

## Desarrollo de aplicaciones web con AngularJS



COLEGIO OFICIAL DE  
INGENIEROS EN INFORMÁTICA  
DE LA COMUNIDAD VALENCIANA



El curso de Desarrollo con AngularJS aporta las herramientas necesarias para crear sitios utilizando las directivas que nos provee el framework de manera de enriquecer nuestro HTML, creando código semántico, utilizando el patrón de Diseño MVC orientado a crear lo que se conoce como SPA "Simple Page Application" o "Aplicaciones de una sola página", que son sitios web donde la página no se recarga, no existe una navegación de una página a otra totalmente diferente, sino que se van intercambiando las "vistas". Técnicamente podríamos decir que, al interactuar con el sitio, el navegador no recarga todo el contenido, sino únicamente vistas dentro de la misma página.

**Objetivos:**

- Conocer el patrón MVC y su aplicación dentro de AngularJS.
- Conocer las aplicaciones del tipo SPA y aprendan a generarlas.
- Conocer las directivas de AngularJS.
- Conocer el sistema de routing y el uso de Ajax.

**Requisitos previos:**

- Conocimientos de Javascript.
- Preferentemente conocimientos de JQuery.
- Conocimientos de Patrón de diseño MVC.

### **Tema 1: Introducción a AngularJS**

- Desarrollo de aplicaciones SinglePage.
- Qué es AngularJS.
- Comparativa con otros frameworks :Backbone,EmberJS.

### **Tema 2: Desarrollo con AngularJS**

- Patrón de arquitectura MVC.
- Estructura de una aplicación.
- Herramientas de desarrollo.
- Gestión de paquetes con npm y bower.

### **Tema 3: MVC (Modelo-Vista-Controlador) en AngularJS**

- Enlace de datos (Data Binding).
- Definición de controladores para la aplicación.
- Usos y funcionamiento del scope de AngularJS.
- Uso de Bootstrap

### **Tema 4: Vistas y filtros**

- Definición y navegación de rutas.
- Filtros predefinidos.
- Creación de filtros personalizados.

### **Tema 5: Formularios y directivas**

- Creación de formularios y validación de datos.
- Directivas y expresiones comunes de AngularJS.
- Creación de directivas personalizadas.

### **Tema 6: Comunicación con el servidor**

- Servicios HTTP con AngularJS.
- Integración con servicios REST.
- Mocks

## **Herramientas básicas**

Angular v.1.5.8(descargar)

<https://angularjs.org/>

NodeJS(instalar en el SO)

<https://nodejs.org/>

Editor brackets(instalar en el SO)

<http://brackets.io/>

Chrome Browser(instalar en el SO)

<https://www.google.com/chrome/browser/desktop/index.html>

## Tema 1: Introducción a AngularJS

### Desarrollo de aplicaciones SinglePage

---

#### Concepto de SPA

Según la wikipedia [http://en.wikipedia.org/wiki/Single\\_page\\_application](http://en.wikipedia.org/wiki/Single_page_application)

“Una Single-Page Application (SPA), también conocido como Single-Page Interface (SPI), es una aplicación web o website que está dentro de una única página web con el objetivo de proporcionar una experiencia de usuario más fluida similar a una aplicación desktop.

En una SPA, o todo el código (HTML, JavaScript y CSS) está cargado en una única página, o los recursos apropiados son cargados dinámicamente y añadidos a la página cuando son necesarios, normalmente como respuesta a acciones del usuario. La página no se recarga en ningún punto del proceso, ni transfiere el control a ninguna otra página, aunque las tecnologías web modernas (como las incluidas en HTML5) puede proporcionar la percepción y navegabilidad a páginas separadas lógicamente en la aplicación. La interacción con SPA usualmente incluye comunicación dinámica con un webserver.”

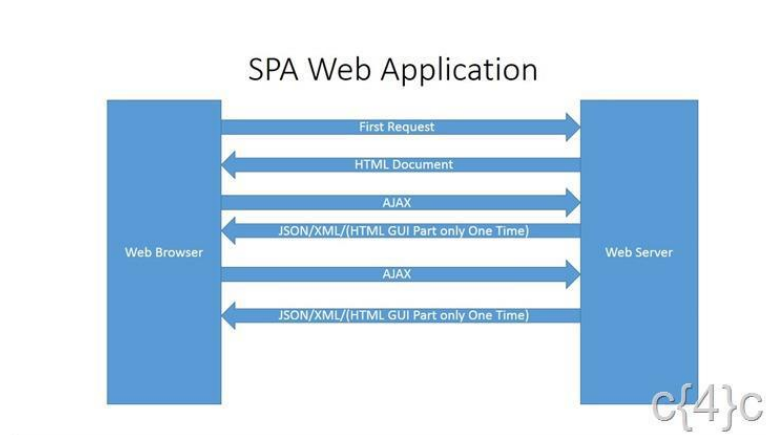
En concreto en angular se puede implementar una arquitectura SPA utilizando la carga bajo demanda de forma que la carga inicial de la página contenedora o principal es más rápida y solo se cargan las páginas conforme se vayan necesitando. Una vez una página está cargada, esta es almacenada en la caché y ya nunca se va a volver a solicitar a no ser que se hagan cambios a propósito que modifiquen el comportamiento por defecto.

Esto tiene la parte buena y la parte mala. Por un lado agiliza y da la sensación de estar trabajando como una aplicación desktop, pero también obliga a tener las páginas completamente desacopladas de los datos. Si el contenido de alguna página dependiera de los datos debería eliminarse a mano de la caché para que la próxima vez que se intente mostrar este html se vuelva a recargar.

Angular promueve y usa patrones de diseño de software. En concreto implementa lo que se llama MVC, aunque en una variante muy extendida en el mundo de Javascript que luego comentaremos con más detalle. Básicamente estos patrones nos marcan la separación del código de los programas dependiendo de su responsabilidad. Eso permite repartir la lógica de la aplicación por capas, lo que resulta muy adecuado para aplicaciones de negocio y para las aplicaciones SPA (Single Page Application).

Las SPA o "Aplicaciones de una sola página", son sitios web donde los usuarios perciben una experiencia similar a la que se tiene con las aplicaciones de escritorio. En este tipo de sitios la página no se recarga, no existe una navegación de una página a otra totalmente diferente, sino que se van intercambiando las "vistas".

Técnicamente podríamos decir que, al interactuar con el sitio, el navegador no recarga todo el contenido, sino únicamente vistas dentro de la misma página.



Una **single-page web application** (en adelante SPA), es una aplicación web que se ejecuta completamente en una única página web, con el objetivo de proporcionar una experiencia más fluida y similar a la que nos encontraríamos en una aplicación de escritorio.

En una aplicación SPA, todos los datos necesarios, como el HTML, CSS o JavaScript, se cargan y añaden en la página cuando es necesario, normalmente respondiendo a acciones del usuario. En ningún momento del proceso veremos una recarga total de la página. Para esto, como os imaginaréis a lo largo del proceso de ejecución de una aplicación SPA existe una comunicación con el servidor en segundo plano.

## Qué es AngularJS

---

AngularJS es un framework JS para la parte cliente o Frontend de una aplicación web, que respeta el paradigma MVC y permite crear Single-Page Applications (Aplicaciones web que no necesitan recargar la página), de manera más o menos sencilla. Es un proyecto mantenido por Google y que actualmente está muy en auge.



## Un poco de historia

"El software sigue al hardware". Esta es una afirmación que nos indica que programamos para aquellas máquinas en las que vamos a ejecutar los programas. Cuando empezó la informática en los años 60 existían ordenadores arcaicos y como programadores estabas limitado a las posibilidades de éstos.

Conforme avanzó el desarrollo de la informática aparecieron otros ordenadores. Al principio no estaban conectados entre sí ya que no existían las redes locales, ni mucho menos Internet. Como no había redes estabas limitado a lo que ocurría dentro de esa máquina y quizás los programadores tenían una vida más sencilla: estas limitaciones provocaban que no tuvieran que preocuparse por ciertas cosas. Incluso las opciones para crear los programas -tecnologías y lenguajes- no eran demasiadas, al contrario, quizás en tu sistema operativo estabas obligado a trabajar con un lenguaje o un par de ellos nada más.

Luego aparecieron las redes, apareció Internet y los ordenadores comenzaron a conectarse entre sí. Existen servidores y terminales que ya no son tontos, pero estamos trabajando con lenguajes sencillos, como HTML -al principio ni existía CSS- y ya acercándose al final del milenio aparecen lenguajes como Javascript capaces de hacer algunas cosas sencillas.

## HTML5 + JS



Si quieres realizar un desarrollo que se adapte a todo tipo de sistemas y dispositivos que puedan llegar a aparecer, una solución es buscar algo que sea común a todos los sistemas y buscar algo que tengas seguridad que lo van a entender todos.

Existe un **consenso en el mundo de la tecnología de dar soporte a HTML5 y Javascript**. La situación actual y la industria nos hace pensar que estos lenguajes estarán disponibles en todo sistema donde circule un bit. Por ello, podemos ver a estas tecnologías de estándares abiertos como un caballo ganador.

Eso sí, HTML5 + Javascript compiten con las soluciones nativas y en términos de rendimiento es muy difícil que puedan llegar a equipararse. El lenguaje nativo, además, siempre tendrá mayor facilidad de acceso a las características específicas y funcionalidades del dispositivo.

También compite con los lenguajes más clásicos como C, PHP, Java donde hay ya muchísimo trabajo ya realizado en forma de librerías disponibles para los programadores. Esta situación está cambiando, pero hay que observar que hasta hace poco era complicado hacer grandes programas con Javascript, pues el lenguaje servía para bien poco. Con la llegada de HTML5 y las diversas API se ha extendido mucho y se ha hecho mucho más poderoso, pero todavía faltaba mucho terreno para que Javascript se considerase un lenguaje robusto, capaz de cumplir las necesidades de aplicaciones grandes.

## Comparativa con otros frameworks :Backbone,EmberJS

---



Recientemente han aparecido una oleada de sistemas que han situado Javascript en otro nivel. AngularJS es uno de ellos, pero están otros muchos como BackboneJS o EmberJS. Son los frameworks que vienen a aportar herramientas y patrones de diseño con los que Javascript se convierte en un lenguaje capaz de servir como motor de aplicaciones enormes.

Y tiene todo el sentido que sea así. Hoy los ordenadores modernos, por muy modestos que sean, son capaces de procesar con velocidad ciertas cosas. Son capaces de recibir simples datos y "cocinarse" ellos mismos el HTML para visualizarlos a base de plantillas. Antes el servidor era el que tenía que enviar el HTML completo al cliente, ahora la tendencia es que solo envíe los datos y que el cliente (navegador o cualquier otro sistema donde desees ver esos datos) sea el que los trate y los muestre debidamente.

Esto ha producido que una parte de la lógica de maquetado y de presentación de la información se haya trasladado del servidor hacia los clientes. La ventaja obvia es que el servidor se ha descargado de trabajo, puesto que simplemente tiene que enviar los datos a través de JSON al cliente y es éste el que se encargará de producir el HTML que sea necesario. Pero no es solo una mejora en relación al servidor en términos de procesamiento, también en términos de bits, porque es más ligero transferir datos simples que el HTML completo para mostrarlos.

En definitiva, el servidor ha repartido la carga de trabajo que solía recaer sobre él entre todos los clientes que se conectan a su servicio. Pero la mejora no se queda solamente en el servidor, sino que el usuario también percibe un mejor desempeño, puesto que las acciones que realiza contra el servidor tienen una **respuesta más rápida**. Con ello poco a poco las aplicaciones cliente/servidor tienen un desempeño más parecido a las aplicaciones de escritorio. El usuario es el rey y demanda aplicaciones que sean rápidas y no le hagan esperar y eso se lo dan los frameworks como AngularJS.

Al programador además le facilitan las cosas, no solo por disponer de un conjunto de librerías, sino porque los frameworks nos traen un conjunto de paradigmas y patrones que facilitan el desarrollo del software y sobre todo su mantenimiento. Nos referimos principalmente al llamado MVC, que es la separación del código en diferentes responsabilidades. Ahora cada parte del código debemos situarlo en un lugar determinado, y ese orden nos facilita que los desarrollos sean más manejables.

Sobre todo esa mejora permite que en un futuro puedas "meter mano" al software para mantenerlo cuando sea necesario. Todo redundando en la **calidad del código**, lo que es indispensable para los proyectos y los programadores.

### **AngularJS mejora el HTML para crear aplicaciones web**

AngularJS y otros frameworks tienen además la característica de mejorar el HTML existente, facilitando el desarrollo de aplicaciones. En este punto cabe recalcar la palabra "aplicaciones" puesto que este tipo de herramientas son adecuadas para realizar las llamadas "aplicaciones de gestión" o "aplicaciones de negocio".

Es importante esta mención porque AngularJS no es adecuado para resolver todo tipo de proyectos, o al menos no te facilitará especialmente ciertos desarrollos. Incluso por sus características habrá necesidades que ni siquiera sean adecuadas realizar en HTML5, como posiblemente un videojuego con gráficos avanzados, donde sería más adecuado una aplicación nativa (aunque esto en el futuro pueda cambiar).

Otro ejemplo es la realización de una aplicación intensiva de SEO. En cuanto a posicionamiento orgánico en buscadores el desarrollo con AngularJS, u otros frameworks Javascript, no es muy interesante porque el HTML que reciben los clientes o los bots del motor de búsqueda están prácticamente vacío de contenido y solo se rellena a posteriori por medio de solicitudes Ajax.

Parece que Google está haciendo esfuerzos para que esta situación cambie y existen diversas soluciones a nivel de programación que pueden paliar en parte la carencia de SEO, pero lo cierto es que el desarrollo de la aplicación se complica al aplicarlas.



En fin, AngularJS nos ofrece muchas facilidades para hacer **aplicaciones web**, aplicaciones de gestión o de negocio, **aplicaciones que funcionan en dispositivos** y que tienen un rendimiento muy similar a las nativas e incluso **aplicaciones de escritorio con un frontal web**, cada vez más habituales.



[Backbone.js](#) es un framework realmente mínimo (¡poco más de 800 líneas de código!) y tiene una única dependencia, la genial librería de utilidades [Underscore.js](#).

Podríamos destacar las siguientes ventajas de Backbone:

- No hay necesidad aprender una sintaxis nueva, si tenemos experiencia con [jQuery](#) y [Underscore.js](#). Podemos programar utilizando Backbone sin apenas curva de aprendizaje. Sin duda, de los tres frameworks que comparamos aquí es el que presenta menor fricción a la hora de empezar o trabajar como estamos acostumbrados.
- Aunque es cierto que perdemos mucha potencia al disponer de menos magia en la punta de nuestros dedos, a cambio **ganamos en control** ya que podemos programar y cambiar el comportamiento de prácticamente cualquier elemento que se nos ocurra. Personalmente, siempre que utilizamos cajas negras prefabricadas nos da auténtico pánico que se dé el caso de lleguemos a necesitar una funcionalidad que por defecto no se proporcione. En ese caso, a menudo la única solución es tener que meterse en las “tripas” de la caja negra, que no está pensada para ser modificada por el usuario *de a pie*, y siempre es mucho más costoso. Esto con Backbone.js nunca nos va a pasar.

[AngularJS](#), de Google, es un framework con una filosofía muy distinta a la de Backbone. Angular se basa en extender comportamientos sobre etiquetas HTML.

[Ember.js](#) es un framework que se basa en una filosofía de **Convention over Configuration** (que traduciremos por algo equivalente a “*seguir las convenciones en vez de configurar*”), lo que permite al framework asumir de forma **automática** muchas tareas que en otros frameworks nos tocaría programar explícitamente.

Este concepto de [Naming Conventions](#) es clave en Ember, y posibilita una generación de código implícita que tiene mucho potencial. Otra de las características interesantes de Ember es el avanzado sistema de Routing, si lo comparamos con los sistemas por defecto de Backbone.js y AngularJS.

Otras alternativas:

<https://www.meteor.com>

## Por qué Angular JS y no otros frameworks

En este sentido podría haber mucho que discutir entre partidarios de uno y otro framework, pero si dejamos a un lado las preferencias personales de cada uno, por aquella tecnología en la que haya apostado en el pasado, AngularJS es objetivamente mejor en muchos sentidos.

Primero y más importante es que con AngularJS requieres escribir menos código que con otros frameworks. Por ejemplo con respecto a BackboneJS hay muchas mejoras que son realmente críticas como el "doble binding" que te permite que los distintos componentes de tu aplicación estén al tanto de los cambios para modificar su estado automáticamente, sin necesidad de suscribirse a eventos y realizar otro tipo de acciones por medio de líneas de código. En este sentido hay tests objetivos que nos permiten ver que la misma aplicación hecha con AngularJS tiene sensiblemente menos código que en BackboneJS y quizás con otros frameworks pase lo mismo.

Segundo la comunidad. AngularJS tiene el apoyo de Google y una gran comunidad detrás. Las búsquedas de AngularJS se han disparado mientras que las de otros frameworks no han mejorado o han caído. Esto quiere decir en definitiva que encontrarás más documentación y más componentes de otros desarrolladores para basar tu trabajo en ellos.

## Dos "mundos" en AngularJS

Ahora tenemos que examinar AngularJS bajo otra perspectiva, que nos facilite entender algunos conceptos y prácticas habituales en el desarrollo. Para ello dividimos el panorama del framework en dos áreas.

- **Parte del HTML:** Es la parte declarativa, con las vistas, así como las directivas y filtros que nos provee AngularJS, así como los que hagamos nosotros mismos o terceros desarrolladores.
- **Parte Javascript puro:** Que serán los controladores, factorías y servicios.



Es importante señalar aquí, aunque se volverá a incidir sobre ese punto, que nunca jamás se deberá acceder al DOM desde la parte del Javascript. Es un pecado mortal ya que esa parte debe ser programada de manera agnóstica, sin tener en cuenta la manera en la que se van a presentar los datos.

En medio tendremos el denominado scope, que como decimos representa al modelo en Angular. En resumen no es más que un objeto Javascript el cual puedes extender creando propiedades que pueden ser datos o funciones. Nos sirve para comunicar desde la parte del HTML a la parte del Javascript y viceversa. Es donde se produce la "magia" en Angular y aunque esto no sea del todo cierto, a modo de explicación para que se entienda algo mejor, podemos decir que AngularJS se va a suscribir a los cambios que ocurran en el scope para actualizar la vista. Y al revés, se suscribirá a los cambios que ocurran en la vista y con eso actualizará el scope.

## **Ventajas de utilizar AngularJS**

- Permite trabajar de manera organizada (MVC).
- Hace más fácil el trabajo en equipo ya que lo simplifica.
- Hace que escribamos menos líneas de código sin perder calidad.
- Reduce el tiempo de entrega para nuestros proyectos o aplicaciones.
- Enriquece el HTML
- La existencia del doble binding nos permite ahorrar líneas de código, reduciendo el uso de eventos, selectores, etc.
- Permite visualizar bloques y contenidos sin recargar la página utilizando Ajax.
- La potencia de las directivas.
- Promueve el patrón MVC y el desarrollo de Single Page Applications
- Agrega mejoras en el html, mediante las directivas.
- Realiza menos request al servidor, obteniendo solo el JSON necesario para actualizar la vista
- HTML enriquecido; promueve un HTML semántico y declarativo.
- Promueve y usa el patrón de desarrollo MVC, dividiendo el trabajo de manera organizada.
- Incorpora de forma nativa el doble-binding.
- Es ideal para la programación de desarrollo de sistemas de trabajo; volcando todo el trabajo en el equipo del cliente; esto permite que el servidor se ahorre tiempo de ejecución. Esto se permite a través del trabajo de AJAX, recargando parte de la página.

## **Características importantes de AngularJS**

- Permite mejorar el HTML.
- Produce automáticamente el “doble binding”, lo cual facilita enormemente el trabajo ya que nos ahorramos muchas líneas de código.
- Al implementar el patrón MVC, permite separar el código en diferentes responsabilidades lo cual nos facilita que los desarrollos sean más manejables. Esto facilita tener que modificar el software en un futuro para su mantenimiento.
- Todo redunda en la calidad del código, lo que es indispensable para los proyectos y los programadores.
- Hay un ecosistema de herramientas alrededor de los frameworks como servicios web y librerías de terceros que nos facilitan una enorme gama de objetivos.
- AngularJS tiene el apoyo de Google y una gran comunidad detrás.
- Es un proyecto de código abierto.
- El patrón MVC muy utilizado en muchos frameworks web que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.
- El ser un framework de código libre muy utilizado por la comunidad con amplio soporte.
- La posibilidad de poder lograr buenos resultados con poco esfuerzo (entiendase como poder hacer lo mismo que con otros framework llevarían muchas más líneas de código).
- Se integra fácilmente con otros frameworks muy populares como bootstrap.
- El patrón MVC nos facilita la tarea de generar código de calidad.
- Cuenta con Doble Binding => Distintos Componentes del App Actualizan su estado automáticamente sin necesidad de programar eventos escuchas.
- Al ser Open Source cuenta con mayor documentación. Es SPA => Single Page Application (La página no se recarga)

## 10 razones por las que elegir AngularJS

---

**MVC bien realizado.** Normalmente en los frameworks MVC se pide al desarrollador que escriba el código de una cierta forma para poder funcionar. Esto supone mucho trabajo. AngularJS sin embargo gestiona los componentes y los interconecta de una forma muy sencilla y sin complicaciones.

1. **HTML para la interaz.** Se utiliza el HTML con su sistema de etiquetado tradicional para la definición de la interfaz gráfica de la aplicación web. Mucho más sencillo e intuitivo que tener que definir toda la UI mediante Javascript.
2. **Modelo de datos POJO.** En AngularJS los modelos de datos son Javascript Objects (POJO) con lo que no requieren de funciones extras estilo *getters & setters*. El código queda limpio y natural. Tradicionalmente los modelos de datos se comportan como proveedores de datos con funcionalidad. En AngularJS son simples datos y los proveedores de datos se realizarán mediante *services*.  
Gracias a *\$scope* AngularJS acerca mucho la vista al controlador y nos permite tener estos modelos de datos vinculados entre los procesos y la vista. AngularJS esta continuamente observando (*watch*) los cambios de estos datos / propiedades actualizando la vista en caso de ser necesario.
3. **Directivas con comportamientos.** Angular JS nos permite añadir funcionalidad extra con nuevas etiquetas HTML. Imagina un mundo donde puedas crear tus propias etiquetas y que sean componentes de alto nivel. Por ejemplo `<accordion></accordion>`, `<grid></grid>`, `<lightbox></lightbox>`...  
Con AngularJS puedes inventar y definir tus propios elementos de HTML. Esto es extremadamente **potente, reutilizable y escalable**.

Puedes definir:

- **Atributos personalizados** `<miDirectiva></miDirectiva>`
  - **Clases personalizadas** `<html><div miDirectiva></div> o <div class="miDirectiva"></div>`
4. **Flexibilidad con los filtros.** De una forma eficiente cómoda y sencilla podemos formatear los datos o filtrarlos con una sólo línea de código: en la definición de visualización de los datos como parámetro. Son funciones desarrolladas totalmente independientes que se encargan de transformar los datos.
  5. **Escribe menos código.** Todos estas características hacen que el desarrollador tenga que programar menos y por tanto menos errores.
    - No requiere de crear tu propia estructura MVC.
    - Las vistas están definidas en HTML y son muy consisas.
    - Los modelos de datos no requieren de *getters & setters*
    - Data-binding simplifica la comunicación vista-controlador.
    - Las directivas separan el código de la aplicación y lo hacen escalable.
    - Los filtros manipulan los datos de una forma muy cómoda.
  6. **Manipular DOM** en el controlador. Tradicionalmente (por ejemplo con JQuery) manipulamos el DOM añadiendo comportamiento. Con AngularJS separamos el comportamiento de manipular el DOM, de hecho en AngularJS se debe realizar en las directivas y no en la vista.
  7. **Services.** Los controladores en Angular acaban haciendo un único trabajo muy simple: manipular el *\$scope*. Estos controladores se apoyan en los servicios (*services*). No se involucran en el MVC de la aplicación y funcionan como simples API obteniendo los datos.
  8. **Sistema de eventos.** Sencilla comunicación por mensajes mediante eventos. Comunicando un particular nodo con sus hijos. Mediante *broadcast()* se envia a todos sus hijos y mediante *emit()* a todos sus padres.

Los controladores se puede comunicar mediante este sistema de eventos pero también se puede simplificar mucho esta comunicación gracias al data-binding.

9. **Unit testing.** AngularJS tiene un buen mecanismo de testeo en parte gracias también a su Inyección de *dependencias* (DI)

¿Qué ventajas tiene esta forma de trabajar?

- La principal ventaja es la rapidez. Las aplicaciones actuales tienen mucho CSS, JavaScript , etc. Al tener una única página real, se carga todo ello al principio una única vez y cuando cambiamos de *página* todo lo anterior ya está cargado , y sólo cambiamos el HTML de la nueva *página*.
- Otra ventaja es que podemos mantener nuestro estado en variables JavaScript ya que nunca salimos de la página , lo que nos simplifica mucho el desarrollo. ¿Recordáis cuando navegar a otra página podía implicar pasar muchas cosas en la URL para pasar el estado de la aplicación de una página a otra?

## Arquitectura

Angular es una librería que permite realizar aplicaciones de una forma muy sencilla y muy rápida. Para ello es necesario organizar el código de una forma especial basada en directivas, módulos, factorías, ... que iremos poco a poco desgranando, pero lo más importante es hacer un cambio de mentalidad en cuanto a la arquitectura que implementa como se maneja la lógica de presentación.

Hasta el momento, en la web estamos acostumbrados a una forma de trabajar muy procedural y basada en suscripciones a eventos que provocan que según los argumentos y los valores de algunos atributos del HTML se modifiquen dinámicamente el DOM de la página.

Por ejemplo si queremos añadir un elemento a una lista, la forma de trabajar es suscribirse a un botón [+], al invocarse el evento se cargarían los datos a través de jQuery y se añadiría un `<li>` al `<ul>` utilizando jquery con los datos que queremos mostrar.

Esta forma de trabajar acopla mucho el html de la página, el código del método suscrito al evento, ...

```
1 <!DOCTYPE>
2 <html>
3   <head></head>
4   <body>
5     <div>
6       <input id="nombre" type="text"></input>
7       <span id="descripcion">El nombre introducido es:</span>
8       <button id="anyadir">+</button>
9       <ul id="lista">
10      </ul>
11    </div>
12    <script src="http://code.jquery.com/jquery-1.9.0.js"></script>
13    <script>
14      $(document).ready(function() {
15        $("#nombre").keyup(function() {
16          var name = $("#nombre")[0].value;
17          $("#descripcion")[0].innerHTML = "El nombre introducido es: <strong>" + name + "</strong>";
18        });
19        $("#anyadir").click(function() {
20          var name = $("#nombre")[0].value;
21          $("#lista")[0].innerHTML = $("#lista")[0].innerHTML + "<li>" + name + "</li>";
22        });
23      });
24    </script>
25  </body>
26 </html>
```

## Tema 2: Desarrollo con AngularJS

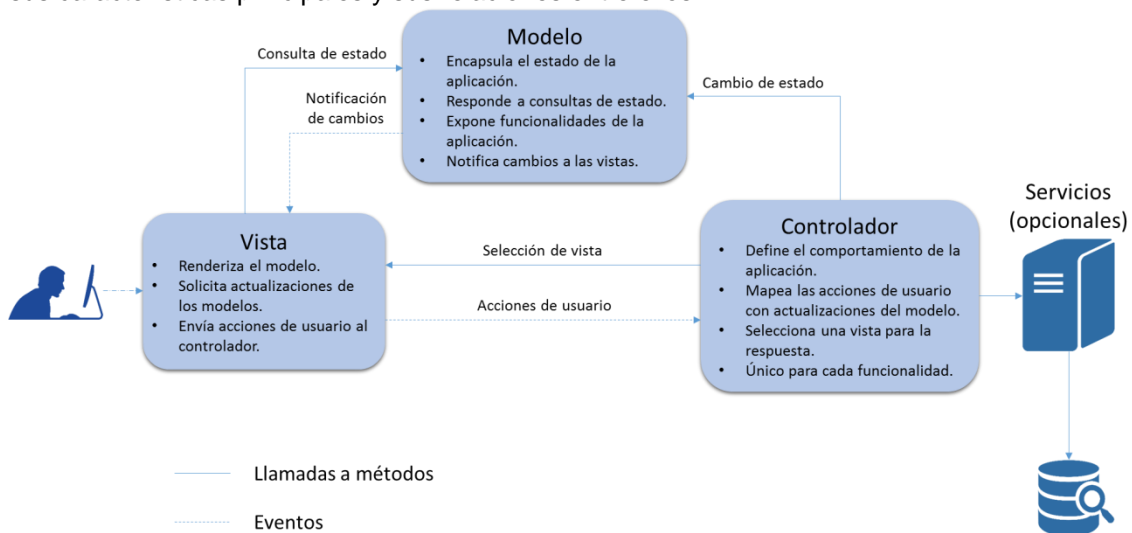
### Patrón de arquitectura MVC

Primeramente tenemos que hablar sobre el gran patrón que se usa en Angular, el conocido Modelo, Vista, Controlador.

- **Vistas:** Será el HTML y todo lo que represente datos o información.
- **Controladores:** Se encargarán de la lógica de la aplicación y sobre todo de las llamadas "Factorías" y "Servicios" para mover datos contra servidores o memoria local en HTML5.
- **Modelo de la vista:** En Angular el "Modelo" es algo más de aquello que se entiende habitualmente cuando te hablan del MVC tradicional, osea, las vistas son algo más que el modelo de datos. En modo de ejemplo, en aplicaciones de negocio donde tienes que manejar la contabilidad de una empresa, el modelo serían los movimientos contables. Pero en una pantalla concreta de tu aplicación es posible que tengas que ver otras cosas, además del movimiento contable, como el nombre de los usuarios, los permisos que tienen, si pueden ver los datos, editarlos, etc. Toda esa información, que es útil para el programador pero que no forma parte del modelo del negocio, es a lo que llamamos el "Scope" que es el modelo en Angular.

Algunos autores dicen que el patrón que utiliza el framework es el **MVVM Model-View-View-Model**. En resumen, el modelo de la vista son los datos más los datos adicionales que necesitas para mostrarlos adecuadamente.

Angular es un framework Javascript que, entre otras funcionalidades, permite implementar un patrón MVC Client Side. En el esquema que se puede ver en la **¡Error! No se encuentra el origen de la referencia.** se muestran los principales componentes de dicho modelo, así como sus características principales y sus relaciones entre ellos:



El código de aplicaciones creadas con Angular se organiza atendiendo a los diferentes elementos mostrados en el esquema anterior, cuyas funcionalidades se detallan a continuación:

1. **Modelo:** Estructura de datos almacenada en la memoria utilizada por el navegador que describe el estado de la aplicación.
2. **Vista:** salida mostrada al usuario. Se basa en una plantilla que define cómo se muestran los datos del modelo al usuario.
3. **Controlador:** Pieza que se encarga de sincronizar la información entre la vista y el modelo en base a la gestión de los eventos disparados o bien por la vista o bien por los cambios en el contenido del modelo.
4. **Servicio (opcional):** Objetos especializados que realizan el trabajo en nombre de otros. Los servicios tienen varios usos, como por ejemplo traer datos remotos o proveer e implementar algún algoritmo. El objetivo es que sean altamente reutilizables y puedan ser intercambiados por otros similares de forma sencilla.

Los diferentes elementos mencionados anteriormente interactúan entre ellos de la siguiente forma:

- La vista contempla todo lo relativo a lo que visualiza el usuario (HTML, CSS, etc.). Se relaciona con un controlador o con un modelo para recibir información a través del scope (contexto) de la misma.
- El controlador recibe las acciones desde la vista, y se lo comunica al modelo, el cual le devolverá la información solicitada, viéndose esta reflejada en la vista. El controlador es el encargado de ejecutar la lógica aplicativa, bien sea debido a la utilización de servicios externos o bien porque la tiene implementada él mismo.
- El modelo contendrá los datos obtenidos por el controlador, sea mediante la utilización de servicios o mediante su propia lógica aplicativa. Dicho controlador también puede modificar el scope de la vista, de manera que cualquier cambio realizado en el modelo se verá reflejado en la vista, y viceversa.

## Patrón MVVM

Ahora con angularjs todo esto cambia y pasa a ser mucho más declarativo. Siguiendo el ejemplo anterior:

- Al botón [+] le añadiríamos la directiva ng-click con el método js que queremos ejecutar, además le pasaríamos los argumentos que este necesitase.
- En este método js que manejaría el evento no le llegaría ningún argumento de forma automática, ni ninguna referencia al elemento HTML que lo ha invocado. Simplemente tendría acceso a los valores que desde la llamada ng-click se han pasado.
- Una vez dentro del método js, este se añadiría el nuevo objeto en la colección bindeada en la directiva ng-repeat.
- En ningún momento se llama al DOM ni se modifica. Como el elemento <li> tendrá una directiva ng-repeat que hará que se cree un elemento por cada elemento de la colección bindeada, el hecho de añadir un elemento más a esta colección añadirá provocará que un elemento <li> aparezca de forma mágica rellenando los campos según la plantilla definida en el html. Todo esto es debido a utilizar una arquitectura MVVM (Model-View-ViewModel) o MVW (Model-View-Whatever) basada en bindings entre la parte visual y los datos.

Siguiendo esta nomenclatura podemos separar 3 capas muy importantes de toda aplicación angularjs:

- **Model.** Entendemos la capa Model como los datos que la aplicación va a utilizar. En angular esta capa se define mediante la variable \$scope. Esta variable es la que contiene la información de la aplicación y sobre la que podemos realizar bindings desde el html para ser mostrado.
- **View.** Llamaremos view a todos los ficheros html y css que se utilizan para formatear la información. La forma en que la parte visual muestra información alojada en el Model es

utilizando plantillas con formato mustache `{{nombre}}` o directivas explícitamente destinadas a ello como `ng-bind`.

- **ViewModel o Whatever.** Esta capa que según el nombre de la arquitectura tiene un nombre u otro contiene todo aquel código que es la encargada de enlazar los eventos y preparar los datos para ser visualizados de la forma requerida. Por ejemplo alojaría en código que se ejecutaría al pulsar el botón de añadir un nuevo elemento, como el formateo de una fecha, ... Los controladores, directivas, servicios, factorías, ... son las entidades que se encargarán de todo esto. Una cosa son los datos de la aplicación, otra como se va a ver la información y otra la lógica que enlaza ambas. Para entenderlo mejor vamos a ver un ejemplo: Donde *whatever* es *whatever works for you*. Angular proporciona mucha flexibilidad para separar de manera sencilla la lógica de presentación de la de diseño y el estado de presentación.

```
1  <!DOCTYPE>
2  <html>
3    <head></head>
4    <body ng-app="ejemplo1">
5      <div ng-controller="listController">
6        <input type="text" ng-model="name"></input>
7        <span>El nombre introducido es: <strong>{{name}}</strong></span>
8        <button ng-click="add();"></button>
9      </div>
10     <li ng-repeat="item in lista">{{item}}</li>
11   </ul>
12 </div>
13 <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.21/angular.js"></script>
14 <script>
15   angular
16     .module("ejemplo1", [])
17     .controller("listController", function($scope) {
18       $scope.lista = [];
19       $scope.add = function() {
20         $scope.lista.push($scope.name);
21       }
22     });
23 </script>
24 </body>
25 </html>
```

En este ejemplo a diferencia del primero con la programación basada en jQuery se puede ver que el código es mucho más sencillo. No es necesario suscribirse al `ready` del doc, ni saber cómo se llaman los eventos, ni inyectar html a la página según se necesite (evitando ataques por inyección de html ya que angular tiene un buen sanitizador), ni llenar el html de ids y class para poder utilizar los selectors, ni podemos olvidarnos de sincronizar el valor de algún elemento si la funcionalidad es compleja, ... En el caso de angular simplemente se debe actualizar el dato (`$scope`) y ya las plantillas y eventos harán el resto.

El patrón MVVM funciona muy bien en aplicaciones con interfaces de usuario ricas, ya que la Vista se vincula al ViewModel y, cuando el estado del ViewModel cambia, la Vista se actualiza automáticamente gracias, precisamente, al **two-way-databinding**. En AngularJS, una Vista es simplemente código HTML compilado con elementos propios del framework. Una vez finaliza el ciclo de compilación, la Vista se vincula al objeto `$scope`, que es el ViewModel. Ya veremos en profundidad el objeto `$scope`, pero adelantamos que es un objeto JavaScript que captura ciertos eventos para permitir el *data binding*. También, podemos exponer funciones al ViewModel para poder ejecutar funciones.



## Estructura de una aplicación

---

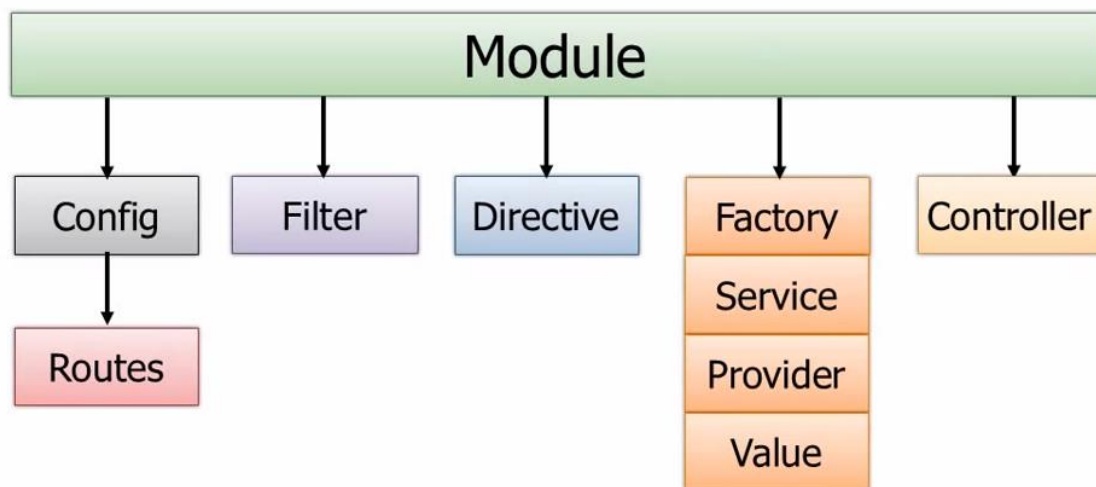
Angular es un framework creado y soportado por Google, la página de su proyecto es <https://angularjs.org> y en ella podemos encontrar la documentación de la **API**, manuales, tutoriales y acceso a mucha documentación.

Además, como complemento para desarrolladores, podemos encontrar también la Guía de estilo de John Papa. Puede accederse a la misma a través de su [página de github](#).

**Módulos:** La manera que nos va a proponer AngularJS para que nosotros como desarrolladores seamos cada vez más ordenados, que no tengamos excusas para no hacer un buen código, para evitar el código piramidal, ficheros gigantescos con miles de líneas de código, etc. Podemos dividir las cosas, evitar el infierno de las variables globales en Javascript, etc. Con los módulos podemos realizar aplicaciones bien hechas, de las que un programador pueda sentirse orgulloso y sobre todo, que nos facilite su desarrollo y el mantenimiento.

Para poder definir rutas, tenemos que estructurar nuestra aplicación un poquito mejor de lo que hemos hecho hasta ahora. Aunque lo que hemos hecho hasta ahora es una aplicación AngularJS, no es la manera recomendada de implementarla.

En primer lugar, tenemos que definir un módulo en nuestra etiqueta ng-app , cosa que no habíamos hecho aún. Dentro de nuestro objeto module es donde vamos a poder configurar nuestras rutas, y también definir filtros, directivas, controladores, factorías, y demás servicios que serán específicos para nuestra app. Podríamos pensar en un module como un contenedor de objetos donde podemos tener todas estas cosas.



Lo importante es definir un nombre para nuestro módulo. Como estamos haciendo una pequeña aplicación que muestra un listado de profesores y asignaturas, la llamaremos booksApp . Éste es el nombre que irá en nuestra etiqueta [ng-app].

```
<html ng-app="booksApp">
```

Y ésta es la manera de crear un módulo en AngularJS:

```
var booksApp = angular.module('booksApp', []);
```

Con esto, estamos creando un módulo, y diciendo a AngularJS que se va a llamar **booksApp**. Vemos que está seguido por un array vacío. Aquí es donde vemos la potencia de la inyección de dependencias. Resulta que nuestro módulo puede incluir otros módulos de los que nuestra aplicación depende en cierto grado. Por ejemplo, en el siguiente fragmento estamos incluyendo un módulo llamado myCustomModule:

```
var booksApp = angular.module('booksApp', ['myCustomModule']);
```

Aquí, estamos diciendo a Angular que busque otro módulo, llamado myCustomModule , y lo inyecte en el nuestro, de manera que todos sus elementos (directivas, filtros, factorías, ...) estarán disponibles en el momento que los inyectemos.

AngularJs dispone de otros tipos de artefactos que iremos viendo a lo largo del curso y que también deben añadirse a los módulos:

- Filtros
- Constantes
- Servicios
- Factorías
- Providers
- Config

*Un módulo es donde se añaden los distintos artefactos que usaremos en nuestra aplicación.*

Podríamos verlo como un paquete de Java. Es como una forma de agrupar funcionalidades de Angular. Por ejemplo, podríamos crear varias directivas relacionadas con google Maps. Lo normal sería agrupar todas esas directivas en un único módulo y luego al crear nuestra aplicación decir que vamos a usar dicho módulo.

A partir de la versión 1.2, el módulo de rutas se extrajo del core de AngularJS como módulo independiente, con lo que lo primero que tendremos que hacer es traernos este módulo:

```
<script src="https://code.angularjs.org/1.2.16/angular-route.min.js"></script>
```

Como hemos mencionado anteriormente, para incluir módulos extras en el nuestro, tenemos que declararlo en la definición:

```
var booksApp = angular.module('booksApp', ['ngRoute']);
```

Una vez hecho esto, ya podemos definir las rutas. Hemos dicho anteriormente que un módulo de AngularJS tenía una función de configuración, donde definíamos las rutas. Vamos a usar esta función, a la que tenemos que inyectarle un objeto llamado \$routeProvider .

## Herramientas de desarrollo

---

Para desarrollar el proyecto nos vamos a apoyar en diversas herramientas.

- **NodeJS** : Es un framework de JavaScript para desarrollar en el lado del servidor. La página de su proyecto con la documentación relativa se puede encontrar en <https://nodejs.org>.
- **NPM** : Es una herramienta de gestión de librerías que nos ayudara a descargar y mantener actualizadas las librerías que necesite nuestro proyecto. La página del proyecto es <https://www.npmjs.com/>. Además tenemos un buscador para localizar librerías existentes en los repositorios de NPM y obtener información sobre los mismos.
- **Bower** : Es un gestor de dependencias de proyectos que permite mantener organizadas las librerías para que todos los desarrollos funcionen correctamente sin que los desarrolladores tengan que preocuparse de incompatibilidades o falta de librerías en el proyecto. Se apoya en NPM para gestionar dichos requisitos.
- **Grunt** : Es un gestor de tareas que nos ayudara a ejecutar la preparación del proyecto, compilación, minimización de fuentes y otras funciones. La página del proyecto es <http://gruntjs.com/> y contiene mucha documentación para la creación y gestión de tareas.

## Angularjs Batarang

---

Necesitamos herramientas para poder depurar el código y ver en todo momento que contiene las variables internas. Para ello, nosotros recomendamos el uso de Chrome y su extensión Angularjs Batarang.

<https://chrome.google.com/webstore/detail/angularjsbatarang/ighdmehidhipcmcojjiloacoafjmpfk>

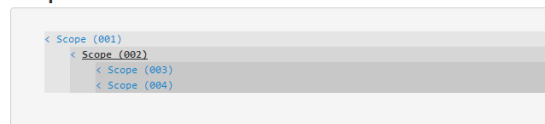
Con estas dos herramientas seremos capaces en todo momento de ver qué valores hay dentro del \$scope, trazar el código js, ver qué plantilla se ha descargado, que CSS se ha aplicado y cual no. Toda una serie de valores e información sin las cuales es muy complejo programar cualquier aplicación.

En la Chrome Store tenemos una extensión para Angular, llamada [Batarang](#). Puedes descargarla desde [aquí](#). Esta extensión te permite visualizar los scope y otras particularidades que son útiles a la hora de depurar.

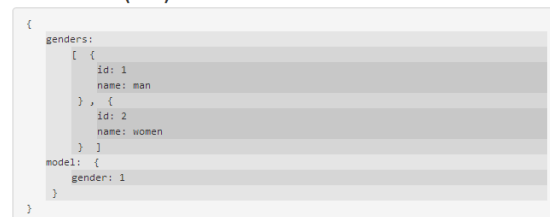
### Depuración

Lo ideal es hacerlo desde Chrome y utilizar [Batarang](#) con esta herramienta se ahorra bastante tiempo a la hora de depurar una app. Una muestra de los scopes creados en nuestro ejemplo.

Scopes



Models for (002)



Si tienes instalada Batarang y quieres trabajar desde la consola, también lo puedes hacer de la siguiente forma.

1. Selecciona un elemento del Dom.
2. Desde la consola escribe \$scope.

## Gestión de paquetes con npm y bower

---

### Instalación de Bower

Bower es un gestor de descarga de dependencias y componentes de la misma forma que está el gestor de paquetes de node(npm).

Otro elemento global que necesitaremos es [Bower](#). Bower es a las librerías JavaScript de front-end lo que NPM a las librerías de backend de node.js. Este gestor de paquetes nos puede descargar librerías como AngularJS, ui-router, jQuery, etc. De manera que ya no es necesario irse al sitio web del framework/librería para descargarnos lo que necesitamos.

Bower se puede instalar con el gestor de dependencias npm:

```
npm install -g bower
```

## Estructura inicial de nuestro proyecto. Gestión de dependencias

Vamos a definir las dependencias de nuestro proyecto. Necesitaremos una serie de librerías de backend, que gestionará npm, y de frontend, que gestionará Bower. Ambas herramientas necesitan un fichero de configuración JSON que define estas dependencias.

El fichero de configuración de node.js se llama `package.json`, y podemos inicializarlo lanzando, desde consola, el comando **npm init**:

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (Desktop) angular-project
version: (1.0.0)
description: angular project
entry point: (index.js)
test command:
git repository:
keywords: angular project
author:
license: (ISC)
```

```
{
  "name": "angular-project",
  "version": "1.0.0",
  "description": "angular project",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "angular",
    "project"
  ],
  "author": "",
  "license": "ISC"
}
```

El fichero de configuración de bower se llama `bower.json`, y se inicializa con el comando **bower init**:

```
? name angular-project
? description angular project
? main file index.js
? what types of modules does this package expose? es6, node
? keywords angular,project
? authors jmortega <jmoc25@gmail.com>
? license ISC
? homepage
? set currently installed components as dependencies? Yes
? add commonly ignored files to ignore list? No
? would you like to mark this package as private which prevents publishing? No

{
  name: 'angular-project',
  description: 'angular project',
  main: 'index.js',
  authors: [
    'jmortega <jmoc25@gmail.com>'
  ],
  license: 'ISC',
  keywords: [
    'angular',
    'project'
  ],
  moduleType: [

```

Ahora las dependencias. En la parte de front-end vamos a instalar la dependencia de AngularJS:

```
bower install angular --save
```

Añadimos la opción -g , para que las dependencias aparezcan en el fichero de configuración:

```
{
  "name": "angular-project",
  "description": "angular project",
  "main": "index.js",
  "authors": [
    "jmortega <jmoc25@gmail.com>"
  ],
  "license": "ISC",
  "keywords": [
    "angular",
    "project"
  ],
  "moduleType": [
    "es6",
    "node"
  ],
  "homepage": "",
  "dependencies": {
    "angular": "~1.5.8"
  }
}
```

También podemos ver que esas dependencias han aparecido en el fichero **package.json**:

```
{
  "name": "angular-project",
  "version": "1.0.0",
  "description": "angular project",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "angular",
    "project"
  ],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "angular": "~1.5.8"
  }
}
```

Para instalar las dependencias podemos ejecutar **npm install --save**

En nuestro caso nos baja la librería de angular dentro de la carpeta node-modules

## Instalación de AngularJS

Para **empezar el desarrollo en AngularJS** lo primero que hay que hacer es bajarse el framework de la página oficial <http://www.angularjs.org>

Al ser AngularJS un framework JavaScript solo tenemos que descargar los ficheros JavaScript e incluirlos en los proyectos con un tag <script> sin embargo hay varias sitios de donde descargar AngularJS o múltiples ficheros a incluir.

Si vamos a la página de descarga nos aparecerá una ventana como ésta:



- **Branch:** La versión de angularJS a descargar es la 1.5.x. En este caso la versión 1.5.8 es la última versión estable.
- **Build**
  - Minified: Solo el fichero angular.js pero minimizado
  - Uncompressed: El fichero angular.js *normal* sin minimizar.
  - Zip: Un zip que contiene los dos ficheros anteriores pero tambien muchos mas ficheros de AngularJS que según lo que vayamos a hacer con AngularJS necesitaremos.
- **CDN:** En vez de bajar el fichero angular.js, alojarlo en el servidor y que nuestra página HTML se lo bajen desde nuestro servidor, Google pone a nuestro servicio una URL desde donde la página HTML pueda directamente usar el fichero angular.js sin que este alojado en nuestros servidores y que en general suele ser más rápida y/o suele estar ya cacheada por lo que aumenta el rendimiento de nuestra aplicación. Por supuesto esto último solo lo podemos usar si los clientes que usan la aplicación tienen acceso a internet, cosa que puede que no tengan si estamos hablando de una aplicación empresarial en unas oficinas sin acceso a internet. Mas información en [CDN](#).
- **Bower:** No vamos a explicar en este curso [bower](#) pero simplemente es poco un gestor de paquetes para la web.
- **Extras:** Es la URL donde se encuentran todos los ficheros de AngularJS, como en el Zip anterior, pero de todas las versiones que se han publicado.

Por otro lado también se pueden descargar **módulos adicionales oficiales** que nos ayudarán en el desarrollo de nuestra Aplicación Web. Entre los que destacan **angular-route.min.js** para realizar el sistema de routing de la aplicación. Para descargarlos hay que hacer click en la ventana de download, la opción Extras/Browse additional modules.

## Javascript necesario

Para ejecutar una aplicación angular es necesario añadir la librería angularjs asociada. Esta la puedes encontrar en <https://angularjs.org/> y en el momento de escribir el post se puede descargar la versión 1.5.8 como última versión estable. También está la 1.3.0-beta 17 pero es recomendable trabajar siempre con versiones estables para evitar sorpresas.

Para utilizarla puedes:

- Bajarte el fichero angular.js deL repositorio oficial de google e incluirlo en nuestro index
- Apuntar a la url <https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular.js> que redirecciona al CDN de Google donde se descarga
- Utilizar el comando Bower "bower install angular#1.5.8" para la descarga

En cualquier caso es importante asociar la referencia a una versión concreta de angular.js que permita que la aplicación funcione aunque salgan nuevas versiones de esta librería.

Además de la librería angular.js existen también otras librerías específicas para animaciones, enrutado, sanitización, recursos, mocks, gestión de cookies, ... que se cargan por separado. Conforme vayamos hablando de algunas de estas características las iremos nombrando y describiendo como cargarlas.

## Primeros pasos con AngularJS

Vamos a ver qué necesitas para trabajar con AngularJS, como descargar el framework y cómo realizar un primer programa, el típico Hola Mundo.

Para trabajar con AngularJS tienes que incluir el script del framework en tu página. Esto lo puedes hacer de dos maneras, o bien te descargas la librería por completo y la colocas en un directorio de tu proyecto, o bien usas un CDN para traerte la librería desde un servidor remoto.

Si descargas el script tendrás la opción de descargar la versión minimizada, que pesa menos en Kb, por lo que será la adecuada para cualquier sitio que tengas en producción. Sin embargo la versión sin comprimir "Uncompressed" tiene el código completo, con comentarios, sangrías, etc. lo que hace que sea más fácil su lectura. Puedes usar si lo deseas la versión sin comprimir en la etapa de desarrollo.

## Incluir AngularJS en una página web

Una vez tienes tu CDN puedes incluir el script de Angular en la página con la etiqueta SCRIPT. Ese script lo puedes colocar en el HEAD o bien antes del final del BODY, en principio no habría diferencias en lo relativo a la funcionalidad, pero sí hay una pequeña mejora si lo colocas antes de cerrar el cuerpo.

Simplemente, si lo colocas en el HEAD estás obligando a que tu navegador se descargue la librería de AngularJS, retrasando quizás la descarga de áreas de la página con contenido. Si lo colocas antes de cerrar el BODY facilitas la vida a tu navegador, y por añadido a tus usuarios, pues podrá descargar todo el HTML, ir renderizando en la pantalla del usuario los contenidos sin entretenerse descargando AngularJS hasta que sea realmente necesario.



## Declarar directivas

Hay un paso más para dejar lista una página donde quieras usar AngularJS. Es simplemente colocar la directiva ng-app en la etiqueta que englobe la aplicación.

Más adelante hablaremos con más detalle de las directivas y daremos algunos tips para usarlas mejor. Típicamente pondrás ng-app en la etiqueta HTML de inicio del documento.

Así como ng-app, existen muchas otras directivas para enriquecer tu HTML, a lo largo del curso iremos viéndolas.

Vamos a colocar un formulario con un campo de texto.

```
<form>  
¿Cómo te llamas? <input type="text" ng-model="nombre">  
</form>
```

Observa que en el campo de texto hemos usado ng-model y le hemos asignado un valor. Ese ngmodel es otra directiva que nos dice que ese campo de texto forma parte de nuestro modelo y el valor "nombre" es la referencia con la que se conocerá a este dato.

Ahora vamos a crear un elemento de tu página donde volcaremos lo que haya escrito en ese campo de texto.

```
<h1>Hola {{nombre}}</h1>
```

Como ves, dentro del H1 tenemos {{nombre}}. Esas dobles llaves nos sirven para indicarle a AngularJS que lo que hay dentro es una expresión. Allí podemos colocar cosas (código) para que Angular resuelva por nosotros. En este caso estamos colocando simplemente el nombre del modelo o dato que queremos mostrar.

Como veras no hemos escrito código Javascript, justamente la potencia de AngularJS es que muchas cosas se pueden hacer en el documento HTML simplemente extendiendo sus posibilidades a través de directivas y expresiones.

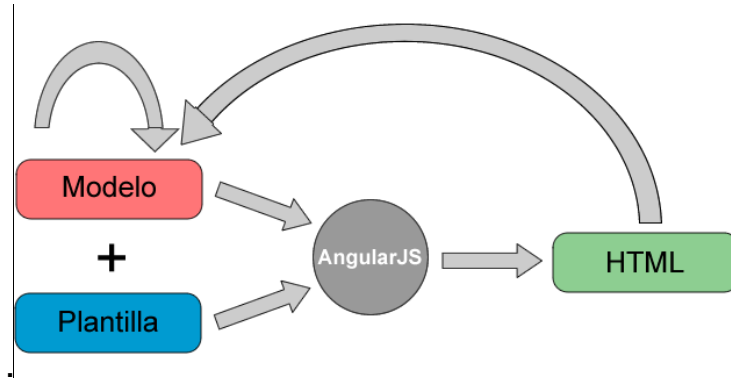
### Tema 3: MVC (Modelo-Vista-Controlador) en AngularJS

#### Enlace de datos (Data Binding)

---

El Data-binding es el concepto más importante de angular y lo que hace es relacionar el HTML con nuestro modelo de datos. Es lo que acabamos de ver de poner las `{{ }}` en el HTML y que se transforme con los datos del JavaScript.

La generación del HTML se produce una única vez al generar la página



Ahora es cuando podemos decir que el sistema de Data-binding y plantillas de AngularJS no se genera una vez sino que constantemente se está regenerando en cuanto cambia alguna parte del modelo o del HTML. Ésta es una diferencia muy importante respecto a otros motores de plantillas. Ya no es necesario que nosotros hagamos nada para sincronizar el HTML y el modelo. Angular lo hace todo por nosotros. Se acabó el programa en el que una parte de la pantalla no estaba actualizada porque se nos había olvidado decir que se refrescara esa parte con los nuevos datos.

Es decir que el Data-binding funciona en ambas direcciones.

- Modelo → HTML
- HTML → Modelo

Y lo mejor de todo es que no cuesta nada hacer las modificaciones en nuestro modelo de JavaScript. Es tan sencillo como modificar una variable de JavaScript y el HTML se actualiza.

#### Binding en AngularJS, y doble binding

Una de los aspectos más importantes de angular son los bindings, de los enlaces de las plantillas de angular utilizan para mostrar y sincronizar los datos en el HTML.

Con AngularJS es muy importante cambiar la mentalidad de lo que normalmente se hace con JavaScript / jQuery. Con jQuery como respuesta a un evento se modifica directamente el DOM de HTML para mostrar la información requerida.

- Para mostrar la información las plantillas de angular hacen binding con los datos del `$scope` para que se muestren donde se requiere y como se quiere.

- Para asociar una función JavaScript a los eventos del DOM se utilizan algunas directivas como `ngClick` que asocian eventos a las funciones que se encargarán de modificar los datos del `$scope`.

La idea es que los eventos nunca modifiquen directamente el DOM sino los datos contenidos en el `$scope`. Estos cambios de forma indirecta ya provocarán los cambios al refrescar la parte visual y volver a aplicar la plantilla.

La traducción de "binding" sería "enlace" y sirve para eso justamente, realizar un nexo de unión entre unas cosas y otras. Data binding sería "enlace de datos".

Una de las características de AngularJS es producir automáticamente lo que se llama el "doble binding" que nos facilita enormemente nuestro trabajo por ahorrarnos muchas líneas de código para realizarlo a mano. El doble binding no existe en todos los frameworks MVC de Javascript como BackboneJS, donde solo tenemos un "bindeo" simple.

El binding no es más que enlazar la información que tenemos en el "scope" con lo que mostramos en el HTML. Esto se produce en dos sentidos:

### **One-way binding:**

En este caso la información solamente fluye desde el scope hacia la parte visual, osea, desde el modelo hacia la representación de la información en el HTML. Lo conseguimos con la sintaxis "Mustache" de las dos llaves.

```
{{ dato }}
```

Ese dato estaría trayéndolo desde el scope y mostrándolo en la página. La información fluye desde el scope hacia la representación quiere decir que, si por lo que sea se actualiza el dato que hay almacenado en el modelo (scope) se actualizará automáticamente en la presentación (página).

El primer tipo de binding es el One-Way. Este tipo de binding se utiliza para autorefreshar la parte visual frente a cambios en el \$scope. El único refresco posible con este tipo es el \$scope ⇒ HTML y no al revés.

Para implementar esto se puede hacer de dos formas: utilizando la nomenclatura mustache {{ ... }} o con la directiva ngBind

### **¿Cómo funciona esto internamente?**

Cuando alguien pone {{nombre}} o la directiva data-ng-bind="nombre" en el DOM esto internamente llama a \$watch. Este método tiene dos argumentos, uno es el path de propiedades dentro del \$scope que queremos escuchar (en nuestro caso nombre), y el otro es una función que se invoca cada vez que el valor de la propiedad referenciada por el primer argumento cambia. Es decir cada vez que \$scope.nombre modifique su valor se ejecutará esta función.

Pero para que angular se entere de que el valor de nombre ha cambiado y sea capaz de llamar a la función alguien tiene que llamar a \$digest. Este método lo que hace es recorrer todo el \$scope y si algún valor ha variado respecto a la anterior ejecución mira los bindings registrados con \$watch y lanza la función correspondiente para que actualice el DOM. En el ejemplo de arriba la llamada la realiza ngClick que al terminar siempre llama a \$digest.

### **One-Time Binding (disponible en angularjs +1.3):**

Existe una variante de One-Way binding que es mucho más óptima. Este tipo de binding tiene como objetivo mostrar una información y una vez el valor esté mostrado y estabilizado ya nunca más se va a refrescar. Internamente lo que se hace es eliminar el binding una vez mostrado el valor de forma que se liberan los recursos que se encargan de sincronizar los datos. **Se emplea la notación {{:: }}**

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body ng-app>
  <input type="text" ng-model="name"/> <!-- Two binding -->
  <div>Hola {{name}}</div> <!-- One Way -->
  <div>Hola {{::name}}</div> <!-- One Time -->
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular.min.js"></script>
</body>
</html>

```

## Ejemplo de databinding

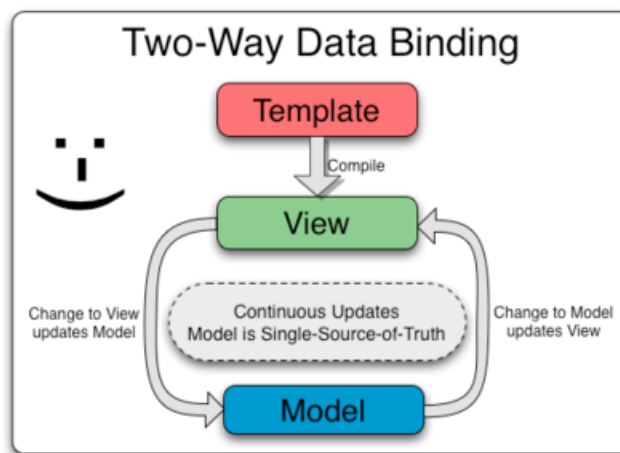
<http://codepen.io/jmortega/pen/NbKday>

### Two-way binding:

En este segundo caso la información fluye desde el scope hacia la parte visual (igual que en "oneway binding") y también desde la parte visual hacia el scope. La implementas por medio de la directiva **ngModel**.

`<input type="text" ng-model="miDato" />`

En este caso cuando el modelo cambie, el dato que está escrito en el campo de texto (o en el elemento de formulario donde lo uses) se actualizaría automáticamente con el nuevo valor. Además, gracias al doble binding (two-way) en este caso, cuando el usuario cambie el valor del campo de texto el scope se actualizará automáticamente.



### Por qué el binding es tan útil

Piensa en una aplicación que realizases con Javascript común (sin usar librería alguna), o incluso con jQuery que no tiene un binding automático. Piensa todo el código que tendrías que hacer para implementar ese pequeño ejemplo. Suscribirte eventos, definir las funciones manejadoras para que cuando cambie el estado del campo de texto, volcarlo sobre la página. Crear el manejador del botón para que cuando lo pulses se envíe el nuevo texto "hola" tanto al campo de texto como a la página, etc.

No estamos diciendo que sea difícil hacer eso "a mano", seguro que la mayoría es capaz de hacerlo en jQuery, pero fíjate que para cada una de esas pequeñas tareas tienes que agregar varias líneas de código, definir eventos, manejadores, especificar el código para que los datos viajen de un lugar a otro. Quizás no es tan complicado, pero sí es bastante laborioso y en aplicaciones complejas comienza a resultar un infierno tener que estar pendiente de tantas cosas.

Otra ventaja, aparte del propio tiempo que ahorras, es una limpieza de código muy destacable, ya que no tienes que estar implementando muchos eventos ni tienes necesidad de enviar datos de un sitio a otro. Tal como te viene un JSON de una llamada a un servicio web lo enganchas al scope y automáticamente lo tienes disponible en la vista, lo que es una maravilla. Esto se ve de manera notable en comparación con otros frameworks como BackboneJS.

O sea que el doble binding de Angular nos ahorra mucho trabajo, pero ojo, quizás no sea necesario para cualquier tipo de aplicación. Si tus necesidades no son muy grandes igual no consigues adelantar tanto. Incluso en aplicaciones con inmensa cantidad de datos puede que el doble binding te pueda ralentizar un poco la aplicación porque toda esa gestión del binding se haga pesada, igual de manera innecesaria. En esos casos quizás otra librería funcione mejor que AngularJS. Sin embargo, la inmensa mayoría de las aplicaciones se beneficiarán de ese doble binding.

La **directiva ngModel** es la que permite la actualización en ambos sentidos y se puede asignar a un input, select o textarea y se encarga de detectar cuando el valor que contiene este control cambia y actualizar el valor del \$scope correspondiente. Luego lanza en método \$digest para que se refresquen todos los valores que sea necesario actualizar.

```
<div ng-app="databinding" ng-controller="MainCtrl">
<div>
  Hola, {{name}}!
</div>
<div>
  Hola, <span ng-bind="name"></span>!
</div>
<div>
  <label>Nombre: <input type="text" ng-model="name" /></label>
</div>
</div>
```

Al usar **ng-model** AngularJS *binda* el valor de scope.name al del elemento input , a través de la directiva ngModel . Aquí es donde se realiza un **two-way data binding**. Se puede observar sencillamente ya que, si modificamos el valor del input, los otros dos elementos modifican el texto.

## **Definición de controladores para la aplicación**

---

Angular utiliza un sistema de databinding declarativo bidireccional para enlazar el \$scope con la vista, utilizando para ello lo que se conoce como directivas. Las directivas tienen el aspecto de elementos o atributos en el código HTML, pero son interpretadas por Angular al generar las vistas, lo que permite añadir nuevos comportamientos.

Una directiva *de atributo* se utiliza como si fuese un atributo más de un elemento HTML y puede llevar parámetros o no, pero además tiene la ventaja de poder hacer referencia al \$scope para realizar acciones o leer datos. En el ejemplo que se muestra en la **¡Error! No se encuentra el origen de la referencia.** aparecen las directivas ng-controller, ng-model y ng-click, que permiten enlazar un controlador a un elemento HTML (línea 1), enlazar la propiedad *person.name* a un cuadro de texto (línea 2) e invocar la función \$scope.save() al pulsar un botón (línea 3):

```

1 <div ng-controller="SampleController">
2   <input ng-model="person.name"/>
3   <button ng-click="save()"/>
4 </div>

```

Las directivas de elemento se utilizan como cualquier otro elemento HTML y simplifican los procesos de creación de fragmentos de código HTML. A alto nivel, las directivas son marcadores al igual que atributos, etiquetas y nombres de clase, que le dicen a Angular cómo adjuntar un determinado comportamiento a un elemento del DOM (transformación, reemplazo, etc.).

Las directivas más utilizadas en aplicaciones Angular son las que se especifican a continuación:

- La directiva *ng-app* es la responsable del arranque de la aplicación, definiendo el ámbito de actuación de la misma (la etiqueta HTML en la que se defina), debido a que Angular permite hacer uso de distintas aplicaciones dentro de la misma página.
- La directiva *ng-controller* define qué controlador será cargado en la vista.
- La directiva *ng-model* es responsable de:
  - o Vincular la vista y el modelo (por ejemplo, a través de otras directivas como *input*, *textarea* o *select*).
  - o Proporcionar comportamiento de validación.
  - o Mantener el control de estados (*valid/invalid*, *dirty/pristine*, *touched/untouched*, errores).
  - o Registrar el control con el formulario padre.

## Módulo, objeto module en AngularJS

Los módulos son una de las piezas fundamentales en el desarrollo con AngularJS y nos sirven para organizar el código en esta librería. Lo puedes entender como un contenedor donde sitúas el código de los controladores, directivas, etc.

La incorporación de módulos en AngularJS es motivada por la realización de aplicaciones con mejores prácticas. Son como contenedores aislados, para evitar que tu código interactúe con otros scripts Javascript que haya en tu aplicación (entre otras cosas dejarán de producirse colisiones de variables, nombres de funciones repetidos en otras partes del código, etc.). Los módulos también permiten que el código sea más fácilmente reutilizable, entre otras ventajas.

En Angular todo gira en torno a la **inyección de dependencias** y piensa que cualquiera de las funciones que puedes definir en un módulo son inyectables.

## Inyección de dependencias

A la hora de definir cualquier componente, ya sea un controlador o un servicio, se debe indicar de qué otros componentes depende para que Angular se encargue de proporcionárselos a través de su función constructora (o de la función factoría que lo construya).

Todos los componentes registrados en angular son **Singleton**, es decir, sólo existe una instancia de ellos en la aplicación y si hay varios componentes que dependen de un mismo objeto, todos recibirán la misma instancia del mismo. Esto es lo que permite utilizar los servicios para almacenar estados, ya que dos controladores que dependan de un mismo servicio estarán utilizando el mismo objeto y, por tanto, podrán compartir información a través de él.

## Indicar el módulo al arrancar la aplicación

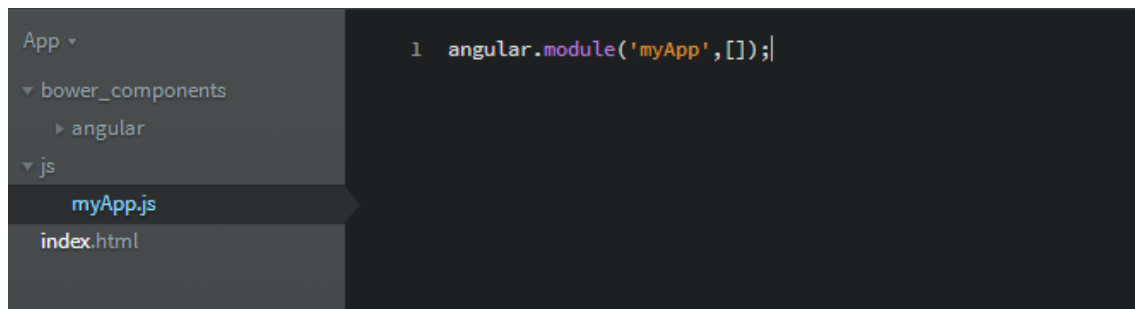
Cuando arrancas una aplicación AngularJS, usando ng-app generalmente pondrás el nombre del módulo que quieres ejecutar en tu aplicación.

```
<HTML ng-app="miAplicacion">
```

## Creación de módulos

Desde Javascript crearás los módulos usando el método `angular.module()` e indicándole una serie de parámetros. Echa un vistazo a esta sintaxis.

**`angular.module('miAplicacion', [ ... ], function(...){ ... })`**



En un archivo JavaScript simplemente tienes que escribir esa línea. El primer parámetro de la función `module` es el nombre de tu módulo y el segundo un array de string con los módulos de los que depende tu módulo, por ejemplo un módulo como `ngRoute`, que nos sirve para gestionar toda la casuística de routing en Angular. **Ambos parámetros son requeridos.**

La variable "angular" la tienes como variable global cuando cargas AngularJS, dentro tiene un objeto que estará disponible en cualquier parte de tu código. Luego ves "module", que es un método del objeto "angular" y que sirve para crear el módulo. El primer argumento es el nombre del módulo, que corresponde con el nombre de tu aplicación. En el segundo parámetro puedes indicar una serie de módulos adicionales, separados por comas, que serían tus dependencias.

Pueden ser módulos de otros autores o puede que tú mismo hayas decidido organizar tu código en diferentes módulos. El tercer parámetro es opcional y en él colocamos una función que sirve para configurar AngularJS.

Para comenzar crearás tus módulos probablemente de una manera sencilla, tal como puedes ver a continuación:

**`angular.module('nuevaApp', []);`**

En el código anterior "nuevaApp" es el nombre de tu módulo. Como se comentó anteriormente, ese módulo se debe indicar en el arranque de tu aplicación, comúnmente en la directiva `ngApp`.

```
<html ngApp="nuevaApp">
```

## El objeto module

Esta llamada a `angular.module()` te devuelve un objeto `module`, que tiene una serie de métodos como `config`, `run`, `provider`, `service`, `factory`, `directive`, `controller`, `value`, etc. que son los que nos sirven para controlar la lógica de presentación y la lógica de negocio. Verás dos variantes de código en este sentido:

### Opción 1

Puedes "cachear" (almacenar) el módulo en una variable y luego usarlo para crear tus controladores, factorías, etc.

```
var nuevaApp = angular.module('nuevaApp', []);
nuevaApp.controller( ... );
```

### Opción 2

Aunque en muchos casos verás que ese objeto module ni siquiera se guarda en una variable, sino que se encadenan las creaciones de los controladores o factorías todo seguido. A esto último se suele llamar estilo "Fluent".

```
angular
.module('nuevaApp', [])
.controller( ... )
.factory( ... );
```

### Trabajar con un módulo

La mejor forma de ver cómo crear objetos dentro de un módulo, es con un sencillo ejemplo, para ello vamos a crear un controlador al que le vamos a crear un método helloWorld, que escriba por consola "Hello world" y bindear este a la vista de forma que podamos asociarlo al click de un button.

En este ejemplo vemos el uso de \$inject para inyectar las dependencias que necesitamos. **\$inject** es un array con los nombres de los servicios a inyectar.

### myController.js



The screenshot shows a code editor with a file explorer on the left and code in myController.js on the right. The file explorer shows a tree structure with 'App' as the root, containing 'bower\_components', 'js', and 'index.html'. Under 'js', there are 'myApp.js' and 'myController.js'. The code in myController.js is as follows:

```
1 (function (myApp) {
2   "use strict";
3
4   var controller = function (scope, window) {
5     scope.helloWorld = function () {
6       window.console.log("Hello Word");
7     };
8   };
9   controller.$inject = ['$scope', '$window'];
10
11   myApp.controller('myController', controller);
12
13 }(angular.module('myApp')));
```

Observa en el código los siguientes detalles.

1. El controlador está en un archivo separado y es así como tienes que escribir cada uno de los componentes en un módulo Angular, entre otras cosas porque te permiten separar cada cosa en un archivo diferente y poder testear una sola unidad.
2. Observa que todo el código está en una función autoejecutable que hace que todo se ejecute en el ámbito privado de esta y a la que se le pasa como parámetro tu módulo.
3. La función privada controller, recibe 2 parámetros "scope y window", y dentro de ella a scope que sería nuestro enlace con la vista (MVVM o MVW) se le define una función "helloWorld", a la que se le implementa la funcionalidad de escribir en consola "Hello World".



4. A la función controller se le define una variable estática llamada \$inject que es necesaria para decirle a Angular, que objetos son los que tiene que inyectar. Si quieres ampliar más sobre la inyección de dependencias en Angular te recomiendo estas lecturas.

5. Por último a nuestro módulo se le agrega un controlador con nombre myController y que apunta a la función privada controller. Es una buena práctica que en todos tus objetos utilices un prefijo tal y como lo estamos haciendo en nuestro caso "my". Angular recomienda que los prefijos sean de dos letras, aunque nadie te impide que utilices 3, 4 o las que quieras, pero hazlo para evitar colisiones con otros módulos en la resolución de la inyección de dependencias.

### index.html



```
1 <!DOCTYPE>
2 <html ng-app="myApp">
3   <head></head>
4   <body>
5     <div ng-controller="myController">
6       <button ng-click="helloWorld()">Hello World</button>
7     </div>
8     <script src="bower_components/angular/angular.js"></script>
9     <script src="js/myApp.js"></script>
10    <script src="js/myController.js"></script>
11  </body>
12 </html>
```

Por último vamos a unir la vista a nuestro código y ver la funcionalidad agregada, para ello tenemos que hacer lo siguiente:

1. Agregar a nuestra vista los dos script necesarios myApp.js y myController.js aparte de angular.js. Esto lo puedes hacer así o bien con herramientas como [require.js](#).
2. En la etiqueta html agregamos un atributo ng-app con el nombre de nuestro módulo en este caso "MyApp" y esto es requerido para poder trabajar con Angular, recuerda el capítulo Bootstrap.
3. En un etiqueta div agregamos un atributo/directiva ng-controller donde establecemos el nombre de nuestro controlador myController.
4. Por último a este div le agregamos un button y un atributo/directiva ng-click desde donde invocamos a la función de nuestro scope helloWorld.

### Controladores, controller en AngularJS

Los controladores nos permiten mediante programación implementar la lógica de la presentación en AngularJS. En ellos podemos mantener el código necesario para inicializar una aplicación, gestionar los eventos, etc. Podemos decir que gestionan el flujo de la parte del cliente, lo que sería programación para implementar la funcionalidad asociada a la presentación.

En líneas generales podemos entender que los controladores nos sirven para separar ciertas partes del código de una aplicación y evitar que escribamos Javascript en la vista. Es decir para que el HTML utilizado para la presentación no se mezcle con el Javascript para darle vida.

Un controlador puede ser agregado al DOM mediante la directiva ngController (con el atributo ngcontroller en la etiqueta HTML) y a partir de entonces tendremos disponible en esa etiqueta (y todas sus hijas) una serie de datos. Esos datos son los que necesitarás en la vista para hacer la parte de presentación y es lo que asociaríamos en el MVC con el "modelo".

A los controladores podemos inyectarles valores o constantes. Como su propio nombre indica, las constantes son las que no van a cambiar a lo largo del uso de la aplicación y los valores son aquellas variables cuyo dato puede cambiar durante la ejecución de una aplicación.

También podremos inyectar servicios y factorías, componentes muy parecidos entre sí y que veremos más adelante.

Sin entrar a describir por el momento lo que son "services" y "factories" puedes simplemente recordar que su utilidad es la de encapsular código.

Los controladores son clases que se encargan de gestionar el flujo de ejecución de la capa de presentación. En concreto se encarga de añadir y manipular la información que está en el \$scope y añadir funciones de código JavaScript que luego será invocado desde los eventos del DOM.

Todo el código que se encargue de modificar el DOM no debería estar en esta capa. Esta capa debe contener simplemente la lógica de presentación de forma que sea el punto más importante sobre el que ejecutar las pruebas. Para pasar información del HTML a JavaScript o viceversa deberá añadirse al \$scope como propiedades los valores que se precisen o como funciones el código que maneje los diferentes eventos a modo de comandos.

**Los controladores deben ser muy sencillos ya que solo deben encargarse de la gestión de la lógica de presentación.** Si fuese necesario añadir más código, hay que pensar en refactorizar y poner este código en factorías, servicios, filtros, decoradores. Además de todo esto también se pueden inyectar valores predefinidos (empiezan por \$ como \$scope, \$http, \$rootScope, \$window, ... )

El **Controlador (controller)** responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud sobre la información.

Cuando un controlador en AngularJS se añade al DOM mediante la directiva **ng-controller** Angular instancia un *objeto new Controller* usando el constructor donde se le puede pasar por parámetro el \$scope (ámbito).



### Separación del modelo de datos.

Como en cualquier aplicación [SPA](#) con Angular nuestros datos en un porcentaje muy alto provienen de servicios [RESTful](#) tanto propios como de terceros y puesto que Angular permite hacer binding a datos, a expresiones y a métodos. Como buena práctica siempre tienes que separar tus datos del resto de la información, por una sencilla razón evitar transportar información innecesaria tanto desde el cliente al servidor como de este hacia el cliente, así como el no transporte de nulos, ya que estudios que hemos realizado, sino transportamos nulos ahorramos más de un 25% de tráfico de red.

### Qué hacer y qué no hacer desde los controladores

Ahora vamos a ver qué tipo de operaciones debemos incluir dentro del código de los controladores.

Entre otras cosas y por ahora debes saber:

- Los controladores son adecuados para inicializar el estado del scope para que nuestra aplicación tenga los datos necesarios para comenzar a funcionar y pueda presentar información correcta al usuario en la vista.

- Además es el lugar adecuado para escribir código que añada funcionalidades o comportamientos (métodos, funciones) al scope.
- Con el controlador no deberías en ningún caso manipular el DOM de la página, pues los controladores deben de ser agnósticos a cómo está construido el HTML del documento donde van a estar trabajando.
- Tampoco son adecuados para formatear la entrada de datos o filtrar la salida, ni intercambiar estados entre distintos controladores. Para hacer todo eso existen dentro de AngularJS diversos componentes especializados.

## Pasos para crear controladores en AngularJS

El proceso de crear y usar un controlador es sencillo, pero no inmediato, requiere un pequeño guion de tareas que paso a detallar para que tengas claro de antemano lo que vamos a hacer. Es más largo expresarlo con texto que con código y aunque al principio parezca bastante, en seguida lo harás mecánicamente y no tendrás que pensar mucho para ello. Básicamente para poner nuestro primer controlador en marcha necesitamos:

- Crear un módulo (module de AngularJS)
- Mediante el método `controller()` de nuestro "module", asignarle una función constructora que Angular usará cuando deba crear nuestro controlador
- Usar la directiva `ng-controller`, asignándole el nombre de nuestro controlador, en el HTML. Colocaremos esa directiva en el pedazo del DOM donde queremos tener acceso al scope.

```
var app = angular.module("miapp", []);

app.controller("miappCtrl", function(){
    var scope = this;
    scope.datoScope = "valor";
    scope.metodoScope = function(){
        scope.datoScope = "otro valor";
    }
});
```

## Directivas y expresiones en AngularJS

Las directivas son la parte más interesante de AngularJS, ya que nos permiten **extender HTML** para que realice todo lo que nosotros queramos.

Una primera aproximación a los componentes que forman parte de las aplicaciones básicas que podremos desarrollar con AngularJS, las directivas y las expresiones.

Hemos visto que Angular tiene como característica que extiende el HTML, pudiendo llegar a programar funcionalidad de la aplicación sin necesidad de escribir código Javascript. Ahora vamos a ver de manera más detallada algunos de los componentes típicos que encontramos en las aplicaciones desarrolladas con esta potente librería.

Al trabajar con AngularJS seguimos desarrollando encima del código HTML, pero ahora tenemos otros componentes útiles que agregan valor semántico a tu aplicación. De alguna manera estás enriqueciendo el HTML, por medio de lo que se conoce como "directiva".

## Qué son las directivas

Las directivas son etiquetas en un elemento del DOM (*un atributo, un nombre de elemento, comentario o clase CSS*) que indican al compilador de HTML de AngularJS (**\$compile**) cuál debe ser su comportamiento.

Las directivas son nuevos "comandos" que vas a incorporar al HTML y los puedes asignar a cualquiera de las etiquetas por medio de atributos. Son como marcas en elementos del DOM de tu página que le indican a AngularJS que tienen que asignarles un comportamiento determinado o incluso transformar ese elemento del DOM o alguno de sus hijos.

Cuando se ejecuta una aplicación que trabaja con Angular, existe un "HTML Compiler" (Compilador HTML) que se encarga de recorrer el documento y localizar las directivas que hayas colocado dentro del código HTML, para ejecutar aquellos comportamientos asociados a esas directivas.

AngularJS nos trae una serie de directivas "de fábrica" que nos sirven para hacer cosas habituales, así como tú o terceros desarrolladores pueden crear sus propias directivas para enriquecer el framework.

## Directiva ngApp (ng-app)

Esta es la marca que indica el elemento raíz de tu aplicación. Se coloca como atributo en la etiqueta que deseas que sea la raíz. Es una directiva que autoarranca la aplicación web AngularJS.

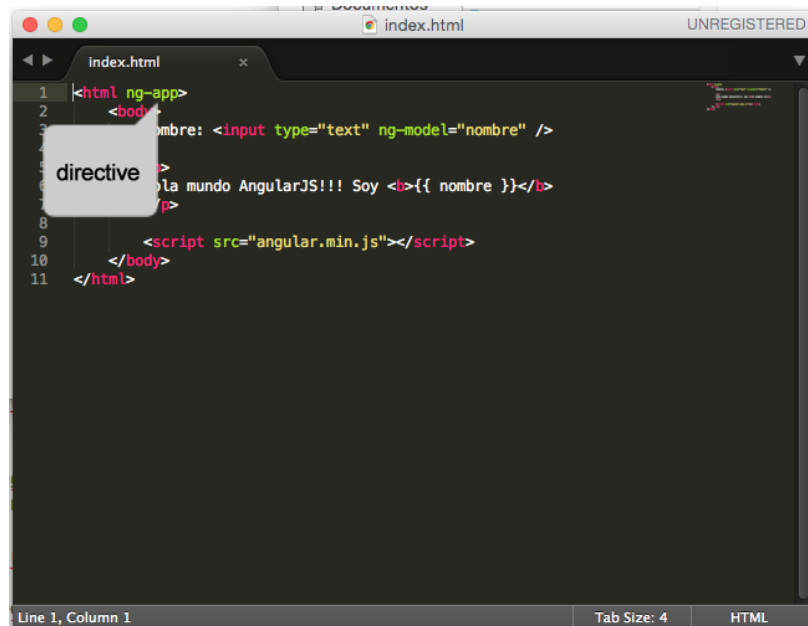
Lo más común es ponerlo al principio de tu documento HTML, en la etiqueta HTML o BODY, pero también lo podrías colocar en un área más restringida dentro del documento en otra de las etiquetas de tu página.

### <html ng-app>

A nivel de Javascript las directivas las encontrarás nombradas con notación "camel case", algo como ngApp. En la documentación también las encuentras nombradas con camel case, sin embargo, como el HTML no es sensible a las mayúsculas y minúsculas no tiene tanto sentido usar esa notación y por ello se separan las palabras de las directivas por un guión "-".

Opcionalmente ngApp puede contener como valor un módulo de AngularJS a cargar. Esto lo veremos más adelante cuando trabajemos con módulos.

**ng-app** es la directiva principal de AngularJS. Determina el elemento root (raíz) de la aplicación. Típicamente se suele poner en <html> o <body>

A screenshot of a code editor window titled 'index.html'. The code is as follows:

```
1 <html ng-app>
2 <body>
3   nombre: <input type="text" ng-model="nombre" />
4   <p>
5     ola mundo AngularJS!!! Soy <b>{{ nombre }}</b>
6   </p>
7   <script src="angular.min.js"></script>
8 </body>
9 </html>
```

A grey callout box with the word 'directive' points to the 'ng-model' attribute in the first line of the code. The status bar at the bottom shows 'Line 1, Column 1', 'Tab Size: 4', and 'HTML'.

## Directiva ngModel

La directiva ngModel informa al compilador HTML de AngularJS que que estás declarando una variable de tu modelo. Las puedes usar dentro de campos INPUT, SELECT, TEXTAREA (o controles de formulario personalizados).

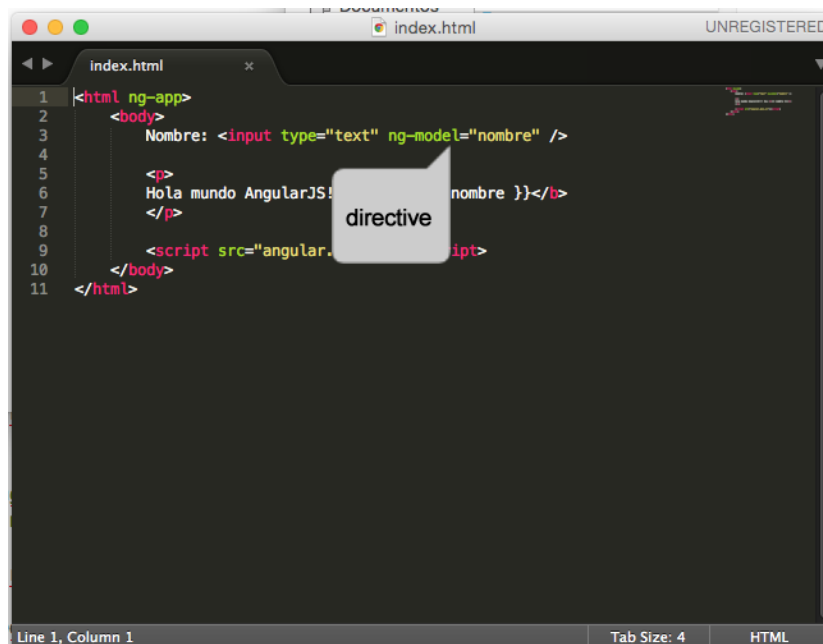
La indicas con el atributo del HTML ng-model, asignando el nombre de la variable de tu modelo que estás declarando.

**`<input type="text" ng-model="busqueda">`**

Con eso le estás diciendo al framework que esté atento a lo que haya escrito en ese campo de texto, porque es una una variable que vas a utilizar para almacenar algo y porque es importante para tu aplicación.

Técnicamente, lo que haces con ngModel es crear una propiedad dentro del "scope" (tu modelo) cuyo valor tendrá aquello que se escriba en el campo de texto. Gracias al "binding" cuando modifiques ese valor en el scope por medio de Javascript, también se modificará lo que haya escrito en el campo de texto. Aunque todo esto se entenderá mejor un poco más adelante cuando nos metamos más de lleno en los controladores.

**ng-model** es la directiva que enlaza (binds) un campo de formulario *input*, *select*, *textarea* con el ámbito(scope) de la vista.



```
1 <html ng-app>
2   <body>
3     Nombre: <input type="text" ng-model="nombre" />
4
5     <p>
6       Hola mundo AngularJS! {{nombre }}</p>
7     </p>
8
9     <script src="angular.js"></script>
10   </body>
11 </html>
```

## Directiva ngInit

Esta directiva nos sirve para inicializar datos en nuestra aplicación, por medio de expresiones que se evaluarán en el contexto actual donde hayan sido definidas. Dicho de otra manera, nos permite cargar cosas en nuestro modelo, al inicializarse la aplicación.

Así de manera general podemos crear variables en el "scope", inicializarlas con valores, etc. Para que en el momento que las vayas a necesitar estén cargadas con los datos que necesitas.

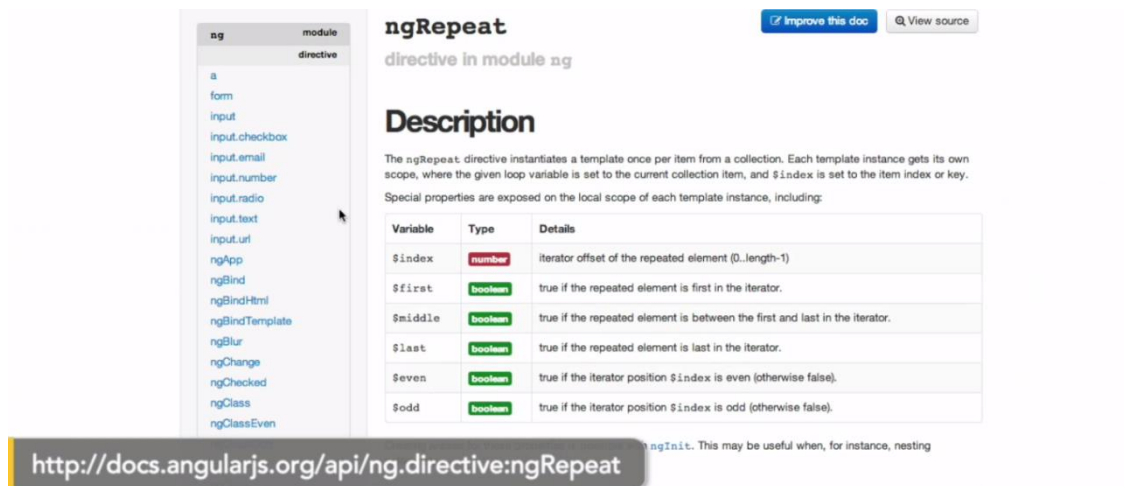
El scope mantiene los datos de tu aplicación, lo que en el paradigma de la programación MVC se llama "modelo". Así pues, si digo que creo variables en el scope lo puedes entender como la definición de un dato de tu aplicación, que se guarda en el denominado "modelo" del MVC.

**<div ng-app ng-init="miArrayDatos = [];">**

Con esto consigues que tu aplicación inicialice en el scope un dato llamado miArrayDatos como un array vacío. Pero no le prestes demasiada atención al hecho de haber colocado la directiva ngInit dentro de la misma etiqueta que inicializa la aplicación, pues podría ir en cualquier otra etiqueta de tu HTML. Realmente, colocarla en esa división marcada con ngApp es considerado una mala práctica. Ten en cuenta lo siguiente cuando trabajes con ngInit:

El único caso apropiado donde se debería de usar ngInit es en enlace de propiedades especiales de ngRepeat. Si lo que quieres es inicializar datos en tu modelo para toda la aplicación, el lugar apropiado sería en el controlador. Enseguida vemos un ejemplo de uso apropiado, cuando conozcamos la directiva ngRepeat.

## Directiva ngRepeat



**ngRepeat**  
directive in module ng

**Description**

The ngRepeat directive instantiates a template once per item from a collection. Each template instance gets its own scope, where the given loop variable is set to the current collection item, and \$index is set to the item index or key. Special properties are exposed on the local scope of each template instance, including:

Variable	Type	Details
\$index	number	iterator offset of the repeated element (0..length-1)
\$first	boolean	true if the repeated element is first in the iterator.
\$middle	boolean	true if the repeated element is between the first and last in the iterator.
\$last	boolean	true if the repeated element is last in the iterator.
\$even	boolean	true if the iterator position \$index is even (otherwise false).
\$odd	boolean	true if the iterator position \$index is odd (otherwise false).

<http://docs.angularjs.org/api/ng.directive:ngRepeat>

Esta directiva te sirve para implementar una repetición (un bucle). Es usada para repetir un grupo de etiquetas una serie de veces. Al implementar la directiva en tu HTML tienes que decirle sobre qué estructura se va a iterar. ngRepeat se usa de manera muy habitual podríamos decir que es como un recorrido for-each en el que se itera sobre cada uno de los elementos de una colección.

La etiqueta donde has colocado el atributo ng-repeat y todo el grupo de etiquetas anidadas dentro de ésta, funciona como si fuera una plantilla. Al procesarse el compilador HTML de AngularJS el código HTML de esa plantilla se repite para cada elemento de la colección que se está iterando.

Dentro de esa plantilla tienes un contexto particular, que es definido en la declaración de la directiva, que equivale al elemento actual en el bucle. Se ve mejor con un ejemplo.

```
<p ng-repeat="elemento in miColeccion">
  Estás en: <span>{{elemento}}</span>
</p>
```

El dato miColeccion sería un dato de tu modelo, habitualmente un array sobre el que puedas iterar, una vez por cada elemento. Pero también podría ser un objeto y en ese caso la iteración se realizaría en cada una de sus propiedades.

En lo relativo al contexto propio del bucle te puedes fijar que dentro de la iteración podemos acceder al dato "elemento", que contiene como valor, en cada repetición, el elemento actual de la colección sobre el que se está iterando.

Vamos a mostrar cómo podemos trabajar con ngRepeat en conjunto con ngInit, para completar la explicación de punto anterior.

```
<p ng-repeat="elemento in miColeccion" ng-init="paso=$index;">
  Elemento con id {{paso}}: <span>{{elemento}}</span>
</p>
```

La directiva ngRepeat maneja una serie de propiedades especiales que puedes inicializar para el contexto propio de cada repetición. Para inicializarlas usamos la directiva ngInit indicando los nombres de las variables donde vamos a guardar esas propiedades. En este caso estamos indicando que dentro de la repetición vamos a mantener una variable "paso" que tendrá el valor de \$index, que equivale al número índice de cada repetición. Es decir, en la primera iteración paso valdrá cero, luego valdrá uno y así todo seguido. Igual que tienes \$index, Angular te proporciona otras propiedades útiles como \$first (que valdrá true en caso que sea la primera iteración) o \$last (true solo para la última iteración)

## Orden de los elementos en la repetición

Ahora veamos cómo expresar el orden de visualización de los elementos en la repetición. Para ello usamos la palabra "orderBy", seguido de ":" y el campo sobre el que se debe ordenar. Opcionalmente colocamos después si el orden es ascendente o descendente.

**ng-repeat="elem in elementos | orderBy:field:true"**

En este caso ordenamos los elementos por el campo "name". Lógicamente debe de ser un atributo del objeto que estás recorriendo en la colección. Luego opcionalmente colocamos un booleano para indicar el orden. De manera predeterminada el orden es ascendente. Si el booleano es true, entonces el orden se hace descendente y si es false, entonces es ascendente.

Hemos visto que dentro del bucle de ng-repeat se crea una nueva variable definida por nosotros llamada provincia. AngularJS también crea otra serie de variables que podemos usar:

- \$index: Un número que indica el nº de elementos(de 0 a n-1).
- \$first : Vale true si estamos en el primer elemento del bucle.
- \$last : Vale true si estamos en el último elemento del bucle.
- \$middle : Vale true si **no** estamos ni en el primer ni en el último elemento del bucle.
- \$even : Vale true si estamos en un elemento con \$index par. (El 0 es par).
- \$odd : Vale true si estamos en un elemento con \$index impar. (El 0 es par).

Todas estas variable son muy útiles para cambiar por ejemplo el estilo de la filas.

De forma que las filas pares sean de fondo blanco y la impares de fondo negro:

**<tr ng-repeat="result in results" style="background-color:{{\$even?'white':'black'}}">**

## Claves primarias

Por defecto, si cambiamos el array de datos ,AngularJS automáticamente borrará todos los tag y los volverá a crear con todos los datos. Si hay muchos datos suele ser poco eficiente. Para paliarlo, AngularJS permite que le digamos qué columna es una clave única de nuestros objeto y así él puede optimizar el proceso.

La forma de hacerlo es añadir el texto track by y la propiedad que actúa como clave primaria.

La directiva entonces quedaría así:

**<tr ng-repeat="result in results track by result.idResult">**

Creamos una lista de libros mediante un array con la directiva **ng-init** que se ejecuta al crear el elemento HTML: ng-init="libros=['El Juego de Ender','Juego de Tronos','I robot']"  
Después utilizamos la directiva **ng-repeat** para iterar una colección de datos mediante un loop y tener así una duplicación de parte de la interfaz gráfica pero con diferentes datos. En definitiva una especie de **for each**.

```
<html ng-app>
  <body>
    <div ng-init="libros=['El Juego de Ender','Juego de Tronos','I robot']">
      <h1>Listado de libros</h1>

      <ul>
        <li ng-repeat="libro in libros">{{ libro }}</li>
      </ul>
    </div>
    <script src="angular.min.js"></script>
```



```
</body>
</html>
```

El uso de la directiva **ng-init** es muy similar a la declaración de variables dentro de código JavaScript. Por suerte, muy pronto veremos que no vamos a inicializar variables de esta manera, ya que como imaginaréis una vista no es el mejor lugar para inicializar una variable.

Por su parte, hemos visto que únicamente hemos declarado una plantilla para la colección, y ha sido la directiva **ng-repeat** quien se ha encargado de instanciarla para cada elemento de la colección. El formato de expresión más utilizado para esta directiva es **variable in expression**, donde **variable** es el nombre de una variable definida por el usuario y **expression** es el nombre de nuestra colección.

## Listas anidadas en AngularJS con ng-repeat

Desde la versión del framework 1.2, Angular nos provee con dos nuevas directivas diseñadas para facilitarnos esto. Estas dos nuevas directivas son **ng-repeat-start** y **ng-repeat-end**. Utilizando ambas junto con la clásica **ng-repeat** podemos conseguir una visualización de datos anidados **sencilla** pero realmente **potente** y **adaptable**.

### Las directivas ng-repeat, ng-repeat-start y ng-repeat-end:

- **ng-repeat**

Una de las primeras directivas con las que todo programador se familiariza en el framework AngularJS es **ng-repeat**. Cumple las funciones de un loop que recorre una colección de datos (array u objeto) repitiendo la etiqueta HTML sobre el que está aplicado (y sus contenidos) por cada elemento en dicha colección.

- **ng-repeat-start**

Esta directiva extiende el ámbito del loop a un bloque que se extiende hasta la directiva asociada **ng-repeat-end**. De esta manera, repetirá **todo el código HTML** (incluyendo la etiqueta HTML sobre el que esté aplicada) hasta el final del bloque.

- **ng-repeat-end**

Marca el final del bloque **ng-repeat-start**

## Proyecto angular-todo-nested-list

Para demostrar el poder de estas directivas podemos ver una sencilla web app de **To Do** o lista de tareas, con la funcionalidad adicional de que las tareas estarán agrupadas en **categorías**. Perfecto para listas las tareas y categorías de forma **anidada** gracias a `ng-repeat-start` y `ng-repeat-end`.

### El modelo de datos

Para esta demo hemos optado por un modelo de datos simplificado al máximo, vamos a tener dos tipos de datos:

- **Categorías:** Simplemente tienen un nombre y una lista de tareas asociadas. Además, para la visualización de los datos hemos añadido un *booleano* para marcar si la debemos mostrar expandida o no.
- **Tareas:** Nombre de la tarea y un *booleano* que indica si está finalizada ya finalizada. Las tareas se encuentran anidadas dentro de las categorías. Este es nuestro modelo de datos inicial de ejemplo, en formato JSON:

```
function DataModel() {
  this.data = [
    { name: 'Personal', expanded: true,
    },
    { name: 'Work', expanded: false,
      items: [
        { name: 'Curso angular', completed: false }
      ]
    },
  ];
}
```

### La vista

Es en la vista donde toda la magia de las directivas **ng-repeat-start** y **ng-repeat-end** sucede. Hemos marcado con comentarios los bloques definidos por estas directivas, pero creemos que viendo el resultado final es fácil entender qué es lo que cada bloque está renderizando al recorrer el array del modelo de datos.

```
<!-- Repeat start, beggining of category group -->
<div ng-repeat-start="category in data" class="animate-repeat-category">
  <button class="btn btn-primary btn-block btn-sm" ng-click="toggleCategory(category)">
    {{ category.name }}
    <span class="expand-indicator" ng-show="category.expanded">&mdash;</span>
    <span class="expand-indicator" ng-show="!category.expanded">+</span>
  </button>
</div>

<!-- Item group inside each category -->
<div class="list-items animate-show-item" ng-show="category.expanded">
  <ul>
    <li ng-repeat="item in category.items" class="animate-repeat-item">
      <span ng-class="{done:item.completed}">{{ item.name }}</span>
      <button ng-show="!item.completed" ng-click="completeTask(item)" class="btn btn-success
btn-xs">done</button>
      <button ng-show="item.completed" ng-click="uncompleteTask(item)" class="btn btn-warning
btn-xs">reset</button>
      <button ng-click="deleteTask(category, item)" class="btn btn-danger btn-xs">del</button>
    </li>
  </ul>
</div>
```

```
<div class="no-items"><small ng-show="category.expanded &&
category.items.length==0">NO ITEMS!</small></div>
</div>

<!-- Repeat end, closing category group -->
<div ng-repeat-end>
  <button ng-show="category.expanded" class="btn btn-primary btn-block btn-sm animate-show-
item" disabled="disabled"> / {{ category.name }}</button>
  <hr>
</div>
```

Personal —

/ Personal

Work —

- Curso angular

del

done

/ Work

Add new category:

type category name

add

Add new task to category:

type task name

Personal ▼

## Directiva ngClick

Vamos a ver ahora una explicación de la directiva ngClick. Como podrás imaginarte es utilizada para especificar un evento click. En ella pondremos el código (mejor dicho la expresión) que se debe ejecutar cuando se produzca un clic sobre el elemento donde se ha colocado la directiva.

Típicamente al implementar un clic invocarás una función manejadora de evento, que escribirás de manera separada al código HTML.

```
<input type="button" value="Haz Clic" ng-click="procesarClic()">
```

Esa función procesarClic() la escribirás en el controlador, factoría, etc. Sería el modo aconsejado de proceder, aunque también podrías escribir expresiones simples, con un subconjunto del código que podrías escribir con el propio Javascript. Incluso cabe la posibilidad de escribir varias expresiones si las separas por punto y coma.

```
<input type="button" value="haz clic" ng-click="numero=2; otraCosa=dato " />
```

No difiere mucho a como se expresan los eventos clic en HTML mediante el atributo onclick, la diferencia aquí es que dentro de tus expresiones podrás acceder a los datos que tengas en tu modelo.

**Nota:** Aunque técnicamente pueda escribir expresiones directamente en el código HTML, en el valor del atributo ng-click, tienes que evaluar con cuidado qué tipo de código realizas porque dentro de la filosofía de AngularJS y la del MVC en general, no puedes escribir en tu HTML código que sirva para implementar la lógica de tu aplicación. **(El código que necesitas para hacer las funcionalidades de tu aplicación no lo colocas en la vista, lo colocas en el controlador).**

## Trabajando con campos checkbox en AngularJS

Explicaciones y prácticas con campos input checkbox con AngularJS, conociendo las directivas ngChecked, ngTrueValue, ngFalseValue, ngChecked.

En el estilo de aplicaciones que se hacen con AngularJS trabajas de manera intensiva con campos de formulario. En Angular los campos input checkbox tienen una serie de directivas que podemos usar:

- **ngModel:** indica el nombre con el que se conocerá a este elemento en el modelo/scope.
- **ngTrueValue:** La utilizas si deseas asignar un valor personalizado al elemento cuando el campo checkbox está marcado.
- **ngFalseValue:** es lo mismo que ngTrueValue, pero en este caso con el valor asignado cuando el campo no está "chequeado".
- **ngChange:** sirve para indicar expresiones a realizar cuando se produce un evento de cambio en el elemento. Se dispara cuando cambia el estado del campo, marcado a no marcado y viceversa. Podemos ejecutar nuestra expresión o llamar a una función en nuestro scope.
- **ngChecked:** sirve para indicar una variable del scope donde se pueda deducir si el elemento debe estar o no marcado (checked).

<https://docs.angularjs.org/api/ng/directive/ngChecked>

### **Directiva ngClick / ngDbclick**

Esta directiva se utiliza para capturar cuando el usuario hace click o doble click sobre un elemento. Dentro del atributo ng-click o ng-dblclick se debe poner una expresión como las comentadas anteriormente. Aunque lo más lógico es llamar a un método del \$scope y que este realice la funcionalidad apropiada.

### **Directiva ngHref / ngSrc**

ngHref y ngSrc se utilizan para asignar el href y src a los elementos HTML que les corresponda. La ventaja de utilizar estas directivas en vez de href y src directamente es la capacidad de poder utilizarlas como ngBind.

### **Directiva ngClass / ngStyle**

Con ngClass y ngStyle la idea es aplicar un class o un style en función del cumplimiento de una expresión de angularjs.

### **Directiva ngVisible / ngHide**

Con ngVisible y su antagónico ngHide se pretende hacer visible o invisible un elemento en función de una expresión de angularjs.

### **Directiva ngEnabled / ngDisabled**

Al igual que los dos anteriores su funcionalidad es habilitar o deshabilitar un elemento en función de una condición.

### **Directiva ngModel**

Directiva que permite establecer un Two data Binding con la propiedad name de nuestro model.

Si quieres usar un checkbox lo más normal es que indiques la referencia de tu modelo donde quieres que se guarde su estado.

```
<input type="checkbox" ng-model="vm.activo" />
```

A partir de este momento el checkbox está asociado a tu scope en vm.activo. Pero un detalle, en el scope todavía no está creada esa propiedad hasta que no pulses encima del checkbox para activarlo o desactivarlo. En ese momento pasa a existir vm.activo en el modelo, aunque también si lo deseamos podemos inicializarla en un controlador.

```
vm.activo = true;
```

Como sabes, durante la vida de tu aplicación, el estado del checkbox se traslada automáticamente desde la vista al modelo y desde el modelo a la vista, por el proceso conocido por "doble binding".

En resumen, si en cualquier momento desde el Javascript cambias el valor de vm.activo, siempre se actualizará la vista. Por supuesto, si en la vista pulsas sobre el campo para activarlo o desactivarlo, en el modelo también quedará reflejado el nuevo estado.

## Directiva ngChange

Esta directiva sirve para especificar acciones cuando cambia el estado del checkbox. Pero atención, porque son solo cambios debidos a la interacción con el usuario. Es decir, si mediante Javascript cambiamos el modelo asociado a ese checkbox, cambiándolo de un estado a otro no se activará el evento ng-change. La vista seguirá alterando el estado del campo, gracias al mencionado binding, pero la expresión que hayas colocado en ng-change no se ejecutará.

```
<input type="checkbox" ng-change="vm.avisar()">
```

## Expresiones

---

Con las expresiones también enriquecemos el HTML, ya que nos permiten colocar cualquier cosa y que AngularJS se encargue de interpretarla y resolverla. Para crear una expresión simplemente la englobas dentro de dobles llaves, de inicio y fin.

Ahora dentro de tu aplicación, en cualquier lugar de tu código HTML delimitado por la etiqueta donde pusiste la directiva ng-app eres capaz de colocar expresiones. En Angular tenemos una gama de tipos de expresiones, por ejemplo hagamos una prueba colocando una expresión así.

```
<h1>{{ 1 + 1 }}</h1>
```

Angular cuando se pone a ejecutar la aplicación buscará expresiones de este estilo y lo que tengan dentro, si es algo que él pueda evaluar, lo resolverá y lo sustituirá con el resultado que corresponda.

Puedes probar otra expresión como esta:

```
{{ "Hola " + "Desarrollo en AngularJS" }}
```

Al ejecutarlo, AngularJS sustituirá esa expresión por "Hola Desarrollo en AngularJS ". Estas expresiones no me aportan gran cosa de momento, pero luego las utilizaremos para más tipos de operaciones.

Lo habitual de las expresiones es utilizarlas para colocar datos de tu modelo, escribiendo los nombres de las "variables" que tengas en el modelo. Por ejemplo, si tienes este código HTML, donde has definido un dato en tu modelo llamado "valor":

```
<input type="text" ng-model="valor" />
```

Podrás volcar en la página ese dato (lo que haya escrito en el campo INPUT) por medio de la siguiente expresión:

```
{{valor}}
```

Otro detalle interesante de las expresiones es la capacidad de formatear la salida, por ejemplo, diciendo que lo que se va a escribir es un número y que deben representarse dos decimales necesariamente. Vemos rápidamente la sintaxis de esta salida formateada, y más adelante la utilizaremos en diversos ejemplos:

```
{{ precio | number:2 }}
```

Cabe decir que estas expresiones no están pensadas para escribir código de tu programa, osea, lo que llamamos lógica de tu aplicación. Como decíamos, están sobre todo pensadas para volcar información que tengas en tu modelo y facilitar el "binding". Por ello, salvo el operador ternario (x ? y : z), no se pueden colocar expresiones de control como bucles o condicionales. Sin embargo, desde las expresiones también podemos llamar a funciones

codificadas en Javascript (que están escritas en los controladores, como veremos enseguida) para poder resolver cualquier tipo de necesidad más compleja.

## Usos y funcionamiento del scope de AngularJS

---

### Concepto de Scope

## View, Controllers & Scope



Para poder introducir los controladores debemos detenernos antes en un concepto que se repite mucho dentro de la literatura de AngularJS, el "scope". De hecho, tal como dice la documentación de AngularJS, el cometido de un controlador consiste en desempeñar una función constructora capaz de aumentar el Scope.

El "scope" es la pieza más importante del motor de AngularJS y es donde están los datos que se tienen que manejar dentro de la parte de presentación.

El scope es un gran contenedor de datos, que transporta y hace visible la información necesaria para implementar la aplicación, desde el controlador a la vista y desde la vista al controlador. En términos de código el scope no es más que un objeto al que puedes asignar propiedades nuevas, con los datos que necesites, o incluso con funciones (métodos).

Esos datos y esas funciones están visibles tanto en el Javascript de los controladores como en el HTML de las vistas, sin que tengamos que realizar ningún código adicional, pues Angular ya se encarga de ello automáticamente. Además, cuando surgen cambios en los datos se propagan entre los controladores y las vistas automáticamente. Esto se realiza por un mecanismo que llamamos "binding", y en AngularJS también "doble binding" (en español sería enlace).

Así pues, desde los controladores vamos a ser capaces de trabajar con el scope de una manera minuciosa, agregando o modificando información según lo requiera nuestra aplicación.

Nota: Hasta el momento en este Manual de AngularJS hemos usado el scope en varias ocasiones y ya hemos dicho que representa al modelo, del paradigma MVC. Siempre que hemos usado el scope en HTML ha sido 1) agregando partes de la página al modelo mediante la directiva ngModel, por ejemplo datos que se escriban en un campo de texto o se seleccionen en un campo select, o 2) volcando información en el HTML mediante expresiones escritas entre dos llaves {{ }}.

## Manejar el \$scope desde el controlador

El \$scope contiene la información necesaria que se desea mostrar en la página y el código para gestionar su flujo.

Para que un valor sea visible desde la capa de presentación solo es necesario añadir al \$scope una propiedad con el valor que se desee pasar. Luego desde la capa de presentación ya enlazará esta propiedad al elemento correspondiente del DOM utilizando directivas o `{{...}}`. Si el enlace que se crea es de tipo two-way-binding (usando ng-model), cuando se modifique el valor en los controles del DOM se actualizará automáticamente el valor del \$scope.

En el caso que se pretenda añadir un comportamiento se actuará de la misma forma, añadiendo en este caso una función al \$scope. Luego se enlazará esta función a los eventos que se lancen desde el DOM (utilizando directivas como ng-click) de forma que cuando se invoquen los eventos se ejecutará el código que contenga la función añadida al \$scope.

Esta forma de trabajar usando bindings es fundamental tenerla clara, así como la forma de enlazar propiedades, objetos y funciones desde el DOM para poder invocar y controlar el flujo de la capa de presentación.

El \$scope es un objeto que se refiere al ámbito/contexto de ejecución de las variables y permite conectar la vista con el controlador y tener las variables *bindadas*.

Reestructuramos ahora el mismo ejemplo del listado de libros añadiendo el concepto del Controlador. Para ello y a partir de la versión AngularJS 1.3 se requiere crear un módulo de la aplicación con lo que se requiere identificar a la aplicación `app="miApp"`.

Por tanto definimos el *controlador* en un contenedor `<div>` y le damos un nombre `<div ng-controller="miControlador">`

Una vez identificada la app y añadido el controlador los definimos en javascript mediante la definición de un módulo y la creación del controlador. En el constructor del controlador crearemos el listado de libros.

```
<script>
  var miApp = angular.module('miApp',[]);

  var controllers = {};
  controllers.miControlador = function($scope) {
    $scope.libros =
    [
      {titulo:'El Juego de Ender', autor:'Orson Scott Card'},
      {titulo:'Juego de Tronos', autor:'George R.R. Martin'},
      {titulo:'I robot', autor:'Isaac Asimov'}
    ];
  };

  miApp.controller(controllers);
</script>
```



## Asociar Controlador a un determinado html

La forma nativa de asociar un controlador al HTML es utilizar la directiva ng-controller:

```
1 <!DOCTYPE>
2 <html>
3   <head></head>
4   <body ng-app="ejemplo1">
5     <div ng-controller="listController">
6       <input type="text" ng-model="item"></input>
7       <span>El nombre introducido es: <strong>{{item}}</strong></span>
8       <button ng-click="add();"></button>
9       <ul>
10        <li ng-repeat="itemList in list">{{itemList}}</li>
11      </ul>
12    </div>
13    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.21/angular.js"></script>
14    <script>
15      angular
16        .module("ejemplo1", [])
17        .controller("listController", function($scope) {
18          $scope.list = [];
19          $scope.add = function() {
20            $scope.list.push($scope.item);
21          }
22        });
23    </script>
24  </body>
25 </html>
```

En este ejemplo se puede ver como dentro del controller listController se inicializa la propiedad list que luego será recorrida en el ng-repeat para listar los elementos. Al mismo tiempo se añade la función add que luego será invocada en el ng-click del botón para añadir un elemento a la colección (el item que se inicializa en el ng-model="item").

Otra forma que es menos verbose y que parece más limpia es utilizando el “controller as xxx”.

```
1 <!DOCTYPE>
2 <html>
3   <head></head>
4   <body ng-app="ejemplo">
5     <div ng-controller="listController as info">
6       <input type="text" ng-model="info.item"></input>
7       <span>El nombre introducido es: <strong>{{info.item}}</strong></span>
8       <button ng-click="info.add();"></button>
9       <ul>
10        <li ng-repeat="item in info.list">{{item}}</li>
11      </ul>
12    </div>
13    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.21/angular.js"></script>
14    <script>
15      angular
16        .module("ejemplo", [])
17        .controller("listController", function() {
18          this.list = [];
19          this.add = function() {
20            this.list.push(this.item);
21          }
22        });
23    </script>
24  </body>
25 </html>
```

Esta forma que elimina la palabra “scope” a cambio del this hace menos verbose el código y más legible, ya que el this de la función es directamente lo que hace bind contra el DOM.

Otra forma de asociar un controller al DOM es el uso de una directiva:

```

1 <!DOCTYPE>
2 <html>
3   <head></head>
4   <body ng-app="ejemplo1">
5     <div mg-list>
6       <input type="text" ng-model="item"></input>
7       <span>El nombre introducido es: <strong>{{item}}</strong></span>
8       <button ng-click="add();"></button>
9     </div>
10    <li ng-repeat="itemList in list">{{itemList}}</li>
11  </ul>
12 </div>
13 <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.21/angular.js"></script>
14 <script>
15   angular
16     .module("ejemplo1", [])
17     .directive("mgList", function() {
18       return {
19         restrict: 'A',
20         controller: function($scope) {
21           $scope.list = [];
22           $scope.add = function() {
23             $scope.list.push($scope.item);
24           }
25         }
26       };
27     })
28 </script>
29 </body>
30 </html>

```

## Anidación de scopes

La anidación de scopes es una de las cosas más potentes que tiene Angular y que permite tener un modelo compartido entre diferentes controladores y después cosas específicas dentro de cada uno de ellos.

Imagina que tienes que hacer una página donde definir el perfil de un usuario de tu aplicación. La puedes diseñar con carpetas y hacer una sola lectura para traer toda la información de tu usuario y en cada carpeta tener un botón save específico para guardar exclusivamente el contenido de esa carpeta.

```

1 function userController(scope){
2
3   //list of genders
4   scope.genders=[{id:1,name:'man'},{id:2,name:'women'}];
5   //model
6   scope.model={gender:1};
7 }
8 userController.$inject=['$scope'];
9
10 angular.module('myApp').controller('userController',userController);
11
12
13
14 function userPersonalDataController(scope){
15
16   scope.save=function(){
17     scope.text= 'name: ' + scope.model.name + ' genderId: ' + scope.model.gender;
18   };
19 }
20 userPersonalDataController.$inject=['$scope'];
21
22 angular.module('myApp').controller('userPersonalDataController',userPersonalDataController);
23
24
25
26
27
28 function userPassWordDataController(scope){
29
30   scope.save=function(){
31     scope.text= 'password: ' + scope.model.newPassWord + ' oldpassword: ' + scope.model.oldPassWord;
32   };
33 }
34 userPassWordDataController.$inject=['$scope'];
35
36 angular.module('myApp').controller('userPersonalDataController',userPersonalDataController);
37

```

Si analizamos el código tenemos tres controladores **UserController** responsable de recuperar la información del usuario, **UserPersonalDataController** responsable de guardar los datos personales de nuestro usuario y por último **UserPassWordDataController** responsable de guardar los datos de password de nuestro user. Gracias a la potencia de la anidación de scopes podemos desde cualquiera de los scopes hijos tener acceso a los datos del scope padre.

```
1 <!DOCTYPE html>
2 <html lang="es" ng-app='myApp'>
3   <head></head>
4   <body>
5     <div ng-controller='UserController'>
6       <div ng-controller='userPersonalDataController'>
7         <input type="text" ng-model="model.name"/>
8         <select ng-model="model.gender" ng-options="gender.id as gender.name for gender in genders"></select>
9         <button ng-click="save()">Save</button>
10        <br>
11        <span>{{text}}</span>
12      </div>
13      <div ng-controller='userPassWordDataController'>
14        <input type="password" ng-model="model.oldPassWord"/>
15        <input type="password" ng-model="model.newPassWord"/>
16        <button ng-click="save()">Save</button>
17        <br>
18        <span>{{text}}</span>
19      </div>
20    </div>
21    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.23/angular.js"></script>
22    <script src="app.js"></script>
23    <script src="controller.js"></script>
24  </body>
25 </html>
```

Observa en la vista como **UserController** anida a los controladores **userPersonalDataController** y **userPassWordDataController**.

## Principales métodos y propiedades de scope

---

**\$id.** Nos devuelve un identificador único para cada scope.

**\$parent.** Devuelve el scope padre o null si es el scope principal o \$rootScope.

**\$new.** Crea un nuevo scope.

**\$apply.** Aplica cualquier cambio realizado en el modelo, llama a \$digest. No se recomienda su uso en un controlador, se utiliza normalmente en una directiva, para avisar a Angular que refresque la vista, por ejemplo ante una respuesta de un evento del DOM.

**\$digest.** Es llamada desde \$apply.

**\$\$phase.** Puede tener un valor null o el valor establecido por \$digest y controla que está en un proceso de refresco de vista.

**\$eval.** Evalúa una expresión.

**\$evalAsync.** Evalúa una expresión al final del ciclo de refresco de la vista.

**\$destroy.** Establece todos los valores del scope a null y lanza un evento al que te puedes suscribir con scope.on('destroy',function(){...}).

**\$watch.** Observa una propiedad o expresión, evita su uso por temas de performance.

**\$watchGroup.** Es muy parecida a \$watch pero permite observar un array de expresiones.

**\$watchCollection.** Observa los cambios en una colección.

Estas últimas propiedades son eliminadas en Angular 2.0 y serán sustituidas por Object.observe y Array.observe.



AngularJS se puede apoyar en los principales frameworks CSS del mercado. El famoso **Bootstrap** puede ser utilizado sin problemas (de hecho es recomendado) en las interfaces de AngularJS. De hecho existe una librería que adapta a la perfección Bootstrap mediante directivas necesarias: **AngularUI**

Bootstrap es una plantilla responsiva, esto es que se adapta (escala) automáticamente al dispositivo en que se esté visualizando la página, desde teléfonos, a tabletas y PCs. Así hace que el desarrollo web front-end más rápido y más fácil.

Es un framework de HTML5 y CSS3, Está programado enteramente en HTML, CSS y Javascript.

<https://angular-ui.github.io/>  
<http://getbootstrap.com/>

## Tema 4: Vistas y filtros

### Definición y navegación de rutas

---

La navegación a través de páginas web se realiza mediante el acceso a las rutas en donde se encuentran almacenadas. En el caso de Angular, cada una de estas rutas podrá estar asociada tanto a un controlador como a una vista (por ejemplo, una plantilla HTML almacenada en un archivo externo), tal y como se verá más adelante en este documento. Dentro de cada uno de los controladores, estos tienen definido un objeto `$scope` con el que se van a comunicar con sus vistas asociadas, lo que quiere decir que la variable `$scope` será el *modelo* de datos de la aplicación MVC.

Puesto que una aplicación Angular puede crecer bastante y ello puede complicar la gestión de sus correspondientes dependencias, Angular implementa un sistema de módulos de tal forma que siempre que se define un componente se tiene que hacer dentro de un módulo. Este concepto es análogo al de los namespaces de XML, además de permitir el establecimiento de dependencias entre ellos para saber qué componentes utilizar en cada momento.

La agrupación de todos los módulos es la que va a definir la *aplicación*, que es el elemento que va a cargar todo el código generado usando Angular dentro de la página web. Para ello, dentro del archivo `index.html` se usará la directiva `ng-app` para definir el nombre de la aplicación Angular que se desea cargar, para posteriormente, mediante la directiva `ng-view`, especificar el lugar de la página en donde se van a cargar las vistas gestionadas por cada uno de los correspondientes módulos.

**ngView** es la directiva que complementa el servicio de `$route` donde se renderizan las vistas que se cargan al hacer el cambio de navegación. Cada vez que una ruta es modificada, la vista cambia y se carga en `ng-view` según la configuración del routing. Requiere también del módulo **ngRoute**.

```
1 | <body ng-app="MyApp">
2 |   <div ng-view>
3 |     <!--
4 |       Aquí se irán cargando las vistas asociadas a las rutas
5 |       de la aplicación MyApp definida en el elemento body
6 |       con la directiva ng-app
7 |     -->
8 |   </div>
9 | </body>
```

## Vistas y sistema de Routing en AngularJS

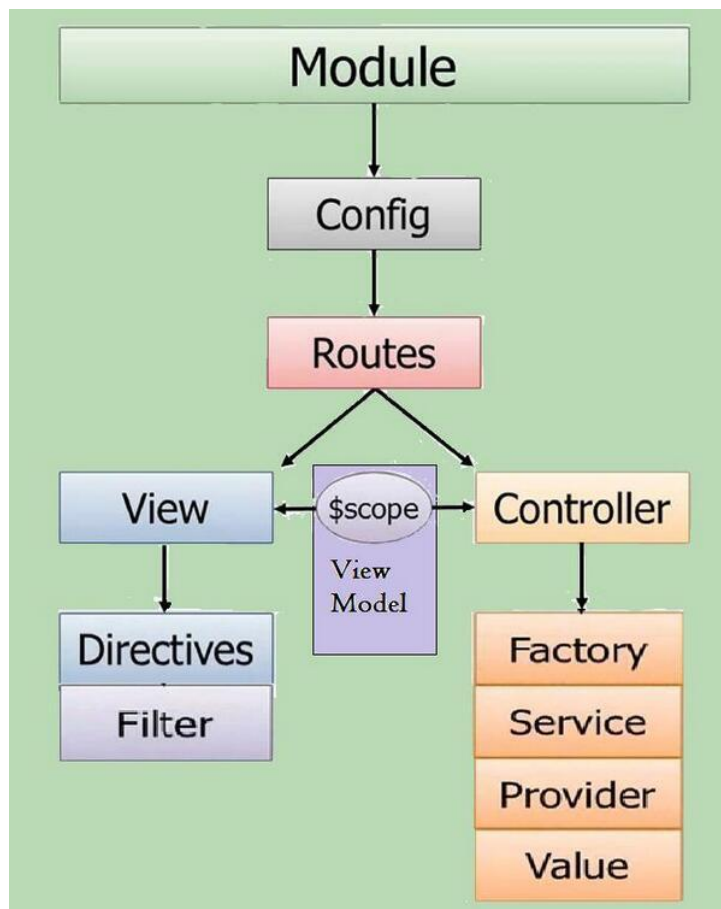
Las vistas nos permiten separar nuestra interfaz en partes distintas y, con el sistema de routing AngularJS, podemos saber qué vistas debe mostrarte atendiendo a la URL que la aplicación esté mostrando.

Además, veremos en detalle cómo trabajar con múltiples vistas y rutas profundas en una aplicación desarrollada con este framework Javascript.

AngularJS posee un potente sistema de **routing** que nos va a permitir indicar las rutas de las diferentes urls/pantallas. El **routing** nos permite definir la navegación de nuestra aplicación SPA.

Para poder configurar las rutas de nuestra aplicación se requiere del módulo **ngRoute** que se incluye en la librería **angular-route.js** (su versión minimizada: **angular-route.min.js**) ya que AngularJS no controla las rutas por defecto.

## Módulo ngRoute para crear rutas e intercambiar vistas en AngularJS



### Instalar ngRoute

El módulo ("module") ngRoute es un potente paquete de utilidades para configurar el enrutado y asociar cada ruta a una vista y un controlador. Sin embargo este módulo no está incluido en la distribución de base de Angular, sino que en caso de pretender usarlo tenemos que instalarlo y luego inyectarlo como dependencia en el módulo principal de nuestra aplicación.

Como no lo tenemos en el script básico de Angular, debemos instalarlo "a mano" lo conseguimos incluyendo el script del código Javascript del módulo ngRoute.

Muchas veces usarás el CDN correspondiente y deberás asegurarte de estar usando en el módulo ngRoute de la misma versión de AngularJS que cargaste inicialmente. Otro detalle es que este script lo tienes que incluir después de haber incluido el script del core de AngularJS.

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular.min.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular-route.min.js"></script>
<script src="app.js"></script>
```

## Inyección de dependencias

El segundo paso sería inyectar la dependencia con `ngRoute` en el módulo general de nuestra aplicación. Esto lo hacemos en la llamada al método `module()` con el que iniciamos cualquier programa AngularJS, indicando el nombre de las dependencias a inyectar en un array.

```
angular.module("app", ["ngRoute"])
```

Hasta ahora este array de dependencias, cuando llamábamos a `angular.module()`, estaba siempre vacío. A medida que se compliquen nuestras aplicaciones podremos necesitar inyectar más cosas.

Por ejemplo, las directivas creadas por terceros desarrolladores o las que haya desarrollado uno mismo, también se inyectarán de esta manera en tu módulo principal de la aplicación.

## Configurar el sistema de enrutado con `$routeProvider`

El sistema de enrutado de AngularJS nos permite configurar las rutas que queremos crear en nuestra aplicación de una manera declarativa. Aunque sea un componente bastante complejo internamente, podemos configurarlo de una manera ciertamente sencilla.

`$routeProvider` tiene un par de métodos. El primero es `when()`, que nos sirve para indicar qué se debe hacer en cada ruta que se desee configurar, y el método `otherwise()` que nos sirve para marcar un comportamiento cuando se intente acceder a cualquier otra ruta no declarada.

Esta configuración se debe realizar dentro del método `config()`, que pertenece al modelo. De hecho, solo podemos inyectar `$routeProvider` en el método `config()` de configuración.

```
angular.module("app", ["ngRoute"])

.config(function($routeProvider){

//configuración y definición de las rutas

});
```

### Definición de rutas

Podemos ver toda la configuración en [https://docs.angularjs.org/api/ngRoute/provider/\\$routeProvider](https://docs.angularjs.org/api/ngRoute/provider/$routeProvider)

En AngularJS, las rutas se definen en la fase de configuración de la aplicación, haciendo uso del servicio **`$routeProvider`**. Éste proporciona una API sencilla donde podemos encadenar métodos para definir rutas (método `when`), y establecer una ruta por defecto (`otherwise`). El método `when` recibe como entrada dos parámetros:

Para usar las rutas en AngularJS lo primero que necesitaremos será configurarlo para poder asociar a la ruta la página HTML que se debe cargar al navegar a esa ruta. Para ello tenemos el servicio [\\$routeProvider](#) de AngularJS.

Este servicio tiene 2 métodos:

- `when` : Asocia una ruta con una página HTML
- `otherwise`: Indica a qué ruta debemos redirigirnos cuando hay una ruta desconocida

El método `when` de `$routeProvider` permite asociar una ruta a una página HTML aunque como mínimo también se indica el controlador que se usará en esa página.



El método `when` acepta dos parámetros:

- **path:** Es el nombre de la ruta, es decir, lo que se pondrá tras el símbolo "#".
- **route:** Es un objeto que define la página HTML a cargar y el controlador que se usará en esa página HTML. El objeto debe tener las siguientes propiedades:
  - **templateUrl:** Nombre de la página HTML. Debe ser una ruta deferida a la página HTML principal que contendrá esta página HTML.
  - **controller:** Es el nombre del controlador de AngularJS que se usará con esta página. Al definir aquí el controlador, ya nos ahorramos el tener que emplear la directiva `ng-controller` en nuestro código.

Para que funcione la navegación definimos las rutas mediante `$routeProvider`.

**\$routeProvider** define las rutas mediante sus dos métodos:

- **when (path, route):** Cuando encuentra una url entonces.... Donde **path** es la url y **route** es el objeto ruta que tiene ciertas propiedades, en nuestro ejemplo cargamos la vista indicada en el parámetro `templateUrl`.
- **otherwise (params):** En caso de no encontrar ninguna de las rutas anteriores entonces aplica esta regla. Normalmente se redirige a otra página mediante **redirectTo**. El método `otherwise` suele recibir un parámetro, consistente en un objeto con un atributo `redirectTo`, indicando la URL a la que redirigir cuando no se encuentra ninguna ruta concordante.

Esta configuración del routing lo hacemos en el módulo de la aplicación principal *myApp*. Como se puede ver requiere la dependencia del módulo *ngRoute* y hay que importarlo en el módulo. Después mediante el método **config** definimos estas rutas gracias a que ya podemos acceder a **\$routeProvider**.

Supongamos una aplicación de dos páginas. Consiste en una aplicación de pedidos, donde en la primera página tenemos un listado de pedidos, y en la segunda un formulario para realizar nuevos pedidos. La página por defecto de nuestra aplicación será el listado de pedidos.

- Un índice, que contiene un listado de todos los pedidos realizados.
- Un formulario de introducción de nuevos pedidos.

## app.js

```
var myApp = angular.module("myApp",['ngRoute'])
.config(
  function($routeProvider)
  {
    $routeProvider
    .when('/orders',
    {
      templateUrl:'orders/list.html',
      controller:'ordersController',
    })
    .when('/orders/new',
    {
      templateUrl:'new-order/order.html',
      controller:'newOrderController',
    })
    .otherwise( {redirectTo: '/orders'});
  }
);
```

Si nos vamos a una de las vistas, por ejemplo orders/list.html vemos que no se ha introducido la directiva ng-controller al haber definido el controlador en la definición de rutas.

```
<h2>Order list</h2>
<div class="row" ng-repeat="order in orders">
<div class="col col-xs-12">
<p>{{ order }}</p>
</div>
</div>
<div class="row">
<div class="col col-xs-12">
<a href="#/orders/new" class="btn btn-default">New order</a>
</div>
```

Las rutas las configuras por medio del método when() que recibe dos parámetros. Por un lado la ruta que se está configurando y por otro lado un objeto que tendrá los valores asociados a esa ruta(templateUrl+controller).

```
var eventsApp = angular.module('eventsApp', ['ngResource'])
.config(function($routeProvider) {
  $routeProvider.when('/newEvent',
    {
      templateUrl: 'templates/NewEvent.html',
      controller: 'EditEventController'
    })
});
```

- "controller", para indicar el controlador.
- "controllerAs", el nombre con el que se conocerá el scope dentro de esa plantilla.
- "templateUrl", el nombre del archivo, o ruta, donde se encuentra el HTML de la vista que se debe cargar cuando se acceda a la ruta.

Las rutas se leen en cascada una debajo de la otra. Si ninguna ruta coincide entonces se ejecuta el método **otherwise**. otherwise( {redirectTo: '/'});De manera que si se pone cualquier url en la barra de navegación la app hará un redirect hacia la ruta raíz (en este caso).

## Ejemplo routing

```
var uazonApp = angular.module("uazonApp",['ngRoute','libros'])
.config(
  function($routeProvider)
  {
    $routeProvider
    .when('/listado',
    {
      templateUrl:'src/views/listado.html',
      controller: 'LibrosCtrl'
    })
    .when('/nuevo-libro',
    {
      templateUrl:'src/views/nuevo-libro.html',
      controller: 'LibrosFichaCtrl'
    })
  })
```

```

    })
    .otherwise( {redirectTo: '/listado'});
  }
);

```

## Rutas parametrizadas con \$routeParams

**\$routeParams** nos permite recoger parámetros de la url.

Para ello en la ruta lo definimos mediante dos puntos y el nombre del parámetro :libroId

Hacer esto con el sistema de routing de AngularJS es bastante sencillo: podemos declarar elementos variables simplemente poniendo el símbolo de los dos puntos ( : ) delante de éste. Así, en nuestra aplicación de libros, introduciremos una ruta nueva para ver el detalle de un libro.

```

var uazonApp = angular.module("uazonApp",['ngRoute','libros'])
.config(
  function($routeProvider)
  {
    $routeProvider
    .when('/ficha-libro/:libroId',
    {
      templateUrl:'src/views/ficha-libro.html',
      controller: 'LibrosFichaCtrl'
    })
  }
);

```

Para recibir los parámetros en nuestro controlador, deberemos hacer uso del servicio **\$routeParams**. Éste nos permite recibir todos los parámetros que hayamos pasado a la URL.

En el controlador la forma de recuperar los parámetros \$routeParams es inyectando \$routeParams:

```

angular.controller("LibrosFichaCtrl", ['$scope', '$routeParams', function($scope, $routeParams) {
  $scope.libroId = $routeParams.libroId;
}]);

```

```

var app = angular.module("app", []);

app.config(function ($routeProvider) {
  $routeProvider
    .when('/map/:country/:state/:city',
      {
        templateUrl: "app.html",
        controller: "AppCtrl"
      })
});

app.controller("AppCtrl", function ($scope, $routeParams) {
  $scope.model = {
    message: "Address: " +
      $routeParams.country + ", " +
      $routeParams.state + ", " +
      $routeParams.city + ", "
  }
});

```



### URLs amigables sin # gracias a HTML5

Si queremos evitar el # podemos activar el modo HTML5 en las rutas evitando así el refresco de la página. Para este cometido tendremos que realizar **dos tareas**:

**1. Configurar \$locationProvider** accediendo a la variable \$locationProvider de ngRoute e indicando que está en html5

**\$locationProvider.html5Mode(true);**

```

var miApp = angular.module("miApp", ['ngRoute'])
.config(
  function($routeProvider, $locationProvider)
  {
    $routeProvider
      .when('/pagina1',
        {
          templateUrl: 'paginas/pagina1.html',
        })
      .when('/pagina2',
        {
          templateUrl: 'paginas/pagina2.html',
        })
      .otherwise( {redirectTo: '/pagina1'});

    //elimina # de la navegación
    $locationProvider.html5Mode(true);
  }
);

```

## 2. Definiendo la base de nuestro *index.html* <base href="/">

```
<html ng-app="miApp">
  <head>
    <base href="/">
  </head>
  <body>
    ...
  </body>
</html>
```

### Propiedad resolve en \$routeProvider

Podemos realizar una precarga de los datos llamando directamente al servicio desde la configuración del \$routeProvider.

```
myApp.config(['$routeProvider', '$locationProvider', function($routeProvider,
  $routeProvider.
  when('/', {
    templateUrl: '../partials/list.html',
    controller: 'ListController',
    resolve: {
      initialData: function(booksService) {
        return booksService.getBooks();
      }
    }
  })
  ]).
```

**resolve.** Es un mapa de dependencias opcional que se inyectan sobre el controlador. Si esas dependencias son promises el enrutador esperará a inyectarlas antes de instanciar el controlador. Si todo va bien se disparará el evento \$routeChangeSuccess y si no el \$routeChangeError. El objeto que resuelve es **key** nombre de la dependencia y **factory** nombre del servicio a ejecutar o la función directamente.

Posteriormente en el servicio basta con inyectar initialData para obtener el listado de libros inicial.

```
.controller('ListController', ['$scope', 'initialData',
function($scope, initialData) {
  $scope.books = initialData;
  $scope.bookOrder = 'name';
}])
```

También es posible realizar a la vez 2 peticiones con \$http a servicios diferentes y combinarlos para devolverlos en una promise con \$q.

```
return {  
  getBooks: function() {  
    console.log('path '+path)  
    var books1=$http.get(path);  
    var books2=$http.get(path);  
  
    return $q.all([books1,books2]).then(function(result) {  
      var books=[];  
      angular.forEach(result,function(response) {  
        books.push(response.data);  
      });  
  
      return books;  
    }).then(function(books) {  
      var finalbooks=books[0].concat(books[1]);  
      return finalbooks;  
    })  
  }  
}
```

---

## Filtros predefinidos

---

Un tipo especial de operador en las expresiones de angular y que merece una detallar son los filtros. Los filtros son un operador especial que a partir de unas entradas devuelve una salida. Principalmente se utiliza para mostrar información formateada o modificada pero también se pueden concatenar para que la salida de uno sea la entrada de otro.

En angular existen ya algunos **filtros predefinidos**:

- **currency**: formatea el número de entrada como moneda según el locale del browser aunque se le puede fijar el tipo de moneda.
- **number**: formatea un número, especialmente útil para mostrar determinado número de decimales.
- **date**: formatea fechas según se defina.
- **json**: devuelve a partir de un objeto javascript con su equivalente json como string.
- **lowercase**: pasa el texto a minúsculas.
- **uppercase**: pasa el texto a mayúsculas.
- **filter**: sirve para filtrar colecciones y solo devolver elementos que cumplan determinada condición.
- **limitTo**: devuelve los primeros o últimos x elementos de un array o string.
- **orderBy**: ordena un array.

También mediante los filtros podemos filtrar el listado de libros de una forma rápida y sencilla simplemente filtrando la colección de datos del ng-repeat mediante `ng-repeat="libro in libros | filter:nombre"` donde *nombre* sería una variable definida por un modelo (*ng-model*) en un input. Así que cuando el usuario rellene ese input *nombre* automáticamente se filtrará la colección de datos dejando en pantalla sólo los datos que concuerden con ese filtro.

```
<html ng-app>
  <body>
    <div ng-init="libros=['El Juego de Ender','Juego de Tronos','I robot']">
      <h1>Listado de libros</h1>
      Nombre: <input type="text" ng-model="nombre" />

      <ul>60
        <li ng-repeat="libro in libros | filter:nombre">{{ libro | uppercase }}</li>
      </ul>
    </div>
    <script src="angular.min.js"></script>
  </body>
</html>
```

AngularJS dispone de una gran cantidad de filtros predefinidos. En el bloque de código anterior, vemos cómo utilizamos, el filtro `uppercase` para transformar una cadena a mayúsculas una cadena, o el filtro `orderBy` nos permite ordenar una colección de elementos. También vemos que podemos encadenar tantos filtros como queramos. En la directiva `ng-repeat` usamos los filtros `filter` y `orderBy`.

Al encadenar filtros, el resultado se irá propagando al siguiente en el orden que los hemos declarado. En este caso, primero Hemos usado el filtro `filter` para filtrar elementos basándonos en el modelo `textFilter`. Luego, hemos ordenado el conjunto de resultados por la propiedad `name`.

## Uso de filter

```
<!doctype html>
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.min.js">
    </script>
    <script src="app.js"></script>
  </head>
  <body ng-app="app" ng-controller="MainController as main">
    <h2>Contact List</h2>
    <hr />
    <label>Search for: <input type="text" ng-model="searchfor" /></label>
    <div ng-repeat="person in main.people | filter:searchfor">
      <p>{{person.name}}<br />
        {{person.phone}}</p>
    </div>
  </body>
</html>
```

## Filtro uppercase

El filtro uppercase convierte una cadena a mayúsculas. En nuestro código HTML se usa de la siguiente forma: {{ uppercase\_expression | uppercase }}

```
<body ng-app="myModule" ng-controller="myController">
  <div>{{ nombre | uppercase }}</div>
  <input type="text" ng-model="nombre" />
  <button ng-click="change();" >click</button>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular.min.js"></script>
  <script>
    angular.module("myModule", [])
      .controller("myController", ['$scope', function($scope) {
        $scope.change=function() {
          $scope.nombre ="my_name"
        }
      }]);
  </script>
</body>
```

## Filtro OrderBy

ng

module

directive

a

form

input

input.checkbox

input.email

input.number

input.radio

input.text

input.url

ngApp

ngBind

ngBindHtml

ngBindTemplate

ngBlur

ngChange

ngChecked

ngClass

ngClassEven

orderBy

filter in module ng

Description

Orders a specified array by the expression predicate.

Usage

1. filter:orderBy(array, expression[, reverse]);

Parameters

Param	Type	Details
array	Array	The array to sort.
expression	function(*) string Array.<function(*)> string	A predicate to be used by the comparator to determine the order of elements. Can be one of: <ul style="list-style-type: none"><li>function: Getter function. The result of this function will be sorted using the &lt;, =, &gt; operator.</li><li>string: An Angular expression which evaluates to an object to order by, such as 'name'. Optionally prefixed with + or - to control ascending or descending sort order (for example, +name or -name).</li></ul>

http://docs.angularjs.org/api/ng.filter:orderBy



Otro filtro interesante aplicable es poder ordenar de forma automática mediante `orderBy`. Cambiamos la colección de datos por una estructura *clave/valor*, por ejemplo, cada libro con su nombre y autor: `ng-init="libros = [ {titulo:'El Juego de Ender', autor:'Orson Scott Card'}, {titulo:'Juego de Tronos', autor:'George R.R. Martin'}, {titulo:'I robot', autor:'Isaac Asimov'} ]` Podríamos después poder ordenar por cada uno de sus campos, en este caso por el nombre del autor `ng-repeat="libro in libros | filter:nombre | orderBy:'autor'"`

```
<html ng-app>
  <body>
    <div ng-init="libros =
      [
        {titulo:'El Juego de Ender', autor:'Orson Scott Card'},
        {titulo:'Juego de Tronos', autor:'George R.R. Martin'},
        {titulo:'I robot', autor:'Isaac Asimov'}
      ]">

      <h1>Listado de libros</h1>
      Nombre: <input type="text" ng-model="nombre" />

      <ul>
        <li ng-repeat="libro in libros | filter:nombre | orderBy:'autor'">{{ libro.titulo |
uppercase }} - {{ libro.autor}}</li>
      </ul>
    </div>
    <script src="angular.min.js"></script>
  </body>
</html>
```

<http://codepen.io/jmortega/pen/kXOzyz>

El filtro `orderBy` nos permite ordenar un array por una propiedad, en sentido normal o inverso. En nuestro código HTML se usa de la siguiente forma:

**{{ orderBy\_expression | orderBy : expression : reverse }}**

```
<script>
angular.module('orderByExample', [])
.controller('ExampleController', ['$scope', function($scope) {
$scope.friends =
[{name:'John', phone:'555-1212', age:10},
{name:'Mary', phone:'555-9876', age:19},
{name:'Mike', phone:'555-4321', age:21},
{name:'Adam', phone:'555-5678', age:35},
{name:'Julie', phone:'555-8765', age:29}];
$scope.predicate = '-age';
}]);
</script>
<div ng-app="orderByExample">
<div ng-controller="ExampleController">
<pre>Sorting predicate = {{predicate}}; reverse = {{reverse}}</pre>
<hr/>
[ <a href="" ng-click="predicate="">unsorted</a> ]
<table class="friend">
<tr>
<th><a href="" ng-click="predicate = 'name'; reverse=false">Name</a>
(<a href="" ng-click="predicate = '-name'; reverse=false">^</a>
a)</th>
```

```

<th><a href="" ng-click="predicate = 'phone'; reverse=!
reverse">Phone Number</a></th>
<th><a href="" ng-click="predicate = 'age'; reverse=!reverse">Age</
a></th>
</tr>
<tr ng-repeat="friend in friends | orderBy:predicate:reverse">
<td>{{friend.name}}</td>
<td>{{friend.phone}}</td>
<td>{{friend.age}}</td>
</tr>
</table>
</div>
</div>

```

La documentación de AngularJs dice que expresión puede ser una función que podemos gestionar, por ejemplo, en nuestro controlador.

```

angular.module('orderByExample', [])
.controller('ExampleController', ['$scope', function($scope) {
  $scope.friends =
    [{name:'John', phone:'555-1212', age:10},
    {name:'Mary', phone:'555-9876', age:19},
    {name:'Mike', phone:'555-4321', age:21},
    {name:'Adam', phone:'555-5678', age:35},
    {name:'Julie', phone:'555-8765', age:29}];
  var predicate = 'age';
  var reverse = false;
  $scope.predicate = predicate;
  $scope.reverse = reverse;
  $scope.setPredicate = function(_predicate){
    if(predicate === _predicate) {
      reverse = !reverse;
    } else {
      predicate = _predicate;
      reverse = false;
    }
    $scope.predicate = predicate;
    $scope.reverse = reverse;
  };
  $scope.getPredicate = function(){
    return (reverse?'-':'') + predicate;
  };
}]);

```

## Filtro date

El filtro date nos permite formatear una fecha de la manera deseada.

En una plantilla HTML lo usaremos de la forma: `{{ date_expression | date : format }}`

<http://codepen.io/jmortega/pen/ENYNep>

```
<!DOCTYPE html>
<html ng-app="myApp">
  <head>
    <script src='http://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular.min.js'></script>
    <script>
      var app = angular.module("myApp", []);

      app.controller("clockController", function ($scope) {
        $scope.currentTime = new Date();
      });
    </script>
    <title>Welcome to AngularJS</title>
  </head>
  <body>
    <h1>Hello, World.</h1>
    <p ng-controller='clockController'>
      The current time is {{currentTime | date:'h:mm:ss a'}}.
    <p>
  </body>
</html>
```

## Filtro json

El filtro json recibe un objeto como entrada, y devuelve una cadena representando dicho objeto en formato JSON. En nuestro código HTML se usa de la siguiente forma:

**`{{ json_expression | json : spacing }}`**

En nuestro código javascript lo usaremos de la siguiente forma:

`$filter('json')(object, spacing)`

El parámetro spacing es opcional, e indica el número de espacios que se utilizará en la indentación (el valor por defecto es 2). En el este enlace podemos ver el siguiente ejemplo funcionando:

```
<div ng-app>
  <h3>Default spacing</h3>
  <pre id="default-spacing">{{ {name:'AngularJS', title:'Frameworks JavaScript: AngularJS'} | json }}</pre>
  <h3>Custom spacing</h3>
  <pre id="custom-spacing">{{ {name: 'AngularJS', title: 'Frameworks de JavaScript: AngularJS'} | json:4 }}</pre>
</div>
```

## Filtro limitTo

<http://codepen.io/jmortega/pen/JKgZgg>

El filtro limitTo recibe como entrada un array, del que tomará un número de elementos igual al parámetro recibido ( limit ). Estos elementos se tomarán del principio si el número **es positivo**, y del final si es negativo. En nuestro código HTML se usa de la siguiente forma:

```
{{ limitTo_expression | limitTo : limit }}
```

```
<script>
angular.module('limitToExample', [])
.controller('ExampleController', ['$scope', function($scope) {
$scope.numbers = [1,2,3,4,5,6,7,8,9];
$scope.letters = "abcdefghi";
$scope.longNumber = 2345432342;
$scope.numLimit = 3;
$scope.letterLimit = 3;
$scope.longNumberLimit = 3;
}]);
</script>
<div ng-app="limitToExample">
<div ng-controller="ExampleController">
Limit {{numbers}} to: <input type="number" step="1" ngmodel="
numLimit">
<p>Output numbers: {{ numbers | limitTo:numLimit }}</p>
Limit {{letters}} to: <input type="number" step="1" ngmodel="
letterLimit">
<p>Output letters: {{ letters | limitTo:letterLimit }}</p>
Limit {{longNumber}} to: <input type="number" step="1" ngmodel="
longNumberLimit">
<p>Output long number: {{ longNumber | limitTo:longNumberLimit }}</p>
</div>
</div>
```

---

## Creación de filtros personalizados

Para crear un filtro, hay que registrar una nueva función factoría de tipo filter en nuestro módulo. Esta factoría debe devolver una función, que recibe como parámetro el elemento de entrada. Se pueden pasar los parámetros adicionales que haga falta.

### Filter.js

Creamos una función que recoge el ítem y lo pase a mayúscula

```
(function(myApp) {  
    myApp.filter('customUpperCase', function() {  
        return function(item) {  
            return item.toUpperCase();  
        }  
    });  
})(angular.module("myApp"));
```

Esto mismo también se puede realizar en app.js

```
var myApp = angular.module('myApp', [  
    'ngRoute',  
    'booksController'  
]);  
  
myApp.filter('customLowerCase', function() {  
    return function(item) {  
        return item.toLowerCase();  
    }  
});
```

Para usar estos filtros lo podemos hacer dentro de un listado y se aplicaría a cada ítem de la lista.

```
<li class="book cf" ng-repeat="item in books | filter: query | orderBy:  
    <a href="/details/{{books.indexOf(item)}}">  
          
        <div class="info">  
            <h2>{{item.name | customUpperCase }}</h2>  
            <h3>{{item.description | customLowerCase}}</h3>  
        </div>  
    </a>
```

En el siguiente ejemplo, construiremos un filtro que se encargue de formatear un número de teléfono:

<http://codepen.io/jmortega/pen/kXOzAd>

```
angular
.module('telephoneSample', [])
.filter('telephone', function(){
  return function(input) {
    var number = input || "";
    number = number.trim().replace(/[-\s\(\)]/g, "");
    if(number.length === 11) {
      var area = ['(', '+', number.substr(0,2), ')'].join("");
      var local = [number.substr(2, 3), number.substr(5, 3),
        number.substr(8, 11)].join('-');
      return [area,local].join(' ')
    }
    if(number.length === 9) {
      var local = [number.substr(0, 3), number.substr(3, 3),
        number.substr(6, 11)].join('-');
      return local
    }
    return number;
  };
});
```

Para probarlo en nuestra vista:

```
<div ng-app="telephoneSample">
<div>{{'965123' | telephone}}</div>
<div>{{'965123456' | telephone}}</div>
<div>{{'34965123456' | telephone}}</div>
</div>
```

## Filtrado en ngRepeat

<http://codepen.io/jmortega/pen/WxVLLa>

El filtrado nos sirve para hacer una búsqueda dentro de los elementos de la colección que tenemos en un array o en un objeto en nuestro modelo.

Por supuesto, la colección está del lado del cliente completa y lo que hace el filter es simplemente definir cuáles elementos de esa colección desean visualizarse en la repetición. O sea, es un filtrado de datos de los que ya disponemos, no un buscador que pueda buscar entre miles de elementos y solo recibir unos pocos en el cliente.

Cuando estás filtrando ya tienes esos elementos en la memoria de Javascript en un array u objeto.

Se indica con la palabra filter, seguida por ":" y la cadena o variable donde está la cadena que nos sirve para filtrar.

**ng-repeat="person in main.people | filter:searchfor"**

Esto nos mostrará solo aquellas personas que tienen la palabra escrita en el campo de búsqueda.

## Reverse Filter

<http://codepen.io/jmortega/pen/oYvBLg>

```
var app = angular.module("MyApp", []);

app.filter("reverse", function() {
  return function(input) {
    var result = "";
    input = input || "";
    for (var i=0; i<input.length; i++) {
      result = input.charAt(i) + result;
    }
    return result;
  };
});
```

## Tema 5: Formularios y directivas

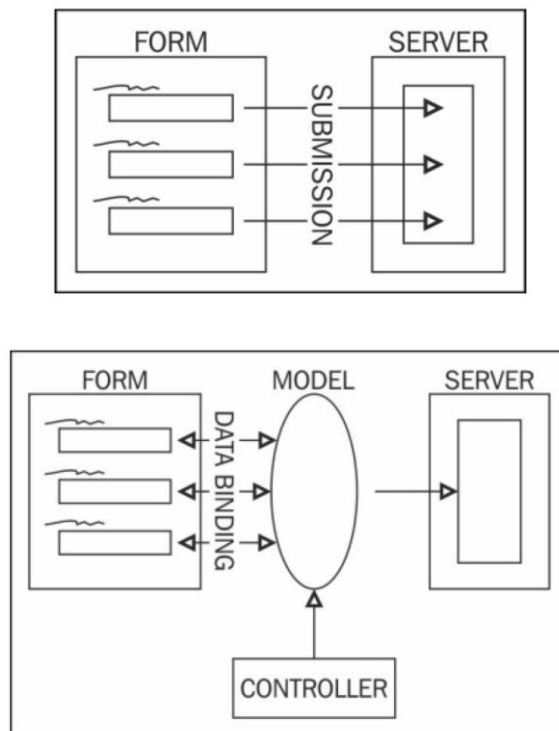
### Creación de formularios y validación de datos

---

Vamos a ver cómo funcionan los formularios en AngularJS, y cómo éste modifica y extiende el comportamiento de los inputs de HTML, y cómo gestiona las actualizaciones del modelo.

También veremos las directivas de validación incluidas en el *core* de AngularJS, para finalmente crear nuestras propias directivas de validación.

En un formulario HTML estándar, el valor de un input es el valor que se enviará al servidor al ejecutarse la **acción submit** del formulario.



Esto se consigue a través de las directivas `form` e `input`, así como con las directivas de validación y los controladores. Estas directivas de validación sobrescriben el comportamiento por defecto de los formularios HTML. Sin embargo, mirando su código, vemos que son prácticamente iguales que los formularios HTML estándar.



## Validaciones

La validación de formularios en el lado del cliente es una de las características de AngularJS. Hay muchas directivas de validación en AngularJS, en esta documentación trataremos las más populares

```
1 <form name="form">
2   <label name="email">Your email</label>
3   <input type="email" name="email" ng-model="email" placeholder="Email Address" />
4 </form>
```

El anterior ejemplo, construye un formulario en HTML5 en el que introducimos un único campo, el correo electrónico, decimos que es HTML5 porque el **type** es email y esto solo está reflejado en la última versión del standard. También podemos observar cómo le asignamos un modelo a través de la directiva **ng-model**, más adelante nos servirá para poder acceder al valor del campo desde un **Controller** (Controlador).

AngularJS hace fácil manejar las validaciones desde el lado del cliente sin añadir un esfuerzo extra. Aunque no podemos depender del lado cliente para mantener nuestra aplicación segura, AngularJS nos proporcionará una respuesta instantánea del estado del formulario. Para usar la validación de formularios, primero debemos asegurarnos de que el formulario tiene un **name** asociado con ello, como en el ejemplo anterior.

Todos los campos de entrada pueden validarse contra validaciones básicas, como por ejemplo, una longitud mínima, una longitud máxima, etc. Todas estas validaciones están disponibles entre los atributos de los formularios HTML5.

Normalmente, es una buena idea usar el flag **novalidate** en el formulario. Esto evitará que el navegador envíe el formulario.

A continuación, vamos a echar un vistazo a todas las validaciones que podemos poner sobre un campo **input**:

### Obligatorio

Para validar que un campo del formulario es obligatorio, simplemente añadir la etiqueta HTML5 **required** al campo de entrada:

```
1 <input type="text" required />
```

También podemos usar la directiva **ngRequired** para especificar aquellos campos obligatorios. Así, todos aquellos campos cuyo valor sea null, undefined o una cadena vacía serán inválidos. Por ejemplo, el campo nombre es uno de los campos obligatorios:

```
<input type="text" ng-model="user.name" required />
```

### Longitud mínima

Para validar que un campo del formulario tiene al menos un {número} de caracteres, simplemente añadir la directiva AngularJS **ng-minlength={number}** al campo de entrada:

```
1 <input type="text" ng-minlength=5 />
```

## Longitud máxima

Para validar que un campo del formulario es igual o menor que un número de caracteres, simplemente añadir la directiva AngularJS `ng-maxlength={number}` al campo de entrada:

```
1 <input type="text" ng-maxlength=20 />
```

## Coincidencia con una expresión regular

Para asegurarnos que un campo de entrada coincide con una expresión regular, simplemente añadir la directiva AngularJS `ng-pattern="/Expresion-regular/"` al campo de entrada:

```
1 <input type="text" ng-pattern="/^[a-zA-Z0-9]+$/" />
```

*Para que la expresión regular funcione deberá estar dentro de `"/^[expresión-regular]+$/"` donde '+' indica que podemos introducir cero o más caracteres y el '\$' es el símbolo de terminación de la cadena.*

## Correo electrónico

La validación de emails es muy sencilla. Simplemente debemos cambiar el `input type="text"` por `input type="email"`. AngularJS ya se encargará realizar las validaciones necesarias para este tipo. Para validar una dirección de correo electrónico, simplemente tendremos que poner el tipo del campo de entrada a `email`, de esta forma:

```
1 <input type="email" name="email" ng-model="user.email" />
```

## Número

Para validar que un campo del formulario solo contenga números, simplemente poner en el tipo del formulario el valor `number`, de esta forma:

```
1 <input type="number" name="email" ng-model="user.age" />
```

## Dirección web(url)

Al igual que el `type="email"`, también disponemos de un `type="url"` para que la validación de URLs sea lo más sencilla posible. Para validar que un campo del formulario representa una dirección web, simplemente poner en el tipo del formulario el valor `url`, de esta forma:

```
1 <input type="url" name="homepage" ng-model="user.facebook_url" />
```

Debemos recordar que los tipos de campo de entrada como `url`, `number` o `email` son un standard de HTML5 adaptados en AngularJS, pero debemos pensar en su otra utilidad, que es la usada en dispositivos como por ejemplo los móviles, ya que cuando pulsamos en cualquiera de estos campos para escribir un texto, dependiendo del tipo de campo el teclado será distinto, por tanto, si el tipo es `number`, en el teclado sólo saldrán números, puntos o comas. Al igual que si es de tipo `URL`, que nos saldrán letras y barra (/) y si es `email` saldría un (@).

## Variables de control en el formulario

AngularJS tiene propiedades disponibles sobre el ámbito `$scope` como resultado de las validaciones del formulario, estas nos permiten reaccionar en tiempo real. Hay que tener en cuenta que estas propiedades estarán disponibles en el siguiente formato:

```
1 formName.inputFieldName.property
```

A continuación, pasamos a explicar cada campo:

- **formName:** es el nombre del formulario.
- **inputFieldName:** es el nombre del campo de entrada.
- **property:** es el nombre de la propiedad.

Los distintos tipos de propiedades serán:

### Formulario sin modificar

Es una propiedad booleana que indica si el usuario ha modificado el formulario. Esta será `true` si el usuario no ha tocado el formulario, y `false` si lo ha hecho:

```
1 formName.inputFieldName.$pristine;
```

### Formulario modificado

Es una propiedad booleana que indica si y solo si, si el usuario ha modificado el formulario. Esta será establecida con independencia del resto de validaciones del formulario:

```
1 formName.inputFieldName.$dirty
```

### Formulario válido

Es una propiedad booleana que indica si el formulario es válido o no ha modificado el formulario. Esta será `true` si el formulario es válido:

```
1 formName.inputFieldName.$valid
```

### Formulario no válido

Es una propiedad booleana que indica si el formulario es no válido. Esta será `true` si el formulario no es válido porque no cumpla alguna validación:

```
1 formName.inputFieldName.$invalid
```

## Errores

Otra propiedad útil de AngularJS es la del objeto `$error`. Este objeto contiene todas las validaciones sobre un formulario en particular y si ellas son válidas o no, si alguna validación falla entonces esta propiedad se establece a `true`, mientras que si es `false`, significará que el campo de entrada ha pasado las validaciones. Para tener acceso a esta propiedad, use la siguiente sintaxis:

```
1 formName.inputfieldName.$error
```

## Estilo de las propiedades

Cuando AngularJS maneja un formulario, este añade clases específicas al formulario basándose en su estado. Estas clases son nombradas de manera similar a las propiedades que está comprobando. Estas clases serán:

```
1 .ng-pristine {}
2 .ng-dirty {}
3 .ng-valid {}
4 .ng-invalid {}
```

Cada uno corresponde a su homologado sobre el campo de entrada particular. Por ejemplo, cuando un campo es no válido, la clase `.ng-invalid` será aplicada al campo, de modo que podemos crearnos un estilo CSS propio para los elementos que sean 'no válidos':

```
1 input.ng-invalid {
2   border: 1px solid red;
3 }
4 input.ng-valid {
5   border: 1px solid green;
6 }
```

## Validación programática

Al tener los objetos **ngModelController** y **ngFormController** en el scope, podemos trabajar con el estado del formulario de manera programática.

En nuestro controlador podríamos utilizar el objeto `ngModelCtrl` para validar los valores de los campos de un formulario.

```
$scope.getCssClasses = function(ngModelCtrl){
  return {
    invalidelement: ngModelCtrl.$invalid && ngModelCtrl.$dirty,
    validelement: ngModelCtrl.$valid && ngModelCtrl.$dirty
  };
};
```

Para mostrar los errores de validación, haremos uso de la directiva `ngShow`

```
$scope.showError = function(ngModelCtrl, error) {
  return ngModelCtrl.$error[error];
};
```

```
<div>
<label>
nombre: <input type="text" ng-model="user.name" required ngminlength="
3" ng-maxlength="25" name="userName" ngclass="
getCssClasses(userForm.userName)" />
</label>
<span ng-show="showError(userForm.userName, 'required')">
Campo obligatorio
</span>
```

```
<span ng-show="showError(userForm.userName, 'minlength')">
Longitud mínima: 3
</span>
<span ng-show="showError(userForm.userName, 'maxlength')">
Longitud máxima: 25
</span>
</div>
```

<http://codepen.io/jmortega/pen/bwbwZA>

Tenemos un ejemplo funcionando con todas las validaciones y comprobaciones del formulario. Además, en él también hemos introducido un botón para enviar el formulario. Sin embargo no nos interesará enviarlo a menos que estén todos los campos correctamente introducidos.

Es por ello que haremos uso de la directiva `ngDisabled` para deshabilitar el botón si el formulario no es válido.

```
<button type="submit" ng-disabled="userForm.$invalid">Registrar</button>
```

```
angular
.module('formExample', [])
.controller('mainCtrl', function($scope, $log){
    $scope.user = {};

    $scope.provinces = [
        { name : 'Castellón', code : '12'},
        { name : 'Valencia', code : '46'},
        { name : 'Alicante', code : '03'}
    ];

    $scope.getCssClasses = function(ngModelCtrl){
        return {
            invalidelement: ngModelCtrl.$invalid && ngModelCtrl.$dirty,
            validelement: ngModelCtrl.$valid && ngModelCtrl.$dirty;
        };
    };

    $scope.showError = function(ngModelCtrl, error) {
        return ngModelCtrl.$error[error];
    };

    $scope.$watch('user', function(){
        $log.info('form', $scope.userForm);
    });

    $scope.submitAction = function(){
        alert('Registrado!!');
        $log.info('Submitted data:', $scope.user);
    };
});
```

## Ejecución

nombre:

apellidos:  Deben ser dos palabras como mínimo

email:  Campo obligatorio

website:  Campo obligatorio

provincia:

suscribirse a la newsletter ☐

Registrar

```
{
  "name": "name"
}
```

nombre:

apellidos:

email:  Email incorrecto

website:  Campo obligatorio

provincia:

suscribirse a la newsletter ☐

Registrar

```
{
  "name": "name",
  "lastName": "surname surname2"
}
```

nombre:

apellidos:

email:

website:  Formato de URL incorrecto

provincia:

suscribirse a la newsletter ☐

Registrar

```
{
  "name": "name",
  "lastName": "surname surname2",
  "email": "user@domain.com"
}
```

nombre:

apellidos:

email:

website:

provincia:

suscribirse a la newsletter ☒

Registrar

```
{
  "name": "name",
  "lastName": "surname surname2",
  "email": "user@domain.com",
  "website": "http://www.domain.com",
  "province": "03",
  "newsletter": true
}
```

Submitted data: 

```
Object {name: "name", LastName: "surname surname2", email: "user@domain.com", website: "http://www.domain.com", province: "03"...}
  email: "user@domain.com"
  lastName: "surname surname2"
  name: "name"
  newsletter: true
  province: "03"
  website: "http://www.domain.com"
  __proto__: Object
```

## Form validation in event mock

### Add event

<b>Name</b> <input type="text" value="event"/>	<b>Description</b> <input type="text" value="eventDescription"/> Please enter and event description
<b>Date</b> <input type="text" value="eventDate"/>	<b>Location</b> <input type="text" value="eventLocation"/>

```
<input type="text" required ng-model='eventCtl.eventForm.description' class="form-control" name="eventDescription" placeholder="Event Description">
<p class="help-block" ng-show='addEventForm.$dirty && addEventForm.eventDescription.$invalid'>Please enter an Event Description</p>
```

## Creación de formularios y validación de datos

```
<div ng-controller='userController'>

  <div ng-controller='userPersonalDataController' ng-form="personaldata" ng-submit="save()">
    <input name="name" type="text" ng-model="model.name" required/>
    <span ng-hide="personaldata.name.$valid">El campo es requerido</span>
    <select name="gender" ng-model="model.gender" ng-options="gender.id as gender.name for gender in genders"></select>
    <button type="submit">Save</button>
    <br>
    <span>{{text}}</span>
  </div>

  <div ng-controller='userPassWordDataController'>
    <form name='password' ng-submit="save()">
      <input name="oldPassWord" type="password" ng-model="model.oldPassWord" ng-required="true"/>
      <span ng-hide="password.oldPassWord.$valid">El campo es requerido</span>
      <input name="newPassWord" type="password" ng-model="model.newPassWord">
      <button type="submit">Save</button>
      <br>
      <span>{{text}}</span>
    </form>
  </div>
</div>
```

Se recomienda utilizar la etiqueta form y no la ng-form

### Validaciones

Una de las cosas más importantes en cualquier aplicación es validar las entradas de usuario, tanto en el cliente y sobre todo en el servidor, en este punto, nos vamos a centrar en la validación del lado del cliente para evitar roundtrips innecesarios, asumiendo que tienes claro que hay que validar sí o sí en el servidor para evitar problemas de seguridad entre otros.

Angularjs define una serie de directivas por defecto que te ayudan con la validación en el lado del cliente.

- required o ng-required
- min
- max
- ng-minlength
- ng-maxlength
- ng-pattern.

Podemos pensar que con estas directivas es imposible cubrir todos los casos posibles en una app. Para ello disponemos del módulo **ui-validation** que se puede obtener desde github.

<https://github.com/angular-ui/ui-validate>

<https://htmlpreview.github.io/?https://github.com/angular-ui/ui-validate/master/demo/index.html>

El código fuente de este módulo lo que hace es hacer un **eval** de la expresión y si es correcto establece **\$setValidity** a una key y su valor a true, en caso contrario a false. Este método pertenece al objeto **ModelController** y en el caso que una entrada no sea correcta en nuestro modelo se establece la propiedad a undefined. La función de validación se agrega a **\$formatters** y **\$parsers** del mismo objeto.

Para más información <https://docs.angularjs.org/guide/forms>

---

## Directivas y expresiones comunes de AngularJS

---

### Ventajas usar directivas

Estos son dos de los motivos fundamentales por lo que es útil crear nuestras propias directivas:

- **Legibilidad:** Las directivas permiten escribir HTML expresivo. En cuanto a index.html se puede entender el comportamiento de la aplicación con solo leer el código HTML.
- **Reusabilidad:** Las directivas permiten crear unidades autónomas de funcionalidad. Podríamos fácilmente conectar en esta directiva a otra aplicación angularjs y evitar escribir un montón de HTML repetitivo.

### Creación de directivas personalizadas

---

Hemos visto algunas directivas en AngularJS pero nosotros podemos crear nuestras propias directivas, para ello deberemos usar el método directive del módulo. Para crear una directiva debemos definirla dentro de un módulo al igual que hacíamos para los controladores, filtros, etc.

El método a usar del módulo es **directive**(nombre,funcionFactory) siendo:

- nombre: El nombre de la directiva siguiendo el forma de los identificadores de JavaScript.
- funcionFactory: Es una función que nos retorna un [objeto con la definición de la directiva](#). Como todas las funciones factory puede ser un array para poder inyectarle dependencias.

El objeto con la definición de la directiva es el verdadero objeto que contiene toda la información de nuestra directiva y comprender todas sus propiedades implicaría saber todo sobre las directivas así que en este curso sólo vamos a ver unas cuantas propiedades del [objeto con la definición de la directiva](#).

Las directivas son los elementos que nos permiten **extender el DOM**, generando componentes con el comportamiento que nosotros queramos. Aunque AngularJS trae un conjunto de directivas muy potente en su core, en alguna ocasión querremos **crear elementos con cierta funcionalidad propia y reusable**. En este capítulo veremos cómo podemos hacerlo a través de la creación de nuevas directivas.

Podemos definir nuevas directivas gracias al método directive() , que nos proporciona un module de AngularJS. La definición es similar a como ya hemos visto para controladores, filtros o servicios. El nombre de la directiva debe definirse en camelCase, y la función debe devolver un objeto, conocido como **Directive Definition Object (DDO)**.

Este objeto tiene las siguientes propiedades:

DefinicionDirectiva
String template
String templateUrl
String restrict
boolean replace
boolean Object scope
Function link



- **template** : Es un string con el HTML por el que se substituirá la directiva. Define el contenido de la directiva. Puede incluir código HTML, *data binding expressions* y otras directivas.
- **templateUrl** : Es un string con una URL de un fichero que contiene el HTML por el que se substituirá la directiva.
- **scope** : Lo utilizamos para crear un scope hijo (scope : true) o un isolate scope (scope : {}). Angular nos permite crear un scope hijo, o lo que se conoce como un isolate scope. Este segundo está completamente separado del scope padre en el DOM.
- **replace** : Si vale false el contenido del template se añadirá dentro del tag de la propia directiva. Pero si vale true se quitará el tag de la directiva y solo estará el contenido del template.
- **restrict**: Un string que indica cómo puede usarse la directiva si como atributo o como elemento:
  - Si vale "E" solo se podrá usar como elemento
  - Si vale "A" solo podrá usarse como atributo
  - Si vale "C" solo podrá usarse como clase
  - Si vale "EA" o "AE" se podrá usar como elemento y como atributo. Este es el funcionamiento por defecto si no se pone nada en la propiedad restrict.

Una directiva no tiene por qué ser de un tipo únicamente. Podemos definir varios tipos en el atributo restrict . En el siguiente ejemplo, **nuestra directiva será capaz de funcionar como atributo, elemento o clase.**

```
angular
.module('directives', [])
.directive('loginButton', function(){
return {
restrict : 'EAC',
template : '<a class="btn btn-primary btn-lg" ng-href="#/login">
<span class="glyphicon glyphicon-log-in"></span> Acceder</a>'
}
});
```

```
<div ng-app="directives">
  <div>
    <h3>Etiqueta</h3>
    <login-button></login-button>
  </div>
  <div>
    <h3>Atributo</h3>
    <div login-button></div>
  </div>
  <div>
    <h3>Clase</h3>
    <div class="login-button"></div>
  </div>
</div>
```

## Función Link

### Función que se ejecutará para inicializar la directiva

El uso principal de la función link es para asociar **listeners** a elementos del DOM, observar cambios en propiedades del modelo, y validación de elementos.

La función link acepta 4 parámetros que son:

- scope: Es el scope de nuestra directiva
- iElement: Es un elemento DOM del tag raíz del template. Está encapsulado como si fuera un elemento de jQuery llamado jqLite

- iAttrs : Son todos los atributos que se han puesto en la directiva
- controller
- transcludeFn

### Función compile

Utilizaremos la función compile para realizar transformaciones en el DOM antes de que se ejecute la función link . Esta función recibe dos elementos:

- element : elemento sobre el que se aplicará la directiva
- attrs : listado de atributos de la directiva.

La función compile no tiene acceso al scope , y debe devolver una función link. El esqueleto de una directiva cuando utilizamos una función compile es:

```
angular
.module('compileSkel', [])
.directive('sample', function(){
return {
compile : function(element, attrs) {
//realizar transformaciones sobre el DOM
return function(scope, element, attrs){
//función link normal y corriente
};
}
};
});
```

En <http://codepen.io/jmortega/pen/yaBxLp> tenemos un ejemplo que usa la función compile para establecer un estilo por defecto a una serie de elementos div.

```
angular
.module('ejemploCompile', [])
.directive('setColor', function(){
return {
restrict: 'A',
compile: function(element, attrs) {
element.css('background-color', attrs.setColor);
element.css('border', attrs.borderWidth + 'px solid ' +
attrs.borderColor);

return function(scope, element, attrs) {

}
}
}
});
```

```
<div ng-app="ejemploCompile">
  <div set-color="#ff0000" border-width="1" border-color="#dedede">Rojo</div>
  <div set-color="#00ff00" border-width="0" border-color="#124356">Verde</div>
  <div set-color="#0000ff" border-width="3" border-color="#11AA66">Azul</div>
</div>
```

## Directivas de validación

Para el ejemplo, introduciremos una directiva que valide DNIs. Como todos sabremos, la letra del DNI es un dígito de control que se obtiene al aplicar una fórmula matemática sobre el número. Nuestra directiva validará que la longitud del DNI y el dígito de control sean correctos. Tendrá una forma similar a:

```
<input type="text" ng-model="user.dni" validate-dni />
```

Como podemos ver en la función `validarNif`, el validador tiene una doble responsabilidad:

- Indicar al `ngModelController` si la validación del nif es correcta o no.
- Devolver el nuevo valor del modelo.

Podemos ver un ejemplo de nuestra directiva funcionando en

<http://codepen.io/jmortega/pen/wzwEWJ>

DNI/NIF:  DNI/NIF válido

DNI/NIF:  DNI/NIF inválido

```
<div ng-app="validationSample" ng-controller="mainCtrl">
  <form ng-submit="submitAction()" name="nifForm">
    <div>
      <label>DNI/NIF: <input type="text" ng-model="user.nif"
name="nif" validate-nif ng-class="getCssClasses(nifForm.nif)"/>
</label>

      <span ng-show="showError(nifForm.nif, 'validateNif')">
        DNI/NIF inválido
      </span>

      <span ng-show="!showError(nifForm.nif, 'validateNif')">
        DNI/NIF válido
      </span>

    </div>

    <div>
      <input type="submit" value="Submit" />
    </div>
  </form>
</div>
```

```

angular
  .module('validator.nif', [])
  .directive('validateNif', function () {
    return {
      restrict: 'A',
      require: 'ngModel',
      link: function(scope, element, attrs, ngModelCtrl) {
        // Acepta NIEs (Extranjeros con X, Y o Z al principio)
        var validarNif = function (value) {
          var numero, letraNif, letra;
          var expresion_regular_dni = /^[XYZ]?[0-9]{5,8}[A-Z]$/;
          var result;

          value = value.toUpperCase();

          if (expresion_regular_dni.test(value) === true) {
            numero = value.substr(0, value.length - 1);
            numero = numero.replace('X', 0);
            numero = numero.replace('Y', 1);
            numero = numero.replace('Z', 2);
            letraNif = value.substr(value.length - 1, 1);
            numero = numero % 23;
            letra = 'TRWAGMYFPDXBNJZSQVHLCKET';
            letra = letra.substring(numero, numero + 1);

            if (letra !== letraNif) {
              result = false;
            } else {
              result = true;
            }
          } else {
            result = false;
          }
        };
      }
    };
  });

```

## Ejercicio

<http://codepen.io/jmortega/pen/WGZgaJ>

Vamos a crear una directiva, llamada **maxDecimals**, que usaremos en el formulario de edición del capítulo anterior. El dato será inválido si el número de decimales del input es mayor que el valor que se asigne a max-decimals.

```
<input type="number" required max-decimals="2" />
```

```
angular
.module('validator.decimals', [])
.directive('maxDecimals', function () {
  return {
    restrict: 'A',
    require: 'ngModel',
    link: function(scope, element, attrs, ngModelCtrl) {
      var validarNumber = function (value) {

        return value;
      };

      ngModelCtrl.$parsers.unshift(validarNumber);
      ngModelCtrl.$formatters.push(validarNumber);
    }
  }
});
```

## Ejemplo de directiva que muestra el componente fecha

Usamos la directiva y especificamos que el datepicker use el formato de "dd/mm/yy"

```
<input id="date" name="date" type="text" ng-model="book.date" required  
datepicker="dd/mm/yy" />
```

```
app.directive('datepicker', [function(dateFormat) {  
    return {  
        restrict: 'A',  
        link: function($scope, element, attributes) {  
            element.datepicker({  
                dateFormat: attributes.datepicker,  
                onSelect: function() {  
                    $(this).trigger('change');  
                }  
            });  
        }  
    };  
}]);
```

- Decimos que solo se puede usar como Atributo HTML con la opción restrict:'A'
- En la función de link llamamos a la función de JQuery UI para que transforme el <input> en un Datepicker. El método datepicker() acepta como argumento un objeto de configuración.
- Al objeto de configuración del método datepicker() le pasamos la propiedad dateFormat con el formato de fecha, sacando dicho formato del valor del atributo de la propia directiva: attributes.datepicker
- Añadimos la función de callback del evento onSelect
- Lanzamos un evento de onChange en el <input> para que AngularJS sepa que algo ha cambiado y actualice consecuentemente la propiedad \$scope.book.date.

## Directivas para realizar validaciones personalizadas

AngularJS hace fácil añadir nuestras propias validaciones también a través de directivas personalizadas. En la directiva podemos indicar que queremos hacer con el valor del campo, por ejemplo, comprobar si en el campo de entrada hemos escrito la palabra **Angular** en cuyo caso nos dejaría continuar.

Para ver cómo funciona, seguiremos el sencillo ejemplo propuesto anteriormente, para ello echamos un vistazo al código de la directiva:

```
.directive('ensureUnique', [ function() {
  return {
    require: 'ngModel',
    restrict: 'A',
    link: function(scope, ele, attrs, c) {
      scope.$watch(attrs.ngModel, function() {
        if(attrs.value==='Angular') {
          c.$setValidity('unique', true);
        }
        else {
          c.$setValidity('unique', false);
        }
      });
    }
  };
}]);
```

A continuación, explicamos cada uno de los atributos de la directiva:

- El atributo **require** nos indica que el elemento debe contener la directiva **ng-model**, la cual podemos hacer uso de ella para crear nuestro propio ámbito.
- El atributo **restrict** nos indica a qué tipo de elementos podemos restringir dicha directiva, los posibles valores son:
  - **'E'**: Elemento HTML, por ejemplo, `<ensure-unique></ensure-unique>`
  - **'A'**: Atributo HTML, por ejemplo como en nuestro caso, sería `<input ensure-unique=""></input>`
  - **'C'**: Clase de HTML, por ejemplo `<div class="ensure-unique"></div>`
- En la función **link** tenemos cuatro parámetros:
  - **Scope**: Es el ámbito pasado a la directiva, en este caso es el mismo que tiene el padre.
  - **Ele**: Contiene el elemento que hay en el interior de la directiva.
  - **Attrs**: Se refiere a atributos que se puedan inyectar a la directiva.
  - **C**: Maneja desde el valor de un campo de entrada hasta los estados de validación del mismo.

La función **link** es principalmente usada para adjuntar los eventos que escuchan a los elementos del DOM, observando cambios en las propiedades del **model** y actualizando el DOM.

Dentro de esta función hemos insertado un elemento **Listener(\$watch)** para obtener los cambios pertinentes al modelo, este será activado cada vez que se modifique el modelo, es decir, cada vez que introduzcamos un carácter en el campo de entrada. Podemos observar cómo hacemos uso del argumento **attrs**, que como hemos dicho antes, accede a los atributos de nuestro elemento, en este caso el atributo **value** que veremos más adelante.

Gracias al método `setValidity` podemos enganchar con el sistema común de validaciones y en caso que el valor sea el correcto, actualizamos la variable `unique` a `true`.

Para que esto funcione, la directiva en el HTML se formará de la siguiente manera:

```
<form name="userForm" ng-submit="submitForm()" novalidate>

  <div class="form-group" ng-submit="submitForm()">
    <label>Escribe <strong>Angular</strong> y podras continuar</label>

    <input ensure-unique=""
           type="text" name="username" class="form-control"
           value="{{username}}" ng-model="username">
    </input>

    <!-- Error message -->
    <p ng-show="userForm.username.$invalid && !userForm.username.$pristine"
       class="help-block">
      Deberías escribir 'Angular' en lugar de '{{username}}'
    </p>
  </div>

  <button type="submit" ng-disabled="userForm.$invalid" class="btn btn-primary">Comprobar</button>

</form>
```

Es un código HTML normal, lo único que cambia es la cuarta línea, donde hemos añadido una nueva directiva llamada `ensure-unique` como un atributo, también tenemos `value` que indicará el valor que vayamos introduciendo en el campo de entrada y que en la directiva nos está sirviendo de validación. Más abajo tenemos un párrafo que muestra un mensaje de error en caso de que el valor del campo no sea válido( a través de `$invalid`) y que ya hayamos escrito algo en el formulario(usando `$pristine`).

También hacemos uso de `$invalid` en el caso del botón de submit para que, hasta que el formulario no sea válido completamente, inhabilitar el botón.



## Tema 6: Comunicación con el servidor

### Servicios HTTP con AngularJS

---

Los servicios son un concepto importante en Angular. Del mismo modo que se pueden definir controladores, se pueden definir servicios en los que se agrupan funcionalidades que luego estarán disponibles para ser usadas en los controladores, mejorando la claridad del código y favoreciendo la reutilización. En Angular los servicios también se pueden utilizar para mantener estados y compartir información entre los controladores.

Un servicio es un objeto JavaScript que nos permite obtener información. Aparentemente nada nuevo que entender, sería por ejemplo un DAO en Java o un servicios de Java. Lo importante de esto es que un servicio nunca interacciona con la propia página, sólo con otros servicios o con un servidor de datos que pueda estar en otro Host.

Ejemplos de servicios serían:

- El servicio [\\$http](#) de AngularJS. Este servicio hace la típica llamada AJAX a un servidor para obtener información de él. Como vemos, cumple perfectamente la definición de obtener información.
- Un servicio que se conecta a un host que nos retorna el valor del Euribor.
- Un posible servicio de cálculo de Hipoteca que dados los datos de una hipoteca (Importe del préstamo, años, diferencial ,etc) nos calculara cuánto hay que pagar mensualmente.
- Un servicio que nos hiciera las operaciones de [CRUD](#) sobre el servidor.
- Un servicio que transformara los String con una fecha en un objeto Date.
- El servicio [\\$log](#) de Angular que nos permite generar un log de nuestra aplicación. Este servicio simplemente llama a console.log pero nos abstrae por si no existe el objeto console. \$log dispone de varios métodos pero por ahora nos quedaremos con el método “debug” que permite pasarle un mensaje para que se muestre por la consola del navegador.

Los servicios definidos en Angular tienen la finalidad de encapsular funcionalidades de tal forma que, por un lado, se abstraiga al desarrollador de los detalles relacionados con ciertas operaciones concretas, mientras que por otro fomentan la reutilización del código fuente y mejoran su mantenibilidad. Un ejemplo de servicio se encuentra en el módulo de *home*, donde se usa uno para hacer una llamada a un servidor esperando una respuesta.

En este ejemplo se está creando un servicio simple que requiere unos datos de un destino. Como se puede ver, la manera de crear un servicio es similar a la de crear un controlador:

```
.service('nombreServicio', function(dependencias) {  
  
    // Código del servicio  
  
});
```

El código del servicio puede ser enviar una petición POST a un servidor unos parámetros obtenidos del almacén de datos de sesión del navegador, del cual esperamos una respuesta, para lo que se usa el método `$http`, devolviendo una promesa de éxito (success) o de fracaso (error).

## Ajax con AngularJS para acceso a API

Nos introducimos en el uso de Ajax mediante AngularJS. Lo vamos a hacer de una manera muy habitual en el desarrollo con este framework que es accediendo a un servicio web que nos ofrece un API REST.

Ajax es una solicitud HTTP realizada de manera asíncrona con Javascript, para obtener datos de un servidor y mostrarlos en el cliente sin tener que recargar la página entera.

### Servicio \$http

El servicio \$http es una funcionalidad que forma parte del núcleo de Angular. Sirve para realizar comunicaciones con servidores, por medio de HTTP, a través de Ajax y vía el objeto XMLHttpRequest nativo de Javascript o vía JSONP.

Después de esa denominación formal, que encontramos en la documentación de AngularJS, te debes de quedar por ahora en que nos sirve para realizar solicitudes y para ello el servicio \$http tiene varios tipos de acciones posibles. Todos los puedes invocar a través de los parámetros de la función \$http y además existen varios métodos alternativos (atajos o shortcuts) que sirven para hacer cosas más específicas.

Tanto el propio \$http() como las funciones de atajos te devuelven un objeto que con el "patrón promise" te permite definir una serie de funciones a ejecutar cuando ocurran cosas, por ejemplo, que la solicitud HTTP se haya resuelto con éxito o con fracaso.

### Inyección de dependencias con \$http

Si vas a querer usar el servicio \$http lo primero que necesitarás será inyectarlo a tu controlador, o a donde quiera que lo necesites usar. Esto es parecido a lo que mostramos cuando estábamos practicando con controladores e inyectábamos el \$scope.

```
angular
.module('app', [])
.controller('appCtrl', ['$scope', '$http', controladorPrincipal]);
```

Como puedes ver, en la función de nuestro controlador, llamada controladorPrincipal, le estamos indicando que recibirá un parámetro donde tiene que inyectar el service \$http.

Al declarar la función recibirás esa dependencia como parámetro.

```
function controladorPrincipal($scope,$http){
}
```

Ahora, dentro del código de ese controlador podrás acceder al servicio \$http para realizar tus llamadas a Ajax.

### Realizar una llamada a \$http.get()

El método get() sirve para hacer una solicitud tipo GET. Recibe diversos parámetros, uno obligatorio, que es la URL y otro opcional, que es la configuración de tu solicitud.

```
$http.get("http://www.example.com")
```

Lo interesante es lo que nos devuelve este método, que es un objeto "HttpPromise", sobre el cual podemos operar para especificar el comportamiento de nuestra aplicación ante diversas situaciones.

## Respuesta en caso de éxito

De momento, veamos qué deberíamos hacer para especificarle a Angular lo que debe hacer cuando se reciba respuesta correcta del servidor.

```
$http.get(url)
.success(function(respuesta){
//código en caso de éxito
});
```

Como puedes ver en este código es que \$http nos devuelve un objeto. Sobre ese objeto invocamos el método success() que sirve para indicarle la función que tenemos que ejecutar en caso de éxito en la solicitud Ajax. Esa función a su vez recibe un parámetro que es la respuesta que nos ha devuelto el servidor.

## Ejemplo de solicitud Ajax con \$http desde un controlador

```
var app=angular.module("app",[]);

app.controller("MyController", ['$scope', '$log', '$http', function($scope, $log, $http) {

    $http({
        method: 'GET',
        url: 'datos.json'
    }).success(function(data, status, headers, config) {
        $scope.datos=data;
    }).error(function(data, status, headers, config) {
        alert("Ha fallado la petición. Estado HTTP:"+status);
    });

}]);
```

## Ejemplo de solicitud Ajax con \$http

### Ver código en países.html

Vistos estos nuevos conocimientos sobre el "service" \$http estamos en condiciones de hacer un poco de Ajax para conectarnos con un API REST que nos ofrezca unos datos. Esos datos son los que utilizaremos en nuestra pequeña aplicación para mostrar información.

Vamos a usar un API llamada REST Countries, que puedes encontrar en <http://restcountries.eu/>

Esta API nos devuelve JSON con datos sobre los países del mundo. En principio puedes hacer diversas operaciones con los comandos sobre el API, pero vamos a ver solamente unas pocas que nos permiten recibir los datos de los países agrupados por regiones.

Las URL que usaremos para conectarnos al API son como estas:

<http://restcountries.eu/rest/v1/region/africa>

<http://restcountries.eu/rest/v1/region/europe>

Puedes abrirlas en tu navegador y observar cómo la respuesta es un conjunto de datos en notación JSON.

```

<div ng-app="apiApp" ng-controller="apiAppCtrl">
<h1>Pruebo Ajax</h1>
<p>
Selecciona:
<select ng-model="url" ng-change="buscaEnRegion()">
<option value="http://restcountries.eu/rest/v1/region/africa">Africa</option>
<option value="http://restcountries.eu/rest/v1/region/europe">Europa</option>
<option
value="http://restcountries.eu/rest/v1/region/americas">America</option>
</select>
</p>
<ul>
<li ng-repeat="pais in paises">
País: <span>{{pais.name}}</span>, capital: {{pais.capital}}
</li>
</ul>
</div>

```

Tenemos un campo SELECT que nos permite seleccionar una región y para cada uno de los OPTION tenemos como value la URL del API REST que usaríamos para obtener los países de esa región.

En el campo SELECT está colocada la directiva ngChange, que se activa cuando cambia el valor seleccionado en el combo. En ese caso se hace una llamada a un método llamado buscaEnRegion() que veremos luego escrito en nuestro controlador.

También encontrarás una lista UL en la que tienes una serie de elementos LI. Esos elementos LI tienen la directiva ngRepeat para iterar sobre un conjunto de países, de modo que tengas un elemento de lista por cada país.

Ahora puedes fijarte en el Javascript:

```

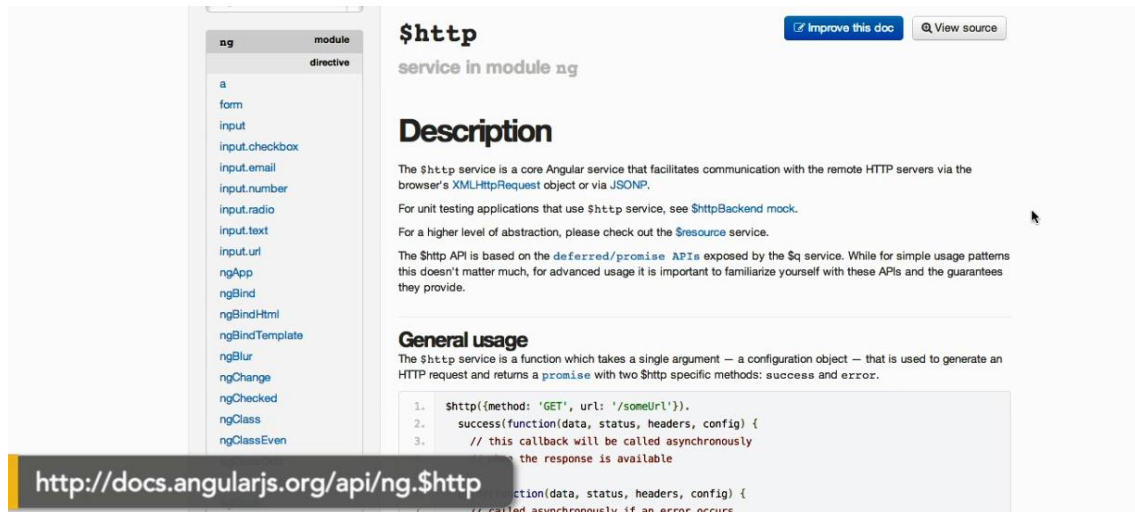
angular
.module('apiApp', [])
.controller('apiAppCtrl', ['$scope', '$http', controladorPrincipal]);
function controladorPrincipal($scope, $http){
    buscaEnRegion = function(){
        $http.get(url).success(function(respuesta){ //llamada Ajax
        //console.log("res:", respuesta);
        paises = respuesta;
        });
    }
}

```

Creamos un módulo y luego un controlador al que inyectamos el \$http.

Luego, en la función que construye el controlador, tenemos un método que se llama buscaEnRegion() que es el que se invoca al modificar el valor del SELECT. Ese método es el que contiene la llamada Ajax.

## Comunicación con el servidor con \$http



The screenshot displays the AngularJS documentation for the `$http` service. On the left, a sidebar lists various AngularJS modules and directives. The main content area is titled `$http` and `service in module ng`. It includes a 'Description' section explaining that `$http` is a core Angular service for communicating with remote HTTP servers via `XMLHttpRequest` or `JSONP`. It also mentions `$httpBackend mock` for unit testing and `$resource` for a higher level of abstraction. The 'General usage' section shows a code example of how to use the `$http` service to make a GET request.

```
1. $http({method: 'GET', url: '/someUrl'}),
2.   success(function(data, status, headers, config) {
3.     // this callback will be called asynchronously
       // the response is available
4.   },
5.   error(function(data, status, headers, config) {
6.     // called asynchronously if an error occurs
7.   })
```

AngularJS dispone de una serie de APIs para comunicarse con cualquier backend, realizando peticiones XMLHttpRequest (XHR), o bien peticiones JSONP a través del servicio `$http`.

Además, existe un servicio llamado `$resource`, especializado en la comunicación con interfaces RESTful.

El servicio `$http` consiste en una API de propósito general para realizar peticiones XHR y JSONP. Es una API bastante sólida y sencilla de usar.

El servicio `$http` ofrece una serie de funciones que reciben como parámetros una URL y un objeto de configuración, para generar una petición HTTP. Devuelve una promesa de resultados con dos métodos: `success` y `error`.

Los métodos son equivalentes a los que podríamos hacer en una petición HTTP.

Para hacer las pruebas haremos uso de los servicios situados en [JSONPlaceholder](http://jsonplaceholder.typicode.com/), que permite hacer uso del servicio `$http` sobre sus servidores, ya que tiene habilitado el soporte para CORS.

<http://jsonplaceholder.typicode.com/>

### `$http.get`

Realiza una petición GET, para obtener datos.

Parámetros:

- **url** : URL destino
- **config** : objeto de configuración opcional. A destacar el atributo `params`, que contiene un mapa de los parámetros a pasar.

```

.module('httpModule', [])
.controller('MainCtrl', function($scope, $http){
  $http.get(
    'http://jsonplaceholder.typicode.com/posts', { params: {id:1} }
  )
  .success(function(data){
    $scope.resultdata = data;
  })
  .error(function(data){
    alert('Se ha producido un error')
  });
});

```

```

<div ng-app="httpModule" ng-controller="MainCtrl">
<h1>Resultado</h1>
<pre>
{{ resultdata[0] | json }}
</pre>
<hr />
</div>

```

<http://codepen.io/jmortega/pen/mAdVmV>

```

angular
  .module('httpModule', [])
  .controller('MainCtrl', function($scope, $http){
    $http.get(
      'http://jsonplaceholder.typicode.com/posts/1/comments', {}
    )
    .success(function(data){
      $scope.resultdata = data;
    })
    .error(function(data){
      alert('Se ha producido un error')
    });
  });
});

```

## \$http.post

Realiza una petición POST , para dar de alta algún dato en el servidor.

Parámetros:

- url : URL destino
- data : datos a enviar.
- config : objeto de configuración opcional.

El siguiente ejemplo se encarga de dar de alta un usuario, pasando los datos del usuario como parámetro de la petición post.

<http://jsonplaceholder.typicode.com/users>

<http://codepen.io/jmortega/pen/ozNbqN>

```
angular
  .module('httpModule', [])
  .controller('MainCtrl', function($scope, $http,$log){
    $http.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded;charset=utf-8';
    $http.post(
      'http://jsonplaceholder.typicode.com/users',
      {
        "name": " name",
        "username": "username",
        "email": "user@domain.com",
        "phone": "123456",
        "website": "http://angular.es"
      }
    )
    .success(function(data){
      $log.info(data)
      $scope.id = data.id;
    })
    .error(function(error){
      alert('Se ha producido un error'+error);
      $log.info(error)
    });
  });
```

## \$http.put

Realiza una petición PUT, para actualizar algún elemento en el servidor.

Parámetros:

- url : URL destino
- data : datos a enviar
- config : objeto de configuración opcional.

El siguiente ejemplo se encarga de actualizar el usuario con id = 10.

<http://codepen.io/jmortega/pen/ozNbqN>

```
angular
.module('httpModule', [])
.controller('MainCtrl', function($scope, $http){
  $http.put(
    'http://jsonplaceholder.typicode.com/users/10',
    {
      "name": "name",
      "username": "username",
      "email": "user@domain.es",
      "address": {
        "street": "address",
      },
      "phone": "123456",
      "website": "http://angular.es"
    }
  )
  .success(function(data){
    $scope.data = data;
  })
  .error(function(data){
    alert('Se ha producido un error')
  });
});
```

## Por qué son útiles los servicios

En lugar de llenar el controlador con código para obtener los datos de tiempo en un servidor, es mejor mover esta lógica independiente en un servicio para que pueda ser reutilizado por otras partes de la aplicación.

Los servicios son una manera de hacer la lógica de comunicación independiente, como en “forecast” allí tenemos todo lo que tenga que ver con la obtención de datos del tiempo de un servidor, sin mezclarlo con el resto de las acciones de nuestro controlador.

De esta forma le damos independencia y si más adelante en otro lugar de la aplicación o en otra aplicación tenemos que tomar datos del tiempo no tenemos más que utilizar el servicio que acabamos de crear y de esta manera estamos dándole **reusabilidad** a nuestro código.



## Service \$location

[https://docs.angularjs.org/api/ng/service/\\$location](https://docs.angularjs.org/api/ng/service/$location)

**\$location** parsea la URL de la barra del navegador (basado en el tradicional `window.location` y crea una URL válida para nuestra aplicación. Los cambios en **\$location** son reflejados en la barra del navegador y las modificaciones de la barra del navegador son reflejados en **\$location**.

**\$location** es un servicio ("service") que nos sirve para mantener y trasladar información de la ruta actual del navegador. **\$location** implementa una interfaz para el acceso a la propiedad nativa de Javascript `window.location`, y nos sirve para acceder a elementos de la ruta actual, conocer su estado y modificarlo. Por tanto, podemos saber en todo momento qué ruta tenemos y conocer cuando el usuario ha navegado a otro lugar, ya sea con los botones de atrás o adelante o pulsando un enlace.

Pero no solo eso, existe un enlace entre **\$location** y `window.location` en las dos direcciones. Todo cambio que realicemos en **\$location** también tendrá una respuesta en lo que se está mostrando en la barra de direcciones del navegador. Así como todo cambio de URL en el navegador tiene un efecto en el servicio **\$location**.

**\$location** tiene una serie de métodos bastante interesante a través de los cuales podemos acceder a partes de la URL que está en la barra de direcciones, desde el dominio, el puerto, protocolo, hasta la ruta en sí de la aplicación.

En el ejemplo de antes estábamos usando **\$location.path()** que nos devuelve el camino actual.

Este método acepta dos juegos de parámetros.

- Si llamamos a **\$location.path()** sin parámetros, nos devuelve la ruta actual.
- Si llamamos a **\$location.path()** indicando un parámetro, entonces se cambia la ruta actual a aquello que le hayamos indicado. La ruta debe comenzar por "/" el método es lo suficientemente listo para que, si no tiene la barra, se la pone automáticamente.

En no pocas ocasiones queremos que la redirección se haga en función de que cierta lógica de negocio se haya aplicado correctamente o no. Esto implica que dicha redirección tenga que hacerse desde un controlador u otro servicio. Para hacerlo desde aquí, recurriremos al **servicio \$location**.

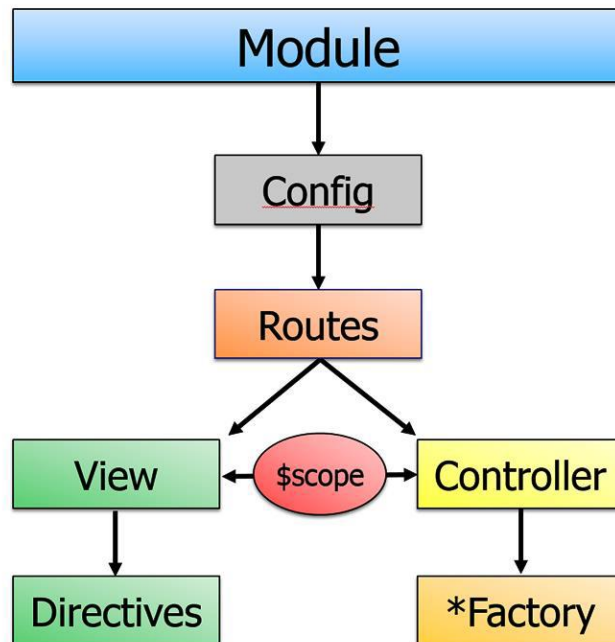
El servicio **\$location** parsea la URL de la barra de direcciones del navegador (según los valores de `window.location`), y hace que la URL esté accesible para nuestra aplicación. Todo cambio que se haga en la URL se verá reflejado en el servicio **\$location** y viceversa.

El **servicio \$location** realiza la siguientes acciones:

- Expone la URL actual de la barra de direcciones del navegador, para que podamos:
  - Observar la URL
  - Modificar la URL
- Sincroniza la URL de la barra de direcciones del navegador cuando el usuario:
  - Modifica la barra de direcciones
  - Hace clic en los botones forward o back del navegador (o hace click en un enlace histórico)
  - Hace clic en un enlace.

## Factorías (factory) en AngularJS

Un controlador no debería tener toda la lógica de la aplicación. En una aplicación bien estructurada, esta lógica la sacaríamos fuera a unos objetos que, en AngularJS, se llaman **Factories**, **Services** y **Providers**.



Las factorías nos podrían solucionar la pérdida de datos de los controladores cuando cambiamos la vista. Por ejemplo, algunas de las necesidades que podríamos tener y que las factorías nos podrían resolver son:

- 1) Compartir datos entre varios controladores, lo que permite tener aplicaciones capaces de memorizar estados entre varias de sus pantallas.
- 2) Compartir datos entre varias vistas distintas. Por supuesto, sin usar variables globales.

### Qué son las factorías

Las factorías son como contenedores de código que podemos usar en nuestros sitios desarrollados con AngularJS. Son un tipo de servicio, "service" en Angular, con el que podemos implementar librerías de funciones o almacenar datos.

Cuando las usamos tienen la particularidad de devolvernos un dato, de cualquier tipo. Lo común es que nos devuelvan un objeto de Javascript donde podremos encontrar datos (propiedades) y operaciones (métodos). Con diferencia de los controladores, las factorías tienen la característica de ser instanciados una única vez dentro de las aplicaciones, por lo que no pierden su estado. Por tanto, son un buen candidato para almacenar datos en nuestra aplicación que queramos usar a lo largo de varios controladores, sin que se inicialicen de nuevo cuando se cambia de vista.

Angular consigue ese comportamiento usando el patrón "**Singleton**" que básicamente quiere decir que, cada vez que se necesite un objeto de ese tipo, se enviará la misma instancia de ese objeto en lugar de volver a instanciar un ejemplar.

**Nota:** El patrón "Singleton" no es algo específico de AngularJS, en realidad es un patrón general de programación orientada a objetos.

## Ejemplo de factoría en AngularJS

Implementamos factorías con el método `factory()` que depende del módulo (objeto `module`) de nuestra aplicación.

El mecanismo para crear la factoría es el mismo que hacemos para crear los controladores. Para crear el controlador usas el método `controller()` y para la factoría el método `factory()`.

## Uso de services y factorías

La forma de poder compartir información o funcionalidades entre diferentes dependencias es utilizar Services y factorías. Con estos tipos de objetos de angularjs se pretende encapsular valores o funcionalidades que luego serán inyectadas a los controladores para ser usados.

Si por ejemplo tenemos un Service que se encarga de hacer un post a la lógica de negocio simplemente hay que poner como argumento en el controller el nombre del Service y este mágicamente aparecerá relleno.

Hay que tener en cuenta que los factories y services son siempre **objetos singleton**, con lo que si se inyectan en varios sitios el objeto que se inyecta es el mismo. No se va a inyectar un objeto nuevo cada vez.

AngularJS nos permite encapsular los datos de nuestra aplicación en una serie de elementos:

- Factorías
- Servicios
- Providers
- Valores
- Constantes

Los tres primeros (Factorías, Servicios y Providers), además de datos, nos permiten encapsular funcionalidades dentro de nuestra aplicación. Por ejemplo, si necesito mi lista de libros en múltiples controladores, lo correcto sería guardarlos en uno de estos elementos.

Estos tres elementos pueden realizar la misma funcionalidad, y la diferencia entre ellos radica en cómo se crean. Lo veremos más adelante.

Además, estos elementos implementan el **patrón singleton**, lo que los convierte en los candidatos perfectos para intercambiar información entre controladores.

Así, ahora vamos a modificar nuestro código para utilizar una factoría, donde podremos obtener el listado de profesores, así como añadir ítems a la lista. Una factoría en AngularJS devuelve un objeto javascript, con lo que su forma será la siguiente:

```
booksApp.factory('booksFactory', function(){
var books = [
{author:'Google', title:'Angular JS for web developers'},
{author:'Oreilly', title:'Angular JS in practice'},
{author:'Apress', title:'Angular JS directives'}];
return {
getBooks : function() {
return books;
},
addBook : function(newBook) {
books.push({
author : newBook.author,
title: newBook.title
})
}
}
});
```

Más adelante en el curso, veremos de qué manera podemos obtener ese listado de libros a través de una llamada AJAX o un servicio REST, en lugar de tener ese array *hardcoded* en nuestra factoría.

Al crear la factoría, podemos usarla en nuestro controlador simplemente inyectándola como parámetro. Así, ya estará disponible y podremos usarla:

```
booksApp.controller('booksCtrl', function($scope, booksFactory){
$scope.books = booksFactory.getBooks();
$scope.addBook = function() {
booksFactory.addBook($scope.newBook);
};
});
```

La inyección de dependencias no se limita sólo a controladores, como veremos más adelante, podemos inyectar una factoría en otra factoría, o cualquiera de los elementos propios de AngularJS.

Como hemos dicho, una factoría implementa el patrón singleton. Esto significa que se instancia una única vez (la primera vez que es requerida), y está disponible durante toda la vida de la aplicación. Ahora, si nos vamos al listado de libros y volvemos al de profesores, veremos que no perdemos los datos que hayamos podido introducir.

Otra cosa que debemos saber, es que AngularJS permite encadenar operaciones. Esto significa que podemos refactorizar nuestro código de la siguiente manera:

```
angular
.module('booksApp', ['ngRoute'])
.config(function($routeProvider){
...
})
.controller('booksCtrl', function($scope, booksFactory){
...
})
.factory('booksFactory', function(){
...
});
```

```
});
```

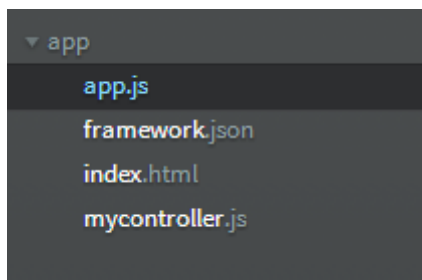
## Lógica de negocio en Angular

---

Al igual que ocurre en cualquier lenguaje de programación y con cualquier framework del tipo MV\*, nunca deberías tener tu lógica de negocio en un **controlador**, sino inyectar a este servicios que te provean de esta.

Planteamos un simple **módulo** con un **controlador** y una llamada \$http para recuperar una lista de valores desde un archivo json en el servidor.

### Estructura del proyecto



#### app.js

Función responsable de la definición del módulo.

```
(function(angular) {  
    var app = angular.module("myModule", []);  
}(angular));
```

#### frameworks.json

Archivo json estático con los datos que queremos recuperar y mostrar en nuestra vista.

```
[  
  {"frameworkId": "1", "name": "AngularJS"},  
  {"frameworkId": "2", "name": "Ember"},  
  {"frameworkId": "3", "name": "Backbone"},  
  {"frameworkId": "4", "name": "Knockout"}  
]
```

### myController.js

En este controlador podemos observar que estamos recuperando los datos llamando al servicio 'frameworksService' y asignándolos al objeto model de nuestro scope

```
(function(myModule) {  
  
    function controller($scope,$http,frameworkService) {  
        $scope.model = $scope.model || [];  
        frameworkService.getAll().then(function(data) {  
            $scope.model = data;  
        });  
    }  
  
    myModule.controller('myController',controller);  
}(angular.module("myModule")));
```

### frameworkService.js

El servicio **frameworkService** realiza una petición http para recuperar el contenido del archivo **framework.json** y devuelve una promise que enlazamos directamente a la vista por medio del objeto model.

```
(function(myModule) {  
    function service($http,$q) {  
        this.getAll=function() {  
            var path='frameworks.json';  
            var promise = $http.get(path).then(  
                function(response) {  
                    return response.data;  
                },  
                function(response) {  
                    return $q.reject(response);  
                }  
            );  
            console.log('Promise'+promise);  
            return promise;  
        };  
    }  
  
    myModule.service('frameworkService',service);  
}(angular.module('myModule')));
```

## index.html

```
<html>
  <head>
    <title>Logica AngularJS</title>
  </head>
  <body ng-app="myModule" ng-controller="myController">
    <ul>
      <li ng-repeat="item in model">
        <span>framework:{{item.frameworkId}}- {{item.name}}</span>
      </li>
    </ul>
    <!-- Include the core AngularJS library -->
    <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular.min.js">
    </script>
    <script src="app.js"></script>
    <script src="myController.js"></script>
    <script src="frameworkService.js"></script>
  </body>
</html>
```

## Añadir capa provider

La receta de un Provider se define como un tipo propio que implementa un método \$get .Este método es una función factoría, como el que se usa en una factory . De hecho, si definimos una factory , lo que se hace es crear un Provider vacío cuyo método \$get apunta directamente a nuestra función.

Deberíamos usar un Provider cuando queremos introducir cierta configuración que esté disponible a para toda la aplicación. Para asegurar esto, esto debe hacerse antes de que se ejecute la aplicación, en una fase llamada fase de configuración. De esta manera, podemos crear servicios reutilizables cuyo comportamiento podría cambiar ligeramente entre aplicaciones.

Un provider es como un factory pero permite que se configure antes de crear el valor del servicio. En el tema anterior vimos el ejemplo del servicio de hash que se configuraba a través de un value llamado algoritmo. Aunque el ejemplo funciona, la verdad es que es un poco chapucero ya que aparentemente no hay relación entre algoritmo y hash . La relación entre ambos queda poco orientada al objeto. El provider viene en nuestra ayuda creando un objeto previo que permite configurar el factory antes de que cree el valor del servicio. Este nuevo objeto se llama **Provider** y en un bloque config podremos acceder a él para poder configurar nuestro servicio.

#### frameworkProvider.js

```
(function(myModule) {  
    function provider() {  
  
        var path;  
        this.setPath=function(paramPath) {  
            path = paramPath;  
        };  
  
        this.$get = ['$http', function($http) {  
            return {  
                getAll: function() {  
                    console.log('path '+path)  
                    return $http.get(path);  
                }  
            };  
        }];  
    }  
    myModule.provider('frameworkProvider',provider);  
})(angular.module('myModule'));
```

#### config.js

Para utilizar el provider lo que hay que hacer es configurarlo en el config.js inyectándolo con el nombre de **frameworkProvider+Provider**, en nuestro caso **frameworkProvider**, con lo cual lo que haremos será establecer el path mediante la función **setPatch** en nuestro config.

```
myModule.config(function(frameworkProviderProvider){  
    frameworkProviderProvider.setPath('framework.js');  
});  
  
(function(myModule) {  
    function config(frameworkProviderProvider) {  
        frameworkProviderProvider.setPath('frameworks.json');  
    }  
    myModule.config(config);  
})(angular.module("myModule"));
```

Observemos que el Provider ha sido inyectado en la función de configuración. Esta inyección la hace un provider injector, que es distinto del inyector de instancias habitual que usaremos en el resto de nuestra aplicación. Este inyector únicamente trabaja con providers, ya que el resto de objetos no se crean en la fase de configuración.

Durante la inicialización de la aplicación, antes de que AngularJS haya creado ningún servicio, configura e instancia los providers. A esto lo llamamos la **fase de configuración** del ciclo de



vida de la aplicación. Durante esta fase, como hemos dicho, los servicios no son accesibles porque aún no se han creado.

Una vez se ha finalizado la fase de configuración, ya no se puede interactuar con un Provider y empieza el proceso de crear servicios. A esta parte del ciclo de vida de la aplicación se le conoce como **fase de ejecución**.

Para invocar al provider, después lo puedes utilizar en un servicio, factoría, controlador o directiva pero inyectándolo sin la palabra Provider, es decir como **frameworkProvider**, teniendo acceso de esta forma sólo a la función `getAll()` que retornas en `this.$get`.

```
(function(myModule) {  
    function service($http,$q,frameworkProvider){  
        this.getAll=function() {  
            var path='frameworks.json';  
            var promise = frameworkProvider.getAll().then(  
                function(response) {  
                    return response.data;  
                },  
                function(response) {  
                    return $q.reject(response);  
                }  
            );  
            console.log('Promise'+promise);  
            return promise;  
        };  
    }  
    myModule.service('frameworkService',service);  
})
```

## Integración con servicios REST: el servicio \$resource

---

El servicio \$http nos da la posibilidad de interactuar con este tipo de servicios de manera sencilla. Sin embargo, disponemos de otro servicio, **\$resource**, que nos permite hacer lo mismo eliminando además el código redundante.

El servicio \$resource se distribuye en un módulo separado del core de AngularJS llamado **ngResource**. Es por ello que tendremos que descargarnos su código fuente y declarar una dependencia con este módulo donde lo vayamos a utilizar.

<https://code.angularjs.org/1.5.8/angular-resource.js>

En primer lugar, crearemos un resource para la colección de usuarios del servicio:

```
var User = $resource('http://jsonplaceholder.typicode.com/users/:id',{id:'@id'});
```

A partir de esta URL, el servicio \$resource creará por nosotros una serie de métodos para interactuar con el servicio RESTful.

Si nos centramos en la sintaxis de la declaración, vemos que recibe dos parámetros:

El primero es obligatorio, y consiste en una URL que puede estar parametrizada. Los parámetros irán siempre prefijados por el símbolo de los dos puntos : , de igual manera que hacíamos con los servicios de routing.

En cuanto al segundo parámetro, es opcional y consiste en el conjunto de valores por defecto para los parámetros de la URL. Podemos sobreescribirlos luego en las llamadas a métodos concretos.

Veamos las operaciones que podemos realizar con él:

### Método Query

```
User.query(params, successCallback, errorCallback)
```

Realiza una petición GET , y espera recibir un array de ítems en la respuesta JSON.

### Método Get

```
User.get(params, successCallback, errorCallback)
```

Realiza una petición GET al servidor, y espera recibir un objeto como resultado de la respuesta JSON.

<http://codepen.io/jmortega/pen/dpyrwk>

Podemos acceder a la promesa de resultados que genera la petición del servicio \$http de la siguiente manera:

```
angular.module('restful', ['ngResource'])
.controller('MainCtrl', function($scope, $resource){
  var User = $resource(
    'http://jsonplaceholder.typicode.com/users/:id',
    {id:'@id'}
  );

  var userList = User
    .query().$promise
    .then(function(userList) {
      $scope.userList = userList;
    });

  var user = User.get({id:1}, function(user) {
    $scope.user = user;
  });
});
```

## Método Save

**User.save(params, payloadData, successCallback, errorCallback)**

Envía una petición POST al servidor. El cuerpo de la petición será el objeto que pasemos en el atributo payloadData .

<http://codepen.io/jmortega/pen/qgWjwN>

```
var User = $resource(
  'http://jsonplaceholder.typicode.com/users/:id',
  {id: '@id'}
);

User.save(
  userToSave,
  function(){
    $scope.message = 'usuario guardado con éxito';
  },
  function(){
    $scope.message = 'error al guardar';
  }
);
});
```

En este caso, hemos introducido función de callback de error, ya que la API no nos permite realizar peticiones POST.

## Método Delete

```
User.delete(params, successCallback, errorCallback)
```

```
User.remove(params, successCallback, errorCallback)
```

Realiza una petición HTTP DELETE al servidor.

<http://codepen.io/jmortega/pen/XNraKm>

```
angular.module('restful', ['ngResource'])
.controller('MainCtrl', function($scope, $resource){
  var User = $resource(
    'http://jsonplaceholder.typicode.com/users/:id',
    {id: '@id'}
  );

  User.delete(
    {id: 1},
    function(){
      $scope.message = 'Usuario eliminado correctamente'
    },
    function(){
      $scope.message = 'Error al eliminar'
    }
  );
});
```

## Definiendo acciones nuevas

Los métodos vistos ( query , get , save y delete ) son los únicos métodos que proporciona el servicio \$resource, con el que podríamos comunicarnos con una gran cantidad de servicios RESTful.

Pero, ¿qué pasa si me comunico con una API que usa POST para guardar ítems nuevos, mientras espera PUT para actualizar ítems existentes? ¿Ya no es válido el servicio \$resource ?

Aunque no viene un método PUT por defecto en el servicio, sí que tenemos la posibilidad de crearlo. Es aquí donde entra en juego el tercer parámetro que habíamos obviado hasta ahora en la creación del servicio.

En él podemos definir nuevas acciones en nuestro servicio. Se trata de un *hash* donde declararemos todas las acciones custom que queramos añadir.

En este ejemplo vemos cómo podemos realizar una petición del tipo PUT para actualizar los datos de un determinado user.

<http://codepen.io/jmortega/pen/WGNxEv>

```
angular.module('restful', ['ngResource'])
.controller('MainCtrl', function($scope, $resource){
  var userToUpdate = {
    "id": 10,
    "name": "name",
    "username": "username",
    "email": "user@domain.com",
  };

  var User =
  $resource('http://jsonplaceholder.typicode.com/users/:id',
  {id:'@id'},
  {update: {method:'PUT'}}
  );

  User.update(userToUpdate, function(data){
    $scope.message = data;
  }, function(){
    $scope.message = 'error al actualizar';
  });
});
```

## angular-mocks

<https://docs.angularjs.org/api/ngMock>

Muy importante la librería angular-mocks. Ésta nos dará soporte para inyectar y mockear servicios de AngularJS en nuestros tests unitarios. También extiende varios servicios del core de AngularJS para que sean controlados de manera síncrona en nuestros tests (como veremos, por ejemplo, a la hora de hacer mocks de servicios HTTP).

### Incluir librería

```
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/angular-mocks.js" data-semver="1.5.8" data-require="angular-mocks@1.5.8"></script>
```

### Incluir dependencia como módulo

```
angular.module('eventApp',['ngRoute','ngMockE2E'])
```

En nuestro controlador, servicio o factoría realizamos la petición http

```
angular.module('eventApp')
.factory('eventFactory',function($http,$q){

    var eventFactory={ }

    eventFactory.getAllEvents=function(){
        var deferred=$q.defer();
        $http.get('events').then(function(response){
            deferred.resolve(response.data);
            console.log('eventFactory.getAllEvents',response.data);
        },function(error){
            deferred.reject(error);
            console.log('eventFactory.getAllEvents',error);
        });
        return deferred.promise;
    }
})
```

[https://docs.angularjs.org/api/ngMock/service/\\$httpBackend](https://docs.angularjs.org/api/ngMock/service/$httpBackend)

Lo más importante es el uso del servicio `$httpBackend`. Éste nos permite implementar llamadas falsas a un servicio y simular los resultados que queramos obtener en cada test. Al inicio de nuestro test escribimos el resultado que queremos probar en cada caso, y una vez llamada a la función que hace uso del servicio, deberemos realizar una llamada al método `$httpBackend.flush()` para que todas las llamadas al servicio `$http` que se hayan hecho en el controlador reciban su respuesta.

```
angular.module('eventApp')
.run(function($httpBackend) {

    $httpBackend.whenGET(/views\/.*/).passThrough();
    $httpBackend.whenGET(/services\/.*/).passThrough();

    var events=[{'id':1,'name':'angularcamp','description':'angular conferences',
    'location':'Barcelona','email':'angularcamp@camp.org','date':'2016-01-01',
    'special':true},
    {'id':2,'name':'angularconf','description':'angular conferences',
    'location':'San francisco','email':'angularconf@conf.org','date':'2016-10-10',
    'special':false}]

    //return current event list
    $httpBackend.whenGET('events').respond(events);

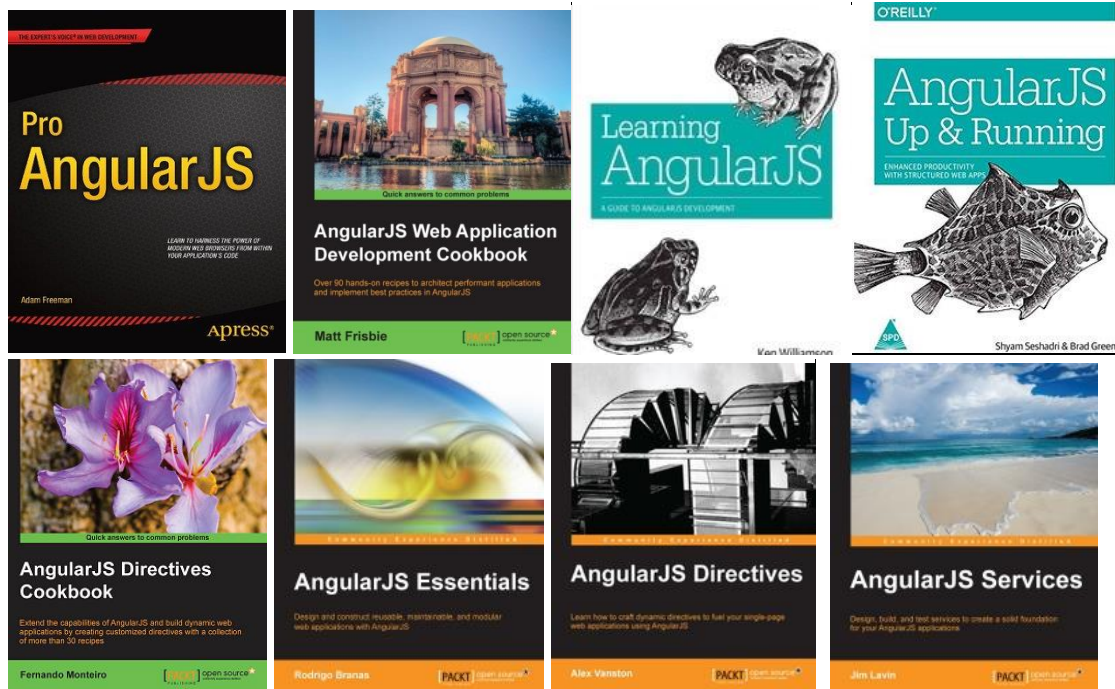
    $httpBackend.whenPOST('add-event').respond(function(method,url,data) {
        var event =angular.fromJson(data);
        event.id=events.length+1;
        events.push(event);
        return [200,{'addEventResult':true},{}]
    });

    $httpBackend.whenPOST('update-event').respond(function(method,url,data) {
        var event =angular.fromJson(data);
        for(i=0;i<events.length;i++){
            if(events[i].id==event.id){
                events[i]=event;
                break;
            }
        };
        return [200,{'events':events},{}]
    });
});
```



## Bibliografía

- Adam Freeman, PRO AngularJS, Apress, 2014
- Matt Frisbie, AngularJS Web Application Development Cookbook, Packt Publishing, 2014
- Ken Williamson, Learning AngularJS, O' Reilly, 2015
- Brad Green, Learning AngularJS Up & Running, O' Reilly, 2015
- Fernando Monteiro, AngularJS Directives Cookbook, Packt Publishing, 2015
- Rodrigo Branas, AngularJS Essentials, Packt Publishing, 2015
- Alex Varnston, AngularJS Directives, Packt Publishing, 2015
- Jim Lavin, AngularJS Services, Packt Publishing, 2015



## Sitios web de referencia y consulta

<https://angularjs.org>

<http://www.w3schools.com/angular>

<http://www.codecademy.com/es/learn/learn-angularjs>

<http://www.desarrolloweb.com/manuales/manual-angularjs.html>

<http://angularjs.blogspot.com.ar>

<https://blog.angularjs.org/>

<http://cursoangularjs.es/doku.php>

## Sitios realizados con angular

<https://www.madewithangular.com>

<https://www.eduonix.com/blog/web-programming-tutorials/top-15-websites-and-apps-built-with-angularjs/>

## Repositorios github

[https://github.com/jmortega/curso\\_angular\\_2016](https://github.com/jmortega/curso_angular_2016)  
<https://github.com/Escuelalt/codigo-bases-angular-curso-frontedge>  
<https://github.com/siddharta1337/Manual-del-querrero-AngularJS>  
<https://github.com/suryatech/codeschool-shaping-up-with-angular>  
<https://github.com/young-zhang/angularjs-fundamentals>  
<https://github.com/planetoftheweb/angular>  
<https://github.com/yearofmoo>

## Angular 2

<http://www.typescriptlang.org>  
<http://academia-binaria.com/angular2-primeras-impressiones/>  
<https://universal.angular.io/>

## Reactive Components for Modern Web Interfaces

<https://vuejs.org>  
<http://www.angular-meteor.com>

Uno de los mejores sitios de videotutoriales de AngularJS

<http://egghead.io>

También, hacer especial mención a la web <http://ngmodules.org/>. En ella encontramos una base de datos con un extenso repertorio de módulos hechos por la comunidad y que pueden servirnos en muchos de nuestros proyectos. Encontramos módulos tan usados como angularui o angularartics.