

Ampliando las vistas

lunes, 7 de agosto de 2017 21:34

- Creación de directivas
- Creación de componentes
- Creación de filtros



Directivas propias

método **directive()** del objeto *module* correspondiente a la aplicación

primer parámetro:
nombre de la directiva

Debe estar en camelCase, que
Angular reorganizara
insertando guiones, como hace
con sus propias directivas

segundo parámetro:
una función que devuelve
un objeto "configuración"

```
.directive("miDirectiva", function () {  
    return {  
        restrict: "AE"  
        template: {...}  
    }  
})
```





Directivas: configuración

objeto "configuración" → el objeto devuelto por el segundo parámetro

Propiedades (API)

- template / templateUrl
- replace (deprecated)
- restrict
- scope
- link
- require
- controller

template → la expresión que generara la directiva
templateURL → el fichero en el que se almacena esta expresión

replace → el elemento que invoca la directiva se mantiene o es sustituido por la salida de esta





Directivas: configuración

restrict → los modos de invocación permitidos

A
E <color-list></color-list>
C
M <!-- directive: color-list -->

require → parámetros referencia a otras directivas

scope → posible scope de la directiva

link → la función "link" de la directiva

controller → posible controller de la directiva



Usuario: directiva propia

Los datos de usuario de muestran mediante el empleo de una **directiva propia**.

Los siguientes datos sólo se muestran con el usuario inicial, como ejemplo de las directivas *ngChange* y *ngHide*

Datos de un Usuario

Nombre	<input type="text" value="Pepe"/>
Apellido	<input type="text" value="Pérez"/>
<input type="button" value="Borrar nombre"/>	

Usuario
Nombre: PEPE
Apellido: PEREZ

Los datos que se muestran a continuación sólo están disponibles para el usuario inicial

Fecha de alta
Mar 23, 2010
Consumo de datos
123.66

Alejandro L. Cerezo.
CLE Formación
Madrid - 2015





scope en las directivas

En el objeto configuración es posible definir el *scope* de la directiva

por defecto la directiva utiliza el *scope* "padre"

scope: true → la directiva utiliza un *scope* "hijo" creado por herencia prototípica a partir del padre

scope: {} → la directiva utiliza un *scope* "aislado"
(isolated) completamente independiente de cualquier otro

@: vinculación en un solo sentido,
=: vinculación en ambos sentidos,
&: vinculación de métodos



Directivas: función "link"



permite hacer operaciones relacionadas con el *scope* desde la directiva con objeto de construir expresiones más complejas en la salida

- modificar los valores del modelo
- comprobar (*watch*) sus posibles cambios
- añadir manejadores de eventos para los elementos del DOM
- realizar cualquier manipulación del DOM

link: function(scope, elem, attrs) {...}

scope → define el objeto *scope* de la directiva.
Por defecto, el *scope* padre

elem → objeto jQLite (o JQuery) del elemento DOM
en el que se utiliza la directiva

attrs → conjunto de atributos HTML de dicho elemento





Directivas: controller

permite declarar un controlador a nivel de la directiva.

código del controlador	se ejecuta antes de la compilación inicialización o manipulación de las propiedades del ámbito de la directiva
	No se debería incluir manipulación de árbol DOM
código del link	se ejecuta después de la compilación manipulación de árbol DOM



Directiva propia

Directiva propia

I love AngularJS

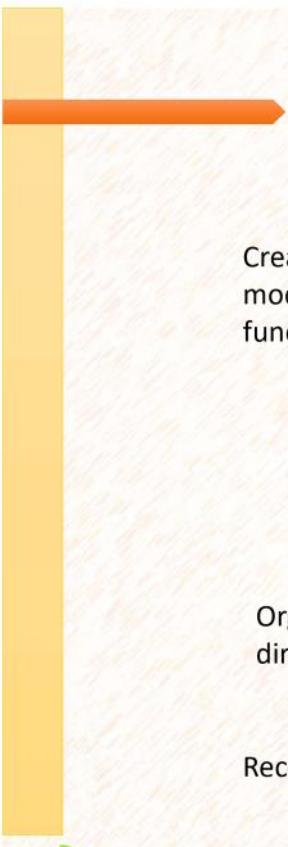
Hello, World! I love AngularJS

Alejandro L. Cerezo - Madrid 2015

La directiva permite:

- Recoger el texto escrito
- Desaparece al hacer clic sobre ella
- Cambia el aspecto del cursor para que se sepa que se puede hacer clic





Práctica

Crear una directiva propia para añadirla en un campo input de modo que al pulsar la tecla *enter* (ASCII 13) se ejecute la función indicada en el atributo

```
<input id="id" type="text"  
      data-ng-model="vm.Usuario.id"  
      data-dp-enter="vm.mostrarUsuario()"
```

Organizar el código de acuerdo con la guía de estilo y almacenar la directiva en un módulo completamente independiente

Recordar scope.\$apply(function
 scope.\$eval(expresión) -> ejecuta una expresión



Directivas propias: otros ejemplos

Solo permite introducir caracteres numéricos en un input

```
<input name="numero" type="text"  
       data-ng-model="numero"  
       data-numbers-only="true">
```

```
(function() {  
  'use strict';  
  
  angular.module('telNumbersOnly', []);  
  
  angular  
    .module('telNumbersOnly')  
    .directive('telNumbersOnly', numbersOnly);  
  
  function numbersOnly() {  
    var directive = {  
      link: link,  
      require: 'ngModel'  
    };  
  
    return directive;  
  
    function link(scope, element, attrs, modelCtrl) {  
      modelCtrl.$parsers.push(function (inputValue) {  
        // this next if is necessary for when using ng-required on your input.  
        // In such cases, when a letter is typed first, this parser will be called  
        // again, and the 2nd time, the value will be undefined  
        if (inputValue === undefined) {  
          return ''  
        }  
  
        if(attrs.telNumbersOnly === 'true') {  
          return inputValue;  
        }  
  
        var transformedInput = inputValue.replace(/\D/g, '');  
        if (transformedInput !== inputValue) {  
          modelCtrl.$setViewValue(transformedInput);  
          modelCtrl.$render();  
        }  
  
        return transformedInput;  
      });  
    }  
  }());
```

Recordar la directiva propia <div animated-view></div> en lugar de ngView en el ejemplo inicial de Dan Wahlin



Directivas y compilación



Además de la función ***link*** es posible definir la función ***compile***.

permite hacer transformaciones en el DOM antes de que se ejecute la función ***link***

sólo tiene dos parámetros

tElem	objeto jQLite (o JQuery) del elemento DOM en el que se utiliza la directiva, pero en la fase de plantilla, cuando aún no existe el scope
attrs	conjunto de atributos HTML de dicho elemento





Componentes

Angular 1.5 / 1.6

.directives() → .components()

Component-Based AngularJS Directives

[https://www.airpair.com/angularjs/posts/
component-based-angularjs-directives](https://www.airpair.com/angularjs/posts/component-based-angularjs-directives)

Anticipa Angular 2.0 y el nuevo enfoque
de los *controllers* ¿su desaparición?

relación con el nuevo estándar
Web Components



Futuro de HTML: Web Components

Web Components

conjunto de **estándares** que, permiten crear y utilizar elementos HTML personalizados.

Se puede así ampliar el “vocabulario” de HTML con elementos propios

En ellos está trabajando la W3C. Ya se soportan en algunos navegadores (Chrome) y esta disponible un *polyfile* para que puedan usarse en otros.

Constan de cuatro especificaciones:

- **Custom elements:** Nos permite definir nuevos elementos HTML.
- **Templates:** Sistema de plantillas nativas en el navegador.
- **Shadow DOM:** DOM scope.
- **HTML Imports:** Carga de documentos HTML.



Continuando con Web Components

<http://webcomponents.org/>

WebComponents.org
a place to discuss and evolve web component best-practices

polymer

Get Started Guides & Resources Element Catalog Blog v. 1.0

Production ready



Polymer is a framework for building reusable components for web pages. It makes it easy to reuse code across multiple pages, making them more modular and easier to maintain. If you haven't used Polymer before, it's time to try it out. If you haven't tried it recently, time to take another look.

[GET POLYMER](https://www.polymer-project.org/1.0/) [VIEW ON GITHUB](#)



Elementos de AngularJS 1.5



I'm Todd, I teach the world Angular through [@UltimateAngular](#).
Conference speaker and Developer Expert at Google.

[Follow @toddmotto](#) • 32.8K followers

POSTS

ABOUT

SPEAKING

A TRAINING

Exploring the Angular 1.5 .component() method

<https://toddmotto.com/exploring-the-angular-1-5-component-method/>





Elementos de AngularJS 1.5

Componentes

- template + controller
- controllerAs
- (por defecto \$ctrl)
- uso de clases
- constructores:
- inyección de dependencias
- ciclo de vida del componente (onInit)
- comunicación entre componentes:
 - parent
 - binding

```
1 /** ...
6 class SampleController {
7
8   /** ...
14   constructor($scope) {
15     'ngInject';
16     this.$scope = $scope;
17   }
18
19   /** ...
23   $OnInit () []
24
25     this.name = "Sample"
26     this.value = parent.value
27
28   // Fin del $OnInit
29
30 } // Fin del controller SampleController
31
32
33 angular.module('moduleName')
34
35 /** ...
40 */
41 .component("sample", {
42   require: {parent : '^appMain'},
43   templateUrl : "components/sample.html",
44   // usa controller as por defecto
45   controller: SampleController,
46   //controllerAs: '$ctrl', valor por defecto
47   bindings: {}
48 })
49 //Fin del componente y del objeto que lo define
```





Directivas v. Componentes

La directiva sigue el modelo Factory : es una función que retorna un objeto instanciado

```
.directive('counter', function counter() {  
  return {  
    };  
});
```

Los componentes se declaran directamente como objetos literales

```
.component('counter', {  
});
```





Directivas v. Componentes: simplificación

	Directiva	Componente
compile function	Si	No
link functions	Si	No
multiElement	Si	No
priority	Si	No
replace	Si (deprecated)	No
restrict	Si (A, E, C, M)	No (siempre E(lements))
templateNamespace	Si	No
terminal	Si	No



Directivas v. Componentes



	Directiva	Componente
bindings	No	Yes (binds to controller)
bindToController	Si (default: false)	No (use bindings instead)
controller	Yes	Si (default function() {})
controllerAs	Si (default: false)	Si(default: \$ctrl)
scope	Si (default: false)	No (scope siempre "isolate")
require	Si	Si
template	Si	Si, injectable
templateUrl	Si	Si, injectable
transclude	Si (default: false)	Si (default: false)





Componentes

```
angular.module('app...')  
.component("nombreComponente", {  
    // require: {'parent' : 'otroComponente'},  
    templateUrl : 'componentes/nombre.template.html',  
    controller : NombreComponente  
    // controllerAs: '$ctrl', // valor por defecto  
    bindings : {},  
    // transclude: true  
})
```

```
<nombre-componente "atributos">  
</nombre-componente >
```





Componentes: inyección de dependencias

Los controllers se definen como clases

```
1  /* ...
2  class SampleController {
3
4      /* ...
5      constructor($scope) {
6          'ngInject';
7          this.$scope = $scope;
8      }
9      /* ...
10     $onInit () []
11
12     this.name = "Sample"
13     this.value = parent.value
14
15     [] // Fin del $onInit
16
17 } // Fin del controller SampleController
18
19
```





Comunicación entre componentes (1)

Herencia del anidamiento mediante "require"

asociar una variable de un componente (normalmente parent) con el componente o la directiva que lo contiene

```
{ ... require: {  
    parent: '^parentComponent'  
},  
  
controller: function () {  
    // use this.parent to access required Objects  
    this.parent.foo();  
}  
...  
}
```





Comunicación entre componentes (2)

Atributos (bindings) → asociar una variable de un componente (normalmente parent) con los atributos usados al invocarlo

```
{ ...  
bindings: {  
oneWay: '<',  
strings: '@',  
twoWay: '=',  
callbacks: '&'  
}, ... }
```



Filtros



Uso de los filtros disponibles en angular

- en expresiones (i.e. vista)
- en el código JS (e.g. desde el controller)

- filter
- currency
- number
- date
- json
- lowercase
- uppercase
- limitTo
- orderBy





Filtros y servicios

Los filtros disponibles en angular corresponden al servicio `$filter`

pueden ser inyectados y utilizados desde el código JS (e.g. desde el controller)

```
class MiController {  
    constructor ($filter) {  
        this.$filter = $filter}  
    }  
    ... this.$filter ('<nombre_filtro>')('parámetros')
```

Igualmente es posible injectar cada filtro de forma independiente como `<nombrefiltro>Filter`

```
this.<nombre_filtro>Filter ('parámetros')
```



Ejemplo: Filtro usado como servicio

```
constructor ($filter) {  
    this.$filter = $filter;  
}
```

Inyección de dependencias

```
console.log(this.$filter('json')(this.oDatos));  
console.log(this.$filter('uppercase')(this.oDatos.titulo));  
console.log(this.$filter('date')(this.oDatos.fecha, "fullDate"));  
console.log(this.$filter('number')(this.oDatos.horas, 2));
```

```
constructor (numberFilter,dateFilter, uppercaseFilter, jsonFilter ) {  
    this.numberFilter = numberFilter;  
    this.dateFilter = dateFilter;  
    this.uppercaseFilter = uppercaseFilter;  
    this.jsonFilter = jsonFilter;  
}
```

Inyección de dependencias

```
console.log(this.jsonFilter(this.oDatos));  
console.log(this.uppercaseFilter(this.oDatos.titulo));  
console.log(this.dateFilter(this.oDatos.fecha, "fullDate"));  
console.log(this.numberFilter(this.oDatos.horas, 2));
```





Creación de filtros

Filtros “nativos” → Servicios que devuelven una función capaz de recoger unos argumentos y devolverlos con determinada modificación

Filtros propios →

```
.filter('nombreFiltro', function(){
  return function (input, [parámetros]) {
    // modificación del input
  }
});
```

- usos →
- en expresiones (i.e. vista)
 - en el código JS (e.g. desde el controller)



Ejemplo: Filtro “truncate”

```
constructor ($filter) {  
    this.$filter = $filter;  
}
```

Inyección de dependencias

```
console.log(this.$filter('json')(this.oDatos));  
console.log(this.$filter('uppercase')(this.oDatos.titulo));  
console.log(this.$filter('date')(this.oDatos.fecha, "fullDate"));  
console.log(this.$filter('number')(this.oDatos.horas, 2));
```

```
constructor (numberFilter,dateFilter, uppercaseFilter, jsonFilter ) {  
    this.numberFilter = numberFilter;  
    this.dateFilter = dateFilter;  
    this.uppercaseFilter = uppercaseFilter;  
    this.jsonFilter = jsonFilter;  
}
```

Inyección de dependencias

```
console.log(this.jsonFilter(this.oDatos));  
console.log(this.uppercaseFilter(this.oDatos.titulo));  
console.log(this.dateFilter(this.oDatos.fecha, "fullDate"));  
console.log(this.numberFilter(this.oDatos.horas, 2));
```





Scope y herencia

La directiva ngApp crea el *scope* "padre" de la aplicación

\$rootScope

Este objeto incluye diversos métodos, e.g.
relacionados con la gestión de los eventos

La directiva ngController crea un *scope* hijo, **\$scope**,
por **herencia prototípica** a partir del anterior

Es en este objeto en el que se definen las
propiedades de la vista, pero igualmente accede a
todos los elementos del prototipo del objeto padre

Otros directivas, e.g. ngRepeat, pueden crear
sus propios *scope* contenidos en el anterior





Herencia prototípica

prototypal inheritance

Basándose en la idea de que los objetos heredan de objetos, **Douglas Crockford** sugirió una función que aceptara un objeto y devolviera uno nuevo que tuviera al primero como prototipo

```
function object(o) {  
    function F() {}  
    F.prototype = o;  
    return new F();  
}
```

Este patrón ha sido recogido por ECMAScript 5 como el nuevo método estático **Object.create()**

Si tengo un objeto oPadre (e.g. creado literalmente)
oHijo = Object.create(oPadre)



Método Object .create()

Js

Incorporado al estándar ECMAScript5

Su **segundo parámetro** son las nuevas propiedades extra que se añaden al objeto hijo

```
var child = Object.create(parent, {  
    age: { value: 2 } // ECMA5 descriptor  
});
```

Este segundo parámetro corresponde a un objeto, en el que cada propiedad se define mediante los nuevos descriptores, también incorporados en ECMA5

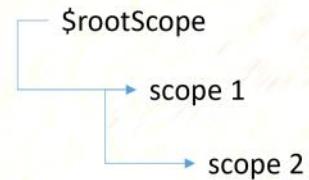
Alternativamente, una vez creado el objeto hijo, puede ser modificado dinámicamente como cualquier objeto JavaScript





Jerarquía de herencia

```
<div ng-app> <!-- creates a $rootScope -->
  <div ng-controller="OuterController">
    <div ng-controller="InnerController">
      </div>
    </div>
  </div>
```



La herencia de los *scopes* reproduce el anidamiento de los elementos del DOM

al encontrar `ng-controller="OuterController"`,
angular utiliza `$rootScope.new()` para crea un *scope*
(*scope 1*) que hereda de `$rootScope`

Al llega a `ng-controller="InnerController"`, es
`scope1.new()` quien crea un nuevo *scope* hijo
(*scope2*) que hereda de *scope1*



Funcionalidades del scope



Entre las funcionalidades incluidas en \$rootScope y en los *scopes* que se crean a partir de él están

- el seguimiento de los cambios
- el potente sistema de gestión de eventos de Angular

\$watch()
\$watchCollection() → seguimiento de los cambios

\$apply()
\$digest → mecanismo del doble binding

\$emit
\$broadcast()
\$on() → emisión y escucha de eventos





Listener: \$watch()

\$watch()

método del \$scope

Define un *listener* que **monitoriza** los cambios en cualquier propiedad del modelo

2 parámetros

- la propiedad del modelo a monitorizar
- la función *callback* que responderá a los cambios (*listener*): parámetros: <nuevo valor>, <valor anterior>

De esta forma se puede mantener el doble *binding* aun cuando otros elementos de Angular (e.g. servicios) realicen cambios en el modelo

Veremos un ejemplo al utilizar un servicio, concretamente localStorage



Ejemplo: Lista de socios / colaboradores

Lista de Socios

El número total de socios es de 5

Número en la lista de colaboradores: 0

- Socio número 1: **Andrés** [Añadir colaborador](#) [Eliminar colaborador](#)
- Socio número 2: **Maria** [Añadir colaborador](#) [Eliminar colaborador](#)
- Socio número 3: **Luis** [Añadir colaborador](#) [Eliminar colaborador](#)
- Socio número 4: **Carmen** [Añadir colaborador](#) [Eliminar colaborador](#)
- Socio número 5: **Antonio** [Añadir colaborador](#) [Eliminar colaborador](#)

Alejandro L. Cerezo - Madrid 2015

Contamos el número de colaboradores elegidos y
mostramos un aviso cuando llega a 2



\$watchCollection()



Equivale a `$watch()` pero la propiedad "vigilada" es un *array* o un *objeto*

La función listener se ejecuta cuando:

- array →
 - se añade un ítem al array
 - se actualiza elimina o reordena algún ítem existente

- objeto →
 - se añade una nueva propiedad
 - se actualiza elimina o reordena alguna propiedad existente



Angular Local-Storage

domingo, 6 de agosto de 2017 22:08

The slide has a yellow background with a faint grid pattern. At the top right is the Angular logo (a stylized 'A' inside a hexagon). The title 'Angular Local-Storage' is centered in a large, bold, teal font. Below the title is a subtitle in a smaller teal font: 'Utilizamos un módulo externo de AngularJS' followed by an arrow pointing to 'Acceso al LocalStorage incorporado en las nuevas APIs de HTML5'. A GitHub screenshot shows the repository 'grevory/angular-local-storage' with 309 commits, 3 branches, 16 releases, and 50 contributors. A blue box highlights the URL 'https://github.com/grevory/angular-local-storage'. The bottom left corner features the COAS TRAINING logo.

Utilizamos un módulo externo de AngularJS

→ Acceso al LocalStorage incorporado en las nuevas APIs de HTML5

<https://github.com/grevory/angular-local-storage>



Módulo Local-Storage

Instalación del módulo externo de AngularJS

```
bower install --save angular-local-storage
```

Desde index.html accedemos al fichero js correspondiente al módulo instalado

```
<script src="../lib/angular-local-storage/dist/angular-local-storage.min.js">  
</script>
```

Añadimos el módulo en la lista de los inyectados en la aplicación, utilizando el método module() del objeto angular, normalmente en app.js

```
angular.module('appModule', [  
  'LocalStorageModule'  
])
```



Configuración Local-Storage



Una vez instalado e injectado el módulo externo en la aplicación de AngularJS se posible configurarlo, gracias al método *config()* del objeto angular

```
angular.module("appModule")
.config(['localStorageServiceProvider',
  function(localStorageServiceProvider) {
    localStorageServiceProvider.setPrefix('ls')
}])
```

1. Se inyecta el servicio en la función responsable de la configuración
2. Se utilizan los elementos del API apropiados para configurara el servicio, e.g. *setPrefix*





Servicio Local-Storage

Una vez instalado e injectado el módulo externo en la aplicación de AngularJS, es posible utilizar en un determinado *controller* el servicio proporcionado en este módulo

```
angular.module('appModule')
  .controller('MainCtrl', ['$scope', 'localStorageService',
    function ($scope, localStorageService) {
      ...
    }
  ])
```

En el caso de *localStorageService*, los principales elementos del API son

isSupported
set / get
remove / clearAll
cookie

<http://gregpike.net/demos/angular-local-storage/demo/demo.html>



Listener de las modificaciones



El servicio `localStorageService` permite lógicamente modificar `localStorage` y al mismo tiempo repercutir estos cambios en las variables del `$scope`.

```
var tempTareas = localStorageService.get('tareas');
$scope.aTareas = tempTareas && tempTareas.split('\n') || [];

$scope.$watch('aTareas', function () {
    localStorageService.add('tareas', $scope.aTareas.join('\n'));
}, true);
```

Para que los cambios en el `$scope` realizados desde el interfaz también se reflejen en `localStorage` es necesario añadir el *listener* de Angular, `.$watch`



Opciones de invocación

Js

La diferencia entre funciones y métodos está en el patrón de invocación , que determina cual es el valor de this

Patrón de invocación
Function

`var var1 = función_a (parámetros)`

this es el objeto global del programa

Patrón de invocación
Method

`var var1 = objeto.función_a (parámetros)`

this es el objeto en el que esta el método

Patrón de invocación
Constructor

`var var1 = new Función_a (parámetros)`

this es el nuevo objeto creado

Patrón de invocación
Apply

`permite definir el valor de this`



Métodos *apply()* y *call()*

Js

Métodos del objeto **Function**, que Permiten ejecutar una función como si fuera un método de otro objeto: cambian el **binding** de la función ligándola al objeto, que pasa a ser **this** dentro de la función

```
function miFuncion(x) {  
    return this.numero + x;  
}  
var oObjeto = new Object();  
oObjeto.numero = 5;  
var resultado = miFuncion.call(oObjeto, 4);  
alert(resultado);
```

El primer parámetro del método es el objeto y la función se trata como si fuera un método del objeto.

Este procedimiento es necesario cuando un método se utiliza como función *callback*, e.g. al definir un manejador de objetos.





\$apply() y \$digest()

\$apply()

método del objeto `$scope` que toma como argumento una función y la ejecuta: de esa forma Angular tiene constancia de cualquier cambio que se produzca como parte de esa ejecución.

Ejemplo: diferencia entre usar el servicio `$http` o utilizar una llamada AJAX en JS estándar

Cualquier código que cambie los valores del modelo puede colocarse dentro de una función para luego llamar a `$scope.$apply()` pasándole la función como argumento

Esto inicia un ciclo de asimilación (`digest`) llamando a la función `$rootScope.$digest()`, que se propaga a todos los `scopes` hijos, haciendo una llamada a todos los `watchers` de los elementos de los modelos



Ejemplo: mensaje programado

Seguimiento de los cambios

¡¡Abrir la consola para ver los resultados!!

Mensaje despues de 3 segundos

Mensaje generado:

Alejandro L. Cerezo - Madrid 2015

Modificamos el modelo desde **setTimeout** y comprobamos como NO se refleja en la vista, a no ser que utilicemos **\$apply()**.
()Que es lo que hace el servicio **\$timeout**)





Eventos

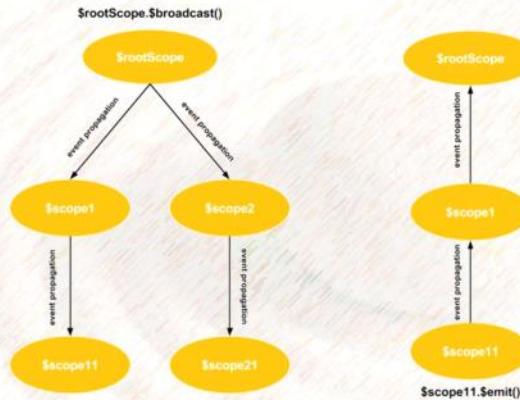
Entre las funcionalidades incluidas en \$rootScope y en los scopes que se crean a partir de él está el potente sistema de gestión de eventos de Angular, incluyendo los métodos

\$emit()
\$broadcast()
\$on()

Comunicación entre distintos controllers

\$broadcast

\$emit



Eventos: métodos



`$emit(<nombre del evento>, argumentos)`

`$broadcast(<nombre de _evento>, argumentos)`

Todos los argumentos incluidos a continuación del nombre se recibidos por las funciones puestas a la escucha del correspondiente evento

`$on(<nombre de _evento>, <función manejadora (oEvento, argumentos)>)`

La función manejadora recibe como parámetros

- El objeto Evento (equivalente al que existe en JS/JQuery)
- los argumentos con los que se emitió el evento



Objeto Evento

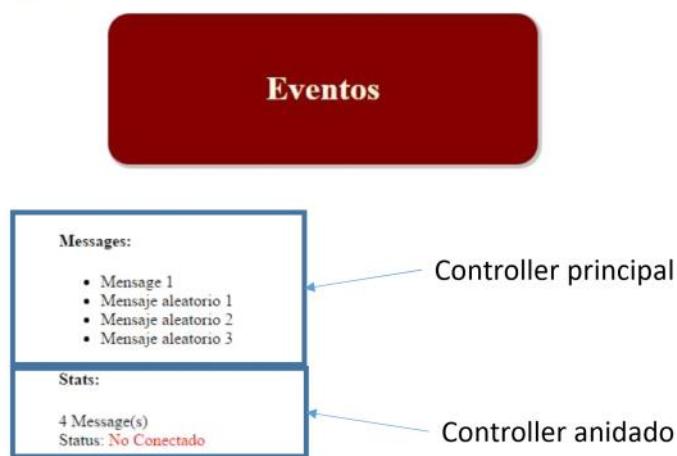


El objeto Evento (equivalente al que existe en JQuery)

targetScope	{Scope}: the scope on which the event was \$emit-ed or \$broadcast-ed.
currentScope	{Scope}: the scope that is currently handling the event. Once the event propagates through the scope hierarchy, this property is set to null.
name	{string}: name of the event.
stopPropagation	{function=}: calling stopPropagation function will cancel further event propagation (available only for events that were \$emit-ed).
preventDefault	{function}: calling preventDefault sets defaultPrevented flag to true.
defaultPrevented	{boolean}: true if preventDefault was called.



Ejemplo: Mensajes y Eventos



Alejandro L. Cerezo - Madrid 2015

Eventos como mecanismo de comunicación entre 2 controllers

