UNIVERSIDAD TECNOLÓGICA NACIONAL

Facultad Regional Córdoba

Secretaria de Extensión Universitaria

Área Tecnológica de Educación a Distancia

Coordinador General de Educación a Distancia:

Magíster Leandro D. Torres

Curso:

DIPLOMATURA EN DESARROLLO WEB EV 15030

Módulo:

"Base de datos no relacionables"

Autor:

Ing. Ricardo Martín Espeche

Referencias

Este material se confecciono en base a las últimas tendencias que hoy se encuentra en Internet. Para eso se buscó el mejor material y se recompilo dicha información para facilitar el aprendizaje del mismo.

El material utilizado para esta primera unidad fue:

- Manual de AngularJS
- Como aprender AngularJS – Raúl Expósito
- MongoDB desde Cero – Jonathan Wiesel

Índice

Capítulo 5
Angular JS

5.1 Introducción básica

AngularJS es Javascript. Es un proyecto de código abierto, realizado en Javascript que contiene un conjunto de librerías útiles para el desarrollo de aplicaciones web y propone una serie de patrones de diseño para llevarlas a cabo. En pocas palabras, es lo que se conoce como un framework para el desarrollo, en este caso sobre el lenguaje Javascript con programación del lado del cliente.

5.1.1 Mejoras del HTML

Este Javascript pretende que los programadores mejoren el HTML que hacen. Que puedan producir un HTML que, de manera declarativa, genere aplicaciones que sean fáciles de entender incluso para alguien que no tiene conocimientos profundos de informática. El objetivo es producir un HTML altamente semántico, es decir, que cuando lo leas entiendas de manera clara qué es lo que hace o para qué sirve cada cosa.

Lógicamente, AngularJS viene cargado con todas las herramientas que los creadores ofrecen para que los desarrolladores sean capaces de crear ese HTML enriquecido. La palabra clave que permite ese HTML declarativo en AngularJS es "directiva", que no es otra cosa que código Javascript que mejora el HTML. Puedes usar el que viene con AngularJS y el que han hecho terceros desarrolladores, puesto que muchas personas están contribuyendo con pequeños proyectos -independientes del propio framework- para enriquecer el panorama de directivas disponibles. Hasta este punto serás un "consumidor de directivas", y finalmente cuando vayas tomando experiencia serás capaz de convertirte en un "productor de directivas", enriqueciendo tú mismo las herramientas para mejorar tu propio HTML.

5.1.2 Patrón de diseño

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 Aplication).

5.1.3 Elementos y conceptos dentro de AngularJS

Ahora vamos a hacer un breve recorrido para nombrar y describir con unos pequeños apuntes aquellos elementos y conceptos que te vas a encontrar dentro de AngularJS.

Primeramente, tenemos que hablar sobre el gran patrón que se usa en Angular, el conocido Modelo, Vista, Controlador (MVC):

- **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.

Además del patrón principal, descrito hasta ahora tenemos los **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 espaguetti, archivos 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.

5.1.4 Dos mundos

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.

5.1.5 AngularJS o jQuery

Si queremos entrar en esta discusión, y para no liar a aquellos desarrolladores con menos experiencia, debemos decir que jQuery y AngularJS son librerías bien diferentes. El alcance y el tipo de cosas que se hacen con una y otra librería son distintos.

jQuery es una librería que nos sirve para acceder y modificar el estado de cualquiera de los elementos de la página. A través de jQuery y los selectores de CSS (así como los selectores creados por el propio jQuery) eres capaz de llegar a los elementos de la página, a cualquiera de ellos, y puedes leer y modificar sus propiedades, suscribirte a eventos que ocurran en esos elementos, etc. Con jQuery podíamos manejar cualquier cosa que ocurra en esos elementos de una manera mucho más cómoda que con Javascript "a pelo" y compatible con la mayor gama de navegadores.

Sin embargo, Angular pasa de ser una librería para convertirse en un framework de aplicaciones web. No solo te permite una serie de funciones y mecanismos para acceder a los elementos de la página y modificarlos, sino que también te ofrece una serie de mecanismos por los cuales extender el HTML, para hacerlo más semántico, incluso ahorrarte muchas líneas de código Javascript para hacer las mismas cosas que antes hacías con jQuery. Pero la principal diferencia y por la cual AngularJS toma la denominación de "framework", es que te marca una serie de normas y hábitos en la programación, principalmente gracias al patrón MVC implementado en AngularJS.

En este caso vamos a comparar el código que vimos en el Hola Mundo de AngularJS. La idea es simplemente escribir un nombre en un campo de texto y volcarlo en una etiqueta H1.

Esto en jQuery se podría hacer con un código como este:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Ej Hola Mundo desarrollado en jQuery</title>
</head>
<body>
    <h1>Hola</h1>
        <form>
            ¿Cómo te llamas? <input type="text" id="nombre">
        </form>

    <script src="https://code.jquery.com/jquery-1.11.1.min.js"></script>
    <script>
    $(function(){
        var campoTexto = $("#nombre");
        var titular = $("h1");
        campoTexto.on("keyup", function(){
            titular.text("Hola " + campoTexto.val());
        });
    });
    </script>
</body>
```

```
</html>
```

En AngularJS ya vimos el código necesario en un artículo anterior, pero lo reproducimos para poder compararlos fácilmente.

```
<!DOCTYPE html>
<html lang="en" ng-app>
<head>
    <meta charset="UTF-8">
    <title>Ej Hola Mundo desarrollado en AngularJS</title>
</head>
<body>
    <h1>Hola {{nombre}}</h1>
        <form>
            ¿Cómo te llamas? <input type="text" ng-model="nombre">
        </form>

    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.23/angular.min.js"></scr
ipt>
</body>
</html>
```

Como habrás observado, con AngularJS se simplifica nuestra vida. No solo que no necesitas hacer muchas de las cosas que en jQuery tienes que programar "a mano", como el acceso al campo de texto para recuperar el valor, la creación del evento de teclado para hacer cosas cuando se escribe dentro del campo de texto, etc. Lo que más debería llamar nuestra atención es que el código de AngularJS es mucho más entendible, incluso por personas que no tengan idea de programar. Como puedes deducir, en Angular escribimos generalmente menos código Javascript, pero sí es importante decir que debido al uso del patrón MV* (separación del código por sus diferentes objetivos), en ejemplos más avanzados observarás que entran en juego diferentes factores que complican las soluciones. Generalmente, aunque en la métrica de líneas de código AngularJS pueda ganar a jQuery, también debes saber que te exigirá mayor capacidad de abstracción y entender diversos conceptos de patrones de diseño de software.

Todo ello nos debe hacer entender mejor la potencia de AngularJS. Pero ojo, sin desmerecer a jQuery, pues debe quedar claro que cada librería es útil en su campo.

5.2 Primeros pasos con AngularJS

Si quieres trabajar con AngularJS tienes que incluir el script del framework en tu página. Esto lo puedes hacer de varias 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. En principio es indiferente a nivel didáctico, así que nosotros vamos a comenzar del modo más sencillo, que es utilizar el CDN.

Accedes a la página de AngularJS: https://angularjs.org/

Pulsas el botón de descarga y encontrarás diversas opciones. Escoges la versión del framework (si es que te lo permite) y que esté minimizada (minified). Luego encontrarás un campo de texto donde está la URL de la librería (esa URL está marcada con las siglas "CDN" al lado). Ya sabes que el CDN te ofrece un contenido, en este caso un script Javascript, que está alojado en otro servidor, pero que lo puedes usar desde tu página para mejorar la entrega del mismo.

5.2.1 Incluir librerías en nuestras paginas

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.

5.2.2 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. Por ahora puedes quedarte simplemente con la necesidad de informar a AngularJS del contenedor HTML donde va a desplegar su "magia".

Típicamente pondrás ng-app en la etiqueta HTML de inicio del documento.

```
<!DOCTYPE html>
<html lang="en" ng-app>
<head>
<meta charset="UTF-8">
<title>Ej de AngularJS</title>
</head>
<body>

... Aquí el cuerpo de tu página ...

<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.23/angular.min.js"></scr
ipt>
</body>
</html>
```

Así como ng-app, existen muchas otras directivas para enriquecer tu HTML, aun veremos alguna más en este artículo.

5.2.3 Hola Mundo

Ahora vamos a poner algo de carne en el asador y vamos a observar la potencia de AngularJS con muy poco código. Realmente, como observarás, se trata de cero códigos (Javascript).

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 ng-model 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. Insisto en que más adelante hablaremos con detalle sobre estos datos y veremos nuevos ejemplos.

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 vemos, 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.

5.3 Binding

El binding es algo que conoces ya en Javascript común, e incluso en librerías como jQuery, aunque es posible que no le has dado ese nombre todavía. 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".

Además, 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.

5.3.1 Que es binding?

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.

5.3.1.1 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 estarías 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).

5.3.1.2 Two-way binding

En este segundo caso la información fluye desde el scope hacia la parte visual (igual que en "one-way 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.

5.3.2 Ejemplo del binding

Realmente en los ejemplos más sencillos de AngularJS ya hemos experimentado el binding. No me hace falta ni Javascript para poder hacer un ejemplo donde se pueda ver en marcha perfectamente.

```
<div ng-app>
  <input type="text" ng-model="dato" />
  {{dato}}
  <input type="button" value="hola" ng-click="dato='hola'" />
</div>
```

Mira, tienes tres elementos destacables.

- Un campo de texto con ng-model="dato". En ese campo tenemos un "doble binding".
- Una expresión {{dato}} donde se mostrará aquello que haya escrito en el campo de texto. En este caso tenemos un binding simple, de un único sentido.
- finalmente tienes un botón que no tiene bindig alguno. Simplemente, cuando lo pulses, estás ejecutando una expresión (con ng-click) para cambiar el scope en su variable "dato".

El binding one-way lo ves muy fácilmente. Simplemente escribe algo en el campo de texto y observa cómo se actualiza el lugar de la página donde estás volcándo el valor con {{dato}}.

Para poder ver bien el doble binding he tenido que crear el botón, que hace cambios en el scope. Aclaro de nuevo que el doble binding de todos modos lo tienes en el INPUT text. Dirección 1) lo ves cuando escribes datos en el campo de texto, que viajan al modelo automáticamente (y sabemos que es cierto porque vemos cómo {{dato}} actualiza su valor. Dirección 2) lo ves cuando haces click en el botón. Entonces se actualiza ese dato del modelo y automáticamente viaja ese nuevo valor del scope hacia el campo de texto.

5.3.4 La utilidad del binding

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.

5.4 Directivas y 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".

5.4.1 Directivas

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.

5.4.1.1 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. Se leería "Enyi ap" y 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>
```

Para no causar confusiones, también podemos agregar que 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.

5.4.1.2 Directiva ngModel (ng-model)

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 lo experimentarás y entenderás mejor un poco más adelante cuando nos metamos más de lleno en los controladores.

5.4.2 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 podemos una gama de tipos de expresiones, de momento 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 " + "DesarrolloWeb" }}
```

Al ejecutarlo, AngularJS sustituirá esa expresión por "Hola DesarrolloWeb". 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.

5.4.3 Otras directivas

Seguimos aprendiendo AngularJS y en este apartado vamos a avanzar un poquito más y hacer un ejemplo práctico en el que no tenemos todavía necesidad de escribir código Javascript. Es interesante entretenerse con estos ejemplos, pues resultan muy sencillos y nos ayudan a mantener una progresión muy asequible en el principio de nuestro aprendizaje. También es apropieado para observar bien lo que a veces llamamos la "magia de AngularJS". Veremos

que, con muy poco o nada de código, se pueden hacer cosas medianamente importantes, al menos para los desarrolladores que sabemos lo laborioso que sería montar esto por nuestra cuenta con Javascript a secas.

Nos servirá también para aprender nuevas directivas, de las más sencillas y usadas habitualmente, como son ngInit y ngClick o ngRepeat. Así que si os parece comenzamos con un poco de teoría para explicar estas directivas.

5.4.3.1 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.

```
<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.

5.4.3.2 Directiva 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 y se verá con detenimiento en decenas de ejemplos. De momento puedes quedarte 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 própio 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.

Esta directiva es bastante sofisticada y explicar cada una de sus posibilidades nos llevaría algo de tiempo. Sin embargo, 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. Osea, 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)

5.4.3.3 Directiva ngClick

Terminamos nuestra dosis de teoría con 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.

5.4.4 Ejemplo completo

Ahora que hemos conocido las directivas, nos falta ponerlo todo junto para hacer un pequeño ejercicio básico con AngularJS.

En esta aplicación tenemos un campo de texto para escribir cualquier cosa, un "pensamiento". Al pulsar sobre el botón se agregará dentro de un array llamado "pensamientos" lo que se haya escrito en el campo de texto. Además, encuentras un bucle definido con ng-repeat que itera sobre el array de "pensamientos" mostrando todos los que se hayan agregado.

```
<div ng-app ng-init="pensamientos = [];">
    <h1>Altavoz AngularJS</h1>
    <p>
        ¿Qué hay de nuevo?
        <br />
        <input type="text" ng-model="nuevoPensamiento" />
        <input type="button" value="Agregar" ng-
click="pensamientos.push(nuevoPensamiento); nuevoPensamiento = '';" />
    </p>
    <h2>Pensamientos que has tenido</h2>
    <p ng-repeat="pensamiento in pensamientos" ng-init="paso = $index">
        Pensaste esto: {{pensamiento}} (Iteración con índice {{paso}})
    </p>
</div>
```

El dato del array "pensamientos" lo generas en el scope con el ng-init de la primera etiqueta. En el campo de texto tenemos la directiva ng-model para indicarle que lo que se escriba formará parte de nuestro modelo y se almacenará en la variable nuevoPensamiento. Como ves, en el ng-click se hace un push de ese nuevo pensamiento dentro del array de pensamientos. En ng-repeat se itera sobre la colección de pensamientos, escribiéndolos todos por pantalla, junto con el índice de ese pensamiento actual en el array "pensamientos".

5.4.5 Directiva ngClass

La idea es poder definir el aspecto de nuestra aplicación en base a los datos que tengamos en el modelo, aplicando unas class de CSS u otras dependiendo de valores de propiedades del scope o de expresiones que podamos construir.

Existen tres posibles variantes de tipos que acepta la directiva ngClass y que podemos usar siempre que deseemos.

- Asignarle una propiedad del scope que sea una cadena de texto. En este caso esa cadena se coloca como valor en el atributo class. Si en esa cadena existen varios nombres de clases separados por un espacio en blanco, esas clases se aplicarán en conjunto al elemento.
- Asignarle una propiedad del scope que contenga un array de cadenas. En ese caso se asignan como clases todos los nombres que haya en las casillas del array.
- Asignarle como valor a ng-class un objeto. En ese caso tendrá pares clave valor para especificar nombres de clases y expresiones que deban cumplirse para que éstas se apliquen. Lo veremos mejor con un ejemplo.

Ahora veremos ejemplos de cada una de las tres posibilidades comentadas.

5.4.5.1 Asignar una propiedad del scope que contiene una cadena

Es tan sencillo como indicar esa propiedad dentro del atributo ng-class, como se ve a continuación.

```
<h1 ng-class="vm.tamTitular">Acumulador</h1>
<select ng-model="vm.tamTitular">
    <option value="titularpeq">Peque</option>
    <option value="titulargran">Gran</option>
  </select>
```

Como puedes ver, tenemos un encabezado H1 con ng-class asignado a vm.tamTitular. Esa propiedad del scope se creó a partir de un campo SELECT que está a continuación. Por tanto, cuando cambie el option seleccionado en el campo, cambiará la class asociada al elemento H1.

Tal como habrás deducido, las dos posibles class que se le van a asignar al encabezado serán los value de los OPTION.

5.4.5.2 Asignar un array de cadenas

Es tan sencillo como definir un array de alguna manera y luego introducirlo dentro de nuestra directiva. Por facilitar las cosas voy a definir el array de manera literal en el controlador.

```
vm.clases = ["uno", "dos", "tres"];
```

Luego podremos asociar ese array de cadenas, colocando todos los nombres de clases a un elemento de la página con la directiva ngClass.

```
<h2 ng-class="vm.clases">Control de operación:</h2>
```

Como resultado de esa directiva a nuestro encabezado H2 se le van a aplicar las tres clases "uno", "dos" y "tres".

5.4.5.3 Asignar un objeto con uno o varios pares clave, valor

Este es el uso más complejo de ngClass, pero también el más potente. Nos permite definir expresiones y gracias a ellas Angular sabrá si debe colocar o no una clase CSS en concreto. Se ve bien con un ejemplo delante.

```
<p ng-class="{positivo: vm.total>=0, negativo: vm.total<0}">
    En el acumulador llevamos <span>{{vm.total}}</span>
  </p>
```

Este párrafo nos muestra un valor total de una cuenta. Ese valor lo sacamos de la propiedad vm.total y esa misma propiedad es la que usamos para definir la clase de CSS que vamos a asociar como estilo al párrafo.

Ahora echa un vistazo al atributo ng-class y verás que lo que le indicamos es un objeto, pues está entre llaves. Ese objeto tiene un número de pares clave/valor indeterminado, puede ser uno o varios. Cada uno de esos pares clave/valor nos sirven para definir si debe o no aplicarse una clase en concreto.

En la clave colocas el nombre de la clase, class de tu CSS, que Angular puede colocar si se cumple una expresión boleana. Como valor colocamos la expresión boleana a evaluar por AngularJS para que el sistema deduzca si se debe aplicar esa clase o no.

En nuestro ejemplo se aplicará la clase "positivo" en caso que la propiedad vm.total sea mayor o igual que cero. Se aplicará la clase "negativo" en caso que la propiedad vm.total tenga un valor menor que cero.

5.4.6 Directiva ngChecked, ngTrueValue, ngFalseValue y ngChecked

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 "checado".
- **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.

5.4.6.1 Directiva ngModel

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.

5.4.6.2 Directiva ngTrueValue y ngFalseValue

En tu modelo, la propiedad vm.activo podrá tener dos valores, true o false, que corresponden a los estados de estar marcado el checkbox o no marcado. Sin embargo, puede resultar útil tener otros valores en caso de estar activos o no, en vez del boleano. Para ello usas estas directivas.

```
<input type="checkbox" ng-model="vm.claroooscuro" ng-true-value="claro" ng-false-
value="oscuro" />
```

5.4.6.3 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()">
```

5.4.6.4 Ejemplo

Ahora puedes ver un código HTML que trabaja con campos checkbox y que pone en marcha los puntos vistos en este apartado.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Trabajando con checkboxes AngularJS</title>
</head>
<body ng-app="app" ng-controller="appCtrl as vm">

    <p>
        <input type="checkbox" ng-model="vm.activo" ng-change="vm.avisar()">
        Este campo es vm.activo y su valor en el modelo es {{ vm.activo }}.
        <br />
        Tiene además un ng-change asociado con el método vm.avisar().
    </p>
    <p>
        <input type="checkbox" ng-model="vm.claroscuro" ng-true-value="claro"
ng-false-value="oscuro" />
        Este campo tiene el value modificado. Ahora vale {{ vm.claroscuro }}
    </p>
    <p>
        <input type="button" ng-click="vm.activo=true" value="pulsa para cambiar
la propiedad del modelo del primer checkbox a true"> Observarás que aunque el
estado pueda cambiar, no se invoca al ng-change de ese primer checkbox.
    </p>

    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.24/angular.min.js"></scr
ipt>
    <script>
    var app = angular.module("app", [])
    app.controller("appCtrl", function(){
        var vm = this;
        //podríamos inicializar valores del modelo
        //vm.activo = false;

        vm.avisar = function(){
            console.log("cambié");
        }
    });
    </script>
</body>
</html>
```

Observa que al iniciarse por primera vez la página los valores del modelo (definidos en los checkboxes con ng-model) no están inicializados. Por ello las expresiones donde se vuelcan estos datos no muestran valor alguno. Luego, cuando cambias el estado de los checkboxes ya se empieza a ver el estado de los elementos. Esta situación podría cambiar solo con inicializar esas propiedades en el controlador. Lo hemos colocado en el código comentado para que se vea bien.

El otro detalle que igual no salta a la vista es la llamada a la función del modelo vm.avisar(), colocada en el ng-change del primer checkbox, que no se llama cuando se cambia el estado del campo como consecuencia del Javascript. Demostrar eso es el motivo de la existencia del botón.

5.5 Módulos

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.

5.5.1 Módulo para 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">
```

Recuerda que hasta ahora, en la directiva ngApp, atributo ng-app, no colocábamos ningún valor. Eso era porque hasta el momento no habíamos trabajado con módulos. Sin embargo, a partir de ahora que los aprendemos a utilizar recuerda que los vas a tener que indicar al arrancar la aplicación.

5.5.2 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(...){ ... })
```

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 bootstrap (arranque) de tu aplicación, comúnmente en la directiva ngApp.

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

5.5.3 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:

5.5.3.1 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( ... );
```

5.5.3.2 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( ... );
```

Estas dos opciones proponen estilos de código distintos. En la segunda opción, al encadenar las llamadas desde la creación del módulo hasta los controladores u otras cosas que necesitemos, nos estamos ahorrando la creación de una variable global, eliminando posibilidad de que colisione con otras del mismo nombre que haya en otras partes de tu aplicación. Se recomienda el uso de la Opción 2

5.5.4 Acceder a un modulo

El módulo lo crearás una vez, pero puedes necesitarlo en diversos puntos de tu código. Si lo has cacheado en una variable global, tal como se comentaba en la OPCIÓN 1) del punto anterior, podrás acceder a él en cualquier lugar de tu aplicación. Sin embargo, habíamos dicho que no era la opción más recomendable, así que necesitamos alguna otra manera de acceder a un módulo ya creado.

Lo conseguimos por medio del mismo método module() que utilizamos para crear el módulo, solo que en esta ocasión solo le indicaremos el módulo al que queremos acceder.

```
angular.module('nuevaApp')
```

Por si no has visto la diferencia con angular.module('nombreModulo', []) creamos un módulo y con angular.module('nombreModulo') recuperamos un módulo que haya sido creado anteriormente con el nombre 'nombreModulo'.

5.6 Controladores

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 ng-controller 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". En la terminología de Angular al modelo se le llama "scope" y dentro de poco vamos a explicarlo un poco mejor.

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.

5.6.1 Scope

Para poder introducir los controladores debemos detenernos antes en un concepto que se repite hasta la saciedad 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), que explicaremos con detalle en futuros artículos.

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.

Todo eso estaba muy bien, sin embargo, en términos de programación necesitamos un lugar donde escribir todo el Javascript necesario para implementar la lógica de la aplicación. Integrar el Javascript dentro del HTML no es nada recomendable, por diversos motivos que ya se conocen. Ya dijimos además que, dentro del código HTML, no se puede (o mejor dicho, no se debería) hacer cálculos, asignaciones de valores y en resumen código que represente la lógica de nuestras aplicaciones. Todo ello irá en los controladores.

5.6.2 Como usar 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.

5.6.2.1 Como aplicarlo

El código necesario para crear un controlador en AngularJS tendrá este aspecto:

```
var app = angular.module("miapp", []);
app.controller("miappCtrl", function(){
        var scope = this;
        scope.datoScope = "valor";

        scope.metodoScope = function(){
                scope.datoScope = "otra cosa";
}
});
```

Como puedes comprobar, resulta un tanto complejo. Además, tenemos que explicarte cómo se engancha este controlador dentro del código HTML. Para explicar todo eso con calma necesitamos alargarnos bastante, así que lo veremos en el siguiente artículo. Además, te explicaremos varias alternativas de código para trabajar con controladores que se usan habitualmente en AngularJS, con mejores y peores prácticas.

5.6.3 Variantes para crear controladores

El proceso de crear y usar un controlador es sencillo, pero no inmediato, requiere un pequeño guión 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.

En este punto queremos aclarar que existen diversas alternativas de código para crear el Javascript necesario para definir nuestro controlador. Vamos a comenzar por una que es muy habitual de ver en diversos tutoriales y documentación, aunque no es la mejor. Todas las formas funcionan, pero hay algunas que en determinadas circunstacias pueden ser mejores. Luego las estudiaremos y explicaremos el motivo por el que serían todavía más recomendadas para crear tus controladores.

5.6.3.1 Opción 1

```
angular
  .module('pruebaApp', [])
  .controller('pruebaAppCtrl', function($scope){
    $scope.algo = "Hola Angular, usando controller más simple";
  });
```

De esta manera estoy creando un módulo llamado "pruebaApp". Sobre el módulo invoco el método controller() y creo un controlador llamado "pruebaAppCtrl". En la creación del controlador verás que se especifica una función, ella es la que hace de constructora de nuestro controlador. A esa función le llega como parámetro $scope que es el nombre de variable donde tendrás una referencia al conodido "scope" que almacena los datos de mi modelo.

Dentro de la función del controlador hacemos poca cosa. De momento simplemente le asigno un valor a un atributo de $scope. He creado un atributo nuevo en el objeto $scope que no existía y que se llama "algo" ($scope.algo). Con ello simplemente le estoy agregando un dato al modelo. Gracias al "binding" o enlace, ese dato luego lo podré utilizar en el HTML.

De hecho, queremos ya echarle un vistazo al HTML que podríamos tener para usar este controller.

```
<div ng-app="pruebaApp" ng-controller="pruebaAppCtrl">
  {{algo}}
</div>
```

Como puedes ver, en el HTML indicamos el nombre del módulo que se usará para implementar esta aplicación (que corresponde con el nombre del módulo creado en el código Javascript) y el nombre del controlador (también corresponde con el nombre del controller que he creado con Javascript). Además, en el HTML podrás encontrar la expresión {{algo}} que lo que hace es volcar como texto en la página el contenido de la variable "algo". Esa variable pertenece al scope y la hemos inicializado en el código del controller.

El resultado es que en nuestra página aparecerá el texto "Hola Angular, usando controller", que es el valor que fue asignado en el atributo "algo" de $scope. De momento la aplicación no es nada espectacular pero lo interesante es ver cómo se usa un controller.

5.6.3.2 Opción 2

He colocado como primera opción porque es más habitual en tutoriales y puedes haberla leído en la documentación del framework o en diversas presentaciones por ahí. Sin embargo, es más recomendada esta segunda alternativa que os presentamos a continuación.

```
angular
```

```
  .module('pruebaApp', [])
  .controller("pruebaAppCtrl", function(){
    this.algo = "Esto funciona! Gracias Angular";
  });
```

Como estarás observando, el controlador es prácticamente igual. Ahora la diferencia es que no estás inyectando el $scope. Ahora el contexto donde adjuntar las variables que queremos enviar a la vista no lo sacamos de $scope, sino que lo obtenemos directamente a través de la variable "this".

En este caso, para cargar un nuevo dato al modelo, llamado "algo", lo hacemos a través de this.algo y le asignamos aquello que deseemos.

Esta alternativa implica algún cambio en el HTML con respecto a lo que vimos anteriormente.

```
<div ng-app="pruebaApp" ng-controller="pruebaAppCtrl as vm">
  {{vm.algo}}
</div>
```

El cambio fundamental lo encuentras al declarar la directiva, en el valor del atributo ng-controller.

```
ng-controller="pruebaAppCtrl as vm"
```

En este caso estamos diciéndole no solo el nombre del controlador, sino además estamos informando que dentro de ese DOM el scope será conocido con la variable "vm". Por tanto, ahora cuando deseemos acceder a datos de ese modelo que nos ofrece el controlador debemos indicar el nombre del scope.

```
{{vm.algo}}
```

Ahora que hemos visto estas dos opciones y dado que nos recomiendan la segunda, la utilizaremos de manera preferente a lo largo del manual. De momento esto es todo, esperamos que con lo que sabes ya tengas ideas para ir probando nuevos ejemplos. Nosotros en el siguiente apartado crearemos un controlador que será un poco más complejo y útil.

5.6.4 Ejercicios con controller

El ejercicio es muy sencillo en realidad. Es un "acumulador" en el que tenemos un campo de texto para escribir una cantidad y un par de botones con operaciones de sumar y restar. Luego tenemos un contador que se va incrementando con esas operaciones de sumas y restas sobre un total. En si no sirve para mucho lo que vamos a construir, lo hacemos más bien con fines didácticos.

5.6.4.1 Código HTML

Veamos primero la parte del HTML usada para resolver este ejercicio.

```
<div ng-app="acumuladorApp" ng-controller="acumuladorAppCtrl as vm">
  <h1>Acumulador</h1>
  <h2>Control de operación:</h2>
  ¿Cuánto? <input type="text" ng-model="vm.cuanto" size="4" />
  <br />
  <input type="button" value="+" ng-click="vm.sumar()"/>
  <input type="button" value="-" ng-click="vm.restar()"/>
  <h2>Totales:</h2>
  En el acumulador llevamos <span>{{vm.total}}</span>
</div>
```

Vamos describiendo los detalles más importantes que debes apreciar.

- Al hacer el bootstrap (arranque) de la aplicación (directiva ngApp) se indica el nombre del módulo: "acumuladorApp".
- Se indica el nombre del controlador con la directiva ngController y el valor "acumuladorAppCtrl".
- Con la sintaxis de "acumuladorAppCtrl as vm" indicamos que el scope dentro del espacio de etiquetas marcado para este controlador, se conocerá por "vm". Podríamos llamar como deseásemos al scope, en lugar "vm", en definitiva es como una variable donde tendremos propiedades y métodos. En otras palabras, se ha creado un namespace (espacio de nombres) para los datos que nos vienen del modelo (scope) gracias al controlador "acumuladorAppCtrl".
- Puedes llamar de cualquier manera también tanto a módulo como a controlador, pero se usan esos por convención.
- El campo INPUT de texto tiene una directiva ngModel para decirle a AngularJS que ese es un dato del modelo. Fítate que el nombre del dato en el modelo se accede a través del espacio de nombres definido en el controlador: "vm.cuanto".
- Luego encuentras dos INPUT tipo button que me sirven para realizar la acción de acumular, positiva o negativamente. Ambos tienen una directiva ngClick que nos sirve para expresar lo que debe ocurrir con un clic. Lo interesante aquí es que llamamos a dos funciones que están definidas en el scope, mediante el espacio de nombres "vm", nuestro modelo. El código de esos métodos (funciones que hacen en este caso de manejadores de eventos) ha sido definido en el controlador, lo veremos más adelante.
- Por último encontramos un {{vm.total}} que es un dato que estará en el scope y en el que llevamos la cuenta de todo lo que se ha sumado o restado con el acumulador.

5.6.4.2 Código Javascript

Ahora pasemos a la parte donde codificamos nuestro Javascript para darle vida a este ejercicio.

```
angular
  .module('acumuladorApp', [])
  .controller("acumuladorAppCtrl", controladorPrincipal);

function controladorPrincipal(){
  //esta función es mi controlador
  var scope = this;
  scope.total = 0;
  scope.cuanto = 0;

  scope.sumar = function(){
    scope.total += parseInt(scope.cuanto);
  }
  scope.restar = function(){
    scope.total -= parseInt(scope.cuanto);
  }
};
```

Vamos describiendo las principales partes del código.

- Con "angular" en la primera línea accedo a la variable que me crea angular, disponible para acceder a las funcionalidades del framework.
- En la segunda línea creamos el módulo, indicando el nombre (que es igual a lo que se puso como valor en la directiva ngApp del HTML) y el array de las dependencias, de momento vacío.
- En la tercera línea, con un encadenamiento (chaining) definimos el controlador. Indicamos como primer parámetro el nombre del controlador, definido en la directiva ngController del HTML, y como segundo parámetro colocamos la función que se encargará de construir el controlador. Indicamos el nombre de la función simplemente, sin los paréntesis, pues no es una llamada a la función sino simplemente su nombre.
- Luego se define la función del controlador. Esa función es capaz de escribir datos en el scope, así como métodos.
- En la función accedemos al scope por medio de "this". Fíjate que en la primera línea de la función tienes var scope = this; esto es simplemente opcional y se puede hacer para mantener la terminología de AngularJS de llamar scope a lo que vas generando en el controlador, pero podría perfectamente referirme a él todo el tiempo con "this".
- En el scope inicializo dos valores, total y cuanto, mediante scope.total=0 y scope.cuanto=0.

- Luego genero dos métodos que usaremos para los manejadores de eventos de los botones de sumar y restar. Esas funciones tienen el scope disponible también y en su código se accede y modifica a los datos del scope.

Con eso tenemos el ejercicio completo, todo lo demás para que esto funcione es tarea de AngularJS. De manera declarativa en el HTML hemos dicho qué son las cosas con las que se va a trabajar en la aplicación y luego hemos terminado de definir e inicializar los datos en el controlador, así como escribir en código las funcionalidades necesarias para que el ejercicio tome vida.

5.6.4.3 Variante sin “controller as”

En la resolución, que hemos comentado antes, a este ejercicio hemos usado una alternativa de la directiva de ngController en la que se le asigna un espacio de nombres al scope "acumuladorAppCtrl as vm". Esto se conoce habitualmente como "controller as" y ya comentamos en el artículo anterior dedicado a los controladores que es algo relativamente nuevo y que muchas veces la codificación que encontrarás en otros textos es un poco diferente.

Solo a modo de guía para quien está acostumbrado a trabajar de otra manera, y para que entiendas otros códigos antiguos que podrás encontrar en otras guías de Angular, pongo aquí el código de este mismo ejercicio, pero sin el "controller as".

El código HTML:

```
<div ng-app="acumuladorApp" ng-controller="acumuladorAppCtrl">
  <h1>Acumulador</h1>
  <h2>Control de operación:</h2>
  ¿Cuánto? <input type="text" ng-model="cuanto" size="4" />
  <br />
  <input type="button" value="+" ng-click="sumar()"/>
  <input type="button" value="-" ng-click="restar()"/>
  <h2>Totales:</h2>
  En el acumulador llevamos <span>{{total}}</span>
</div>
```

El código Javascript:

```
var acumuladorApp = angular.module('acumuladorApp', []);
acumuladorApp.controller("acumuladorAppCtrl", ["$scope", function($scope){
  //esta función es mi controlador
  //var $scope = this;
  $scope.total = 0;
  $scope.cuanto = 0;
```

```
  $scope.sumar = function(){
    $scope.total += parseInt($scope.cuanto);
  }
  $scope.restar = function(){
    $scope.total -= parseInt($scope.cuanto);
  }
}]);
```

Es muy parecido al anterior, simplemente se deja de usar el espacio de nombres y al definir la función del controlador se inyecta el $scope de otra manera.

5.7 Ajax

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.

5.7.1 Service $http

El servicio $http (service en inglés, tal como se conoce en AngularJS) 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.

Entre los shortcuts encuentras:

- $http.get()
- $http.post()
- $http.put()
- $http.delete()
- $http.jsonp()
- $http.head()
- $http.patch()

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.

El servicio $http es bastante complejo y tiene muchas cosas para aportar soluciones a las más variadas necesidades de solicitudes HTTP asíncronas. De momento nos vamos a centrar en lo más básico que será suficiente para realizar un ejemplo interesante.

5.7.1.1 Realizar una llamada a $http.get()

El método get() sirve para hacer una solicitud tipo GET. Recibe diversos parámetros, uno obligatirio, 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.

5.7.1.2 Respuesta en caso de éxito

De momento, veamos qué deberíamos hacer para especificarle a Angular lo que deve de 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.

5.7.2 Ejemplo completo

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.

Aunque sencilla, esta aplicación ya contene varias cosillas que para una mejor comprensión conviene ver por separado.

Este es nuestro HTML:

```
<div ng-app="apiApp" ng-controller="apiAppCtrl as vm">
  <h1>Pruebo Ajax</h1>
  <p>
      Selecciona:
      <select ng-model="vm.url" ng-change="vm.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 vm.paises">
        País: <span>{{pais.name}}</span>, capital: {{pais.capital}}
      </li>
  </ul>
</div>
```

Puedes fijarte que 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.

Aprecia que 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', ['$http', controladorPrincipal]);

function controladorPrincipal($http){
    var vm=this;

    vm.buscaEnRegion = function(){
        $http.get(vm.url).success(function(respuesta){
            //console.log("res:", respuesta);
            vm.paises = respuesta;
        });
    }
}
```

Creamos un module y luego un controlador al que inyectamos el $http como se explicó al inicio del artículo.

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.

Realmente el Ajax de este ejercicio se limita al siguiente código:

```
$http.get(vm.url).success(function(respuesta){
//console.log("res:", respuesta);
vm.paises = respuesta;
});
```

Usamos el shortcut $http.get() pasando como parámetro la URL, que sacamos del value que hay marcado en el campo SELECT del HTML. Luego se especifica una función para el caso "success" con el patrón "promise". Esa función devuelve en el parámetro "respuesta" aquello que nos entregó el API REST, que es un JSON con los datos de los países de una región. En nuestro ejemplo, en caso de éxito, simplemente volcamos en un dato del modelo, en el "scope", el contenido de la respuesta.

En concreto ves que la respuesta se vuelca en la variable vm.paises, que es justamente la colección por la que se itera en el ng-repeat de nuestro HTML.

Capítulo 6
MongoDB

6.1 Introducción básica

Las bases de datos relacionales están pasando de moda, los desarrolladores optan cada vez más por opciones novedosas de NoSQL debido a sus altos niveles de rendimiento y fácil escalabilidad. Hace unas semanas hablamos de las bondades de Redis; sin embargo, algunos andan temerosos por tener poco tiempo y prefieren una solución con un poco más de reputación, es por esto que esta semana hablaremos de la base de datos NoSQL más utilizada, MongoDB.

6.1.1 ¿Qué es MongoDB?

Es una base de datos NoSQL de código abierto, este tipo de soluciones se basan en el principio de almacenar los datos en una estructura tipo llave-valor; MongoDB por su lado se enfoca específicamente en que los valores de estas llaves (llamadas colecciones) son estructuras tipo JSON (llamados documentos), es decir objetos Javascript, lenguaje sobre el cual se basa esta solución de base de datos. Esto facilitará su manipulación a muchos que ya conozcan el lenguaje.

MongoDB posee varias estrategias de manejo de datos que la han posicionado donde se encuentra hoy en día, tales como sus procesos de división de datos en distintos equipos físicos o también conocido como clusterización, también el caso similar de documentos muy grandes que superen el límite estipulado de 16MB se aplica una estrategia llamada GridFS que automáticamente divide el documento en pedazos y los almacena por separado, al recuperar el documento el driver se encarga de armar automáticamente el documento nuevamente.

6.2 Instalación

Dirígete a la página de oficial de MongoDB y descarga el comprimido según la arquitectura de tu sistema operativo.

https://www.mongodb.org/downloads

Cabe destacar que a partir de la versión 2.2, MongoDB no es compatible con Windows XP.

Luego lo descomprimiremos y, según las recomendaciones, creamos un directorio mongodb en el directorio raíz C: donde colocaremos el contenido del comprimido, luego crearemos en C: un directorio data y dentro de este un directorio db, aquí será donde MongoDB almacenará la información de las bases de datos.

Para hacer que MongoDB funcione como un servicio primero crea el directorio log dentro de C:\mongodb\ y luego ejecutaremos el siguiente comando para crear el archivo de configuración asociado:

```
$ echo logpath=C:\mongodb\log\mongo.log > C:\mongodb\mongod.cfg
```

Luego instalemos el servicio:

```
$ C:\mongodb\bin\mongod.exe --config C:\mongodb\mongod.cfg --install
```

Ahora para gestionar el servicio de MongoDB basta con ejecutar:

```
$ net [ start | stop ] MongoDB
```

6.2.1 Configuración

Hablemos de algunas de las variables de mayor uso en este archivo.

port especificación del puerto donde escucha la base de datos.

bind_ip permite delimitar especificamente qué IPs pueden interactuar con la base de datos.

maxConns cantidad máxima de conexiones que serán aceptados, el valor por defecto depende de la cantidad de descriptores de archivos que maneja el sistema operativo.

objcheck habilitado por defecto, obliga a mongod a verificar cada petición para asegurar que la estructura de los documentos que insertan los clientes sea siempre válida. Cabe destacar que para documentos complejos esta opción puede afectar un poco el rendimiento.

fork inhabilitado por defecto, permite ejecutar mongod como un daemon.

auth inhabilitado por defecto, permite limitar el acceso remoto a la base de datos al implementar un mecanismo de autenticación.

dbpath especifica el directorio donde la instancia de base de datos almacena toda su información.

directoryperdb inhabilitado por defecto, ofrece la opción de que la información de cada base de datos presente en la instancia se almacene en carpetas separadas.

journal al habilitarse permite que las operaciones realizadas sobre la data sean almacenadas en una bitácora para en caso de ocurrir una falla, el sistema sea capaz de reconstruir la información que haya podido perderse.

smallfiles inhabilitado por defecto, ofrece la opción de que los archivos creados sean más pequeños y, por ende, más fáciles de entender, procesar y monitorear en varias ocasiones.

syncdelay especifica el lapso en segundos que tardará la instancia en pasar la información en la bitácora a persistencia.

También se ofrecen varias opciones para el uso de SSL, configuración de replicación y clusterización lo cual no tocaremos aquí.

6.2.2 Entrar a la consola

Bien ahora ya estamos listos para entrar a la base de datos y comenzar a jugar con ella. Para ello luego de tener el servicio de MongoDB corriendo, ejecutaremos el comando mongo y esto nos llevará a la consola interna de la instancia.

6.3 Operaciones básicas

Luego que hemos instalado MongoDB, comenzaremos a realizar inserciones y queries para probar las ventajas de esta solución y poner tus habilidades en práctica, comencemos con algunas operaciones básicas para saber cómo manipular los datos en MongoDB.

6.3.1 Creación de registros - .insert()

Las operaciones son como funciones de Javascript, así que llamaremos al objeto base de datos db y crearemos una nueva propiedad o lo que se asemejaría al concepto de tabla con el nombre de autores y le asociaremos su valor correspondiente (un objeto autor), es decir, una colección con un documento asociado:

```
> db.autores.insert({
  nombre    : 'Jonathan',
  apellido  : 'Wiesel',
  secciones : ['Como lo hago' , 'MongoDB']
});
```

Inclusive es posible declarar el documento como un objeto, almacenarlo en una variable y posteriormente insertarlo de la siguiente manera:

```
> var autorDelPost = {
  nombre    : 'Jonathan',
  apellido  : 'Wiesel',
  secciones : ['Como lo hago' , 'MongoDB']
};

> db.autores.insert(autorDelPost);
```

Ahora si ejecutamos el comando show collections podremos ver que se encuentra nuestra nueva colección de autores:

```
autores
...
```

Agreguemos un par de autores más:

```
> db.autores.insert({
  nombre    : 'Oscar',
  apellido  : 'Gonzalez',
  secciones : ['iOS' , 'Objective C' , 'NodeJS' ],
```

```
  socialAdmin : true
});
> db.autores.insert({
  nombre    : 'Alberto',
  apellido  : 'Grespan',
  secciones : 'Git',
  genero    : "M"
});
```

Veamos que insertamos nuevos documentos en la colección de autores que tienen otra estructura, en MongoDB esto es completamente posible y es una de sus ventajas.

6.3.2 Búsqueda de registros - .find()

Hagamos un query o una búsqueda de todos registros en la colección de autores.

```
> db.autores.find();


{ "_id" : ObjectId("5232344a2ad290346881464a"), "nombre" : "Jonathan",
"apellido" : "Wiesel", "secciones" : [  "Como lo hago",  "Noticias" ] }
{ "_id" : ObjectId("523236022ad290346881464b"), "nombre" : "Oscar",
"apellido" : "Gonzalez", "secciones" : [  "iOS",  "Objective C",  "NodeJS" ],
"socialAdmin" : true }
{ " id" : ObjectId("5232383a2ad290346881464c"), "nombre" : "Alberto",
"apellido" : "Grespan", "secciones" : "Git", "genero" : "M" }
```

```
SELECT * FROM autores
```

Notemos que la búsqueda nos arroja los objetos resultantes, en este caso los documentos de los 3 autores que insertamos acompañados del identificador único que crea MongoDB, este campo _id se toma además como índice por defecto.

6.3.2.1 Filtros

Digamos que ahora queremos hacer la búsqueda, pero filtrada por algún parámetro. Para esto sólo debemos pasar el filtro deseado a la función find(), busquemos a los administradores sociales para probar:

```
> db.autores.find({ socialAdmin: true });


{ " id" : ObjectId("523236022ad290346881464b"), "nombre" : "Oscar",
"apellido" : "Gonzalez", "secciones" : [  "iOS",  "Objective C",  "NodeJS" ],
"socialAdmin" : true }
```

```
SELECT * FROM autores WHERE socialAdmin = true
```

Probemos ahora filtrar por varias condiciones, primero probemos con filtros donde TODOS se deben cumplir:

```
> db.autores.find({ genero: 'M', secciones: 'Git' });


{ " id" : ObjectId("5232383a2ad290346881464c"), "nombre" : "Alberto",
"apellido" : "Grespan", "secciones" : "Git", "genero" : "M" }
```

```
SELECT * FROM autores WHERE genero = 'M' AND secciones = 'Git'
```

Veamos ahora un ejemplo un poco más avanzado de filtros con condiciones donde queremos que solo ALGUNA de ellas se cumpla:

```
> db.autores.find({
  $or: [
    {socialAdmin : true},
    {genero: 'M'}
  ]
 });


{ " id" : ObjectId("523236022ad290346881464b"), "nombre" : "Oscar",
"apellido" : "Gonzalez", "secciones" : [  "iOS",  "Objective C",  "NodeJS" ],
"socialAdmin" : true }
{ "_id" : ObjectId("5232383a2ad290346881464c"), "nombre" : "Alberto",
"apellido" : "Grespan", "secciones" : "Git", "genero" : "M" }
```

```
SELECT * FROM autores WHERE socialAdmin = true OR genero = 'M'
```

En este caso estamos filtrando por aquellos autores que son administradores sociales ó aquellos que tengan el campo género con el carácter M.

6.3.2.2 Limitar y Ordenar

Si quisiéramos limitar los resultados a un número máximo especificado de registros es tan fácil como agregar .limit(#) al final del comando .find():

```
> db.autores.find().limit(1)


{ " id" : ObjectId("5232344a2ad290346881464a"), "nombre" : "Jonathan",
"apellido" : "Wiesel",  "secciones" : [  "Como lo hago",  "MongoDB" ] }
```

```
SELECT * FROM autores LIMIT 1
```

La misma modalidad sigue la funcionalidad de ordenar los registros por un campo en particular, el cual servirá de argumento a la función .sort():

```
> db.autores.find().sort({apellido : 1})


{ "_id" : ObjectId("523236022ad290346881464b"), "nombre" : "Oscar",
"apellido" : "Gonzalez", "secciones" : [  "iOS",  "Objective C",  "NodeJS" ],
"socialAdmin" : true }
{ "_id" : ObjectId("5232383a2ad290346881464c"), "nombre" : "Alberto",
"apellido" : "Grespan", "secciones" : "Git", "genero" : "M" }
{ "_id" : ObjectId("5232344a2ad290346881464a"), "nombre" : "Jonathan",
"apellido" : "Wiesel", "secciones" : [  "Como lo hago",  "MongoDB" ] }
```

```
SELECT * FROM autores ORDER BY apellido DESC
```

También podemos combinar ambas funciones tan solo llamando una después de otra:

```
> db.autores.find().sort({apellido : 1}).limit(1)


{ "_id" : ObjectId("523236022ad290346881464b"), "nombre" : "Oscar",
"apellido" : "Gonzalez", "secciones" : [  "iOS",  "Objective C",  "NodeJS" ],
"socialAdmin" : true }
```

6.3.3 Eliminación de registros - .remove() drop()

Si entendiste como buscar registros pues eliminarlos es igual de fácil. Para esto existen 4 posibilidades:

- Eliminar los documentos de una colección que cumplan alguna condición.
- Eliminar todos los documentos de una colección.
- Eliminar la colección completa.

Probemos eliminandome a mí de la colección de autores:

```
> db.autores.remove({ nombre: 'Jonathan' });
```

```
DELETE FROM autores WHERE nombre = 'Jonathan'
```

¿Fácil no?. Eliminemos ahora a los demás autores:

```
> db.autores.remove();
```

```
DELETE FROM autores
```

Ahora que la colección ha quedado vacía deshagámonos de ella:

```
> db.autores.drop();
```

```
DROP TABLE autores
```

6.4 Actualizaciones y inserciones

Como mencionamos anteriormente, la parte de actualizaciones la dejamos por separado para tratar de ser un poco más detallados y extendernos en esta área para que puedas dominar con mayor destreza la manipulación de los datos.

6.4.1 Estructura

Para modificar los documentos que ya se encuentran almacenados usaremos el comando .update() el cual tiene una estructura como esta:

```
db.coleccion.update(
    filtro,
    cambio,
    {
        upsert: booleano,
        multi:  booleano
    }
);
```

Aclaremos un poco lo que nos indica la estructura.

filtro - debemos especificar como encontrar el registro que desemos modificar, sería el mismo tipo de filtro que usamos en las búsquedas o finders.

cambio - aquí especificamos los cambios que se deben hacer. Sin embargo ten en cuenta que hay 2 tipos de cambios que se pueden hacer:

Cambiar el documento completo por otro que especifiquemos.

Modificar nada más los campos especificados.

upsert (opcional, false por defecto) - este parametro nos permite especificar en su estado true que si el filtro no encuentra ningun resutlado entonces el cambio debe ser insertado como un nuevo registro.

multi (opcional, false por defecto) - en caso de que el filtro devuelva más de un resultado, si especificamos este parametro como true, el cambio se realizará a todos los resultados, de lo contrario solo se le hará al primero (al de menor Id).

6.4.2 Actualización sobrescrita (overwrite)

Bien, probemos insertando nuevo autor, el cual modificaremos luego:

```
> db.autores.insert({
    nombre      :   'Ricardo',
    apellido    :   'S'
});
```

Ahora probemos el primer caso, cambiar todo el documento, esto significa que en lugar de cambiar solo los campos que especifiquemos, el documento será sobreescrito con lo que indiquemos:

```
> db.autores.update(
    {nombre: 'Ricardo'},
    {
        nombre: 'Ricardo',
        apellido: 'Sampayo',
        secciones: ['Ruby','Rails'],
        esAmigo: false
    }
);
```

Notemos que como primer parámetro indicamos el filtro, en este caso que el nombre sea Ricardo, luego indicamos el cambio que hariamos, como estamos probando el primer caso indicamos el documento completo que queremos que sustituya al actual. Si hacemos db.autores.find({nombre:'Ricardo'}); podremos ver que en efecto el documento quedó como especificamos:

```
{ " id" : ObjectId("523c91f2299e6a9984280762"), "nombre" : "Ricardo",
"apellido" : "Sampayo", "secciones" : [  "Ruby",  "Rails" ], "esAmigo" :
false }
```

6.4.3 Operadores de Modificación

Ahora probemos cambiar los campos que deseamos, para este caso haremos uso de lo que se denominan como operadores de modificación.

Hablemos un poco sobre algunos de estos operadores antes de verlos en acción:

$inc - incrementa en una cantidad numerica especificada el valor del campo a en cuestión.

$rename - renombrar campos del documento.

$set - permite especificar los campos que van a ser modificados.

$unset - eliminar campos del documento.

Referentes a arreglos:

$pop - elimina el primer o último valor de un arreglo.

$pull - elimina los valores de un arreglo que cumplan con el filtro indicado.

$pullAll - elimina los valores especificados de un arreglo.

$push - agrega un elemento a un arreglo.

$addToSet - agrega elementos a un arreglo solo sí estos no existen ya.

$each - para ser usado en conjunto con $addToSet o $push para indicar varios elementos a ser agregados al arreglo.

Hagamos una prueba sobre nuestro nuevo documento:

```
> db.autores.update(
    { nombre: 'Ricardo' },
    {
        $set: { esAmigo: true , age : 25 }
    }
);
```

En este caso estamos usando el operador $set para 2 propositos a la vez:

- Actualizar el valor de un campo (cambiamos esAmigo de false a true).
- Creamos un campo nuevo (age) asignandole el valor 25.

Supongamos que Ricardo cumplió años en estos días, así que para incrementar el valor de su edad lo podemos hacer así:

```
> db.autores.update(
    { nombre: 'Ricardo' },
    {
        $inc: { age : 1 }
    }
```

```
);
```

Aquí hablamos español, así que cambiemos ese campo age por lo que le corresponde:

```
> db.autores.update(
    { nombre: 'Ricardo' },
    {
        $rename: { 'age' : 'edad' }
    }
);
```

Los que trabajamos en Codehero somos todos amigos así que no es necesario guardar el campo esAmigo:

```
> db.autores.update(
    { nombre: 'Ricardo' },
    {
        $unset: { esAmigo : '' }
    }
);
```

Pasemos ahora a la parte de modificación de arreglos, agreguemosle algunas secciones extra a nuestro autor:

```
> db.autores.update(
    { nombre: 'Ricardo' },
    {
        $push: { secciones : 'jQuery' }
    }
);
```

Esto agregará al final del arreglo de secciones el elemento jQuery.

Agreguemos algunas secciones más en un solo paso:

```
> db.autores.update(
    { nombre: 'Ricardo' },
    {
        $push: { secciones : { $each : ['Haskell','Go','ActionScript'] } }
    }
);
```

Bueno en realidad Ricardo no maneja desde hace un tiempo ActionScript así que eliminemos ese último elemento del arreglo:

```
> db.autores.update(
    { nombre: 'Ricardo' },
    {
        $pop: { secciones : 1 }
    }
);
```

Ricardo hace tiempo que no nos habla sobre jQuery así que hasta que no se reivindique quitémoslo de sus secciones:

```
> db.autores.update(
    { nombre: 'Ricardo' },
    {
        $pull: { secciones : 'jQuery' }
    }
);
```

Pensandolo bien, Ricardo nunca nos ha hablado de Haskell ni Go tampoco, eliminemoslos también:

```
> db.autores.update(
    { nombre: 'Ricardo' },
    {
        $pullAll: { secciones : ['Haskell','Go'] }
    }
);
```

6.4.4 Comando .save()

Otra manera para actualizar o insertar registros es mediante el uso del comando .save(). Este comando recibe como parámetro únicamente un documento.

Insertar un registro es tal cual como si hicieramos un .insert():

```
> db.autores.save({
    nombre:     'Ramses'
});
```

En cuanto al caso de actualización de registros te estarás preguntando:

En estos casos puedes hacer el equivalente a una actualización sobrescrita con tan solo indicar el _id del registro a actualizar como parte del nuevo documento.

```
> db.autores.find({nombre: 'Ramses'});

{ "_id" : ObjectId("5246049e7bc1a417cc91ec8c"), "nombre" : "Ramses" }

> db.autores.save({
    _id:        ObjectId('5246049e7bc1a417cc91ec8c')
    nombre:     'Ramses',
    apellido:   'Velasquez',
    secciones:  ['Laravel', 'PHP']
});
```

6.5 Modelado de Datos

Una de las dificultades que encuentran aquellos que se adentran al mundo del NoSQL es al tratar de transformar su esquema de base de datos relacional para que funcione de la mejor manera segun el enfoque NoSQL orientado a documentos, como lo es MongoDB. Aquí aprenderemos sobre como realizar correctamente el modelado de datos para que puedas comenzar a pensar en migrar tus proyectos a este tipo de base de datos con la menor dificultad posible.

6.5.1 Tipos de Datos

Al comienzo de la serie explicamos que los documentos de MongoDB son como objetos JSON, para ser especificos son de tipo BSON (JSON Binario), esta estrategia permite la serialización de documentos tipo JSON codificados binariamente. Veamos algunos de los tipos de datos que soporta:

String - Cadenas de caracteres.

Integer - Números enteros.

Double - Números con decimales.

Boolean - Booleanos verdaderos o falsos.

Date - Fechas.

Timestamp - Estampillas de tiempo.

Null - Valor nulo.

Array - Arreglos de otros tipos de dato.

Object - Otros documentos embebidos.

ObjectID - Identificadores únicos creados por MongoDB al crear documentos sin especificar valores para el campo _id.

Data Binaria - Punteros a archivos binarios.

Javascript - código y funciones Javascript.

6.5.2 Patrones de Modelado

Existen 2 patrones principales que nos ayudarán a establecer la estructura que tendrán los documentos para lograr relacionar datos que en una base de datos relacional estarían en diferentes tablas.

6.5.2.1 Embeber

Este patrón se enfoca en incrustar documentos uno dentro de otro con la finalidad de hacerlo parte del mismo registro y que la relación sea directa.

6.5.2.2 Referenciar

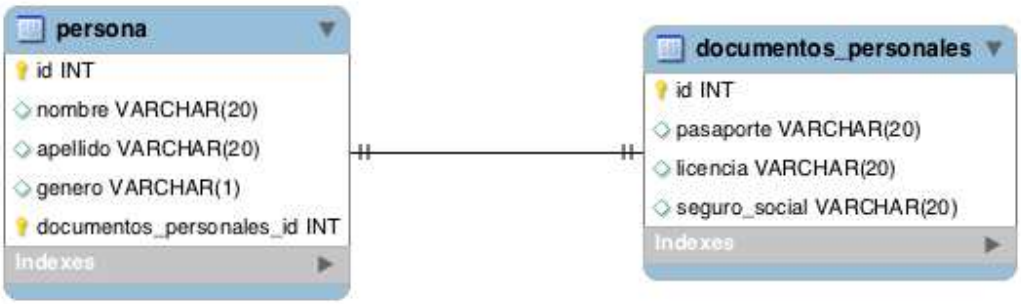
Este patrón busca imitar el comportamiento de las claves foráneas para relacionar datos que deben estar en colecciones diferentes.

6.5.3 Modelado de Relaciones

Bien, ha llegado el momento de aprender a transformar las relaciones de las tablas en las bases de datos relacionales. Empecemos con lo más básico.

6.5.3.1 Relaciones 1-1.

Muchas opiniones concuerdan que las relaciones 1 a 1 deben ser finalmente normalizadas para formar una única tabla; sin embargo, existen consideraciones especiales donde es mejor separar los datos en tablas diferentes. Supongamos el caso que tenemos una tabla persona y otra tabla documentos personales, donde una persona tiene un solo juego de documentos personales y que un juego de documentos personales solo puede pertenecer a una persona.



Si traducimos esto tal cual a lo que sabemos hasta ahora de MongoDB sería algo así:

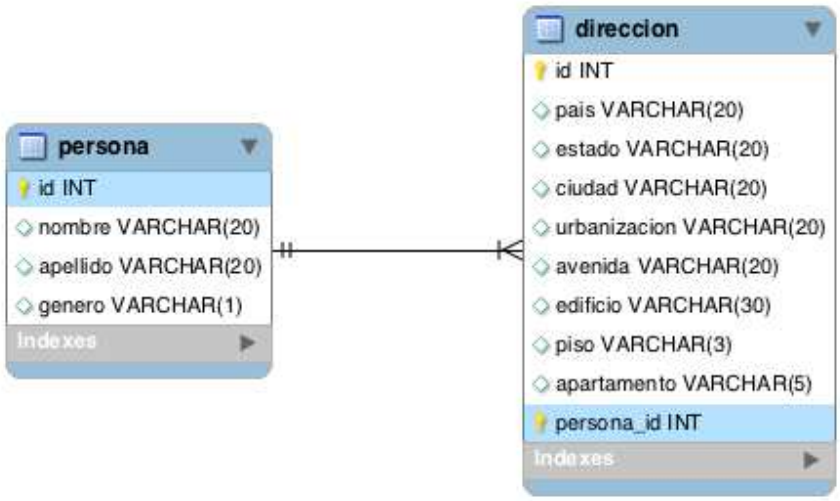
```
Persona = {
    nombre      :   'Jonathan',
    apellido    :   'Wiesel',
    genero      :   'M'
}

DocumentosPersonales = {
    pasaporte       :   'D123456V7',
    licencia        :   '34567651-2342',
    seguro_social   :   'V-543523452'
}
```

Para los casos de relaciones 1-a-1 se utiliza el patrón de embeber un documento en otro, por lo que el documento final quedaría así:

```
Persona = {
    nombre      :   'Jonathan',
    apellido    :   'Wiesel',
    genero      :   'M',
    documentos  :   {
        pasaporte       :   'D123456V7',
        licencia        :   '34567651-2342',
        seguro_social   :   'V-543523452'
    }
}
```

6.5.3.2 Relaciones 1-*

Supongamos ahora el caso de una tabla persona y otra tabla dirección. Donde una persona puede poseer varias direcciones.



Traduciendolo tal cual a MongoDB tendríamos algo así:

```
Persona = {
    nombre      :   'Jonathan',
    apellido    :   'Wiesel',
    genero      :   'M'
}

Direccion1 = {
    pais            :   'Venezuela',
    estado          :   'Distrito Capital'
    ciudad          :   'Caracas'
    urbanizacion    :   'La Florida',
    avenida         :   ...,
    edificio        :   ...,
    piso            :   ...,
    apartamento     :   ...
}

Direccion2 = {
    pais            :   'Estados Unidos',
    estado          :   'Florida'
    ciudad          :   'Miami'
    urbanizacion    :   'Aventura',
```

```
    avenida         :   ...,
    edificio        :   ...,
    piso            :   ...,
    apartamento     :   ...
}
```

Ahora para transformar la relación tenemos 2 opciones.

Podemos embeber las direcciones en el documento de la persona al establecer un arreglo de direcciones embebidas:

```
Persona = {
    nombre      :   'Jonathan',
    apellido    :   'Wiesel',
    genero      :   'M',
    direcciones :   [{
        pais            :   'Venezuela',
        estado          :   'Distrito capital'
        ciudad          :   'Caracas',
        urbanizacion    :   'La Florida',
        avenida         :   ...,
        edificio        :   ...,
        piso            :   ...,
        apartamento     :   ...
    },{
        pais            :   'Estados Unidos',
        estado          :   'Florida'
        ciudad          :   'Miami'
        urbanizacion    :   'Aventura',
        avenida         :   ...,
        edificio        :   ...,
        piso            :   ...,
        apartamento     :   ...
    }]
}
```

ó podemos dejarlo en documentos separados. Para esta segunda opción tenemos 2 enfoques.

Uno sería agregar un campo de referencia a dirección en persona:

```
Direccion1 = {
    _id             :   1,
    pais            :   'Venezuela',
    estado          :   'Distrito Capital',
    ciudad          :   'Caracas'
    urbanizacion    :   'La Florida',
    avenida         :   ...,
    edificio        :   ...,
    piso            :   ...,
    apartamento     :   ...
}

Direccion2 = {
    _id             :   2,
    pais            :   'Estados Unidos',
    estado          :   'Florida',
    ciudad          :   'Miami'
    urbanizacion    :   'Aventura',
    avenida         :   ...,
    edificio        :   ...,
    piso            :   ...,
    apartamento     :   ...
}

Persona = {
    nombre      :   'Jonathan',
    apellido    :   'Wiesel',
    genero      :   'M',
    direcciones :   [1,2]
}
```

y el otro sería agregar un campo de referencia a persona en dirección:

```
Direccion1 = {
    _id             :   1,
    pais            :   'Venezuela',
    estado          :   'Distrito Capital',
    ciudad          :   'Caracas'
```

```
    urbanizacion    :   'La Florida',
    avenida         :   ...,
    edificio        :   ...,
    piso            :   ...,
    apartamento     :   ...,
    persona_id      :   1
}

Direccion2 = {
    _id             :   2,
    pais            :   'Estados Unidos',
    estado          :   'Florida',
    ciudad          :   'Miami'
    urbanizacion    :   'Aventura',
    avenida         :   ...,
    edificio        :   ...,
    piso            :   ...,
    apartamento     :   ...,
    persona_id      :   1
}

Persona = {
    _id         :   1
    nombre      :   'Jonathan',
    apellido    :   'Wiesel',
    genero      :   'M'
}
```

En lo posible trata de utilizar la opción de embeber si los arreglos no variarán mucho ya que al realizar la búsqueda de la persona obtienes de una vez las direcciones, mientras que al trabajar con referencias tu aplicación debe manejar una lógica múltiples búsquedas para resolver las referencias, lo que sería el equivalente a los joins.

En caso de utilizar la segunda opción, ¿Cual de los 2 últimos enfoques utilizar?. En este caso debemos tomar en cuenta que tanto puede crecer la lista de direcciones, en caso que la tendencia sea a crecer mucho, para evitar arreglos mutantes y en constante crecimiento el segundo enfoque sería el más apropiado.

6.5.3.3 Relaciones *-*

Finalmente nos ponemos creativos a decir que, en efecto, varias personas pueden pertenecer a la misma dirección.

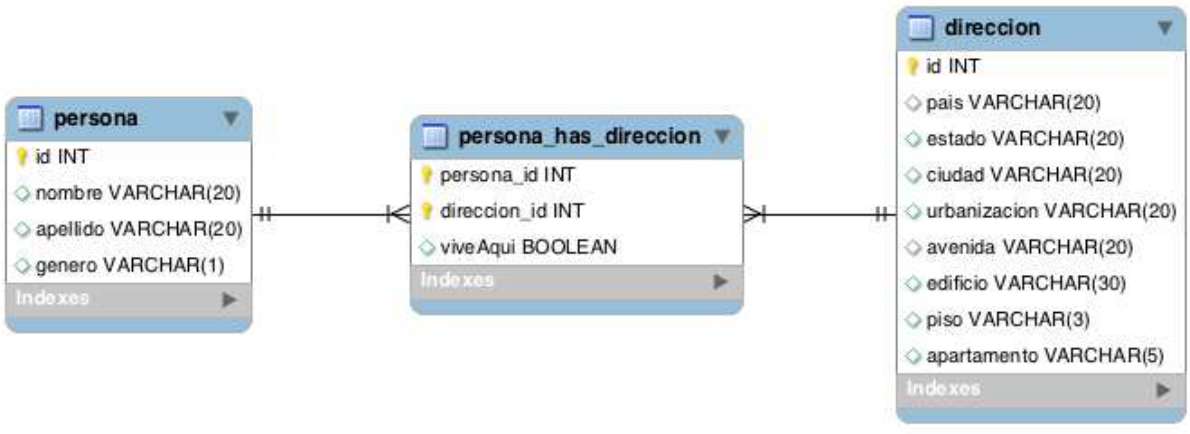
Al aplicar normalización quedaría algo así:

Para modelar este caso es muy similar al de relaciones uno a muchos con referencia por lo que colocaremos en ambos tipos de documento un arreglo de referencias al otro tipo. Agreguemos una persona adicional para demostrar mejor el punto:

```
Direccion1 = {
    _id             :   1,
    pais            :   'Venezuela',
    estado          :   'Distrito Capital',
    ciudad          :   'Caracas'
    urbanizacion    :   'La Florida',
    avenida         :   ...,
    edificio        :   ...,
    piso            :   ...,
    apartamento     :   ...,
    personas        :   [1000]
}

Direccion2 = {
    _id             :   2,
    pais            :   'Estados Unidos',
    estado          :   'Florida',
```

```
    ciudad          :   'Miami'
    urbanizacion    :   'Aventura',
    avenida         :   ...,
    edificio        :   ...,
    piso            :   ...,
    apartamento     :   ...,
    personas        :   [1000,1001]
}

Persona1 = {
    _id         :   1000,
    nombre      :   'Jonathan',
    apellido    :   'Wiesel',
    genero      :   'M',
    direcciones :   [1,2]
}

Persona2 = {
    _id         :   1001,
    nombre      :   'Carlos',
    apellido    :   'Cerqueira',
    genero      :   'M',
    direcciones :   [2]
}
```

Seguro debes estar esperando el caso más complejo de todos, aquellas ocasiones donde la tabla intermedia tiene campos adicionales.



Tomando como base el ejemplo anterior, agregaremos el campo adicional usando el patrón para embeber de la siguiente manera:

```
Direccion1 = {
    _id             :   1,
    pais            :   'Venezuela',
    estado          :   'Distrito Capital',
    ciudad          :   'Caracas'
    urbanizacion    :   'La Florida',
    avenida         :   ...,
    edificio        :   ...,
    piso            :   ...,
    apartamento     :   ...,
    personas        :   [1000]
}

Direccion2 = {
    _id             :   2,
    pais            :   'Estados Unidos',
    estado          :   'Florida',
    ciudad          :   'Miami'
    urbanizacion    :   'Aventura',
    avenida         :   ...,
    edificio        :   ...,
    piso            :   ...,
    apartamento     :   ...,
    personas        :   [1000,1001]
}

Persona1 = {
    _id         :   1000,
    nombre      :   'Jonathan',
    apellido    :   'Wiesel',
    genero      :   'M',
    direcciones :   [{
            direccion_id    :   1,
            viveAqui        :   true
```

```
        },{
            direccion_id    :   2,
            viveAqui        :   false
        }]
}

Persona2 = {
    _id         :   1001,
    nombre      :   'Carlos',
    apellido    :   'Cerqueira',
    genero      :   'M',
    direcciones :   [{
            direccion_id    :   2,
            viveAqui        :   true
        }]
}
```