**3.4 Tipos de servicios**

Hasta ahora hemos visto cómo usar varios tipos de servicios pero pasemos ahora a crear nuestros propios servicios.

Como ya dijimos al principio de esta unidad realmente hay 5 tipos de servicios que son:

* [constant](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:05_constant)
* [value](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:06_value)
* [service](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service)
* [factory](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:08_factory)
* [provider](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:09_provider)

Bloques

Antes de ver las diferencias entre ellos veamos 2 nuevos artefactos de AngularJS:

* [Bloque config](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:04_tiposservicios#bloque_config)
* [Bloque run](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:04_tiposservicios#bloque_run)

Bloque config

Un bloque config es una función que definimos en nuestro código pero que se ejecutará al iniciar un programa en AngularJS para configurar un [provider](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:09_provider). Se define llamando al método config de un módulo.

Veamos un ejemplo:

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:04_tiposservicios)

|  |  |
| --- | --- |
| 1  2  3 | app.config(function() {    //Pon aquí el código que quieras  }); |

*siendo app un módulo de Angular*

Podemos definir tantos bloques config como queramos en una aplicación

Bloque run

El bloque run es, al igual que el bloque config, una función que se ejecuta al iniciar un programa en AngularJS. Se define llamando el método run de un módulo.

Veamos un ejemplo:

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:04_tiposservicios)

|  |  |
| --- | --- |
| 1  2  3 | app.run(function() {    //Pon aquí el código que quieras  }); |

*siendo app un módulo de Angular*

Podemos definir tantos bloques run como queramos en una aplicación

Diferencias

¿Cuáles son las diferencias entre un bloque config y un bloque run?

La primera diferencia es que primero se ejecutan todos los bloques config y luego todos bloques run.

El bloque run sería mas parecido al típico main de cualquier programa , podemos poner el código que queramos. Un bloque config por otro lado sirve para configurar los [provider](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:09_provider).

En un bloque run se puede acceder al [$rootScope](http://cursoangularjs.es/doku.php?id=unidades:04_masdirectivas:11_rootscope) mientras que en un bloque config no se puede. Ya veremos más adelante qué es el [$rootScope](http://cursoangularjs.es/doku.php?id=unidades:04_masdirectivas:11_rootscope).

Ya veremos en detalle qué es un [provider](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:09_provider) pero por ahora sabed que es la forma de configurar nuestros servicios antes de que podamos usarlos en un bloque run o en el resto de la aplicación.

Así que tenemos un servicio que queremos que sea configurable, así que usamos un [provider](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:09_provider) para permitir que dicho servicio sea configurable y usamos el bloque config para configurar el [provider](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:09_provider) que a su vez configura nuestro servicio.

Tipos

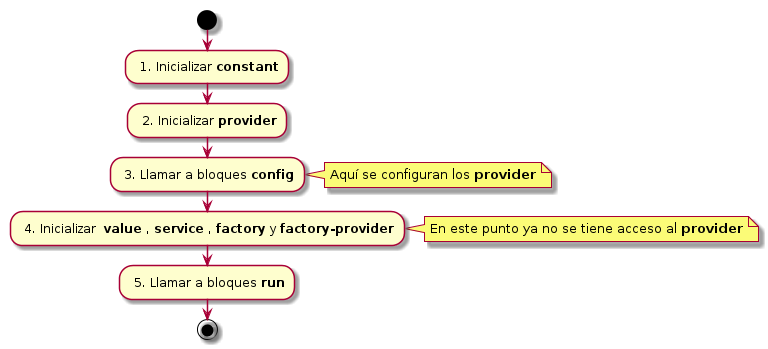
Volvamos ahora a los tipos de servicios. La diferencia principal entre ellos es dónde se pueden inyectar mientras se inicializa la aplicación, es decir que según el tipo de servicio los podremos inyectar en un bloque config o en un bloque run o en otro servicio, etc.

La siguiente tabla indica dónde se pueden inyectar o donde no inyectar cada uno de los servicios al **iniciar la aplicación**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Se puede inyectar en un** | | | |
| **Tipo** | **Provider** | **Bloque config** | **Servicio**[**1)**](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:04_tiposservicios#fn__1) | **Bloque run** |
| constant | Si | Si | Si | Si |
| provider | Si [2)](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:04_tiposservicios#fn__2) | Si | No | No |
| value | No | No | Si | Si |
| service | No | No | Si | Si |
| factory | No | No | Si | Si |
| factory-provider [3)](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:04_tiposservicios#fn__3) | No | No | Si | Si |

Una vez iniciada la aplicación, cualquiera de ellos , excepto el provider, ya se puede inyectar en cualquier otro artefacto como un controlador, directiva , etc.

El siguiente diagrama de actividad explica los pasos que realiza AngularJS al iniciar una aplicación.

[](http://cursoangularjs.es/lib/plugins/plantuml/img.php?width=0&height=0&title=PlantUML%20Graph&align=&version=2011-07-16&md5=58be6a3decd72e1f33c6eb1a6c6b8396&DokuWiki=7hok28cr5lsch94gcqeg327jo7)

* Actividad 1: Se puede ver cómo lo primero que se hace es inicializar las constant, por lo tanto éstas se pueden inyectar en cualquier sitio.
* Actividad 2: Seguidamente se crean los provider
* Actividad 3: Se llaman a los bloques config. Por lo tanto en un bloque config se puede inyectar tanto una constant como un provider. En un bloque config lo que se hace es configurar los provider.
* Actividad 4: Ahora es cuando se llama a todos los servicios *normales* que son los value , service , factory y factory-provider. Llegados a este punto los provider no están accesibles ya que al estar ya configurados solo se tiene acceso a los factory-provider correspondientes a sus provider.
* Actividad 5: Se llama al bloque run para inicializar la aplicación pudiendo inyectar cualquier tipo de servicio excepto los provider

Como es de esperar, AngularJS no permite la referencias circulares entre servicios. En ese caso se producirá el error:

Uncaught Error: [$injector:cdep] Circular dependency found: s1 <- s2 <- s1

Siendo 's1' y 's2' los servicios con referencias entre ellos.

Para acabar, decir que no creo que se haya entendido todo lo que he dicho en esta página ya que es bastante densa. En los siguientes temas seguimos hablando con todo ésto por lo que espero que finalmente se entienda.

[**1)**](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:04_tiposservicios#fnt__1) service, factory o factory-provider

[**2)**](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:04_tiposservicios#fnt__2) Pero únicamente si está definido en otro módulo

[**3)**](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:04_tiposservicios#fnt__3) Es la función factory llamada $get que genera el servicio dentro del provider

**3.7 service**

Hemos visto por ahora 2 tipos de servicio , las constant y los value. En ambos casos le pasábamos directamente el valor que debía tener el servicio. Con el tipo service [1)](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service#fn__1) le debemos pasar una *clase* [2)](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service#fn__2) JavaScript y será AngularJS el que cree internamente una instancia de la clase.

Lo veremos mas claro con el siguiente ejemplo:

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | var app=angular.module("app",[]);    function Rectangulo() {    this.ancho=0;    this.alto=0;      this.setAncho=function(ancho) {      this.ancho=ancho;    }      this.setAlto=function(alto) {      this.alto=alto;    }      this.getArea=function() {      return this.ancho \* this.alto;    }  }    app.service("rectangulo",Rectangulo);      app.controller("PruebaController",["$scope","rectangulo",function($scope,rectangulo) {    rectangulo.setAncho(3);    rectangulo.setAlto(6);      $scope.area=rectangulo.getArea();  }]); |

* Línea 3: Aquí hemos definido la función JavaScript con el constructor de nuestra clase llamada “Rectangulo”.
* Línea 20: Llamamos al método service y le pasamos como segundo parámetro la función con el constructor de la clase “Rectangulo”.

Inyectando el constructor

Una funcionalidad que permite AngularJS es que le podamos inyectar servicios en el propio constructor.

En el ejemplo anterior veíamos que el ancho y alto inicial era 0 , pero es un problema que los valores iniciales están *grabados a fuego* en el interior de la clase. Es mucho mas interesante que se puedan incluir en el constructor para que podamos elegir los que queramos.

Así que la clase quedará de la siguiente forma:

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | function Rectangulo(tamanyoInicial) {    this.ancho=tamanyoInicial.ancho;    this.alto=tamanyoInicial.alto;      this.setAncho=function(ancho) {      this.ancho=ancho;    }      this.setAlto=function(alto) {      this.alto=alto;    }      this.getArea=function() {      return this.ancho \* this.alto;    }  } |

* Línea 1: Ahora el constructor acepta como parámetro un objeto con las propiedades ancho y alto.
* Línea 2: Se asigna como ancho inicial el valor de la propiedad tamanyoInicial.ancho
* Línea 3: Se asigna como alto inicial el valor de la propiedad tamanyoInicial.alto

Lo siguiente es hacer que haya un servicio value que contenga el tamaño inicial del rectángulo.

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service)

|  |  |
| --- | --- |
| 1  2  3  4 | app.value("tamanyoInicialRectangulo",{    ancho:2,    alto:3  }); |

Y ahora nos queda inyectar el value “tamanyoInicialRectangulo” en el constructor de “Rectangulo” y AngularJS por suerte ya tiene esa funcionalidad programada [3)](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service#fn__3).

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service)

|  |  |
| --- | --- |
| 1 | app.service("cuadrado",['tamanyoInicialRectangulo',Cuadrado]); |

Vemos que junto al nombre del constructor que es Cuadrado hemos incluido el típico array de AngularJS con el nombre de los servicios a inyectar, que en nuestro caso es el value llamado tamanyoInicialRectangulo

El ejemplo completo quedaría de la siguiente forma:

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | var app=angular.module("app",[]);    app.value("tamanyoInicialRectangulo",{    ancho:2,    alto:3  });    function Rectangulo(tamanyoInicial) {    this.ancho=tamanyoInicial.ancho;    this.alto=tamanyoInicial.alto;      this.setAncho=function(ancho) {      this.ancho=ancho;    }      this.setAlto=function(alto) {      this.alto=alto;    }      this.getArea=function() {      return this.ancho \* this.alto;    }  }    app.service("rectangulo",['tamanyoInicialRectangulo',Rectangulo]);      app.controller("PruebaController",["$scope","rectangulo",function($scope,rectangulo) {    $scope.area=rectangulo.getArea();  }]); |

Hay que hacer notar que el nombre del value que se inyecta se llama “tamanyoInicialRectangulo” mientras que el parámetro del constructor se llama “tamanyoInicial”. Es decir que tienen nombres distintos. No hay problema en ello ya que una vez inyectado el valor da igual el nombre del parámetro del constructor.

Si no hubiéramos usado el array sí que se deberían de haber llamado igual , por lo que éste es un motivo más para usar siempre el array.

Service vs value

Ahora, ¿qué hemos ganado usando un service en vez de un value cuando el valor es una objeto de una clase? Si no necesitamos inyección de dependencias en el constructor la verdad es que no hemos ganado mucho.

El ejemplo inicial se podría haber hecho como un value de la siguiente forma:

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | var app=angular.module("app",[]);    function Rectangulo() {      this.setAncho=function(ancho) {      this.ancho=ancho;    }      this.setAlto=function(alto) {      this.alto=alto;    }      this.getArea=function() {      return this.ancho \* this.alto;    }  }    app.value("rectangulo",new Rectangulo());      app.controller("PruebaController",["$scope","rectangulo",function($scope,rectangulo) {    rectangulo.setAncho(3);    rectangulo.setAlto(6);      $scope.area=rectangulo.getArea();  }]); |

* Línea 18: Le pasamos directamente el objeto de la clase en vez de que internamente sea AngularJS el que cree el objeto.

Realmente no hay mucha diferencia ya que como veremos mas adelante, tanto los [value](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:06_value) como los [service](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service) como los [factory](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:08_factory) son solamente *azucar sintáctico* respecto a los [provider](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:09_provider). Es decir , internamente para AngularJS son únicamente un [provider](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:09_provider) pero al ser los [provider](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:09_provider) un poco complejos de programar se han creado estas funciones para simplificar su uso.

Pero aun así, yo veo una ventaja de un service sobre un value cuando es un objeto de una clase. Si un servicio nunca se va a usar , AngularJS no lo inicializará por lo tanto usando un service, nos podríamos ahorrar el gasto de crear la instancia si no llegara a usarse o al menos retrasarlo hasta que se use, mientras que con un value siempre se crearía.

Esto que puede parecer poca cosa, podría implicar gran gasto de recursos en grandes aplicaciones en las que hay gran cantidad de servicios y también debemos añadir todos los servicios que crean las librerías de terceros que usemos.

Es preferible usar un service en vez de un value cuando es una instancia de una clase ya que:

* Nos podemos ahorrar la creación de la instancia o al menos retrasarla hasta que sea necesaria
* Permite inyectar dependencias en el constructor

Ejemplo

Referencias

* [/ Developer Guide / Providers / Service](https://docs.angularjs.org/guide/providers#service-recipe)

[**1)**](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service#fnt__1) que en este tema no referimos al tipo concreto service y no a los servicios en general

[**2)**](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service#fnt__2) Es una función que actúa como constructor

[**3)**](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:07_service#fnt__3) A mí me recuerda a Spring que también permite inyectar dependencias en los constructores

**3.8 factory**

En este tema vamos a ver el servicio de factory. La principal diferencia es que al método factory le pasamos ahora una función para que ésta retorne ahora el valor del servicio. Es decir que tenemos una función JavaScript que actúa como factoría, retornando la propia función de factoría el valor del servicio.

Para que se entienda, vamos a ver los ejemplos que teníamos con el value ahora como un factory.

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:08_factory)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | var app=angular.module("app",[]);    app.factory("idioma",function() {    return "es-es";  });    app.factory("matematicas\_simples",function() {    return {      sumar:function(a,b) {        return a+b;      },      restar:function(a,b) {        return a-b;      }    }  });    app.factory("radio",function() {    return 10;  });    app.factory("area",function() {    return function(radio) {      return 3.1416\*radio\*radio;    }  });    app.controller("PruebaController",["$scope","idioma","matematicas\_simples","radio","area",function($scope,idioma,matematicas\_simples,radio,area) {    $scope.idioma=idioma;    $scope.suma=matematicas\_simples.sumar(3,6);    $scope.area=area(radio);  }]); |

* Líneas 3-5: Ahora al servicio de “idioma”, le pasamos una función anónima y dicha función retorna el String “es-es”, que es el verdadero valor del sevicio. Es decir la función anónima es una factoría que crea el valor . Aunque crearlo es simplemente crear el String.
* Líneas 7-16: El valor del servicio “matematicas\_simples” era un objeto javaScript con los métodos sumar y restar. Al ser un factory ahora le pasamos la función anónima de factoría que retorna el objeto JavaScript.
* Líneas 18-20: En el servicio de “radio” , la función anónima retorna el valor del servicio que es un 10.
* Líneas 22-26: El valor del servicio “area” es una función que calcula el área, por lo tanto la función anónima debe retornar la función que calcula el área. Es decir es una función que retorna otra función.
* Línea 28: Por último vemos que en el controlador se usan igual los servicios independientemente de si son value o factory, etc.

Inyectando dependencias

Al igual que en el service se podían inyectar dependencias en el constructor , en un factory se pueden inyectar dependencias en la función de factoría.

Vamos a hacer el ejemplo del tema anterior del service con la clase Rectangulo usando un servicio:

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:08_factory)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | var app=angular.module("app",[]);    app.value("tamanyoInicialRectangulo",{    ancho:2,    alto:3  });    function Rectangulo(tamanyoInicial) {    this.ancho=tamanyoInicial.ancho;    this.alto=tamanyoInicial.alto;      this.setAncho=function(ancho) {      this.ancho=ancho;    }      this.setAlto=function(alto) {      this.alto=alto;    }      this.getArea=function() {      return this.ancho \* this.alto;    }  }    app.factory("rectangulo",['tamanyoInicialRectangulo',function(tamanyoInicialRectangulo) {    var rectangulo=new Rectangulo(tamanyoInicialRectangulo);      return rectangulo;  }]);      app.controller("PruebaController",["$scope","rectangulo",function($scope,rectangulo) {    $scope.area=rectangulo.getArea();  }]); |

* Línea 25: Como ya hemos visto en las funciones de los controladores, inyectamos la dependencia tamanyoInicialRectangulo simplemente añadiendo el array con el nombre y poniendo la variable como argumento de la función.
* Línea 26: Aquí es cuando se ve realmente qué es una función de factoría. Estamos creando el valor del servicio al crear el objeto rectangulo. Vemos cómo le estamos pasando al constructor, el valor que inyectamos a la función.
* Línea 28: como ya sabemos se debe retornar el valor del servicio.

La necesidad del factory

Vamos a explicar ahora un ejemplo en el que necesitamos la función factory ya que la creación del valor del servicio va a ser un poco compleja.

Vamos a suponer que necesitamos enviar el hash de una contraseña a un servidor. Vamos a tener un servicio de AngularJS llamado hash que es una función. Esta función aceptará como parámetro un String y nos retornará el hash en formato Base64. Pero queremos que este servicio sea reutilizable por lo que va a soportar varios tipos de funciones de Hash: MD5, SHA1, SHA2-256 y SHA-2-512.

Para implementar las distintas funciones de Hash vamos a usar la librería [CryptoJS](https://code.google.com/p/crypto-js/). Las distintas funciones de Hash que tiene son:

* <https://code.google.com/p/crypto-js/#MD5>: CryptoJS.MD5
* <https://code.google.com/p/crypto-js/#SHA-1>: CryptoJS.SHA1
* <https://code.google.com/p/crypto-js/#SHA-2>: CryptoJS.SHA256 y CryptoJS.SHA512

y la forma de generar el resultado en Base64 es mediante [The Hasher Output](https://code.google.com/p/crypto-js/#The_Hasher_Output) usando la función: hash.toString(CryptoJS.enc.Base64)

Un ejemplo de todo ello es el siguiente:

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:08_factory)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <script src="<http://crypto-js.googlecode.com/svn/tags/3.1.2/build/rollups/sha256.js>"></script>  <script src="<http://crypto-js.googlecode.com/svn/tags/3.1.2/build/components/enc-base64-min.js>"></script>  <script>      var hash = CryptoJS.SHA256("Message");        alert(hash.toString(CryptoJS.enc.Base64)); // L3dmip37+NWEi57rSnFFypTG7ZI25Kdz9tyvpRMrL5E=  </script> |

Para configurar el servicio es necesario crear un value llamado algoritmo con el algoritmo que queremos usar.

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:08_factory)

|  |  |
| --- | --- |
| 1 | app.value("algoritmo","SHA-1"); |

Es este caso hemos configurado el algoritmo para que sea “SHA-1”.

Gastar el servicio hash es tan sencillo como pasar el String y nos retorna otro.

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:08_factory)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | app.controller("PruebaController",["$scope","hash",function($scope,hash) {    $scope.password="s3cret";    $scope.getHash=function(message) {      var hashResult=hash(message);      return hashResult;    }  }]); |

* Línea 1: En el controlador inyectamos la función de hash llamada hash.
* Línea 3: Se ha creado en el $scope una función llamada getHash que acepta un String como parámetro y retorna el resultado de llamar a la función de Hash.

El código HTML es el siguiente:

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:08_factory)

|  |  |
| --- | --- |
| index.html | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | <!DOCTYPE html>  <html ng-app="app">      <head>      <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.min.js"></script>      <script src="script.js"></script>      <script src="<http://crypto-js.googlecode.com/svn/tags/3.1.2/build/rollups/md5.js>"></script>      <script src="<http://crypto-js.googlecode.com/svn/tags/3.1.2/build/rollups/sha1.js>"></script>      <script src="<http://crypto-js.googlecode.com/svn/tags/3.1.2/build/rollups/sha256.js>"></script>      <script src="<http://crypto-js.googlecode.com/svn/tags/3.1.2/build/rollups/sha512.js>"></script>      <script src="<http://crypto-js.googlecode.com/svn/tags/3.1.2/build/components/enc-base64-min.js>"></script>    </head>      <body ng-controller="PruebaController">      Contrase&ntilde;a:<input ng-model="password" />      <br>      El hash de la contrase&ntilde;a es:      <br>      {{getHash(password)}}    </body>  </html> |

* Líneas 7-11: Cargamos la librería con los 4 algoritmos de Hash a usar y la codificación en Base64.
* Línea 15: Tenemos un tag <input> para que el usuario escriba la contraseña.
* Línea 18: Se llama a la función getHash para que muestre el hash.

Ahora viene la parte interesante, tenemos que crear un servicio mediante un factory que según el valor del value “algoritmo” utilice una función de Hash u otra pero además queremos que el resultado de llamar a dicha función siempre esté codificado en Base64.

El factory

Como ya hemos dicho nuestro servicio llamado hash nos retornará una función que realiza el hash pero con el algoritmo configurable y el resultado codificado en Base64.

Veamos como queda la función:

[?](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:08_factory)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | app.factory("hash",['algoritmo',function(algoritmo) {    var hashFunction;      if (algoritmo==="MD5") {      hashFunction=CryptoJS.MD5;    } else  if (algoritmo==="SHA-1") {      hashFunction=CryptoJS.SHA1;    } else  if (algoritmo==="SHA-2-256") {      hashFunction=CryptoJS.SHA256;    } else  if (algoritmo==="SHA-2-512") {      hashFunction=CryptoJS.SHA512;    } else {      throw Error("El tipo de algoritmo no es válido:"+algoritmo);    }      var hash=function(message) {      var objHashResult=hashFunction(message);        var strHashResult=objHashResult.toString(CryptoJS.enc.Base64);        return strHashResult;    }      return hash;    }]); |

* Línea 1: Inyectamos el valor del algoritmo a usar.
* Líneas 4 a 14: En función del valor de algoritmo usamos una función u otra de Hash. Pero ésta no va a ser la verdadera función que vamos a retornar ya que el resultado de esta función es un Objeto pero nosotros queremos que sea un String codificado en Base64.
* Línea 16: Ahora creamos la verdadera función , la cual es la que retornará el servicio.Y que es la que realmente será llamada desde el controlador. Vemos que acepta como único argumento el mensaje.
* Línea 17: Llamamos a la función que verdaderamente genera el Hash
* Línea 19: Ahora transformamos el objeto en un String en Base64 llamando al método toString
* Línea 24: Aquí es donde la función factory retorna la función de hash que hemos creado.

Si has llegado hasta aquí y lo has entendido todo: ¡Felicidades! pero sino , tampoco pasa nada. Era un ejemplo para ver la potencia de las funciones factory y lo útiles que pueden llegar a ser. Así que lo realmente útil siempre es el factory ya que nos permite crear cualquier tipo de valor de un servicio por muy complejo que sea.

value vs service vs factory

Esta es la típica pregunta que aparece siempre en los foros de AngularJS y la respuesta es bastante sencilla. Realmente da igual cuál uses porque todos acaban siendo un provider para AngularJS. Pero aun así ¿cuál es más recomendable?

1. Si tienes una clase de la que es necesario crear una instancia mejor usa un service: Sólo tendrás que pasarle el nombre de la clase y ya está.
2. Si tienes directamente el valor, mejor usar el value: Solo tendrás que pasarle ese valor y ya está.
3. Si no queda más remedio entonces usa el factory: Es lo más complejo de usar así que debería ser siempre la última opción.

En el siguiente tema por fin veremos el [provider](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:09_provider) el cual tiene una funcionalidad extra al factory que si la necesitamos estaremos obligados a usarlo.

En esta discusión no hemos hablado del constant ya que éste es distinto a los 3 anteriores [1)](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:08_factory#fn__1) puesto que permite ser inyectado en un bloque config y en un provider, cosa que ninguno de los 3 anteriores permite, por lo que si necesitamos esa funcionalidad deberemos usar obligatoriamente constant.

Referencias

* [/ Developer Guide / Providers / Factory](https://docs.angularjs.org/guide/providers#factory-recipe)

[**1)**](http://cursoangularjs.es/doku.php?id=unidades:03_servicios:08_factory#fnt__1) value , service y factory