

**Angular**  **JS**

# 

Contenido

[2](#_Toc21381575)

[**Lo escencial** 16](#_Toc21381576)

[**¿Qué es AngularJS?** 17](#_Toc21381577)

[**¿Cómo se ve una aplicación angular?** 18](#_Toc21381578)

[**¿Cómo se arquitectura una aplicación angular?** 19](#_Toc21381579)

[**Template** 20](#_Toc21381580)

[**Scope** 22](#_Toc21381581)

[**controller** 23](#_Toc21381582)

[**Relación con MVC** 24](#_Toc21381583)

[**Resumen** 26](#_Toc21381584)

[**Una aplicación simple** 28](#_Toc21381585)

[**Escribiendo el controlador** 30](#_Toc21381586)

[**Aumentando el $scope** 36](#_Toc21381587)

[**Poniendolo todo junto.** 38](#_Toc21381588)

[**El módulo raíz-root module** 41](#_Toc21381589)

[**El controller** 45](#_Toc21381590)

[**Accediendo al scope** 46](#_Toc21381591)

[**presiones** 48](#_Toc21381592)

[**Resumen** 50](#_Toc21381593)

[Inyección de dependencias 51](#_Toc21381594)

[**La versión realmente corta** 51](#_Toc21381595)

[**La versión ligeramente más larga, Parte I: dependencia de no inyección** 52](#_Toc21381596)

[**La versión ligeramente más larga, Parte II: Inyección de dependencias** 54](#_Toc21381597)

[**La versión ligeramente más larga, Parte III: ¿Por qué hacemos esto?** 57](#_Toc21381598)

[**Enlaces bidireccionales** Two-way bindings 61](#_Toc21381599)

[**ngModel** 63](#_Toc21381600)

[**Ejemplo** 64](#_Toc21381601)

[**Enlace de clases CSS** 68](#_Toc21381602)

[Binding CSS classes 68](#_Toc21381603)

[**ngClass** 69](#_Toc21381604)

[**Ejemplo** 70](#_Toc21381605)

[**Repetidores** 75](#_Toc21381606)

[Repeaters 75](#_Toc21381607)

[**Ejemplo** 79](#_Toc21381608)

[**Visibilidad** 86](#_Toc21381609)

[Visibility 86](#_Toc21381610)

[**Mostrando y escondiéndose** Showing and hiding 87](#_Toc21381611)

[**Agregar y quitar** Adding and removing 89](#_Toc21381612)

[**Ejemplo** 93](#_Toc21381613)

[Services 99](#_Toc21381614)

[**Complejidad de manejo** 100](#_Toc21381615)

[Handling Complexity 100](#_Toc21381616)

[**Inyección de dependencia** 102](#_Toc21381617)

[Dependency injection 102](#_Toc21381618)

[**Servicios** 104](#_Toc21381619)

[Services 104](#_Toc21381620)

[**La receta del valor** 107](#_Toc21381621)

[The Value Recipe 107](#_Toc21381622)

[**¡Cuidado!** 110](#_Toc21381623)

[Watch out! 110](#_Toc21381624)

[**Módulos** 112](#_Toc21381625)

[Modules 112](#_Toc21381626)

[**Mas inyeccion** 115](#_Toc21381627)

[More injection 115](#_Toc21381628)

[**Ejemplo** 119](#_Toc21381629)

[**La receta de la fábrica** 122](#_Toc21381630)

[The Factory Recipe 122](#_Toc21381631)

[**Ejemplo** 124](#_Toc21381632)

[**La receta de servicio** 129](#_Toc21381633)

[The Service Recipe 129](#_Toc21381634)

[**Ejemplo** 132](#_Toc21381635)

[**La receta del proveedor** 139](#_Toc21381636)

[The Provider Recip 139](#_Toc21381637)

[**Ejemplo** 142](#_Toc21381638)

[**Definiendo el proveedor** 142](#_Toc21381639)

[**Usando el proveedor** 146](#_Toc21381640)

[**La receta constante** 153](#_Toc21381641)

[The Constant Recipe 153](#_Toc21381642)

[**Ejemplo** 157](#_Toc21381643)

[$watch 160](#_Toc21381644)

[**¿Pero por qué?** 163](#_Toc21381645)

[**Una visión general de $ watch** 164](#_Toc21381646)

[**Ejemplo** 167](#_Toc21381647)

[**Filters** 169](#_Toc21381648)

[**Sintax** 171](#_Toc21381649)

[**Ejemplo** 174](#_Toc21381650)

[**Encadenamiento de filtros** Chaining filters 178](#_Toc21381651)

[**Validación de datos** 182](#_Toc21381652)

[Data Validation 182](#_Toc21381653)

[**Definiendo validez** 185](#_Toc21381654)

[**Ejemplo** 186](#_Toc21381655)

[**Ajax con $resource** 193](#_Toc21381656)

[**Introducción a $resource** 194](#_Toc21381657)

[**Los basicos** 195](#_Toc21381658)

[**Ejemplo** 202](#_Toc21381659)

[**Configuración** 204](#_Toc21381660)

[**Código** 207](#_Toc21381661)

[**$ resource en el servidor** 213](#_Toc21381662)

[**Más sobre $resource ...** 215](#_Toc21381663)

# **Lo escencial**

## **¿Qué es AngularJS?**

**AngularJS es un marco de JavaScript que se utiliza para crear aplicaciones web ricas y extensibles. Se ejecuta en JavaScript y HTML simples, por lo que no necesita ninguna otra dependencia para que funcione, y es independiente de CSS, por lo que puede usar el marco / metodología CSS que desee al diseñar su aplicación Angular.**

## **¿Cómo se ve una aplicación angular?**

**El código de la derecha es un ejemplo de una aplicación angular "hello world". Cuando se ejecuta, "¡Hola mundo!" se imprime en pantalla usando AngularJS. No se preocupe demasiado por entender el código todavía. Trabajaremos este ejemplo paso a paso en la próxima lección.**

## **¿Cómo se arquitectura una aplicación angular?**

**Las aplicaciones angulares en su forma más básica consisten en tres componentes.**

* **Template**
* **Controller**
* **scope**

### **Template**

**La plantilla es la parte HTML de la aplicación. Escribir una plantilla es exactamente como escribir una página HTML estática, excepto que las plantillas contienen una sintaxis adicional que permite inyectar datos en ellas para proporcionar una experiencia de usuario personalizada. Si alguna vez ha escrito una página HTML utilizando un marco web del lado del servidor, se sentirá como en casa escribiendo plantillas en Angular. Sin embargo, la característica que diferencia las plantillas angulares de las páginas generadas por el servidor es que, en Angular, los datos se pueden inyectar, modificar y eliminar de las plantillas sin necesidad de actualizar la página. Esta característica proporciona una experiencia más fluida para el usuario final y mejora la sensación general de las aplicaciones web angulares.**

### **Scope**

**El alcance es un componente muy importante en aplicaciones angulares. El alcance es el objeto que representa el "modelo" de su aplicación. Contiene campos que almacenan datos que se presentan al usuario a través de la plantilla, así como funciones a las que se puede llamar cuando el usuario realiza ciertas acciones, como hacer clic en un botón.**

### **controller**

**El controlador desempeña un papel de soporte en aplicaciones angulares. El controlador es una función que generalmente toma un objeto de ámbito vacío como parámetro y le agrega los campos y funciones que luego se expondrán al usuario a través de la vista.**

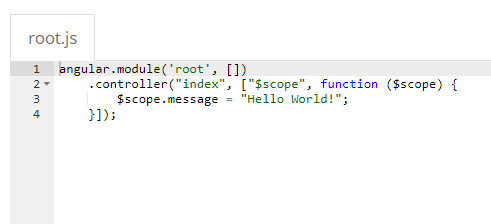
## **Relación con MVC**

**Hasta ahora, he tratado intencionalmente de evitar hacer comparaciones entre Angular y MVC en beneficio de aquellos que no han trabajado antes con el patrón MVC. Si ha trabajado con MVC, entonces probablemente no necesitará ayuda para identificar cuán estrechamente se parece la arquitectura de Angular a la de una aplicación basada en MVC. Como los dos están tan estrechamente vinculados, de aquí en adelante puedo referirme a la plantilla como la "vista" y el alcance como el "modelo". Si no está familiarizado con MVC, tenga en cuenta que estos términos son intercambiables y que no estoy hablando de nada que no haya visto antes.**

## **Resumen**

**La plantilla, el alcance y el controlador son los tres bloques de construcción principales de una aplicación web angular. Si bien Angular tiene mucho más que estos tres componentes, con una comprensión de lo que hacen y cómo interactúan, ya podrá conectar una simple aplicación Angular.**

****

****

# **Una aplicación simple**

**Estas lecciones están destinadas a ser interactivas. Siga a lo largo de las páginas de códigos a la derecha a medida que avanzamos.**

**Ahora que tenemos una comprensión básica de qué se trata Angular y cómo encaja una aplicación, entremos e integremos Angular en una página web simple.**

## **Escribiendo el controlador**

**Para esta lección, comenzamos con una página web muy básica que contiene una sola, vacía <div>. Usemos Angular para insertar el texto "Hello World" dinámicamente en la página.**

**Lo primero que debemos hacer es definir nuestro controlador. Recuerde, el controlador es una función que aumenta el alcance con datos, funciones y demás.**

**Para definir nuestro controlador, primero debemos definir lo que se conoce como módulo . No discutimos los módulos en la última lección, pero no hay que preocuparse; Son extremadamente simples de entender. Los módulos actúan como contenedores lógicos para componentes en aplicaciones angulares. Cada vez que definimos un componente como un controlador, debe ir en un módulo. La sintaxis para definir un módulo es la siguiente.**

**angular.module("root", [])**

**¡Eso es todo al respecto! El primer argumento, la cadena, define el nombre del módulo, que en este caso es root. El segundo argumento, la matriz vacía, se usa para la inyección de dependencia. Todavía no vamos a entrar en el mecanismo de inyección de dependencia de Angular, pero si no está familiarizado con el concepto de DI, le recomiendo que al menos lo conozca antes de continuar. \***

**De todos modos, en este ejemplo no necesitamos inyectar dependencias en nuestro módulo, por lo que la matriz está vacía. Ahora, debo enfatizar el siguiente punto porque es un lugar común para que los principiantes se tropiecen al declarar módulos. No olvides la matriz vacía . Si lo olvida, Angular intentará recuperar un módulo existente para usted en lugar de crear una instancia nueva y se generará un error.**

**De todos modos, ahora que hemos declarado nuestro módulo, podemos pasar a declarar nuestro controlador. Declaramos nuestro controlador directamente en el módulo que acabamos de crear usando la controller()función de esta manera.**

**angular.module("root", [])**

**.controller("index", ["$scope", function($scope) {**

**}]);**

**La sintaxis de la función del controlador es muy similar a la sintaxis del módulo. El primer argumento especifica el nombre del controlador que estamos declarando. El segundo argumento es una matriz cuyo último argumento es una función. Esta es la función del controlador que usaremos para aumentar nuestro alcance.**

**Los elementos en la matriz que preceden a la función del controlador son los nombres de las dependencias que pasaremos a la función como argumentos. En este caso, estamos pasando la $scopedependencia al controlador. Este es el alcance al que agregaremos nuestras funciones y variables en un momento. Es importante tener en cuenta que el orden de las dependencias en la matriz debe coincidir con el orden de los argumentos de la función.**

## **Aumentando el $scope**

**Ahora que nuestra función de controlador está definida, podemos comenzar a trabajar en $scope. Definamos una variable llamada messageque contendrá la cadenaHello World!**

**angular.module("root", [])**

**.controller("index", ["$scope", function($scope) {**

**$scope.message = "Hello World";**

**}]);**

**Excelente. Ahora todo lo que tenemos que hacer es conectar nuestra plantilla para mostrar este valor en el vacío div.**

## **Poniendolo todo junto.**

**Volvamos nuestra atención a nuestro HTML. Por ahora solo tenemos una página simple.**

**Necesitamos hacer cinco cosas para cablear en Angular.**

1. **Incluya una referencia al archivo de script Angular.**
2. **Incluya una referencia a nuestro archivo de script de controlador.**
3. **Dile a Angular qué módulo estamos usando como nuestro "módulo raíz"**
4. **Dígale a Angular a qué parte de la página debe aplicar nuestro controlador y el alcance correspondiente.**
5. **Dile a Angular que muestre nuestro mensaje en el <div>.**

**De hecho, para esta lección y en todo este sitio, ya me he ocupado de los pasos 1 y 2 para usted detrás de escena. En las próximas lecciones, a menudo también me encargaré del paso 3. Sin embargo, en esta lección, los pasos 3, 4 y 5 dependen de usted.**

### **El módulo raíz-root module**

**En Angular-speak, el "módulo raíz" es el módulo base para la aplicación. En nuestro caso, solo tenemos un módulo, por lo que el proceso de selección es fácil. Necesitamos decirle a Angular que queremos que el rootmódulo sea nuestra raíz. Hacemos esto agregando lo que se conoce como directiva al htmlelemento de la página. A medida que avancemos, escuchará mucho sobre las directivas. Las directivas son esencialmente instrucciones que usamos para decirle a Angular qué queremos que haga.**

**La directiva que necesitamos aplicar es la ngAppdirectiva. Aplicamos esta directiva al elemento HTML en forma de un atributo, así.**

**<html ng-app="root">**

**Hay un par de cosas a tener en cuenta aquí.**

* **El nombre de la directiva es ngApppero se aplica en forma de un atributo llamado ng-app. Discutiremos las complejidades de esta sintaxis en una lección posterior, pero por ahora solo comprenda que las directivas tienen nombres en camelCase pero están escritas en minúsculas delimitadas por guiones cuando se aplican como atributos.**
* **El valor del atributo es el nombre de nuestro módulo raíz.**

**Eso es todo lo que necesitamos hacer. Angular ahora entiende que el módulo nombrado rootes el módulo raíz de la aplicación.**

### **El controller**

**Ahora necesitamos aplicar nuestro controlador a la página. Aplicamos el controlador utilizando la ngControllerdirectiva. El elemento al que aplicamos el ng-controlleratributo es importante. Este elemento y todos sus elementos secundarios podrán acceder al $scopeobjeto aumentado por el controlador.**

**Para esta lección, vamos a aplicar la directiva al divelemento así.**

**<div ng-controller="index"></div>**

### **Accediendo al scope**

**¡Todo lo que queda por hacer es mostrar nuestro mensaje en la página! Para imprimir un valor de variable en nuestra plantilla, utilizamos la sintaxis de llaves dobles, especificando el nombre de la $scopevariable que deseamos imprimir.**

**{{message}}**

**Agregue esta sintaxis al <div>elemento en el que aplicamos nuestro controlador y ejecute el código. Si todo está bien, ¡debería ver Hello Worldimpreso en alto y claro en la página para que todos lo vean! Si algo salió mal, no te preocupes. Puede ver la respuesta haciendo clic en "Estoy atascado". botón, o restablecer el ejemplo por completo y comenzar de nuevo.**

## **presiones**

**Ahora que ha visto cómo vincular un valor de ámbito a la plantilla, intente esto. Reemplace {{message}}con una ecuación matemática como {{1 + 1}}y ejecute la aplicación. ¡Verás que Angular habrá evaluado la ecuación e impreso el resultado! Esto se debe a que Angular considera el valor dentro de las llaves como una expresión . Las expresiones son como simples declaraciones de JavaScript, lo que significa que, además de admitir $scopecampos, también admiten funciones simples como esta. Los detalles de exactamente qué es y no es compatible con las expresiones angulares se pueden encontrar**[**aquí en el sitio angular**](http://docs.angularjs.org/guide/expression)**.**

## **Resumen**

**¡Aprendimos mucho esta lección! Aprendimos sobre los módulos, cómo declarar controladores y, lo más importante, cómo conectar todo para crear una aplicación completa. Si bien no hemos visto nada muy impresionante en términos de resultados, no se preocupe porque estamos llegando allí. En la próxima serie de lecciones, analizaremos en profundidad las directivas y cómo pueden enriquecer las aplicaciones angulares.**

# Inyección de dependencias

### **La versión realmente corta**

**La inyección de dependencia significa dar a un objeto sus variables de instancia. De Verdad. Eso es.**

### **La versión ligeramente más larga, Parte I: dependencia de no inyección**

**Las clases tienen estas cosas que llaman métodos. Llamemos a esas "dependencias". La mayoría de las personas los llaman "variables". A veces, cuando se sienten elegantes, los llaman "variables de instancia".**

**Ejemplo de clase pública {  
  private DatabaseThingie myDatabase;  
  
  Ejemplo público () {  
    myDatabase = new DatabaseThingie ();  
  }  
  
  público vacío DoStuff () {  
    ...  
    myDatabase.GetData ();  
    ...  
  }  
}**

**Aquí, tenemos una variable ... uh, dependencia ... llamada "myDatabase". Lo inicializamos en el constructor.**

### **La versión ligeramente más larga, Parte II: Inyección de dependencias**

**Si quisiéramos, podríamos pasar la variable al constructor. Eso "inyectaría" la "dependencia" en la clase. Ahora, cuando usamos la variable (dependencia), usamos el objeto que nos dieron en lugar del que creamos.**

**Ejemplo de clase pública {  
  private DatabaseThingie myDatabase;  
  
  Ejemplo público () {  
    myDatabase = new DatabaseThingie ();  
  }**

**Ejemplo público (DatabaseThingie useThisDatabaseInstead) {  
    myDatabase = useThisDatabaseInstead;  
  }**

**público vacío DoStuff () {  
    ...  
    myDatabase.GetData ();  
    ...  
  }  
}**

**Eso es realmente todo lo que hay que hacer. El resto son solo variaciones sobre el tema. Puede establecer la dependencia (<cough> variable ) en ... espere ... un método setter. Puede establecer la dependencia llamando a un método setter definido en una interfaz especial. Puede hacer que la dependencia sea una interfaz y luego pasar polimórficamente un poco de multijugos. Lo que sea.**

### **La versión ligeramente más larga, Parte III: ¿Por qué hacemos esto?**

**Entre otras cosas, es útil para aislar clases durante las pruebas.**

**public class ExampleTest {  
  TestDoStuff () {  
    MockDatabase mockDatabase = new MockDatabase ();  
  
    // MockDatabase es una subclase de DatabaseThingie, por lo que podemos  
    // "inyectarla" aquí:  
    Ejemplo ejemplo = nuevo Ejemplo (mockDatabase);  
  
    ejemplo.DoStuff ();  
    mockDatabase.AssertGetDataWasCalled ();  
  }  
}**

**Ejemplo de clase pública {  
  private DatabaseThingie myDatabase;  
  
  Ejemplo público () {  
    myDatabase = new DatabaseThingie ();  
  }  
  
  Ejemplo público (DatabaseThingie useThisDatabaseInstead) {  
    myDatabase = useThisDatabaseInstead;  
  }  
  
  público vacío DoStuff () {  
    ...  
    myDatabase.GetData ();  
    ...  
  }  
}**

**Eso es. La inyección de dependencia en realidad solo pasa una variable de instancia.**

# **Enlaces bidireccionales** Two-way bindings

**En esta sección, veremos algunas directivas básicas que puede usar para agregar interactividad y flexibilidad a sus aplicaciones web. Hay mucho más sobre el marco angular de lo que acabamos de ver, pero mirar algunas directivas de inmediato le dará una idea de lo que puede hacer con Angular y, con suerte, lo inspirará a aprender más sobre el marco en las secciones que siguen. .**

## **ngModel**

**Vamos a empezar por mirar a una de las directivas más importantes en angular, ngModel. A diferencia de los valores enlazados utilizando la sintaxis de doble llave, ngModelnos permite enlazar valores a elementos HTML como los campos de entrada. Cuando se usa ngModel, no solo los cambios en el alcance se reflejan en la vista, sino que los cambios en la vista se reflejan nuevamente en el alcance.**

## **Ejemplo**

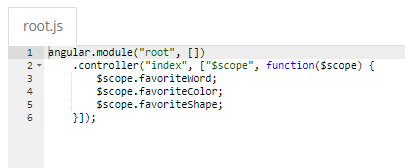
**Para ver ngModelen acción, vamos a usarlo para vincular el favoriteWordcampo en el ámbito al campo de entrada en la página HTML. Para hacer esto, simplemente necesitamos aplicar ngModelal campo de entrada en forma de un atributo como este.**

**<input type="text" ng-model="favoriteWord" />**

**Como puede ver, ngModeltoma una expresión que casi siempre es el nombre del $scopecampo al que queremos unir nuestro elemento. Ahora, continúe e intente vincular las otras dos variables de ámbito a sus respectivos campos de entrada. Como recordatorio, recuerde que los valores de los botones de opción se recuperan de cada <input>etiqueta individual , y que los valores desplegables se recuperan de la <select>etiqueta que lo abarca todo .**

**Una vez que los valores están vinculados, use la sintaxis de llaves dobles para mostrar cada valor junto a su entrada correspondiente y ejecute el ejemplo para ver el enlace bidireccional en acción.**

****

****

# **Enlace de clases CSS**

# Binding CSS classes

**Así como Angular nos permite vincular los valores del ámbito a los campos de entrada, también nos permite vincular los valores del ámbito como nombres de clase a elementos HTML. Como puede imaginar, esta característica es extremadamente útil y probablemente una de las características más utilizadas en Angular por completo.**

## **ngClass**

**Para vincular una clase CSS a un elemento HTML, usamos la ngClassdirectiva. ngClasstoma como entrada una expresión que debe evaluar a una de las siguientes.**

* **Una cadena de nombres de clase delimitados por espacios.**
* **Una matriz de nombres de clase.**
* **Un mapa (objeto) donde las claves son nombres de clase y los valores son valores booleanos que indican si se aplica o no la clase.**

## **Ejemplo**

**Para esta demostración, he definido tres clases de CSS para usted detrás de escena:**

* **bold**
* **italic**
* **underline**

**Intentemos aplicar estas tres clases al spantexto de muestra en la ventana HTML de acuerdo con la siguiente lógica:**

* **Si el valor en el cuadro de texto% 2 == 0, se aplica bold.**
* **Si el valor en el cuadro de texto% 3 == 0, se aplica italic.**
* **Si el valor en el cuadro de texto% 5 == 0, se aplica underline.**

**Para este ejemplo, intentemos hacer esto con un mapa.**

**Comenzaremos definiendo una función para cada uno de los criterios en el controlador.**

**$scope.isBold = function () { return $scope.value % 2 === 0; };**

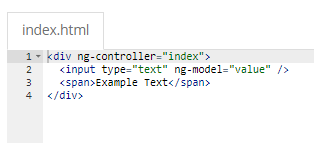
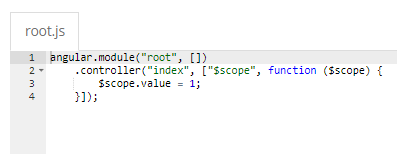
**$scope.isItalic = function () { return $scope.value % 3 === 0; };**

**$scope.isUnderlined = function () { return $scope.value % 5 === 0; };**

**Ahora, apliquemos la ngClassdirectiva al tramo de texto en forma de atributo y pasémosle un mapa de nombres de clase y valores booleanos.**

**ng-class="{bold: isBold(), italic: isItalic(), underline: isUnderlined()}"**

**Una vez aplicado, ejecute el programa. Cuando cambia el número en el campo de texto, las clases aplicadas al texto de muestra deberían cambiar en consecuencia.**

** **

# **Repetidores**

# Repeaters

**A menudo sucede que necesita mostrar una recopilación de datos al usuario. Para esta tarea, Angular le proporciona la ngRepeatdirectiva. ngRepeatle da el poder de definir una plantilla para un solo elemento en una colección, y luego hacer que se repita para todos los elementos de una colección.**

**La sintaxis para ngRepeates súper fácil. En forma de atributo, se ve así.**

**ng-repeat="element in collection"**

**Where collectiones una colección en alcance y elementes una variable utilizada para hacer referencia a cada elemento a lo largo de la iteración.**

**Además de esto, Angular expone varias variables que le permiten obtener información sobre el elemento que se está iterando actualmente.**

* **$index - El índice del elemento actual.**
* **$first - Booleano que indica si el elemento es el primero en la colección.**
* **$middle - Booleano que indica si el elemento no es ni el primero ni el último en la colección.**
* **$last - Booleano que indica si el elemento es el último en la colección.**
* **$even - Booleano que indica si el índice del elemento es par.**
* **$odd - Booleano que indica si el índice del elemento es impar.**

**Estas variables se pueden usar libremente dentro del alcance de la iteración.**

## **Ejemplo**

**Para este ejemplo, seguí adelante y definí la clase CSS oddRowdetrás de escena. oddRowcambiará el color de fondo del elemento al que se aplica a un color gris claro. Lo que vamos a hacer en este ejemplo es llenar la tabla HTML que actualmente está vacía con la lista de productos definidos en root.js.**

**Lo primero que debe hacer es definir cómo queremos que se vea una fila de la tabla. Hagamos eso ahora, completando la <tbody>etiqueta vacía con una sola fila.**

**<tbody>**

**<tr>**

**<td></td>**

**<td></td>**

**</tr>**

**</tbody>**

**Excelente. Ahora agreguemos la directiva de repetición. Es importante entender que ngRepeatse repetirá y sus contenidos para cada elemento de la colección. Por lo tanto, debemos aplicarlo al <tr>y no al <tbody>.**

**<tbody>**

**<tr ng-repeat="product in products">**

**<td></td>**

**<td></td>**

**</tr>**

**</tbody>**

**Avancemos ahora y llenemos las celdas con los valores de los objetos de la colección.**

**<tr ng-repeat="product in products">**

**<td>{{product.id}}</td>**

**<td>{{product.name}}</td>**

**</tr>**

**¡Casi termino! Ahora apliquemos la oddRowclase a cada índice impar <tr>. Necesitaremos usar la ngClassdirectiva que aprendimos en la última lección para esto.**

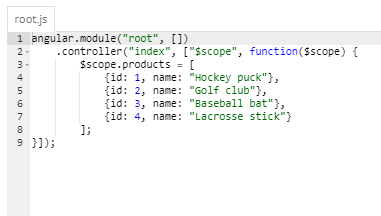
**<tr ng-repeat="product in products" ng-class="{oddRow: $odd}">**

**<td>{{product.id}}</td>**

**<td>{{product.name}}</td>**

**</tr>**

**Ya hemos terminado! ¡Ejecute el programa y vea los resultados!**

** **

# **Visibilidad**

# Visibility

## **Mostrando y escondiéndose** Showing and hiding

**Ya hemos visto que Angular le permite vincular clases CSS a elementos por medio de la ngClassdirectiva. Sin embargo, si simplemente desea mostrar u ocultar un elemento, hay un acceso directo. Las directivas ngShowy ngHidemuestran u ocultan un elemento de la vista manipulando su displaypropiedad CSS. Ambos toman una expresión como argumento que se espera que devuelva un valor booleano.**

## **Agregar y quitar** Adding and removing

**Si ngShowy ngHidecontrola la visibilidad, ngIfcontrola la existencia . Es decir, en lugar de simplemente alterar la forma en que se muestran los elementos, los ngIfagrega y elimina del DOM por completo.**

**Hay tres cosas importantes a tener en cuenta al usar ngIf.**

* **El código que realiza cálculos basados ​​en la posición de los elementos en el DOM se comportará de manera diferente con ngIfrespecto a ngShowo ngHide, ya que los dos últimos dejan elementos ocultos en el DOM.**
* **El estado no se conserva cuando Angular agrega / elimina elementos del DOM usando ngIf. Angular regenera los elementos que se eliminan y luego se leen al DOM y reaparecen en su estado inicial, no en el estado en que se encontraban cuando se eliminaron por primera vez.**
* **Elementos insertados con ngIflive en su propio ámbito que hereda del ámbito de su elemento padre. En términos simples, esto significa que aplicar ngIfa un elemento es como aplicar una segunda controllerdirectiva. Tendrá acceso a los mismos campos que en el padre, pero residirán en un objeto de ámbito separado. Por lo tanto, no puede confiar en que las variables de un ámbito permanezcan sincronizadas con las del otro ámbito. \***

## **Ejemplo**

**Para este ejemplo, vamos a conectar la visibilidad de dos <div>elementos a dos campos en nuestro $scope. Los campos ya están declarados para usted y hay dos casillas de verificación para permitirle establecer cada valor. Intente aplicar ng-show="isFirstElementVisible"al primero <div>y su ngIfequivalente al segundo <div>. Luego ejecute el ejemplo y pruébelo. Si lo desea, puede echar un vistazo al HTML correspondiente para el documento utilizando las herramientas de desarrollo de su navegador para ver cómo ngIfy ngShowocultar los elementos a los que se aplican.**

**\* Para ver esto en acción, intente pegar el siguiente código en los editores y modificar los valores del cuadro de texto.**

**HTML:**

**<div ng-controller="index">**

**<input type="text" ng-model="message" />**

**{{message}}**

**<div ng-if="true">**

**<input type="text" ng-model="message" />**

**{{message}}**

**</div>**

**</div>**

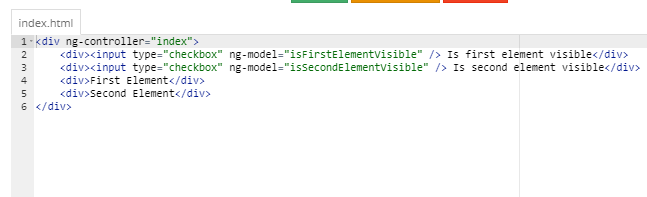
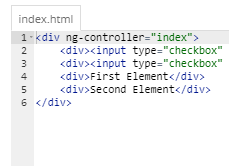
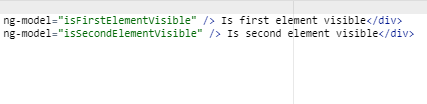
**JavaScript:**

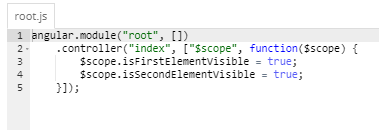
**angular.module("root", [])**

**.controller("index", ["$scope", function($scope) {**

**$scope.message = "Hello World";**

**}]);**

**  index**

****

# Services

# **Complejidad de manejo**

# Handling Complexity

**A medida que las aplicaciones angulares comienzan a hacerse más grandes y complejas, inevitablemente llega el momento en que queremos externalizar parte de la lógica contenida en nuestros controladores en objetos externos. Angular nos proporciona dos características clave que funcionan de la mano para ayudarnos a hacer exactamente esto: inyección de dependencia y servicios.**

### **Inyección de dependencia**

### Dependency injection

**La inyección de dependencia es una parte central del desarrollo de aplicaciones angulares. Es tan central, de hecho, que lo hemos estado usando desde el primer ejemplo. Cada vez que declaramos un controlador con un $scopeestamos utilizando el mecanismo de inyección de dependencia interna de Angular.**

**.controller("index", ["$scope", function ($scope) {**

**// $scope is dependency-injected!**

**}]);**

**Lo que es tan especial acerca de esto es que no invocamos la función del controlador, pasándole un argumento. Angular hace! El inyector de Angular ve desde el "$scope"elemento de matriz que queremos que el primer (y único) argumento de función sea una instancia $scope, y Angular lo hace así que este es el caso. \***

### **Servicios**

### Services

**En una aplicación angular, los servicios son componentes lógicos de la aplicación que se conectan entre sí mediante inyección de dependencia. Nos permiten externalizar la lógica de la aplicación de los controladores y encapsularla en contenedores fáciles de usar. Angular se entrega con una serie de servicios integrados que nos permiten hacer cosas como realizar solicitudes de Ajax o cambiar la URL en la barra de direcciones del navegador, pero el verdadero poder de los servicios es la capacidad de escribir la nuestra.**

**Angular define cinco tipos de servicios diferentes, o "recetas". Cada receta da como resultado un servicio que tiene características especiales que lo hacen útil para resolver un cierto tipo de problema. Las siguientes cinco lecciones cubren cada una de estas recetas en detalle.**

**¿Listo para comenzar?**

**\* Quizás se pregunte por qué necesitamos esta sintaxis de matriz algo torpe. ¿No debería ser capaz Angular de detectar que queremos un $scopebasado en el nombre del argumento de la función? La razón por la que usamos esta sintaxis es porque permite que el inyector angular funcione correctamente cuando se minimiza el JavaScript. La minificación altera los nombres de las variables, pero no las cadenas.**

# **La receta del valor**

# The Value Recipe

**La receta de valor es la receta de servicio más simple de todas. Un servicio de valor hace exactamente lo que esperarías. ¡Proporciona un valor! Si bien esto puede sonar como una característica trivial, los valores juegan un papel clave en el diseño de aplicaciones angulares flexibles y modulares.**

**La sintaxis para definir un valor es muy simple.**

**angular.module("root", [])**

**.value("message", "Hello world!");**

**Aquí hemos definido un valor llamado messageque contiene la cadena "Hello world!".**

**Pasamos el messagevalor a un controlador de la misma manera que pasamos a $scope. Agregamos la "message"cadena a la matriz del controlador y terminamos agregando un argumento correspondiente a la función del controlador.**

**angular.module("root", [])**

**.value("message", "Hello world!")**

**.controller("index", ["$scope", "message", function($scope, message) {**

**// Do something with message and/or $scope.**

**}]);**

## **¡Cuidado!**

## Watch out!

**Hay un detalle muy importante que debemos tener en cuenta antes de continuar. Cuando Angular crea servicios, los crea como objetos únicos que se almacenan en un caché. Cuando se inyecta un servicio en un método, el objeto en caché es lo que realmente se pasa como argumento. Esto significa que si su servicio devuelve un tipo de referencia, todos los métodos que toman el servicio como argumento recibirán una referencia al mismo objeto . Por lo tanto, debe tener cuidado si elige modificar el objeto, ya que los cambios que realice serán para toda la aplicación. \***

## **Módulos**

## Modules

**Si bien el código anterior está en camino de convertirse en una aplicación angular modular y extensible, en realidad podemos llevar todo esto un paso más allá. El equipo de AngularJS recomienda que las aplicaciones se dividan en varios módulos, definidos de acuerdo con la siguiente lógica.**

* **Crea un módulo para cada característica**
* **Cree un módulo para cada componente reutilizable**
* **Cree un módulo de nivel de aplicación que dependa de los módulos anteriores y contenga cualquier código de inicialización.**

**Para hacer que nuestra aplicación sea un poco más flexible, podemos dividirla en dos módulos: uno que contendrá el servicio de valor y otro que contendrá el controlador.**

**// Root.js**

**angular.module("root", [])**

**.controller("index", ["$scope", "message", function($scope, message) {**

**// Do something with message and/or $scope.**

**}]);**

**// Services.js**

**angular.module("services", [])**

**.value("message", "Hello world!");**

## **Mas inyeccion**

## More injection

**El código anterior se ve muy bien. Sin embargo, si seguimos adelante y lo ejecutamos, veremos rápidamente que tenemos un problema. El inyector de dependencia angular no podrá inyectar el messageservicio en la función del controlador. Esto se debe a que, por defecto, el inyector no inyectará componentes definidos en un módulo en componentes de otro. Más concretamente, los componentes definidos en el servicesmódulo no están disponibles por defecto para su uso en el rootmódulo en lo que respecta al inyector.**

**Para decirle al inyector que queremos hacer referencia a los componentes services, necesitamos inyectar el servicesmódulo en el rootmódulo de esta manera.**

**// Root.js**

**angular.module("root", ["services"])**

**.controller("index", ["$scope", "message", function($scope, message) {**

**// ...**

**Ahora, cuando el inyector ve que queremos inyectar messageen nuestra función de controlador, mirará en el servicesmódulo para ver si puede encontrar una coincidencia. Tener que inyectar módulos de esta manera es una gran característica. No solo deja en claro de qué depende un módulo desde la primera línea de código, sino que también ayuda a evitar colisiones de nombres.**

## **Ejemplo**

**Para este ejemplo, intente replicar el ejemplo de código anterior, mostrando messageen la página HTML proporcionada por medio de $scope. Estás bastante solo para este. Tiene dos archivos .js vacíos a su disposición, uno para cada módulo y una página HTML en la que tendrá que hacer referencia a su controlador y a su módulo raíz . Los pasos exactos que deberá seguir son los siguientes.**

* **Declare un módulo de servicios services.jsy cree un servicio de valor denominado message.**
* **Declare un módulo raíz en root.js, inyectando su servicesmódulo recién creado .**
* **Cree un controlador en su módulo raíz. El controlador debe tomar una $scopey una instancia de su messageservicio.**
* **Consulte su módulo raíz en la página HTML.**
* **Consulte su controlador en la página HTML.**
* **Muestre el mensaje en la página usando la sintaxis de llaves dobles.**

**Si puede completar este ejemplo con éxito, está en camino de desarrollar aplicaciones angulares completas. Si tiene problemas, no dude en hacer clic en "Estoy atascado". botón para ver cómo se hace e intente nuevamente.**

**\* Esto no es necesariamente algo malo. Este comportamiento no solo mejora el rendimiento, sino que también le permite compartir datos entre controladores si es necesario.**

# **La receta de la fábrica**

# The Factory Recipe

**Los valores son geniales, pero ¿no sería bueno si pudiéramos inyectar una función en lugar de una variable? Este es el propósito de la receta de fábrica. Las fábricas se declaran utilizando la siguiente sintaxis.**

**angular.module("services", [])**

**.factory("myFactory", [function () {**

**// Logic goes here.**

**}]);**

**También podemos inyectar dependencias en una fábrica como lo hacemos con los controladores.**

**.factory("myFactory", ["dependency", function (dependency) {**

**// ...**

## **Ejemplo**

**Para demostrar el uso de la receta de fábrica, vamos a escribir una aplicación que tome un número y lo cuadre. Para hacer las cosas un poco más interesantes, el número al cuadrado será proporcionado por un servicio de valor.**

**Comencemos definiendo nuestro servicio de valor en un nuevo módulo en services.js**

**angular.module("services", [])**

**.value("factor", 6);**

**Ahora creemos un servicio "cuadrado" que cuadre su argumento, que será el número recuperado del factorservicio.**

**angular.module("services", [])**

**.value("factor", 6)**

**.factory("square", ["factor", function (factor) {**

**return factor \* factor;**

**}]);**

**Ahora agreguemos esta lógica a nuestro controlador y asignemos el resultado a la productvariable. ¡No olvides hacer referencia al servicesmódulo!**

**angular.module("root", ["services"])**

**.controller("index", ["$scope", "square",**

**function ($scope, square) {**

**$scope.product = square;**

**}**

**]);**

**Observe cómo usamos squarecomo si fuera una variable. No lo llamamos como si fuera una función.**

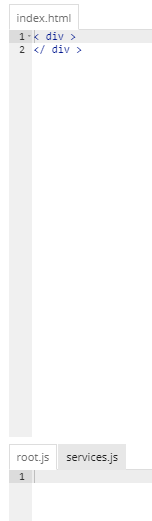
**Terminemos dirigiendo nuestra atención al HTML. Haga referencia al controlador y visualice producten el vacío <div>.**

**<div ng-controller="index">**

**{{product}}**

**</div>**

**Ya hemos terminado. ¡Ejecute el programa y pruébelo!**

****

# **La receta de servicio**

# The Service Recipe

**La receta de servicio es un paso adelante de los valores y las fábricas, en la medida en que se usa para exponer objetos. Su sintaxis se ve así.**

**angular.module("services", [])**

**.service("myService", ["dependency", MyObject]);**

**// Declared elsewhere...**

**function MyObject(dependency) {**

**this.value = dependency.value;**

**}**

**Tenga en cuenta que esto es equivalente a una fábrica que devuelve a MyObject.**

**angular.module("services", [])**

**.factory("myService", ["dependency", function (dependency) {**

**return new MyObject(dependency);**

**}]);**

**// Declared elsewhere...**

**function MyObject(dependency) {**

**this.value = dependency.value;**

**}**

**La verdadera ventaja del patrón de servicio es su sintaxis más limpia.**

## **Ejemplo**

**Para practicar el uso de la receta de servicio, ampliaremos nuestro ejemplo de la última lección. En lugar de escribir una fábrica que cuadre un número, vamos a escribir un servicio que multiplique dos números juntos. Mostraremos el resultado de la multiplicación en nuestra plantilla.**

**El constructor del objeto en el que basaremos nuestro servicio ya está escrito para usted multiplier.js. Se parece a esto.**

**function Multiplier(valueFactor) {**

**this.multiply = function (controllerFactor) {**

**return valueFactor \* controllerFactor;**

**};**

**}**

**Como puede ver, se necesitan dos números. Un número se pasa al constructor, y el otro se pasa a la multiplyfunción. Como en el último ejemplo, vamos a inyectar el argumento constructor por medio de un valor.**

**Vamos y agreguemos nuestra declaración de servicio a nuestro servicesmódulo. Ya he declarado el factorvalor para ti.**

**angular.module("services", [])**

**.value("factor", 6)**

**.service("multiplier", ["factor", Multiplier]);**

**Hora de escribir el controlador. Declare un nuevo controlador en el rootmódulo. Inyecte el multiplierservicio así como también el $scope. ¡No te olvides de inyectar servicesen la declaración del módulo!**

**angular.module("root", ["services"])**

**.controller("index", ["$scope", "multiplier",**

**function ($scope, multiplier) {**

**}]);**

**Ahora, agregue la llamada a multiply, pasando el número de su elección. Asignar el resultado a $scope.product.**

**ngular.module("root", ["services"])**

**.controller("index", ["$scope", "multiplier",**

**function ($scope, multiplier) {**

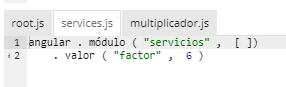
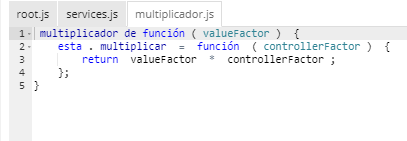
**$scope.product = multiplier.multiply(2);**

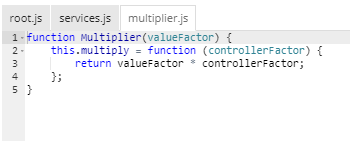
**}]);**

**Para finalizar, haga referencia al controlador en la plantilla y visualice producten el vacío <div>.**

**<div ng-controller="index">{{product}}</div>**

**Eso es todo lo que necesitamos hacer. ¡Inicia la aplicación y pruébala!**

**  **

****

# **La receta del proveedor**

# The Provider Recip

**Imagine que ha escrito un servicio angular que se puede utilizar para recuperar datos meteorológicos de una API pública. El objeto devuelto por el servicio es el siguiente.**

**{**

**function Get3DayForecast(city, apiKey) {**

**// Implementation**

**}**

**}**

**Ahora, aunque este servicio puede ser bastante útil, hay un punto de redundancia en su diseño. ¿No sería bueno no tener que pasar la clave API cada vez que queramos hacer una llamada?**

**Si este servicio fuera solo para nuestro uso, simplemente podríamos incrustar la clave directamente en el objeto.**

**{**

**var apiKey = "A1B2C3";**

**function Get3DayForecast(city) {**

**// Implementation**

**}**

**}**

**Sin embargo, si quisiéramos compartir este servicio con otros, esta no sería una muy buena solución. Una mejor manera sería permitir que el consumidor del servicio establezca la clave de una vez por todas antes de usarla. Esto es lo que los proveedores nos permiten hacer.**

## **Ejemplo**

### **Definiendo el proveedor**

**La mejor manera de ver cómo funcionan los proveedores es a través de un ejemplo. Nuestro ejemplo será extremadamente simple. Vamos a definir un proveedor que servirá un Messageobjeto. El objeto se define de la message.jssiguiente manera.**

**function Message(text) {**

**this.text = text;**

**}**

**El problema aquí es que, al igual que la clave API en el ejemplo anterior, configuraremos el texto del mensaje solo una vez en toda nuestra aplicación. Cada vez que solicitemos un mensaje, nos será devuelto con su texto ya configurado.**

**Veamos el código que nos permitirá hacer esto. Agregue este código a la servicesdefinición del módulo en services.js.**

**.provider("message", [function () {**

**var text = null;**

**this.setText = function (textString) {**

**text = textString;**

**};**

**this.$get = [function () {**

**return new Message(text);**

**}];**

**}]);**

**Veamos qué está pasando aquí. Lo primero que tenemos es una textvariable. Esto mantendrá el texto que se utilizará para crear una instancia de nuestro Message.**

**Lo segundo que tenemos es una función llamada setText. setTextes la función que llamaremos para configurar nuestro proveedor. Le pasaremos una cadena que se almacenará en la textvariable, solo para ser finalmente utilizada en la llamada al Message()constructor.**

**Finalmente, tenemos un método llamado $get. Como probablemente haya imaginado debido a la guía $, $getes una función que tiene un significado especial para Angular. Cuando inyectamos a nuestro proveedor en una función por medio del nombre del proveedor ( "message"), Angular llamará $getpara recuperar el objeto a inyectar. En este caso, inyectará una Messageinstancia con la que se ha creado una instancia text.**

### **Usando el proveedor**

**Ahora que nuestro proveedor está definido, usémoslo para mostrar un mensaje en nuestra página. Comencemos configurando nuestro proveedor. Hacemos esto así.**

**angular.module("root", ["services"])**

**.config(["messageProvider", function (messageProvider) {**

**messageProvider.setText("Hello world!");**

**}])**

**Esta sintaxis es bastante fácil de entender. Llamamos setText, en la config()función, pasando el mensaje con el que queremos instanciar nuestro Messageobjeto. Lo único que puede parecer fuera de lugar es la "messageProvider"cuerda. ¿De dónde viene? Y ¿por qué estamos inyectando "messageProvider"en config()y no "message"en su lugar?**

**La razón es porque "message"no se refiere al objeto que definimos en la llamada a provider(). "message"refiere el objeto definido por la $getfunción del proveedor . Para inyectar el objeto definido en la llamada del proveedor, en el que declaramos la setText()función, inyectamos "messageProvider".**

**Para ser claros, no definimos lo que "messageProvider"significaba en cualquier parte de nuestro código. Es simplemente una convención angular que se "[provider name]Provider"refiere al resultado de la función del proveedor, y que se "[provider name]"refiere al $getresultado.**

**Ahora que nuestro proveedor está configurado para devolver un valor Messagecuya textconfiguración se establece en "Hello world!", lo único que queda por hacer es inyectar una instancia en la función de nuestro controlador y mostrar el resultado.**

**angular.module("root", ["services"])**

**.config(["messageProvider", function (messageProvider) {**

**messageProvider.setText("Hello world!");**

**}])**

**.controller("index", ["$scope", "message",**

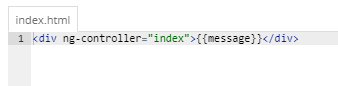
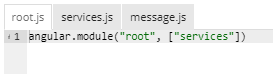
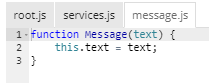
**function ($scope, message) {**

**$scope.message = message.text;**

**}**

**]);**

**Cuando se ejecuta, la aplicación debería mostrar "¡Hola, mundo!" en la pantalla.**

**   **

# **La receta constante**

# The Constant Recipe

**En la última lección, vimos cómo se pueden configurar los proveedores usando la configfunción. En la aplicación de ejemplo que codificamos, la configllamada se veía así.**

**.config(["messageProvider", function (messageProvider) {**

**messageProvider.setText("Hello world!");**

**}])**

**Esto funcionó bien para nuestro ejemplo. Sin embargo, podemos imaginar fácilmente un escenario en el que preferiríamos inyectar en "Hello world!"lugar de codificarlo en la función.**

**Si intentara intentar esto, podría intentar definir un servicio de valor e inyectarlo en la configfunción.**

**.value("messageText", "Hello world!");**

**Sin embargo, pronto te darás cuenta de que esto no funciona.**

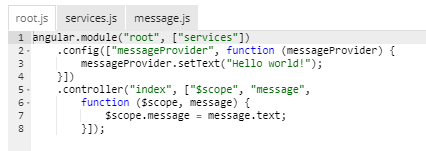
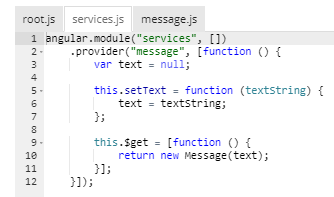
**La razón es que las config()funciones se ejecutan antes de que los servicios se instancian y se registran en Angular. Esto tiene sentido, porque todo el propósito de las config()funciones es modificar el comportamiento de los servicios. Como resultado, no puede inyectar un servicio de valor en una config()función porque, en lo que respecta a Angular, ¡todavía no existe!**

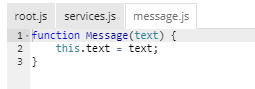
**Este es el problema que resuelven las constantes. Las constantes se definen durante la fase de configuración de una aplicación angular. Esto significa que, a diferencia de otros servicios, pueden inyectarse en config()funciones. La sintaxis para definir constantes es similar a la sintaxis para definir valores.**

**.constant("messageText", "Hello world!");**

## **Ejemplo**

**El código de inicio en los editores es exactamente el mismo código que escribimos en la lección sobre proveedores. Para este ejemplo, intente crear una constante llamada que messageTextcontenga la cadena "Hello constant!"e inyectarla en la config()función para crear una instancia del objeto del mensaje. Al igual que en el ejemplo de los proveedores, Hello constant!debe mostrarse en pantalla una vez que haya terminado.**

**  **

****

# $watch

**Echa un vistazo al código en los editores. Verá que la interfaz de usuario contiene un cuadro de texto en el que se ingresa el número 6, y que el valor del cuadro de texto está vinculado a la variable $scope.factor. Al lado del cuadro de texto se muestra $scope.product, que se calcula en la función del controlador como $scope.factor \* 2.**

**Intenta ejecutar este código. Todo se muestra normalmente, con el cuadro de texto que contiene el número 6 de forma predeterminada y el número 12 que se muestra al lado. Ahora intente modificar el valor en el cuadro de texto. ¡Te sorprenderá ver que $scope.productno se actualiza!**

## **¿Pero por qué?**

**La razón por la que esto sucede es porque $scope.productse inicializa al valor devuelto por $scope.factor \* 2cuando el controlador aumenta el modelo. Se no inicializado en el resultado de la función durante la ejecución del programa . Cuando $scope.factorse cambia, no existe un mecanismo para actualizar en $scope.productconsecuencia.**

## **Una visión general de $ watch**

**Aquí es donde $watchentra la función. $watchNos proporciona una forma de mantener actualizados los valores calculados cuando cambian los valores de los que dependen. La sintaxis de se $watchve así.**

**$scope.$watch([expression returning watched value],**

**[change handler],**

**[objectEquality?]);**

**Aclaremos esto un poco. El primer argumento básicamente apunta al valor a mirar. Esto puede ser una cadena que contiene el nombre de una variable en el alcance, o una función que devuelve la variable en sí.**

**El segundo argumento toma una función que se ve así.**

**function (newValue, oldValue) {**

**// Take action.**

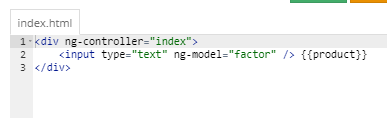
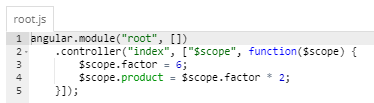
**}**

**Aquí es donde se realiza el trabajo real. En nuestro ejemplo, aquí es donde actualizaríamos $scope.productsegún el valor de newValue.**

**El tercer argumento es un valor booleano que, cuando se establece en verdadero, compara los objetos por igualdad en lugar de referencia. Este argumento puede omitirse, y a menudo, se omite en lugar de establecerse en false.**

## **Ejemplo**

**Ahora depende de ti retomar donde lo dejé. Use una $watchexpresión para actualizar $scope.productcuando se cambie el valor en el cuadro de texto. Específicamente, deberá mirar $scope.factory configurar $scope.product2 veces factorcada vez que factorse cambie.**

** **

# **Filters**

**En pocas palabras, los filtros son funciones que transforman un valor en otro. Se usan comúnmente en dos escenarios diferentes.**

* **Para transformar un valor para fines de visualización. Por ejemplo, tomar el número 6 y transformarlo en $ 6.00 podría lograrse muy fácilmente con la ayuda de un filtro.**
* **Para filtrar colecciones. Se podría escribir un filtro para tomar una matriz de nombres y devolver una matriz más pequeña, filtrada en función de una condición proporcionada por el usuario.**

## **Sintax**

**Se declara un filtro utilizando el filtermétodo El método toma dos argumentos: un nombre y una función que devuelve la función que realmente hará el filtrado. Esto es importante para recordar. No es la función pasada al filtermétodo la que hace el filtrado, sino la función devuelta por la función pasada al método.**

**Así es como se ve una definición de filtro.**

**angular.module("root", [])**

**.filter("doSomething", function () {**

**return function(input) {**

**// Transform the input and return a value.**

**}**

**});**

**La función que realiza el filtrado puede tomar tantos argumentos como desee, sin embargo, el primero siempre está reservado para la entrada.**

**Los filtros se pueden usar en controladores o en vistas. Para usar un filtro en un controlador, simplemente lo inyecta como lo haría con cualquier otro componente. Sin embargo, hay una pequeña trampa. Debe agregar "Filtro" al nombre del filtro al especificarlo para el inyector. Por lo tanto, para inyectar el doSomethingfiltro definido anteriormente en un controlador, se requiere la siguiente sintaxis.**

**.controller("index", ["doSomethingFilter", function(filter) {**

**// ...**

**});**

**Usar un filtro en una vista es un poco diferente. Aprenderemos cómo hacer esto a modo de ejemplo.**

## **Ejemplo**

**Para este ejemplo, vamos a definir un filtro que redondeará un valor numérico a un cierto número de decimales.**

**Comencemos definiendo el filtro en filters.js.**

**angular.module("filters", [])**

**.filter("round", function () {**

**return function(input, precision) {**

**return input ?**

**parseFloat(input).toFixed(precision) :**

**"";**

**};**

**});**

**Observe cómo estamos haciendo que este filtro sea más flexible al permitir que el usuario especifique una precisión.**

**Ahora integremos este filtro en nuestra vista. Por el momento, nuestra vista contiene un único cuadro de texto cuyo valor se asigna a la $scope.valuevariable. Modifiquemos nuestra plantilla para que salga $scope.valuedirectamente a la página.**

**<p>Value: <input type="text" ng-model="value" /></p>**

**<p>{{value}}</p>**

**Ahora apliquemos el filtro. Para aplicar un filtro a un valor en una plantilla, utilizamos la siguiente sintaxis.**

**{{ expression | filter:arg1:arg2... }}**

**Por el bien de este ejemplo, digamos que queremos redondear nuestro valor a dos decimales. Por lo tanto, debemos reemplazarlo {{value}}con lo siguiente.**

**{{value | round:2}}**

**Eso es todo lo que necesitamos hacer. Anímate y pruébalo.**

### **Encadenamiento de filtros** Chaining filters

**Definamos otro filtro que antepondrá $a nuestro número redondeado. Agregue esta definición a la filters.jsderecha después del roundfiltro.**

**.filter("dollars", function () {**

**return function(input) {**

**return input ? "$" + input : "";**

**};**

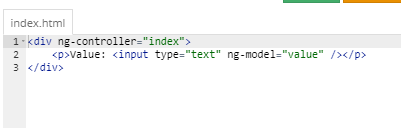
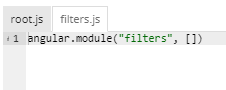
**});**

**Ahora integremos esto en nuestra plantilla. Angular nos permite encadenar filtros juntos usando a |. Por lo tanto, para encadenar nuestro dollarsfiltro a nuestro roundfiltro, usamos la siguiente sintaxis.**

**{{value | round:2 | dollars}}**

**Tenga en cuenta que el orden es importante en este caso. Queremos redondear nuestro número antes de aplicarlo $, no después.**

**¡Eso es! Ahora, cuando ingresamos un valor numérico en el cuadro de texto, nuestros filtros lo transforman en su equivalente monetario.**

**  **

# **Validación de datos**

# Data Validation

**La validación de formularios es un componente clave de todas las aplicaciones web, excepto las más simples. AngularJS nos cubre una vez más al proporcionar un marco integrado para decirles a los usuarios si los datos que ingresaron en un formulario son válidos o no. Lo hace aplicando un subconjunto de las siguientes cuatro clases CSS.**

* **ng-valid**
* **ng-invalid**
* **ng-pristine**
* **ng-dirty**

**Como desarrolladores, podemos seguir adelante y escribir reglas CSS para aplicar diferentes estilos a campos y formularios dependiendo de su estado.**

## **Definiendo validez**

**Las clases CSS que expone Angular son extremadamente útiles, pero para usarlas necesitamos decirle a Angular cómo validar realmente nuestros datos. Para esto, Angular se basa en el marco de validación de datos de HTML5. Aquí no entraremos en detalles sobre la validación de datos HTML5, pero si está interesado en lo que tiene para ofrecer,**[**este es un excelente lugar para comenzar**](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint_validation)**.**

## **Ejemplo**

**Para este ejemplo, vamos a validar los dos campos de entrada definidos en el formulario en nuestra plantilla. Vamos a hacer esto aplicando la siguiente lógica de validación.**

* **El campo de nombre es obligatorio.**
* **El campo de correo electrónico es opcional, pero debe ser un correo electrónico válido si se proporciona.**

**Además de esto, aplicaremos las siguientes pautas de diseño.**

* **Los campos solo deben tener estilo si han sido tocados por el usuario.**
* **Los campos deben estar diseñados con un borde verde cuando vaild.**
* **Los campos deben tener un estilo con un borde rojo cuando no sea válido.**
* **El formulario en sí no debe ser diseñado de ninguna manera.**

**Vamos a codificar. Lo primero que debemos hacer es asignar un $scopevalor a cada campo de formulario. Los nombres no importan mucho porque realmente no vamos a enviar estos datos, pero Angular no validará los campos que no están asignados al, $scopepor lo que este es un primer paso lógico.**

**<input type="text" ng-model="name" />**

**<input type="text" ng-model="email" />**

**Ahora escribamos nuestros estilos. Recuerda nuestras pautas de diseño. Los campos que están sucios y válidos tienen un borde verde, y los que están sucios e inválidos obtienen un borde rojo. Los campos limpios no tienen estilo. También necesitamos declarar explícitamente que queremos diseñar inputetiquetas, porque no queremos que también tengamos formestilo.**

**input.ng-dirty.ng-valid {**

**border-color: green;**

**}**

**input.ng-dirty.ng-invalid {**

**border-color: red;**

**}**

**Excelente. Todo lo que nos queda por hacer es aplicar nuestra lógica de validación. Comencemos haciendo que el campo de nombre sea obligatorio agregando el requiredatributo booleano.**

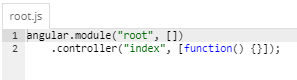
**<input type="text" ng-model="name" required />**

**Ahora solicitemos que el valor del campo de correo electrónico sea en realidad una dirección de correo electrónico válida. Hacemos esto cambiando typela entrada de texta email.**

**<input type="email" ng-model="email" />**

**Ahora ejecute el código e intente jugar con los valores en cada uno de los diferentes campos. El borde de cada campo debe cambiar dependiendo de si se han ingresado datos y si los datos que contiene son válidos de acuerdo con las reglas que hemos definido.**

**Nota: Esto probablemente no sea necesario para muchos visitantes de este sitio, pero la validación de datos angular es una solución de validación del lado del cliente. Es fundamental que todos los datos enviados a una aplicación web, incluidos los validados en el cliente, se validen en el servidor.**

** **

# **Ajax con $resource**

**Las llamadas Ajax son una parte integral de prácticamente todas las aplicaciones web modernas. Una vez más, Angular no decepciona y nos da no una sino dos API diferentes para manejar las solicitudes de Ajax. En esta lección vamos a ver el nivel superior de los dos, que se conoce como $resource.**

## **Introducción a $resource**

**Como se mencionó hace un momento, $resourcees una API de Ajax de alto nivel. En lugar de proporcionar un control detallado sobre cómo se realizan las solicitudes, $resourcehace una serie de suposiciones sobre cómo debería funcionar un punto final de servicio web, y con una información mínima del desarrollador construye un objeto que se puede utilizar para realizar una amplia variedad de solicitudes a Servicio.**

## **Los basicos**

**La mejor manera de entender cómo $resourcefunciona es ver cómo se hace un objeto de recursos y de qué es capaz. En su forma más simple, podemos crear una instancia usando el siguiente código.**

**var resourceObject = $resource(url);**

**Esto puede no parecer mucho, pero créalo o no, resourceObjectya contiene las siguientes funciones.**

* **get([params], [success], [error]) - Hace una llamada GET estándar.**
* **save([params], postData, [success], [error]) - Hace una llamada POST estándar.**
* **query([params], [success], [error]) - Realiza una llamada GET estándar y devuelve la respuesta como una matriz.**
* **remove([params], postData, [success], [error]) - Realiza una llamada BORRAR estándar.**

**Repasemos brevemente los argumentos de estos métodos. successy errorson funciones de devolución de llamada que se ejecutarán en sus respectivos casos. No hay sorpresas allí. postData, donde corresponda, son simplemente los datos que se publicarán en el servidor. De nuevo, no hay sorpresas.**

**Lo que podría ser un poco menos claro es el propósito del paramsargumento. Para entender para qué se utiliza, primero debemos volver a nuestra $resourcellamada inicial . Como vimos hace un momento, la cantidad mínima de datos necesarios para crear un $requestobjeto es una URL. Sin embargo, esta URL puede contener marcadores de posición para los valores de los parámetros que luego se pueden reemplazar cuando se realizan llamadas Ajax. Los marcadores de posición se integran en las URL de la siguiente manera.**

**`http://www.mysite.com/users/:paramName`**

**Si estuviéramos haciendo una GETsolicitud usando un objeto de recurso configurado para esta URL y paramNamequisiéramos establecer un valor para , todo lo que tendríamos que hacer es pasar el siguiente objeto como primer argumento para el getmétodo del objeto .**

**{ paramName: value }**

**Con estos cuatro métodos y las características que proporcionan, las solicitudes CRUD entre el cliente y el servidor se vuelven fáciles de configurar y ejecutar. ¡Pero hay más! Cuando un objeto de recurso recibe una respuesta del servidor, deserializa automáticamente la respuesta en un objeto JavaScript, agregando los dos métodos siguientes en el proceso.**

* **$save([params], [success], [error]) - PUBLICA el objeto de regreso al punto final, guardando los cambios**
* **$remove([params], [success], [error]) - DELETE el objeto**

**Tenga en cuenta que estos son esencialmente los mismos métodos que existen en el $resourceobjeto, excepto que no toman un postDataparámetro. Estos métodos están realmente personalizados para el objeto en el que están configurados. Eso significa que si modifica un objeto y luego llama $save, el objeto se serializará y se PUBLICARÁ en el servidor donde se puede actualizar. Lo mismo se aplica $remove, lo que hace una solicitud DELETE en su lugar. Esta es realmente una característica sorprendente que ahorra mucho tiempo y esfuerzo por parte del desarrollador.**

## **Ejemplo**

**Para este ejemplo, vamos a utilizar $resourcepara crear un servicio que realizará las siguientes dos acciones.**

1. **Recupere un "usuario" ficticio de un punto final RESTful.**
2. **PUBLICA una versión modificada del usuario de vuelta al punto final.**

**Como esta lección realmente trata sobre el uso de la $resourceAPI, todo lo que no está relacionado $resourceya se ha solucionado. Comenzamos con una interfaz HTML básica que contiene dos campos, uno para un nombre y otro para un correo electrónico. También contiene un botón para recuperar al usuario desde el punto final RESTful y un botón para guardarlo de nuevo. Nuestro objetivo para esta lección será completar las funciones getUsery postUseradjuntas a estos dos botones para que esta interfaz sea realmente funcional.**

### **Configuración**

**Antes de que podamos comenzar a usar $resource, se requiere un poco de configuración. Lo primero a tener en cuenta es que $resourceno forma parte de la API principal de AngularJS. En cambio, el equipo angular ha decidido incluir $resourceen un módulo separado, en un archivo de código separado. Para este ejemplo, el archivo de código se ha incluido automáticamente, pero si lo usa $resourceen sus propias aplicaciones, naturalmente necesitará agregarle una referencia. El nombre del archivo a incluir es angular-resource[.min].js. El archivo se puede descargar en el sitio web de AngularJS o se puede hacer referencia directamente desde Google CDN.**

**Como $resourcevive en su propio módulo, necesitamos inyectar este módulo en el nuestro. Para hacer esto, agregue "ngResource"a la matriz vacía en la función de declaración del módulo.**

**Ahora, inyecte $resourceen el controlador agregándolo a la controllerllamada, tanto como una dependencia y como un argumento de función del controlador.**

**.controller("index", ["$scope", "$resource", function($scope, $resource) {**

**Eso es todo lo que necesitamos hacer. ¡Ahora comencemos a usarlo!**

### **Código**

**Comenzaremos definiendo nuestro objeto de recurso. Como lo vamos a usar en los métodos getUsery postUser, debemos definirlo directamente en la función del controlador. La URL al punto final es la siguiente.**

**http://www.learn-angular.org/ResourceLesson/Users/**

**Para recuperar un usuario determinado del servidor, nuestro servicio RESTful requiere que agreguemos una ID al final de la URL. Nuestro servicio devolverá al usuario correspondiente a la identificación que proporcionamos. Implementaremos esta integración de ID en la url usando un marcador de posición, de la siguiente manera.**

**http://www.learn-angular.org/ResourceLesson/Users/:id**

**Avancemos ahora y usemos esta url para crear una instancia del objeto de recurso.**

**var users = $resource("http://www.learn-angular.org/ResourceLesson/Users/:id");**

**Ahora implementaremos nuestra getfunción. El servicio web que escucha en la url proporcionada contiene un usuario ficticio con el ID de 1. Por lo tanto, para recuperar este usuario y vincularlo a nuestro modelo, necesitamos agregar la siguiente línea de código al cuerpo de la función.**

**$scope.user = users.get({id: 1});**

**Cuando se ejecuta, esta línea de código se insertará 1en nuestra URL de solicitud, enviará la solicitud GET al servidor, deserializará los datos devueltos en un objeto JavaScript y lo asignará a la $scope.uservariable. Los campos de nombre y correo electrónico en nuestro HTML están preconfigurados para enlazarse con los campos de nombre y correo electrónico en el objeto devuelto por el servicio web.**

**Ahora implementemos nuestra función de guardar. Debido a que estaremos salvando a un usuario que ya hemos recuperado del servicio web, esta tarea es extremadamente simple. Todo lo que tenemos que hacer es llamar al $save()método que Angular ha adjuntado al objeto de usuario para nosotros. Agregue el siguiente código a la postUserfunción.**

**var response = $scope.user.$save(function () {**

**alert("User saved!");**

**});**

**Aquí simplemente hemos pasado una devolución de llamada que nos alertará si el usuario se ha guardado correctamente.**

**Eso es todo lo que necesitamos hacer. Ejecute el código e intente lo siguiente.**

1. **Haga clic en el botón "Obtener usuario". Los dos campos del formulario deben llenarse con datos de usuario ficticios.**
2. **Modifique los datos y haga clic en el botón "Publicar usuario". La devolución de llamada que pasó al $savemétodo debería ejecutarse y debería aparecer una alerta en la pantalla indicando que el usuario ha sido guardado.**
3. **Borre los campos del formulario y haga clic en el botón "Obtener usuario" una vez más. Los campos deben llenarse con los datos que guardó hace un momento.**

## **$ resource en el servidor**

**Debido a que $resourcehace muchas suposiciones sobre cómo debería funcionar un punto final de servicio web, a menudo resulta ser el más útil cuando usted es quien tiene el control del servicio web. Si está escribiendo un servicio web para usar $resource, tenga en cuenta las siguientes pautas.**

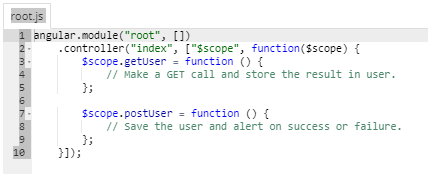
* **El servicio debe ser RESTful. La acción que debe tomar el código del lado del servidor debe estar determinada por el verbo HTTP utilizado en la solicitud.**
* **Se debe utilizar la misma URL para todas las solicitudes. Se pueden agregar segmentos de parámetros o argumentos de cadena de consulta para solicitudes GET.**
* **JSON debe usarse como el estándar de serialización preferido. Como vimos anteriormente, Angular deserializará automáticamente las respuestas JSON en objetos sin ninguna acción adicional de su parte.**

## **Más sobre $resource ...**

**$resourcees una API extremadamente poderosa. Si bien en esta lección hemos podido recorrer los conceptos básicos, $resourcecontiene muchas más características de las que razonablemente podríamos cubrir aquí. Estas características incluyen:**

* **Métodos de solicitud personalizados**
* **Valores de parámetros predeterminados**
* **Lógica de (des) serialización personalizada**
* **Soporte de caché**
* **y mucho más...**

[**Todo el conjunto de características se documenta aquí**](https://docs.angularjs.org/api/ngResource/service/$resource)**en la referencia de la API AngularJS. Le recomiendo que se tome un momento y lo vea.**

****

**http://www.learn-angular.org/#!/lessons/the-essentials**