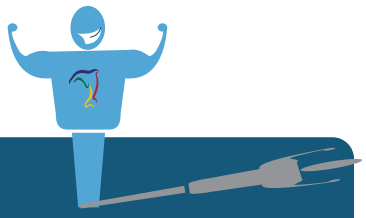
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

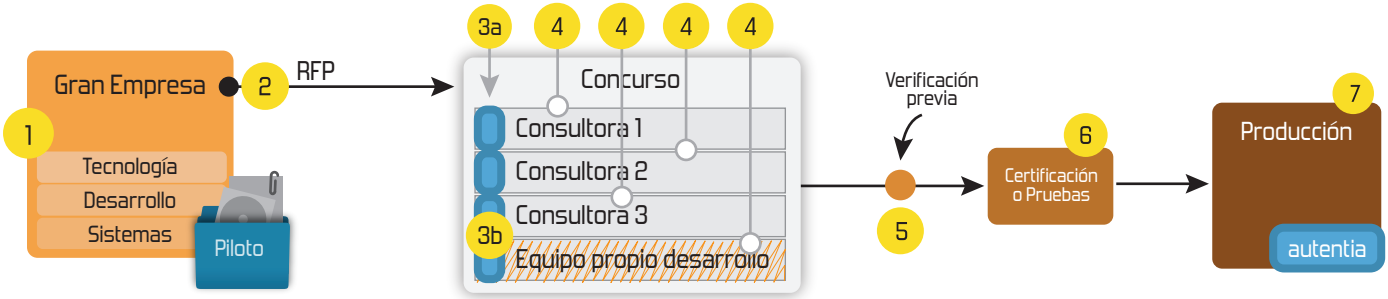
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

- Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery
- Gestor portales (Liferay)
- Gestor de contenidos (Alfresco)
- Aplicaciones híbridas
- Control de autenticación y acceso (Spring Security)
- UDDI
- Web Services
- Rest Services
- Social SSO
- SSO (Cas)
- JPA-Hibernate, MyBatis
- Motor de búsqueda empresarial (Solr)
- ETL (Talend)
- Dirección de Proyectos Informáticos.
- Metodologías ágiles
- Patrones de diseño
- TDD
- Tareas programadas (Quartz)
- Gestor documental (Alfresco)
- Inversión de control (Spring)
- BPM (jBPM o Bonita)
- Generación de informes (JasperReport)
- ESB (Open ESB)

Compartimos nuestro conociemiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Miguel Arlandy Rodríguez

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/JEE

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2014-12-09

AngularJS y los tests unitarios.

0. Índice de contenidos.

1. Introducción

No vamos a descubrir nada nuevo si ensalzamos las virtudes de los tests unitarios. Sin embargo, en lenguajes como Javascript corriendo en el lado del cliente adquieren incluso mayor importancia. Cuando se ejecuta código Javascript en un navegador de algún usuario de nuestro sistema, no tenemos forma de saber qué ha ocurrido si se produce algún error (salvo que lo controlemos y reportemos) como sí que pasaría si ese fallo se produjese en nuestro servidor, donde podremos ir a consultar las trazas de log.

El otro día tuve la oportunidad de asistir a un evento sobre aseguramiento de la calidad del software y dijeron una frase que me gustó bastante: ***La calidad es innegociable, forma parte inherente e inseparable del producto software***. Pues bien, podríamos considerar a los tests unitarios como los pilares de este proceso de aseguramiento de la calidad.

En este tutorial intentaremos explicar cómo realizar tests unitarios en AngularJS haciendo uso de Karma y Jasmine, jugaremos con mocks y aprenderemos a generar informes.

2. Entorno.

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 15' (2.2 Ghz Intel Core I7, 8GB DDR3).
- Sistema Operativo: Mac OS X Mavericks 10.9
- NetBeans IDE 8.0.1
- AngularJS 1.2.26
- Node Package Manager (npm) 2.1.8
- Google Chrome 39
- Mozilla Firefox 33

Últimas Noticias

» Curso JBoss de Red Hat

» Si eres el responsable o líder técnico, considérate desafortunado. No puedes culpar a nadie por ser gris

» Portales, gestores de contenidos documentales y desarrollos a medida

» Comentando el libro Start-up Nation, La historia del milagro económico de Israel, de Dan Senor & Salu Singer

» Screencasts de programación narrados en Español

Histórico de noticias

Últimos Tutoriales

» Introducción a Typescript

» Tutorial Apple Watch

» Analizando WSO2 Business Rules Server

» Desarrollo de aplicaciones móviles multiplataforma con

3. ¿Qué vamos a hacer?.

Desarrollaremos una pequeña aplicación que nos **muestre un listado de dos coches** y sus características: marca, modelo y motorización. Además, queremos que la marca del coche aparezca en **mayúsculas y con dos asteríscos** al principio.

Por tanto nuestros requisitos son:

- Debemos listar un conjunto de coches, en concreto dos.
- Debemos mostrar las características de los coches al usuario. La marca del coche debe aparecer en mayúsculas y con dos asteríscos al principio.

Para implementar nuestra solución haremos uso de **AngularJS** y nos apoyaremos en tres componentes:

- **Un filtro**: que se encargará de mostrar la marca del coche en el formato requerido.
- **Un servicio**: que se encargará de obtener la lista de coches.
- **Un controlador**: que se encargará de exponer la lista de coches.

Estamos buscando algo así:

Marca: **VOLKSWAGEN

Modelo: golf

Motor: 2.0 TDI 150CV

Marca: **FORD

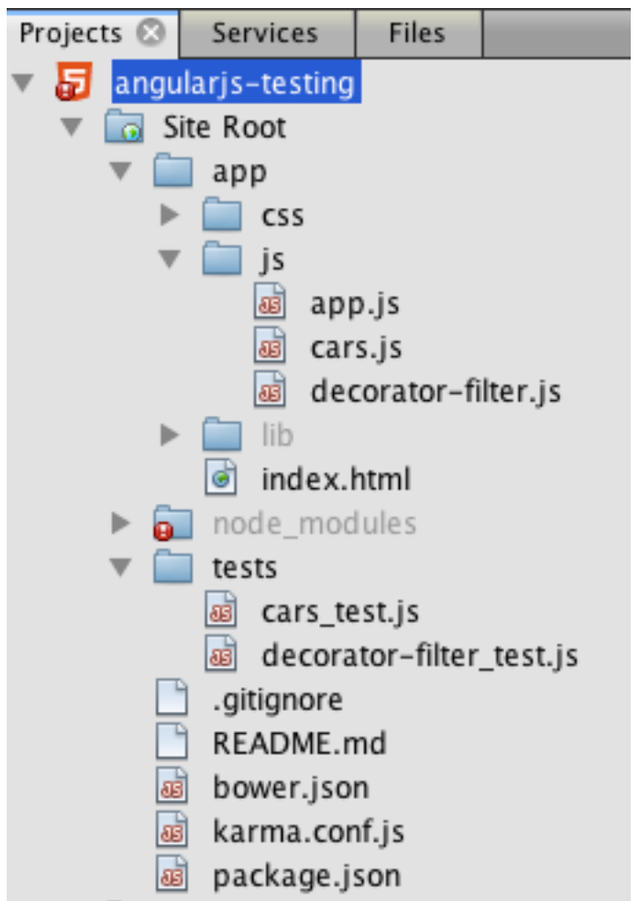
Modelo: focus

Motor: 1.8 TDCI 115CV

4. Configuración.

4.1 Estructura del proyecto.

La estructura de nuestro proyecto tendrá un aspecto similar al de la siguiente figura (el directorio "Site Root" no existe, lo añade NetBeans al mostrar el proyecto):



- Directorio **app/**: Contendrá nuestro código de producción. Para el que esté acostrumbrado a trabajar con Maven, sería lo equivalente al directorio main. Estará compuesto por:
  - **app/css/**: Contendrá las hojas de estilo de nuestro proyecto.
  - **app/js/**: Contendrá el código Javascript que escribiremos para implementar la lógica funcional del proyecto.
  - **app/lib/**: Contendrá las librerías de terceros. En este caso, las dependencias de AngularJS. Lo gestionaremos mediante Bower.
  - **app/index.html**: Página inicial de nuestra web.
- Directorio **node_modules/**: Módulos de node para la gestión del proyecto.
- Directorio **tests/**: Contendrá el código de nuestros tests unitarios.
- Fichero **bower.json**: Gestionará las dependencias de librerías externas de nuestro proyecto. En nuestro caso serán dependencias de AngularJS.
- Fichero **karma.conf.js**: Fichero de configuración del framework Karma, que será la herramienta en la que nos apoyemos para configurar el comportamiento de la ejecución nuestros tests unitarios.
- Fichero **package.json**: Fichero de configuración general de nuestro proyecto que será interpretado por el gestor de

Apache Cordova utilizando AngularJS, Ionic y ngCordova

» Paradigma publish/subscribe con Spring Data Redis

Últimos Tutoriales del Autor

» Jugando con AngularJS y Google Maps

» Phonegap/Cordova y las Notificaciones Push

» Configurando Notificaciones Push para desarrollos Android con Google Cloud Messaging.

» Introducción a WSO2 API Manager

» SOA vs. SOAP y REST

Categorías del Tutorial

Javascript / JQuery

paquetes de node (npm).

4.2 package.json

Nuestro fichero package.json tendrá un aspecto similar a este:

```
{
  "name": "angular-testing",
  "private": true,
  "version": "0.0.1",
  "description": "AngularJS testing project",
  "repository": "https://github.com/marlandy/angularjs-testing",
  "license": "MIT",
  "devDependencies": {
    "bower": "^1.3.1",
    "karma": "~0.10"
  },
  "scripts": {
    "postinstall": "bower install",
    "pretest": "npm install",
    "test": "karma start karma.conf.js"
  }
}
```

Además de los metadatos relativos al proyecto, podemos destacaremos dos cosas:

- Añadimos las **dependendencias** de **bower** y **karma**. El primero lo necesitaremos para gestionar las librerías de terceros que usará nuestro proyecto (AngularJS) y el segundo para la gestión de nuestros tests.
- Añadimos los diferentes **scripts** vinculados a diferentes fases del ciclo de vida de nuestro paquete. En concreto: instalamos las dependencias a nivel de paquete (npm install), instalamos las librerías de terceros que necesitará nuestro proyecto (bower install) y nos aseguramos de que los tests se lancen haciendo uso de karma. ¿Que te traduzca esto a cristiano? pues muy fácil, cuando lancemos el comando **npm test** tendremos todas nuestras dependencias instaladas correctamente y todo funcionará de maravilla :-).

4.3 karma.conf.js

Mediante el fichero karma.conf.js configuraremos el comportamiento que queremos te tenga la ejecución de nuestros tests. El fichero sería algo como esto:

```
module.exports = function (config) {
    config.set({
        basePath: './',
        files: [
            'app/lib/angular/angular.js',
            'app/lib/angular-mocks/angular-mocks.js',
            'app/js/**/*.js',
            'tests/**/*.js'
        ],
        autoWatch: false,
        frameworks: ['jasmine'],
        browsers: ['Chrome', 'Firefox'],
        singleRun: true,
        plugins: [
            'karma-chrome-launcher',
            'karma-firefox-launcher',
            'karma-jasmine'
        ]
    });
};
```

Destacamos lo siguiente:

- **files**: Listado de ficheros que tendrá en cuenta karma para la ejecución de nuestros tests. En concreto le indicamos las librerías de terceros (que están en el directorio app/lib/), nuestro código de producción (app/js/) y nuestros tests unitarios (tests/).
- **autoWatch**: Esta propiedad es muy interesante. Sirve para indicar si queremos que karma esté continuamente observando cambios en alguno de los ficheros y, en caso afirmativo, lance los tests. Le diremos que no ya que en este ejemplo los lancaremos "a demanda".
- **frameworks**: Karma es un framework extraordinario, pero tiene una característica en concreto que a mí me encanta. Es totalmente imparcial a la hora de trabajar con el framework de testing que perfiera el usuario (QUnit, Jasmine, etc...) para escribir los tests. Con esta propiedad le indicaremos dicho framework. En este caso haremos uso de Jasmine.
- **browsers**: Listado de navegadores en los que queremos que se lancen los tests. Recordemos que vamos a escribir código que se ejecuta en un cliente (navegador) y para comprobar su correcto funcionamiento debemos hacer uso alguno. Si eres "Javero" y esto te resulta algo extraño, debes comprender que aquí no tenemos una maravillosa máquina virtual de Java (JVM) instalada en nuestro equipo que nos ejecute nuestros tests y nos verifique su correcto funcionamiento. Esto es otro mundo...
- **singleRun**: Indicamos que, después de correr los tests, queremos que se termine el proceso.
- **plugins**: Listado de plugins necesarios para Karma. Añadimos el lanzador de Chrome, Firefox y el adaptador de Jasmine.

Pues con esto hemos terminado la configuración aunque nos hemos saltado la parte de Bower donde lo que hacemos es añadir las dependencias de AngularJS y Angular-Mocks (librería que nos ofrece el soporte de Angular al manejo de "mocks"). Si quieres saber más tienes todo el código fuente del ejemplo aquí.

5. Testeando el filtro.

Como dijimos anteriormente, necesitaremos un componente cuyo objetivo sea transformar cadenas de caracteres de forma que el resultado sea una cadena en mayúsculas con dos asteriscos al principio. Para tal propósito AngularJS nos proporciona los **filtros**.

Para ello crearemos un nuevo módulo llamado **app.decorator-filter**. En dicho módulo añadiremos nuestro filtro al que llamaremos **decorator**. Para ello crearemos dos ficheros: **app/js/decorator-filter.js** y **tests/decorator-filter_test.js**. En el primero añadiremos el código de producción y en el segundo el test.

Como no podía ser de otra manera, escribiremos primero nuestro test describiendo el comportamiento de nuestro componente (nuestro filtro), lo que esperamos de él en cada caso y después el código de producción (comprobaríamos que todo funciona y finalmente refactorizaríamos si hiciese falta). A esta técnica se la conoce como BDD.

Nuestro fichero **decorator-filter_test.js** quedaría así:

```
'use strict';

describe('Modulo app.decorator-filter', function () {

    beforeEach(function(){
        module('app.decorator-filter');
    });

    describe('Filtro decorator', function () {
        it('debe transformar en mayusculas cualquier string y anteponer asteriscos',
                inject(function (decoratorFilter) {
            var input = 'someThing';
            var expectedOutput = '**SOMETHING';
            expect(decoratorFilter(input)).toEqual(expectedOutput);
        }));
    });
});
```

Creo que el código habla por sí solo pero si te cuesta entender el anterior fragmento te recomiendo que le eches un ojo a este tutorial. Observemos que estamos inyectando el filtro mediante la convención: **nombre del filtro + "Filter"**. Lo hacemos de esta forma ya que es la manera en la que Angular registra los filtros en el objeto $injector.

Pasamos ahora a escribir nuestro código de producción (el filtro). El fichero **decorator-filter.js** quedaría:

```
'use strict';

(function () {
    var module = angular.module('app.decorator-filter', []);

    module.filter('decorator', function () {
        return function (text) {
            return '**' + String(text).toUpperCase();
        };
    });
})();
```

Pues este sería el código de nuestro filtro. Para lanzar los tests lo haremos con el siguiente comando:

```
npm test
```

MUY IMPORTANTE: en entornos **Unix** debemos contar con los **permisos** necesarios para instalar paquetes (npm install) y para las librerías de terceros que utilizará nuestro proyecto (bower install). Para entornos **Windows** te recomiendo el uso de alguna herramienta tipo Cygwin.

Y este es el resultado:

```
INFO [karma]: Karma v0.10.10 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 39.0.2171 (Mac OS X 10.9.5)]: Connected on socket FHFOUiNQBQWEH4YtCbz7
Chrome 39.0.2171 (Mac OS X 10.9.5): Executed 1 of 1 SUCCESS (0.137 secs / 0.016 secs)
```

6. Testeando el servicio.

Necesitaremos contar con un servicio que se encargue de devolver el listado de coches. Dicho servicio tendrá un método al que llamaremos **getAll** que nos devolverá los coches. El servicio formará parte un un módulo llamado **app.cars**. A dicho servicio le identificaremos con el nombre "CarsService".

Nuestro fichero **cars_test.js** quedaría así:

```
'use strict';

describe('Modulo app.cars', function () {

    beforeEach(function () {
        module('app.cars');
    });

    describe('Cars service', function () {

        var carsService;

        beforeEach(function () {
            inject(['CarsService', function (service) {
                    carsService = service;
                }
            ]);
        });

        it('debe devolver una lista de dos coches', function () {
            var cars = carsService.getAll();
            expect(cars).toBeDefined();
            expect(cars.length).toBe(2);
        });
    });
});
```

En este caso, a diferencia del test anterior, hemos inyectado la dependencia a nivel del juego de pruebas del servicio

(describe, dentro de beforeEach) y no en la propia especificación (it). Probablemente esta sería la forma más indicada cuando vamos a definir diferentes comportamientos (especificaciones) a un mismo componente.

Y nuestro código de producción con el servicio en el fichero **cars.js** sería algo como:

```
1  'use strict';
2
3  (function () {
4      var module = angular.module('app.cars', []);
5
6      module.factory('CarsService', function () {
7
8          var cars = [
9              createCar('volkswagen', 'golf', '2.0 TDI 150CV'),
10             createCar('ford', 'focus', '1.8 TDCI 115CV')
11         ];
12
13         function createCar(brand, model, engine) {
14             return {
15                 brand: brand,
16                 model: model,
17                 engine: engine
18             };
19         }
20
21         function getCars() {
22             return cars;
23         }
24
25         return {
26             getAll: getCars
27         };
28     });
29
30 })();
```

Lanzamos los tests:

```
1  npm test
```

Y comprobamos que todo está perfecto :)

```
INFO [karma]: Karma v0.10.10 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 39.0.2171 (Mac OS X 10.9.5)]: Connected on socket MD4GiSo-Qb8V33ppFG3l
Chrome 39.0.2171 (Mac OS X 10.9.5): Executed 2 of 2 SUCCESS (0.137 secs / 0.022 secs)
```

7. Testeando el controlador.

Bueno, pues esto ya está casi terminado. Lo último que nos quedará será nuestro controlador. Su misión es muy sencilla: haciendo uso del servicio de consulta de coches, deberá exponer el listado mediante un atributo de su $scope.

En este caso añadiremos una particularidad que no tenían los puntos anteriores. Con el fin de probar cada uno de los componentes de nuestro proyecto de manera independiente, **haremos uso de mocks** para hacer el test de este controlador puesto que hace uso de otro colaborador (servicio de consulta de coches).

Añadiremos el controlador al mismo módulo (app.cars) que contiene el servicio de consulta.

Nuestro test quedaría de la siguiente manera:

```
1  'use strict';
2
3  describe('Modulo app.cars', function () {
4
5      beforeEach(function () {
6          module('app.cars');
7      });
8
9      describe('Cars controller', function () {
10
11         var $scope, CarsService;
12
13         beforeEach(function () {
14             module(function ($provide) {
15                 // inyectamos del mock
16                 $provide.value('MockedCarsService', {'getAll': function () {
17                         return [];
18                     }});
19             });
20         });
21
22         beforeEach(inject(function ($rootScope) {
23             $scope = $rootScope.$new();
24         }));
25
26         it('debe exponer la lista de coches', inject(function ($controller, MockedCar
27             $controller('CarsController', {'$scope': $scope, 'CarsService': MockedCar
28             expect($controller).toBeDefined();
29             expect(CarsService);
30             expect($scope.cars.length).toBe(MockedCarsService.getAll().length);
31         }));
32     });
33
34 });
```

Prestemos especial atención a las siguientes líneas:

- **línea 16:** Estamos creando nuestro servicio mock (objeto de mentira que sustituirá al original) para que posteriormente pueda ser inyectado en el controlador.

- **línea 24:** Creamos un nuevo $scope que será utilizado por nuestro controlador a través del cual expondremos la lista de coches.
- **línea 27:** Inicializamos el controlador con el nuevo $scope y nuestro servicio de consulta de coches "mockeado". Como vemos, expondremos el listado de coches a través del **objeto "car"** del ámbito de nuestro controlador.

A continuación vemos cómo quedaría nuestro controlador:

```
1  'use strict';
2
3  (function () {
4      var module = angular.module('app.cars', []);
5
6      module.factory('CarsService', function () {
7          // punto anterior...
8      });
9
10     module.controller('CarsController', ['$scope', 'CarsService', function ($scope, C
11             $scope.cars = CarsService.getAll();
12         }]);
13 })();
```

Lanzamos los tests...

```
INFO [karma]: Karma v0.10.10 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 39.0.2171 (Mac OS X 10.9.5)]: Connected on socket gJP7ucEDLy4A6rz1_mLp
Chrome 39.0.2171 (Mac OS X 10.9.5): Executed 3 of 3 SUCCESS (0.148 secs / 0.027 secs)
```

Pues con esto ya estaría todo, únicamente nos quedaría dar forma a nuestra aplicación (vista) mediante el fichero **index.html**:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8">
5          <meta http-equiv="X-UA-Compatible" content="IE=edge">
6          <title>AngularJS - Testing</title>
7          <meta name="viewport" content="width=device-width, initial-scale=1">
8          <link href="css/default.css" rel="stylesheet">
9      </head>
10     <body ng-app="app">
11
12         <ul ng-controller="CarsController">
13             <li ng-repeat="car in cars">
14                 <div>
15                     <p>
16                         <span>Marca:</span> {{car.brand | decorator}}
17                     </p>
18                     <p>
19                         <span>Modelo:</span> {{car.model}}
20                     </p>
21                     <p>
22                         <span>Motor:</span> {{car.engine}}
23                     </p>
24                 </div>
25             </li>
26         </ul>
27
28         <script src="lib/angular/angular.js"></script>
29         <script src="js/cars.js"></script>
30         <script src="js/decorator-filter.js"></script>
31         <script src="js/app.js"></script>
32     </body>
33 </html>
```

AQUÍ TIENES TODO EL CÓDIGO FUENTE DEL EJEMPLO.

8. Generando informes: junit y cobertura.

Todo esto está muy bien, pero todavía podemos ir más lejos. Vamos a ver cómo podemos generar informes con la información relativa a la ejecución de nuestros tests gracias al soporte que nos da Karma.

Por defecto Karma nos proporciona un reporter al que llama **progress**, que no es más que la salida por consola que nos indica qué ha sucedido con nuestros tests. Añadiremos dos más:

- **junit**: Generará un fichero .xml en formato junit. Este informe está especialmente indicado cuando queremos que sea consumido por otras herramientas como **Jenkins o Bamboo**.
- **coverage**: Generaremos informes visuales (html) mostrando el porcentaje de código cubierto por nuestros tests. Además generaremos un fichero **lcov** especialmente indicado si queremos exportar nuestro informe a alguna herramienta tipo **Sonar**.

Para ello hacemos unos ligeros ajustes en nuestro fichero **package.json**:

```
1  "devDependencies": {
2    "bower": "^1.3.1",
3    "karma": "~0.10",
4    "karma-junit-reporter": "^0.2.2",
5    "karma-coverage": "~0.1"
6  },
7  "scripts": {
8    "postinstall": "bower install",
9    "pretest": "npm install",
10   "test": "karma start karma.conf.js --reporters progress,junit,coverage"
11 }
```

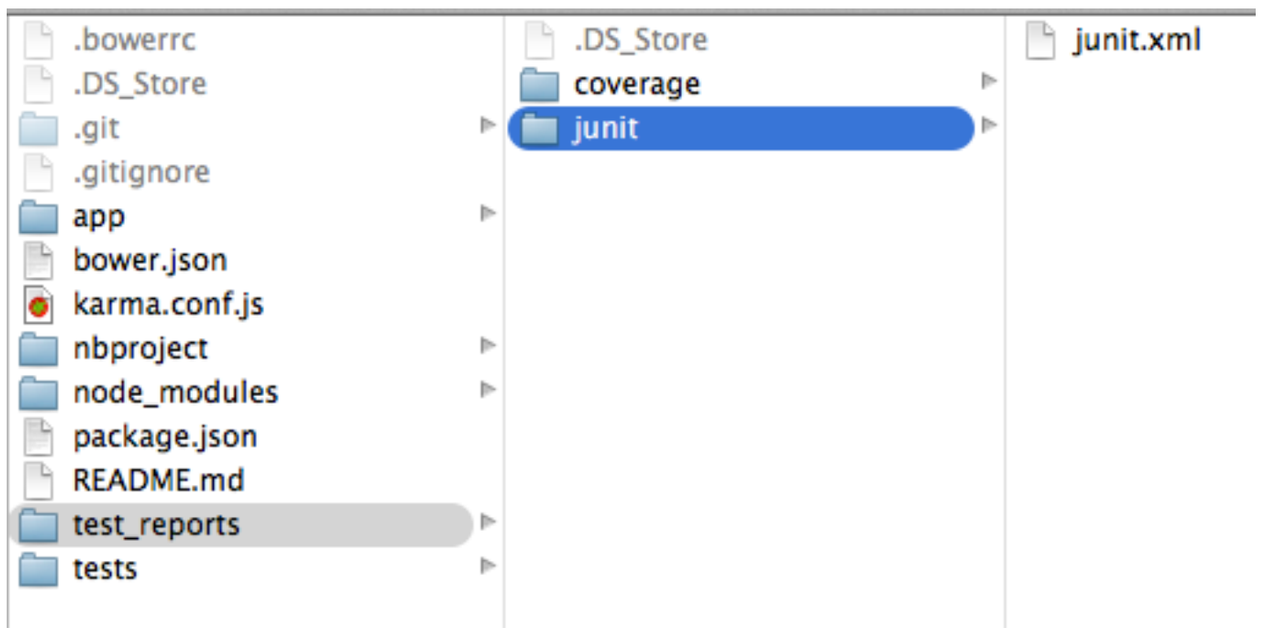
Y retocamos también nuestro **karma.conf.js**:

```
1  module.exports = function (config) {
2      config.set({
3          basePath: './',
```

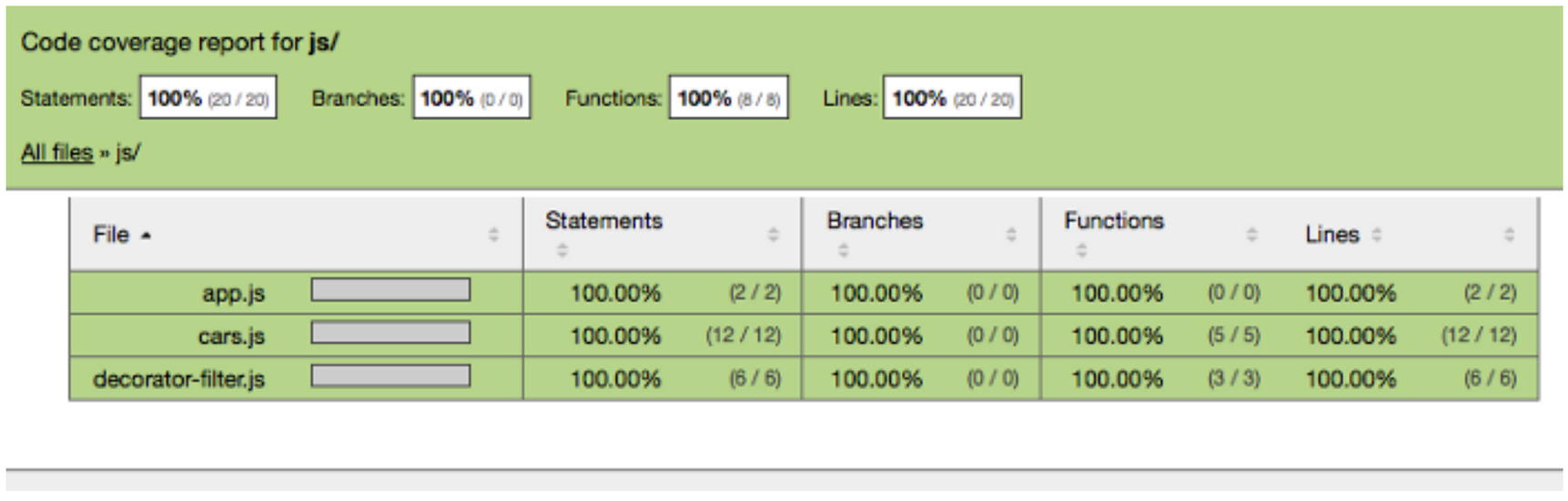
```
        reporters: ['junit', 'coverage'],
        files: [
            'app/lib/angular/angular.js',
            'app/lib/angular-mocks/angular-mocks.js',
            'app/js/**/*.js',
            'tests/**/*.js'
        ],
        autoWatch: true,
        frameworks: ['jasmine'],
        browsers: ['Chrome'],
        singleRun: true,
        plugins: [
            'karma-chrome-launcher',
            'karma-firefox-launcher',
            'karma-jasmine',
            'karma-junit-reporter',
            'karma-coverage'
        ],
        junitReporter: {
            outputFile: 'test_reports/junit/junit.xml',
            suite: 'unit'
        },
        preprocessors: {
            'app/js/**/*.js': ['coverage']
        },
        coverageReporter: {
            dir: 'test_reports/coverage/',
            reporters: [
                {type: 'lcov', subdir: '.'},
                {type: 'cobertura', subdir: '.', file: 'cobertura.xml'}
            ]
        }

    });
};
```

Ahora, cuando lancemos nuestros tests observaremos que se nos creará un directorio **test_reports** con dos subdirectorios: **junit** y **coverage** con los respectivos informes.



Y podremos ver gráficamente la cobertura de nuestro código.



9. Referencias.

- Código fuente del ejemplo
- AngularJS unit testing
- Karma: documentación oficial
- AngularJS: primeros pasos.
- Introducción a Jasmine

10. Conclusiones.

AngularJS es un excelente framework para desarrollo de aplicaciones Javascript que corren en el lado del cliente. El verdadero potencial de este framework reside en las directrices y herramientas que proporciona para **diseñar** aplicaciones altamente mantenibles.

Angular nos permite separar la lógica de negocio (o lógica de presentación) de la propia vista. Nos proporciona múltiples mecanismos para separar esa lógica en diferentes unidades funcionales como son los servicios, filtros, directivas, controladores, etc... y que podamos aislarlos y testearlos de manera independiente: principio de **alta cohesión y bajo acoplamiento**.

Cualquier duda en la sección de comentarios.

Espero que este tutorial os haya sido de ayuda. Un saludo.

Miguel Arlandy

marlandy@autentia.com

Twitter: @m_arlandy

A continuación puedes evaluarlo:

Regístrate para evaluarlo

Por favor, vota +1 o compártelo si te pareció interesante

Anímate y coméntanos lo que pienses sobre este **TUTORIAL:**

» **Registrate** y accede a esta y otras ventajas «