# Tests unitarios en AngularJS

Juan Ferrer
Álvaro de la Serna

# Introducción

Empecemos definiendo qué es un test unitario: un test unitario es una forma de comprobar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

La idea es escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto. Para que un test unitario tenga la calidad suficiente debe ser:

- **Automatizable**
- **Completo**
- **Repetible**
- **Independiente**
- **Profesional**

## Ventajas de hacer pruebas unitarias

El objetivo de las pruebas unitarias es aislar cada parte del programa y mostrar que las partes individuales son correctas. Proporcionan un contrato escrito que el trozo de código debe satisfacer. Estas pruebas aisladas proporcionan cinco ventajas básicas:

1. **Fomentan el cambio**

   Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura (lo que se ha dado en llamar refactorización), puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.

2. **Simplifica la integración**

   Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente. De esta manera se facilitan las pruebas de integración.

3. **Documenta el código**

   Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.

4. **Separación de la interfaz y la implementación**

   Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro, a veces usando objetos mock (mock object) para simular el comportamiento de objetos complejos.

5. **Los errores están más acotados y son más fáciles de localizar**

   Dado que tenemos pruebas unitarias que pueden desenmascararlos.

# Tests unitarios en AngularJS

Para poder realizar tests unitarios en Angular, lo primero que necesitamos es un *test runner* con el que correr los tests. La mejor opción es Karma, con el que podremos correr los tests en distintos navegadores y ver la compatibilidad de nuestro código con ellos o hacer que corran en un PhantomJS.
También necesitaremos un framework de testing BDD (Behavior Driven Development) para JavaScript: Jasmine.
Para ver la cobertura de código podemos usar Istanbul, que generará un informe con las líneas de código cubiertas y porcentajes de cobertura de nuestros tests.

Para hacer independiente el código que vamos a testear, por ejemplo una función, necesitamos sustituir todas las funciones de las que dependa por *fakes*. Según del tipo de dependencia podremos espiar o crear nuestro propio espía. Con esta última y *$provide* de Angular podremos crear *mocks* inyectables como proveedores. Uso:

- **Espías**: se crean con la función *spyOn()* de Jasmine. Se usan para hacer un *fake* de una función de una clase y sólo existen dentro de los bloques *describe* e *it* que los contengan.
- **Crear espías**: se crea con *jasmine.createSpy()* o *jasmine.createSpyObj()*. Se usan cuando no podemos espiar una función. Con *jasmine.createSpyObj()* podemos crear objetos de espías.
- **Mocks**: usando *module()* e inyectando *$provide* podremos sustituir cualquier dependencia por un mock creado con *jasmine.createSpy()*. Lo usaremos cuando no podamos espiar desde el test, por ejemplo cuando tengamos en el código constructores con instancias de clases u objetos inicializadas con ***new***.

Todos las funciones espiadas o espías propios tienen disponible funcionalidades con las que podremos saber si los espías se han ejecutado o no, sustituir los valores que devuelve la función, ejecutar funciones para sobreescribir el comportamiento original, etc. En definitiva, todo lo que necesitemos para hacer correctamente nuestros tests.

## Ejemplo de declaración de tests

Con todo esto ya podemos hacer nuestros tests. Una posible estructura a seguir sería la siguiente:

```
describe('Class description', function () {
        beforeEach(function () {
                // before each block
        });
        afterEach(function () {
                // after each block
        });
        describe('Method description', function () {
                beforeEach(function () {
```

```
                        // before each block
                });
                afterEach(function () {
                        // after each block
                });
                it('should description', function () {
                        // test 1
                });
                it('should description', function () {
                        // test 2
                });
        });
});
```

# Caso práctico

Vamos a empezar viendo un ejemplo sencillo. Teniendo el siguiente controlador:

```
'use strict';
angular
        .module('moduleOne')
        .controller('controllerOne',[
                '$scope', 'serviceOne',
                function ($scope, serviceOne) {
                        $scope.a = 0;

                        $scope.doSomething = function (number) {
                                $scope.a += number;
                                serviceOne.fn();
                                $scope.log($scope.a);
                        };

                        $scope.log = function (something) {
                                console.log('log something: ' + something)
                        };
                }
        ]);
```

Podríamos declarar nuestros tests de la siguiente forma:

```
'use strict';

describe('controllerOne', function () {
        var $scope, controllerOne, serviceOne;

        beforeEach(function () {
                module('moduleOne'); // Importar módulo

                inject(function ($rootScope, _serviceOne_) { // Para injectar cualquier dependencia
                        $scope = $rootScope.$new();
                        serviceOne = _serviceOne_;

                        // Instancia del controlador
                        controllerOne = $controller('controllerOne', {
                                $scope: $scope,
                                serviceOne: serviceOne
                        });
                });
        });
```

```
describe('$scope.doSomething', function () {
        it('should do something', function () {
                $scope.a = 0;
                spyOn($scope, 'log');
                spyOn(serviceOne, 'fn');

                $scope.doSomething(2);

                expect($scope.a).toBe(2);
                expect(serviceOne.fn).toHaveBeenCalled();
                expect($scope.log).toHaveBeenCalledWith(2);
        });
});
});
```

Lo primero es crear un bloque general donde se encapsulará toda la lógica de nuestros tests para este controlador. Esto lo hacemos con la función ***describe***, que recibe dos parámetros:

- El primero es un literal que describe lo que vamos a testear
- El segundo un callback que ejecutará.

Después creamos un bloque ***beforeEach***, que ejecutará un callback al inicio de cada bloque definido dentro del *describe* padre. En el callback del *beforeEach* del describe del controlador importaremos el módulo donde se encuentra nuestro controlador, inyectaremos sus dependencias y crearemos una instancia de él.

Seguido al bloque *beforeEach*, generamos otro bloque *describe* que describirá una función (la que queremos testear). Por cada función podemos crear un bloque *describe* que encapsule todos sus tests. Dentro del *describe* de la función a testar, llamaremos a las funciones de bloque *it*. En éstas será donde hagamos los tests. Cada *it* debe probar un caso concreto e independiente del resto.

Para hacer un buen test, como hemos dicho, debe ser independiente de los demás. Esto es, cualquier dependencia que tenga la función, ya sea otra función del mismo controlador, una variable de ámbito mayor, una función pasada por DI (Dependency Injection), etc. En nuestro caso, la función ***log*** y ***fn*** debe ser espiada con *spyOn* y la variable ***a*** inicializada.
Por cada test hay que llamar a la función a testar después de *mockear* sus dependencias y antes de las comprobaciones *expect*. Estas últimas sirven para comprobar que la función ha hecho lo que esperamos.

## Testeando promesas

Ahora veremos cómo hacer **un test de promesas**.
Basándonos en el controlador anterior, ahora la función *fn* del servicio *serviceOne* devuelve una promesa:

```
$scope.doSomething = function (number) {
        $scope.a += number;
```

```
        serviceOne.fn($scope.a).then(
                function () {
                        $scope.log($scope.a);
                },
                function () {
                }
        );
};
```

Su test correspondiente se podría programar de cualquiera de las siguientes formas:

## Usando la función 'callFake'

```
describe('$scope.doSomething', function () {
        it('should do something', inject(function ($q) {
                $scope.a = 0;
                spyOn($scope, 'log');
                spyOn(serviceOne, 'fn').and.callFake(function () {
                        var deferred = $q.defer();
                        deferred.resolve();
                        return deferred.promise;
                });

                $scope.doSomething(2);
                $scope.$digest();

                expect($scope.a).toBe(2);
                expect(serviceOne.fn).toHaveBeenCalled();
                expect($scope.log).toHaveBeenCalledWith(2);
        }));
});
```

Para que la promesa sea resuelta hay que llamar a la función *$digest()* de $scope. Esto fuerza un ciclo de Angular. De no llamarse, la promesa nunca se resolvería y la función *log* no se llamaría, haciendo que nuestro test fallase. Para resolver la dependencia de la función *fn* con un promesa, hay que crear un fake con la función ***callFake()*** de Jasmine, implementarla y retornarla.

## Usando la función 'returnValue'

Los espías de Angular también nos permiten retornar un valor (de la misma manera que lo hacíamos llamando a *callFake*) llamando a *returnValue*. Como ejemplo de uso del módulo *beforeEach*, declaramos las llamadas a las inicializaciones de las variables de entorno y los espías de las funciones antes de la declaración del test (módulo *it*) de la siguiente forma:

```
describe('$scope.doSomething using beforeEach()', function () {
        var deferred;
        beforeEach(function () {
                deferred = $q.defer();
                $scope.a = 0;
                spyOn($scope, 'log');
                spyOn(serviceOne, 'fn').and.returnValue(deferred.promise);
        });
        it('should do something', inject(function ($q) {
                deferred.resolve();
                $scope.doSomething(2);
```

```
                $scope.$digest();

                expect($scope.a).toBe(2);
                expect(serviceOne.fn).toHaveBeenCalled();
                expect($scope.log).toHaveBeenCalledWith(2);
        }));
});
```

De igual forma, para que la promesa sea resuelta hay que llamar a la función *$digest()* de $scope.

### Testeando una promesa fallida:

En cualquiera de los dos casos, si quisiéramos testar el caso en que la promesa falla, haríamos una llamada a la función *reject()* que lanza un error en la promesa, y veríamos cómo la función *log* no se llamaría, al haber forzado un fallo en la promesa.

```
describe('$scope.doSomething using beforeEach()', function () {
        var deferred;
        beforeEach(function () {
                deferred = $q.defer();
                $scope.a = 0;
                spyOn($scope, 'log');
                spyOn(serviceOne, 'fn').and.returnValue(deferred.promise);
        });
        it('should NOT do something', inject(function ($q) {
                // aquí estamos forzando un fallo en la promesa
                deferred.reject();
                $scope.doSomething(2);

                $scope.$digest();

                expect($scope.a).toBe(2);
                // aquí comprobamos que se ha llamado a serviceOne.fn()...
                expect(serviceOne.fn).toHaveBeenCalled();
                // … pero que no se ha llamado a la función log()
                expect($scope.log).not.toHaveBeenCalled();
        }));
});
```

## Testeando llamadas a servicios: $httpBackend

Ahora veamos un ejemplo con llamadas ***$http***.
Basándonos en el controlador anterior, vamos a implementar una función nueva que haga una llamada con $http:

```
$scope.getFrom = function (url) {
        $http({
                method: 'GET',
                url: url
        })
        .success(function (response) {
          $scope.log(response);
        })
```

```
        .error(function (error) {
          $scope.log(error);
        });
};
```

En este caso, el test unitario asociado a esta función tendría esta estructura:

```
describe('$scope.getFrom', function () {
        it('should get something from url', inject(function ($httpBackend) {
                var url = 'http://google.es', response = 'Success';
                spyOn($scope, 'log');
                $httpBackend.expectGET(url).respond(200, response);

                $scope.getFrom(url);

                expect($scope.log).toHaveBeenCalledWith(response);

                $httpBackend.flush();

                $httpBackend.verifyNoOutstandingExpectation();
                $httpBackend.verifyNoOutstandingRequest();
        }));
});
```

Para *mockear* llamadas con *$http*, Angular nos proporciona un módulo llamado ***$httpBackend***. Con él podremos capturar las llamadas de nuestro código y devolver una respuesta. Si queremos que espere la llamada y la intercepte usaremos el método *expect* y sus variantes. Si solo deseamos que la intercepte y que no valide si se efectúa o no la llamada usaremos *when* o alguno de sus variantes. Para limpiar las peticiones pendientes usaremos la función *flush()* de *$httpBackend*. La función *verifyNoOutstandingExpectation()* comprueba que se hicieron todas las peticiones definidas a través del API expect de *$httpBackend*. Si alguna de las solicitudes no se hizo, *verifyNoOutstandingExpectation* lanza una excepción. La función *verifyNoOutstandingRequest()* verifica que no hay peticiones pendientes que deban ser limpiadas. Estas dos últimas funciones suelen usarse dentro de un bloque *afterEach*. En este caso no sería necesario declarar el bloque *afterEach* ya que sólo hemos programado un test unitario.

## Testeando $interval y $timeout

Si usamos el módulo ***$interval*** o ***$timeout*** en nuestro código, también habrá que hacer sus respectivos test. Siguiendo con el ejemplo, si tenemos una función que implementa código con *$interval*:

```
$scope.otherFunction = function () {
        var stop = $interval(function () {
                $scope.log($scope.a);
                if ($scope.a >= 5) {
                        $interval.cancel(stop);
                }
        }, 1000);
};
```

Su test se hará de la siguiente forma:

```
describe('$scope.otherFunction', function () {
        it('should do other thing', inject(function ($interval) {
                $scope.a = 5;
                spyOn($scope, 'log');

                $scope.otherFunction();
                $interval.flush(1000);

                expect($scope.log).toHaveBeenCalledWith(5);
                expect($scope.log.calls.count()).toBe(1);
        }));
});
```

Para hacer que la función asociada a un *$interval* se ejecute tendremos que avanzar en milisegundos con la función ***flush()***, pasándole como parámetro los milisegundos. Para *$timeout* sería igual pero sin pasarle ningún parámetro a *flush()*.

## Testeando instancias de una clase

Otra casuística que nos podemos encontrar a la hora de hacer nuestros tests, será en la que tengamos una **instancia de una clase**. Para este caso no nos vale con usar *spyOn* en el método, ya que cuando se ejecute el test contra el código la instancia será nueva y perderemos el espía. Para estos casos es necesario **sobreescribir el proveedor**.

Veamos el ejemplo:

```
$scope.doSomething = function (something) {
        var one = new classOne();
        one.setSomething(something);
};
```

Su test se haría de la siguiente forma:

```
module('moduleOne', function ($provide) {
        // Sustituimos el proveedor por uno fake
        $provide.value('classOne', jasmine.createSpy('classOne').and.returnValue(
           jasmine.createSpyObj('classOne', ['setSomething'])
        ));
});
describe('$scope.doSomething', function () {
        it('should do something', inject(function (classOne) {
                var something = 'something'

                $scope.doSomething(something);

                expect(classOne.setSomething).toHaveBeenCalledWith(something);
        }));
});
```

Ahora todas las instancias que tengamos en el código harán referencia a nuestro *mock*, de forma que podemos testearlas.

# Testeando servicios

Para hacer **pruebas unitarias de servicios** solo tenemos que inyectar la dependencia del servicio y podremos acceder a sus funciones internas. Por ejemplo, dado el servicio:

```
"use strict";

angular.module('moduleOne').service("serviceOne", [
  function () {
    return {
      doSomething: function () {
        // do something
      }
    }
  }
]);
```

Su test sería:

```
"use strict";

describe("serviceOne", function () {
  var serviceOne;

  beforeEach(function () {
    module("moduleOne");

    inject(function (_serviceOne_) {
      serviceOne = _serviceOne_;
    });
  });

  describe('doSomething', function () {

    it('should do something', function () {
      // spies

      serviceOne.doSomething();

      // expects
    });
  });
});
```

# Testeando directivas

Otro aspecto importante de los tests en Angular son las **directivas**. Para poder testear una directiva tenemos que compilarla previamente. Además, si en el *template* de ésta llamamos a otras directivas tendremos que sustituirlas por un mock para poder independizar nuestros tests correctamente.
Veamos un ejemplo:

```
'use strict';

angular.module('moduleOne').directive("directiveOne", [
  function () {

    return {
      restrict: 'EA',
```

```
        template: 'ParamOne: {{paramOne}}. <directive-two></directive-two>',
        scope: {
          paramOne: '=',
          functionOne: '&'
        },
        link: function (scope) {

          scope.doSomething = function () {

            scope.functionOne();
            scope.log(scope.paramOne);
          };

          scope.log = function (message) {
            console.log(message);
          }
        }
      };
    }
]);
```

El test de esta directiva tendría esta pinta:

```
'use strict';

describe('doSomething', function () {
  var scope, compiledElement, element, $compile, $rootScope;

  beforeEach(function () {
    module('moduleOne', function ($provide, $compileProvider) {

      // Fake de la directiva directiveTwo, la cual se llama en el template
      $compileProvider.directive('directiveTwo', function () {
        return {
          priority: 100,
          terminal: true,
          restrict: 'EA',
          template: '<div class="directive-two-fake"></div>'
        };
      });
    });

    inject(function (_$rootScope_, _$compile_) {
      $rootScope = _$rootScope_;
      $compile = _$compile_;

      // Se crea un scope hijo de $rootScope
      scope = $rootScope.$new();

      scope.paramOne = 'paramOne';
      scope.functionOne = function () {};

      // Se crea la directiva
      element = angular.element('<directive-one paramOne="paramOne"
functionOne="functionOne()"></directive-one>');

      // Se compila la directiva
      compiledElement = $compile(element)(scope);

      // Refrescamos el ciclo de Angular
      scope.$digest();
    })
  });

  describe('scope.doSomething', function () {
```

```
    it('should do something', function () {
      var isolatedScope = compiledElement.isolateScope(); // Con este método accedemos al scope isolado de la directiva
      isolatedScope.foo = 0;

      spyOn(isolatedScope, 'functionOne');
      spyOn(isolatedScope, 'log');

      isolatedScope.doSomething();

      expect(isolatedScope.foo).toBe(1);
      expect(isolatedScope.functionOne).toHaveBeenCalled();
      expect(isolatedScope.log).toHaveBeenCalledWith(isolatedScope.paramOne);
    });
  });
});
```

Vemos que para aislar la directiva completamente necesitamos sustituir la dependencia de otra directiva, la cual es llamada desde el template. Para ello, en el callback de la importación del módulo usamos *$compileProvider.directive()*, reescribiendo la directiva existente *directiveTwo*. Ahora, cuando el test compile la directiva, será reemplazada por la que acabamos de crear.
Para compilar la directiva necesitamos inyectar el módulo *$compile*. Éste recibe como parámetro un elemento generado con *angular.element*, que contiene la llamada HTML de la directiva a compilar, y devuelve un constructor al que pasamos como parámetro un *scope* y que devuelve a su vez el elemento compilado. Una vez hecho esto, tendremos disponible el método *isolateScope()* con el que podremos acceder al *scope* isolado de la directiva y hacer tests de las funciones de ésta.

Si queremos, podemos testar el código HTML de la directiva:

```
expect(element.find('directive-two').length).toBe(1);
expect(element.find('div[class="[directive-two-fake]"]').length).toBe(1);
```

Y según vaya cambiando podemos hacer distintos tests para validar su correcto renderizado.

# Referencias

https://docs.angularjs.org/guide/unit-testing
https://karma-runner.github.io/0.8/index.html
http://phantomjs.org/
http://jasmine.github.io/
https://gotwarlost.github.io/istanbul/
https://docs.angularjs.org/guide/module
https://docs.angularjs.org/api/auto/service/$provide
https://docs.angularjs.org/api/ngMock/service/$httpBackend
https://docs.angularjs.org/api/ngMock/service/$interval
https://docs.angularjs.org/api/ngMock/service/$timeout