UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Un marco de trabajo para el desarrollo y ejecución de pruebas automatizadas aplicadas al front-end de una aplicación web

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Sergio Fernández Rodríguez

Tutor: Patricio Orlando Letelier Torres

2015-2016

Resumen

Este proyecto se centra en la automatización de pruebas funcionales y todo el marco de trabajo que lo envuelve para aplicarlas en regresión de forma automática. El desarrollo de software implica la realización de pruebas funcionales, ya que son esenciales para asegurar que se ha implementado lo que el cliente requiere. Sin embargo, en la actualidad no existe un marco de trabajo para llevar a cabo la automatización de pruebas en un software desarrollado sobre AngularJS.

Se ha hecho uso de Protractor para elaborar las pruebas automatizadas y de diferentes herramientas para añadir funcionalidad, como reportes con los resultados. Además, se ha desarrollado un gestor de pruebas automatizadas en AngularJS, con el que podemos seleccionar y ejecutar directamente las pruebas utilizando Jenkins.

Palabras clave: pruebas automatizadas, pruebas de regresión, desarrollo ágil, testing, calidad de software, Protractor, AngularJS, Jenkins.

Abstract

This project deals with the automation of functional tests and the entire framework that surrounds it to apply them automatically. Software development implies performing functional testing since it is essential to ensure that it has been implemented what the customer requires. However, at present there is no way to carry out testing automation in a software developed on AngularJS.

We have made use of Protractor to develop automated tests, and more different tools to add functionality such as the test reports. We have also developed an automated test manager in AngularJS, with which we can select and run tests directly using Jenkins.

Keywords: automated testing, regression testing, agile development, testing, quality assurance, Protractor, AngularJS, Jenkins.

Tabla de contenido

1. Glosario

- **API REST**: API, o librería de funciones, a la que se accede por el protocolo HTTP. Es decir, a una API REST se accede a través de direcciones web o URL en las que enviamos los datos de nuestra consulta.

- **Back-end**: Lado del servidor, que se encarga de interactuar con la base de datos y de manipular los datos. Recibe, procesa y envía información al navegador del usuario.

- **Framework**: Entorno o ambiente de trabajo para desarrollo que, dependiendo del lenguaje, normalmente integra componentes que facilitan el desarrollo de aplicaciones como el soporte de programa, bibliotecas, plantillas y más.

- **Front-end**: Lado del cliente, encargado de maquetar la estructura semántica del contenido (HTML), codificar el diseño en hojas de estilo (CSS) y agregar la interacción con el usuario (JavaScript).

- **JSON**: Formato para el intercambio de datos. Básicamente, JSON describe los datos con una sintaxis dedicada que se usa para identificar y gestionar los datos.

- **Prueba de aceptación (PA)**: Prueba que tiene como propósito demostrar al cliente el cumplimiento de un requisito del software.

- **Prueba de regresión**: Pruebas cuya aplicación se repite para asegurar que al hacer cambios en el software no se han introducido fallos en funcionalidad ya existente.

- **Scope**: Objeto utilizado por AngularJS para comunicarse entre componentes, particularmente entre el HTML y JavaScript.

- **Sprint**: Periodos cortos de tiempo (entre dos y cuatro semanas) en los que se abordan UTs.

- **Suite de pruebas**: Colección de pruebas que se agrupan para ejecutarlas a la vez.

- **Unidad de trabajo (UT)**: Cada cambio elemental que se realiza sobre el producto.

2. Introducción

Las pruebas del software nos permiten garantizar en cierta medida la ausencia de fallos en el producto. Aunque no es posible asegurarlo en un 100%, una buena combinación de técnicas de pruebas junto con unas buenas baterías de pruebas nos permitirá detectar oportunamente muchos de los posibles fallos.

Este TFG se enmarca en la realización de una práctica de empresa en la cual formo parte de un equipo de testers, participando en el desarrollo de un producto dirigido al sector sanitario. En el proyecto se están estableciendo diferentes técnicas de pruebas, en un contexto de desarrollo ágil centrado en las pruebas de aceptación (ver Glosario).

Uno de los principales desafíos en el trabajo de pruebas del software es la aplicación de pruebas de regresión. Las pruebas de regresión son pruebas cuya aplicación se repite para asegurar que al hacer cambios en el software no se han introducido fallos en funcionalidad ya existente. En un producto que está en desarrollo o mantenimiento que amplía su funcionalidad, la cantidad de pruebas y el esfuerzo asociado a aplicarlas en regresión puede ir incrementándose significativamente. Esta situación no sostenible llevará en algún momento a que no será posible aplicar pruebas de regresión para todas las pruebas existentes y menos el hacerlo de forma manual. Así pues, la automatización de pruebas y su ejecución automática es una interesante alternativa para la aplicación de pruebas de regresión. La automatización de pruebas tampoco está libre de inconvenientes, tales como:

a) La automatización de pruebas conlleva escribir código de pruebas, código que debe organizarse adecuadamente para que sea mantenible y reutilizable.
b) Hay pruebas que son difíciles o que no conviene automatizar.
c) Se requiere una buena infraestructura para ejecutar las pruebas automatizadas para poder contar con los resultados en poco tiempo.
d) Las pruebas aplicadas con resultado KO deben ser revisadas una a una, lo cual exige un esfuerzo no despreciable.

e) Hay que seleccionar y formar al equipo en técnicas y herramientas específicas para el tipo de pruebas que se automatizará.

Este TFG se centrará en la automatización de pruebas de aceptación aplicadas a la parte *front-end* (ver Glosario) del producto. La tecnología utilizada en *front-end* es web, utilizando el *framework* (ver Glosario) AngularJS. El proceso de desarrollo ágil está basado en el enfoque y herramienta TUNE-UP (desarrollado en el departamento DSIC de la UPV). Para realizar la automatización de pruebas se ha elegido la herramienta Protractor, un *framework* para pruebas automatizadas específico para AngularJS. Mediante Protractor se desarrollarán y ejecutarán automáticamente las pruebas, obteniendo un informe con los resultados, todo ello integrado en WebStorm.

Protractor trae un mecanismo muy rudimentario para lanzar las pruebas; necesita de un fichero de configuración donde se indiquen las rutas de los scripts de pruebas que se quieren ejecutar. En el marco del proceso de desarrollo ágil con el que se trabaja interesa poder ejecutar suites de pruebas (ver Glosario) a medida de la situación. Por ejemplo, ejecutar solo las pruebas de la parte de la aplicación afectada por un cambio, o ejecutar todas las pruebas automatizadas del conjunto de partes de la aplicación afectadas por todos los cambios en un Sprint (ver Glosario). En Protractor no existe funcionalidad específica para realizar dicha definición de suites.

2.1. Objetivos

Según lo expuesto en la sección anterior, los objetivos de este trabajo de fin de grado son:

- Definir un proceso de automatización de pruebas que se integre con el enfoque TUNE-UP.
- Aplicar y evaluar el uso de Protractor para la aplicación automática de pruebas de regresión.
- Diseñar y construir un gestor de suites de pruebas de regresión que genere el fichero de configuración que requiere Protractor para lanzar las pruebas automatizadas.

3. Desarrollo ágil de software

"El desarrollo de software no es una tarea fácil. Prueba de ello es que existen numerosas propuestas metodológicas que inciden en distintas dimensiones del proceso de desarrollo. Por una parte tenemos aquellas propuestas más tradicionales que se centran especialmente en el control del proceso, estableciendo rigurosamente las actividades involucradas, los artefactos que se deben producir, y las herramientas y notaciones que se usarán. Estas propuestas han demostrado ser efectivas y necesarias en un gran número de proyectos, pero también han presentado problemas en otros muchos. Una posible mejora es incluir en los procesos de desarrollo más actividades, más artefactos y más restricciones, basándose en los puntos débiles detectados. Sin embargo, el resultado final sería un proceso de desarrollo más complejo que puede incluso limitar la propia habilidad del equipo para llevar a cabo el proyecto. Otra aproximación es centrarse en otras dimensiones, como por ejemplo el factor humano o el producto software. Esta es la filosofía de las metodologías ágiles, las cuales dan mayor valor al individuo, a la colaboración con el cliente y al des arrollo incremental del software con iteraciones muy cortas. Este enfoque está mostrando su efectividad en proyectos con requisitos muy cambiantes y cuando se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad. Las metodologías ágiles están revolucionando la manera de producir software, y a la vez generando un amplio debate entre sus seguidores y quienes por escepticismo o convencimiento no las ven como alternativa para las metodologías tradicionales." [1].

Como ya se ha dicho en el capítulo anterior, el contexto de este TFG es un proyecto de desarrollo de un producto dirigido al sector sanitario, y en este trabajo se va a explicar todo el marco de trabajo utilizado en el equipo de *testing*, centrándonos en la automatización de pruebas de aceptación (PAs). En dicho proyecto se ha optado por seguir un desarrollo ágil centrado en las pruebas de aceptación. Para ello, haremos uso de TUNE-UP Process.

TUNE-UP es una metodología y su herramienta, ambas dirigidas a apoyar la gestión ágil de proyectos de desarrollo de software. Incorpora los principales

elementos de las metodologías ágiles lo cual le permite ser configurada para trabajar siguiendo las prácticas de los principales métodos ágiles. Sin embargo, las características de TUNE-UP asociadas a la gestión del proyecto van más allá que las ofrecidas por las metodologías y herramientas ágiles, lo cual permite complementar la forma de trabajo del equipo con aspectos no incluidos en Scrum o XP. Para la organización y fácil acceso al trabajo del equipo TUNE-UP aporta una variación de la técnica Kanban, la cual se integra con un simple pero potente mecanismo de workflows flexibles, permitiendo orquestar de forma automática gran parte de la colaboración necesaria entre actividades.

La base fundamental que seguimos con esta metodología consiste en dividir el trabajo completo en bloques que pueden ser abordados en periodos cortos de tiempo (entre dos y cuatro semanas), denominados *Sprints*. Cada Sprint puede contener un número variable de unidades de trabajo (ver Glosario), pues cada UT puede necesitar un esfuerzo diferente.

Una unidad de trabajo es cada cambio elemental que se realice sobre el producto. Usamos tres tipos de UT: nuevo requisito, mejora y corrección de fallo; y contienen pruebas de aceptación, que son básicamente la especificación de requisitos. Cada UT atraviesa por diferentes actividades, conocido como *workflow* o flujo de trabajo. En la Figura 1 se muestra el *workflow* de desarrollo que se está aplicando actualmente.

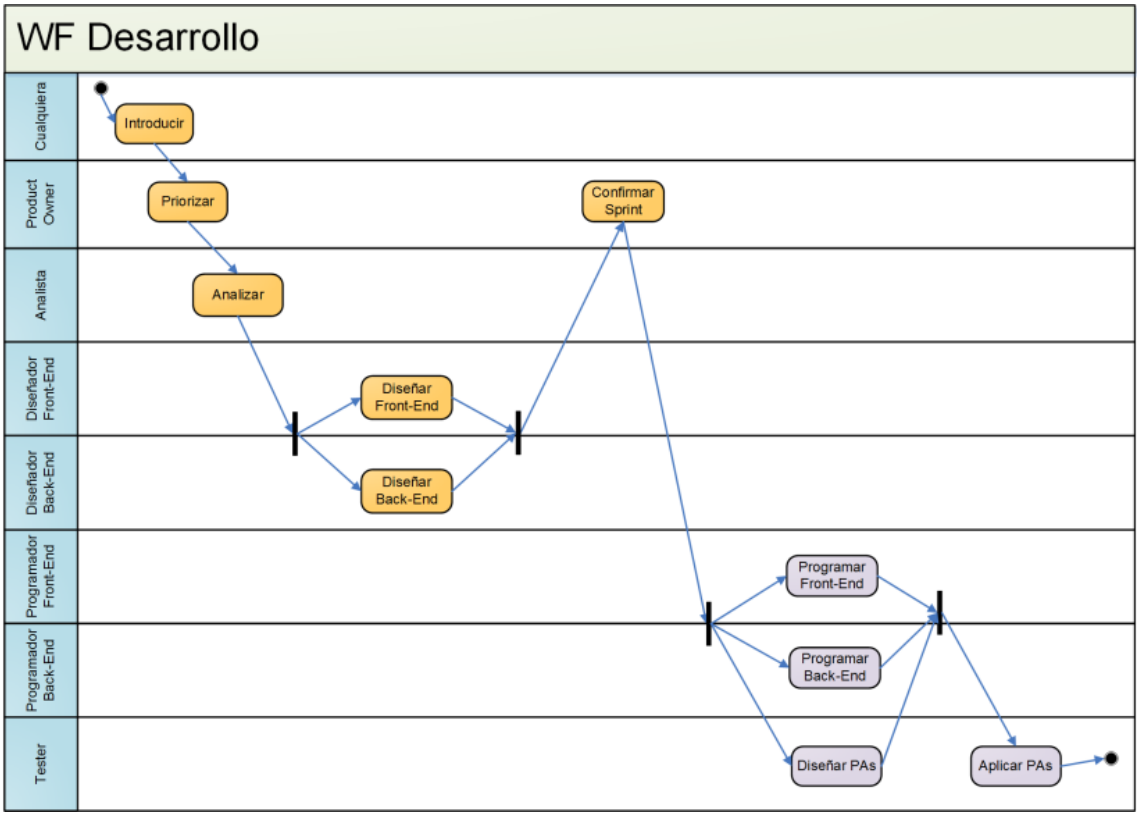


Figura 1. Flujo de trabajo

- Introducir: Se establece el nombre de la UT y diferentes configuraciones como las relaciones con otras UTs o al ámbito al que pertenecen.
- Priorizar: El *product owner* (cliente) debe valorar la importancia de la UT, asignando un orden en el contexto de todo el trabajo pendiente en el *Backlog* (lista de UTs priorizadas y sin implementar).
- Analizar: El analista crea la documentación necesaria e introduce las PAs en el sistema.
- Diseñar *front-end*/Diseñar *back-end* (ver Glosario): Se llevan a cabo los diseños del análisis.
- Confirmar Sprint: En este punto la UT se encuentra lista para ser introducida en próximos Sprints.
- Programar *front-end*/Programar *back-end*: Se lleva a cabo toda la programación, tanto de la parte *front-end* como de la parte *back-end*, en base a las pruebas de aceptación.
- Diseñar PAs: Esta actividad la realizamos en paralelo con la anterior y consiste en realizar todos los diseños que precise cada PA.

- Aplicar PAs: Una vez se ha terminado de programar, el siguiente paso es probar la aplicación, comprobando una por una que todas las PAs se han implementado correctamente. Si la prueba es correcta, esta se marca como OK. En caso contrario, se marca como KO y se devuelve la UT a la actividad de programación.
- Terminar: Estado final de una UT terminada.

Como puede verse en la Figura 2, una PA tiene un ciclo de vida, y todo el desarrollo está dirigido por las pruebas de aceptación.

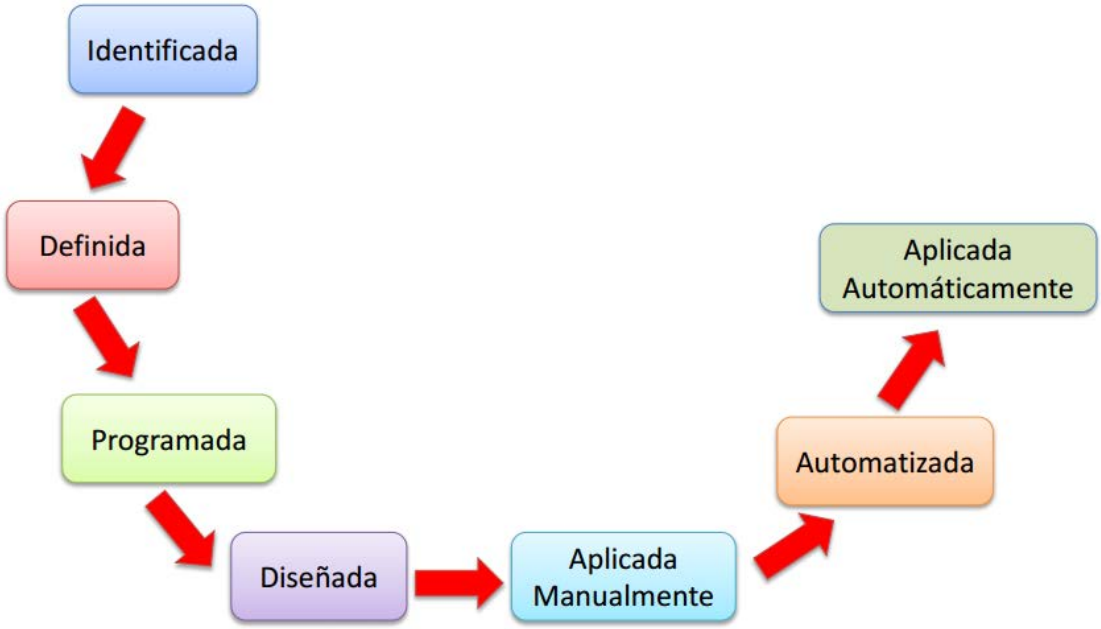


Figura 2. Ciclo de vida de una prueba de aceptación

Así, cuando una unidad de trabajo es finalizada, las pruebas de aceptación han sido marcadas como realizadas con éxito. No obstante, al hacer cambios en el software es posible introducir fallos en funcionalidad ya existente y, por ello, aparecen las pruebas de regresión, que son pruebas que se deben realizar cada cierto tiempo para comprobar que el funcionamiento del producto sigue siendo correcto. Conforme un proyecto va creciendo, el número de pruebas de regresión aumenta considerablemente, y es por ello por lo que se tiene en cuenta la automatización de pruebas, que es el trabajo en el que nos embarcamos. Por tanto, las pruebas automatizadas son pruebas de regresión.

No obstante, en el desarrollo de software, la cantidad de pruebas de aceptación que pueden automatizarse es inabordable desde el punto de vista de

los costes. Podríamos dedicar horas y horas a automatizar, por lo que no sería rentable. Por ello, debemos hacer balance de qué pruebas son las adecuadas para ser automatizadas. Sin embargo, no existe ningún criterio para tomar la decisión ya que son muchos los factores que intervienen.

Este acuerdo se debe valorar entre el analista y el *tester*. Se debe analizar qué pruebas tienen más riesgo de fallar a lo largo del tiempo, qué pruebas se encargan de las partes críticas de la aplicación, y el coste que tendría automatizar la prueba, pues a veces es más rentable probar algo a mano que hacer la automatización. Debemos tener siempre en mente que el objetivo principal de las pruebas automatizadas es rentabilizar su coste.

Esto es lo que ocurre en el proyecto en el que nos encontramos. Entre los analistas, el *product owner* y los *testers* se deciden qué pruebas deben ser automatizadas. Es entonces cuando se empieza a programar. Al finalizar la automatización de la prueba, en TUNE-UP cambiamos el estado de la prueba a 'Automatizada', algo de lo que haremos uso en el siguiente capítulo.

PA	Orden	Tipo	Nombre PA	Estado	Modificada	Sprint	UT
13-1			[redacted]	Automatizada	04/05/2016 15:52:55	Sprint 3	2
31-1			[redacted]	Automatizada	04/05/2016 15:53:43	Sprint 3	2
1194-1			[redacted]	Definida	09/05/2016 19:12:43	Sprint 6	1143
1118-1			[redacted]	Diseñada	21/04/2016 18:14:15	Sprint 4	1116

Figura 3. Extracto de pruebas de aceptación en TUNE-UP

Como ya sabemos, las pruebas automatizadas son pruebas de regresión, las cuales se ejecutan cada cierto período para comprobar que todos los componentes de la aplicación siguen funcionando correctamente, y facilitan bastante la tarea del *tester*. No obstante, tener automatizadas las pruebas conlleva realizar una serie de tareas. Una vez ejecutadas, debemos analizar los resultados y, si ha fallado alguna, entender por qué. El tiempo que dedicamos a analizar los resultados también se ha de tener en cuenta ya que es importante para valorar el esfuerzo invertido.

Supongamos que hemos ejecutado una *suite* de pruebas y una de ellas ha fallado. Tras haberla analizado y haber detectado el defecto, actuaremos de forma muy parecida a cómo actuamos con una prueba de aceptación manual:

- Si el fallo se corresponde con una UT abierta del Sprint actual (imaginemos que se está mejorando un componente y se ha cambiado algo que ha provocado el fallo), crearemos una nueva prueba de aceptación.
- Si el fallo se corresponde con una UT ya cerrada, crearemos una nueva UT de tipo ‘Corrección Fallo’.

De este modo, tras haber puesto en contexto este trabajo de fin de grado, podemos comenzar con la automatización de pruebas. Dado que la aplicación web se está desarrollando con AngularJS, vamos a utilizar Protractor, una herramienta desarrollada por Google al igual que AngularJS y construida especialmente para dicho *framework* [2].

4. Pruebas automatizadas con Protractor

4.1. Instalación y primeros pasos

Protractor es un *framework* de pruebas *end-to-end* para aplicaciones en AngularJS [3]. Ejecuta los tests que hayamos programado en un navegador web

real, interactuando como lo haría un usuario. Uno de los mayores problemas es la localización de elementos en una página web, ya que si no están bien definidos, las pruebas podrían fallar al hacer modificaciones en la web. Esto no ocurre con Protractor dado que nos proporciona estrategias de localización específicas para AngularJS, permitiéndonos localizar dichos elementos de forma única. Además, como las aplicaciones en AngularJS funciona de forma dinámica, Protractor se encarga de la sincronización, por lo que no tenemos que preocuparnos de añadir '*waits*' y '*sleeps*', sino que se espera a que la web finalice las tareas que tenga pendiente.

Para poder comenzar a utilizar Protractor, debemos realizar los siguientes pasos [4]:

1) Protractor es un programa que se basa en Node.js, por lo que para poder ejecutarlo debemos tenerlo instalado. Así, lo descargamos de Internet (https://nodejs.org/en/) y lo instalamos.
2) Comprobamos que Node.js se ha instalado correctamente escribiendo en la consola:

   npm --version

3) Instalamos Protractor:

   npm install -g protractor

4) Comprobamos que Protractor se ha instalado correctamente escribiendo en la consola:

   protractor –version

5) Con:

   npm -g list

   Se nos muestra la lista de módulos instalados. Comprobamos que se incluye *'selenium-webdriver'*.

6) Antes de continuar, deberíamos tener instalado el JDK (Java SE Development Kit) y Google Chrome para el correcto funcionamiento de las pruebas.

7) Al instalar Protractor también hemos instalado *'webdriver-manager'*, una herramienta para obtener una instancia de Selenium Server ejecutándose. Debemos actualizarla a la última versión con:

webdriver-manager update

Veremos que se actualiza Selenium Standalone y Chromedriver (que es el que utilizará Selenium en lugar de otros como el de Firefox).

8) Iniciamos *'webdriver-manager'* con:

webdriver-manager start

9) Abrimos una nueva consola para ver un ejemplo de Protractor. Estando en nuestra carpeta personal (C:\Users\<tuUsuario>) escribimos:

cd AppData/Roaming/npm/node_modules/protractor/example

Veremos que hay dos archivos, los cuales se muestran en las Figuras 4 y 5.

```
exports.config = {
  directConnect: true,

  // Capabilities to be passed to the webdriver instance.
  capabilities: {
    'browserName': 'chrome'
  },

  // Framework to use. Jasmine is recommended.
  framework: 'jasmine',

  // Spec patterns are relative to the current working directly when
  // protractor is called.
  specs: ['test_spec.js'],

  // Options to be passed to Jasmine.
  jasmineNodeOpts: {
    defaultTimeoutInterval: 30000
  }
};
```

Figura 4. Archivo 'conf.js' (prueba de ejemplo)

```
describe('angularjs homepage', function() {
  it('should greet the named user', function() {
    browser.get('http://www.angularjs.org');

    element(by.model('yourName')).sendKeys('Julie');

    var greeting = element(by.binding('yourName'));

    expect(greeting.getText()).toEqual('Hello Julie!');
  });
});
```

Figura 5. Extracto de 'example_spec.js' (prueba de ejemplo)

El archivo 'conf.js' contiene la configuración necesaria para lanzar las pruebas y el archivo 'example_spec.js' contiene las pruebas que vamos a realizar. Para que Protractor pueda ejecutar las pruebas (podemos ejecutar tantos archivos como queramos a la vez) debemos incluir su ruta en el atributo 'specs' del archivo 'conf.js'. Una vez lo tengamos listo, tecleamos:

protractor conf.js

Así, veremos que automáticamente se abre Google Chrome, realiza las pruebas y se cierra. En la consola debe aparecer que el test se ha superado correctamente (en caso de que así sea).

Una vez conocemos cómo utilizar Protractor, el siguiente paso es integrarlo en un entorno de desarrollo. En nuestro caso, haremos uso de WebStorm, un entorno de desarrollo integrado para JavaScript ligero y potente, equipado para el desarrollo del lado del cliente y del lado del servidor con Node.js. Podemos descargarlo de su página web oficial (https://www.jetbrains.com/webstorm/).

4.2. Integración en WebStorm

Lo primero que debemos hacer es crear un nuevo proyecto e incluir en él el archivo de configuración de Protractor y la prueba [5]. Pulsamos en 'Edit Configurations…' en el desplegable de la parte superior derecha que mostramos en la Figura 6.

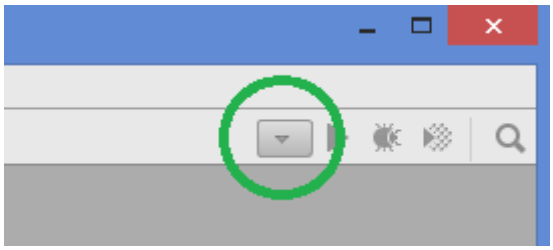


Figura 6. Botón para acceder a la configuración de ejecución en WebStorm

Se abrirá una nueva ventana. Pulsamos sobre el '+' de la esquina superior izquierda y seleccionamos 'Node.js'.

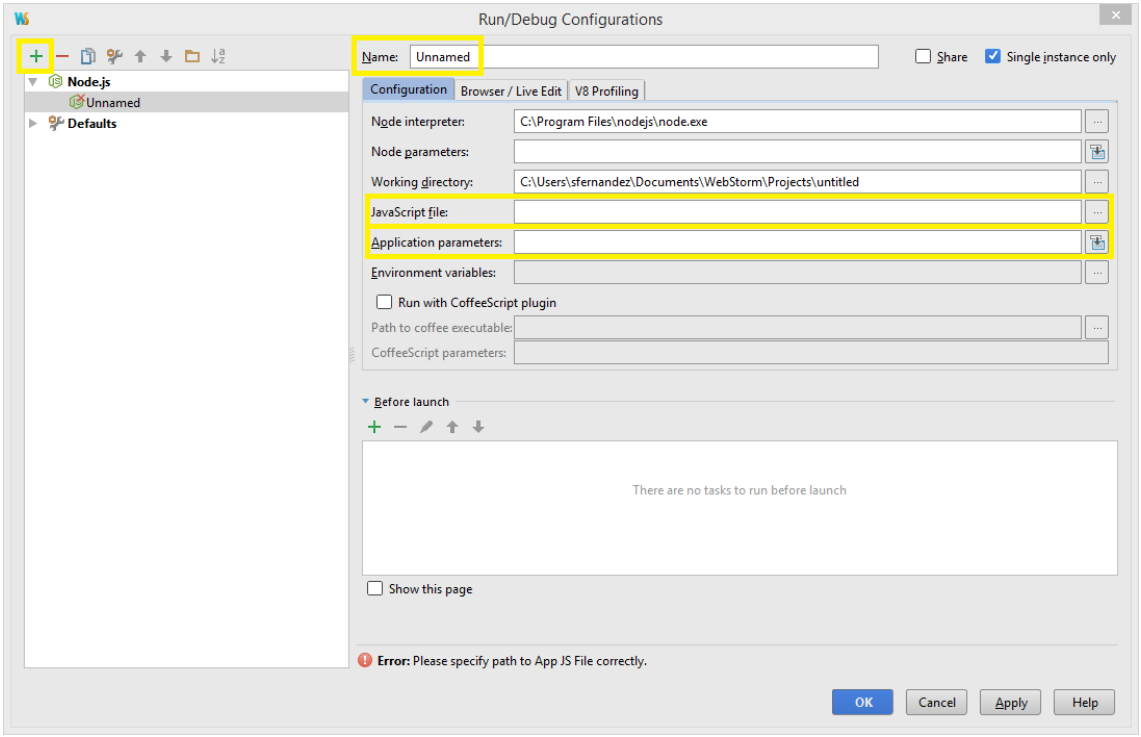


Figura 7. Configuración de ejecución en WebStorm

Debemos rellenar 3 campos:

1) *Name*: 'Protractor'

2) *JavaScript file*: Ruta del archivo 'cli.js'.
   El archivo se encuentra en:
   "C:\Users\<tuNombreDeUsuario>\AppData\Roaming\npm\node_modules\protractor\lib\cli.js"
3) *Application parameters*: Ruta del archivo 'conf.js' que utilizaremos para lanzar las pruebas.
4) Deberíamos tener dicho archivo en el proyecto, por lo que pulsando con el botón derecho sobre él podemos copiar directamente la ruta con 'Copy Path'.

Finalmente, la ventana queda como mostramos en la Figura 8.

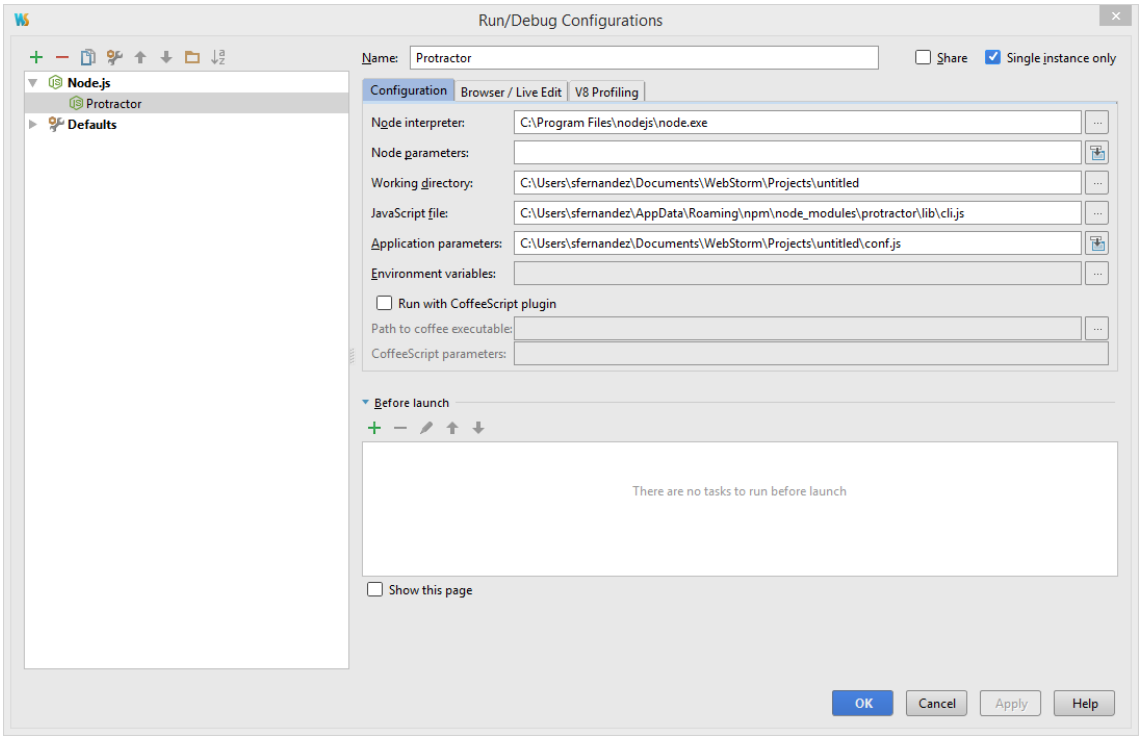


Figura 8. Configuración de ejecución completada en WebStorm

De esta forma, ya tenemos configurado Protractor en WebStorm y podemos comenzar a utilizarlo.

4.3. Localización y manipulación de elementos

Gracias a Protractor podemos hacer uso de funciones para la identificación de elementos en una página web. No obstante, algunos elementos

no son tan sencillos de identificar si no tenemos los suficientes conocimientos de JavaScript.

Para localizar un elemento se utiliza la función 'element()', que nos devolverá el objeto deseado (o el primero del array resultante si encuentra más de uno). Tendremos que pasarle como parámetro el localizador apropiado. Por ejemplo, si deseamos buscar un botón con el id 'acceptButton' tendríamos que

```
var button = element(by.id('acceptButton'));
```

escribir el código:

Figura 9. Localización de un botón por id

Los localizadores comienzan con 'by' seguido de la función apropiada para el elemento que deseamos buscar. Es posible localizar elementos a partir de bastantes atributos en AngularJS, aunque estos son los más utilizados: 'ng-bind', 'ng-model', 'ng-repeat' o 'id'. En Protractor los buscamos de la siguiente

```
var w = element(by.binding('<ng-bind-attribute>'));
var x = element(by.model('<ng-model-attribute>'));
var y = element.all(by.repeater('<ng-repeat-attribute>')).first();
var z = element(by.id('<id-attribute>'));
```

forma:

Figura 10. Localización de diferentes elementos

Como puede observarse en el código, al utilizar el localizador 'repeater' hemos cogido el primer elemento. Esto se debe a que nos proporciona todos los elementos de una lista, por lo que deberíamos procesarla para poder obtenerlos todos. Como aquí entran en juego las promesas (utilizadas para computaciones asíncronas), veremos cómo utilizar este localizador más adelante.

Al tener los elementos ya localizados, podemos hacer uso de ellos. Los controles más sencillos de una página web son los campos de introducción de

```
Input -> element(by.id('<id-attribute>')).sendKeys("textoEjemplo");
Botón -> element(by.id('<id-attribute>')).click();
```

texto (*input*) y los botones, en los que deseamos introducir texto y pulsarlos, respectivamente.

Figura 11. Uso de un input y un botón

Con 'sendKeys("textoEjemplo")' escribiríamos "textoEjemplo" en el input, y con 'click()' pulsaríamos el botón. Supongamos que en el input en el que deseamos escribir ya hay texto y queremos borrarlo. La forma de hacerlo sería

```
element(by.id('<id-attribute>'))
.sendKeys(protractor.Key.chord(protractor.Key.CONTROL, "a"), "textoEjemplo");
```

la siguiente:

Figura 12. Uso extra de un input

Lo que acabamos de realizar es la pulsación de las teclas 'Control' y 'A' a la vez, consiguiendo que el texto que haya en el input se seleccione, borrándose al escribir "textoEjemplo".

Con lo explicado en esta sección, ya somos capaces de localizar elementos básicos y hacer uso de ellos, por lo que podríamos crear pruebas básicas de formulario completando todos los campos y pulsando el botón de aceptar. Al final de la prueba colocaríamos un 'expect' que nos permitiría comprobar si la web tiene el comportamiento deseado.

```
describe('Mi primera prueba', function() {
    it('Comprobar título', function() {
        element(by.id('<id-attribute1>')).sendKeys("textoEjemplo");
        element(by.id('<id-attribute2>')).click();
        expect(browser.getTitle()).toEqual('Título 2');
    });
});
```

Figura 13. Una prueba básica

Así sería una prueba en Protractor. La sintaxis con 'it' y 'describe' proviene del *framework Jasmine*. Para comprobar que obtenemos el resultado esperado utilizamos 'expect'. En este caso, estamos comprobando que tras rellenar un campo de texto y pulsar un botón, el título de la página en la que nos

encontremos es ‘Título 2’. Si el título es realmente ese, la prueba pasará sin problemas, pero si no es correcto la prueba fallará.

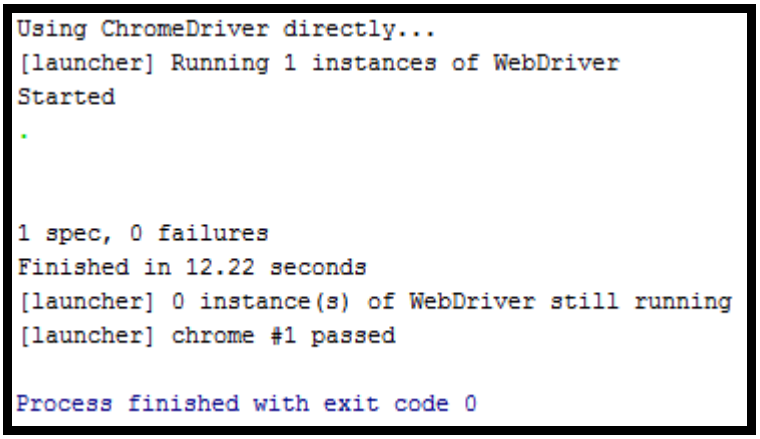


Figura 14. Resultado de una prueba en la consola de WebStorm

Como puede verse en la Figura 14, se ha ejecutado la prueba en Google Chrome y ha sido superada correctamente.

4.4. Patrón de diseño: *Page Objects*

A la hora de desarrollar numerosas pruebas, un patrón bastante utilizado es el conocido como *Page Objects* [6]. Este patrón nos permite escribir pruebas de forma más limpia encapsulando información de nuestra aplicación. Un *Page Object* puede ser reutilizado en todas las pruebas que queramos y, si la aplicación web cambiara, tan solo deberíamos modificar el *Page Object* y todas las pruebas seguirían funcionando correctamente. Así, conseguimos una capa extra con la que podemos separar las pruebas en sí del tratado de datos de la web. Veamos un ejemplo, utilizando una parte reducida de datos reales de este

```
var main = function() {
  this.changeData = function(data, id){
    if(data) {
      element(by.id(id))
      .sendKeys(protractor.Key.chord(protractor.Key.CONTROL, "a"), data);
    }
  }
  this.changeDataEnter = function(data, id){
    if(data) {
      this.changeData(data, id);
      browser.sleep(600);
      browser.actions().sendKeys(protractor.Key.ENTER).perform();
    }
  }
  this.click = function(id){
    element(by.id(id)).click();
  }
};
module.exports = new main();
```

trabajo:

Figura 15. Archivo ‘main.js’

Observamos tres funciones:

- *changeData*

Recibe como parámetro el id del elemento que queremos buscar (un input en este caso) y el texto que deseamos escribir en el campo.

- *changeDataEnter*

  Es una extensión de la función anterior. Hay campos en los que tenemos una lista desplegable y se filtran los resultados conforme se va escribiendo en ellos, por lo que una vez hemos escrito el texto, se necesita pulsar *Enter* para elegir la opción del desplegable. Además, estamos hablando de un proyecto en el que todas las consultas a base de datos se realizan en el *back-end*, por lo que hay que hacer la petición desde el *front-end*, con el consecuente tiempo de espera hasta recibir la respuesta (el tiempo de espera es despreciable en la navegación a nivel de usuario, pero hay que tenerla en cuenta a la hora de realizar las pruebas automáticas). Por ello, introducimos también una espera de 600 ms.

- *click*

  Recibe como parámetro el id del elemento que estamos buscando y pulsa sobre él. Puede parecer inútil poner esta única línea de código en un *Page Object*, pero veremos como las pruebas quedan mucho más elegantes y limpias. Además, reiteramos que estamos implantando una capa extra para separar las pruebas de las características propias de la web.

Haríamos uso de estos métodos de la siguiente forma, buscando el elemento con id ‘name’ y escribiendo ‘Sergio’ en el campo.

```
var main = require(<ruta al archivo main.js>);
main.changeData('Sergio', 'name');
```

Figura 16. Uso de un *page object*

Y ahora nos hacemos la siguiente pregunta: ¿qué pasaría si el archivo ‘main.js’ cambia de ubicación? Tendríamos que cambiar la ruta del ‘require’. Pero, ¿y si hemos usado el archivo en más de, por ejemplo, veinte pruebas? Tendríamos que ir cambiándolo en cada prueba una por una. Y no solo eso, sino que tendríamos que encontrar primero dónde hemos hecho uso del archivo. Llegaría un momento que sería inabordable. Para resolver este problema hemos

tenido que pensar cuál es el archivo que no va a cambiar nunca de ubicación, y partir de él. Solo hay un archivo 'conf.js' por proyecto y siempre lo tendremos en la misma ubicación, por lo que obtenemos una ruta base en él:

Figura 17. Obtención de la ruta base en 'conf.js'

Añadiendo lo anterior a nuestro 'conf.js', obtendremos una variable global

```
beforeLaunch: function() {
    global.basePath = __dirname;
}
```

```
{
     "init": "\\libs\\init"
   , "main": "\\libs\\main"
   , "search": "\\libs\\search"
   , "personModule": "\\libs\\personModule\\personModule"
   , "personalData": "\\libs\\personModule\\personalData"
   , "location": "\\libs\\personModule\\location"
}
```

con una ruta base, a partir de la cual podremos formar las demás. Para ello, hemos creado el archivo 'routes.json' (ver Glosario) incluyendo la segunda parte de las rutas necesarias:

Figura 18. Extracto del archivo 'routes.json'

En la Figura 18 se muestran algunas de las rutas utilizadas en el proyecto (no es necesario poner '.js' al final de cada ruta), y en la Figura 19 la jerarquía utilizada (el archivo 'routes.json' se encuentra dentro de la carpeta 'libs').

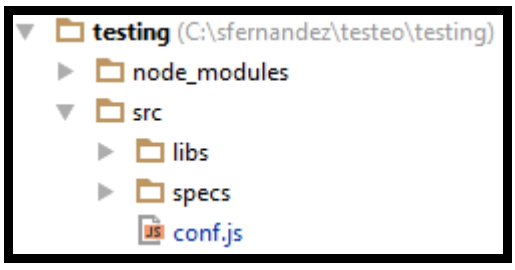


Figura 19. Jerarquía parcial del proyecto

Como puede observarse, hay una carpeta llamada 'node_modules' de la cual no hemos hablado. Lo haremos más adelante, cuando expliquemos la función de obtener informes con los resultados de las pruebas automáticamente.

En este punto, ya tenemos la primera parte de la ruta en la variable global ‘basePath’ y la segunda parte de las rutas en el archivo ‘routes.json’, por lo que ya tenemos resuelto el problema. Modificamos el archivo ‘conf.js’ de la siguiente forma:

```
beforeLaunch: function() {
    global.basePath = __dirname;
    global.routes = require(basePath+'\\libs\\routes.json');
}
```

Figura 20. Archivo ‘routes.json’ incluido en ‘conf.js’

Así, partiendo de la Figura 16, haremos uso de las funciones de los *Page Object* con:

```
var main = require(basePath+routes.main);
main.changeData('Sergio', 'name');
```

Figura 21. Uso de las rutas

Si quisiéramos acceder a otra ruta, tan solo tendríamos que poner la clave correspondiente. Por ejemplo, pondríamos ‘routes.init’ para acceder a la ruta de ‘init.js’, como vemos en la Figura 18.

Teniendo ya claro cómo funcionan los *Page Object*, vamos a añadir algo más de complejidad: la herencia.

4.5. *Prototype*

En JavaScript no existe la herencia tal y como se conoce en otros lenguajes de programación, sino que es una programación basada en prototipos [7]. En este estilo de programación las clases no están presentes, por lo que la reutilización de comportamiento se lleva a cabo a través de un proceso de decoración de objetos existentes que sirven de prototipos.

“Todos los objetos en JavaScript tienen un enlace interno a otro objeto: su prototipo. Ese objeto prototipo tiene su propio prototipo, y así sucesivamente

hasta que se alcanza un objeto cuyo prototipo es *null*. Por definición, *null* no tiene prototipo, y actúa como el final de esta cadena de prototipos.” [8].

Para nuestro proyecto, hemos implantado la herencia en tres niveles del siguiente modo:

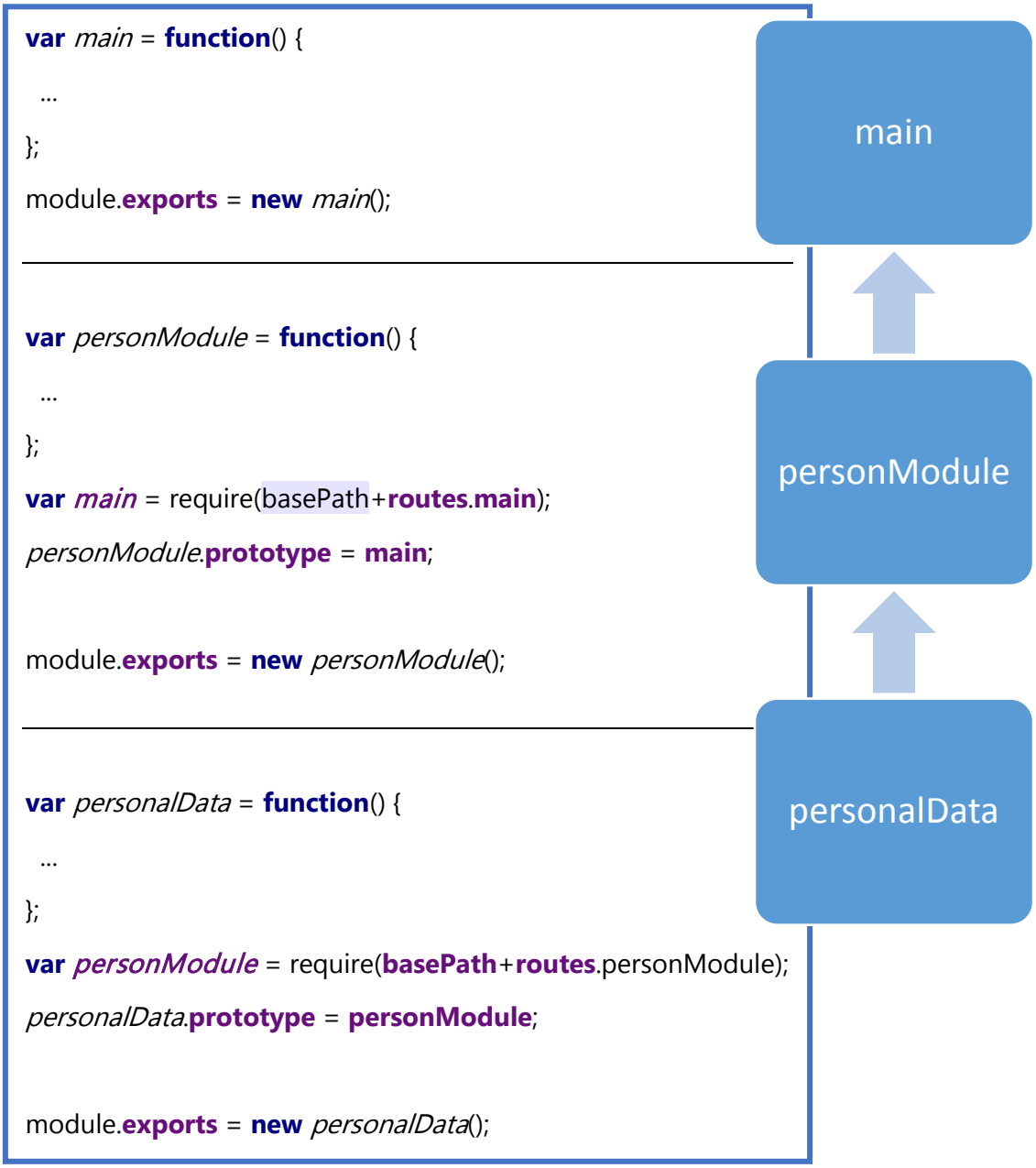


Figura 22. Herencia en tres niveles en JavaScript

Al haber realizado este trabajo en el contexto de una empresa privada, los nombres de las clases han sido modificados, intentando poner nombres clarificadores para que se comprenda correctamente cómo está funcionando el código. Simplemente, tenemos un objeto *main* que no hereda de nadie; un objeto *personModule* que hereda de *main*; y objetos como *personalData* y *location* que heredan de *personModule*. A su vez, personalData y location son *Page Objects* de los que nuestras pruebas automatizadas hacen uso. Con esta distribución de código, conseguimos no tener duplicidad, llamando a una clase superior si vamos a realizar una acción que es común a otro objeto. En caso de que la función a la que llamemos no se encuentre en el objeto inmediatamente superior, JavaScript seguiría la cadena de prototipos hasta encontrarla. Teniendo ya la cadena de prototipos establecida, podemos llamar a cualquier función como si se encontrara en el mismo objeto: utilizando *this*. Hemos visto en la Figura 15 que el objeto *main* incluye una función llamada *click(id)*, de la que podemos hacer uso en, por ejemplo, *personalData*:

```
var personalData = function() {
  this.functionA = function() {
    ...
    var id = 'buttonId';
    this.click(id);
    ...
  }
};
var his_mpi = require(basePath+routes.his_mpi);
personalData.prototype = his_mpi;

module.exports = new personalData();
```

Figura 23. Uso de ‘click(id)’ en ‘personalData’

4.6. Diseño de las pruebas

Tras comprender y conocer todos los conceptos de los que vamos a hacer uso, podemos comenzar a desarrollar nuestra primera prueba automatizada.

La prueba consiste en rellenar un formulario de datos personales y comprobar que el guardado resulta un éxito. Veamos en primer lugar el código y la explicación a continuación.

```
describe('Página de datos personales', function() {
    beforeAll(function() {
        browser.get('<url>');
    });
    it('Formulario y guardado', function() {
        var personalData = require(basePath+routes.personalData);
        var search = require(basePath+routes.search);

        var person = new Object();
        person.id = '18';
        person.name = 'Sergio';
        person.surname1 = 'Fernández';
        person.surname2 = 'Rodríguez';
        person.gender = 'Masculino';
        person.birthday = '11/08/1994';
        person.nationality = 'Española';
        person.country = 'España';
        person.birthPlace = 'Benidorm';
        person.language = 'Español';
        person.email = 'serferr5@inf.upv.es';

        search.load (person.id);
        personalData.editionMode();
        personalData.update(person);
        personalData.saveProfile();

        var toasts = require(basePath+routes.toasts);
        expect(toasts.checkSaveOK()).toEqual(true);
    });
});
```

Figura 24. Prueba para comprobar el funcionamiento correcto de un formulario

En primer lugar, le decimos al navegador la dirección web que debe abrir. Podríamos ponerlo en el 'it', pero como podemos tener más de un 'it' en el mismo 'describe', lo ponemos dentro de 'beforeAll()' para evitar la duplicidad de código.

En segundo lugar, obtenemos acceso a los objetos *personalData* y *search* ya que vamos a utilizarlos para la prueba.

En tercer lugar, creamos el objeto particular para esta prueba. Como queremos rellenar un formulario de datos personales, hemos creado un objeto cuyos atributos son los campos a completar del formulario.

En cuarto lugar, vemos como los *Page Objects* nos facilitan enormemente el trabajo. Simplemente leyendo, podemos comprender qué realiza cada línea de código, ya que todo el trabajo de localización de elementos se ha llevado a *personalData.js* que, además, hace uso de la herencia con *prototype*.

```
search.load (person.id); //Busca a la persona mediante su 'id'.
personalData.editionMode(); //Cambia el formulario de modo consulta a edición.
personalData.update(person); //Completa el formulario con los datos de 'person'.
personalData.saveProfile();//Pulsa el botón de guardar.
```

Figura 25. Uso de *page objects* en una prueba

Echemos un vistazo a parte de la función *update* de personalData para ver cómo localizamos los elementos.

```
this.update = function(patient) {
    this.changeData(patient.name, 'nameId');
    this.changeData(patient.surname1, 'surname1Id');
    this.changeData(patient.surname2, 'surname2Id');
    this.changeDataEnter(patient.gender, 'genderId');
    this.changeData(patient.birthday, 'birthdayId');
    ...
}
```

Figura 26. Localización de elementos mediante *page objects* en una prueba

En quinto y último lugar, hacemos la comprobación que determina si la prueba es superada o no. En nuestro caso, hemos programado otro módulo llamado *toasts* para obtener el resultado.

4.7. Generación automática de informes

A medida que un proyecto crece, va aumentando el número de pruebas automatizadas a ejecutar, por lo que llegará un momento en el que no será viable ver los resultados en la consola de WebStorm. Para evitar esto, vamos a instalar y configurar un paquete que nos facilitará enormemente la tarea, creando informes con los resultados de las pruebas.

El paquete adecuado a nuestro proyecto es 'protractor-jasmine2-screenshot-reporter', ya que estamos utilizando Jasmine 2 con Protractor [9]. Además de recopilar los resultados en un informe HTML, este paquete realiza capturas de pantalla tras ejecutar cada test individual, por lo que podremos ver el estado en el que queda nuestra aplicación tras realizarse el test.

Empecemos con la descarga e instalación del paquete. En la consola de WebStorm, escribimos:

```
npm install protractor-jasmine2-screenshot-reporter --save-dev
```

Veremos que nos crea una carpeta en la raíz del proyecto llamada 'node_modules', la cual incluye el paquete. Para configurarlo, abrimos el archivo 'conf.js' y lo completamos del siguiente modo:

```
var HtmlScreenshotReporter = require('protractor-jasmine2-screenshot-
reporter');
var reporter = new HtmlScreenshotReporter({
    dest: '../reporter',
    filename: 'report.html'
});
```

Figura 27-1. Configuración del *reporter* en 'conf.js'

```
exports.config = {
  ...
  // Setup the report before any tests start
  beforeLaunch: function() {
     global.basePath = __dirname;
     global.routes = require(basePath+'\\libs\\routes.json');

     return new Promise(function(resolve){
        reporter.beforeLaunch(resolve);
     });
  },
  // Assign the test reporter to each running instance
  onPrepare: function() {
     jasmine.getEnv().addReporter(reporter);
  },
  // Close the report after all tests finish
  afterLaunch: function(exitCode) {
     return new Promise(function(resolve){
        reporter.afterLaunch(resolve.bind(this, exitCode));
     });
  }
};
```

Figura 27-2. Configuración del *reporter* en 'conf.js'

Como puede observarse, el informe se va a nombrar como 'report.html' y se guardará en la carpeta 'reporter'. Sin embargo, cada vez que ejecutemos alguna prueba, el informe se sobrescribirá, por lo que realizamos las siguientes modificaciones:

```
var d = new Date();
d = '['+d.getFullYear()+"-"+(d.getMonth()+1)+"-"
+d.getDate()+']('+d.getHours()+'.'+d.getMinutes()+'.'+d.getSeconds()+')';
var HtmlScreenshotReporter = require('protractor-jasmine2-screenshot-
reporter');
var reporter = new HtmlScreenshotReporter({
    dest: '../reporter/'+d,
    filename: 'report.html'
});


exports.config = {
  ...

};
```

Figura 28. Modificación del *reporter* para crear un archivo por ejecución

Así, obtenemos la fecha y hora actual y la formateamos a nuestro gusto. Añadimos la variable a la ruta de la carpeta del destino, por lo que a partir de ahora nos creará una carpeta cada vez que ejecutemos alguna prueba, y contendrá el informe HTML y las capturas de pantalla.

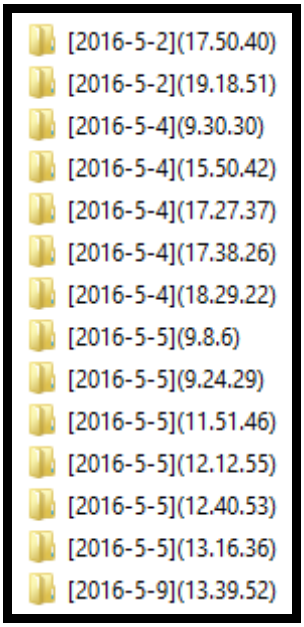
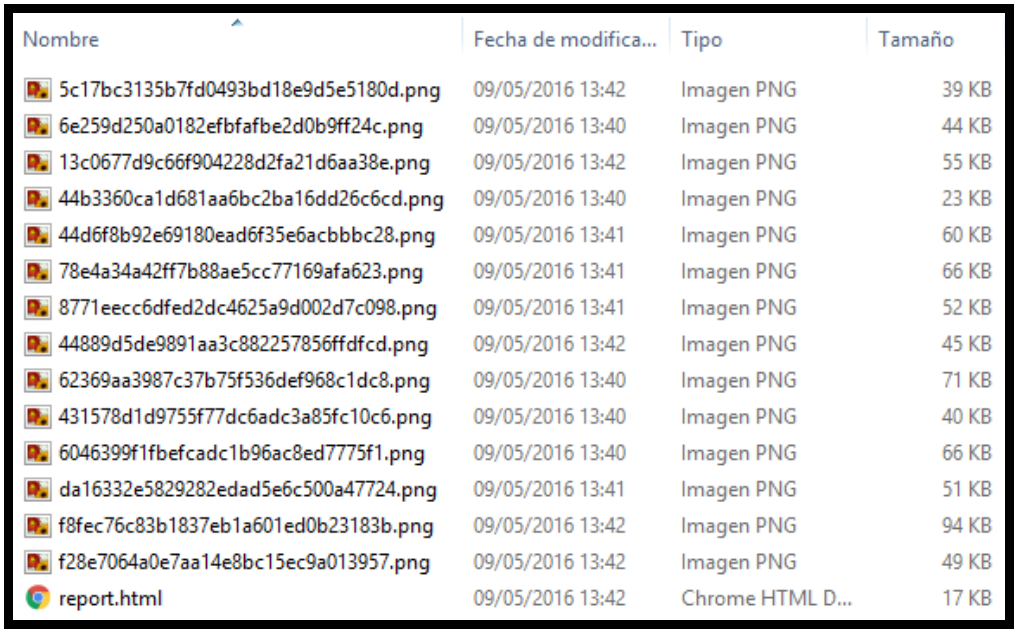


Figura 29. Una carpeta por cada ejecución de Protractor

Si accedemos dentro de una carpeta, veríamos tantas capturas como pruebas seleccionadas tuviera la ejecución, como se muestra en la Figura 30. En este caso tenemos catorce pruebas automatizadas.



Nombre	Fecha de modifica...	Tipo	Tamaño
5c17bc3135b7fd0493bd18e9d5e5180d.png	09/05/2016 13:42	Imagen PNG	39 KB
6e259d250a0182efbfafbe2d0b9ff24c.png	09/05/2016 13:40	Imagen PNG	44 KB
13c0677d9c66f904228d2fa21d6aa38e.png	09/05/2016 13:42	Imagen PNG	55 KB
44b3360ca1d681aa6bc2ba16dd26c6cd.png	09/05/2016 13:40	Imagen PNG	23 KB
44d6f8b92e69180ead6f35e6acbbbc28.png	09/05/2016 13:41	Imagen PNG	60 KB
78e4a34a42ff7b88ae5cc77169afa623.png	09/05/2016 13:41	Imagen PNG	66 KB
8771eecc6dfed2dc4625a9d002d7c098.png	09/05/2016 13:41	Imagen PNG	52 KB
44889d5de9891aa3c882257856ffdfcd.png	09/05/2016 13:42	Imagen PNG	45 KB
62369aa3987c37b75f536def968c1dc8.png	09/05/2016 13:40	Imagen PNG	71 KB
431578d1d9755f77dc6adc3a85fc10c6.png	09/05/2016 13:40	Imagen PNG	40 KB
6046399f1fbefcadc1b96ac8ed7775f1.png	09/05/2016 13:40	Imagen PNG	66 KB
da16332e5829282edad5e6c500a47724.png	09/05/2016 13:41	Imagen PNG	51 KB
f8fec76c83b1837eb1a601ed0b23183b.png	09/05/2016 13:42	Imagen PNG	94 KB
f28e7064a0e7aa14e8bc15ec9a013957.png	09/05/2016 13:42	Imagen PNG	49 KB
report.html	09/05/2016 13:42	Chrome HTML D...	17 KB

Figura 30. Contenido de una carpeta de pruebas

En la figura 31 podemos ver cómo es el informe que nos genera.

Report

Summary

- Total specs tested: 14
- Total failed: 3

(5 s)

✗ (2 s)
- Failed: No element found using locator: By.id(") [stack]

(11 s)

✗ (9 s)
- Expected false to equal true. [stack]

(10 s)

✗ (8 s)
- Expected false to equal true. [stack]

(10 s)

✓ (9 s)

(7 s)

✓ (6 s)

Figura 31. Extracto de un informe de resultados

En primer lugar, aparece el número de pruebas que se han realizado y el número de las que han fallado.

En segundo lugar, vemos todas las pruebas que se han ejecutado, una detrás de otra. Los títulos en negrita son los que en cada prueba hemos puesto en ‘describe’ y los subtítulos con aspa o *tick* son los ‘it’ que contenga cada ‘describe’. A la derecha de cada uno de ellos aparece el tiempo que ha tardado en ejecutarse.

En tercer lugar, si la prueba ha fallado, justo debajo se muestra el error. Si hacemos clic en ‘stack’ veremos la pila de errores y la ruta de donde han aparecido.

En cuarto y último lugar, sobre cada prueba hay un hipervínculo, y si accedemos a él, nos abre directamente la captura de pantalla que ha realizado.

De esta forma, hemos conseguido un informe de resultados sencillo y eficaz con el que podemos analizar las pruebas detenidamente. Cuando tengamos, por ejemplo, más de cincuenta pruebas automatizadas, agradeceremos tener estos informes con los que poder hacer balance de los resultados.

4.8. Localizador de elementos: *repeater*

En el apartado 3.2 hemos mencionado la existencia del localizador *repeater*, y es ahora cuando vamos a ver detalladamente cómo funciona.

En AngularJS existe una directiva llamada ‘ng-repeat’ con la que podemos instanciar una plantilla una vez por cada elemento de una colección [10]. ¿Qué conseguimos con esto? Conseguimos que AngularJS se encargue de la generación del HTML sin que el programador tenga que preocuparse por cuantos elementos pueda contener la colección.

```
<div ng-repeat="cat in pets">
    <span>{{cat.name}}</span>
    <span>{{cat.age}}</span>
</div>
```

Figura 32. Extracto de una vista (HTML) de un proyecto en AngularJS

Por cada elemento ‘cat’ que tengamos en la colección ‘pets’, se crearán las dos etiquetas ‘span’.

En Protractor, para acceder a los elementos que tienen el atributo ‘ng-repeat’ utilizamos el localizador *repeater* [11]. Veamos un ejemplo sencillo:

```
// Returns a promise that resolves to an array of WebElements containing
// all top level elements repeated by the repeater. For 2 pets rows resolves
// to an array of 2 elements.
var rows = element.all(by.repeater('cat in pets'));
```

Figura 33. Ejemplo de uso del localizador *repeater*

Con esta línea de código obtenemos un array con todos los elementos que contenga la colección ‘pets’. Al localizador tenemos que pasarle como parámetro el nombre que tenga en la etiqueta ‘ng-repeat’. Debemos remarcar que los elementos que tenemos en este punto son *WebElements*, por lo que no podemos utilizarlos como si fueran los objetos que tendríamos en nuestro proyecto en AngularJS.

Para este trabajo particular, nos centraremos en cómo obtener el nombre (*String*) de cada elemento de la colección. Y aquí entran en juego las promesas de JavaScript y el cómo tratar con ellas. JavaScript es un lenguaje asíncrono, es decir, las líneas de código no se ejecutan en orden, algo que sí ocurre en otros lenguajes (C++ o Java). Por ello, aparece el concepto de promesa [12]. Al ejecutar una función como la de la Figura 33, lo que devuelve es una promesa, la cual devolverá el resultado final cuando se resuelva a lo largo del tiempo. Debido a esto, si deseamos utilizar el resultado, tendremos que hacer uso de una sintaxis especial.

```
element.all(by.repeater('result in results')).then(function (resultados) {
    ...
});
```

Figura 34. Sintaxis para tratar el resultado de una promesa

Lo que estamos haciendo aquí es decirle al programa que cuando haya resuelto la promesa (*then*) realice la función siguiente, pasándole como parámetro el resultado que nos ha devuelto la promesa.

Como ya hemos dicho, lo que en nuestro caso queremos es obtener el nombre de cada elemento de la colección, por lo que incluimos el siguiente código:

```
element.all(by.repeater('result in results')).then(function (resultados) {
    resultados[selectedResult].getText().then(function(txt) {
        ...
    });
});
```

Figura 35. Tratamiento del resultado de una promesa

Con la función 'getText()' obtenemos el texto que estábamos buscando, pero dicha función devuelve una promesa, por lo que tenemos que volver a utilizar la sintaxis del 'then'. Así, ya tendríamos acceso al texto y podríamos utilizarlo para realizar innumerables pruebas. Por ejemplo, podríamos comprobar si un elemento X se encuentra dentro de una lista. Y ya tendríamos una prueba automatizada más lista para ejecutar.

4.9. La automatización de pruebas en un contexto de desarrollo de software

Actualmente, tenemos completamente programadas decenas de pruebas automatizadas, un número que va aumentando progresivamente a medida que se van finalizando UTs y analizando las PAs. Al estar en las primeras fases del

proyecto, hay componentes de la aplicación que, aunque ya hubieran sido finalizados, se vuelven a modificar para mejorarlos, como en todo desarrollo ágil. Por ello, estamos ejecutando las pruebas automatizadas cada semana, obteniendo en numerosas ocasiones nuevos fallos que reportar a la parte *front-end*.

Finalmente, se ha podido demostrar que las pruebas automatizadas son realmente útiles, encontrando fallos que de otra forma habríamos tardado más tiempo en encontrar, y que habría provocado que el coste del *retrabajo* fuera mayor.

5. Gestor de suites de pruebas automatizadas

5.1. Introducción

Como ya hemos visto en el capítulo anterior, las pruebas que podemos llegar a automatizar son muchas, por lo que debemos valorar de forma muy precisa cuáles son las más convenientes para automatizarse. A pesar de esta minuciosa selección, llegará un momento en el que tendremos tantas pruebas automatizadas que no será conveniente ejecutarlas todas ya que, o bien tardarían mucho tiempo en ejecutarse, o bien tardaríamos nosotros como *testers* mucho tiempo en revisar los resultados. Es por ello que queremos tener suites de pruebas.

En Protractor, cada archivo 'conf.js' es una suite ya que en él ponemos las pruebas que queremos que se ejecuten. Supongamos que tenemos mil pruebas automatizadas y queremos realizar una suite que contenga las pruebas relacionadas con un módulo X de la aplicación. Sería bastante tedioso hacer la selección y escribir una a una la ruta del archivo de cada una de las pruebas. Ante esta situación, nos hemos propuesto realizar un gestor de suites de pruebas automatizadas utilizando el *framework* AngularJS.

Este gestor contendrá las siguientes funciones:

- Conexión con TUNE-UP para obtener las pruebas de aceptación que hay automatizadas.
- Búsqueda y filtros para encontrar las pruebas deseadas.
- Sistema de selección de pruebas y creación de la suite en un archivo de configuración de Protractor.
- Carga de una suite ya creada para su modificación.
- Ejecución automática de una suite haciendo uso de Jenkins.

De este modo, procedemos a la creación del gestor empezando por sentar las bases de AngularJS.

5.2. Qué es AngularJS y primeros pasos

"AngularJS es un conjunto de librerías de código abierto para JavaScript desarrollado por Google que nos sirve para hacer aplicaciones web avanzadas del lado del cliente. Usa el patrón de diseño habitualmente encontrado en el desarrollo web MVC (modelo-vista-controlador) que, junto con otras herramientas disponibles en Angular, nos permite un desarrollo ordenado, sencillo de realizar y, sobre todo, más fácil de mantener en un futuro." [13]

"Este *framework* nos ofrece muchas facilidades para hacer aplicaciones web, aplicaciones de gestión o de negocio, aplicaciones que funcionan en dispositivos y que tienen un rendimiento muy similar a las nativas e incluso aplicaciones de escritorio con una frontal web, cada vez más habituales." [14]

Veamos cuáles son los elementos y conceptos que vamos a encontrar dentro de AngularJS [15]:

- Vistas: Es el HTML y todo lo que representa datos o información.
- Controladores: Se encargan de la lógica de la aplicación.
- Modelo de la vista: Es toda aquella información que resulta útil para el programador pero que no forma parte del modelo de negocio. Es lo que llamamos '*Scope'* (ver Glosario), que es el modelo en AngularJS.
- Módulo: Es la manera que nos propone AngularJS para que los desarrolladores tengan un código ordenado, de forma que se puedan dividir las cosas, evitar variables globales de JavaScript, etc.

Figura 36. Descripción gráfica de AngularJS

Como observamos en la Figura 36, el '*Scope*' es un objeto JavaScript que nos sirve para comunicar desde la parte del HTML a la parte del JavaScript y viceversa. Poco a poco iremos explicando este concepto.

AngularJS nos ofrece muchas posibilidades, aunque nosotros no entraremos en funciones avanzadas, sino que realizaremos una web sencilla que cumpla con los requerimientos que nos hemos propuesto.

Al igual que hemos hecho con Protractor, también utilizaremos WebStorm como nuestro entorno para trabajar con AngularJS. Sigamos los siguientes pasos:

1) Creamos un nuevo proyecto vacío y una carpeta, a la que llamaremos 'app'. Además, creamos un archivo 'app.js' en dicha carpeta y un archivo 'index.html' en la raíz.
2) Descargamos e instalamos AngularJS escribiendo en la consola de nuestro proyecto:

bower install angular --save

En caso de que no nos funcione la instrucción, ejecutamos antes:

npm install -g bower

3) Introducimos la base del archivo 'app.js':

```
angular.module("app", [])
    .controller("appCtrl", ["$scope", function($scope){
        ...
    }])
```

Figura 37. Base del archivo 'app.js'

Esté código es la parte más básica que cualquier aplicación debe contener, ya que estamos creando un módulo (llamado "app") con un controlador, al que hemos llamado "appCtrl". El primer "$scope" que aparece es una inyección, es decir, introducimos un objeto de JavaScript del que queremos hacer uso dentro del controlador, y el segundo "$scope" lo pasamos como parámetro a la función ya que ya lo tenemos inyectado (podríamos llamarle "scope" simplemente o ponerle otro nombre debido a que los parámetros los

recibe en el mismo orden en el que los recibe, pero optamos por ponerle el mismo nombre para mayor simplicidad.

4) Introducimos la base del archivo ‘index.html’:

```
<!DOCTYPE html>
<html lang="es">

  <head>
    <meta charset="UTF-8">
    <title>Gestor de suites</title>
    <link rel="stylesheet">
  </head>

  <body ng-app="app" ng-controller="appCtrl">
    <h1>Gestor de suites de pruebas automatizadas</h1>

    <!-- Scripts -->
    <script src="bower_components/angular/angular.min.js"></script>
    <!-- Custom scripts -->
    <script src="app/app.js"></script>
  </body>

</html>
```

Figura 38. Base del archivo ‘index.html’

Como puede observarse en la etiqueta ‘body’, hemos colocado el nombre de la aplicación y del controlador que habíamos especificado en el archivo ‘app.js’. A continuación, hemos colocado un título para que pueda verse algo visualmente en este momento. Y finalmente se han colocado los dos scripts que tenemos: el de AngularJS y el de nuestra aplicación.

Con lo que tenemos hasta ahora ya podríamos ejecutar la aplicación desde WebStorm y comprobar que se lanza correctamente. Tras haberla probado, pasamos a elaborar funcionalidad para nuestra aplicación.

5.3. Selección de pruebas automatizadas

El objetivo de esta aplicación es poder gestionar las pruebas que tenemos ya automatizadas, por lo que vamos a empezar implementando su obtención. Para ello, haremos una petición a la API REST (ver Glosario) de TUNE-UP para obtener todas las pruebas de aceptación que hay actualmente en el proyecto, utilizando el servicio *http*. Es tan sencillo como:

```
.controller("appCtrl", ["$scope", "$http", function($scope, $http){
    $scope.pas = [];
    $http({
        method: 'GET',
        url: 'url del servicio'
    }).then(function(res){
        $scope.pas = res.data;
    }, function(res){
        $scope.pas = [{name: "Error! " + res.status}];
    });
    ...
}])
```

Figura 39. Petición http para obtener las pruebas automatizadas

Inyectamos el servicio *$http* para poder hacer la petición y creamos la variable *pas* en el *Scope*. Al haberla introducido en el *Scope*, podremos acceder a ella tanto desde la parte de JavaScript como desde la parte del HTML. Realizamos la petición 'GET': si resulta un éxito, obtendremos el resultado en la variable *pas*; si falla, en dicha variable se guardará el error que ha devuelto el servicio.

En nuestro caso, la API REST de TUNE-UP devuelve el resultado en formato JSON, por lo que tenemos el objeto tal cual en la variable. Esto nos da muchas posibilidades ya que podemos filtrar y mostrar por pantalla lo que realmente nos interesa. Nosotros, para hacer este gestor, queremos que el usuario pueda ver el código de la PA, el nombre, a qué UT pertenece y su código, y el nombre del nodo. Además, el servicio nos ha devuelto todas las pruebas de

aceptación y nosotros solo queremos aquellas que están automatizadas, por lo que también tenemos que hacer un filtro.

```
$scope.isAutomaticFilter = function(pa){
   return pa.EstadoPA == 'Automatizada';
}
```

Figura 40. Ejemplo de filtro en AngularJS

Volvemos a hacer uso así del *Scope*, uno de los objetos más útiles de JavaScript. Se le pasa como parámetro una única prueba de aceptación y comprueba si su estado actual en TUNE-UP es 'Automatizada'. Como ya vimos en el capítulo anterior, cuando terminamos de programar una prueba, cambiamos su estado en TUNE-UP y, con ello, conseguimos poder filtrarlas al recibirlas en la parte *front-end*.

En la parte HTML, hemos creado una tabla y cada fila se rellena de la siguiente forma:

```
<tr ng-repeat="pa in pas | filter:isAutomaticFilter">
   <td> {{pa.CodigoPA}} </td>
   <td> {{pa.NombrePA}} </td>
   <td> {{pa.IdUT}} - {{pa.NombreUT}} </td>
   <td> {{pa.NombreNodo}} </td>
</tr>
```

Figura 41. Fila de la tabla de pruebas automatizadas

Hacemos uso de la directiva *ng-repeat* que, como vimos en el apartado 2.8, instancia una plantilla una vez por cada elemento de una colección. Así, por cada *pa* tras haber aplicado el filtro, se creará una fila de la tabla.

Además del filtro para buscar las pruebas que estén automatizadas, también se han programado dos filtros más. Por una parte, un campo de búsqueda que, conforme el usuario escribe, busca dinámicamente en todos los atributos de las pruebas, mostrando únicamente aquellas que contengan la

búsqueda realizada. Por otra parte, un campo de selección de UT que, al escribir en él la UT deseada, busca todas las PAs que estén contenidas en dicha UT.

El siguiente paso ha sido incorporar la funcionalidad de selección, con la que podemos marcar las pruebas que deseamos añadir a la Suite. Cada prueba tiene una casilla que podemos marcar y desmarcar, y se han desarrollado dos botones: 'Marcar todas' y 'Desmarcar todas'. La particularidad de estos botones es que no actúan sobre el conjunto total de pruebas automatizadas, sino sobre el contexto en el que nos encontramos en cada momento. Por ejemplo, supongamos que en el campo de UT hemos puesto el número 2 y pulsamos sobre 'Marcar todas'; el resultado obtenido es que solo se marcan las PAs de la UT 2. Si ahora cambiamos a la UT 4 y pulsamos sobre 'Desmarcar todas', las PAs de la UT 2 no se desmarcarán ya que no estamos en el mismo contexto. Con esta funcionalidad conseguimos grandes beneficios que no podrían conseguirse con un único *check box* de doble funcionalidad (Marcar/Desmarcar).

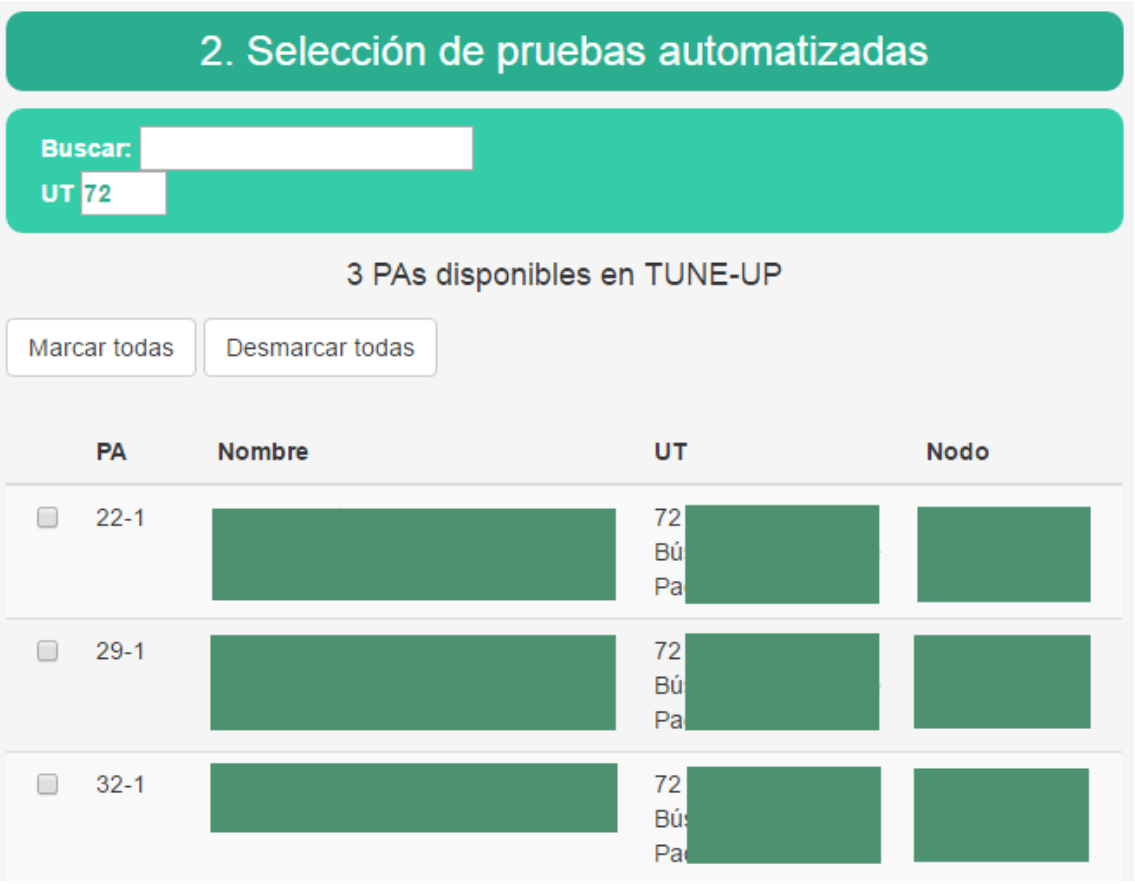


Figura 42. Selección de pruebas automatizadas en el Gestor de suites

Cabe añadir que también se ha agregado un contador debajo de los campos de búsqueda, de forma que muestre dinámicamente el número de pruebas que hay disponibles tras aplicar los filtros. Además, se ha agregado un archivo CSS al proyecto para poder modificar la visualización y se ha hecho uso de Bootstrap.

Bootstrap es un *framework* en HTML, CSS y JavaScript para desarrollo de proyectos web con un diseño adaptable (*responsive*) [16]. Es una herramienta para crear interfaces de usuario limpias y totalmente adaptables a todo tipo de dispositivos y pantallas, sea cual sea su tamaño. Además, Bootstrap ofrece las herramientas necesarias para crear cualquier tipo de sitio web utilizando estilos y elementos de sus librerías, consiguiendo diseños simples, limpios e intuitivos, compatibles con la mayoría de navegadores web del mercado.

Instalamos Bootstrap utilizando Bower:

bower install bootstrap

Y modificamos el archivo 'index.html':

```
<!DOCTYPE html>
<html lang="es">
    ...
  <body ng-app="app" ng-controller="appCtrl">
    ...
   <!-- Scripts -->
   <script src="bower_components/jquery/dist/jquery.js"></script>
   <script src="bower_components/bootstrap/dist/js/bootstrap.js"></script>
   <script src="bower_components/angular/angular.min.js"></script>
   <!-- Custom scripts -->
   <script src="app/app.js"></script>
  </div>
 </body>
</html>
```

Figura 43. Scripts para Bootstrap en 'index.html'

Si empezamos a utilizar componentes de HTML podremos observar como estos han cambiado. Además, tenemos acceso a las múltiples clases disponibles para modificar visualmente nuestra aplicación. Bootstrap cuenta con una amplia documentación, que podemos encontrar directamente en su página web [17].

5.4. Generación de la suite de pruebas

Una vez tenemos todo listo para seleccionar las pruebas, procedemos a la generación de la suite que contendrá dichas pruebas.

En la parte izquierda de la aplicación tenemos el apartado de selección, y en la parte derecha aparecerán las pruebas que hayan sido seleccionadas. Además, también podemos desmarcarlas desde el apartado de generación. Una vez hemos terminado de seleccionarlas, tan solo tendremos que pulsar sobre un botón y nos generará el archivo, que podremos guardarlo dónde elijamos. Veamos cómo llevarlo a cabo.

Mostrar una tabla con las pruebas de aceptación marcadas es muy sencillo. Lo haremos de la misma forma que tenemos en el apartado de selección, pero aplicando un filtro que nos devuelva únicamente las marcadas.

Para elaborar el botón con la funcionalidad mencionada haremos uso del objeto *Blob*, que nos permite hacer uso de la interfaz 'Guardar como'. Añadamos el siguiente código al archivo 'app.js':

```
angular.module("app", [])
.config(['$compileProvider', function ($compileProvider) {
    $compileProvider.aHrefSanitizationWhitelist(/^\s*(https?|ftp|mailto|chrome-
extension|blob|):/);
}])
.controller("appCtrl", ["$scope", "$http", function($scope, $http){
    ...
  $scope.exportConf = function(){
    var texto = ... ;
    var blob = new Blob([texto], {type: 'text/plain'});
    var url = window.URL.createObjectURL(blob);
    $scope.fileUrl = url;
  }
    ...
}])
```

Figura 44. Implementación de la generación de suites

En la parte de configuración conseguimos que las URL funcionen correctamente, ya que lo que estamos haciendo es guardar el contenido de la variable 'texto' (variable que contiene la configuración de la suite de pruebas) en una URL, para que el usuario pueda posteriormente acceder y descargarlo. No obstante, todo esto es transparente para el usuario ya que únicamente se pulsará un botón. La parte HTML para el botón quedaría así:

```
<a ng-href="{{ fileUrl }}" download="conf.js">
   <button class="btn btn-default center-block" id="btnExport"
           ng-click="exportConf()">Generar Suite</button>
</a>
```

Figura 45. HTML para generar las suites de pruebas

Tenemos un botón al que le hemos aplicado varias clases de Bootstrap y, al pulsar sobre él, se activa la función 'exportConf()', la cual acabamos de crear en el paso anterior. Esta función guarda finalmente la URL para descargar el archivo en el *Scope*. Así, en el HTML hemos colocado un hipervínculo a esta URL mediante "{{ fileUrl }}", con lo que conseguimos que todo funcione como deseábamos.

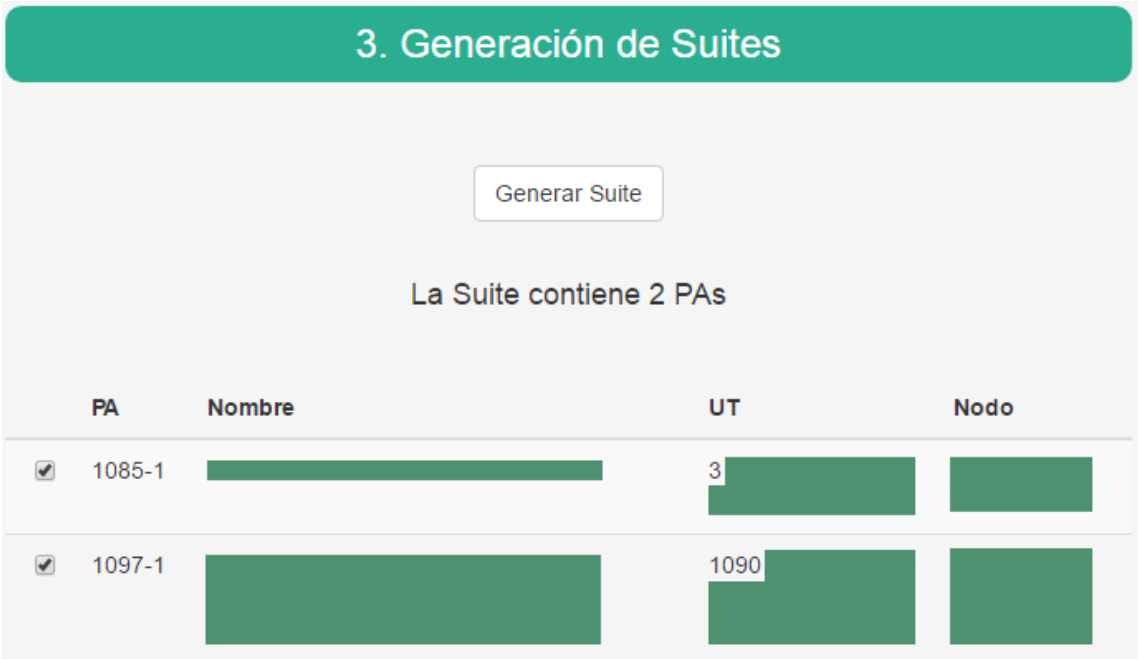


Figura 46. Generación de suites en el gestor de suites.

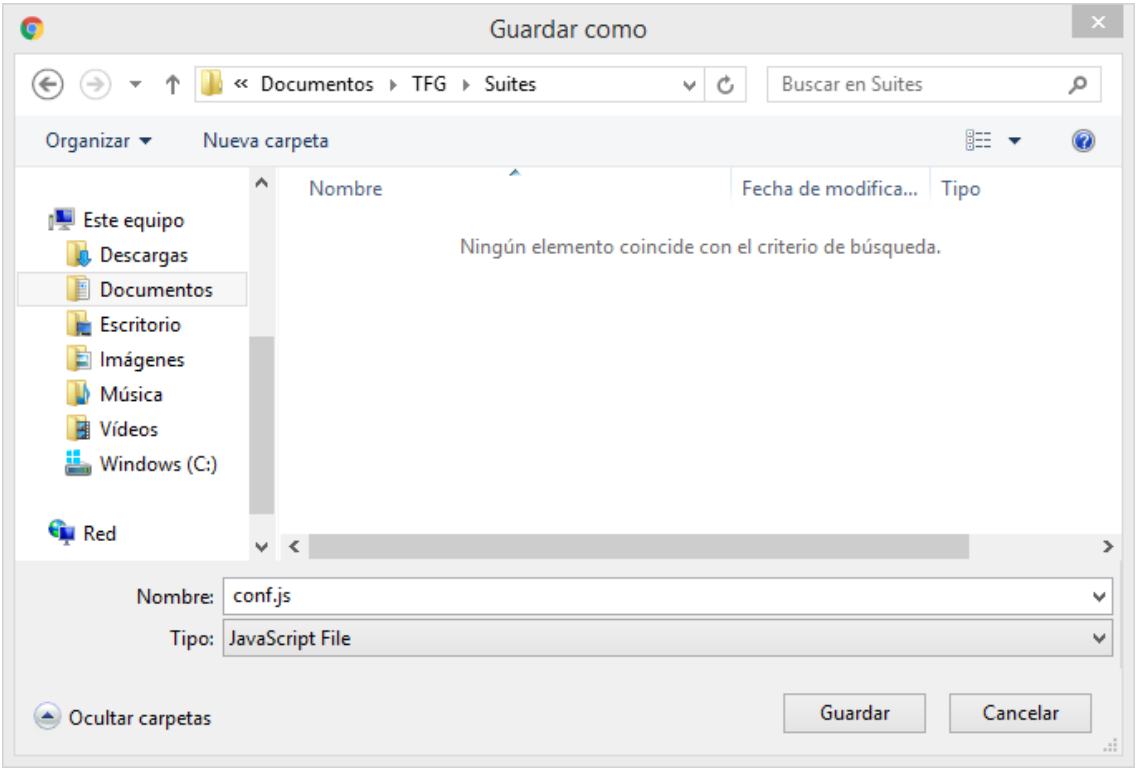


Figura 47. Resultado tras pulsar sobre el botón 'Generar Suite'

La pregunta que surge en estos momentos es: ¿cómo se ha conseguido obtener la configuración de la suite en una variable? El archivo de configuración de Protractor es un archivo que puede ir evolucionando a lo largo del proyecto, por lo que no podemos tenerlo guardado en la aplicación web ya que se quedaría desactualizado. Para ello, el primer paso que debe hacer el usuario al acceder a nuestra aplicación web es cargar una suite de pruebas, como observamos en la Figura 48. No importa que contenga pruebas o no; lo importante es que sea una suite con las últimas actualizaciones que hayamos realizado en el proyecto de Protractor. Así, cuando la aplicación web cargue el archivo, buscará el lugar donde se deben introducir las rutas de las pruebas automatizadas y guardará en memoria el resto del archivo. Tras esto, seleccionaremos las pruebas y pulsaremos sobre 'Generar Suite', con lo que la aplicación introducirá dichas pruebas en el archivo que se ha guardado en memoria, consiguiendo que el usuario obtenga la suite de pruebas con la última versión disponible.

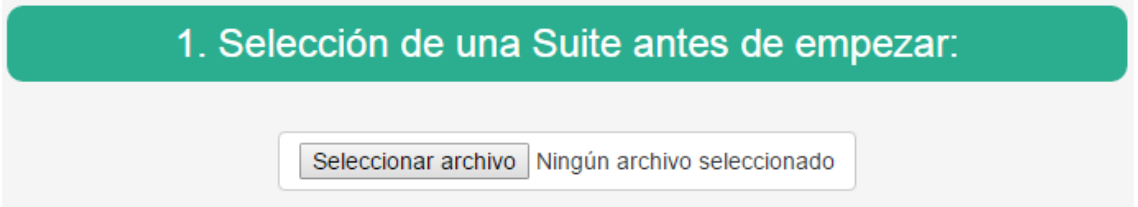


Figura 48. Selección de una suite en el gestor de suites

De esta forma, con tan solo cargar una suite actualizada, la aplicación web funcionará correctamente. Además, hacerlo de esta forma nos reporta más beneficios. Ahora, como podemos cargar suites de pruebas, también podremos editarlas. Es decir, imaginemos que tenemos una suite con mil pruebas y queremos la misma suite pero con una pruebas más. No nos gustaría tener que volver a marcar las mil pruebas para simplemente añadir una más. Por ello, al cargar una suite, las pruebas que contenga aparecerán marcadas directamente, por lo que solo tendremos que marcar las pruebas extra y guardar la nueva suite. Del mismo modo, también podremos eliminar pruebas. Así, obtenemos un mecanismo para modificar las suites libremente.

En este punto, ya tenemos nuestra aplicación web terminada y funcionando. Tan solo nos quedaría ejecutar la suite que hemos obtenido, y podríamos hacerlo tanto desde la consola de Windows como desde Protractor. Pero, ya que estamos en el ámbito de la automatización, ¿por qué no automatizar también el proceso de ejecución y poder hacerlo todo desde nuestra aplicación?

5.5. Ejecución automática con Jenkins

Jenkins es un sistema que proporciona integración continua para el desarrollo de software [18]. Soporta herramientas de control de versiones como Git, además de poder extenderse mediante *plugins*, con los que cambiar el comportamiento de Jenkins o añadir nueva funcionalidad. Es utilizado para levantar (*build*) y testar proyectos de software continuamente, haciendo que integrar cambios en un proyecto sea mucho más sencillo para los desarrolladores.

Como se ha comentado, nos encontramos en un proyecto de software hospitalario desarrollado en AngularJS. Y con Jenkins es posible que, por ejemplo, dicho proyecto se despliegue en el servidor cada vez que se suben cambios al repositorio Git. Esta es solo una de las innumerables acciones que se pueden conseguir con Jenkins.

Para nuestro caso particular, nos interesa que pueda descargarse el código fuente de nuestras pruebas automatizadas de un repositorio Git y que después ejecute Protractor. Para ello, hemos creado un repositorio en Bitbucket donde se va subiendo todo el código conforme se van programando las pruebas. Si queremos ejecutar Protractor, primero tendremos que instalarlo en el servidor en el que se encuentre Jenkins de la misma forma que lo instalamos en la sección 2.1.

Una vez configurado Jenkins, podríamos lanzar la tarea (así se llama cada "proyecto" en Jenkins). Mientras se está ejecutando podemos ver la consola y, finalmente, un icono nos informa si la ejecución ha sido correcta. Como tenemos el generador de informes preparado, habremos obtenido uno, con el que podremos ver qué pruebas han fallado y analizarlo.

Ahora solo queda poder lanzarlo desde nuestra aplicación web. Además, queremos poder pasarle como parámetro la suite que queramos ejecutar en cada momento. Para ello, crearemos un botón llamado "Ejecutar Jenkins" y una función dentro del controlador.

```
$scope.execJenkins = function(){
   var suite = $scope.confFileName;
   $http({
      method: 'POST',
      url: 'http://<JenkinsURL>/job/<jobName>/buildWithParameters?
           suiteName='+suite,
      withCredentials: true,
      beforeSend: function (xhr){
         xhr.setRequestHeader('Authorization', "Basic " + btoa("<username>" +
           ":" + "<apiToken>"));
      }
   }).then(function(res){ console.log('OK: '+res);
   }, function(res){ console.log('Error: '+res);
   });
}
```

Figura 49. Función de ejecución de Jenkins

En primer lugar, debemos haber cargado una suite de pruebas. Con ello, nuestra aplicación guarda en el *Scope* el nombre de la suite, en la variable 'confFileName'.

En segundo lugar, haremos una petición http a nuestra tarea de Jenkins, pasándole como parámetro el nombre de la suite [19]. Para ello necesitamos usar *withCredentials* [20] y enviar una cabecera de autorización [21], que cuenta con el nombre de usuario y la clave del API (*Token*), y que podemos encontrar en nuestro panel de configuración de Jenkins [22], tal cual se muestra en la Figura 50.

Figura 50. Nombre de usuario y clave del API en Jenkins

En tercer lugar, pasamos a configurar Jenkins para que pueda ejecutar tareas con parámetros [23]. Para ello debemos definir un nuevo parámetro como se indica en la Figura 51, al cual podemos hacer referencia de la siguiente forma: '%nombreDelParametro%'. En nuestro caso, lo referenciamos con '%suiteName%' ya que nuestro parámetro es 'suiteName'.

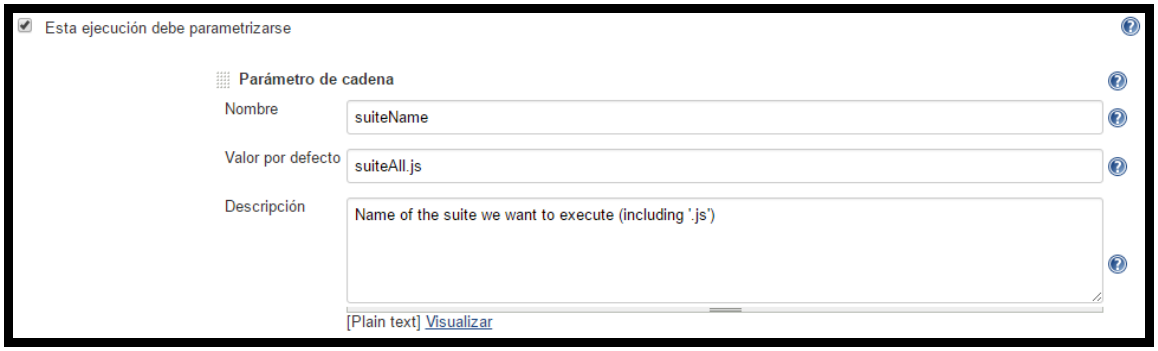


Figura 51. Parametrización en Jenkins

En cuarto y último lugar, hemos añadido al archivo HTML un nuevo botón. Al pulsar sobre él, el nombre de la última suite cargada en la aplicación se pasará como parámetro a Jenkins, y él buscará dicho archivo en la ruta que le hayamos indicado.

Tras esto, nuestra aplicación web está finalizada completamente y cumple con la funcionalidad que pretendíamos que tuviera. En la Figura X49 podemos ver cómo ha quedado finalmente y en la Figura X50 una de los primeras versiones que teníamos de la aplicación (al principio se utilizaba una API REST

pública de países para poder implementar la funcionalidad antes de tener acceso a la API REST de TUNE-UP).

Figura 52. Versión final del gestor de suites

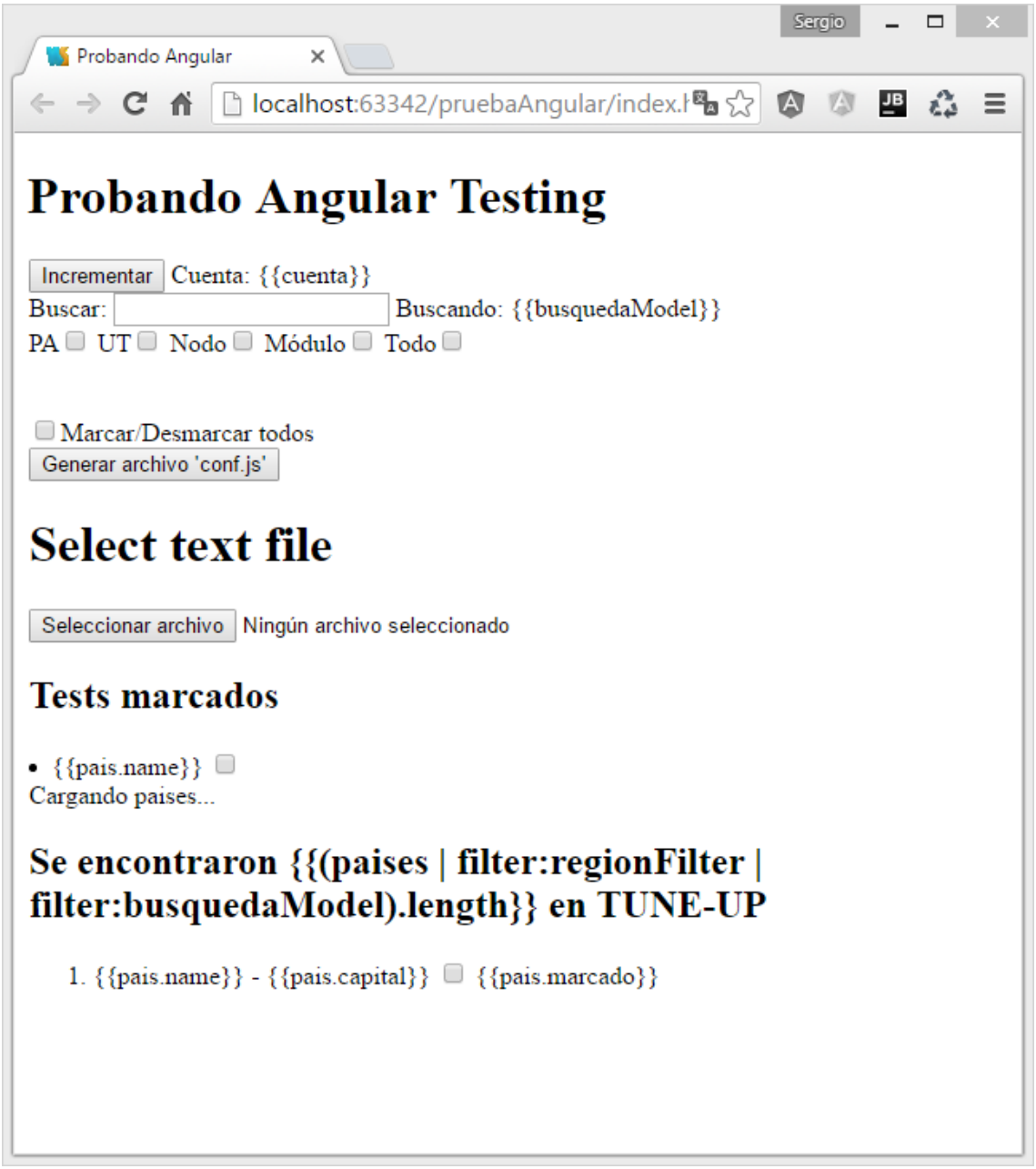


Figura 53. Gestor de suites en una de sus primeras fases de desarrollo

6. Conclusiones

Hemos conseguido abordar con éxito los tres objetivos que habíamos marcado:

En primer lugar, hemos conocido un proceso de automatización de pruebas, desde la primera vez que se aplica la prueba de aceptación manualmente hasta que ejecutamos la prueba ya automatizada. Para ello, hemos hecho uso de TUNE-UP y de un desarrollo ágil centrado en las pruebas de aceptación.

En segundo lugar, hemos aprendido a utilizar Protractor en detalle, conociendo múltiples maneras de manipular elementos de la aplicación web y elaborando numerosas pruebas automatizadas. Además, se ha extendido su utilidad con la creación de informes de resultados, con los que se reduce aún más el trabajo posterior. Para conseguir las pruebas automatizadas hemos utilizado JavaScript y, concretamente, Protractor con patrones de diseño.

En tercer lugar, hemos desarrollado una aplicación web con la que gestionar todas nuestras pruebas automatizadas, la cual se conecta a la REST API de TUNE-UP para obtener las pruebas que tenemos automatizadas. Tras ello, podemos elegir las pruebas que queramos y guardarlas en un archivo de configuración que podemos ejecutar con WebStorm o con la consola de comandos. Hemos mejorado la funcionalidad incorporando Jenkins, con el que logramos ejecutar el archivo de configuración directamente desde nuestra aplicación. Por tanto, hemos conseguido obtener toda la funcionalidad necesaria para la automatización centralizada en un mismo sitio, haciendo uso de tecnologías como AngularJS, HTML5, CSS3 o Bootstrap.

En la Figura 54 puede observarse un diagrama con las tecnologías empleadas en este trabajo.

Figura 54. Diagrama con las tecnologías empleadas

Finalmente, cabe decir que he obtenido gran satisfacción personal con este proyecto. He aprendido todo el marco de trabajo de un desarrollo ágil y, más en detalle, todo el proceso de creación de pruebas automatizadas. He logrado los objetivos que me propuse desde un principio, pudiendo ampliarlos incluso con más funcionalidad, consiguiendo así el marco de trabajo para el desarrollo y ejecución de pruebas automatizadas aplicadas al front-end de una aplicación web, que buscábamos con este trabajo de fin de grado.

Anexo.
Automatización de una prueba

Veamos el proceso que sigue una prueba de aceptación. A modo de ejemplo, vamos a utilizar una prueba que comprueba si un formulario funciona correctamente, comprobando que los campos obligatorios son admitidos al rellenarse. Concretamente, será un formulario de la residencia de un paciente en una base de datos hospitalaria.

En primer lugar, la prueba ha de ser identificada y definida. En la Figura 55 mostramos una de las pruebas que hemos automatizado.

Prueba de aceptación 1085-1 | Guardado datos de localización OK

Condición: Paciente existente

Pasos: Introducir/editar los datos correctamente e intentar guardar

Resultado Esperado: El sistema muestra la pantalla en modo consulta con los datos actualizados. Comprobar que al volver a cargar la pantalla de datos del paciente, el sistema muestra los datos actualizados.

Figura 55. Definición de la prueba de aceptación 1085

En segundo lugar, se realiza el diseño. Para esta prueba, hay dos caminos posibles, por lo que se han de realizar dos diseños, los cuales se observan en las Figuras 56-1 y 56-2.

Diseño 1. Editar un paciente que ya está registrado (Hotel).

Condición:

- Existe el paciente "49 – Anacleto Aicardi de la Vega".

Pasos:

- Modificar los siguientes datos del paciente "49 – Anacleto Aicardi de la Vega" y guardar:
    - Domicilio Habitual
        - Tipo Dirección: Ronda
        - Dirección: Norte
        - Número: 2
        - Piso: 3
        - Puerta : 7
        - País: España
        - Comunidad : Comunitat Valenciana
        - Provincia: Valencia
        - Población: Valencia
        - Código Postal: 46090
    - Residencia Vacacional
        - Tipo Dirección: Hotel
        - Hotel: Acapulco
        - Habitación: 186

Resultado Esperado:

El sistema muestra la pantalla en modo consulta con los datos actualizados. Comprobar que al volver a cargar la pantalla de datos del paciente, el sistema muestra los datos actualizados.

Figura 56-1. Diseño 1 de la prueba de aceptación 1085

Diseño 2. Editar un paciente que ya está registrado (Residencia).

Condición:

- Existe el paciente "49 – Anacleto Aicardi de la Vega".

Pasos:

- Modificar los siguientes datos del paciente "49 – Anacleto Aicardi de la Vega" y guardar:
    - Domicilio Habitual
        - Tipo Dirección: Ronda
        - Dirección: Norte
        - Número: 2
        - Piso: 3
        - Puerta : 7
        - País: España
        - Comunidad : Comunitat Valenciana
        - Provincia: Valencia
        - Población: Valencia
        - Código Postal: 46090
    - Residencia Vacacional
        - Tipo Dirección: Residencia
        - Tipo dirección: Paseo Marítimo
        - Dirección: 9 d'octubre
        - Número: 5
        - Bloque: A
        - Piso: 2
        - Puerta: 10
        - País: España
        - Comunidad: Comunitat Valenciana
        - Provincia: Valencia
        - Población: Canet d'en Berenger
        - Código Postal: 46529

Resultado Esperado:

El sistema muestra la pantalla en modo consulta con los datos actualizados. Comprobar que al volver a cargar la pantalla de datos del paciente, el sistema muestra los datos actualizados.

Figura 56-2. Diseño 2 de la prueba de aceptación 1085

En tercer lugar, se realiza la automatización. Para ello, debemos hacer uso de los scripts mostrados en las Figuras 57 a 61.

```
var init = function() {

    this.load = function(){
        browser.get('<url>');
    }

    this.loadMPI = function(){
        this.load();
    }

};
module.exports = new init();
```

Figura 57. Archivo init.js

```
var main = function() {

    this.changeData = function(data, id){
        if(data) {
            element(by.id(id)).sendKeys(protractor.Key.chord(protractor.Key.CONTROL, "a"),
data);
        }
    }

    this.changeDataEnter = function(data, id){
        if(data) {
            this.changeData(data, id);
            browser.sleep(600);
            browser.actions().sendKeys(protractor.Key.ENTER).perform();
        }
    }

    this.changeLabel = function(data, id) {
        if (data) {
            var defaultLabel = element(by.id(id));
            browser.executeScript("arguments[0].click();", defaultLabel.getWebElement());
        }
    }

    this.click = function(id){
        element(by.id(id)).click();
    }
```

Figura 58-1. Archivo main.js

```
    /**
     * @param id Id del campo del dropdown
     * @param column (Opcional) Columna que deseamos obtener de los elementos del
dropdown
     * @returns {*} Array con los elementos del dropdown
     */
    this.getElementsDropdown = function(id, column){
        return element(by.id(id)).evaluate('items').then(function(elements){
            if(column) {
                elements.forEach(function (element, index) {
                    elements[index] = element[column];
                });
            }
            return elements;
        });
    }
    /**
     * @param id Id del campo del dropdown
     * @param element Elemento que deseamos buscar
     * @param column Atributo de los elementos del dropdown con el que queremos
comparar nuestro elemento
     * @returns {*}
     */
    this.findInDropdown = function(id, element, column){
        return this.getElementsDropdown(id, column).then(function(elements){
            var found = false;
            elements.forEach(function(el){
                if(el === element){
                    found = true;
                }
            });
            return found;
        })
    }
    /**
     * @param parentId Id del campo en el que se encuentra el elemento padre
     * @param parent Elemento padre
     * @param childId Id del campo en el que se encuentra el elemento hijo
     * @param child Elemento hijo
     * @param column Columna de la que obtenemos el nombre de los elementos del
dropdown
     */
    this.verifySync = function(parentId, parent, childId, child, column){
        this.changeDataEnter(parent, parentId);
        return this.findInDropdown(childId, child, column);
    }

};
module.exports = new main();
```

Figura 58-2. Archivo main.js

```
var location = function() {
   /**
    * @param locationData = {streetType, street, streetNumber, stair, block, floor, door,
    *                word, country, ccaa, province, city, postalCode,
    *                location2ResidenceType {location2Hotel, location2HotelRoom,
    *                location2StreetType, location2Street, location2StreetNumber, location2Stair,
    *                location2Block, location2Floor, location2Door, location2Word,
    *                location2Country, location2Ccaa, location2Province, location2City,
    *                location2PostalCode}
    *                }
    */
   this.update = function(locationData) {
      this.changeDataEnter(locationData.streetType, 'streetType');
      this.changeData(locationData.street, 'street');
      this.changeData(locationData.streetNumber, 'streetNumber');
      this.changeData(locationData.stair, 'stair');
      this.changeData(locationData.block, 'block');
      this.changeData(locationData.floor, 'floor');
      this.changeData(locationData.door, 'door');
      this.changeData(locationData.word, 'word');

      this.changeDataEnter(locationData.country, 'country');
      this.changeDataEnter(locationData.ccaa, 'ccaa');
      this.changeDataEnter(locationData.province, 'province');
      this.changeDataEnter(locationData.city, 'city');
      this.changeData(locationData.postalCode, 'postalCode');

      this.changeDataEnter(locationData.location2ResidenceType,
'location2ResidenceType');

      this.changeDataEnter(locationData.location2Hotel, 'location2Hotel');
      this.changeData(locationData.location2HotelRoom, 'location2HotelRoom');

      this.changeDataEnter(locationData.location2StreetType, 'location2NationalityId');
      this.changeData(locationData.location2Street, 'location2Street');
      this.changeData(locationData.location2StreetNumber, 'location2StreetNumber');
      this.changeData(locationData.location2Stair, 'location2Stair');
      this.changeData(locationData.location2Block, 'location2Block');
      this.changeData(locationData.location2Floor, 'location2Floor');
      this.changeData(locationData.location2Door, 'location2Door');
      this.changeData(locationData.location2Word, 'location2Word');

      this.changeDataEnter(locationData.location2Country, 'location2Country');
      this.changeDataEnter(locationData.location2Ccaa, 'location2Ccaa');
      this.changeDataEnter(locationData.location2Province, 'location2Province');
      this.changeDataEnter(locationData.location2City, 'location2City');
      this.changeData(locationData.location2PostalCode, 'location2PostaCode');

   }
```

Figura 59-1. Archivo location.js

```
    this.verifySyncCountryRegion = function(country, region) {
        var countryId = 'country';
        var regionId = 'ccaa';
        var column = 'es';
        return this.verifySync(countryId, country, regionId, region, column);
    }
    this.verifySyncRegionProvince = function(region, province) {
        var regionId = 'ccaa';
        var provinceId = 'province';
        var column = 'es';
        return this.verifySync(regionId, region, provinceId, province, column);
    }
    this.verifySyncProvinceCity = function(province, city) {
        var provinceId = 'province';
        var cityId = 'city';
        var column = 'es';
        return this.verifySync(provinceId, province, cityId, city, column);
    }
};

var his_mpi = require(basePath+routes.his_mpi);
location.prototype = his_mpi;

module.exports = new location();
```

Figura 59-2. Archivo location.js

```
var search = function() {
    this.onlySearch = function(value){
        browser.sleep(500);
        element(by.id('patientPredictiveSearch_value')).sendKeys(protractor.Key.chord
            (protractor.Key.CONTROL, "a"), value);
    }

    this.loadFirst = function(value){
        this.onlySearch(value);
        element.all(by.repeater('result in results')).first().click();
        return 0; //Devolvemos 0 ya que hacemos click en el primer resultado.
    }

};

var main = require(basePath+routes.main);
search.prototype = main;

module.exports = new search();
```

Figura 60. Archivo search.js

```
var toasts = function() {

    //Hace click en un toast tantas veces como indique la variable 'times'.
    this.click = function(times){
        var toasts = element(by.id('toast-container'));
        for(var i = 0; i<times; i++){
            toasts.click();
        }
    }
    //Devuelve 'true' si encuentra el toast.
    this.check = function(value){
        var toasts = element(by.id('toast-container'));
        return toasts.getText().then(function(txt){
            toasts = txt.split("\n");
            var encontrado = false;
            toasts.forEach(function(toast){
                if(toast === value) {
                    encontrado = true;
                }
            });
            return encontrado;
        });
    }
    this.checkNotNulls = function(){
        var requiredToast = 'Faltan campos por rellenar';
        return this.check(requiredToast);
    }
    this.checkPatientRegisterOK = function(){
        var requiredToast = 'Paciente creado correctamente';
        return this.check(requiredToast);
    }
    this.checkSaveOK = function(){
        var requiredToast = 'Registro guardado correctamente';
        return this.check(requiredToast);
    }
    this.checkUpdateOK = function(){
        var requiredToast = 'Registro actualizado correctamente';
        return this.check(requiredToast);
    }
};
module.exports = new toasts();
```

Figura 61. Archivo toasts.js

Teniendo estos scripts como base, podemos realizar la automatización de la prueba de aceptación, como aparece en las Figura 62-1 y 62-2.

```
describe('PA1085-1 - UT3 - Pestaña Localización - Spec', function() {
    beforeAll(function() {
        var init = require(basePath+routes.init);
        init.loadMPI();
    });
    it('Guardado datos de localización OK (Hotel)', function() {
        var location = require(basePath+routes.location);
        var search = require(basePath+routes.search);

        var patient = new Object();
        patient.name = 'Anacleto';

        var locationData = new Object();
        locationData.streetType = 'Ronda';
        locationData.street = 'Norte';
        locationData.streetNumber = '2';
        locationData.floor = '3';
        locationData.door = '7';
        locationData.country = 'España';
        locationData.ccaa = 'Comunitat Valenciana';
        locationData.province = 'Valencia';
        locationData.city = 'Valencia';
        locationData.postalCode = '46090';

        locationData.location2ResidenceType = 'Hotel';
        locationData.location2Hotel = 'Acapulco';
        locationData.location2HotelRoom = '186';

        search.loadFirst(patient.name);

        location.openTab('Localización').then(function(){
            location.editionMode();
            location.update(locationData);
            location.saveProfile();

            //Comprobamos si aparece el toast.
            var toasts = require(basePath+routes.toasts);
            expect(toasts.checkSaveOK()).toEqual(true);
            toasts.click(1);
        });
    });
```

Figura 62-1. Archivo js de la prueba de aceptación 1085

```
    it('Guardado datos de localización OK (Residencia)', function() {
        var location = require(basePath+routes.location);

        var patient = new Object();
        patient.name = 'Anacleto';

        var locationData2 = new Object();
        locationData2.streetType = 'Ronda';
        locationData2.street = 'Norte';
        locationData2.streetNumber = '2';
        locationData2.floor = '3';
        locationData2.door = '7';
        locationData2.country = 'España';
        locationData2.ccaa = 'Comunitat Valenciana';
        locationData2.province = 'Valencia';
        locationData2.city = 'Valencia';
        locationData2.postalCode = '46090';

        locationData2.location2ResidenceType = 'Residencia';
        locationData2.location2StreetType = 'Paseo Marítimo';
        locationData2.location2Street = '9 d'octubre';
        locationData2.location2StreetNumber = '5';
        locationData2.location2Block = 'A';
        locationData2.location2Floor = '2';
        locationData2.location2Door = '10';
        locationData2.location2Country = 'España';
        locationData2.location2Ccaa = 'Comunitat Valenciana';
        locationData2.location2Province = 'Valencia';
        locationData2.location2City = 'Canet d'en Berenguer';
        locationData2.location2PostalCode = '46529';

        location.editionMode();
        location.update(locationData2);
        location.saveProfile();

        //Comprobamos si aparece el toast.
        var toasts = require(basePath+routes.toasts);
        expect(toasts.checkSaveOK()).toEqual(true);
    });
});
```

Figura 62-2. Archivo js de la prueba de aceptación 1085

Finalmente, tan solo tendríamos que ejecutar la prueba y obtener los resultados. En la Figura 63 se muestra la interfaz en la cual se ejecuta la prueba automatizada.

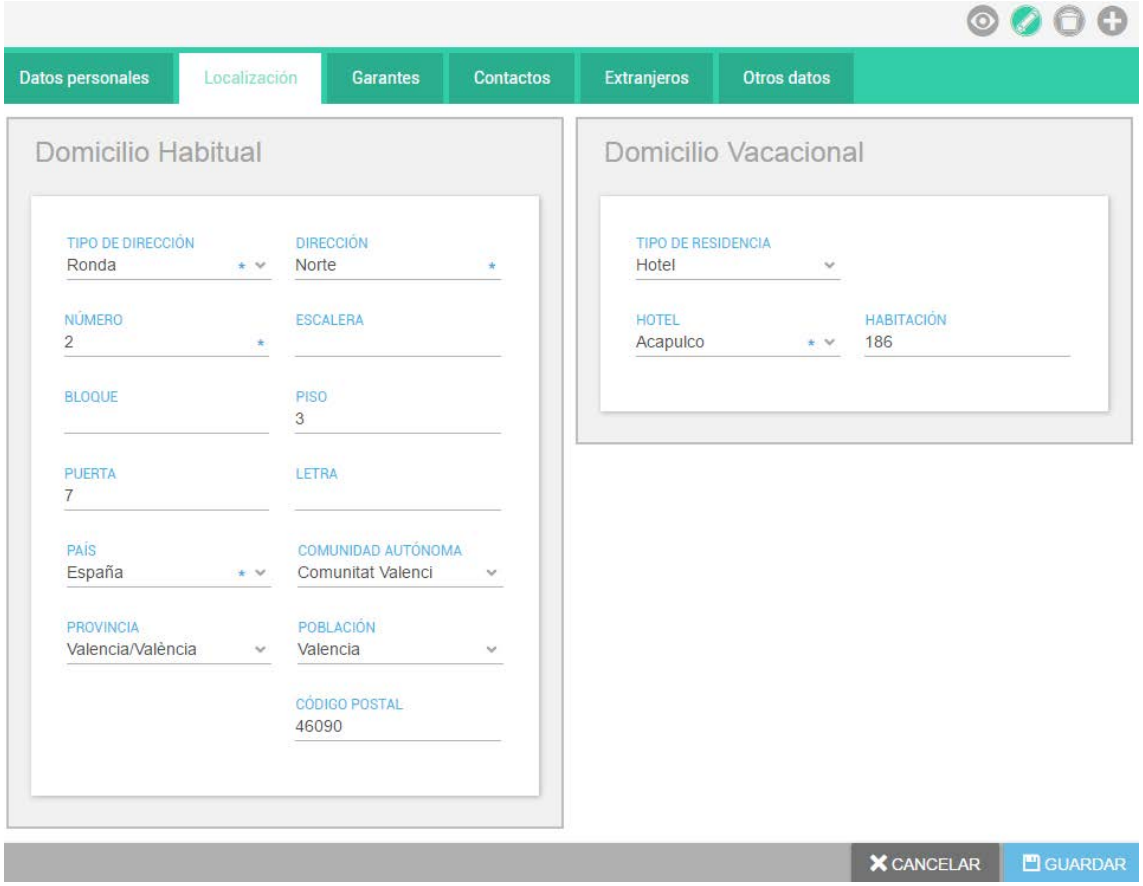


Figura 63. Interfaz de la pestaña de Localización del proyecto

Bibliografía

[1] Patricio Letelier Torres y Emilio A. Sánchez López (2003). Metodologías Ágiles en el Desarrollo de Software. <http://issi.dsic.upv.es/archives/f-1069167248521/actas.pdf> [Última consulta: 31 de mayo de 2016].

[2] AngularJS API. E2E Testing. <https://docs.angularjs.org/guide/e2e-testing> [Última consulta: 31 de mayo de 2016].

[3] Protractor. End to end testing for AngularJS. <http://angular.github.io/protractor/> [Última consulta: 31 de mayo de 2016].

[4] Protractor. Tutorial. <https://angular.github.io/protractor/#/tutorial> [Última consulta: 31 de mayo de 2016].

[5] GitHub Protractor (2015). Debugging Protractor Tests. <https://github.com/angular/protractor/blob/master/docs/debugging.md> [Última consulta: 31 de mayo de 2016].

[6] GitHub Protractor (2016). Using Page Objects to Organize Tests. <https://github.com/angular/protractor/blob/master/docs/page-objects.md> [Última consulta: 31 de mayo de 2016].

[7] Mozilla Developer Network (2016). Introducción a JavaScript orientado a objetos. <https://developer.mozilla.org/es/docs/Web/JavaScript/Introducci%C3%B3n_a_JavaScript_orientado_a_objetos> [Última consulta: 31 de mayo de 2016].

[8] Mozilla Developer Network (2015). Herencia y la cadena de prototipos. <https://developer.mozilla.org/es/docs/Web/JavaScript/Herencia_y_la_cadena_de_protipos> [Última consulta: 31 de mayo de 2016].

[9] Npm (2016). Protractor screenshot reporter for Jasmine2. <https://www.npmjs.com/package/protractor-jasmine2-screenshot-reporter> [Última consulta: 31 de mayo de 2016].

[10] AngularJS API. ngRepeat. <https://docs.angularjs.org/api/ng/directive/ngRepeat> [Última consulta: 31 de mayo de 2016].

[11] Protractor API. by.repeater. <https://angular.github.io/protractor/#/api?view=ProtractorBy.prototype.repeater > [Última consulta: 31 de mayo de 2016].

[12] Mozilla Developer Network (2016). Promesa – JavaScript. <https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promesa> [Última consulta: 31 de mayo de 2016].

[13] Desarrolloweb (2016). Manual de AngularJS. <http://www.desarrolloweb.com/manuales/manual-angularjs.html> [Última consulta: 31 de mayo de 2016].

[14] Desarrolloweb (2014). Por qué AngularJS. <http://www.desarrolloweb.com/articulos/por-que-angularjs.html> [Última consulta: 31 de mayo de 2016].

[15] Desarrolloweb (2014). Qué es AngularJS. <http://www.desarrolloweb.com/articulos/que-es-angularjs-descripcion-framework-javascript-conceptos.html> [Última consulta: 31 de mayo de 2016].

[16] Bootstrap. The world’s most popular mobile-first and respondive front-end framework. <http://getbootstrap.com/> [Última consulta: 31 de mayo de 2016].

[17] Bootstrap. CSS. <http://getbootstrap.com/css/> [Última consulta: 31 de mayo de 2016].

[18] Jenkins. Build great things at any scale. <https://jenkins.io/> [Última consulta: 31 de mayo de 2016].

[19] Jenkins Wiki (2015). Remote access API. <https://wiki.jenkins-ci.org/display/JENKINS/Remote+access+API> [Última consulta: 31 de mayo de 2016].

[20] Stackoverflow (2014). AngularJS $http, CORS and http authentication. <http://stackoverflow.com/questions/21455045/angularjs-http-cors-and-http-authentication> [Última consulta: 31 de mayo de 2016].

[21] Rajasreesite (2015). Jenkins API client. <https://rajasreesite.wordpress.com/2015/12/18/jenkins-api-client/> [Última consulta: 31 de mayo de 2016].

[22] Jenkins Wiki (2014). Authenticating scripted clients. <https://wiki.jenkins-ci.org/display/JENKINS/Authenticating+scripted+clients> [Última consulta: 31 de mayo de 2016].

[23] Jenkins Wiki (2016). Parameterized Build. <https://wiki.jenkins-ci.org/display/JENKINS/Parameterized+Build> [Última consulta: 31 de mayo de 2016].