Tutorial para pruebas unitarias con Karma y Jasmine.

1. **Introducción.**

Cuando desarrollamos una aplicación, es necesario generar test sobre los distintos elementos de nuestra App para comprobar que cada uno de los elementos mínimos de código funcionan de manera correcta.

Hay que diferenciar entre prueba funcional y prueba unitaria. La prueba funcional nos permite comprobar si se cumple una especificación, un requisito o una funcionalidad de la aplicación, aunque está utilice diferentes componentes, servicios y capas (Ej: para hacer el log necesitaremos un servicio, un componente y algunos métodos).

La prueba unitaria se utiliza para probar una unidad mínima funcional de código (Ej: comprobar si un método nos devuelve un número o una cadena de caracteres, si existe una etiqueta o si ante un evento se genera código específico en el DOM). Es muy importante ser plenamente consciente de que cada prueba debe quedar totalmente encapsulada y no requerir ningún elemento externo a la propia prueba, es decir, si un método tiene valores de entrada, estos deberán ser proporcionados por la persona que realiza la prueba y no enviárselos desde otro método/servicio.

1. **¿Qué es Jasmine?**

Jasmine es un Framework para pruebas unitarias basado en JavaScript el cual nos permite hacer pruebas, si no también obtener los resultados de estas de una manera visual y con lenguaje humano.

1. **¿Qué es karma?**

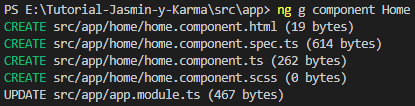
Karma es lo que se denomina un “test runner”, es decir, nos permite automatizar las pruebas creadas por otros Frameworks (como Jasmine).

1. **¿Cómo instalamos y utilizamos estos 2 Frameworks?**

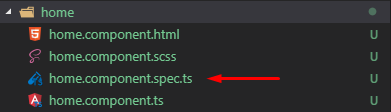
Ambos entornos vienen instalados en la creación básica de un proyecto Angular, por lo que no vamos a necesitar hacer ninguna instalación o configuración inicial.

¿Como creamos una prueba unitaria?

Si estamos utilizando Angular CLI para la creación de componentes (Si no se está haciendo así, empieza a hacerlo), cuando generas un componente nuevo se creara automáticamente un fichero con el nombre del documento pero con la especificación “**.spec.ts**”.



\*1



\*2

En este fichero realizaremos todas las pruebas unitarias sobre el componente homónimo.

En caso de querer hacer pruebas unitarias sobre un servicio (no lo generamos como un componente, si no como un “service” será necesario generar el fichero de manera manual, generando el código básico de un componente de pruebas que se explicará a continuación.

1. **Estructura básica de un componente de pruebas utilizando Jasmine.**

Cuando generamos el componente mediante el CLI de Angular, este ya viene construido con una estructura y un par de comprobaciones básicas cuya estructura es la siguiente.

\*3

1. Zona de importaciones: como se ha mencionado antes, todas las pruebas deben de ser autónomas e independientes, por lo que seguramente tengamos que añadir directivas como las de “Angular Material”, el componente o librerías propias de Jasmine. Para hacer la importación es exactamente igual que en el resto de los componentes Angular.

Aquí tenemos que destacar (por que probablemente no hayas trabajado nunca con ellas) tas librerías “TestBed” y “ComponentFixture”.

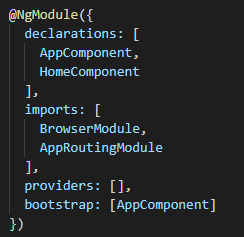
- TestBed: Esta librería nos va a permitir generar una clase “@NgModule” para este componente de manera automática. Se utilizará en la siguiente sección de este punto.

- ComponentFixture: Está librería nos va a permitir crear una instancia del componente sobre el que vamos a realizar todas nuestras pruebas.

1. Definición de variables: En esta sección podemos definir variables que utilizaremos a lo largo de nuestras pruebas unitarias.
2. Módulo principal: También generaremos el módulo de la aplicación gracias a la librería TestBed.

Como podemos ver, sigue la misma estructura que los módulos utilizados en Angular, por lo que tendremos las mismas secciones en caso de ser necesarias.

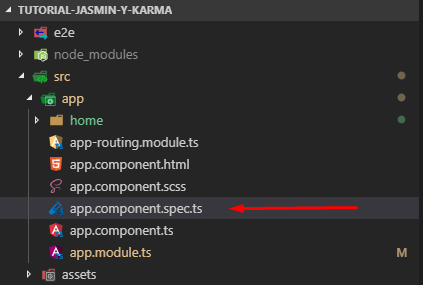
\*Ejemplo del modulo principal de la aplicación.



\*4

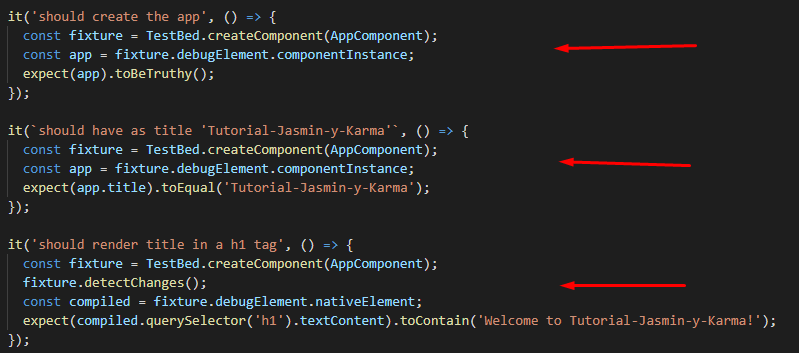
1. Comprobación de creación de componente: este método nos va a permitir crear una instancia del componente sobre el que vamos a hacer todas las pruebas. Permite crear un método que detecte cambios es nuestro componente para poder actualizar la batería de pruebas de manera constante.
2. Pruebas unitarias: aquí ya podemos comenzar a generar nuestras pruebas funcionales. Tema que trataremos en este manual más adelante.
3. **Lanzando las pruebas y entendiendo el menú.**

Ahora que ya tenemos conocimientos generales sobre que son las pruebas unitarias y como generar sus componentes, vamos a comenzar a utilizarlas. Para empezar, vamos a lanzar las pruebas unitarias básicas creadas por Angular para el componente “App” del proyecto de ejemplo.



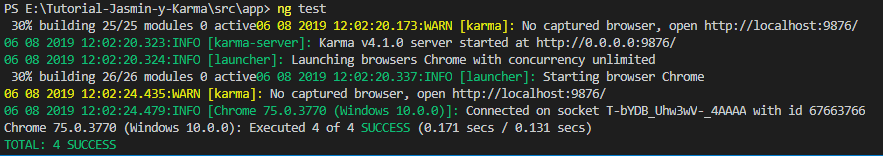
\*5

Como se puede ver, en este fichero encontramos 3 pruebas unitarias distintas, las cuales veremos en profundidad mas adelante, ahora solo nos interesa como ejecutar estas pruebas y ver qué información nos devuelven.



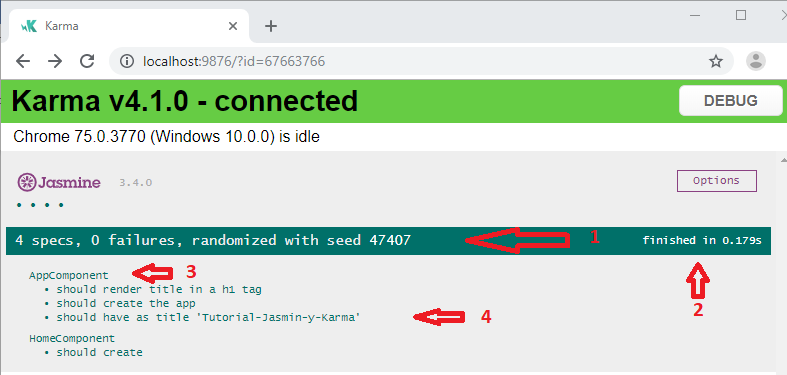
\*6

Para lanzar las pruebas debemos ejecutar el comando “**ng test**” desde la terminal, de manera similar a cuando hacemos el “ng serve” para lanzar nuestro proyecto.



\*7

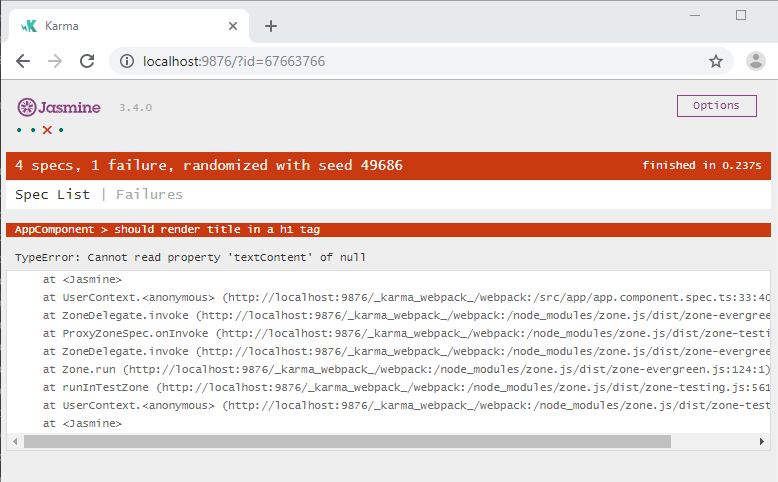
Cuando el servidor en localhost se haya lanzado completamente nos aparecerá una ventana en nuestro navegador similar a esta si es correcta:



\*8

1. Total de pruebas unitarias realizadas: hay 4 pruebas unitarias que hemos realizado. En las líneas de abajo podemos ir viendo cuales se han realizado y a que componente pertenecían. También se especifica el numero origen o semilla que se ha utilizado para realizar estas pruebas de manera aleatoria.
2. Tiempo que han tardado en realizarse: en cuantas mas pruebas se incluyan, mas tiempo se tardará en realizar, esto te permite conocer cuanto es exactamente (para que, si de media son 5 minutos, no te quedes mirando a la pantalla).
3. Componentes: esta es la lista de componentes sobre los que se han realizado todas las pruebas unitarias.
4. Acciones: acciones que debería haber realizado la prueba y si se ha hecho de manera correcta o no.

Ahora vamos a ver como es la salida de una prueba unitaria errónea:



\*9

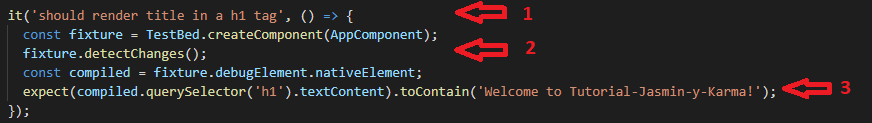
Como se puede ver, la estructura es muy similar, con la diferencia que ahora se especifica la prueba que no ha sido superada correctamente y el componente al que pertenece dicha prueba, además del código de error correspondiente a la prueba que nos permitirá ubicar más fácilmente el fallo.

1. **Creando nuestras primeras pruebas.**

Ahora que ya sabemos para que valen, que estructura tienen sus componentes, como se ejecutan y como utilizar sus resultados, vamos a crear nuestras pruebas unitarias personalizadas.

Lo primero es distinguir entre pruebas unitarias del componente(typescript) y de la parte HTML(DOM), ya que, aunque su formato, estructura y métodos son iguales, existen diferencias a la hora de instanciar las variables que vamos a utilizar.

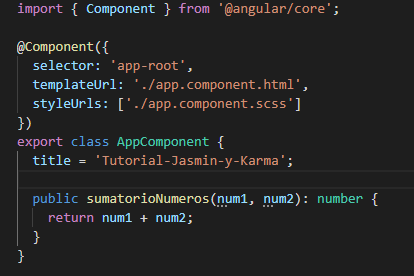
Para ello vamos a ver su estructura, en este caso, la prueba unitaria comprueba si la página contiene una etiqueta “**<h1>**” y si el contenido de esta coincide con el que se muestra en la foto.



\*10

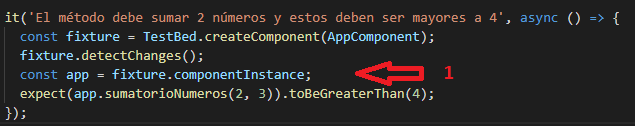
1. Método contenedor de la prueba y descripción de la prueba unitaria: aquí vamos a especificar mediante una cadena de texto, cual es el objetivo de la prueba, por lo que debería ser lo mas específico posible. Este texto es el que se muestra en la pantalla de resultados de las pruebas (Foto número 8, punto 4).
2. Generación del componente y subscripción a cambios: aquí generamos la instancia al componente y nos subscribimos a sus cambios. Esto ya se ha explicado en la foto 3, punto 4.
3. Prueba unitaria: aquí es donde realmente hacemos la prueba unitaria.
   1. Lo primero que hacemos es usar el comando “expect” para decir al método que es lo que debería hacer esta prueba.
   2. Utilizamos la variable “compiled” creada anteriormente (a la que hemos asignado el valor de “fixture.debugElement.nativeElement” por que vamos a trabajar con el DOM) para empezar a poner nuestras exigencias, en este caso queremos que posea una etiqueta “**<h1>**”.
   3. También queremos exigir que esta etiqueta dentro de nuestro código posea una cadena de caracteres concreta que especificamos dentro de la función “**toContain ()**”.

Ahora vamos a hacer lo mismo, pero con una prueba que se encargue de trabajar con la parte del componente(typescript) que sume 2 números y devuelva un número mayor que 4. Para esto, lo primero que tenemos que hacer es crear este método en el componente (en nuestro caso, lo estamos creando en “app.component.ts”.



\*11

Una vez ya hemos creado el método que queremos depurar con nuestras pruebas unitarias, tenemos que crear nuestra prueba. Para ello vamos a nuestro componente de pruebas (app.component.spec.ts) y creamos el método que hará la prueba.



\*12

Como podemos ver, ahora ya no utilizamos el contenido de la variable “fixture.debugElement.nativeElement”, si no que ahora estamos utilizando “fixture.componentInstance”, es decir, la instancia del componente sobre el que hacemos la pruebas.

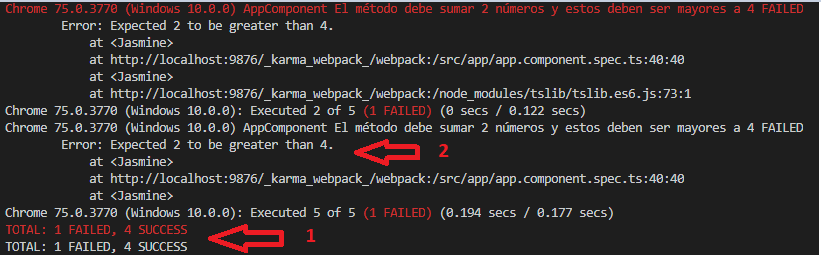
En este caso, desde el método “expect()” accedemos al método que hemos creado anteriormente en el componente (app.component.ts) y le pasamos los valores que nosotros queramos. Después le especificamos que queremos que el resultado de ese método debe ser superior a 4.

Si hemos hecho todo correctamente, al guardar los cambios, la terminal debería informarnos de que se ha añadido una prueba unitaria más y que esta se está realizando correctamente.



\*13

Podemos comprobar que la prueba cambiando el número que le pasamos como parámetro a 0, de tal manera que no se cumpla la condición.



\*14

Como podemos ver en el punto 1, la terminal nos informa que se han realizado 5 pruebas unitarias, pero que 1 de ellas ha fallado. En el punto 2 vemos como nos especifica el error, diciendo que nos ha devuelto un 2, el cual debería ser mayor a 4.

1. **Manuales, documentación y otros conceptos.**

Es muy probable que después de haber visto esto, entiendas como funciona una prueba unitaria, pero no sepas que puedes llegar a comprobar con ellas, por lo que, para no extenderme mucho y hacer un manual de cientos de páginas, te dejo las referencias a la API de Jasmine, en la que te vienen explicados todos los métodos que posee este gran Framework de pruebas.

Es recomendable que antes de seguir con este manual, le des un vistazo a estos enlaces para que tu aprendizaje sobre la utilización mas “avanzada” de Jasmine sea correcta.

1. <https://jasmine.github.io/2.0/introduction.html>
2. <https://jasmine.github.io/tutorials/custom_argument_matchers>
3. <https://jasmine.github.io/tutorials/async>
4. **Creación de pruebas unitarias en servicios con conexiones HTTP.**

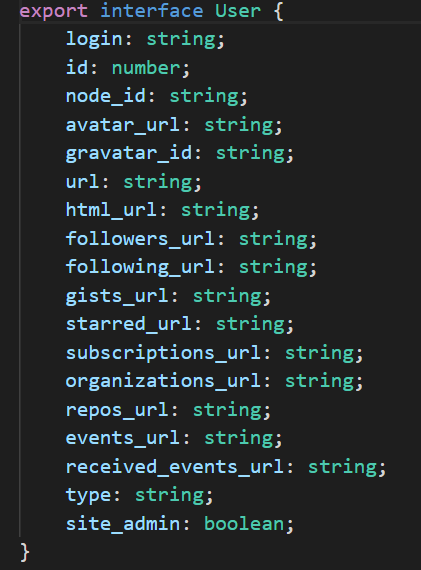
Ahora que ya sabemos para que valen, que estructura tienen y como se ejecutan, vamos a ir un paso más adelante y vamos a comenzar a ver cómo hacer pruebas unitarias sobre servicios que atacan a una API externa mediante el protocolo Http.

Llegados a este punto es necesario destacar que se considera que el lector tiene conocimientos sobre servicios, implementación de librerías y creación de servicios. Si no se posee alguno de estos conocimientos es mejor que primero aprendas esos conceptos antes de continuar con este punto.

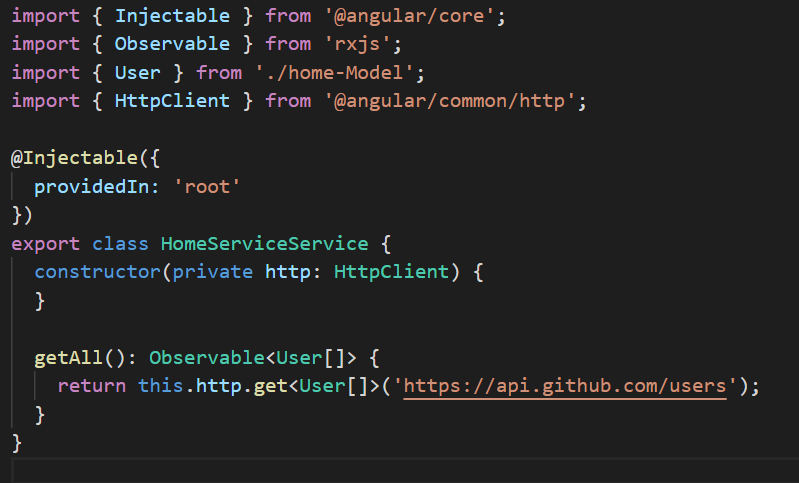
Por último, es necesario aclarar, que las pruebas unitarias de un servicio jamás, y debo hacer mucho hincapié en esta palabra “**JAMÁS**” se harán sobre la BBDD real, ya que en una aplicación pequeña no suponen mucho problema, pero si se hacen en una gran aplicación, tantas llamadas simultaneas pueden perjudicar el rendimiento del servidor

**8.1. Resumen del servicio creado y como se implementa.**

Aquí os voy a dejar un pequeño resumen del servicio que hemos creado, a donde ataca y que modelo usa.



\*15. Interfaz

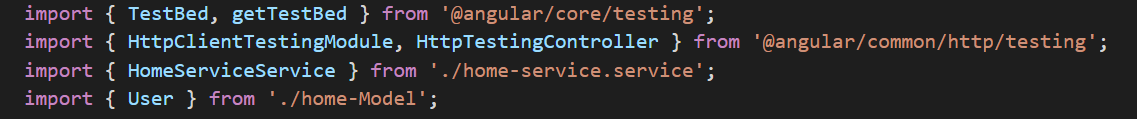


\*16. Servicio

En cualquier caso, Esto se encuentra en el proyecto de ejemplo que acompaña a este tutorial dentro de la carpeta “home/Servicio”.

Ahora vamos a comenzar con el fichero para las pruebas unitarias que genera de manera automática el CLI de Angular.

**8.2. Importaciones de librerías necesarias.**



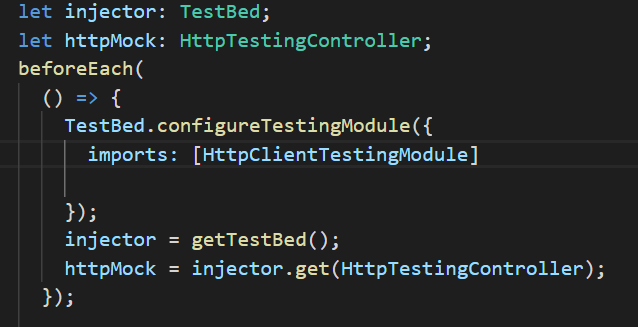
\*17

La primera importación que hacemos es “TestBed” que ya la hemos visto anteriormente y “getTestBed”, la cual nos va a permitir crear la instancia del módulo principal.

La segunda importación son las librerías “HttpClientTestingModule”, la cual, nos va a permitir simular las llamadas Http y “HttpTestingController” que es con la que haremos el mock de los datos para simular las llamadas a la BBDD.

Las 2 últimas importaciones son el servicio y la interfaz que vamos a utilizar para el ejemplo.

**8.3. Creación de variables y asignaciones.**



\*18

A nivel del componente de pruebas, debemos declarar 2 variables que nos ayudarán a trabajar con las pruebas.

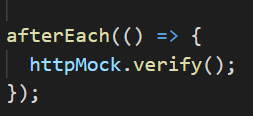
1. Injector: obtendrá las configuraciones del módulo para poder utilizar las librerías importadas.
2. httpMock objeto que nos va a permitir simular nuestra llamada a la BBDD.

¡Importante!: recuerda que el módulo de test unitarios no utiliza en “app.module” que utiliza el componente real, por lo que debemos importar la clase “HttpClientTestingModule” en la configuración del módulo.

**8.4. Integridad después de cada prueba.**

Como las llamadas a un servicio son siempre asíncronas, ya que no podemos saber cuánto tiempo vamos a tardar en obtener la respuesta, es necesario comprobar después de cada llamada, que esta ha finalizado. El cómo ha terminado lo validaremos en cada una de nuestras pruebas, pero después de cada una, es necesario comprobar que de verdad han terminado y su conexión ha sido completamente cerrada.

Para esto vamos a utilizar la función “**verify()**” de la clase “HttpTestingController” la cual hemos almacenado en “**httpMock”.**



\*19

8.5. Haciendo nuestras pruebas unitarias.

Ahora vamos a comenzar a hacer nuestras pruebas. Ya tenemos todo configurado y listo para empezar a simular nuestras conexiones