UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

**Grado en Ingeniería Informática**

Trabajo Fin de Grado

Análisis de frameworks de testing unitario para JavaScript

**Autor:** Múgica Gómez Carlos

**Tutor/es:** Hilera González José Ramón

**TRIBUNAL:**

**Presidente:**

**Vocal 1º:**

**Vocal 2º:**

**FECHA**:

INDICE

[Capítulo 1. Introducción 5](#_Toc37580816)

[Capítulo 2. Testing de software 6](#_Toc37580817)

[Tests unitarios 6](#_Toc37580818)

[Tests de integración 8](#_Toc37580819)

[Tests de sistema 9](#_Toc37580820)

[2.1. Test Driven Development 10](#_Toc37580821)

[2.2. Behaviour Driven Development 10](#_Toc37580822)

[Capítulo 3. Análisis de un framework de Unit Testing 12](#_Toc37580823)

[Composición de un framework de UT 12](#_Toc37580824)

[Características de un framework de UT 12](#_Toc37580825)

[Entorno de ejecución: 13](#_Toc37580826)

[La calidad de las afirmaciones 13](#_Toc37580827)

[Dependencias 13](#_Toc37580828)

[BDD/TDD Friendly 13](#_Toc37580829)

[Asincronía 14](#_Toc37580830)

[Dobles de test 16](#_Toc37580831)

[Integración continua 20](#_Toc37580832)

[Tamaño de la comunidad 20](#_Toc37580833)

[Testing visual 20](#_Toc37580834)

[Capítulo 4. Análisis de los frameworks 21](#_Toc37580835)

[4.1. Jasmine 21](#_Toc37580836)

[Instalación: 22](#_Toc37580837)

[Probando Jasmine 24](#_Toc37580838)

[Resumen 26](#_Toc37580839)

[4.2 Mocha 27](#_Toc37580840)

[Probando Mocha 27](#_Toc37580841)

[Resumen: 28](#_Toc37580842)

[Karma 29](#_Toc37580843)

[AVA 29](#_Toc37580844)

[Tape 29](#_Toc37580845)

[Jest 29](#_Toc37580846)

[Sinon 29](#_Toc37580847)

[Capitulo 5: Conclusiones 30](#_Toc37580848)

[Capítulo 6. Caso práctico 31](#_Toc37580849)

[Preparando el entorno de test 31](#_Toc37580850)

[Desarrollando los tests 32](#_Toc37580851)

[Testing unitario en Node JS 32](#_Toc37580852)

[Referencias 38](#_Toc37580853)

**Abstracto**

El siguiente documento pretende mostrar una imagen del estado actual del testing unitario en JavaScript y ofrecer una comparativa entre los principales frameworks disponibles para facilitar la elección de uno de ellos.

Además pretende ser una guía para el inicio en el testing unitario en JavaScript mediante el uso de uno de los frameworks analizados.

The following document shows the state of the art in the JavaScript Unit Testing and presents a comparation between the most important frameworks, helping to choose one.

Also it wants be a starter guide to JavaScript unit testing using one of the analyzed frameworks.

Palabras clave

JavaScript, Framework, testing, unit test, TDD .

[Figura1 Pirámide de testing (Iturregi, 2019) 6](#_Toc37580854)

[Figura 2 Ciclo de vida TDD (Proynov, s.f.) 10](#_Toc37580855)

[Figura 3 Ejemplo de sintaxis de Gherkin (Cucumber, 2019) 11](#_Toc37580856)

[Figura 4 Ejemplo callback 14](#_Toc37580857)

[Figura 5 Ejemplo de uso de Promises 15](#_Toc37580858)

[Figura 6 Ejemplo uso async/await 16](#_Toc37580859)

[Figura 7 Ejemplo de uso de “mock” en Sinon 17](#_Toc37580860)

[Figura 8 Ejemplo de uso “Stub” en Sinon 18](#_Toc37580861)

[Figura 9 Ejemplo uso "Spy" en Sinon 19](#_Toc37580862)

[Figura 10 ejemplo fake 19](#_Toc37580863)

[Figura 11 Ejemplo de uso de dummy 20](#_Toc37580864)

[Figura 12 Interfaz de la calculadora 21](#_Toc37580865)

[Figura 13 Ejemplo test Jasmine 22](#_Toc37580866)

[Figura 14 Estructura Jasmine 22](#_Toc37580867)

[Figura 15 Detalle SpecRunner.html 23](#_Toc37580868)

[Figura 16 Ejemplo de ejecución en Maven (Pérez García, 2012) 24](#_Toc37580869)

[Figura 17 Tests en Jasmine 25](#_Toc37580870)

[Figura 18 Resultado de ejecución correcta de Jasmine 25](#_Toc37580871)

[Figura 19 Test erróneo 25](#_Toc37580872)

[Figura 20 Resultado de ejecución correcta de Jasmine 26](#_Toc37580873)

[Figura 21 Ejecución correcta en Mocha 28](#_Toc37580874)

[Figura 22 Ejecución errónea Mocha 28](#_Toc37580875)

[Figura 23 https://blog.testproject.io/2016/09/01/front-end-development-unit-test-automation-trends2/ 30](#_Toc37580876)

[Figura 24 Instalación de mocha 31](#_Toc37580877)

[Figura 25 Fragmento de package.json 31](#_Toc37580878)

[Figura 26 Instalación de Chai 32](#_Toc37580879)

[Figura 27 Instalación de Sinon 32](#_Toc37580880)

[Figura 28 Método de login de los usuarios 33](#_Toc37580881)

[Figura 29 Inicialización de los test 34](#_Toc37580882)

[Figura 30 Inicializacion de los test del método de login 35](#_Toc37580883)

[Figura 31 Primer test case del login 35](#_Toc37580884)

[Figura 32 Segundo test case del login 36](#_Toc37580885)

[Figura 33 Tercer test case del login 36](#_Toc37580886)

[Figura 34 Resultados de los test case del login 37](#_Toc37580887)

# Capítulo 1. Introducción

En este trabajo se realiza una introducción al mundo del testing unitario, tras la cual se pasa a realizar un repaso por los principales Frameworks de Testing unitario en JavaScript disponibles en internet.

Tras la realización de dicho repaso, se presenta una comparativa entre los mismos y un ejemplo práctico con uno de ellos.

Los objetivos planteados son los siguientes:

* Analizar los frameworks más utilizados que se encuentran en el mercado enfocados a Unit Testing para JavaScript.
* Definir criterios de comparación entre frameworks de Unit Testing para JavaScript.
* Evaluar los criterios de comparación para cada uno de los frameworks seleccionados.
* Realizar una comparación entre los frameworks seleccionados basada en los criterios de comparación evaluados.
* Realizar un caso práctico con uno de los frameworks.

# Capítulo 2. Testing de software

“Program testing can be used to show the presence of bugs, but never to show their absence!” - **Edsger W. Dijkstra**

El testing de software pretende comprobar, de manera automatizada que el código resultante se comporta de la manera esperada. Este ayuda al desarrollo, ya sea identificando errores, requerimientos no cubiertos, o comprobando que en sucesivos mantenimientos del desarrollo, todo sigue funcionando como debería.

Existen múltiples niveles en los que desarrollar las pruebas:

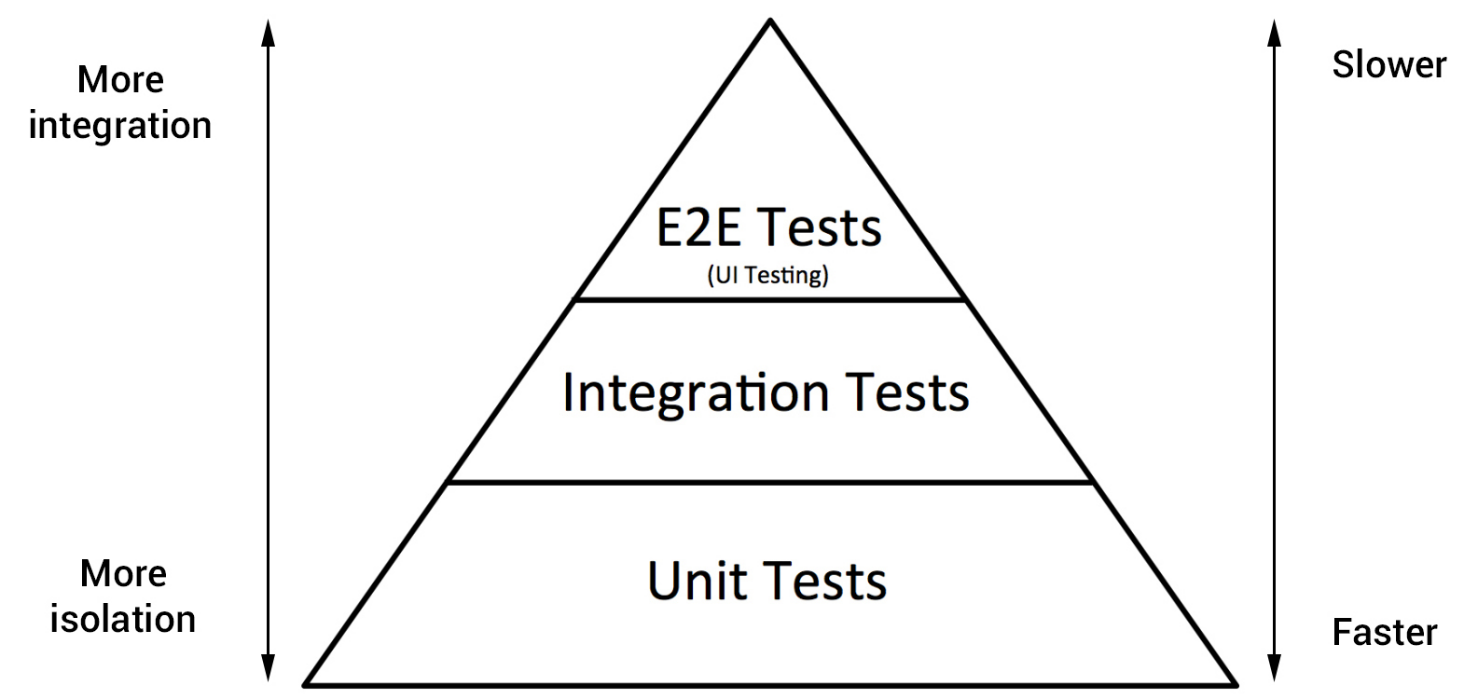


Figura1 Pirámide de testing (Iturregi, 2019)

Como podemos ver en la figura 1, existe multitud de tipos de tests, y su complejidad va aumentando según subimos por la pirámide.

### Tests unitarios

Se centran en comprobar el funcionamiento individual de cada unidad de código.

Un test unitario consta de:

* La descripción de los requisitos que debe cumplir el código para pasar el test.
* La ejecución del SUT (Subject Under Test).
* La aserción, en la que comprobamos que el resultado de la ejecución coincide con lo esperado.

Dependiendo del resultado de la aserción, se dará el test como válido o erróneo.

##### Características

Los test unitarios deben cumplir con las características descritas por el principio F.I.R.S.T. (Martin, 2009):

###### Rápidos (Fast)

Todas las pruebas de nuestro proyecto deberán poder ejecutarse rápidamente, lo cual nos permitiría poder ejecutarlos de manera frecuente.

###### Independientes (Independent)

Los tests deben devolver los mismos resultados, con independencia del orden o la frecuencia en que se ejecuten, o lo que es lo mismo, cada test debe ser independiente del resto.

###### Repetibles (Repeteable)

Las pruebas unitarias no deben depender del entorno en que se ejecuten, es decir, deben reproducir los mismos resultados independientemente del servidor en que lo ejecutemos.

###### Auto-evaluables (Self-validating)

El resultado de un test siempre debe ser OK o KO, de esta forma el test puede evaluarse a si mismo y el resultado es fácilmente legible.

###### A tiempo (Timely)

Las pruebas deben estar desarrolladas lo antes posible, y siempre antes de subir el código a producción, idealmente, siguiendo enfoques de desarrollo como TDD, deberían desarrollarse antes incluso que el código de la propia aplicación.

Aparte de las descritas anteriormente, otras características que deberían cumplir nuestros tests unitarios serían las siguientes:

###### Automatizable

La ejecución de los tests debe poder automatizarse, de tal manera que podamos ejecutarlos sin intervención manual. Esta característica es especialmente útil para poder incorporar nuestros tests en un ciclo de integración continua.

###### Completas

La cobertura de código debe ser lo más alta posible, idealmente debería ser un 100%, aunque en un proyecto real alcanzar dicha cobertura es prácticamente imposible, es el objetivo al que todo proyecto debería aspirar (“*Apunta a la luna y llegarás a las nubes, apunta a las nubes y ni siquiera despegarás*”).

##### Ventajas

Las ventajas de desarrollar tests unitarios son múltiples, por enumerar algunas de ellas:

###### Fomentan la refactorización del código

Si tenemos nuestro código está debidamente testado, podemos modificarlo con mayor seguridad, ya que si cometemos algún error no contemplado, nuestros tests nos avisarán de ello.

###### Reducen el tiempo dedicado a la integración

Al simular las dependencias, no es necesario que otros módulos de los que dependa el nuestro estén desarrollados, basta con la interfaz del mismo, permitiendo el desarrollo en paralelo y simplificando la integración

###### Nos ayudan a entender el código

Si generamos correctamente los tests, haciéndolos autodescriptivos, pueden servir como documentación del propio código

###### Facilitan las pruebas y la depuración del código

Gracias a las pruebas unitarias, podemos evitar montar el entorno completo para probar o corregir una funcionalidad.

##### Limitaciones

Por su propia definición, los test unitarios tienen sus limitaciones, que harán que nuestro código no esté necesariamente exento de bugs, ya que como las unidades de código se prueban de forma aislada, puede que no se descubran errores como pueden ser fallos en la integración entre módulos, problemas de rendimiento, etc.

Además, no siempre se es capaz de anticipar todas las posibles entradas que puede tener un módulo.

Por lo tanto debemos tener en cuenta que los tests unitarios deben ser complementados con otro tipo de pruebas, y mantenidos, incorporando nuevos casos de test conforme vayan siendo necesarios.

### Tests de integración

En este nivel las unidades de código se combinan para ser probadas como grupo en su conjunto para comprobar que no haya errores en la interacción entre los mismos.

El principal objetivo de estas pruebas es comprobar la correcta interacción entre los componentes que probamos en las pruebas unitarias.

En el caso de las pruebas de integración, tenemos varias posibles estrategias a seguir

###### Bottom-Up

Las pruebas se llevan a cabo de abajo a arriba, empezando por las piezas software que no tienen interacción con otras piezas, y continuando por aquellas partes que tan sólo tienen interacción con módulos que ya tienen desarrolladas sus pruebas de integración.

###### Top-Down

De forma análoga a Bottom-Up, esta estrategia se lleva a cabo de arriba abajo.

###### End to End

En esta estrategia se integran en cada caso los componentes necesarios por parte de un proceso de negocio, de esta forma se valida si el flujo completo de la aplicación que estamos probando funciona tal y como se esperaba.

###### Big-Bang

Es la única estrategia de testing de integración que no es incremental, es decir, se realizan las pruebas de integración una vez se disponen de todos los componentes, con los inconvenientes de tiempo y complejidad que eso supone.

### Tests de sistema

Se pretende comprobar que el sistema en su conjunto, y las posibles interacciones con otros sistemas funcionan correctamente una vez se ha podido probar el correcto funcionamiento e integración de los elementos de que se compone nuestro sistema.

Existen varios tipos de pruebas de sistema, entre los cuales pueden encontrarse los siguientes:

###### Pruebas funcionales.

Deben basarse en los requisitos del sistema especificados en los casos de uso y deben verificar el correcto funcionamiento de la aplicación en formato de caja negra, mediante la interacción con la interfaz y analizando que los resultados coincidan con los esperados.

###### Pruebas de comunicaciones

Comprueban que las interfaces de comunicación, tanto con sistemas externos como con el usuario (mediante la GUI, por ejemplo) funcionan correctamente.

###### Pruebas de volumen

Se comprueba que el funcionamiento del sistema es correcto con grandes volúmenes de datos, similares a los esperados en el sistema en producción o definidos en los requerimientos.

###### Pruebas de sobrecarga

Se pretenden superar los límites que soporta la aplicación desarrollada para identificar las condiciones y los puntos en los que el sistema puede fallar.

###### Pruebas de seguridad

Se comprueban los accesos al sistema, así como posibles vulnerabilidades detectadas tanto en el sistema como en las dependencias del mismo.

El desarrollo de software y de los tests ha evolucionado hacia una metodología más Ágil, generando técnicas como TDD o BDD.

## 2.1. Test Driven Development

El TDD es “es una práctica de programación que consiste en escribir primero las pruebas (generalmente unitarias), después escribir el código fuente que pase la prueba satisfactoriamente y, por último, refactorizar el código escrito” (Herranz, 2011). Este ciclo de vida puede verse reflejado en la figura 2.

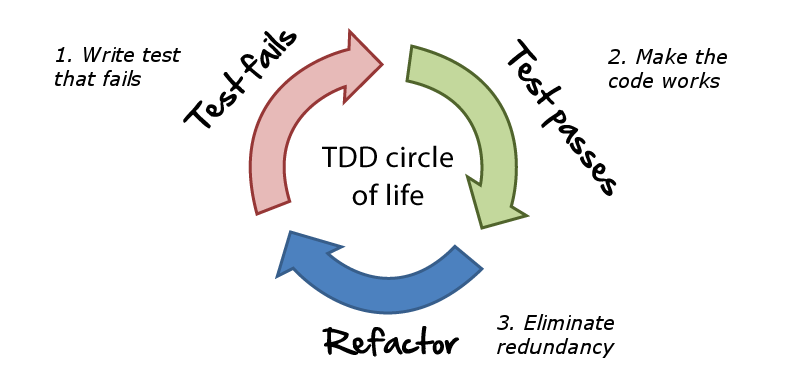


Figura Ciclo de vida TDD (Proynov, s.f.)

Las ventajas principales de aplicar TDD son:

* **Limpieza de código:** Al refactorizar en cada iteración, la redundancia de código se elimina, y los propios tests nos ayudan a que dicha refactorización no provoque nuevos errores.
* **Ayuda en el diseño:** Al comenzar el desarrollo por los tests, nos aseguramos conocer bien la funcionalidad antes de desarrollarla.
* **Generamos los tests:** Al tener que implementar los tests antes de desarrollar la funcionalidad, evitamos la posibilidad de que se desarrolle una pieza de software sin sus tests asociados.

## 2.2. Behaviour Driven Development

El BDD es una evolución del TDD (Wynne , Hellesoy, & Tooke, 2017), mientras el segundo se enfoca en las pruebas unitarias, es decir, que cada pieza funcione correctamente, el BDD centra sus esfuerzos en las pruebas de más alto nivel, en las pruebas funcionales, las de aceptación.

Para la creación de dichas pruebas de aceptación está el lenguaje Gherkin, un lenguaje que una máquina puede interpretar, pero que se puede generar sin necesidad de conocimientos de programación. En la figura 3 podemos ver un ejemplo de la sintaxis de este lenguaje.

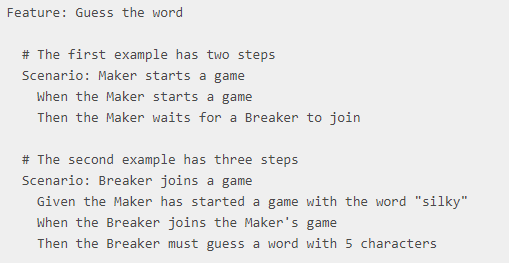


Figura Ejemplo de sintaxis de Gherkin (Cucumber, 2019)

No me extenderé más en el tema del BDD, ya que el objetivo de este trabajo no está en las pruebas de alto nivel sino en las unitarias, pero es necesario exponer el contexto para poder valorar adecuadamente aquellos frameworks que ofrezcan facilidades para trabajar en ámbitos más allá de los tests unitarios.

# Capítulo 3. Análisis de un framework de Unit Testing

## Composición de un framework de UT

Podemos diseccionar un framework de testing unitario de múltiples formas, en este caso, si los dividimos por niveles de abstracción (Zotti, 2015), tenemos la siguiente catalogación del software de testing:

#### Test Runner

Un test runner es la herramienta que se encarga de orquestar y ejecutar los tests recibiendo como entrada la propia definición de los tests y cierta configuración.

Entre las ventajas del uso de un tests runner tenemos, entre otras, la automatización de la ejecución de los tests, por ejemplo al realizar una compilación, puede ir ejecutando los tests que prueben el código que estamos modificando para detectar posibles errores de forma casi instantánea, ejecutar sólo una parte en concreto de los test para acelerar el proceso si estamos debugeando nuestro código, puede levantar múltiples entornos para ejecutar los tests en paralelo, etc.

#### Testing Framework

Es la librería o el conjunto de elementos que nos permiten definir nuestros tests y preparar el entorno para su ejecución.

#### Librería de aserciones

Son las funciones que nos permiten evaluar el resultado de los tests definidos para poder dar el test como OK o KO.

#### Testing plugins

Un plugin de testing es una ampliación del framework, que nos permite crear una mayor diversidad de tests, por ejemplo dándonos la capacidad de crear los llamados “dobles de test”.

Cada una de estas clasificaciones corresponde a un nivel distinto de la ejecución de nuestras pruebas. Como veremos más adelante, los frameworks analizados no necesariamente nos ofrecen todas las piezas aquí descritas, pero generalmente, nos redirigen a software de terceros para cubrir estas carencias.

## Características de un framework de UT

Antes de analizar los distintos frameworks que existen, debemos establecer ciertos criterios en los cuales nos vamos a fijar a la hora de compararlos.

Existen tres criterios fundamentales a la hora de seleccionar un framework (Benitez, 2011):

### Entorno de ejecución:

Por lo general, en JavaScript, el entorno de ejecución es el propio navegador web, por lo que para su funcionamiento se debe cargar un fichero html que referencie a los ficheros JavaScript necesarios (tanto los archivos a testear como los propios del framework).

Aunque también tenemos la posibilidad de desarrollar el lado del servidor con JavaScript gracias a NodeJS o hasta el desarrollo de la interfaz de escritorio como en GNOME Shell (Klapper, 2019), por lo que podría no ser suficiente con el navegador.

### La calidad de las afirmaciones

Cuando desarrollamos tests, nuestro objetivo es asegurarnos de que el resultado de nuestro código es el que esperamos. Esta comprobación se expresa formalmente mediante aserciones.

**aserción**.

Del lat. assertio, -ōnis.

* 1. f. Acción y efecto de afirmar o dar por cierto algo.
  2. f. Proposición en que se afirma o da por cierto algo.

(Real Academia Española, 2019)

Tenemos múltiples tipos de aserciones, desde aserciones que comprueban el valor de objetos como AssertTrue(), AssertFalse(), AssertNull(), AssertNotNull() hasta aserciones que comprueban si el código ha generado una excepción como assert.throws().

La calidad de los tests que generamos, depende en gran parte del conjunto de afirmaciones que nos otorga nuestro framework. Un amplio catálogo de afirmaciones nos permite generar test más fáciles de leer, mantener y modificar.

### Dependencias

Un framework debe tener el menor número de dependencias posibles para garantizar su funcionamiento en el mayor número de entornos.

También tenemos otros criterios a tomar en cuenta a la hora de elegir un framework con el que desarrollar los tests de nuestro proyecto (McPeak, 2018)

### BDD/TDD Friendly

La metodología que vayamos a seguir a la hora de desarrollar nuestro proyecto puede ser un factor a tener en cuenta a la hora de elegir entre frameworks. Por ejemplo, algunos se adaptan mejor que otros al lenguaje Gherkin (Cucumber, 2019) utilizado en el desarrollo guiado por comportamiento o BDD.

### Asincronía

La asincronía es uno de los pilares fundamentales de JavaScript. Este fue diseñado para ser ejecutado en navegadores, trabajar con peticiones sobre la red y procesar las interacciones del usuario, al tiempo que se mantiene una interfaz fluida (Calzado, 2018).

Existen tres tipos de implementaciones de la asincronía en JavaScript:

#### Callbacks

Son funciones que se pasan como argumento al llamar a otra función, de esta forma, esta función se ejecutará en el futuro, una vez la función asíncrona haya terminado.

Por ejemplo, si disponemos del código de la figura 4:

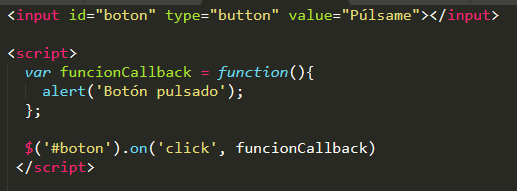


Figura Ejemplo callback

Podemos ver que asignamos al evento clic del botón la función de mostrar una alerta que diga que el botón se ha pulsado. Es un ejemplo sencillo, pero basta para ilustrar que la función ‘funcionCallback’ se ejecutará una vez el evento asíncrono (que el usuario pulse en el botón) haya terminado.

#### Promises

Una promise es un objeto que representa un valor que puede estar disponible ahora, en el futuro, o nunca (Mozilla, 2019).

Al llamar a una función que implementa este patrón, le podemos enlazar dos funciones de callback, uno en caso de que la promesa se resuelva correctamente y otra en caso de fallo.

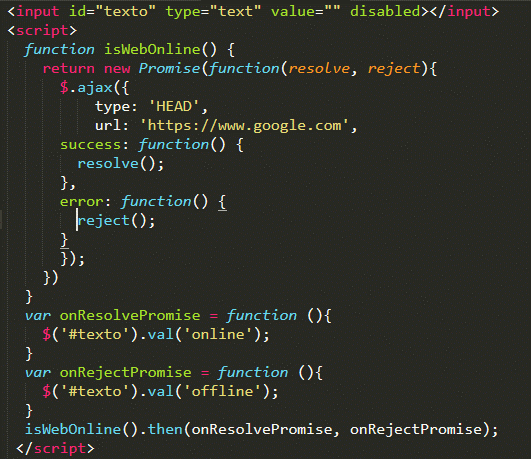


Figura Ejemplo de uso de Promises

En el caso de la figura 5 asignamos al texto el valor “online” u “offline” según el estado en que se encuentre el sitio web de Google, devolviendo desde la función isWebOnline() una promesa y resolviéndola o rechazándola según el resultado de la llamada Ajax .

#### Async/await

Las palabras clave async y await surgieron para simplificar el manejo de las promesas.

La etiqueta async declara como asíncrona una función e indica que la misma devolverá una promesa.

La etiqueta await se sitúa en el interior de una función marcada como async y espera automáticamente a que una promesa se resuelva.

Si transformamos el código anterior para que se realice mediante await/async en lugar de promesas quedaría de la forma que se ve en la figura 6.



Figura Ejemplo uso async/await

El comportamiento del código es exactamente el mismo, pero podemos ver cómo con el uso de estas etiquetas deja de ser necesaria la gestión de la resolución o rechazo de la promesa, dejando que la etiqueta await se encargue automáticamente de la promesa que nos devuelve la llamada Ajax, simplificando y haciendo nuestro código más legible.

### Dobles de test

Los dobles de test u objetos simulados, son objetos que imitan el comportamiento de objetos reales de manera controlada. Se pueden usar tanto en el desarrollo de software, para simular funcionalidades aun no implementadas, como en el desarrollo de tests. En el testing unitario concretamente, con esta clase de objetos, podemos monitorizar que las llamadas a métodos de otros objetos se realizan correctamente, sin la necesidad de que estas sean ejecutadas.

Tenemos varios tipos de objetos simulados (Fowler, MartinFowler.com, 2007):

#### Mock

Son objetos que durante los tests interceptan llamadas a otros métodos de la aplicación. Aunque generalmente se refiera como mock a cualquier tipo de objeto simulado, la principal característica de estos es que el mock pretende imitar el comportamiento real del objeto al que sustituye.

En la figura 7 se puede ver un ejemplo de un mock sobre un objeto “Torno” que registra el número de accesos de los usuarios a unas instalaciones.

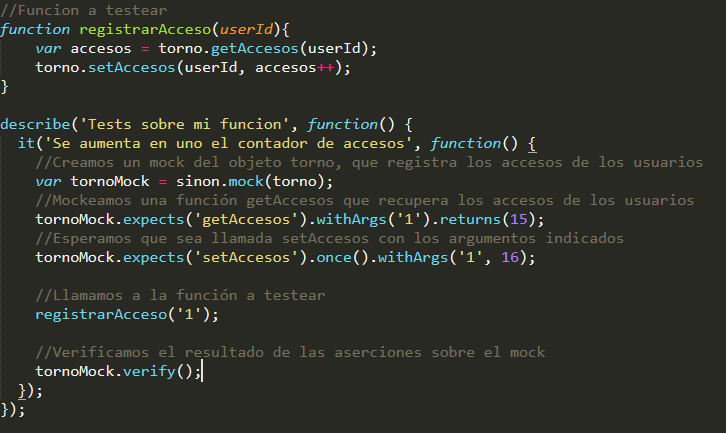


Figura Ejemplo de uso de “mock” en Sinon

#### Stub

De forma similar a los mock, interceptan y anulan llamadas a otros métodos, con la diferencia de que no tratan de imitar el comportamiento del objeto al que están sustituyendo, tan sólo ofrecen una serie de respuestas predefinidas. Uno de los usos más frecuentes es interceptar acciones sobre una base de datos para no alterar la información.

En la figura 8 podemos ver el ejemplo de uso de un Stub, evitando la llamada real sobre la función post de Ajax.

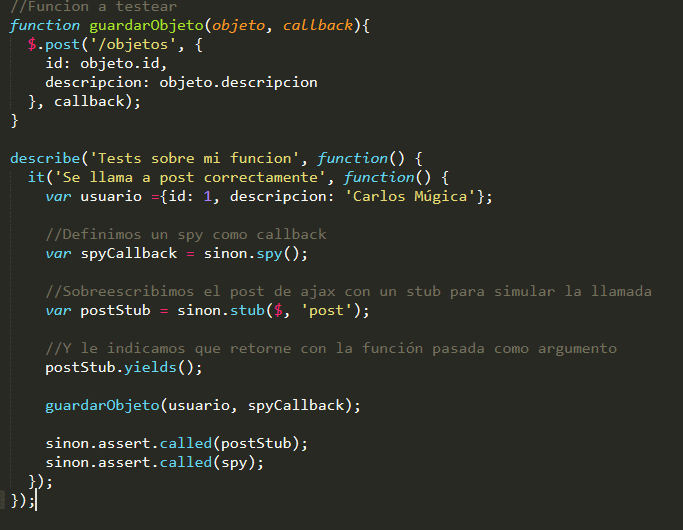


Figura Ejemplo de uso “Stub” en Sinon

#### Spy

Un spy o espía no interfiere en el resultado de la llamada. Se limita a registrar las llamadas que se han realizado sobre el objeto, entre la información registrada podemos encontrar el número de veces que se ha llamado a la función, los argumentos que recibe o si ha generado una excepción entre otras.

En la figura 9 podemos ver un ejemplo del uso de spy usando Sinon.js

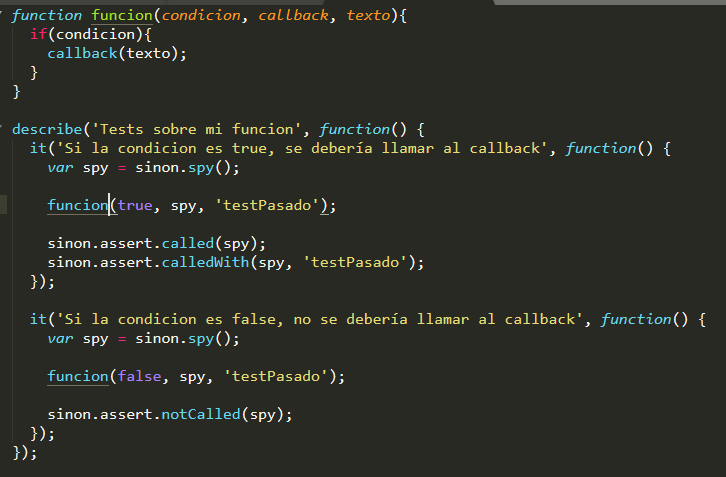


Figura Ejemplo uso "Spy" en Sinon

#### Fake

Son implementaciones mínimas de un componente de nuestra aplicación, con lo mínimo necesario para pasar las pruebas.

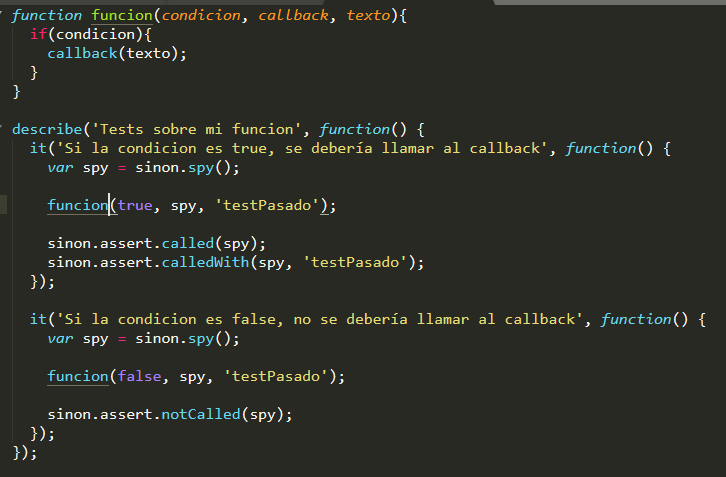


Figura ejemplo fake

#### Dummy

Son objetos que ni siquiera se usan, tan sólo para cumplir con los parámetros en las llamadas a algunos métodos. En la figura 11 podemos ver un pequeño ejemplo de su uso:

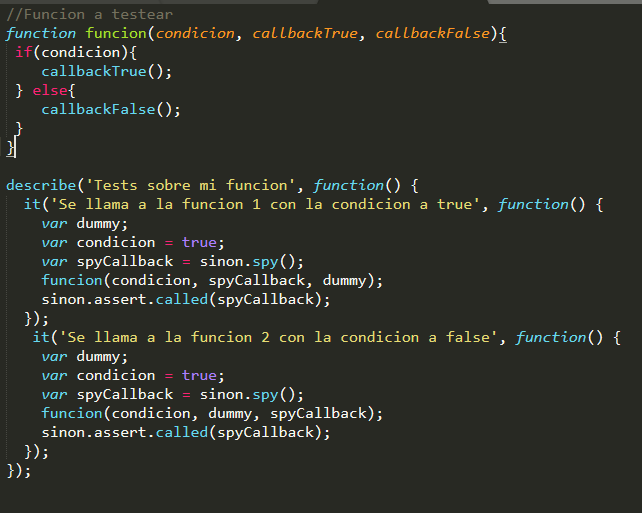


Figura Ejemplo de uso de dummy

### Integración continua

La integración continua se define como una “práctica de desarrollo de software en la que los miembros del equipo integran su trabajo frecuentemente. Cada integración es verificada por un compilado automático (incluyendo tests) para detectar errores tan rápido como sea posible” (Fowler, MartinFowler.com, 2006).

Si el desarrollo se está llevando a cabo bajo integración continua, es un aspecto a tener en cuenta que el framework elegido sea fácil de integrar con la herramienta de CI elegida, como puede ser Jenkins (Jenkins, 2019).

### Tamaño de la comunidad

El número de usuarios del framework es importante, ya que es fuente de tutoriales, documentación y solución a errores que puedas encontrar.

### Testing visual

Algunos frameworks están mejor preparados que otros para el control de efectos visuales o capturas de pantalla. Los tests visuales se centran en cómo se ve la aplicación, más que en cómo funciona. Si la aplicación que se está desarrollando debe ser responsive o debe funcionar sobre diferentes navegadores o sistemas operativos puede ser útil comprobar si el framework elegido facilita estos aspectos de las pruebas.

# Capítulo 4. Análisis de los frameworks

Una vez hemos entendido por qué son necesarios los frameworks y qué características son importantes, podemos empezar a analizarlos.

Para poder hacer un mejor análisis de los frameworks, se ha diseñado una pequeña calculadora sobre la que se podrán probar los mismos, dicha calculadora tan sólo tiene 4 métodos básicos (suma, resta, multiplicación y división), pero será suficiente para evaluarlos y darnos una idea del funcionamiento y de las diferencias entre frameworks.

En la figura 12 podemos ver un ejemplo de la interfaz de la misma.

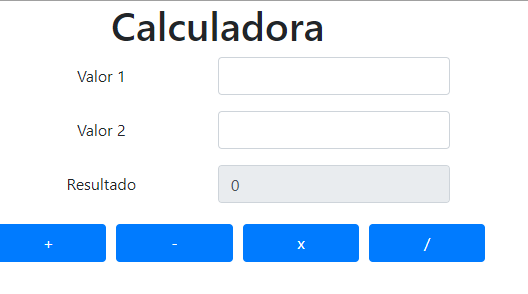


Figura Interfaz de la calculadora

Tanto el código de la calculadora, como de los tests realizados con cada framework se encuentra disponible en el repositorio del proyecto.

## 4.1. Jasmine

Jasmine es un pequeño framework BDD, creado por Pivotal Labs que te permite escribir tests unitarios automatizados para JavaScript (Ragonha, 2013).

La primera versión estable es de Septiembre de 2010 y la última actualización es de Abril de 2019, está programado en JavaScript y opera bajo la “MIT License” (Open source initiative, 2019).

Tiene las ventajas de que no necesita de ninguna otra librería de JavaScript y que no requiere que exista un DOM.

La sintaxis de los tests es sencilla y clara, podemos ver el ejemplo de la figura 13 en la página oficial de Jasmine.

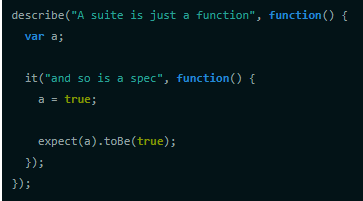


Figura Ejemplo test Jasmine

En el ejemplo podemos ver que el test se compone de los siguientes elementos:

1. **describe:** Una función que define la ‘suite’ de tests.
2. **it:** Los propios tests
3. **expect:** El resultado esperado para pasar el test como correcto.

### Instalación:

La instalación de la versión standalone es simple. Accedemos a la URL del repositorio de GitHub de Jasmine (<https://github.com/jasmine/jasmine/releases>) y nos descargamos el archivo comprimido de la versión que queramos (en este caso la v3.4.0).

Dentro del fichero comprimido nos encontramos la estructura de archivos de la figura 14.



Figura Estructura Jasmine

* **lib:** Contiene las librerias propias de Jasmine.
* **spec:** Contendrá los ficheros de especificaciones (nuestros tests)
* **src:** Aquí se deposita el código a testear.
* **SpecRunner.html:** Fichero HTML encargado de orquestar las pruebas.

Podemos ver el fichero SpecRunner más en detalle en la figura 15:



Figura Detalle SpecRunner.html

Se divide en la siguiente estructura:

* **Líneas 7-12**: referencias a los ficheros propios de Jasmine.
* **Líneas 14-16**: referencias a los ficheros JavaScript a testear.
* **Líneas 18-20**: referencias a los ficheros que contienen los tests.

Una de las ventajas de Jasmine es su fácil integración con Maven (The Apache Software Foundation, 2019). De esta forma si nuestro código JavaScript forma parte de un proyecto mayor, podemos integrar la ejecución de sus tests en el ciclo de vida, mostrando el resultado de la ejecución de la forma que se ve en la figura 16.

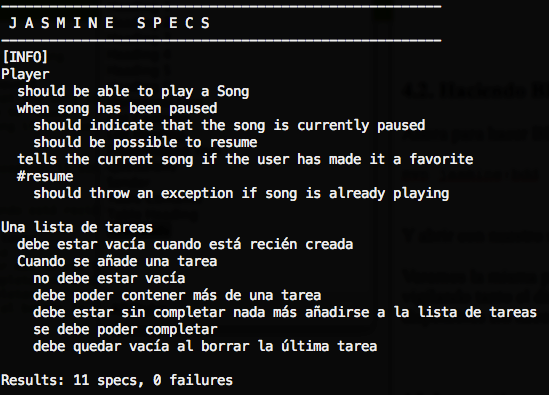


Figura Ejemplo de ejecución en Maven (Pérez García, 2012)

Esto se haría añadiendo el plugin de Maven (jasmine-maven-plugin)en el fichero pom.xml de nuestro proyecto, que, por simplicidad, queda fuera del alcance de este trabajo.

Otra ventaja importante que nos ofrece Jasmine es la integración con NodeJS, instalándolo con npm con la orden:

npm install jasmine-node

Soporte angular

### Probando Jasmine

Una vez hemos aprendido la sintaxis y modificado correctamente el fichero specRunner, creamos una breve suite de tests para probar.

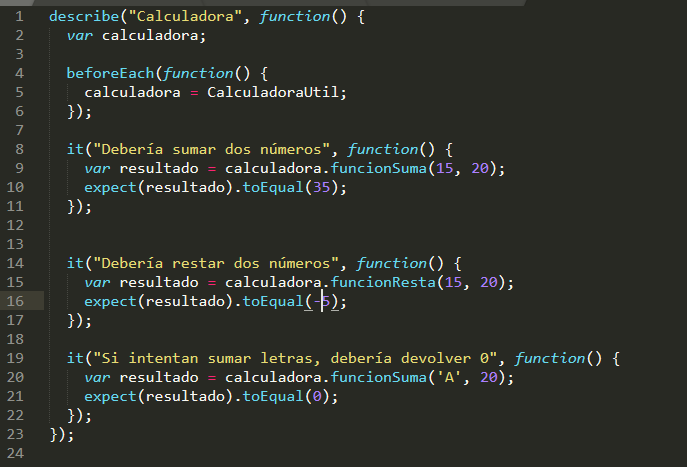


Figura Tests en Jasmine

Y abrimos el fichero SpecRunner.html en el navegador:

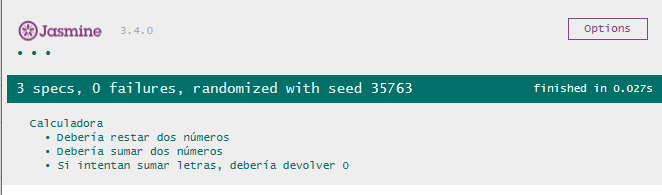


Figura Resultado de ejecución correcta de Jasmine

Como podemos ver nos presenta un simple resumen con el resultado de la ejecución. Si introducimos algún test que no pase, por ejemplo, que esté controlado que no se puede dividir por 0.

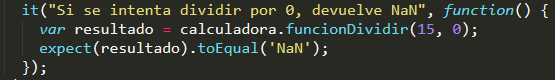


Figura Test erróneo

Jasmine nos devuelve claramente el error



Figura Resultado de ejecución correcta de Jasmine

Se esperaba NaN y se ha devuelto ‘Infinity’.

### Resumen

Guiándonos por los parámetros que definimos antes, este serían las conclusiones sobre Jasmine:

##### Entorno de ejecución

Como hemos podido ver, en su versión “standalone”, tan sólo necesita el navegador web para su ejecución, por lo que no requiere de nada extra.

##### Afirmaciones

La versión por defecto viene con una buena cantidad de aserciones, que podemos consultar en su web oficial (<https://jasmine.github.io/api/edge/matchers.html>), que nos permitirán desarrollar sin problema.

##### Dependencias

Carece de dependencias más allá de los ficheros propios de Jasmine

##### BDD/TDD Friendly

Compatible con ambos sistemas de desarrollo

##### Asincronía

Pdte. investigar

##### Integración continua

Al poder integrarse con Maven, lo podemos integrar con facilidad en el ciclo de vida de nuestros proyectos.

##### Tamaño de la comunidad

Es uno de los frameworks más utilizados para JS, con una comunidad activa que lo mejora constantemente, dándole mayor cantidad de aserciones (Mason, 2018),

##### Testing visual

Pdte.

## 4.2 Mocha

Es un framework simple, flexible y divertido, según las propias palabras de sus creadores. Diseñado para el testing unitario en Node.js, sacó su primera versión en Noviembre de 2011.

Con una sintaxis sencilla y una gran capacidad de compenetración con librerías de aserciones externas, tenemos un framework completo para el testing en Node.js.

### Probando Mocha

Dado que es un framework de desarrollo de tests para Node.js, se ha adaptado el pequeño proyecto de la calculadora a Node.js, de esta manera podremos ver mejor las posibilidades que nos ofrece este framework.

Su instalación es sencilla, a través del Node Package Manager



Con esto tendremos Mocha instalado, e indicando en el fichero Package.json el script para test con mocha, podremos lanzar nuestros tests de forma automática con la orden npm test.

La sintaxis de los test es la misma que en Jasmine, componiéndose también de los bloques describe e it. Por defecto como librería de aserciones usaremos la integrada en el propio node, pero una de las ventajas de Mocha como framework es la integración con otras librerías, por ejemplo Chai.

Ejecutaremos una suite de test similar a la que teníamos en el análisis de Jasmine, con las mínimas diferencias posibles, ya que un ejemplo se realizó con JavaScript puro y este con node, aunque son similares.

Si ejecutamos la suite de tests correcta:

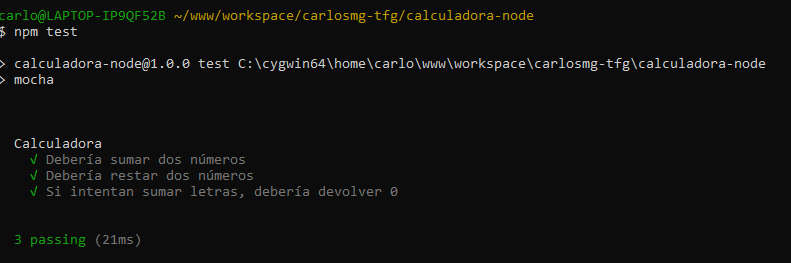


Figura Ejecución correcta en Mocha

Y si introducimos el test erróneo en la suite:

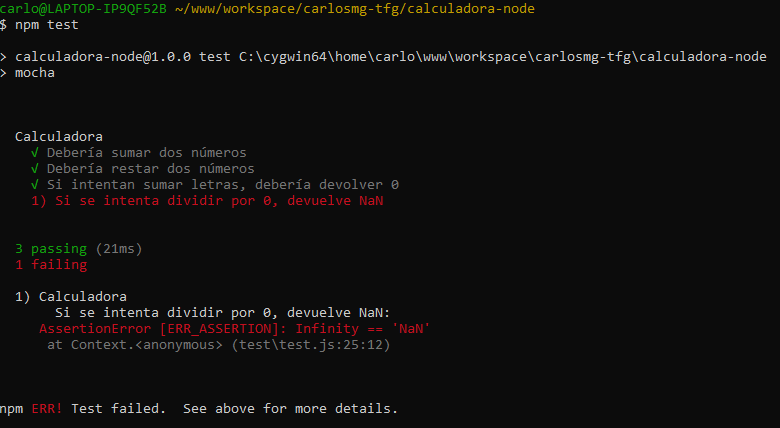


Figura Ejecución errónea Mocha

### Resumen:

Guiándonos por los parámetros que definimos antes, este serían las conclusiones sobre Mocha:

##### Entorno de ejecución

Requiere de instalar el propio entorno de Mocha, aunque esto es sencillo gracias al NPM.

Por otro lado, para poder “experimentar el verdadero poder de Mocha”, necesitaríamos instalar librerías externas como Chai.

##### Afirmaciones

Mocha no dispone de librería de aserciones propia, no obstante como se indica en su propia web, puedes usar cualquier librería cuyos métodos de aserción hagan un “throw” de “Error” (Mocha.js, 2019). Recomiendan librerías de aserciones tales como should.js, expect.js o chai entre otras.

##### Dependencias

Requiere del propio Mocha, aparte, no implementa ninguna librería de aserciones, como mínimo requiere la que incluye node u otras externas como podría ser Chai, should.js, expect.j, etc.

Tampoco trae manera propia de crear dobles de test, delegando esta parte también el librerías externas, en este caso recomiendan el uso de Sinon.js.

##### BDD/TDD Friendly

Compatible con ambos sistemas de desarrollo

##### Asincronía:

En teoría es bastante simple, basta con añadir la función de callback como parámetro de la función “it()”, de esta forma Mocha esperará a que la función sea llamada antes de dar por completado el test.

##### Integración continua:

Pdte.

##### Tamaño de la comunidad

Es uno de los frameworks más utilizados para NodeJs.

##### Testing visual

Pdte.

## 4.3 Karma

Karma es un test runner creado por el equipo de AngularJS. Su principal objetivo es proporcionar a los desarrolladores entornos de pruebas para los test (Karma, 2020), de tal forma que simplifica poder ejecutar los tests contra diferentes navegadores y dispositivos.

### Probando Karma

Para probar Karma, vamos a aprovechar la suite de tests creada en Jasmine y a integrar la misma con Karma. Para ello comenzamos instalando Karma en el proyecto vía npm.

npm install -g karma karma-cli

Y a continuación configuramos karma como se indica en la Figura 23

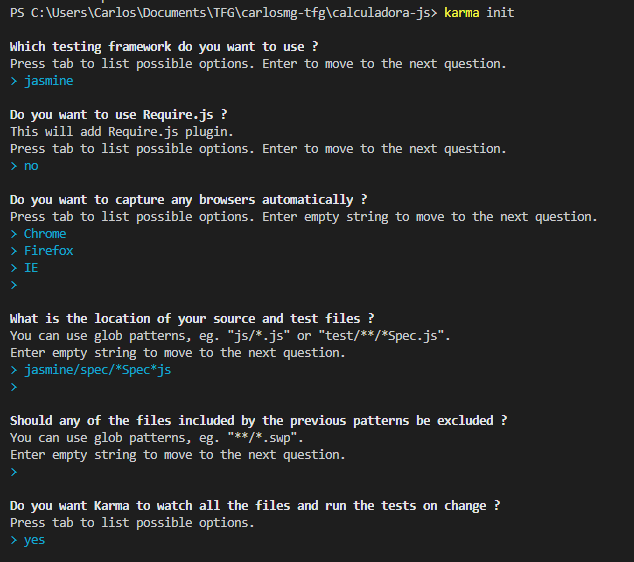


Figura Configuración de Karma

Esta breve configuración nos habrá generado el fichero karma.conf.js, que tal y como podemos ver en la Figura 24 contiene los parámetros de configuración de Karma, tales como los recursos js a cargar, los navegadores sobre los que se ejecutará, etc.

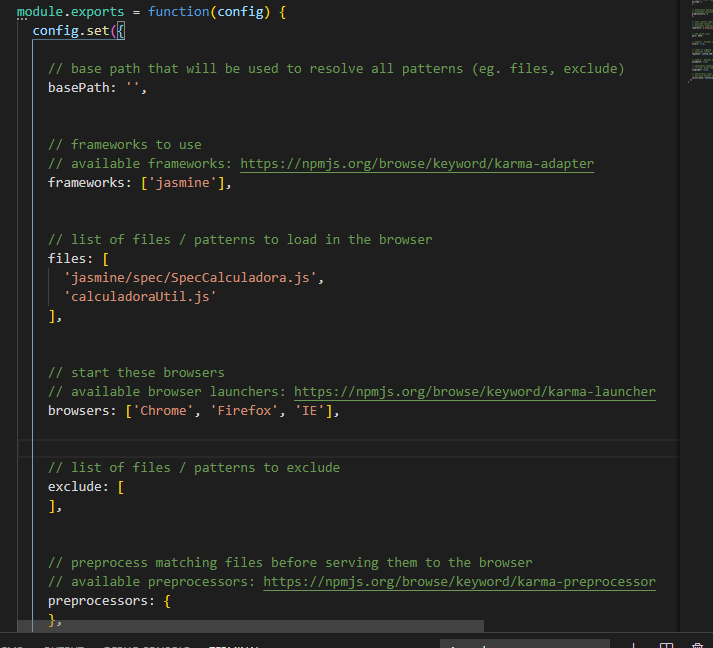


Figura Extracto del fichero karma.conf.js

Ya sólo queda ejecutar y comprobar que Karma lanza los tests contra los navegadores definidos y ver los resultados, para ello lanzamos el siguiente comando.

karma start

Y tal y como vemos en el log de la ejecución del mismo en la Figura 25, podemos comprobar que ha inicializado los navegadores de Chrome, Firefox e IE y ha ejecutado los tests que definimos en las pruebas de Jasmine, devolviendo un reporte del resultado en cada uno de los navegadores.

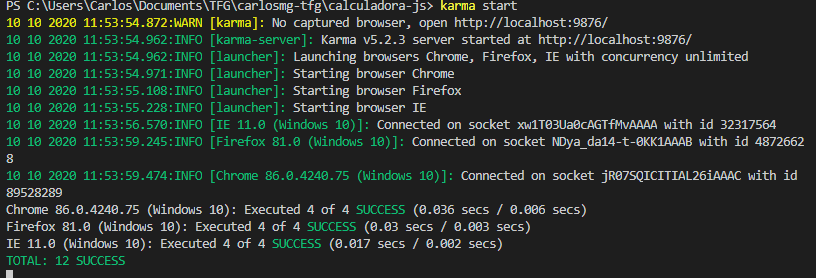


Figura Resultado ejecución correcta en Karma

Para realizar una prueba de que funciona correctamente, y de forma independiente en los tres navegadores, se modifica la función suma como se muestra en la Figura 26.

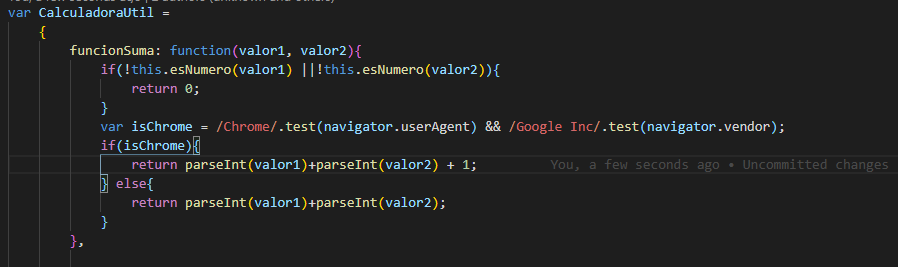


Figura Modificación de test para mostrar error en Karma

De esta forma si el navegador es Chrome añadirá 1 al resultado de la suma, de tal forma que el test definido debería fallar al ser ejecutado en Chrome, pero no en el resto.

Como podemos ver en la Figura 27, Karma se ha comportado de la forma esperada, reportando un error en la ejecución en Chrome, pero funcionando correctamente tanto en Firefox como en Internet Explorer

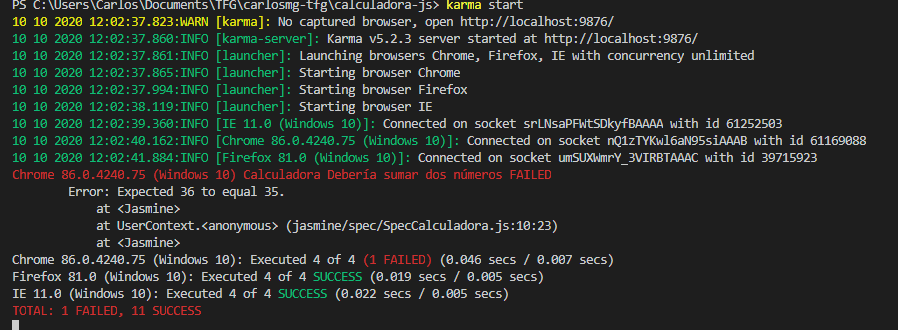


Figura Resultado ejecución incorrecta en Karma

## AVA

## Tape

## Jest

## Sinon

# Capítulo 5: Conclusiones

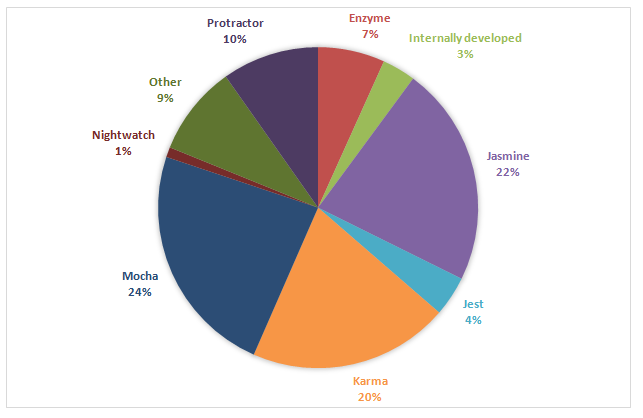


Figura https://blog.testproject.io/2016/09/01/front-end-development-unit-test-automation-trends2/

# Capítulo 6. Caso práctico

A continuación, se hará un uso más profundo y detallado de lo que nos ofrecen algunos de los frameworks analizados.

Para ello se ha desarrollado un API REST sobre el que implementar los tests, implementado con el framework de NodeJs y como sistema de persistencia de los datos se utiliza una base de datos de MongoDB.

Para el testing usaremos los siguientes de los elementos que hemos analizado:

* Mocha como framework
* Chai como librería de aserciones
* Sinon para la creación de los dobles de test
* Karma como test runner.

## Preparando el entorno de test

En primer lugar instalaremos el framework de Mocha vía NPM con el comando de la figura 24



Figura Instalación de mocha

Y modificamos en el fichero package.json la etiqueta scripts para ejecutar fácilmente los tests, de la forma en que se puede ver en la figura 25.

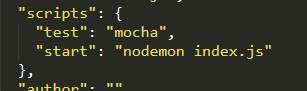


Figura Fragmento de package.json

De esta forma con el comando ”npm test” Mocha ejecutará todos los tests que encuentre.

A continuación, instalaremos Chai como librería de aserciones con el comando de la figura 26.



Figura Instalación de Chai

Y de la misma forma que con los casos anteriores, vía NPM instalaremos Sinon para poder realizar los dobles de test necesarios en el proyecto, ejecutando el comando de la figura 27.



Figura Instalación de Sinon

Una vez tenemos todo instalado estamos listos para empezar a desarrollar los tests.

## Desarrollando los tests

Todos los tests se encuentran disponibles en el repositorio del proyecto, a continuación, se mostrarán algunos de los casos más representativos del desarrollo de tests unitarios en JavaScript con Mocha.

### Testing unitario en Node JS

En la figura 28 podemos ver el código de la parte servidor de NodeJS. Concretamente la función que se encarga de comprobar el resultado de los intentos de login de los usuarios, en los que se comprueba tanto que el usuario existe como que la contraseña coincide.



Figura Método de login de los usuarios

Como podemos observar en el propio código, el email, el cual utilizaremos como identificador del usuario, y la contraseña deben venir incluidas en el “body” de la “request” que recibe la función como parámetro.

En caso de que el email o la contraseña no vengan informados, se deberá responder con un 200, y con un mensaje informando que los datos introducidos son insuficientes. Por lo que nos salen los siguientes cuatro tests unitarios:

1. Si el email es nulo, el estado de la respuesta debe ser 200.
2. Si el email es nulo el mensaje de la respuesta debe mostrar “datos insuficientes”.
3. Si la password es nula, el estado de la respuesta debe ser 200.
4. Si la password es nula el mensaje de la respuesta debe mostrar “datos insuficientes”.

Si el usuario ha informado email y password debemos comprobar que el usuario existe, y en caso de que no, el mensaje de respuesta que dará la función.

Si el email informado existe en nuestra base de datos deberemos comprobar entonces que la contraseña coincide con la que tenemos guardada para el usuario.

A continuación, veremos cómo realizar los tests para esta función.

En la figura 29 tenemos el comienzo del fichero de tests de la clase. En dicha imagen podemos ver como se realizan los imports de las dependencias y se inicializan varios de los dobles de test que vamos a necesitar en el desarrollo de los mimos.



Figura Inicialización de los test

Como podemos ver, el contexto en que se ejecutarán los tests se crean con la función *describe*, la cual recibe como parámetros un literal (la descripción de lo que se pretende probar) y una función que contiene los tests.

Utilizaremos la función *beforeEach*() para inicializar los mocks de request y response antes de cada ejecución de los test.

En la figura 30 tenemos la definición de los test del método de loginUser el cual vimos anteriormente.

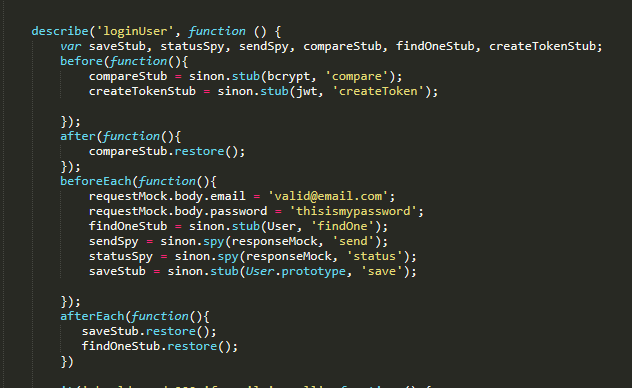


Figura Inicializacion de los test del método de login

Con la función before() declaramos los Stub de la función compare del bcrypt y de la función createToken de jwt, de tal forma que cada test elegirá la respuesta que devolverán dichas funciones cuando sean llamadas.

Antes de cada test, con la función de beforeEach introduciremos valores válidos en el mock de la request e inicializaremos los stubs y los “spy” que se ven en la figura 30.

Ahora que hemos definido el entorno en el que se van a ejecutar los tests, podemos entrar en la definición de los mismos.

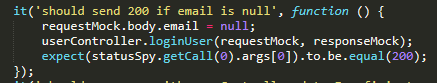


Figura Primer test case del login

En la figura 31 podemos ver el primer test case que definimos para la función del login.

Como ya vimos previamente al definir la estructura de los test cases, el texto de este debe ser descriptivo, en este caso, se define que el objetivo del test es comprobar que el estado de la respuesta es un 200 cuando el email que se informa en la entrada es nulo.

Lo primero que hace el test case es inicializar los datos de prueba, estableciendo a null el email en el objeto mock de la request (el cual como ya vimos se reinicia al comienzo de cada test).

Tras esto ejecuta la función bajo test y lanza la aserción, la cual en este caso es que la función status haya sido llamada con el argumento 200, la cual podemos comprobar gracias al “spy” que creamos sobre dicho objeto.

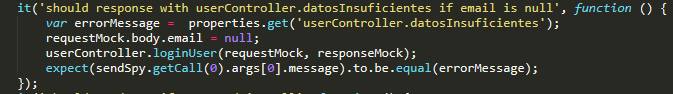


Figura Segundo test case del login

En el caso del test de la figura 32 podemos ver que es similar al anterior, pero en este caso la aserción es que la función send haya sido llamada con un objeto, cuyo parámetro message sea igual al mensaje de error esperado, el cual comprobamos también con el spy correspondiente.

Podría dar la sensación de que se podrían haber introducido ambas aserciones en el mismo test, pero como vimos en la introducción, cada test unitario debe realizar una única aserción.

En el test de la figura 33 podemos ver un ejemplo del uso de un Stub.

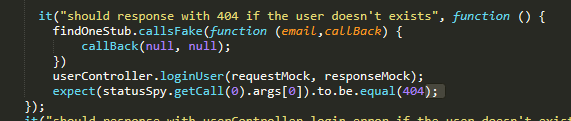


Figura Tercer test case del login

En este caso haremos que el stub de la función “*findOne*” del usuario, el cual reiniciamos al principio de cada test, retorne un null, fingiendo que no ha podido encontrar al usuario en la base de datos, y por tanto debe devolver un 404.

En la figura 34 podemos ver el resultado de la ejecución de todos los tests desarrollados para la función del login.

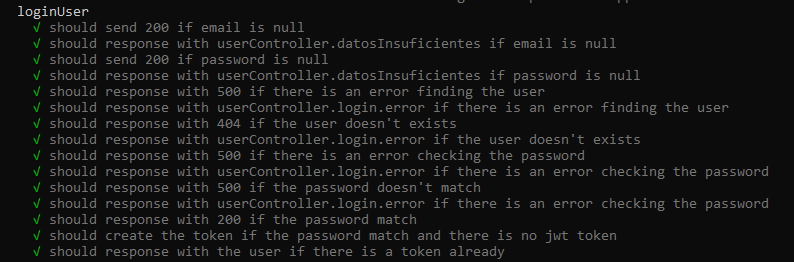


Figura Resultados de los test case del login

# Referencias

ApiumHub. (24 de 08 de 2017). *ApiumHub*. Obtenido de BENEFICIOS DE LAS PRUEBAS UNITARIAS: https://apiumhub.com/es/tech-blog-barcelona/beneficios-de-las-pruebas-unitarias/

Benitez, C. (13 de Febrero de 2011). *TDD en Javascript: II parte*. Obtenido de Funcionamiento e instalación Android SDK: http://www.etnassoft.com/2011/02/13/tdd-en-javascript-2/

Calzado, J. (29 de 1 de 2018). *Lemon Code*. Obtenido de Javascript Asíncrono: La guía definitiva: https://lemoncode.net/lemoncode-blog/2018/1/29/javascript-asincrono

Cucumber. (2019). *Cucumber.io*. Obtenido de Gherkin Reference: https://cucumber.io/docs/gherkin/reference/

Fowler, M. (01 de Mayo de 2006). *https://martinfowler.com*. Obtenido de Continuous Integration: https://martinfowler.com/articles/continuousIntegration.html

Herranz, J. I. (17 de Enero de 2011). *Paradigma Digital*. Obtenido de TDD como metodología de diseño de software: https://www.paradigmadigital.com/dev/tdd-como-metodologia-de-diseno-de-software/

Iturregi, L. (15 de 01 de 2019). *Irontec.* Obtenido de Introducción a la automatización de tests E2E con Cypress.io: https://blog.irontec.com/introduccion-automatizacion-tests-e2e-cypress-io/

Jenkins. (2019). *Jenkins*. Obtenido de https://jenkins.io/

Klapper, A. (15 de 06 de 2019). *https://wiki.gnome.org/*. Obtenido de GNOME Wiki: https://wiki.gnome.org/Projects/GnomeShell

Martin, R. C. (2009). *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall.

Mason, J. (2018). *Github*. Obtenido de Write Beautiful Specs with Custom Matchers for Jest and Jasmine : https://github.com/JamieMason/Jasmine-Matchers#jasmines-default-matchers

McPeak, A. (19 de Enero de 2018). *Web Dev Zone*. Obtenido de The Criteria to Consider for Choosing JavaScript Testing Frameworks: https://dzone.com/articles/the-criteria-to-consider-for-choosing-javascript-testing-frameworks

Mocha.js. (2019). *Mocha, simple, flexible, fun*. Obtenido de Assertions: https://mochajs.org/#assertions

Mozilla. (23 de Marzo de 2019). *MDN Web docs*. Obtenido de Promise: https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\_globales/Promise

Open source initiative. (2019). *Open source initiative*. Obtenido de The MIT License: https://opensource.org/licenses/MIT

Pérez García, A. (10 de Diciembre de 2012). *Adictos al trabajo*. Obtenido de Hello Jasmine! Primeros pasos para hacer BDD/TDD con JavaScript: https://www.adictosaltrabajo.com/2012/12/10/jasmine-hello-world/

Proynov, H. (s.f.). *Devision*. Obtenido de Software Testing Practices and Strategies: https://www.devision.com/software-testing-practices-strategies/

Ragonha, P. (2013). Jasmine And Behavior-Driven Development. En *Jasmine JavaScript Testing* (pág. 27). PacktPub.

Real Academia Española. (19 de Agosto de 2019). *Diccionario de la lengua española*. Obtenido de Aserción: https://dle.rae.es/srv/search?m=30&w=aserci%C3%B3n

Sávio De Oliveira, G., & Duarte, A. (22 de Septiembre de 2014). *A Framework for Automated Software Testing on the Cloud.* Obtenido de Institute of Electrical and Electronics Engineers: https://ieeexplore.ieee.org/document/6904278#targetText=A%20Framework%20for%20Automated%20Software%20Testing%20on%20the%20Cloud&targetText=The%20use%20of%20a%20cloud,and%20heterogeneity%20for%20testing%20coverage.

Scharfstein, A. (12 de Septiembre de 2016). *www.CQSE.EU*. Obtenido de Migrating our JUnit 3 Tests to JUnit 4: https://www.cqse.eu/en/blog/junit3-migration/

The Apache Software Foundation. (2019). *Apache Maven Project*. Obtenido de Welcome to Apache Maven: https://maven.apache.org/

Wynne , M., Hellesoy, A., & Tooke, S. (2017). En *The Cucumber Book: Behaviour-Driven Development for Testers and Developers* (pág. 28). The pragmatic programmers.

Zotti, A. (16 de Marzo de 2015). *What is the difference between a test runner, testing framwork, assertion library, and a testing plugin?* Obtenido de https://amzotti.github.io/testing/2015/03/16/what-is-the-difference-between-a-test-runner-testing-framework-assertion-library-and-a-testing-plugin/