



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



TFG del Grado en Ingeniería Informática

TestingAWE



Sistema para generar test
automáticos End-to-end de aplicaciones
Angular/Web/Electron

Presentado por Mario Ubierna San Mamés

en la Universidad de Burgos – 3 de julio de 2019

Tutor Universidad de Burgos: Dr. Raúl Marticorena Sánchez

Tutores Observatorio HP: Alicia Gago Pérez,

David Calvo Duarte,

Juan Manuel Zamarreno Pérez



UNIVERSIDAD DE BURGOS

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática



D. Raúl Marticorena Sánchez, profesor del departamento de Ingeniería Civil, área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. Mario Ubierna San Mamés, con DNI XXXXXXXXXX, ha realizado el Trabajo Final de Grado en Ingeniería Informática titulado “TestingAWE - Sistema para generar test automáticos End-to-end de aplicaciones Angular/Web/Electron”.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 3 de julio de 2019

Vº.Bº. del Tutor:

D. Raúl Marticorena Sánchez

Resumen

La actividad de testing es un indicador importante sobre la robustez de un proyecto software. Sin embargo, este proceso puede llegar a ser costoso si no se tiene los conocimientos adecuados y puede originar que el proyecto tenga una tasa de error elevada.

Hasta día de hoy, todo este proceso se tenía que realizar de forma manual. Además, se necesitaba una gran curva de aprendizaje ya que se tenían que aprender a utilizar las diferentes tecnologías.

En este trabajo se propone una nueva herramienta más accesible al público, que permita facilitar la estructura básica de testing en un proyecto software de forma automática, y que simplifique el uso de librerías externas para el desarrollo de los test.

Esta herramienta solo puede ser utilizada para aplicaciones de escritorio, pero no para todas ellas, ya que está ideada y desarrollada para aplicaciones Angular/Electron, las cuales son muy demandadas hoy en día.

TestingAWE es la *toolkit* resultante y se encuentra disponible a través de Node Package Manager <https://www.npmjs.com/package/testing-awe>.

Descriptores

Estructura de testing, automatización, simplificación de test, identificación de elementos HTML, módulo NPM, aplicaciones de escritorio Angular/Electron.

Abstract

The testing activity is an important indicator about the robustness of a software project. However, this process can become expensive if you do not have the proper knowledge and can cause the project to have a high error rate.

To this day, all this process had to be done manually. In addition, a great learning curve was needed since they had to learn to use different technologies.

In this work we propose a new toolkit more accessible to the public, that allows to facilitate the basic structure of testing in a software project automatically, and that simplifies the use of external libraries for the development of the tests.

This tool can only be used for desktop applications, but not for all of them, since it is designed and developed for Angular / Electron applications, which are in great demand today.

TestingAWE is the resulting toolkit and is available through Node Package Manager <https://www.npmjs.com/package/testing-awe>.

Keywords

Test structure, automation, test simplification, identification of HTML elements, NPM module, Angular/Electron desktop applications.

Índice de Contenido

Índice de Contenido	7
Índice de tablas	9
Índice de ilustraciones.....	10
1. Introducción.....	11
1.1. Estructura de la memoria.....	14
1.2. Materiales adjuntos.....	15
2. Objetivos del proyecto.....	16
2.1. Objetivos generales	16
2.2. Objetivos técnicos.....	16
2.3. Objetivos personales	17
3. Conceptos teóricos	18
4. Técnicas y herramientas.....	19
4.1. Metodologías	19
4.2. Patrones de diseño.....	21
4.3. Control de versiones.....	22
4.4. Hosting del repositorio.....	22
4.5. Gestión del proyecto	23
4.6. Gestión del repositorio.....	23
4.7. Comunicación.....	23
4.8. Editor de código.....	24

4.9.	Documentación	24
4.10.	Frameworks y librerías.....	25
4.11.	Otras herramientas.....	27
5.	Aspectos relevantes del desarrollo del proyecto.....	28
5.1.	Inicio del proyecto	28
5.2.	Metodologías.....	29
5.3.	Formación.....	30
5.4.	Pre-Desarrollo	31
5.5.	Desarrollo de la librería	31
5.6.	Desarrollo de la automatización.....	35
5.7.	Testing	38
5.8.	Publicación.....	38
6.	Trabajos relacionados.....	40
7.	Conclusiones y Líneas de trabajo futuras.....	44
7.1.	Conclusiones	44
7.2.	Líneas de trabajo futuras.....	45
8.	Bibliografía	46

Índice de tablas

Tabla 1 - Tablero Kanban	20
Tabla 2 - Generación del fichero para elemento estático.....	37
Tabla 3 - Generación del fichero para elemento dinámico.....	37
Tabla 4 - Comparativa de los trabajos	40

Índice de ilustraciones

Ilustración 1 - Funcionamiento de TestingAWE	13
Ilustración 2 - Patrón Page Object Model.....	18
Ilustración 3 - Patrón MVC.....	21
Ilustración 4 - Patrón Singleton.....	22
Ilustración 5 - Uso de WebdriverIO sin TestingAWE	32
Ilustración 6 - Uso de WebdriverIO con TestingAWE	33
Ilustración 7 - Inicialización, arranque y cierre sin TestingAWE	34
Ilustración 8 - Inicialización, arranque y cierre con TestingAWE.....	34
Ilustración 9 - Uso de Spectron sin TestingAWE	34
Ilustración 10 - Uso de Spectron con TestingAWE	35
Ilustración 11 - Fichero generado por TestingAWE	36
Ilustración 12 - Módulo TestingAWE en NPM	39
Ilustración 13 - Desarrollo de test sin TestingAWE	42
Ilustración 14 - Desarrollo de test con TestingAWE	43

1. Introducción

El *testing* es una pieza clave en el desarrollo del software, siempre y cuando queramos obtener una alta calidad en nuestro proyecto. Antes de empezar vamos a comentar, ¿qué es el *testing*?

Son numerosas las definiciones, sin embargo hay una que mejor lo define:

“Software testing consists of dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior” [1].

En resumidas palabras, el testing no es más que un proceso para poder validar y verificar el comportamiento de un software, es decir, asegurarnos que realmente el programa hace lo que tiene que hacer bajo unas determinadas condiciones.

Hoy en día, el decidir si dedicar gran parte del presupuesto a hacer pruebas viene determinado por el coste que conlleva realizarlas. Para hacernos una idea, cuando el proyecto tiene restricciones en el tiempo, el coste de la calidad representa un 57,3% del coste total del proyecto, sin embargo, si no hay restricciones representa un 51,5% [2].

Al observar los datos anteriores podemos ver que son porcentajes muy altos, esto se debe a varios motivos, el primero es que no hay una automatización de la estructura de las pruebas, el segundo que el usuario tiene que aprender a manejar diferentes librerías, es decir, utilizarlas de una forma correcta, esto supone un gran coste.

En este trabajo se propone un método para poder reducir el coste del testing tanto en tiempo como en dinero, a través de una *toolkit*, en la cual vamos a poner solución a la automatización de la estructura de pruebas, es decir, de una forma

muy simple se va a poder generar todos los elementos necesarios para poder realizar pruebas sobre una aplicación de escritorio. Además, vamos a simplificar la sintaxis a la hora de realizar test gracias a una nueva librería, de tal forma que cualquier programador de una forma sencilla pueda hacer uso de ella.

TestingAWE pone solución a los problemas descritos. Dicha *toolkit* podría revolucionar el campo del desarrollo del software, ya que lo hace accesible para la comunidad de los programadores tanto *juniors* como *seniors*. No es necesario tener un *hardware* muy costoso para hacer uso de esta herramienta. Además, para poder utilizarla solo tenemos que instalar el módulo TestingAWE. Esto hace que su instalación sea muy fácil.

En la siguiente ilustración podemos observar el funcionamiento de TestingAWE:

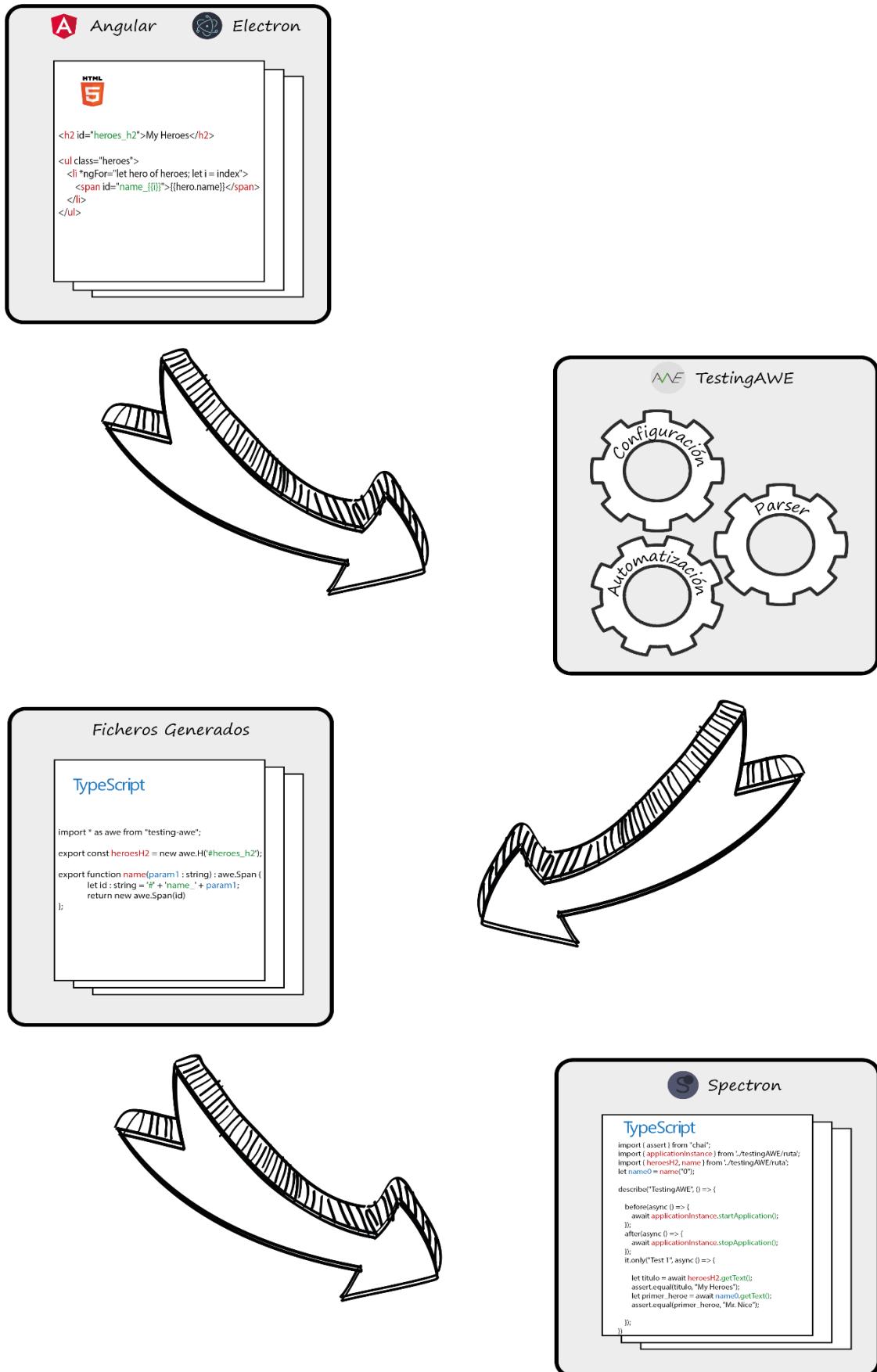


Ilustración 1 - Funcionamiento de TestingAWE

Como podemos ver en la anterior ilustración, partimos de una aplicación de escritorio desarrollada con Angular/Electron.

TestingAWE lo que va a realizar es, por cada fichero HTML que se encuentra en ese proyecto Angular/Electron, va a generar un archivo. Este archivo va a contener los elementos HTML que se han detectado en el fichero. El proceso descrito se realiza de forma automática a partir de una configuración inicial, y gracias al parseo vamos a poder obtener los elementos HTML mencionados.

Una vez que TestingAWE ha generado dichos ficheros, vamos a poder hacer uso de éstos para el desarrollo de los tests. Además, los elementos HTML que se encuentran en los ficheros generados nos van a permitir hacer uso de la librería de TestingAWE. Ésta nos permite simplificar el desarrollo de los tests, gracias a la reducción en la sintaxis de librerías externas.

Para resumir, TestingAWE nos va a proporcionar todo lo necesario para poder comenzar y realizar nuestras pruebas de una forma rápida, simple y sencilla.

1.1. Estructura de la memoria

La memoria presenta la siguiente estructura:

- **Introducción:** concreto resumen del problema a resolver y la solución que se propone. Además de la estructura de la memoria y el material adjunto.
- **Objetivos del proyecto:** definición de los objetivos tanto personales como generales.
- **Conceptos teóricos:** breve explicación de los aspectos más relevantes a nivel teórico del proyecto.
- **Técnicas y herramientas:** resumen de las técnicas metodológicas y herramientas usadas para la gestión y desarrollo de la *toolkit*.
- **Aspectos relevantes del desarrollo:** breve explicación sobre los aspectos más importantes en el desarrollo del proyecto.
- **Trabajos relacionados:** proyectos relacionados a la herramienta que hemos desarrollado.
- **Conclusiones y líneas de trabajo futuras:** resumen sobre las opiniones en el desarrollo del software, y el cómo se puede mejorar o ampliar el proyecto.

Además de la memoria, se proporciona los anexos:

- **Plan del proyecto del software:** planificación que hemos seguido en el desarrollo del proyecto y estudio de viabilidad.
- **Especificación de requisitos del software:** en ella se define la fase de análisis, objetivos, y requisitos de la *toolkit*.
- **Especificación de diseño:** se define el diseño de datos, de procedimientos, y arquitectónico de la aplicación.
- **Manual del programador:** ayuda para el usuario, es decir, estructura, compilación, instalación, uso de la *toolkit*...
- **Manual del usuario:** guía para el uso de la herramienta.

1.2. Materiales adjuntos

Además de toda la documentación, se proporciona:

- Módulo TestingAWE.
- Batería de pruebas.
- TestingAWE en Node Package Manager [3].
- Repositorio de la *toolkit* [4].
- Repositorio Azure DevOps (gestión de proyecto) [5].
- Videos de TestingAWE [6].

2. Objetivos del proyecto

En este apartado, se definen tanto los objetivos generales y técnicos del proyecto como los personales.

2.1. Objetivos generales

- Desarrollar una *toolkit* que permita automatizar la creación de la estructura de pruebas, y que permita desarrollar los test de una forma más fácil.
- Facilitar la creación de la estructura de pruebas de una forma eficiente y estructurada, para que así sea más cómodo para el usuario.
- Crear una librería que simplifique los test, para así facilitar la sintaxis del desarrollo de pruebas. Dicha librería tiene que tener una organización estructurada para no tener código duplicado.
- Crear una amplia documentación, para que así el usuario tenga una mayor facilidad a la hora de utilizar la *toolkit*.

2.2. Objetivos técnicos

- Desarrollar una aplicación web en Angular para así poder adquirir conocimientos en dicho campo.
- Transformar la aplicación web en una aplicación de escritorio utilizando Electron.
- Aprender a manejar el framework de pruebas Spectron, para así poder hacer los test.
- Utilizar Mocha para la estructura de los ficheros de pruebas.
- Utilizar WebdriverIO como librería que nos permita “manejar” los elementos HTML.
- Aprender a utilizar TypeScript para el desarrollo de la *toolkit*.

- Aplicar el patrón de diseño MVC (Modelo-Vista-Controlador) para hacer uso de las librerías externas.
- Utilizar Git como sistema de control de versiones junto con la plataforma GitLab.
- Utilizar Azure DevOps como herramienta de gestión de proyectos.
- Realizar el desarrollo de la aplicación a través de la metodología Scrum.
- Desarrollar pruebas para comprobar que la *toolkit* funciona correctamente, haciendo uso de una aplicación Angular/Electron.
- Distribuir la aplicación en Node Package Manager.

2.3. Objetivos personales

- Mejorar el proceso del *testing*, reduciendo tiempo y dinero gracias a la *toolkit*.
- Adquirir conocimientos sobre metodologías, herramientas, librerías, y tecnologías nuevas que tienen una gran salida laboral.
- Desarrollar la *toolkit* en TypeScript, ya que es un lenguaje que se demanda mucho junto con Angular.
- Aplicar los conocimientos que he adquirido durante la carrera.

3. Conceptos teóricos

En este apartado se va a explicar el principal concepto teórico sobre el que se basa el desarrollo de la *toolkit*, y éste es el patrón Page Object Model.

Page Object Model

El patrón POM nos permite representar cada una de las pantallas que componen el sitio web o la aplicación, como un conjunto de objetos que encapsulan las características y funcionalidades representadas en la página [7] [8].

En otras palabras, POM nos permite separar el comportamiento de una página de su implementación, es decir, encapsular en una clase las diferentes funcionalidades que puede haber respecto a una página.

En resumen, aplicando este patrón vamos a tener un Page Object por cada página HTML que tenga la aplicación, dicho Page Object va a contener las funcionalidades de la página, es decir, los diferentes comportamientos que tienen los elementos HTML.

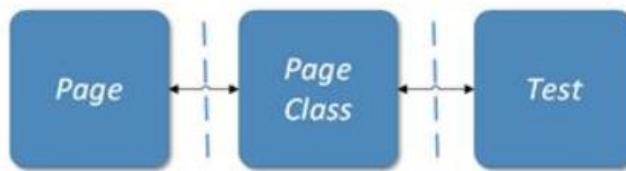


Ilustración 2 - Patrón Page Object Model

Este patrón lo que nos permite definir es la arquitectura de los ficheros que genera TestingAWE (la cual veremos más adelante), ya que esta *toolkit* genera un fichero por cada HTML que encuentra en la aplicación Angular/Electron, y dicho fichero contiene todos los elementos que hay dentro de esa página, es decir, nos permite agrupar los comportamientos de una página HTML.

4. Técnicas y herramientas

4.1. Metodologías

Scrum

Scrum es un marco de trabajo para el desarrollo software, el cual pertenece a la metodología ágil, se caracteriza por hacer uso de un desarrollo iterativo e incremental a través de iteraciones (*sprints*). Tiene como objetivo proporcionar el mejor resultado de un proyecto, y para ello hay que realizar entregas parciales del producto al final de cada *sprint* [9] [10].

Gitflow

Gitflow es una metodología para el desarrollo de repositorios Git, con el objetivo de gestionar de una forma más eficiente las ramas. Esta metodología se centra en el uso de ramas dependiendo de la tarea que estemos realizando o los problemas que nos vamos encontrando.

Gitflow define que para gestionar de una forma más eficiente hay que dividir el repositorio en dos ramas: la rama principal, y la de soporte [11] [12].

En nuestro caso hemos usado una simplificación de esta forma de trabajo, es decir, solo hemos utilizado la rama principal, ya que para las dimensiones de este proyecto usar dicha rama simplificó la gestión del repositorio. Esta rama se divide en dos subramas:

- Master: es la rama principal de nuestro proyecto, la cual contiene el código que se puede utilizar, es decir, el que se puede desplegar a producción. Esta rama no se puede modificar.
- Develop: esta rama es la secundaria, y en ella hemos estado realizando el desarrollo de la aplicación.

Kanban

Para la organización de las tareas que teníamos que hacer en cada *sprint* hemos usado el tablero Kanban que proporciona la web Azure DevOps [13].

Dicha aplicación divide las tareas en cuatro bloques: nuevos, activos, resueltos y cerrados. En nuestro proyecto hemos usado tres de ellos que han sido: nuevos, activos y cerrados.

Nuevos	Activos	Cerrados
Aquellas tareas que tenemos que realizar	Las actividades que estamos haciendo actualmente.	Las tareas que han sido hechas.

Tabla 1 - Tablero Kanban

Para estimar las tareas se ha utilizado la técnica de planning poker, en el cual los miembros del proyecto (Raúl Marticorena, el equipo de HP, y yo) hemos estimado el esfuerzo de forma consensuada, para ello expresábamos todos la opinión de lo que podíamos tardar y establecíamos el esfuerzo según la serie Fibonacci [14].

Técnica Pomodoro

Para aumentar la productividad hemos seguido al técnica pomodoro [15], la cual establece una división del tiempo en intervalos, hay tres tipos de intervalos:

- De 25 minutos, en los cuales hay que realizar el trabajo planificado sin ninguna distracción.
- De 5 minutos de descanso tras los 25 minutos de trabajo.
- De 20 a 30 minutos de descanso tras cuatro periodos de 25 minutos de trabajo.

4.2. Patrones de diseño

En este proyecto hemos hecho uso de tres patrones, el Modelo Vista Controlador (MVC), el patrón Singleton, y el patrón Page Object Model.

Modelo Vista Controlador (MVC)

El Modelo-Vista-Controlador es un patrón de diseño software, el cual nos permite separar los datos y la lógica de la aplicación de la lógica de la vista [16] [17]. Este patrón tiene tres conceptos:

- **Modelo:** se encarga del manejo de los datos, aunque su principal uso suele ser acceder a la base de datos, no tiene por qué.
- **Controlador:** es el intermediario, ya que se encarga de pedir los datos al modelo y de dárselos a la vista.
- **Vista:** no es más que la representación de los datos, es decir, la parte gráfica.

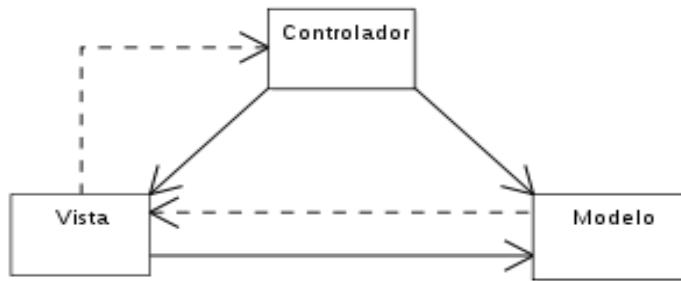


Ilustración 3 - Patrón MVC

Patrón Singleton

El patrón de diseño Singleton es uno de los patrones más sencillos y usados del desarrollo software, tiene como objetivo garantizar que solo hay una única instancia [18]. Para poder aplicar dicho patrón debemos realizar los siguientes puntos:

- El constructor de la clase tiene que ser privado, para asegurarnos que solo puede haber una instancia de la aplicación.
- Tiene que tener un atributo privado y estático, ya que éste va a ser el que vamos a utilizar para tener una única instancia.
- La clase debe tener un método público y estático, en el cual vamos a generar una instancia o a devolver la instancia que ya hay.

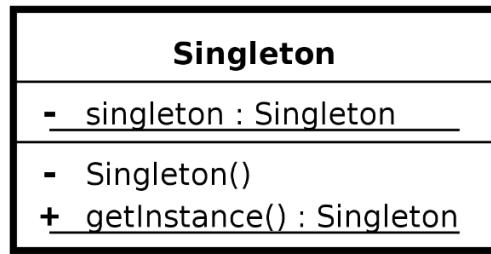


Ilustración 4 - Patrón Singleton

Page Object Model

Este patrón ha sido explicado en el apartado de “Conceptos teóricos”.

4.3. Control de versiones

- Herramientas consideradas: [Git](#), [Mercurial](#), [Subversion](#).
- Herramienta elegida: [Git](#)

Git fue la herramienta seleccionada ya que permite tener una copia del repositorio de forma local, lo cual nos permite una mayor descentralización que Subversion. Además de ser software libre, tiene una gran cantidad de usuarios.

Por otro lado, Git y Mercurial son muy parecidos, sin embargo esta segunda solo es gratuita durante dos semanas. Es por todo ello que elegimos Git como la herramienta para el control de versiones.

4.4. Hosting del repositorio

- Herramientas consideradas: [GitHub](#) y [GitLab](#).
- Herramienta elegida: [GitLab](#).

GitLab fue la herramienta seleccionada ya que nos permite trabajar con Git, tanto GitHub como GitLab son herramientas muy similares, en las que ofrecen revisión de código, documentación...

Esta fue nuestra elección porque estamos más acostumbrado a trabajar con GitLab en vez de con GitHub.

4.5. Gestión del proyecto

- Herramientas consideradas: [GitLab Board](#), [Azure DevOps](#).
- Herramienta elegida: [Azure DevOps](#).

Aunque GitLab Board proporciona una verdadera integración con GitLab, fue Azure DevOps la herramienta utilizada ya que es muy simple, además en grandes empresas como HP es la tecnología que utilizan. La creación de *users stories*, de *epics* y de *tasks* es muy sencillo, además te permite establecer una dificultad a las tareas, se le puede asignar un número de horas que esperas que va a costar... Es por ello que hemos elegido dicha herramienta.

4.6. Gestión del repositorio

- Herramientas consideradas: [GitKraken](#) y [SmartGit](#).
- Herramienta elegida: [GitKraken](#).

Ambas herramientas proporcionan una aplicación de escritorio para poder gestionar tanto el repositorio local como el repositorio de GitLab.

Elegimos GitKraken ya que posee una interfaz de usuario mucho más intuitiva. Además, fue la herramienta que usé en la asignatura Gestión de Proyecto en tercero, por lo que ya sabía manejarla con dicha herramienta.

4.7. Comunicación

- Herramientas consideradas: [Gmail](#), [Outlook](#), [Skype](#), [Zoom](#).
- Herramientas elegidas: [Outlook](#), [Zoom](#).

En cuanto a comunicación escrita Outlook fue la herramienta seleccionada, ya que permite de una manera muy sencilla la integración del correo y el calendario, además posee una aplicación de escritorio con un buen sistema de búsqueda.

Por otro lado, Zoom fue la herramienta elegida para realizar videoconferencias, ya que permite realizar colaboraciones de una forma muy sencilla, esto se debe a que no tienes que instalar nada, por lo que haciendo *click* sobre el *link* que proporcione el *host* de la reunión entras al *meeting*.

4.8. Editor de código

TypeScript

- Herramientas consideradas: [Visual Studio Code](#), [Atom](#).
- Herramienta elegida: [Visual Studio Code](#).

Visual Studio Code fue la herramienta seleccionada ya que integra muy bien la parte de Git y el *Debugger*, ambos dos son “gratuitos”. A día de hoy está más extendido el uso de Visual Studio Code, por lo que tiene más y mejores extensiones, lo cual facilita el desarrollo.

Markdown

- Herramientas consideradas: [Haroopad](#) y [Typora](#).
- Herramienta elegida: [Haroopad](#).

Haroopad fue la herramienta que seleccionamos ya que proporciona una mejor interfaz que Typora, menos minimalista. Haroopad proporciona muchas extensiones para poder escribir un fichero markdown de una forma más sencilla.

4.9. Documentación

- Herramientas consideradas: [LaTeX](#) y [Microsoft Word](#).
- Herramienta elegida: [Microsoft Word](#).

La documentación relativa al repositorio se ha realizado con la herramienta Markdown que he comentado en el apartado anterior.

Para realizar la parte de memoria y anexos elegimos Microsoft Word, el principal motivo fue que todas las tecnologías con las teníamos que trabajar eran nuevas, y si a esto le añadimos el aprender a manejarnos con LaTeX supondría un tiempo adicional el cual no teníamos.

Utilizar LaTeX proporciona grandes ventajas, permite redactar fácilmente documentos estructurados, de macros... Sin embargo, no es un procesador que puedas utilizar desde el primer día ya que tienes que aprender a usar dicha sintaxis, ya que son todo órdenes.

Realmente el punto fuerte de LaTeX es el ámbito científico, ya que la calidad de las fórmulas matemáticas es muy alta, y queda más elegante. Sin embargo, consideramos que para hacer esta documentación no era necesario, ya que nuestro proyecto no tiene una gran cantidad de fórmulas, y el diseño que podemos conseguir con LaTeX lo podemos conseguir con Word de una forma más sencilla.

Es por todo ello, que nuestra elección fue usar Microsoft Word.

4.10. Frameworks y librerías

Angular

[Angular](#) es un framework que permite el desarrollo de aplicaciones web a partir de componentes, estos componentes están formados por ficheros HTML, CSS y TypeScript.

Dicho *framework* fue el que utilizamos para crear las aplicaciones que luego nos sirvieron para probar la *toolkit*.

Electron

[Electron](#) es un *framework* el cual permite que cualquier aplicación desarrollada en JavaScript/TypeScript pueda convertirse a escritorio, es decir, que una misma aplicación web pueda ser también una aplicación de escritorio.

Es un *framework* de código abierto, y permite “transformar” de una aplicación web a escritorio gracias a Node.js en el lado del servidor y Chromium como interfaz.

En nuestro proyecto, primero tuvimos que desarrollar la aplicación en Angular (la aplicación sobre la que íbamos a probar la *toolkit*), y una vez que tuvimos la aplicación web, cambiando la configuración pudimos obtener la aplicación de escritorio gracias a Electron.

Spectron

[Spectron](#) es un *framework*, el cual permite crear test de integración y test *end-to-end* para aplicaciones Electron.

Para poder hacer uso de este *framework* se necesita ChromeDriver y WebdriverIO.

ChromeDriver

[ChromeDriver](#) es una herramienta de código abierto para crear pruebas automáticas de aplicaciones web, al ser un driver de Chromium implementa dicho protocolo.

WebdriverIO

[WebdriverIO](#) nos proporciona una API, es decir, toda la funcionalidad necesaria para poder realizar pruebas sobre una aplicación web. El funcionamiento de esta API es sencillo, WebdriverIO realiza peticiones a un servidor y es ésta misma quién se encarga de gestionar las respuestas que recibe.

Mocha

[Mocha](#) es un *framework* de pruebas para JavaScript, el cual se puede ejecutar en Node.js y en el navegador, y su principal característica es que permite hacer test asíncronos de una forma muy fácil. Además, es compatible con múltiples librerías de aserciones.

Chai

[Chai](#) es una librería de aserciones que sirve tanto para Node.js como para el navegador, la principal ventaja de esta librería es que presenta diferentes tipos de sintaxis, y además es compatible con la gran mayoría de *frameworks* de JavaScript/TypeScript.

Htmlparser2

[Htmlparser2](#) es una librería la cual nos permite parsear de una forma muy sencilla cualquier fichero HTML, de tal forma que podemos obtener el árbol DOM correspondiente a dicho fichero.

El motivo por el que elegimos esta librería es porque a la hora de parsear podemos obtener el DOM, o a medida que vamos parseando podemos obtener lo que nos interesa sin hacer falta recorrer todo el árbol DOM.

4.11. Otras herramientas

Bibliografía

[Zotero](#) es un programa de software libre para la gestión de todas las referencias bibliográficas. Con una simple extensión y una aplicación de escritorio es capaz de identificar y gestionar todo tipo de artículos científicos, investigaciones, páginas web, etc.

Su uso con Word es muy fácil, ya que al instalar Zotero, en el Word tenemos una pestaña más con todas las opciones necesarias para poder añadir citas, modificar, eliminar y generar de forma automática la bibliografía.

Generación de diagramas

[Astah](#) es una herramienta de diseño que soporta una gran variedad de diagramas, por ejemplo:

- Diagramas UML.
- Diagramas de secuencia.
- Diagramas de flujo.
- Diagramas de componentes.

Esta herramienta no es de software libre, aunque si eres estudiante la versión Astah UML es gratis, ya que te permite poder usarla con una licencia de estudiante.

5. Aspectos relevantes del desarrollo del proyecto

En este apartado se va a explicar el resumen de la experiencia del proyecto, así como los problemas a los que nos hemos tenido que enfrentar, y las soluciones para cada uno de ellos.

5.1. Inicio del proyecto

El motivo por el que elegí este trabajo fue porque encontré un tema que me gusta (el *testing*), además de las innovadoras tecnologías con las que me iba a manejar. Al cursar la asignatura de validación y pruebas, descubrí un nuevo punto de vista que hasta entonces no había tenido.

Al cursar dicha asignatura como bien mencioné, vi por mí mismo la dificultad que supone hacer los test, no me refiero a si es más o menos complejo escribir el código necesario para que pasen las pruebas, sino más bien al tiempo que lleva realizar las mismas.

El proceso de *testing* es algo tedioso, además implica que el *tester* necesita conocimientos sobre el desarrollo de la aplicación. Es por ello, que cuando vino el equipo de HP a presentar los trabajos, éste fue uno que me llamó mucho la atención, básicamente por dos motivos, el primero que trataban de simplificar el código para desarrollar test, lo cual desde mi punto de vista es bastante importante ya que para un *tester junior* le facilita el aprendizaje, y para un *senior* le va a ahorrar mucho tiempo. El segundo motivo fue que buscaban automatizar la estructura de pruebas, es decir, que de manera automática se pudiera identificar todos los elementos que hay dentro de una aplicación. Gracias a esto, un *tester* apenas necesita conocimientos sobre las tecnologías utilizadas en el desarrollo de una aplicación, en nuestro caso una aplicación de escritorio, para la realización de las pruebas.

Una vez que se confirmó que ese trabajo era para mí, nos pusimos a trabajar para conseguir dichos objetivos.

5.2. Metodologías

Al comienzo del proyecto no definimos una metodología, ya que las primeras reuniones que hicimos fueron para saber cuáles eran los objetivos que se querían conseguir, las tecnologías con las que teníamos que trabajar...

Una vez planteado el proyecto, lo acepté, por lo que nos pusimos manos a la obra, pero no lo queríamos hacer de cualquier forma. Es por ello que decidimos aplicar una de las metodologías ágiles más conocidas que es SCRUM.

Aunque no hemos aplicado la metodología SCRUM al cien por cien, hemos tratado de seguirla conceptualmente:

- El desarrollo fue iterativo, es decir, se utilizaron *sprints*.
- Fue incremental, ya que al final de cada *sprint* se entregaba una parte del producto.
- La duración del *sprint* fue de dos semanas mayoritariamente.
- Dentro de un mismo *sprint* se realizaban tres reuniones, siendo la primera reunión dónde definíamos los objetivos de dicho *sprint*, en la segunda reunión era como una reunión diaria (explicaba lo que había hecho, los problemas que había tenido, y lo que iba a hacer hasta el final del *sprint*), y la tercera reunión servía tanto para cerrar el *sprint*, como para indicar de nuevo los objetivos del siguiente *sprint*.
- Al comienzo del *sprint* definíamos las tareas que íbamos a realizar.
- Las tareas se estimaban dependiendo de la complejidad de las mismas.
- Se utilizó un sistema Kanban para la organización de las tareas dentro del *sprint*.

Aunque el desarrollo fue incremental, es verdad que al principio no podíamos entregar una parte del producto, ya que nuestro proyecto no es algo visual. Una vez pasado el ecuador del proyecto, sí que pudimos entregar una parte funcional del producto al final de cada *sprint*.

Para la parte de la librería, la metodología que seguimos fue la de probar y probar, ya que no podíamos hacer test unitarios sobre los métodos que habíamos

hecho, sino que para poder probarlo teníamos que hacer uso de una aplicación Angular/Electron.

Por otro lado, para la automatización de la estructura de pruebas sí que pudimos hacer test unitarios y de integración, es verdad que podíamos haber seguido la metodología TDD, sin embargo, decidimos que lo mejor era hacer primero el código y luego las pruebas, ya que la tecnología con la que estábamos trabajando la desconocíamos, de esa forma pudimos “jugar” un poco con ella mientras realizábamos el código.

5.3. Formación

Aunque el desarrollo del proyecto ha durado cinco meses, realmente los dos primeros fueron para aprender a manejarnos con las tecnologías que íbamos a trabajar, esto se debe a que debíamos de tener unos conocimientos previos antes de poder “empezar” a desarrollar el proyecto.

Cabe destacar que el proyecto ha sido desarrollado en TypeScript, sin embargo, para poder entender como funcionan las aplicaciones Angular/Electron tuvimos que desarrollar una pequeña aplicación en dichas tecnologías, por ello realizamos los siguientes cursos:

- Angular: De cero a experto creando aplicaciones (Angular 7+) [19].
- Learn Angular 5 from Scratch [20].

Por otro lado, tuvimos que aprender a manejarnos con Spectron, Mocha, Chai, WebdriverIO y TypeScript, en todos ellos aprendimos de dos formas, leyendo la documentación de todas las tecnologías mencionadas, y jugando con ellas, cuando nos encontrábamos un error mirábamos en [StackOverflow](#) o en alguna página de internet cual era la solución.

- Documentación Spectron [21].
- Documentación Mocha [22].
- Documentación Chai [23].
- Documentación WebdriverIO [24].
- Documentación TypeScript [25].

5.4. Pre-Desarrollo

Antes de comenzar el desarrollo de la *toolkit*, lo primero que tuvimos que hacer fue programar una aplicación Angular para así poder adquirir conocimientos de cómo funcionaban las tecnologías con las que íbamos a trabajar. La aplicación elegida fue Tour of Heroes [26].

Una vez desarrollada la aplicación Angular la tuvimos que convertir a una aplicación Electron, para así poder pasar de aplicación web a aplicación de escritorio.

Posteriormente, el objetivo fue aprender a manejarnos con las tecnologías de *testing*, de tal forma que creamos test para poder adquirir conocimientos, algunos test fueron más complejos y otros más sencillos, ya que el objetivo era aprender a manejarnos en estas tecnologías.

En esta fase tuvimos problemas principalmente con las librerías, ya que no sabíamos muy bien cómo utilizarlas, es por ello que esta fase duró dos meses, hasta que adquirimos los conocimientos necesarios para desarrollar tanto la librería como la automatización, es decir, a partir de este momento comenzó el desarrollo de TestingAWE.

5.5. Desarrollo de la librería

La primera parte del proyecto se utilizó para simplificar la librería de WebdriverIO y así que cualquier *tester* la pudiera utilizar sin ninguna dificultad.

El problema a resolver fue el cómo íbamos a organizar dicha librería por varios motivos:

- Había que crear una estructura lógica.
- Dicha estructura debía de ser lo más eficiente posible.
- Tenía que abstraer el *framework* Spectron de los test, de tal forma que el desarrollo de éstos fuera lo más simple.
- La librería tenía que dar soporte a una gran cantidad de elementos HTML.
- Había que agrupar de forma adecuada los métodos de WebdriverIO para que así el test fuera más legible.

El desarrollo comenzó con una fase de investigación, para saber cuál iba a ser la mejor forma de estructurar la librería, además el equipo de HP me proporcionó varias ideas de cómo solucionar dicho problema.

Finalmente, se decidió que la mejor forma de organizar la librería iba a ser a través del uso de interfaces, clases abstractas y clases.

Cabe mencionar que vamos a tener una clase por cada elemento HTML que da soporte nuestra librería, por lo tanto, muchos métodos que se pueden aplicar sobre un elemento pueden ser usados por otros, es por ello que decidimos hacer uso de clases abstractas para así reducir lo máximo posible el código duplicado. Aunque las interfaces en TypeScript no proporcionan un valor añadido (no se puede definir el cuerpo de los métodos), éstas han sido importantes a la hora de elaborar un “contrato”, es decir, nos permite que las clases abstractas tengan obligatoriamente un conjunto de métodos definidos. El cómo se produce la relación entre clases e interfaces, lo podemos ver en los anexos.

El siguiente problema que tuvimos que hacer frente fue el idear cómo iban a ser los métodos que proporcionaba la librería, es decir, cómo podíamos simplificar la sintaxis que proporciona WebdriverIO y Spectron, para facilitar el desarrollo de los test y por lo tanto que éstos quedaran lo más legible posible.

Para ello, se ideó que los métodos inicialmente fueran lo más básico posible, es decir, agrupar las acciones más simples de WebdriverIO en nuestra librería, como por ejemplo obtener el texto de un párrafo... Posteriormente, con el objetivo de ampliar la librería, se crearon métodos algo más complejos, para así poder reducir aún más la sintaxis en los test, como por ejemplo el determinar si el texto de un párrafo es el esperado...

En las siguientes ilustraciones podemos ver cómo se han simplificado los métodos, lo cual proporciona una facilidad en el uso de nuestra librería.

```
let heroes = await app.client.elements('#dashboard_div > a').then((res) => {
  return res.value.length;
});
```

Ilustración 5 - Uso de WebdriverIO sin TestingAWE

```
let heroes = await dashboardDiv.getCount('#dashboard_div > a');
```

Ilustración 6 - Uso de WebdriverIO con TestingAWE

Para concluir con esta parte, el otro objetivo que nos propusimos fue abstraer la parte de Spectron en los test, para que así cualquier usuario *tester* pudiera desarrollar las pruebas sin tener conocimientos de cómo funciona.

Spectron es un *framework* el cual nos permite crear test para una aplicación Electron, su uso radica en tres apartados:

- Inicializar Spectron, es decir, indicarle la aplicación Electron sobre la que tiene que hacer las pruebas.
- Arrancar la aplicación al inicio de los test y cerrarla cuando éstos finalicen.
- Obtener la instancia de la aplicación para así poder ejecutar los métodos que proporciona WebdriverIO.

La solución que se propuso fue la de diseñar una clase que nos permitiese garantizar que durante la ejecución de los test hubiera solo una única instancia del *framework*. Para ello la mejor solución es aplicar un patrón de diseño, y siempre que se quiere una instancia única se aplica el patrón Singleton.

El resultado de la aplicación de este patrón, lo podemos observar en la clase “ApplicationInstance”, la cual se encuentra dentro de la librería.

En las siguientes ilustraciones podemos ver el cómo se abstrae Spectron, gracias a TestingAWE.

```

before() => {
  app = new Application({
    path: join(__dirname, "..", "node_modules", "electron", "dist", "electron.exe"),
    args: [join(__dirname, "../../tour-of-heroes")]
  });
  return app.start().then((startApp) => {
    client = startApp.client;
  });
};

after() => {
  if (app && app.isRunning()) {
    return app.stop();
  }
};

```

Ilustración 7 - Inicialización, arranque y cierre sin TestingAWE

```

before(async () => {
  await applicationInstance.startApplication();
});

after(async () => {
  await applicationInstance.stopApplication();
});

```

Ilustración 8 - Inicialización, arranque y cierre con TestingAWE

Como podemos apreciar en las anteriores ilustraciones, respecto a la inicialización, arranque y cierre de Spectron se ha simplificado de una forma muy notoria. Además, con TestingAWE no hace falta inicializar en el test la aplicación sobre la que va a realizar las pruebas, ya que como veremos más adelante, de esta parte se encarga la automatización de la *toolkit*, lo único que debemos hacer es importar el objeto “applicationInstance”, el cual será generado en la automatización.

El uso de la clase “ApplicationInstance” nos permite reducir la sintaxis en los test, esto se debe al igual que antes, a la abstracción de Spectron, tal y como podemos apreciar en las siguientes ilustraciones.

```

await app.client.$('#button_dashboard').click().pause(1000);

```

Ilustración 9 - Uso de Spectron sin TestingAWE

```
await appButtonDashboard.click()
```

Ilustración 10 - Uso de Spectron con TestingAWE

Cabe destacar que, durante el desarrollo de esta fase las pruebas que pudimos hacer fueron directamente sobre la aplicación de *Tour of Heroes*, de tal forma que no pudimos hacer test unitarios como tal, sino que los test que realizamos fueron *end-to-end*, ya que necesitábamos de Spectron para poder probar los métodos de nuestra librería.

5.6. Desarrollo de la automatización

La segunda fase del desarrollo consistió en definir el cómo íbamos a obtener los elementos HTML, cómo se iba a generar los ficheros (los cuales contienen la instanciación de los elementos HTML identificados), y cómo se iba a producir la automatización de este proceso para así reducir el tiempo.

Como bien he mencionado en el párrafo anterior, este desarrollo lo podemos dividir en tres fases.

Respecto a la primera fase, nos propusimos como objetivo investigar cual podía ser la mejor forma para poder identificar los elementos HTML que tiene una página. Para ello surgieron dos posibilidades, crear un parser, o investigar si en estas tecnologías había ya algún parser de HTML.

Tras investigar, descubrimos que había tres *parsers* que nos podían servir. Finalmente, elegimos *htmlparser2* ya que nos proporcionaba la estructura DOM más simple posible. De primeras, pensamos que la mejor forma para poder obtener todos los elementos HTML era la de recorrer el árbol DOM, sin embargo investigamos un poco más, y con este parser no hacía falta recorrer el árbol DOM ya que a la vez que se parsea el fichero HTML podíamos indicarle que obtuviera el nombre de la etiqueta HTML junto con su atributo id, de esta forma hemos conseguido ahorrarnos mucho tiempo ya que podemos hacer dos cosas a la vez.

En cuanto a la segunda fase, el objetivo fue generar los ficheros intermedios, los cuales hacen uso de la librería, en ellos se instancian todos los objetos necesarios que hemos identificado gracias al parser.

Para generar estos ficheros, lo que hacemos es recorrer la aplicación Angular/Electron, y por cada archivo HTML que encontramos lo parseamos, y luego nuestra *toolkit* va a crear un fichero correspondiente a esa página.

En la siguiente ilustración podemos ver un ejemplo de fichero generado por nuestra *toolkit*:

```
import * as awe from "testing-awe";

export const dashboardTitle = new awe.H('#dashboard_title');
export const dashboardDiv = new awe.Div('#dashboard_div');
export function dashboardADetailsHero(param1 : string) : awe.A {
    let id : string = '#' + 'dashboard_a_details_hero_' + param1;
    return new awe.A(id)
};
export function dashboardDivDetailHero(param1 : string) : awe.Div {
    let id : string = '#' + 'dashboard_div_detail_hero_' + param1;
    return new awe.Div(id)
};
export function dashboardH4DetailHeroName(param1 : string) : awe.H {
    let id : string = '#' + 'dashboard_h4_detail_hero_name_' + param1;
    return new awe.H(id)
};
```

Ilustración 11 - Fichero generado por TestingAWE

Como podemos observar en la anterior ilustración nos genera el objeto necesario para cada elemento HTML identificado en la página correspondiente.

En esta fase hubo un punto crítico ya que en Angular/Electron, puede haber elementos que se generan dinámicamente, es decir, que su id se calcula de forma dinámica cuando la aplicación se ejecuta, sin embargo, el parser identifica los elementos del fichero HTML, es decir, un fichero de texto plano que no está siendo ejecutado.

La solución por la que nos decantamos fue, a la hora de parsear clasificar los elementos HTML dependiendo de si su id es dinámico o no. Gracias a esta clasificación a la hora de generar los ficheros lo hacemos de dos formas distintas:

- Si el id del elemento es estático, instanciamos un objeto del tipo de elemento HTML que se ha identificado, ya que su id (necesario para

poder identificar el objeto a la hora de hacer pruebas) no va a cambiar durante la ejecución.

- Si el id del elemento es dinámico, seguimos instanciando un objeto, pero el procedimiento cambia, ya que ahora para obtener un objeto correspondiente a ese elemento lo que se hace es proporcionar un método, el cual se le pasa por parámetro el “valor” distintivo del id, dicho valor se lo tiene que proporcionar el usuario cuando vaya a realizar las pruebas.

A continuación, se va a explicar el cómo se genera el fichero cuando es un elemento HTML con id estático o dinámico, y cómo el usuario va a tener que hacer uso de dicho fichero.

Caso para elemento HTML con id estático

Fichero Generado por TestingAWE

Fichero Test

```
export const heroesH2 = new awe.H('#heroes_h2'); let titulo = await heroesH2.getText();
```

Tabla 2 - Generación del fichero para elemento estático

Como podemos apreciar en este caso, importando el objeto generado por TestingAWE ya podemos hacer uso del mismo en el test.

Caso para elemento HTML con id dinámico

Fichero Generado por TestingAWE

Fichero Test

```
export function heroesADetailHero(param1 : string) : awe.A {
  let id : string = '#' + 'heroes_a_detail_hero_' + param1;
  return new awe.A(id)
};
```

```
let heroesADetailHero1 = heroesADetailHero("1");
```

```
await heroesADetailHero1.click();
```

Tabla 3 - Generación del fichero para elemento dinámico

Cuando el id es dinámico, no podemos hacer uso del objeto generado por TestingAWE directamente, sino que es el usuario quien debe instanciar el objeto a partir del método que le proporciona la *toolkit*.

Para terminar con este apartado, cabe mencionar que los ficheros generados no se realizan de forma manual, sino que es el usuario quién los genera de forma automática, lo cual nos permite reducir el tiempo.

Para que esta automatización tenga éxito hay que hacer uso de un fichero de configuración, el saber cómo se tiene que configurar dicho fichero viene explicado de forma detallada en los anexos.

5.7. Testing

Las pruebas que se han realizado en este proyecto son:

- Pruebas unitarias.
- Prueba de integración.
- Pruebas *end-to-end*.

Para la parte de la automatización, se han realizado tanto pruebas unitarias como pruebas de integración, ya que esta parte no tiene la necesidad de lanzar Spectron para poder realizar test.

Dichas pruebas se encuentran dentro del módulo de TestingAWE.

Respecto a la parte de la librería, solamente se han podido hacer pruebas *end-to-end*, ya que para poder *testear* esta funcionalidad necesitamos lanzar Spectron, y para lanzar éste necesitamos una aplicación Angular/Electron.

Estas pruebas no se encuentran dentro del módulo de TestingAWE, ya que para poder probarlo tuvimos que desarrollar una aplicación Angular/Electron con todos los elementos HTML que soporta nuestra *toolkit*, y después tuvimos que crear los test *end-to-end* para comprobar que para cada elemento los métodos que tiene asociados funcionan correctamente.

5.8. Publicación

Una vez finalizada toda la fase de desarrollo, la *toolkit* se publicó en la plataforma NPM, en ella se encuentran todos los paquetes de Node.

NPM permite que con una sola línea de código en la consola, podamos obtener cualquier librería. Gracias a esto TestingAWE tiene una instalación muy sencilla.

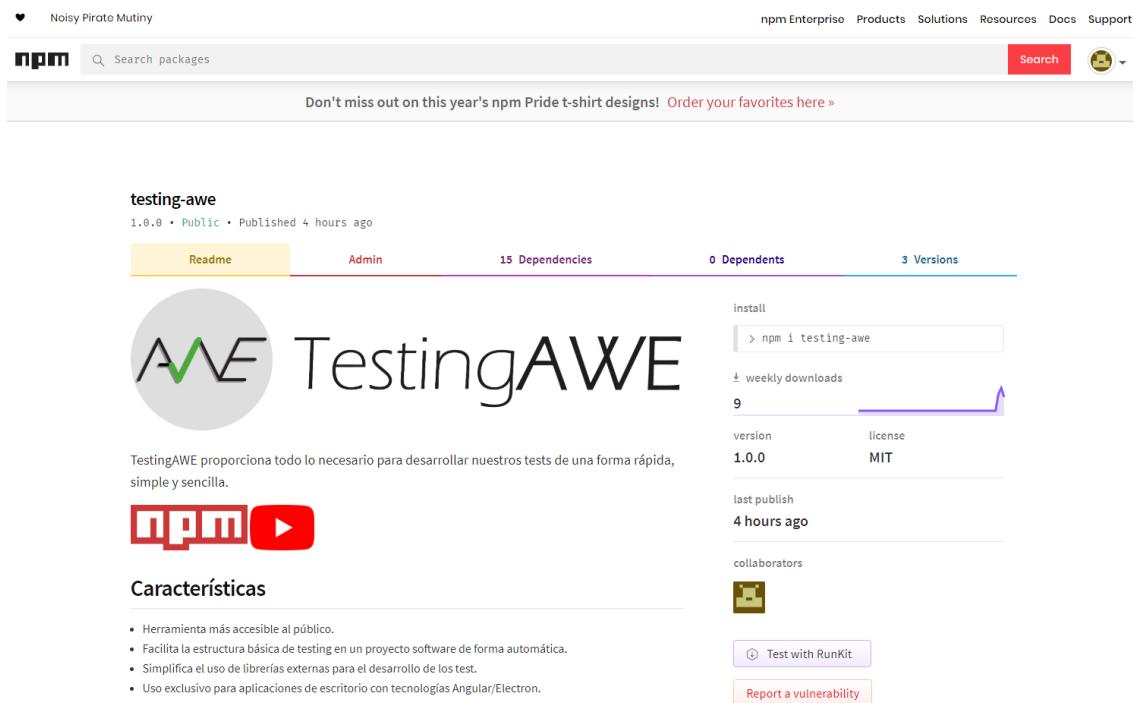


Ilustración 12 - Módulo TestingAWE en NPM

Para finalizar, se ha creado un canal de YouTube [6] en el cual se explica los requisitos para el uso de TestingAWE, el cómo se puede instalar, configurar y utilizar.

6. Trabajos relacionados

Como bien comenté en la introducción de esta memoria, a día de hoy no hay una tecnología que permite hacer lo que hace la nuestra, es decir, si queríamos hacer pruebas sobre una aplicación Angular/Electron, lo que debíamos hacer era descargar el *framework* de pruebas, múltiples librerías, aprender a manejarnos con todas ellas y a desarrollar código.

Por lo tanto, no hay otros trabajos con los que me pueda comparar, sino que solo me puede comparar con el trabajo manual de *testing*.

Características	TestingAWE	Trabajo Manual
No requiere hardware específico	✓	✓
Instalación sencilla	✓	✗
Simplificación del uso de la librería	✓	✗
Facilita la identificación de elementos HTML con los que hacer pruebas	✓	✗
Apto para todo el público tester	✓	✗
Sintaxis más legible	✓	✗
Ahorra tiempo y dinero	✓	✗

Tabla 4 - Comparativa de los trabajos

Como podemos observar en la anterior tabla, las principales fortalezas son:

- Instalación sencilla, es decir, instalando el módulo NPM con una sola línea ya podemos hacer uso de toda la *toolkit*. Además, con indicarle a TestingAWE el proyecto Angular/Electron podemos usar la librería y podemos generar los ficheros que crea la *toolkit*.
- Simplificación en el uso de la librería, como bien hemos mencionado a lo largo de la memoria, al agrupar la funcionalidad que nos proporciona WebdriverIO en métodos más simples, y creando métodos nuevos, hemos simplificado la sintaxis de las pruebas.
- Facilita la identificación de elementos HTML, hasta día de hoy cuando queríamos hacer pruebas sobre una aplicación Angular/Electron, teníamos que saber del desarrollo de esa aplicación, es decir, de los elementos HTML que tenía. Sin embargo, gracias TestingAWE podemos obtener de una forma muy sencilla todos los elementos con los que podemos hacer pruebas.
- Apto para todo el público *tester*, al simplificar el uso de la librería y facilitar la identificación de elementos HTML de forma automática, conseguimos que tanto un *tester junior* como un *tester senior* puedan usar nuestra *toolkit* sin ningún problema, facilitando en la medida de lo posible el proceso *testing*.
- Sintaxis más legible, al simplificar al máximo la librería hemos conseguido que a la hora de desarrollar las pruebas el código sea mucho más simple, por lo tanto, incluso una persona que no sea *tester* pero que tenga unos conocimientos muy básicos de HTML, va a poder saber qué hace cada línea de una forma muy sencilla.
- Ahorra tiempo y dinero, al juntar todas las ventajas anteriores, obtenemos como resumen que conseguimos desarrollar las mismas pruebas que hacíamos antes de forma manual en menos tiempo, gracias a la simplificación de la librería y a la automatización para identificar elementos HTML con los que podamos hacer pruebas. Todo lo que implique ahorrar tiempo significa a su vez que estamos ahorrando dinero.

Las principales debilidades son:

- Actualmente solo se encuentra disponible para Windows, aunque al estar desarrollado en TypeScript se permite que se pueda ejecutar en diferentes entornos, cabe la posibilidad que la configuración de los proyectos no sea la misma.
- Por otro lado, para hacer uso de TestingAWE como mínimo debes de tener conocimientos básicos de HTML, por lo que esta herramienta no es apta para todo el público.
- Además, la librería depende de WebdriverIO, por lo que puede ser que esta en algún momento deje de existir, o que ya no sea compatible con las tecnologías que hemos trabajado.

Tal y como podemos apreciar en las siguientes ilustraciones, TestingAWE permite simplificar el uso de la librería proporcionando así una sintaxis más legible, esto a su vez nos da la ventaja de que ahorramos tiempo y que cualquier persona *tester* pueda hacer uso de nuestra *toolkit*.

```
it("Test 1", async() => {
    //Accedo a la página app-dashboard
    await app.client.$('#button_dashboard').click().pause(1000);
    //Me quedo con el título de la página app-dashboard
    let titulo = await app.client.$('h3').getText();
    assert.equal(titulo, "Top Heroes");
    //Me quedo con el número de top heroes
    let heroes = await app.client.elements('#dashboard_div > a').then((res) => {
        return res.value.length;
    });
    assert.equal(heroes,4);
});
```

Ilustración 13 - Desarrollo de test sin TestingAWE

```
it.only("Test 1 TOOLKIT", async() => {
    //Accedo a la página app-dashboard
    await appButtonDashboard.click()
    //Me quedo con el título de la página app-dashboard
    let titulo = await dashboardTitle.getText();
    assert.equal(titulo, "Top Heroes");
    //Me quedo con el número de top heroes
    let heroes = await dashboardDiv.getCount('#dashboard_div > a');
    assert.equal(heroes,4);
});
```

Ilustración 14 - Desarrollo de test con TestingAWE

Aunque a simple vista en desarrollo vertical solo nos hemos quitado una línea en este ejemplo, es verdad que en desarrollo horizontal, hemos conseguido reducir mucho la sintaxis.

Además, tal y como podemos ver, es más legible un test desarrollado con TestingAWE, lo cual hace que sea más fácil de usar.

Como consecuencia en la reducción de la sintaxis, también conseguimos disminuir el tiempo de desarrollo del test, para este ejemplo cuando no usamos TestingAWE el tiempo de desarrollo ha sido de tres minutos aproximadamente, sin embargo, usando TestingAWE hemos tardado un minuto y medio.

Realizando varias pruebas para estimar cuánto tiempo se consigue reducir usando TestingAWE, sale que aproximadamente reducimos de media entre un 40% y un 50%, lo cual es muy significativo.

7. Conclusiones y Líneas de trabajo futuras

En este apartado vamos a explicar las conclusiones del proyecto, y las diferentes líneas futuras de trabajo que se pueden seguir para poder ampliar esta herramienta.

7.1. Conclusiones

Una vez finalizado el proyecto podemos concluir los siguientes puntos:

- El principal objetivo de este proyecto se alcanzó, ya que conseguimos facilitar el desarrollo de pruebas a los tester tanto si son *juniors* como *seniors*. Además, hemos conseguido crear una herramienta la cuál es muy fácil de instalar gracias al soporte que ofrece Node Package Manager para Node.js.
- Todo el desarrollo de la aplicación ha sido sobre TypeScript, dicho lenguaje es muy demandado hoy en día, su desarrollo no está de todo terminado (se puede mejorar), por lo que ha habido momentos en los que su uso ha sido algo costoso. Sin embargo, la gran ventaja de usar este lenguaje para el desarrollo es que prácticamente se va a poder utilizar en cualquier sistema operativo de escritorio.
- Gran parte del desarrollo de este proyecto ha estado basado en los conocimientos adquiridos durante el grado, como el uso de patrones de diseño, la realización de pruebas, documentar bien el proyecto...
- Uno de los puntos fuertes de este proyecto son las diferentes tecnologías con las que hemos tenido que trabajar, todas ellas son conocidas y prácticamente en cualquier oferta de trabajo se solicitan. El principio del proyecto no fue fácil ya que tuvimos que aprender a manejarnos con todas ellas, sin embargo, el conocimiento que hemos adquirido durante el desarrollo nos va a servir para un futuro cercano.

- Al no haber apenas documentación del uso de estas tecnologías, por lo novedosas que son, hemos tenido que realizar una gran fase de investigación lo cual nos ha enseñado, aunque parezca mentira, a usar Google, a manejarnos en foros de programación...
- La aplicación de SCRUM en el desarrollo del proyecto nos ha permitido adquirir grandes conocimientos sobre realmente cómo funcionan las metodologías ágiles. Además, la principal ventaja que vimos fue la gran capacidad al cambio que te permite esta metodología.

7.2. Líneas de trabajo futuras

Desde nuestro punto de vista, consideramos que hay varias ramas fuertes para continuar con el desarrollo de esta herramienta:

- Ampliar la librería, es decir, poder incorporar diferentes librerías similares a WebdriverIO que permitan realizar una infinidad de acciones sobre los elementos HTML, o ampliar la librería de TestingAWE para que pueda realizar dichas acciones.
- Hacer que TestingAWE sea realmente compatible con todos los sistemas operativos de escritorio que hay en el mercado.
- Crear una sintaxis al estilo [Cucumber](#), para que cualquier usuario independientemente si es *tester* o no, sea capaz de hacer uso de TestingAWE sin ningún problema. Este punto daría un gran salto de calidad a este producto, ya que la fase que hay desarrollada es muy novedosa (no hay nada parecido), si lo juntamos con una buena sintaxis simplificaríamos mucho el desarrollo del *testing*.

8. Bibliografía

- [1] A. Bertolino, «Software testing», *SWEBOK*, p. 69, 2001.
- [2] M. R. Carreira y I. R. Román, «Estimación del coste de la calidad del software a través de la simulación del proceso de desarrollo», *I*, vol. 2, n.º 1, p. 13, jun. 2001.
- [3] «TestingAWE», *npm*. [En línea]. Disponible en: <https://www.npmjs.com/package/testing-awe>. [Accedido: 23-jun-2019].
- [4] «Repositorio TestingAWE», *GitLab*. [En línea]. Disponible en: <https://gitlab.com/mus0007/testing-awe>. [Accedido: 25-jun-2019].
- [5] «TestingAWE - Azure DevOps». [En línea]. Disponible en: <https://dev.azure.com/juanmanuelzamarrenoperez/Observatorio-TestingAWE>. [Accedido: 19-jun-2019].
- [6] «YouTube TestingAWE Oficial», *YouTube*. [En línea]. Disponible en: <https://www.youtube.com/channel/UCqUSDpoDw3RbqPb6PjAAAnBA>. [Accedido: 25-jun-2019].
- [7] «Patrones de diseño en automatización: PageObjects», *QA:news*, 07-ago-2014. .
- [8] Mohsin, «Page Object Model (POM) | Design Pattern», *tajawal*, 24-may-2018. .
- [9] «Scrum (software development)», *Wikipedia*. 24-ene-2017.

- [10] «Qué es SCRUM», *Proyectos Ágiles*, 04-ago-2008. .
- [11] «GitFlow mejora la gestión de tu repositorio Git». .
- [12] G. Galdámez, «Cómo ser un Super Desarrollador: introducción a git-flow (Parte 1)», *Medium*, 14-jul-2017. .
- [13] «Azure DevOps Services | Microsoft Azure». [En línea]. Disponible en: <https://azure.microsoft.com/es-es/services/devops/>. [Accedido: 27-abr-2019].
- [14] «Planning poker», *Wikipedia, la enciclopedia libre*. 05-abr-2019.
- [15] «Técnica Pomodoro», *Wikipedia, la enciclopedia libre*. 27-dic-2018.
- [16] «Modelo–vista–controlador», *Wikipedia, la enciclopedia libre*. 13-abr-2019.
- [17] «MVC (Model, View, Controller) explicado.», *CódigoFacilito*. [En línea]. Disponible en: <https://codigofacilito.com/articulos/mvc-model-view-controller-explicado>. [Accedido: 27-abr-2019].
- [18] «Singleton», *Wikipedia, la enciclopedia libre*. 10-oct-2018.
- [19] «Angular: De cero a experto creando aplicaciones (Angular 7+)», *Udemy*. [En línea]. Disponible en: <https://www.udemy.com/angular-2-fernando-herrera/>. [Accedido: 27-abr-2019].
- [20] «Learn Angular 5 from Scratch», *Udemy*. [En línea]. Disponible en: <https://www.udemy.com/angular-5/>. [Accedido: 27-abr-2019].
- [21] *Spectron documentation*. Electron, 2019.

- [22] «Mocha documentation». [En línea]. Disponible en: <https://mochajs.org/api/>. [Accedido: 27-abr-2019].
- [23] «Chai documentation». [En línea]. Disponible en: <https://www.chaijs.com/api/>. [Accedido: 27-abr-2019].
- [24] «WebdriverIO documentation». [En línea]. Disponible en: <https://webdriver.io/index.html>. [Accedido: 27-abr-2019].
- [25] «TypeScript documentation». [En línea]. Disponible en: <https://www.typescriptlang.org/docs/home.html>. [Accedido: 27-abr-2019].
- [26] «Tutorial Tour of Heroes». [En línea]. Disponible en: <https://angular.io/tutorial>. [Accedido: 27-abr-2019].



Esta obra está bajo una licencia Creative Commons Reconocimiento 4.0
Internacional ([CC-BY-4.0](#))