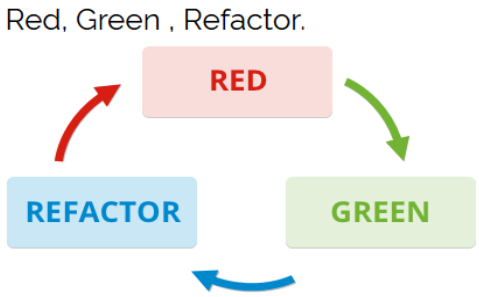
Test – Para lenguajes basados en JavaScripts

Colocaremos los test aparte.
En un fichero .spect.ts



Índice de contenido

Test – Para lenguajes basados en JavaScripts....1
Código para test con Karma y Jasmine: JAVASCRIPT....2
Karma: 2
Jasmine:....2
Instalación para Angular....3
Sintaxis para Jasmine....5
Ejemplo: definir una clase para Test....9
Testing validación asíncrona....9
Servicio virtual (mockeando) Mockito y los dobles....11
¿Cómo voy probando los test que hago?....11
1. Crear un ejemplo:....11
2. Comando para ejecutar pruebas:....12
3. Explorador:....12
Ejemplo test sobre el html....13
Mucho código: Una alternativa....14
Ejemplo con Todos los pasos. Red. Green. Refactor. 15
Tutoriales en la Documentación de ayuda....16
Your first suite....16
Async....16
Custom asymmetric equality testers....16
Custom boot....16
Custom equality (aquí muestra el ToEqual del ej. anterior)....16
Custom matcher....16
Custom object formatters....16
Custom reporter....16
Default spy strategy....16
Mocking ajax....16
React with browser....16
React with node....16
Sharing behaviors....16
Spying on properties....16
Upgrading to jasmine 4.0....16
Use without globals....16

Otro enlace recomendado.....17

Código para test con Karma y Jasmine: JAVASCRIPT

Karma:

Herramienta instalable mediante **npm**.

Jasmine:

Jasmine permite escribir y ejecutar pruebas unitarias . Es necesario, en ocasiones, permitir la ejecución de este conjunto de pruebas en otro entorno diferente al desarrollador/a.

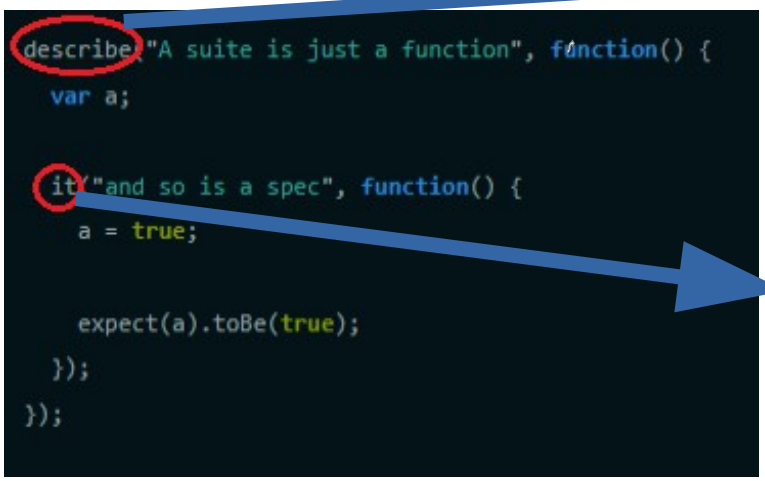
Karma puede utilizarse también para ejecutar nuestras pruebas Jasmine directamente en el seno del IDE, o durante un build para trabajar en integración continua.

Mira información https://books.google.es/books?id=bRLO5jM6-v0C&pg=PA289&lpg=PA289&dq=desarrollador+%2B+Karma&source=bl&ots=yivbxLM5P1&sig=ACfU3U0Qyxy3bcEjLqYOIovpJBtAzO9q7A&hl=es&sa=X&ved=2ahUKEwjZi7jKm5fnAhUSlhQKHWMGCM8Q6AEwBnoECAoQAQ#v=onepage&q=desarrollador%20%2B%20Karma&f=false

Instalación para Angular

Angular ya viene **preconfigurado** para trabajar con **Jasmine**.

Si vamos a la web Jasmine https://jasmine.github.io/, en la página principal veremos lo básico que es crear una prueba unitaria.



Una serie de especificaciones:

describe (" una serie", funtion()

{ }

Los describe son una descripción de lo que harán las pruebas.

"it" como **"describe"** es una función que toma **dos parámetros:**

--el primero es una cadena de caracteres para describir un **título** o **nombre** de la prueba unitaria

--y como **segundo** parámetro una función donde se ejecuta un bloque de código, y la sintaxis es la siguiente it("", function(){})

Los its son Nuestros **Requisitos de Usuarios**

Tenemos instalado en angular.

Por lo que, siempre que creamos un nuevo elemento (un componente, servicio) siempre se creará un archivo de **pruebas**, que está configurado para trabajar con Jasmine.

Pasos Previos:

Nos vamos a saltar la parte de instalar Node.js LTS, Google Chron, Github, **Visual Studio Code** y otros . Como angular: npm install -g @angular/cli

Dónde se crean los test en Angular:

Al crear un componente, servicio, etc. se crará un fichero automáticamente.

El archivo que se crea es app.**NombreDELCOMPONENTE**.**spect.ts**

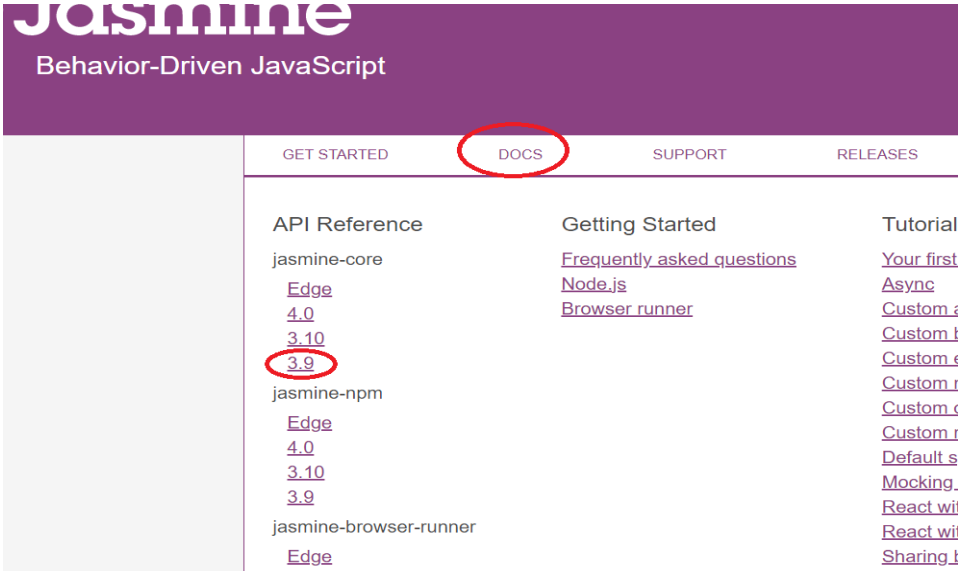
En la **misma** carpeta del componente.

- En este caso, para angula, **no es necesario crear una carpeta aparte** para los test.
- Ya que cada test estará **junto al resto del código** del componente (algo mezclado, no tan claro como en otros lenguajes, o framewords, pero es la forma que lo hace angular).

Sintaxis para Jasmine

Haré una breve introducción, pero te recomiendo entrar en Docs de la web de Jasmine, y mirar ahí la documentación de la última versión, más actualizada.

En la web de Jasmine entra donde dice **docs**



Accede con un clic a la **última versión** . En esta imagen la última versión en 3.9

Aunque si descubres que hay una posterior por ej. **la 4.0 has clic** en esa.

Sus **sintaxis** se basa en **dos funciones**:

- una describe una serie de **especificaciones**,
- y otra describe una **especificación**.

Una serie de especificaciones también permite describir el entorno en el que se van a ejecutar las pruebas unitarias, y configurar los objetos que se van a probar.

Cada especificación permite entonces definir la funcionalidad que se va a probar y el resultado esperado.

Una serie de especificaciones:

describe (" una serie", funtion()

{ }

describe (description, specDefinitions)

{ }

Es la descripción de lo que harán las pruebas.

Cada componente, interface, etc, al ser generado por Angular en su archivo de pruebas vendrá con un **describe**. Y describirá las pruebas sobre dicho componente, interface, etc.

Specs:

Las expectativas en jasmine (son afirmaciones que pueden ser ciertas o falsas dependiendo de cuál resultado el programador espere en la prueba unitaria.

Orden de ejecución:

Jasmine también viene con **funciones** que se pueden ejecutar **antes** de realizar un test, o **después**:

- **beforeAll:** Se ejecuta antes de pasar(de que comiencen) todos los tests de una suite.
- **afterAll:** Se ejecuta después de pasar/finalizar todos los tests de una suite, sobre cada componente.
- **beforeEach**(function(){ } Se ejecuta antes de cada test de una suite.
- **afterEach:** Se ejecuta después de cada test de una suite. Se ejecuta después de cada **beforeEach**.

Comprobar el resultado de una prueba:expect()

expect()

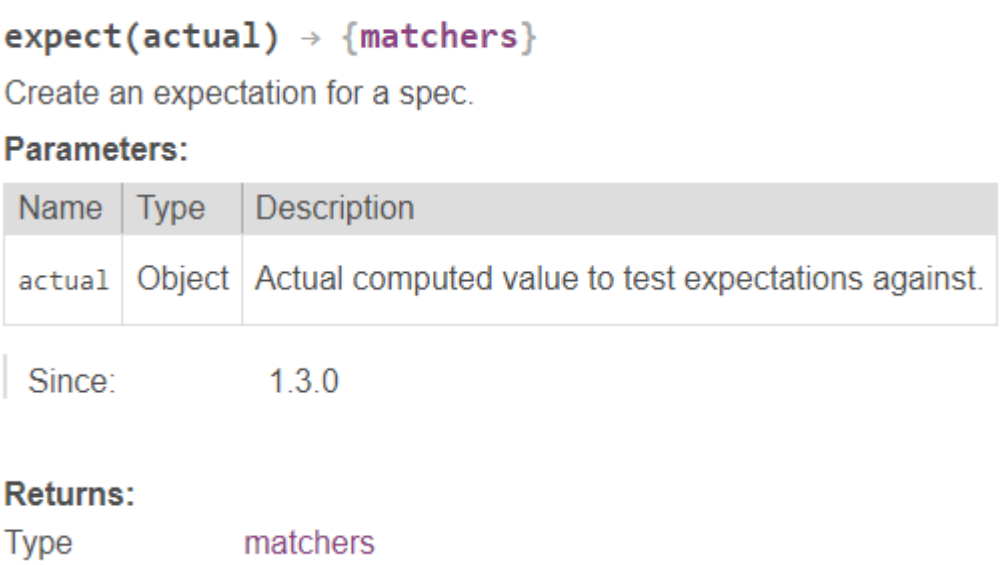
Son construidas con la función "expect" el cual toma el **valor actual**, ese valor está enlazado con la función matcher(inglés) que toma el valor **esperado**.

La sintaxis es la siguiente **expect (valor actual).matchers(valor esperado)**

expect (actual) →{matchers}

Los expct son métodos que espera que el valor que le pasamos como parámetro (actual) cumpla la condición.

Los matchers serían las condiciones que se espera que se cumplan al pasar el test.



`expect(actual)` → `{matchers}`

Create an expectation for a spec.

Parameters:

Name	Type	Description
`actual`	Object	Actual computed value to test expectations against.

Since: 1.3.0

Returns:

Type matchers

fdescribe(description, specDefinitions):

Si al describe le añadimos una f (fdescribe) se utiliza para hacer foco sobre dicho set de pruebas (sobre ese conjunto de pruebas)

Es decir que usamos fdescribe, solo se lanzarán las pruebas de este componente y se omitirán las del resto de componentes, interfaces, etc...

Esto se usa para ahorrar **tiempo** en los ciclos de prueba si queremos hacer unas pruebas y no otras.

it (description, testFunction, timeout):

"it" como **"describe"** es una función que toma **dos parámetros:**

Reciben una descripción y una función.

- el primero es una cadena de caracteres para describir un **título** o **nombre** de la prueba unitaria

-el **segundo** parámetro una **función** donde se ejecuta un bloque de código, y la sintaxis es la siguiente it("", function(){});

Los **it** son nuestros **Requisitos de Usuarios**. Cada uno de nuestros requisitos tendrá un it.

fit(description, testFunction, timeout):

También se les puede añadir la f. Para como antes, hacer foco sobre estos métodos, y omitir el resto (para ir al grano, y no ejecutar otros).

toBe(): Indica que el valor pasado a expect debe ser igual al valor pasado como parámetro.

expect (true).toBe(true);

not: La propiedad not es para invertir el comportamiento de la verificación:

expect(false).not.toBe(tre);

toBeTruthy(), toBeFalsy(): Estos métodos permiten especificar que el valor pasado al expect debe ser igual a True o False (con ayuda del operador ==)

expect (true).toBeTruty();

Anidación("**Nesting**") y Bloques("**Blocks**")

spyOn(obj, methodName) **Objetos Espías:**

Jasmine proporciona un tipo de objeto capaz de trazar todas las acciones realizadas sobre él.

Este tipo de objeto resulta útil, por ejemplo, en el caso de que sea necesario verificar que un método se ha invocado correctamente al menos una vez, o que cierto parámetro se ha pasado durante una llamada.

Básicamente, el spyOn nos permite hacer **peticiones falsas** para probar nuestros métodos GET o POST de nuestros servicios

(es decir, en nuestra Pruebas, nosotros **no vamos a hacer peticiones reales**, usaremos los espías y simularemos una petición http)

Un espía se crea mediante una llamada al método **createSpy** sobre el objeto Jasmines, pasándole el **nombre** del espía.

```
it ("Método invocado al menos una vez", function(){
    var mySpy= jasmine.createSpy("mySpy");
```

spyOn(obj, methodName) → {Spy}

Install a spy onto an existing object.

Parameters:

Name	Type	Description
obj	Object	The object upon which to install the Spy.
methodName	String	The name of the method to replace with a Spy.

Returns:

Type Spy

Otros métodos, mira la documentación:

En la documentación hay más aspecto de las sintaxis de interés.

Por ejemplo para comparar el valor de dos variables toEqual, Comparadores positivos y negativos.

Ejemplo: definir una clase para Test

```
describe('Test unitario de elemento HTML', function() {

  var $compile,
      $rootScope;

  beforeEach(module('adictosTestingDirectivaHTML'));

  // Se almacenan las referencias a $rootScope y $compile
  // para que estén disponibles en todos los tests de este bloque
  beforeEach(inject(function(_$compile_, _$rootScope_){
    $compile = _$compile_;
    $rootScope = _$rootScope_;

    // Sección en la que se encuentra el usuario
    $rootScope.section = 3;
  }));

  it('debería sustituir el elemento tnt-warning por el contenido indicado en el template', function() {


    // Compila código HTML que contiene la directiva
    var element = $compile("<tnt-warning></tnt-warning>")($rootScope);

    // Ejecuta los wathches registrados en el $scope, de esta manera
    // los valores del modelo se propagan al DOM
    $rootScope.$digest();

    // Comprueba si el elemento compilado contiene el contenido de la
    // plantilla
    expect(element.html()).toContain("Revisa que el apartado 3 es correcto");

  });
});
```

Testing validación asíncrona

Para explicar la validación asíncrona se realizó la directiva **tnt-user-signedup.**

Esta directiva simulaba con una promesa la llamada http al **servicio**. Para esta ocasión y poder mostrar los **mocks** de llamadas http para hacer tests se ha implementado la **llamada al servicio.**

Por tanto, lo que queremos es **realizar una llamada a un servicio back-end** que nos indique **si un alias de usuario** está dado de alta en el sistema o no.

El módulo que nos permite realizar **mocks** en los tests de **AngularJS** es ng-mock,

el cual hay que **importar** en el módulo de la aplicación.

El módulo **ng-mock** nos provee el servicio **$httpbackend** para **simular** las **llamadas** a back-end.

Las llamadas a servicios mock pueden responder de manera **síncrona**, por lo que es necesario llamar a la función **flush()** para que se **liberen** las peticiones pendientes.

import {LoginComponent} **from** './login.component';

El test a realizar es el siguiente:

tnt-user-signedup-test.js

```
describe("testsDirectivaValidacionFormularioAsincrona", function() {

  var $compile,
      $rootScope,
      $httpBackend;

  beforeEach( function(){
    module('formAdictos')
  });
  beforeEach(inject(function(_$compile_, _$rootScope_, _$httpBackend_){

    $compile = _$compile_;
    $rootScope = _$rootScope_;
    $httpBackend = _$httpBackend_;

    $httpBackend.when('GET', 'api/users/loginManagement?alias=juan')
        .respond(400, {mensaje: 'usuario no registrado'});

    $httpBackend.when('GET', 'api/users/loginManagement?alias=adrian')
        .respond(200, {mensaje: 'usuario registrado'});
    $rootScope.user = {alias: null};

    var element = angular.element(
        &apos;<form name="userForm"><input name="alias" type="text" ng-model="user.alias" tnt-
user-signedup /></form>&apos;
    );
    $compile(element)($rootScope);
    userForm = $rootScope.userForm;
  }));

  it('debería detectar que el alias no está dado de alta', function () {
        userForm.alias.$setViewValue('juan');
        $rootScope.$digest();
        $httpBackend.flush();
        expect(userForm.alias.$error.tntusersignedup).toBeTruthy(); // lo del error
  });
  it('debería detectar que el alias está dado de alta', function () {
        userForm.alias.$setViewValue('adrian');
```

```
        $rootScope.$digest();
        $httpBackend.flush();
        expect(userForm.alias.$valid).toBeTruthy();
    });
});
```

Servicio virtual (mockeando) Mockito y los dobles

Mockito es una **librería** Java que permite simular el comportamiento de una clase de forma **dinámica**.

De esta forma **nos aislamos de las dependencias con otras clases** y sólo testeamos la funcionalidad concreta que queremos.

La **simulación** del comportamiento de una clase se hace mediante los "**dobles**" que pueden ser de distintos **tipos**:

- **Dummy**: Son objetos que se utilizan para realizar llamadas a otros métodos, pero no se usan.
- **Stub**: es como un dummy ya que sus métodos no hacen nada, pero devuelven cierto valor que necesitamos para ejecutar nuestro test con respecto a ciertas condiciones.
- **Spy**: Es un objeto real que permite **verificar** el uso que se hace del propio objeto, por ejemplo el **número de veces que se ejecuta** un método o los argumentos que se le pasan.
- **Mock**: Es un stub en el que sus métodos sí implementan un comportamiento, pues esperan recibir unos valores y en función de ellos devuelve una respuesta.
- **Fake**: Son objetos que tienen una implementación que funciona pero que no son apropiados para usar en producción (por ejemplo, una implementación de HttpSession).

¿Cómo voy probando los test que hago?

1. Crear un ejemplo:

Si has creado el proyecto y los componentes usando Angular cli, te habrás dado cuenta de que **al generar un componente**, también se crea un **archivo .spec.ts**,

Eso es porque Angular cli se **encarga** por nosotros de generar un archivo para testear cada uno de los componentes.

Además mete en el archivo el código necesario para **empezar a probar** y testear los componentes.

Por ejemplo, el archivo **notes.component.spec.ts** que se creó cuando generé un componente para crear y mostrar notas tiene esta pinta:

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { NotesComponent } from './notes.component';
describe('NotesComponent', () => {
	let component: NotesComponent;
	let fixture: ComponentFixture<NotesComponent>;
	beforeEach(async(() => {
		TestBed.configureTestingModule({ declarations: [ NotesComponent ]
		})
		.compileComponents();
	}));
	beforeEach(() => {
		fixture = TestBed.createComponent(NotesComponent);
		component = fixture.componentInstance;
		fixture.detectChanges();
	});
	it('should create', () => {
		expect(component).toBeTruthy();
	});
});
```

2. Comando para ejecutar pruebas:

Para **correr** los tests y ver los resultados con Angular cli
el **comando** es:

ng test

3. Explorador:

Abrirá un explorador y lanzará las pruebas previamente configuradas con el código que hemos puesto.
Se lanzarán los app.NombreDELComponente.**spect.ts** que hayas codificado previamente.

Ejemplo test sobre el html

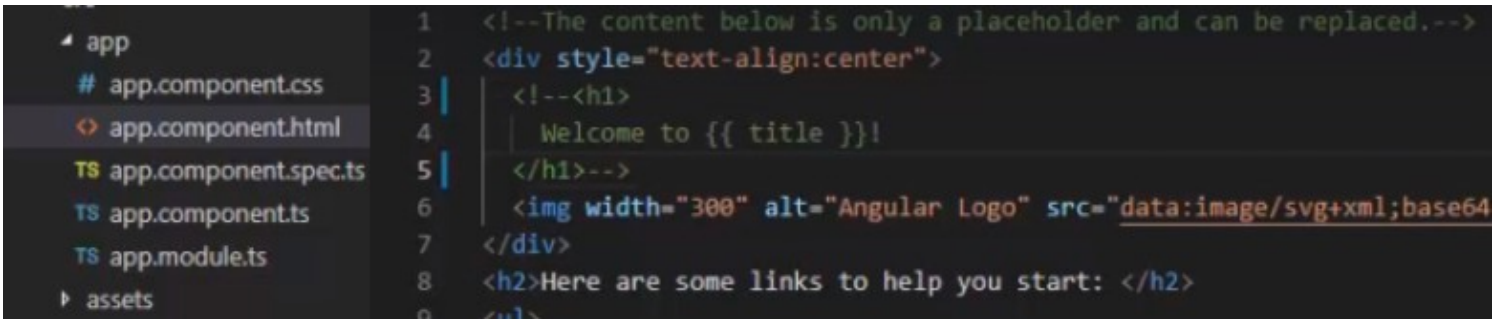
```
it(`should have as title 'IntroPruebasUnitarias'`, async(() => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.debugElement.componentInstance;
  expect(app.title).toEqual('IntroPruebasUnitarias');
}));
it('should render title in a h1 tag', async(() => {
  const fixture = TestBed.createComponent(AppComponent);
  fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement;
  expect(compiled.querySelector('h1').textContent).toContain('Welcome to IntroPruebasUnitarias!');
}));
```

- El **it(' should habe as title** …. Lo que hace es comprobar que el titulo del html es ese.

- En este caso el **it (' sholul render title in a h1 tag', async(() => {** } Lo que haces es comprobar eque en el html de este componente existe un h1, con ese texto.

```
expect(compiled.querySelector('h1').textContent).toContain('Welcome to IntroPruebasUnitarias!');
```

– Si vamos al html y comentamos el h1, esa prueba fallará.



Falla

```
Chrome 69.0.3497 (Windows 10 0.0.0): Executed 3 of 3 (1 FAILED) (0 secs / 0.159 secs)
Chrome 69.0.3497 (Windows 10 0.0.0) AppComponent should render title in a h1 tag FAILED
        Failed: Cannot read property 'textContent' of null
        TypeError: Cannot read property 'textContent' of null
            at UserContext.eval (./src/app/app.component.spec.ts?:28:44)
```

Abre: localhost:9876/ para ver el resultado del Karma

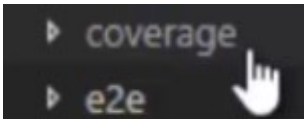
Mucho código: Una alternativa

Si tenemos mucho código, las pruebas pueden ser muy lentas.
Con el comando:

```
ng test --code-coverage
```

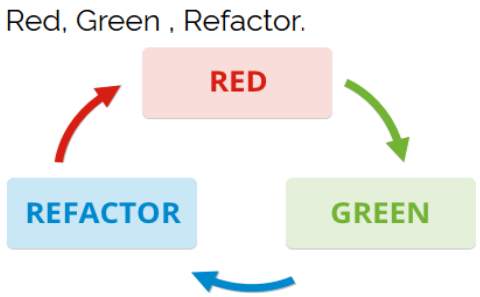
Este comando abre el explorador y lanza las pruebas.

De esta forma una vez lanzadas las pruebas, Instanbul guardaré un **informe** en la carpeta coverage (de angular). Se guardará un conjunto de html, unidos por un index.html.



Ejemplo con Todos los pasos. Red. Green. Refactor.

Partimos del requisito, luego test, luego Código



Paso 1: Voy a mi fichero .spect.ts del Componente.

En mi caso el componente lo estoy llamando app.component

1º Creo un it

```
it ( ' El valor de myVar debe ser Hola Mundo', () => {
```

2º Instancio el componente

```
const appComponent = new AppComponent();
```

3º Declaro una variable para comparar su valor

```
const valor =appComponent.myVar
```

4º Hacer la validación de la prueba, lo hacemos con el método expect.

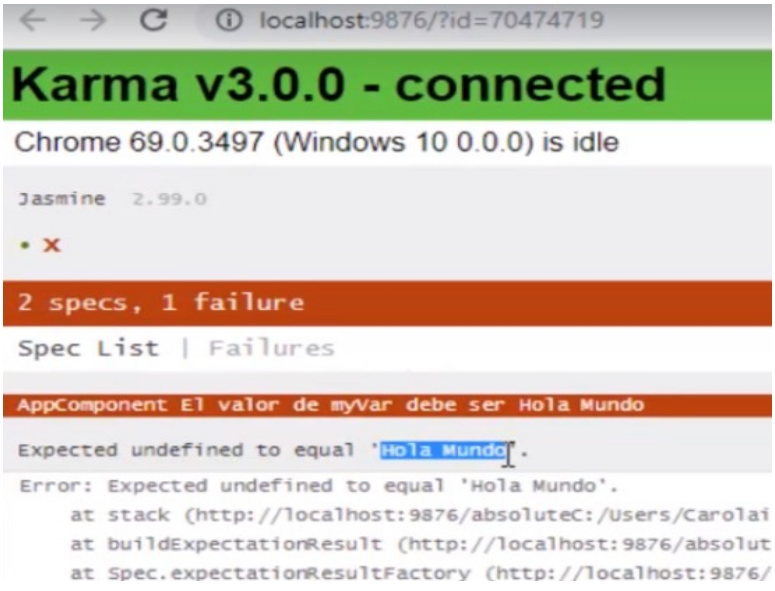
Necesito el método toEqual que nos permite comparar dos valores.

```
expect(valor).toEqual('Hola Mundo')
```

Paso 2: Paso la prueba. Y Falla

ng test

Falla, porque **no hemos declarado** la variable en nuestro componente.



Paso 3: Voy al componente y Creo el método que quiero probar.

En app.component.ts Declaro mi variable.

```
export class AppComponet {
        myVar= 'Hola Mundo'
}
```

Paso 4: Pruebo la prueba
Y ahora es ok. Funciona.

Paso 5: Si nosotros cambiamos el valor de la variable
y volvemos a pasar el test, ahora fallaría.

Y con esto hemos comprobado un caso de IGUALDAD, ES IGUAL A … toEqual

Tutoriales en la Documentación de ayuda

En la página de Jasmine en el apartado de Docs tenemos los siguientes tutoriales.

Tutorials

- Your first suite
- Async
- Custom asymmetric equality testers
- Custom boot
- Custom equality (aquí muestra el ToEqual del ej. anterior)
- Custom matcher
- Custom object formatters
- Custom reporter
- Default spy strategy
- Mocking ajax
- React with browser
- React with node
- Sharing behaviors
- Spying on properties
- Upgrading to jasmine 4.0
- Use without globals

Otro enlace recomendado

- https://testing-angular.com/