* y toda la lógica en un solo script *app.js*.

El funcionamiento del *plugin* es como sigue. Nada más empezar la ejecución, se acopla al *framework* Protractor, con la configuración definida por el desarrollador de las pruebas, el cual le avisa cada vez que una prueba o grupo de pruebas empieza y termina, además del estado en el cual ha finalizado, guardando en memoria principal éste y otros metadatos relacionados con la ejecución de esa prueba o grupo de pruebas. Si está configurado para hacerlo y la prueba resulta fallida, la aplicación guardará una captura de pantalla del navegador en el momento del fallo y la asociará a la prueba fallida.

Antes de terminar con la ejecución de Protractor, el *plugin* copia los archivos de la plantilla web a memoria principal y reemplaza algunos *tokens* dentro de ésta por algunas configuraciones y los datos guardados de la ejecución y lo vuelve a guardar en disco en la carpeta que se haya configurado como objetivo de los archivos de reporte.

Finalmente, para visualizar el reporte del error se abriría el archivo *reporter.html* creado por el *plugin* en el navegador predeterminado.

Figura 1: Vista del reporte creado con el plugin inicial. Fuente (7)

4.2.2. Arquitectura final

Como se pretende integrar una base de datos y ya existe parte de la GUI creada con tecnologías web, se ha decidido que la arquitectura de la aplicación seguirá una estructura típica de base de datos, un cliente web como GUI y un servidor que haga de mediador entre los dos.

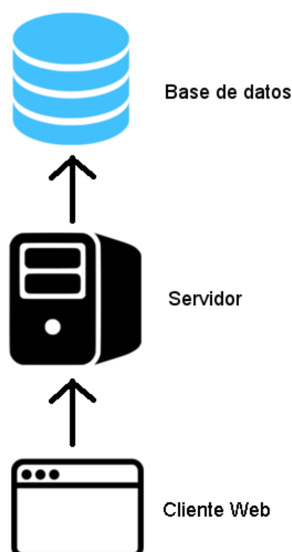

Figura 2: Arquitectura base de datos, servidor y cliente web.

4.2.2.1. Base de datos

Las entidades de datos necesarias de guardar en base de datos son los metadatos de los grupos de pruebas o *suites*, de las pruebas en sí mismas y de la ejecución en su conjunto.

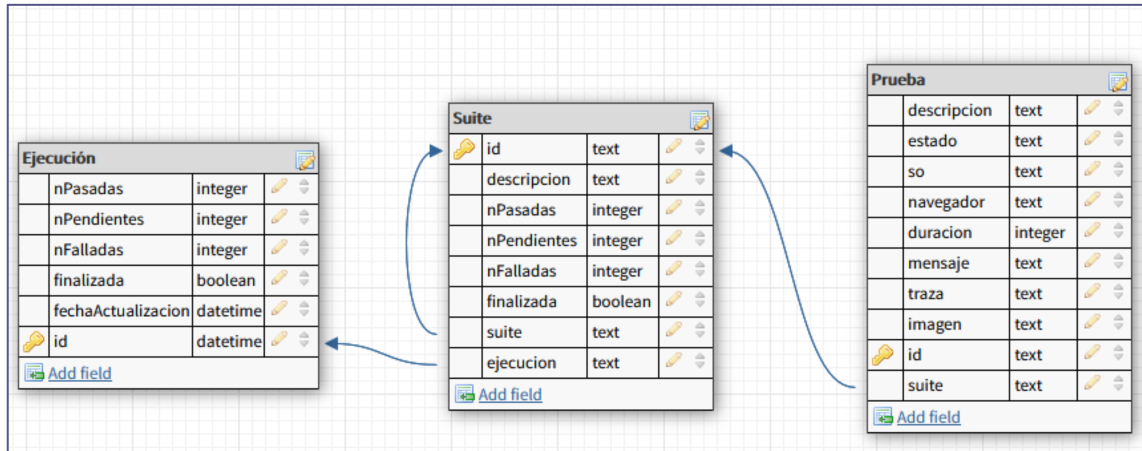

Figura 3: Entidades en base de datos

4.2.2.2. Servidor

Para fomentar el mantenimiento de la herramienta por parte del equipo de desarrollo, el servidor es desarrollado con un lenguaje de programación con el que están familiarizados en el desarrollo del *frontend* del SUT, TypeScript. Además, el *framework* escogido para ello promueve una estructura interna similar a la del *framework* Angular.

Al ser un servidor creado para acceder a la base de datos, los servicios y controladores desarrollados no son demasiado complejos y son en esencia un CRUD.

El servidor constaría de cuatro paquetes o módulos que hacen referencia a las entidades de datos que manejaría la aplicación y a un módulo de configuración.

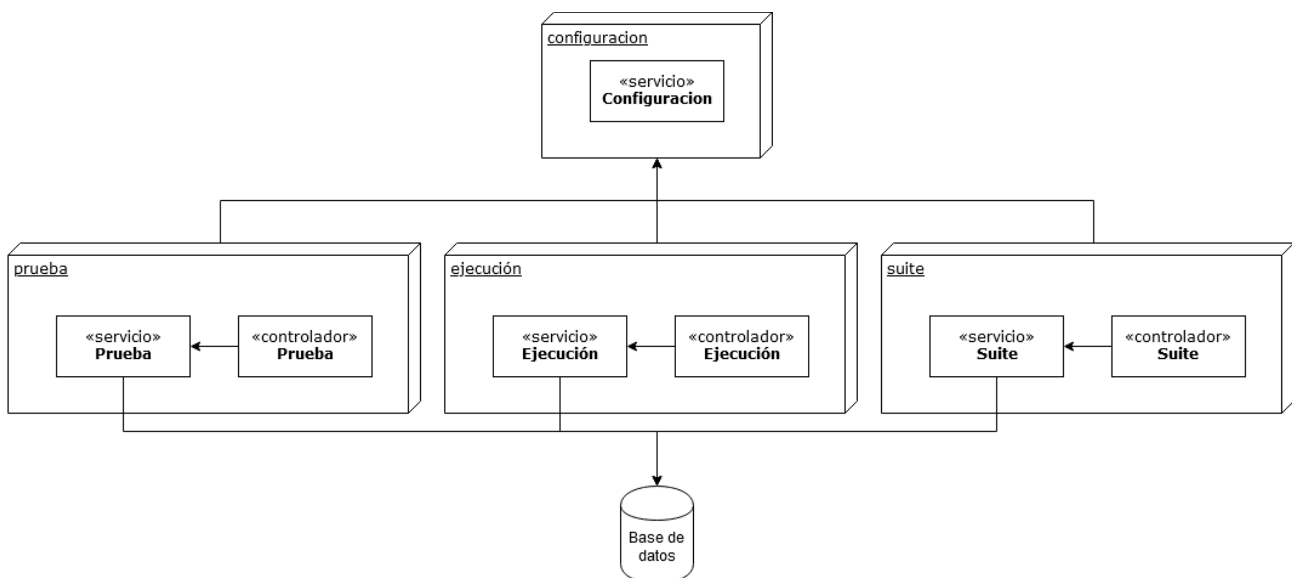

Figura 4: Arquitectura del servidor de la aplicación

4.2.2.3. Cliente web

El cliente web se refactorizaría en componentes AngularJS siguiendo una arquitectura de software comúnmente usada en aplicaciones con GUI denominada *Model-View-Controller* o MVC. Básicamente consiste en separar la descripción de la vista, los datos que usa y su manera de manejarlos.

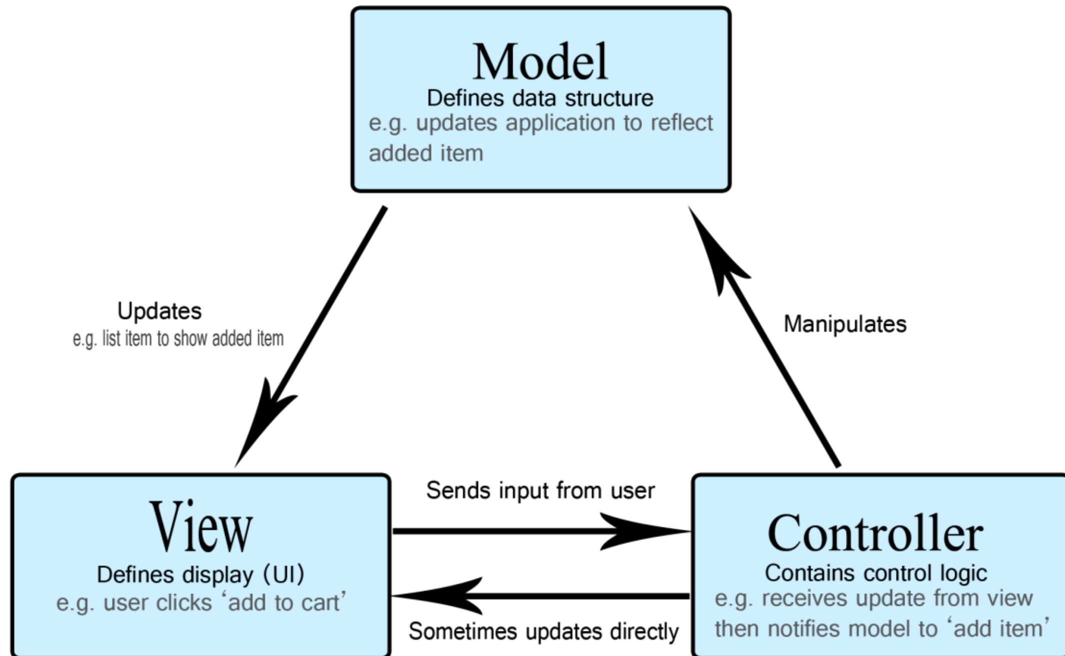

Figura 5: Diagrama Model-View-Controller. Fuente (8)

En la pantalla generada por el *plugin* inicial, podemos extraer los componentes que se detallan más abajo, cada uno de los cuales tendrían un fichero HTML que describiría la vista, un controlador JavaScript e implícitamente el modelo de datos recibido por el servidor en el controlador.

En esta parte del diseño se pueden diferenciar los siguientes componentes en el reporte de la prueba.

Figura 6: La fila de la prueba

CU-BCEX-01-1:

	Time	Description	Browser	Message	
✔	16.3 s	Acceder a de	chrome	Passed.	1 1
✔	13.4 s	Buscar el	chrome	Passed.	
✔	17.9 s	Rellenar	chrome	Passed.	
✔	5.8 s	Buscar	chrome	Passed.	
✔	40.2 s	Confir	chrome	Passed.	1

Figura 7: Los datos de la suite

Status Time Description Browser OS Message Log Stack Screen

CU-BCEX-01:

CU-BCEX-01-1:

	Time	Browser	Message	
✔	16.3 s	chrome	Passed.	1 1
✔	13.4 s	chrome	Passed.	
✔	17.9 s	chrome	Passed.	
✔	5.8 s	chrome	Passed.	
✔	40.2 s	chrome	Passed.	1

CU-BCEX-01-2:

	Time	Browser	Message	
✔	10.9 s	chrome	Passed.	1
✔	13.6 s	chrome	Passed.	

CU-BCEX-01-3:

	Time	Browser	Message	
✔	35.0 s	chrome	Passed.	1
✔	20.9 s	chrome	Passed.	1

Figura 8: La lista de suites

Search | ALL | PASSED | FAILED | PENDING | WITH LOG

Figura 9: El filtro de búsqueda de las suites

CU-BCEX-01-11:

Añadir

```
Error: Expected 1 to equal 2.
    at <Jasmine>
    at UserContext.<anonymous> (C:\Users
    at <Jasmine>
    at processTicksAndRejections (internal/process/task_queues.js:93:5)

Error: Expected 'No se pudo recuperar ' not to contain ''.
    at <Jasmine>
    at UserContext.<anonymous> (C:\Users\
    at <Jasmine>
    at processTicksAndRejections (internal/process/task_queues.js:93:5)

Error: Failed:  cannot be performed
    at <Jasmine>
    at UserContext.<anonymous> (C:\Users
    at <Jasmine>
    at processTicksAndRejections (internal/process/task_queues.js:93:5)
```

Browser logs:

SEVERE 2019-11-19, 2:18:21 PM
http://localhost/

SEVERE 2019-11-19, 2:18:21 PM
http://localhost Object

Smart Stack Trace | Close

Figura 10: Visor del log del navegador y de las trazas de error

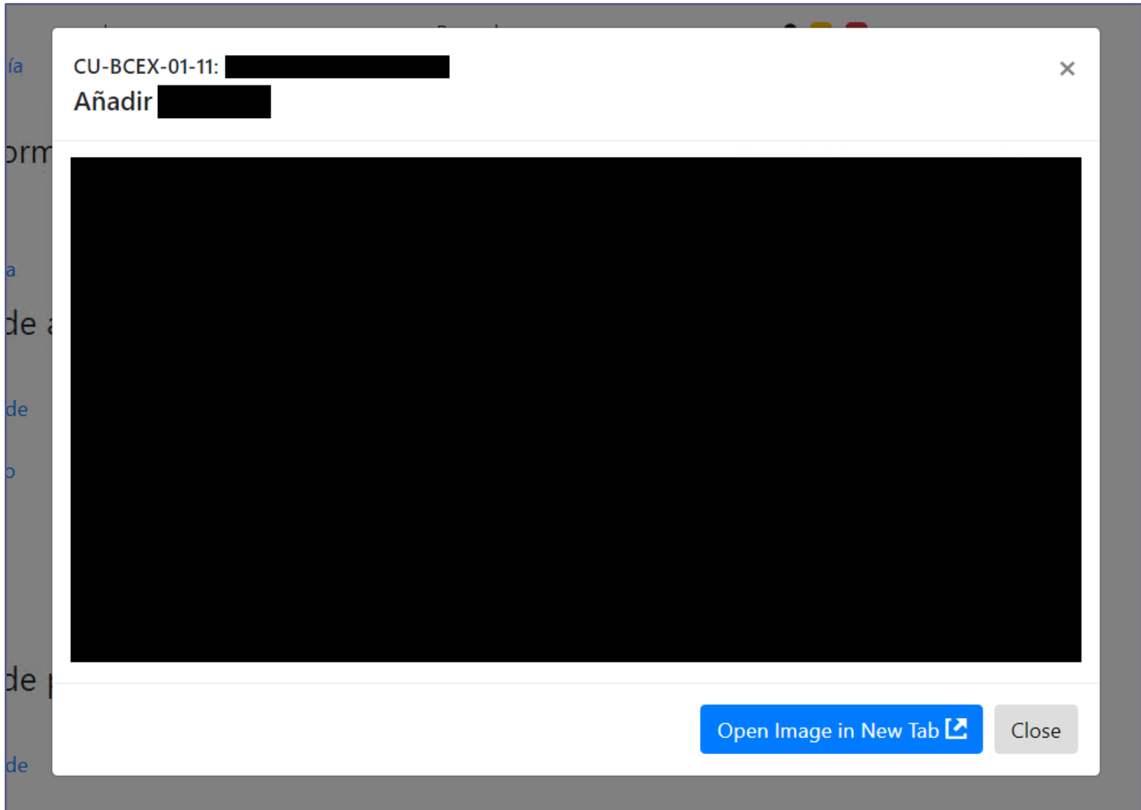

Figura 11: Vista de capturas de pantalla

Figura 12: Barra de recuento de pruebas pasadas y falladas

Figura 13: Componente de detalles de la ejecución

4.3. Diseño de la UI de la aplicación de apoyo

Para completar los requisitos definidos en el análisis sobre persistencia y trazabilidad, se ha visto conveniente añadir las siguientes pantallas a la interfaz de la aplicación.

4.3.1. Lista de ejecuciones

Como medida básica de explotación de la persistencia de los resultados de la prueba se desarrollará una lista de ejecuciones de las pruebas que se convertirá en la pantalla principal de la aplicación.

Cada ejecución debe ser representada dentro de la lista como un conjunto que muestra de manera resumida los resultados de la ejecución. Así, en este resumen constará necesariamente el número de pruebas pasadas, el número de pruebas fallidas, si ha terminado o está en ejecución, la versión del SUT en la que se ha ejecutado y la fecha de ejecución. Además, al clicar sobre el elemento, el navegador debe redirigir a la pantalla de la aplicación con el detalle de la ejecución. Ésta será en esencia la pantalla que generaba inicialmente el *plugin*.

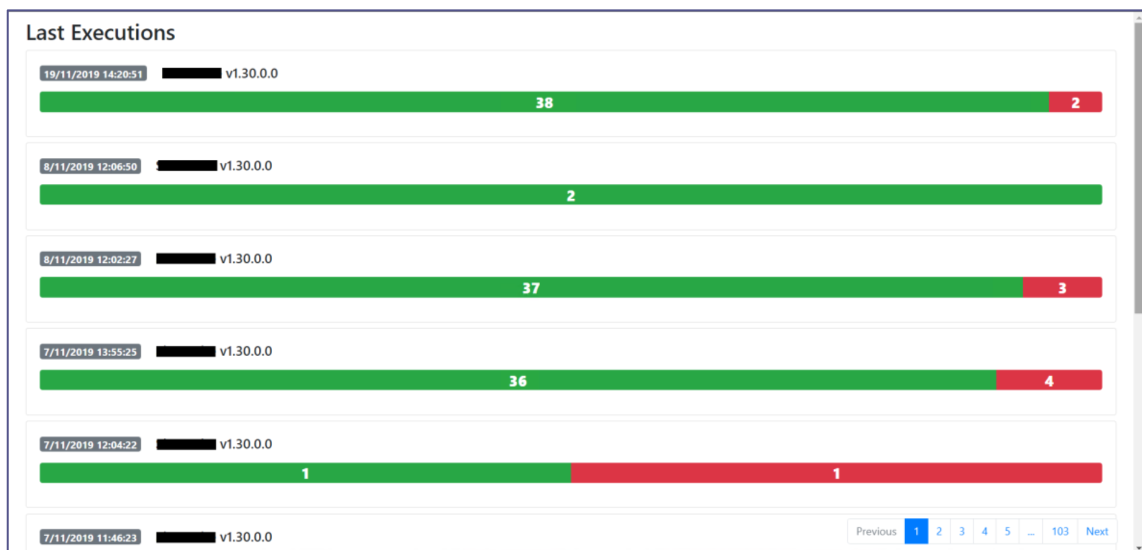

Figura 14: Lista de ejecuciones

Como podemos observar, para evitar problemas de rendimiento futuros y mejorar la usabilidad de la pantalla, la lista debe estar paginada.

4.3.2. Registro de llamadas HTTP

Como medida de trazabilidad de errores se desarrollará también otra pantalla nueva donde se pueda observar las llamadas HTTP realizadas en cada prueba por el cliente web del SUT, los datos con las que han sido invocadas, los datos devueltos y en qué estado han acabado.

Como medida adicional de usabilidad, ya que la lista de llamadas puede ser grande, el registro de llamadas debe estar paginado y debe tener un filtro que facilite encontrar las llamadas fallidas.

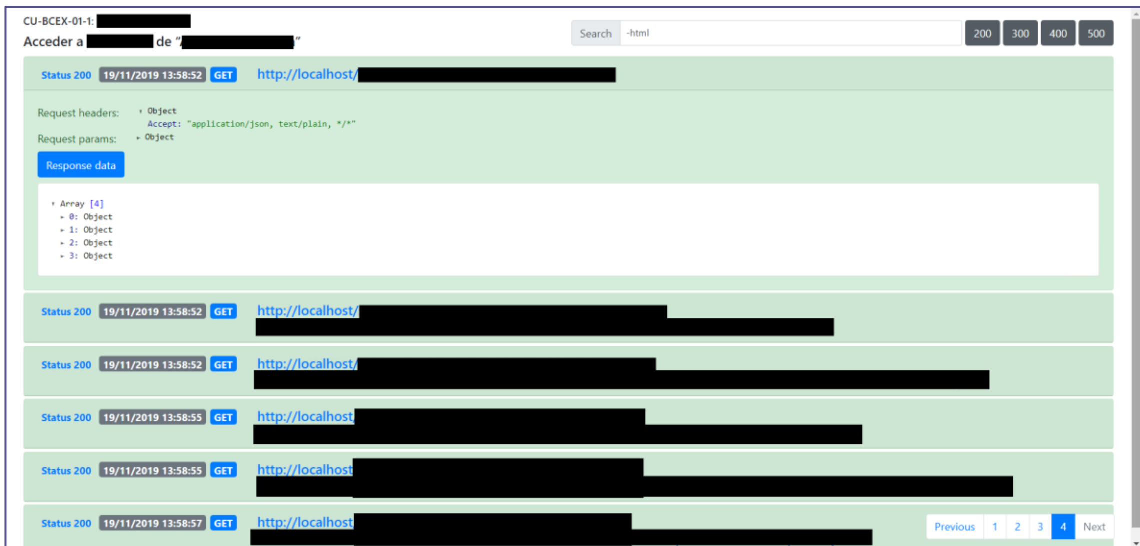

Figura 15: Registro de llamadas HTTP

Como existirá un registro de llamadas HTTP para cada prueba, se accederá a la pantalla desde la vista detallada. Será necesario añadir un icono junto a los de error que nos permita acceder al registro de llamadas realizadas durante la prueba.

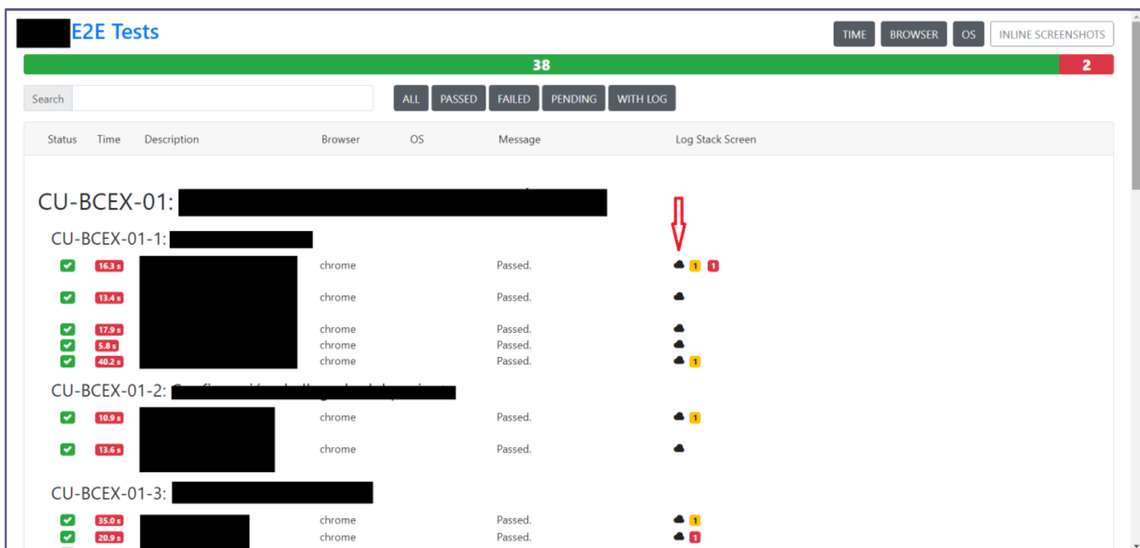

Figura 16: Modificación de la pantalla original de detalles de la ejecución

4.4. Nueva funcionalidad del *plugin*. Grabación de pantalla.

Al objeto de mejorar la trazabilidad de errores en las pruebas, se implementará otra función adicional dentro del *plugin* de Protractor que permita grabarlas mediante su integración con herramientas de captura de video. Se debe añadir una opción de configuración en el *plugin* que permita definir una carpeta donde volcar el video capturado. Dentro de esta carpeta se deberá crear una carpeta con el *id* de la ejecución y dentro de esa nueva carpeta se volcará un video por cada *suite* ejecutada.

4.5. Tecnología utilizada

4.5.1. TypeScript

TypeScript es un superconjunto tipado de JavaScript que se compila a código JavaScript. El objetivo de este lenguaje no es más que añadir la seguridad de la teoría de tipos en tiempo de compilación al código JavaScript y es completamente interoperable con éste.

La decisión de escoger este lenguaje para la implementación del *backend* de la aplicación de apoyo a las pruebas es motivar a aquellos que desarrollen y mantengan la *suite* de pruebas ya existente, que está programada en este lenguaje, a mantener y mejorar también este servicio ofreciéndoles un lenguaje familiar. Además, el uso de tipos ayuda a detectar errores en la estructura de los datos internos.

4.5.2. JavaScript

JavaScript (JS) es un lenguaje ligero e interpretado, orientado a objetos con funciones de primera clase, más conocido como el lenguaje de script para páginas web, pero también usado en muchos entornos sin navegador, tales como node.js, Apache CouchDB y Adobe Acrobat. Es un lenguaje script multi-paradigma basado en prototipos, dinámico, soporta estilos de programación funcional, orientado a objetos e imperativo.

JavaScript es el lenguaje elegido para implementar el *frontend* de la aplicación de apoyo a las pruebas por reutilizar código del punto de partida.

4.5.3. AngularJS

AngularJS es un *framework* creado en 2010 y actualmente mantenido en estado LTS por Google para facilitar el desarrollo de páginas web dinámicas mediante el uso de código JavaScript.

Este *framework* era usado por la plantilla del *plugin* del punto de partida y ha sido mantenido como *framework* sobre el que está desarrollado el *frontend* de la aplicación de apoyo a las pruebas.

4.5.4. NodeJS

"Node.js® es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome"

NodeJS[18] es el entorno de ejecución JavaScript más popular, con mayor actividad de desarrollo y con más comunidad hasta el momento. Existen algunas alternativas, pero son menos usadas y por tanto menos probadas.

4.5.5. ESLint

ESLint es una herramienta de *linting* para JavaScript, y recientemente para también para TypeScript, creada en junio de 2013. Cuando hablamos de *linting* de código nos referimos a un tipo de análisis estático que es usado a menudo para encontrar patrones comunes posiblemente problemáticos o código que no sigue una cierta guía de estilo.

Debido a que las pruebas las podrían mantener diferentes personas a lo largo del tiempo es importante que todas escriban el código usando un estilo similar.

4.5.6. NestJS

[18] NodeJS - https://nodejs.org/es/

NestJS[19] es un framework para construir servidores de aplicaciones web hecho en TypeScript y orientado a usarse con este lenguaje.

La elección de este framework para montar el servidor sobre él es que la arquitectura que promueve basada en servicios y su uso de inyección de dependencias se parece mucho al framework utilizado en el servidor del SUT, mientras que también emplea el mismo lenguaje que la *suite* de pruebas.

4.5.7. MongoDB

MongoDB[20] es una base de datos distribuida, basada en documentos y de uso general, que ha sido diseñada para desarrolladores de aplicaciones modernas.

Como solución de persistencia fue escogida la base de datos MongoDB por ser una base de datos popular y fácilmente integrable en entornos JavaScript porque permite almacenar datos sin una estructura bien definida.

4.5.8. FFmpeg

FFmpeg[21] es una solución multiplataforma de código abierto y libre para grabar, convertir y reproducir audio y video. Será la herramienta usada para implementar la grabación en video de las ejecuciones de los test.

4.5.9. Git

Git es un sistema de control de versiones o CVS, distribuido, de código libre y abierto para manejar desde pequeños hasta grandes proyectos.

Fue elegido sistema de control de versiones para la aplicación de apoyo a las pruebas por ser popular y usado en todos los repositorios del equipo de desarrollo.

---

[19] NestJS - https://nestjs.com/
[20] MongoDB - https://www.mongodb.com/es
[21] FFmpeg - https://www.ffmpeg.org/

Capítulo 5. Desarrollo de la aplicación

Para crear el proyecto se usó la aplicación de consola del *framework* NestJS que nos crea las carpetas básicas y código de ejemplo dentro de la carpeta donde se encuentra el código fuente del servidor, llamada *src*.

5.1. Refactorización de la aplicación inicial

Para comenzar, se clonó el código fuente de (7) y se movieron los dos archivos fuentes de la carpeta *app,* que componen el *plugin* en sí mismo, a nuestra *suite* de pruebas y los archivos de la carpeta *lib,* que son los archivos que sirven de plantilla para la generación del reporte, a una carpeta *public* dentro de proyecto generado por NestJS.

Así pues, habríamos separado el *plugin* de la plantilla y tendríamos en *public* una aplicación base sobre la que desarrollar nuestros requisitos.

La siguiente fase, y la más costosa en este punto, consistió en identificar y separar los componentes comentados en el punto 4.2.2.3 de manera que facilitara la modificación y mantenimiento de la lógica de éstos. La mayor dificultad fue comprender cómo funcionaba el código para mantener la funcionalidad en la refactorización.

5.2. Separación entre *frontend* y *backend*. Integración con una base de datos.

Una vez que tenemos el código inicial mejor organizado, instalamos una base de datos local en nuestro equipo de desarrollo tal como indican las instrucciones en (9) sin mayor problema.

5.2.1. Implementando el servidor

Para implementar el servidor, se requiere desarrollar los servicios y controladores para el CRUD de las entidades guardadas en la base de datos. Esto es, para la entidad de las ejecuciones, para la de los grupos de pruebas o *suites*, para la de las pruebas y para las imágenes asociadas a las pruebas fallidas.

Los controladores expondrían estos servicios CRUD definiendo una api REST e intercambiando datos JSON mediante el protocolo HTTP. Además, es el servidor quien se encarga de proveer por HTTP los archivos de la carpeta *public* como recursos web para hacerlos accesibles desde el navegador.

No hubo problemas significativos en esta fase del desarrollo pues la lógica era simple y los patrones de diseño conocidos.

5.2.2. Modificando el *plugin*

Se requirió de una gran refactorización del fichero *repoter.js* para cambiar el comportamiento del plugin y conseguir que los datos acabasen volcados en la base de datos y no en ficheros.

Lo primero que se hizo fue migrarlo de JavaScript a TypeScript renombrándolo como *reporter.ts*. Ya que la sintaxis de TypeScript es un superconjunto del de JavaScript, no hubo ahí gran problema y mejoró la experiencia con el entorno de desarrollo pues se hacía más fácil navegar por el código y el autocompletado de las funciones mejoraba.

A continuación borramos el archivo *util.js* pues era el que contenía la lógica de persistencia de los datos en ficheros. Después añadimos como dependencia de la *suite* de pruebas el código del servidor de la aplicación de apoyo a las pruebas. La solución para introducir los metadatos en la base de datos fue importar las clases servicio del servidor en el *plugin* e integrarlas directamente en el código del *plugin* con el ciclo de vida de Protractor.

La principal dificultad en esta fase fue entender bien cómo funcionaba el *plugin* a refactorizar. También requirió actualizar el *framework* Protractor a la versión 6 porque la interfaz que suministraban las versiones anteriores para crear *plugins* propios no permitía la integración con NestJS.

Como parte de la mejora de trazabilidad, se añade la URL en la que acaba cada prueba y se añade a los metadatos de ésta que posteriormente se guarda en base de datos.

5.2.3. Modificando el cliente web

Como último paso en la etapa de separación de responsabilidades del *plugin* inicial, se sustituyeron aquellos lugares del cliente web donde había *tokens* usados por el *plugin* inicial para insertar datos como si fuera una plantilla por llamadas HTTP al servidor de la aplicación desarrollado anteriormente.

Hubo también que refactorizar parte del código y de la vista ya que en el *plugin* inicial los datos estaban disponibles desde el momento de creación de la vista, pero en la nueva versión de la aplicación, los datos se suministran a la vista después de su creación por la naturaleza de las llamadas AJAX[22] . Además, ahora los datos llegaban más particionados pues antes el cliente web solamente tenía que lidiar con un JSON, insertado en el código, y ahora la información le llega de varias llamadas HTTP diferentes.

Como parte de las mejoras de trazabilidad, se añadió una propiedad al modelo de Prueba de la base de datos que guardaba la URL donde acaba. Ahora, por lo tanto, el cliente web puede utilizar esa propiedad nueva para enlazar la descripción de la prueba que aparece en el componente de la fila de la prueba mostrado en el diseño de la aplicación de manera que redirija a la URL del SUT a la que nos interesa acceder.

5.3. Funcionalidad con cambios en el *frontend*

5.3.1. Lista de ejecuciones realizadas

Para la creación de los componentes de la UI de la lista de ejecuciones se separó en dos de éstos. El componente que hace de elemento de la lista y que representa el resumen de una ejecución por un lado y por el otro el componente que es la lista entera de ejecuciones.

Para el diseño del componente del resumen de la ejecución se ha reutilizado el componente contador de pruebas pasadas y fallidas que aparece en la pantalla de detalles de la ejecución. Véase la Figura 12.

[22] AJAX - https://developer.mozilla.org/es/docs/Web/Guide/AJAX

El componente de la lista consiste en una vista HTML usando la directiva de AngularJS llamada ngRepeat con una lista de ejecuciones pedida al servidor.(10)

Además, se crea un filtro para el ngRepeat para que pagine los componentes que aparecen en la lista apoyado por un componente de paginación.

5.3.2. Trazabilidad mediante el registro de llamadas HTTP durante la prueba

Para integrar esta funcionalidad con el SUT, en el *plugin* se inicia un servidor HTTP escuchando en un puerto determinado a la espera de recibir registros en formato JSON. Hubo que modificar el componente interceptor de llamadas HTTP del SUT para que todas llamadas realizadas por el cliente web del SUT a direcciones diferentes al servidor HTTP en el *plugin* se enviaran a éste.

De esta manera, al persistir la prueba, persiste también su registro de llamadas HTTP realizadas durante la ejecución de ésta.

Por último, se desarrolla una vista nueva que es una lista de componentes desplegables en el que cada uno de los componentes muestra la información de una llamada HTTP: los parámetros con los que se invoca, los datos que devuelve y el código que retorna. También se crea un buscador en el que se pueden poner filtros excluyentes si se precede la palabra a excluir por un guion.

Al igual que la lista de ejecuciones, el número de llamadas HTTP puede ser considerable y para mejorar su usabilidad se hace emplea un componente de paginación.

5.4. Grabación de la ejecución de las pruebas

El desarrollo de esta funcionalidad consiste en integrar la herramienta de captura de video FFmpeg con el *plugin* de Protractor de manera que, si está configurado para realizar grabaciones, empiece una grabación de la pantalla cada vez que se inicia una *suite* y si la *suite* termina sin fallo, la grabación se borra. Si falla, la grabación debe guardarse en la ruta configurada. Las grabaciones son visibles con el reproductor de video VLC[23].

El primer problema encontrado en el desarrollo de esta característica fue que en sistemas con más de una pantalla capturaba la imagen de todas a la vez en lugar de tan solo la correspondiente al navegador automatizado. Como primera solución se intentó ejecutar las pruebas en un contenedor Docker con un entorno virtual, pero se encontró otro problema diferente con esa solución; ahora sí que se grababa una sola pantalla, pero si la ejecución se alargaba alrededor de los 10 minutos el contenedor se paraba sin dar ningún error comprensible.

[23] Reproductor de video VideoLAN - https://www.videolan.org/vlc/index.es.html

Capítulo 6. SOLID y DRY con un *framework* adhoc

6.1. Patrón *Page object* vs *Componente*. DRY

En el desarrollo de todo proyecto software hay que elegir formas de organizar nuestro código para, al menos, mejorar su legibilidad. Algunas de estas maneras vienen condicionadas por el entorno en el que se desarrolla mientras que en otros casos son diseñadas por desarrolladores más o menos experimentados que creen que pueden existir otras formas mejores de hacerlo. El software es complejo y no existen soluciones que cubran todos los casos de uso.

En el entorno de desarrollo de pruebas automatizadas con Protractor, si actuamos sin análisis previo, podemos poner todo el código de la prueba en un solo fichero tal como se muestra en la siguiente figura.

```
// spec.js
describe('Protractor Demo App', function() {
  it('should add one and two', function() {
    browser.get('http://juliemr.github.io/protractor-demo/');
    element(by.model('first')).sendKeys(1);
    element(by.model('second')).sendKeys(2);

    element(by.id('gobutton')).click();

    expect(element(by.binding('latest')).getText()).
        toEqual('5'); // This is wrong!
  });
});
```

Figura 17: Demo de una prueba Protractor. Fuente (11)

Leyendo ese código podemos advertir cómo la prueba consiste en sumar 1 + 2, clica en un botón y después comprueba que el resultado sea 5. La descripción de la prueba en las dos primeras líneas nos puede dar una pista de qué se está probando, pero hemos tenido que leer el resto del código, que tiene una considerable carga cognitiva para tener hacer tan poca cosa. Debido a esto podemos intuir que quizás no es la mejor manera de organizar nuestro código de pruebas en casos algo más complejos que éste.

6.1.1. Page object

El equipo de desarrollo de Protractor, consciente de este problema, expone un patrón en su documentación para mejorar la legibilidad de la prueba y encapsular su lógica para su posible reutilización más adelante. Este patrón se puede ejemplificar en la siguiente figura.

```
var AngularHomepage = function() {
  var nameInput = element(by.model('yourName'));
  var greeting = element(by.binding('yourName'));

  this.get = function() {
    browser.get('http://www.angularjs.org');
  };

  this.setName = function(name) {
    nameInput.sendKeys(name);
  };

  this.getGreetingText = function() {
    return greeting.getText();
  };
};
module.exports = new AngularHomepage();
```

Figura 18: Ejemplo de Page object para la página de AngularJS. Fuente (11)

Y se utilizaría donde fuera necesario de la manera siguiente.

```
var angularHomepage = require('./AngularHomepage');
describe('angularjs homepage', function() {
  it('should greet the named user', async function() {
    await angularHomepage.get();

    await angularHomepage.setName('Julie');

    expect(await angularHomepage.getGreetingText()).toEqual('Hello Julie!');
  });
});
```

Figura 19: Uso del Page object de ejemplo. Fuente (11)

Si leemos ahora la prueba secuencialmente, no encontramos tanto código que nos moleste a la hora de entender cómo funciona la prueba. Este es, en efecto, un gran avance con respecto a la manera básica de desarrollar una prueba, pero para algunos casos de uso donde el SUT tiene una interfaz más compleja puede quedarse corto en cuanto a la reutilización de los propios *Page object*. Por ello, quizás conviene usar otra manera de organizar el código con un grano más fino.

6.1.2. Los problemas con *Page object*

Para empezar, parece curioso el hecho de que el *framewok* de desarrollo de pruebas automáticas creado por el equipo de AngularJS proponga un patrón que abstraiga la funcionalidad de la prueba a nivel de página web o URL cuando el propio *framework* de AngularJS está orientado al desarrollo de componentes reusables en diferentes páginas web. Esto puede ocasionar que se dé el caso en el que tengas un componente medianamente complejo en la web de tu SUT usado en varias de sus páginas mientras que, en las pruebas, el código que accede a la funcionalidad de ese componente esté duplicada en varios *Page object*.

Si se escoge este patrón como convención en el desarrollo de una *suite* de pruebas, la propia convención nos estaría haciendo caer en una *Imposed Duplication* o en este caso sería más preciso llamarlo *Self-Imposed Duplication*.(6)

6.1.3. Componente

Como una opción más natural de un SUT hecho con AngularJS, se propone el uso de la abstracción *Componente* como manera de encapsular funcionalidad de interacción con el navegador.

Esta abstracción sería similar en la práctica a *Page object* con dos diferencias importantes. Tiene una granularidad más fina, es decir, no trata de encapsular la funcionalidad de una página, sino que encapsula las interacciones con subsecciones de ésta. La otra diferencia sería que la abstracción determina la URL donde se va a realizar la interacción, lo que permite que se lleve a cabo en varias páginas diferentes donde se use el mismo componente de la UI del SUT.

```
class Calendario {
    constructor(elementoRaiz) {
        this.elementoRaiz = elementoRaiz;
        this.entradaAnyo = elementoRaiz.$("input.anyo");
        this.entradaMes = elementoRaiz.$("input.mes");
        this.elementosDia = elementoRaiz.$$(".dia");
    }

    async seleccionarFecha(fecha) {
        await this.elementoRaiz.click();
        await this.entradaAnyo.sendKeys(fecha.anyo);
        await this.entradaMes.sendKeys(fecha.mes);
        await this.elementosDia.get(fecha.dia).click();
    }
}
```

Figura 20: Ejemplo de un Componente con las interacciones de un calendario

```
describe("Vista del usuario de la web del restaurante", function() {
    const calendario = new Calencario(element(by.model("diaReserva")))
    const nombre = element(by.model("nombreReseva"));
    const reservar = element(by.css("button.ok"))
    const mensaje = element(by.id("reservaOk"))

    it("debería reservar mesa", async function() {
        await browser.get(urlDeReservasUsuario);
        await nombre.sendKeys("David Olmos")
        await calendario.seleccionarFecha({
            dia: 1,
            mes: "Diciembre",
            anyo: "2019"
        });
        await reservar.click();
        expect(await mensaje.isPresent()).toBeTruthy();
    })
})
```

Figura 21: Ejemplo de prueba que usa el Componente

De esta manera, se sigue manteniendo la legibilidad de la selección de la fecha en la prueba, pero ya no está atada a la página probada. Ese mismo componente se puede reutilizar, por ejemplo, en el filtro de la lista de reservas del dueño del restaurante, que

presumiblemente tendrá una URL diferente. También se podría considerar crear un *Componente* que contuviera un *Calendario* para terminar de abstraer mejor los detalles, como introducir el nombre o clicar en el botón.

6.2. Diseño del *framework*

El diseño del *framework* gira en torno a cuatro interfaces: *Requisito*, *CasoDeUso*, *Builder* y *Componente*.

6.2.1. *Componente*

Se ha creado una interfaz componente y una clase abstracta que la implementa con el fin de añadir métodos de utilidad para facilitar el uso de esperas manuales nombradas como solución a medio plazo del problema del indeterminismo mencionado en el capítulo 3 – Análisis. Es, principalmente, una interfaz que se usa para recordar al desarrollador de la prueba que está usando un *Componente* y que lo tenga presente.

6.2.2. Encapsulación de *Requisito* y *CasoDePrueba*

Estas abstracciones nacieron para encapsular los casos de prueba como el de reservar mesa de manera que fuera parametrizable.

```
class ReservaMesa implements CasoDePrueba {
    calendario = new Calendario(element(by.model("diaReserva")));
    nombre = element(by.model("nombreReseva"));;
    reservar = element(by.css("button.ok"));
    mensaje = element(by.id("reservaOk"));

    constructor(public readonly parametroNombre: string, public readonly fecha: any) {}

    definir() {
        it("debería reservar mesa", async function () {
            await browser.get(urlDeReservasUsuario);
            await this.nombre.sendKeys(this.parametroNombre);
            await this.calendario.seleccionarFecha(this.fecha);
            await this.reservar.click();
            expect(await this.mensaje.isPresent()).toBeTruthy();
        })
    }
}
```

Figura 22: Caso de prueba de reservar mesa implementando la abstracción CasoDePrueba

De esta manera externalizamos los datos parametrizables separándolos de la lógica de la prueba y encapsulamos los detalles de implementación, tales como los elementos usados para ejecutarla. Eso nos permite definir el mismo caso de prueba con argumentos diferentes de manera más sencilla, evitando introducir duplicidades en caso de que necesitemos volver a emplearlo.

```
new ReservaMesa("David Olmos", { dia: 12, mes: "Enero", anyo: "2020" }).definir();
new ReservaMesa("Borja Davó", { dia: 19, mes: "Junio", anyo: "2020" }).definir();
new ReservaMesa("Amparo Ferrer", { dia: 2, mes: "Abril", anyo: "2021" }).definir();
```

Figura 23: Mismo CasoDePrueba definido con diferentes argumentos

Además, durante el desarrollo de casos de prueba usando el *framework* se encontró la necesidad de dejar el SUT en un estado apropiado para cada caso de prueba. Esto se modeló mediante la interfaz *Requisito*.

Al igual que la abstracción *CasoDeUso,* la abstracción *Requisito* debe separar las acciones necesarias que debe realizar para dejar el SUT en el estado correcto de los datos parametrizables usados durante el proceso.

```
class CrearUsuario implements Requisito {
    constructor(public readonly nombre: string) {}

    async ejecutar() {
        await crearUsuario(this.nombre);
        return {nombre: this.nombre}
    }
}
```

Figura 24: Implementación del requisito de tener un usuario creado

```
new CrearUsuario("David Olmos").ejecutar()
    .then(resultado => {
        new ReservaMesa(resultado.nombre, { dia: 12, mes: "Enero", anyo: "2020" }).definir()
    });
```

Figura 25: Forma normal de encadenar un Requisito con un CasoDePrueba

6.2.3. Un DSL para la composición y parametrización de flujos de usuario y su integración con el *framework*

Como se puede observar en la Figura 25, encadenar un *Requisito* con un *CasoDePrueba* se puede complicar rápidamente si nos ponemos a encadenar varios requisitos previos debido a la naturaleza asíncrona que tiene insertar datos en una base de datos o enviar datos al servidor del SUT.

El objetivo del DSL consistiría en encapsular esa sintaxis tan poco amigable en otra que se abstraiga hasta los conceptos de flujo de usuario y sus parámetros.

```
// Caso de prueba de reservar mesa
CrearUsuarioRequisito.nombre("David Olmos")
    .entonces(RegistrarVisaRequisito).visa("ES000000000")
    .finalmente(ReservarMesaPrueba).dia(18).mes("Abril").anyo("2020")
```

Figura 26: Definición del caso de prueba de ReservarMesa y sus requisitos mediante el DSL

Así pues, al ejecutarse la definición, debería hacerlo de manera secuencial de arriba hacia abajo. En este ejemplo, primero crearía un usuario llamado David Olmos, después le añadiría a ese usuario una tarjeta de crédito con ese número y procedería a ejecutar el caso de prueba de *ReservarMesa* con el nombre parametrizado en *CrearUsuarioRequisito*.

6.2.3.1. Parametrización compleja usando el patrón Builder

El patrón de diseño *Builder* se usa a menudo para hacer más legible la construcción de objetos o estructuras de datos complejas. (12)

```
class ReservarMesaBuilder extends BaseBuilder implements Builder {
    _nombre: string;
    _fecha: any;
    nombre(valor: string): this {
        this._nombre = valor;
        return this;
    }

    dia(valor: number): this {
        if (!this._fecha) this._fecha = {};
        this._fecha.dia = valor;
        return this;
    }

    mes(valor: string): this {
        if (!this._fecha) this._fecha = {};
        this._fecha.mes = valor;
        return this;
    }

    anyo(valor: string): this {
        if (!this._fecha) this._fecha = {};
        this._fecha.anyo = valor;
        return this;
    }

    private fechaCompleta(): boolean {
        return !this._fecha || (this._fecha.dia && this._fecha.mes && this._fecha.anyo);
    }

    build() {
        if (!this.fechaCompleta()) throw Error("Has introducido una fecha a medias");
        return new ReservaMesa(this._nombre, this._fecha);
    }
}
```

Figura 27: Ejemplo de un Builder para el caso de prueba ReservaMesa

En la Figura 26, *CrearUsuarioRequisito*, *RegistrarVisaRequisito*, *ReservarMesaPrueba* son variables que contienen una instancia de una clase similar a la de la Figura 27.

6.2.4. Autoconfiguración mediante un bus de eventos

Hasta ahora, como se puede observar en la Figura 24, la única comunicación que se ha visto entre elementos diferentes del flujo ha sido en una sola dirección, desde el primer requisito hacia el caso de prueba, pero hemos de tener en cuenta también que la especificación del caso de prueba puede necesitar que alguno de los requisitos previos sea parametrizado con un conjunto de datos determinado.

Por ejemplo, digamos que la aplicación de reserva de mesas no debería dejar reservar si se tiene un nombre que esté vetado por el restaurante. Dada esta situación, podríamos usar un flujo parecido al de la Figura 26 pero cambiando el nombre de *CrearUsuarioRequisito* por un nombre vetado. En principio con eso sería suficiente, pero si repasamos la motivación de crear el DSL, ¿no era abstraer los detalles sobre la

ejecución de los requisitos y el caso de prueba del flujo? Si lo hiciéramos así estaríamos añadiendo un argumento que filtraría detalles de la ejecución del caso de prueba en la definición del flujo.

Para conseguir comunicar los elementos del flujo en dirección ascendente, éstos deberían tener la referencia del elemento anterior formando una estructura de lista enlazada. Esto es engorroso de implementar y se optó por una solución alternativa.

La solución consiste en declarar dos nuevas interfaces, *Configurador* y *Configurable*. Los elementos que implementen *Configurable* tendrán una referencia a un emisor de eventos con el que se podrán subscribir a eventos de configuración. Es importante que los elementos configurables se subscriban a los eventos de configuración lo antes posible, siendo lo óptimo en el constructor. Los elementos que implementen la interfaz *Configurador* enviarán eventos de configuración dentro de un método que deberán implementar.

```
0 implementaciones
interface Configurador {
    configurar(emisorEventos: EventEmitter);
}

0 implementaciones
interface Configurable {
    emisorEventos: EventEmitter;
}
```

Figura 28: Interfaces de configuración

Se creará una sola instancia del emisor de eventos simultáneamente para todos los elementos del mismo flujo para garantizar la comunicación entre ellos, pero no se compartirá entre flujos diferentes pues no queremos que un flujo de prueba influya en otro flujo de prueba diferente.

Capítulo 7. Conclusiones

7.1. Conclusiones

Llegados a este momento, creemos se pueden señalar algunas conclusiones positivas respecto a los objetivos marcados.

Hemos analizado porqué los principios SOLID no eran fácilmente aplicables a las *suites* de pruebas realizadas con Protractor y porqué la abstracción propuesta en la documentación de éste framework para la encapsulación de la lógica de pruebas, el patrón *Page object*, no era suficiente para algunas aplicaciones web como era nuestro caso. Después se ha propuesto una solución en forma de *framework* ad-hoc basado en Protractor y se ha implementado la *suite* sobre él.

En lo que respecta a la conclusión sacada del desarrollo de la herramienta de apoyo, se han podido desarrollar todas las funcionalidades descritas con éxito con alguna excepción. Para el problema de la grabación en video de la ejecución de las pruebas descrito en el capítulo 5 no se ha encontrado una solución completa. También sigue quedando pendiente la búsqueda de una solución más permanente al problema del determinismo en las pruebas.

7.2. Problemas encontrados

El principal problema ha sido la velocidad de ejecución de las pruebas, algo inherente a las herramientas de simulación de eventos de usuario en navegadores como Selenium. La *suite* entera con sus doce casos de uso tardaba unos veinte minutos en ejecutarse. Cada vez que se probaba un cambio importante se invertía mucho tiempo en su comprobación por lo que se aumentaba en exceso el tiempo empleado en el ciclo de desarrollo del *framework*.

Otro problema más liviano es la dificultad ya comentada de desplegar la base de datos del SUT limpia. Esto ha llevado a usar una base de datos común con el equipo de QA durante gran parte del trabajo, lo que ocasionaba falsos positivos porque alguien estaba haciendo pruebas que interferían con las ejecuciones en curso de la *suite.*

7.3. Errores cometidos

Crear la *suite* de pruebas y el *framework* ad-hoc en un repositorio del sistema de control de versiones diferente al SUT ha podido jugar en contra de la mantenibilidad de la propia *suite* ya que si estuviera en el mismo repositorio que el SUT quizás el equipo de desarrollo se acordaría más a menudo de ella tanto como para usarla como para mantenerla. Además, la *suite* podría actualizarse mejor acoplada a una versión del SUT.

El otro error que creo que se ha cometido fue no tratar de investigar más a fondo si en lugar de crear un DSL nuevo se hubiese integrado el *framework* ad-hoc a Cucumber que podría hacer los flujos de usuario todavía más legibles. Así pues, este es un tema propuesto en el capítulo posterior como un futuro desarrollo relacionado con este trabajo.

7.4. Conocimiento adquirido

He ampliado mi conocimientos sobre pruebas de software y de las herramientas que hay en la actualidad para automatizarlas, sobre todo en entornos con GUI.

He adquirido confianza en el desarrollo de pruebas automatizadas usando Protractor y con el lenguaje de programación TypeScript.

He aprendido a dominar el patrón de diseño *Builder* por su abundante uso para desarrollar el DSL.

He profundizado en mis conocimientos de diseño de software con el descubrimiento de nueva literatura sobre el tema.

7.5. Relación con los estudios cursados

Diseño de Software es la asignatura que más me ha ayudado a realizar este trabajo pues cada vez que me encontraba con un problema de diseño de clases recordaba los patrones de diseño de software que se enseñan en esta asignatura. También me descubrió el mundo de la literatura sobre diseño de software y gracias a eso descubrí (3) y (6).

Aun así, la inspiración del tema de este trabajo se lo debo al tema de introducción de la asignatura de Mantenimiento del Software pues es la que me inculcó la importancia de desarrollar software mantenible y en general de la fase de mantenimiento ya que la mayoría de trabajo en nuestro campo hoy en día consiste en mantener software. Como muy acertadamente se comenta en (6), el mantenimiento no es la fase final del software ya desarrollado sino que es un estado mental en el que se encuentran los desarrolladores todo el tiempo que pasan programando.

Respecto a las competencias transversales aplicadas en este trabajo creo que he aplicado las siguientes:

- Comprensión e integración porque he necesitado entender cómo funcionaba Protractor a nivel interno para poder entender, refactorizar y modificar el *plugin* que integra la aplicación con la *suite*.
- Aplicación y pensamiento crítico para diseñar soluciones a problemas que se me presentaban en el contexto de las prácticas en empresa a la hora de mantener la *suite* de pruebas.
- Aprendizaje permanente por la necesidad de investigar por mi cuenta herramientas desconocidas previamente como puede ser TypeScript, NestJS y Protractor o sobre nuevos principios de desarrollo de software como DRY en literatura sobre desarrollo de software.
- Y por finalizar, quizás la más evidente, la competencia de Instrumental específica.

Capítulo 8. Trabajos futuros

8.1. Desarrollo de funcionalidad deseable en la aplicación

8.1.1. Marcar falsos positivos

Sería conveniente disponer de una funcionalidad en la aplicación que permitiese marcar fallos en la ejecución de las pruebas como falsos positivos y añadir las otras a un backlog para revisar más adelante.

8.1.2. Gráficas y estadísticas que exploten mejor los datos recogidos

Esta funcionalidad unida a la anterior permitiría visualizar en que parte de la aplicación surgen más problemas y cuales han sido las versiones del SUT que han provocado más fallos inesperados.

8.1.3. Reproductor de las pruebas GUI en video

Mejoraría la usabilidad de la aplicación de apoyo a las pruebas si se pudiera reproducir el video de la grabación sin salirse del navegador.

8.1.4. Usar SQL en lugar de NoSQL

Se escogió una base de datos MongoDB para experimentar con el modelo de datos requerido para la aplicación, pero realmente cuando se terminó no requería de la escalabilidad ni de la flexibilidad de una base de datos NoSQL, pero se mantuvo porque ya estaba funcionando. Además, tal como se desarrolló no requiere propiedades ACID en sus transacciones ni consistencia en los datos, pero podría ser deseable si los datos se pueden modificar una vez terminada la ejecución como se podría implementar en la propuesta del punto 8.1.1 – Marcar falsos positivos.

8.1.5. Mejorar la integración con otros *framework* de pruebas

Actualmente la integración con el *framework* de pruebas acopla las pruebas al código del servidor de la aplicación de apoyo. Por tanto, también restringe la suite de pruebas al lenguaje en el que está desarrollado el servidor, es decir, JavaScript. Si se deseara integrar la aplicación con una *suite* de pruebas que usara otro lenguaje de programación, como Java o C#, sería imposible. La propuesta de este punto es definir una interfaz mejor para integrarse con la *suite* por HTTP, consiguiendo así que la integración sea independiente del lenguaje de programación.

Se podría adaptar el CRUD del servidor de la aplicación de apoyo a las pruebas para que fuera usado también por el *plugin* a la hora de guardar los metadatos de la ejecución de la *suite* de pruebas.

8.2. Mejoras en el *framework* ad-hoc

8.2.1. Mejorar la grabación de pruebas dentro de Docker

A la finalización de este trabajo se han detectado problemas al ejecutar la *suite* de pruebas dentro de un contenedor Docker con la grabación en video activada. Como la *suite* está destinada a pruebas de regresión ejecutadas por el desarrollador no obstaculiza, pero si se deseara lanzar en algún entorno de integración continua se deberían corregir estos problemas.

8.2.2. Integración con otros *frameworks*

El DSL creado para facilitar la parametrización de los casos de prueba podría tratar de integrarse con el *framework* Cucumber[24].

Ya que el DSL está inspirado en la sintaxis Gherkin[25] usada por Cucumber podría ser interesante explorar las formas de traducir las especificaciones hechas para Cucumber en su correspondiente DSL o buscar la forma de integrarlo con el *framework* creado durante este trabajo para facilitar la lectura de la especificación de los casos de prueba a todos los *stakeholders* implicados en el proceso de pruebas.

[24] Cucumber - https://cucumber.io/docs

[25] Sintaxis Gherkin - https://cucumber.io/docs/gherkin/reference/

Capítulo 9. Referencias

1. BOLAÑOS ALONSO, Daniel. *Pruebas de software y JUnit : un análisis en profundidad y ejemplos prácticos*. book. Madrid : Pearson Educación, 2008. ISBN 9788483223543.

2. ALÉGROTH, Emil, FELDT, Robert and KOLSTRÖM, Pirjo. Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. *Information and Software Technology* [online]. 1 May 2016. Vol. 73, p. 66–80. [Accessed 27 November 2019]. DOI 10.1016/j.infsof.2016.01.012. Available from: https://www.sciencedirect.com/science/article/abs/pii/S0950584916300118

3. MARTIN, RC. *Agile software development: principles, patterns, and practices*. 2002. ISBN 978-1292025940.

4. MARTIN, RC. Getting a SOLID start. - Clean Coder. *02/12/2009* [online]. 2009. [Accessed 28 November 2019]. Available from: https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start

5. MARTIN, RC. Principles of OOD. [online]. 2005. [Accessed 29 November 2019]. Available from: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

6. HUNT, Andrew. *The pragmatic programmer : from journeyman to master*. book. Reading, Mass : Addison-Wesley, 2000. ISBN 9780201616224.

7. EVILWEED, BCOLE and JINTOPPY. Evilweed/protractor-beautiful-reporter: An npm module which provides html reports of your Protractor tests with screenshots. [online]. [Accessed 29 November 2019]. Available from: https://github.com/Evilweed/protractor-beautiful-reporter

8. MOZILLA. MVC - Glosario | MDN. [online]. [Accessed 29 November 2019]. Available from: https://developer.mozilla.org/es/docs/Glossary/MVC

9. MONGODB INCORPORATED. Install MongoDB. [online]. [Accessed 29 November 2019]. Available from: https://docs.mongodb.com/guides/server/install/

10. GOOGLE. AngularJS: API: ngRepeat. [online]. [Accessed 30 November 2019]. Available from: https://docs.angularjs.org/api/ng/directive/ngRepeat

11. ANGULAR TEAM. Protractor Tutorial. [online]. [Accessed 30 November 2019]. Available from: https://www.protractortest.org/#/tutorial

12. ELGAZAR, Itay. The Builder Pattern in Node.js & Typescript. [online]. [Accessed 30 November 2019]. Available from: https://medium.com/@itayelgazar/the-builder-pattern-in-node-js-typescript-4b81a70b2ea5

Capítulo 10. Anexo

10.1. Código de ejemplo

A continuación se presenta el código de ejemplo desarrollado exprofeso para ilustrar el capítulo 6 – SOLID y DRY mediante un *framework* ad-hoc.

El código aquí presente se puede compilar con la última versión de TypeScript pero no está completo y por tanto no funciona, pues tiene un objetivo ilustrativo.

```
import EventEmitter from "events";
type ProtractorElement = any;

interface Requisito {
    ejecutar(resultadosPrevios: any): Promise<any>;
}

interface CasoDePrueba {
    definir(): void;
}

interface Configurador {
    configurar(emisorEventos: EventEmitter);
}

interface Configurable {
    emisorEventos: EventEmitter;
}

interface Builder {
    entonces<T extends Builder>(siguiente: T): T;
    finalmente<T extends Builder>(ultimo: T): T;
    build(): any;
}

// Dummy implementation
abstract class BaseBuilder implements Builder {
    entonces<T extends Builder>(siguiente: T) {
        return siguiente;
    }

    finalmente<T extends Builder>(prueba: T) {
        return prueba;
    }

    abstract build(): any;
}

interface Componente {
```

```
    elementoRaiz: ProtractorElement;
}

// Dummy implementation
abstract class BaseComponente implements Componente {
    elementoRaiz: any;
}

class Calendario extends BaseComponente implements Componente {
    entradaAnyo;
    entradaMes;
    elementosDia;

    constructor(public readonly elementoRaiz: ProtractorElement) {
        super();
        this.elementoRaiz = elementoRaiz;
        this.entradaAnyo = elementoRaiz.$("input.anyo");
        this.entradaMes = elementoRaiz.$("input.mes");
        this.elementosDia = elementoRaiz.$$(".dia");
    }

    async seleccionarFecha(fecha) {
        await this.elementoRaiz.click();
        await this.entradaAnyo.sendKeys(fecha.anyo);
        await this.entradaMes.sendKeys(fecha.mes);
        await this.elementosDia.get(fecha.dia).click();
    }
}

class CrearUsuario implements Requisito {
    constructor(public readonly nombre: string) {}

    async ejecutar() {
        await crearUsuario(this.nombre);
        return {nombre: this.nombre}
    }
}

class RegistrarVisa implements Requisito {
    constructor(public readonly visa: string) { }
    async ejecutar({nombre}) {
        await registrarVisa(nombre, this.visa);
    }
}

class ReservaMesa implements CasoDePrueba {
    calendario = new Calendario(element(by.model("diaReserva")));
    nombre = element(by.model("nombreReseva"));;
    reservar = element(by.css("button.ok"));
```

```
    mensaje = element(by.id("reservaOk"));

    constructor(public readonly parametroNombre: string, public readonl
y fecha: any) {}

    definir() {
        it("debería reservar mesa", async function () {
            await browser.get(urlDeReservasUsuario);
            await this.nombre.sendKeys(this.parametroNombre || this.res
ultados.nombre);
            await this.calendario.seleccionarFecha(this.fecha);
            await this.reservar.click();
            expect(await this.mensaje.isPresent()).toBeTruthy();
        })
    }
}

class ReservarMesaBuilder extends BaseBuilder implements Builder {
    _nombre: string;
    _fecha: any;
    nombre(valor: string): this {
        this._nombre = valor;
        return this;
    }

    dia(valor: number): this {
        if (!this._fecha) this._fecha = {};
        this._fecha.dia = valor;
        return this;
    }

    mes(valor: string): this {
        if (!this._fecha) this._fecha = {};
        this._fecha.mes = valor;
        return this;
    }

    anyo(valor: string): this {
        if (!this._fecha) this._fecha = {};
        this._fecha.anyo = valor;
        return this;
    }

    private fechaCompleta(): boolean {
        return !this._fecha || (this._fecha.dia && this._fecha.mes && t
his._fecha.anyo);
    }

    build() {
```

```
        if (!this.fechaCompleta()) throw Error("Has introducido una fec
ha a medias");
        return new ReservaMesa(this._nombre, this._fecha);
    }
}

class CrearUsuarioBuilder extends BaseBuilder implements Builder {
    _nombre: string;
    nombre(valor: string): this {
        this._nombre = valor;
        return this;
    }

    build() {
        if (!this._nombre) throw Error("Falta el nombre en el registro
de usuario");
        return new CrearUsuario(this._nombre);
    }
}

class RegistrarVisaBuilder extends BaseBuilder implements Builder {
    _numeroVisa: string;
    visa(valor: string): this {
        this._numeroVisa = valor;
        return this;
    }

    build() {
        if (!this._numeroVisa) throw Error("Falta el numero de tarjeta
de crédito");
        return new RegistrarVisa(this._numeroVisa);
    }
}

declare const CrearUsuarioRequisito: CrearUsuarioBuilder;
declare const RegistrarVisaRequisito: RegistrarVisaBuilder;
declare const ReservarMesaPrueba: ReservarMesaBuilder;

// Caso de prueba de reservar mesa
CrearUsuarioRequisito.nombre("David Olmos")
    .entonces(RegistrarVisaRequisito).visa("ES000000000")
    .finalmente(ReservarMesaPrueba).dia(18).mes("Abril").anyo("2020")

declare const browser: any;
declare const by: any;
declare const urlDeReservasUsuario: any;
declare function expect(check: any): any;
declare function element(locator: any): any;
declare function it(description: string, block: () => void)
```

```
declare function crearUsuario(nombre: string): Promise<void>;
declare function registrarVisa(nombre: string, numeroVisa: string): Pro
mise<void>;
declare function funcionGeneradoraNombres(): string;
declare function funcionGeneradoraVisa(): string;
```