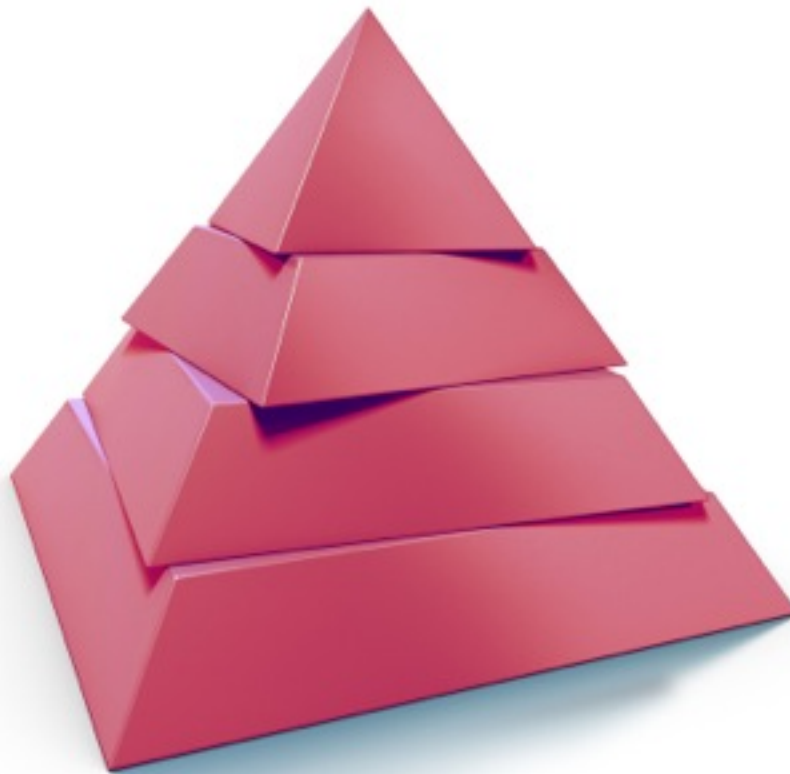




# **Arquitecturas Web con Angular.js**

*Cecilio Álvarez Caules*



[www.arquitecturajava.com](http://www.arquitecturajava.com)

# Autor



Cecilio Álvarez Caules es Oracle Enterprise Architect, Sun Certified Business Component Developer, Sun Certified Web Component Developer y Sun Certified Java Programmer. Es además Microsoft Certified Solution Developer, Microsoft Certified Enterprise Developer y Microsoft Certified Trainer. Trabaja como Arquitecto, Consultor y Trainer desde hace más de 15 años para distintas empresas del sector.

Puedes seguirme a través de las siguientes redes

LinkedIn

[https://www.linkedin.com/profile/view?id=2949192&trk=tab\\_pro](https://www.linkedin.com/profile/view?id=2949192&trk=tab_pro)

Twitter

<https://twitter.com/arquitectojava>

# Prologo

Este es el tercer libro que escribo sobre Arquitecturas y buenas prácticas . Los dos primeros fueron :

Arquitectura Java Sólida descargable desde <http://www.arquitecturajava.com> sin coste alguno

Arquitectura Java JPA Domain Driven Design ( editado en formato kindle)

Este tercer libro está orientado a Arquitecturas Web y concretamente a aprender a manejar y perder el miedo a uno de los framework JavaScript Web que están más de moda: **Angular.js**

Espero que el libro sirva de introducción y ayude a solventar las dudas sobre este framework tan interesante.

## Agradecimientos

Quiero aprovechar a agradecer a mi pareja y mi editora Olga Pelaez Tapia la paciencia que ha tenido al revisar un documento tan técnico. Por otro lado quiero agradecer también a Raúl Arabaolaza que me pusiera al tanto de la existencia de este framework hace ya más de un año y lo interesante que era comenzar a trabajar con él.

# Introducción, Angular.js vs jQuery

Angular.js es un framework Javascript relativamente nuevo y no es fácil manejarlo en un principio. Vamos a comenzar como siempre desde cero. Para ello construiremos, como punto de partida, un ejemplo utilizando jQuery para después ir evolucionando paso a paso hacia el mundo de Angular.js. Así pues, debemos descargar la última versión de jQuery de su página web:

<http://www.jquery.com>

## jQuery y Facturas

Vamos a trabajar **con el concepto de Factura (id,concepto,importe)** y, utilizando jQuery, procederemos a imprimir las diferentes propiedades por pantalla. Para ello usaremos el siguiente código:

```
<html>
<head>
<script type="text/javascript" src="jquery-1.11.1.js">
</script>
<script type="text/javascript">

$(document).ready(function() {

var factura= {"id":1,"concepto":"mac","importe":1000};

$("p:eq(0)").text(factura.id);
$("p:eq(1)").text(factura.concepto);
$("p:eq(2)").text(factura.importe);
```

```
});  
  
</script>  
</head>  
<body>  
<p><p>  
<p></p>  
<p></p>  
</body>  
</html>
```

En primer lugar, hemos construido una Factura como un objeto de Javascript:

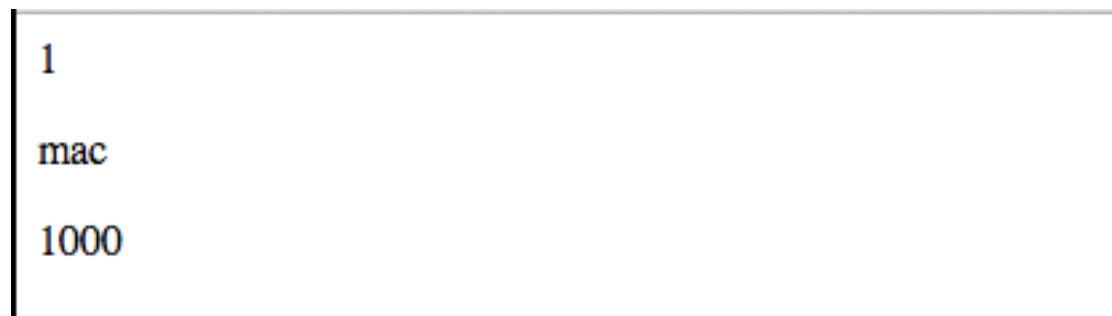
```
var factura= {"id":1,"concepto":"mac","importe":1000};
```

Después hemos utilizado selector de párrafo “p” más el selector de posición “eq” de jQuery para ir dibujando las diferentes propiedades de la Factura en el primer, segundo y tercer párrafo:

```
$("p:eq(0)").text(factura.id);  
$("p:eq(1)").text(factura.concepto);  
$("p:eq(2)").text(factura.importe);
```

De esta forma, jQuery nos imprime el contenido de la Factura

---



```
1  
mac  
1000
```

No hay mucho más que reseñar, el código funciona correctamente y simplifica significativamente el uso puro de Javascript y DOM que tendríamos que haber usado para realizar la misma operación.

## Añadir Botones

Vamos a realizar algunos cambios que en principio nos parecen poco relevantes, de tal forma que tengamos diferentes botones para cambiar cada una de las distintas propiedades de la Factura.

```
<p>
<input type="button" id="botonId" value="cambiarID"/>
</p>
<p>
<input type="button" id="botonConcepto" value="cambiarConcepto"/>
</p>
<p>
<input type="button" id="botonImporte" value="cambiarImporte"/>
</p>
```

El resultado será el siguiente :

1

mac

1000

cambiarID

cambiarConcepto

cambiarImporte

No hemos hecho nada drástico. Ahora añadiremos código a nuestro ejemplo para que cada botón responda de forma correcta al evento de click:

```
$("#botonId").click(function() {  
  
    factura.id=2;  
    $("p:eq(0)").text(factura.id);  
  
})  
  
$("#botonConcepto").click(function() {  
  
    factura.concepto="iphone"  
    $("p:eq(1)").text(factura.concepto);  
  
});  
  
$("#botonImporte").click(function() {  
  
    factura.importe=700;  
    $("p:eq(2)").text(factura.importe);  
  
});
```

Si pulsamos cada uno de los botones, estaremos cambiando tanto los datos que la Factura almacena como los datos que se presentan en los párrafos utilizando el método **.text()** de **jQuery**.

2

iphone

700

cambiarID

cambiarConcepto

cambiarImporte

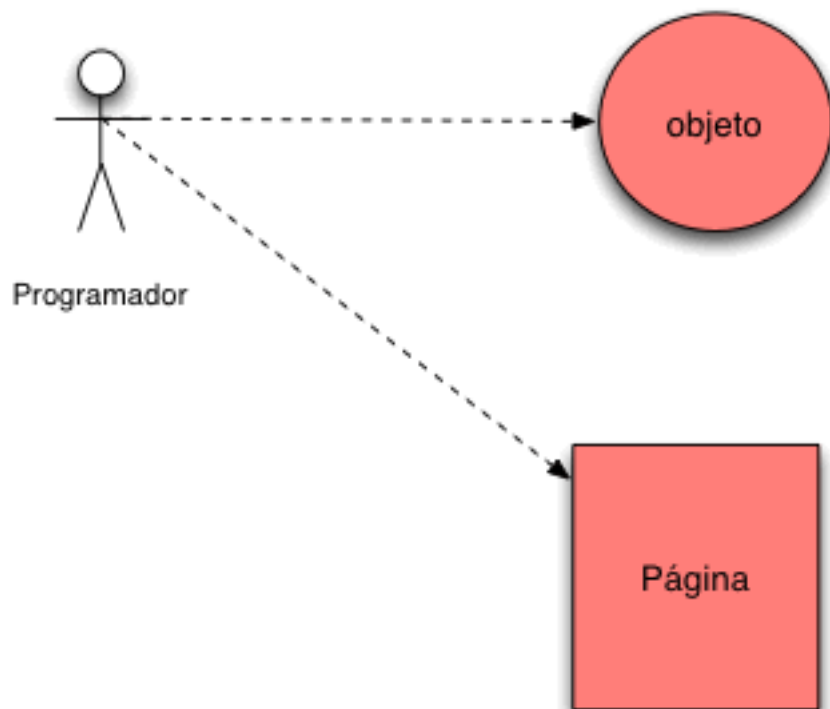
## Problemas con JQuery

Todo parece normal en el código, pero si lo analizamos un poco más detalladamente, nos podemos dar cuenta de que resulta repetitivo cambiar las propiedades del objeto y a continuación cambiar al mismo tiempo el contenido del párrafo:

```
factura.importe=700;
```

```
$("p:eq(2)").text(factura.importe);
```

Con jQuery nos estamos viendo obligados a cambiar tanto el objeto como la capa de presentación (párrafos) de la página.





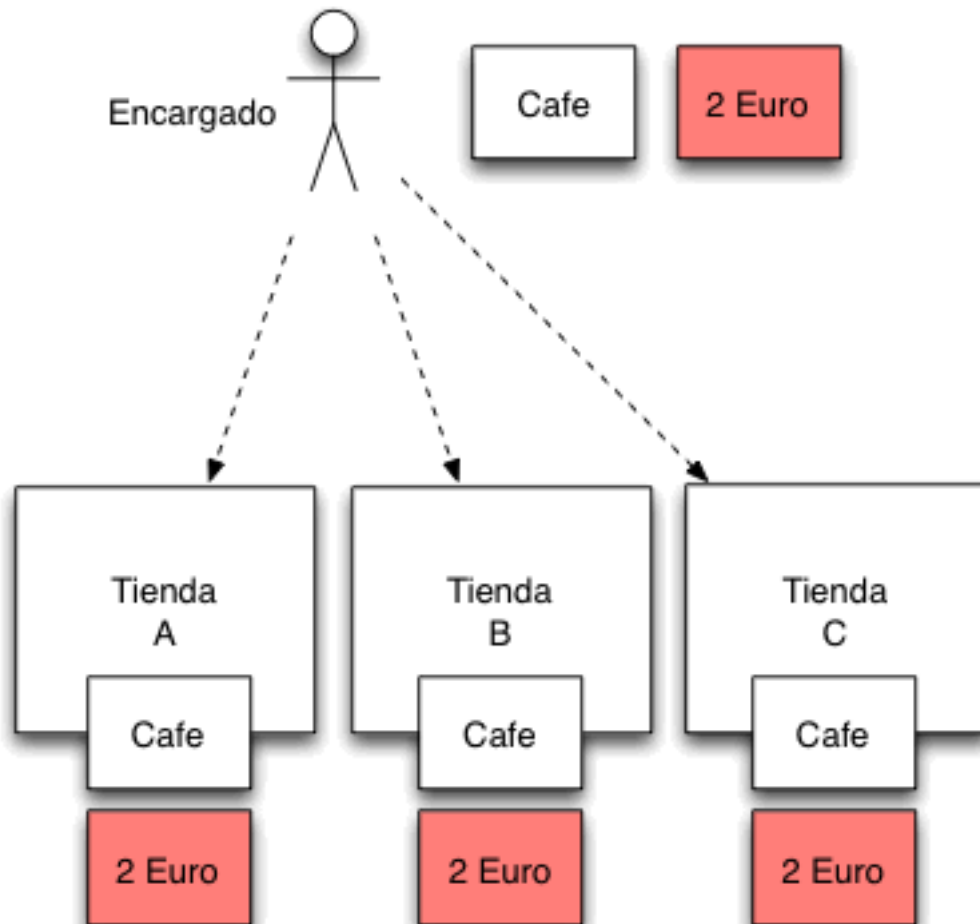
Estamos trabajando el doble. ¿**Quiere decir esto que JQuery está mal diseñado?** La respuesta es **NO**. jQuery es un framework que está orientado a gestionar la capa de presentación y es algo que hace de forma correcta, simplificando claramente el trabajo con ella. Sin embargo, no es capaz de cubrir otro tipo de problemas como con el que nos encontramos ahora.

## Asignación de Responsabilidades

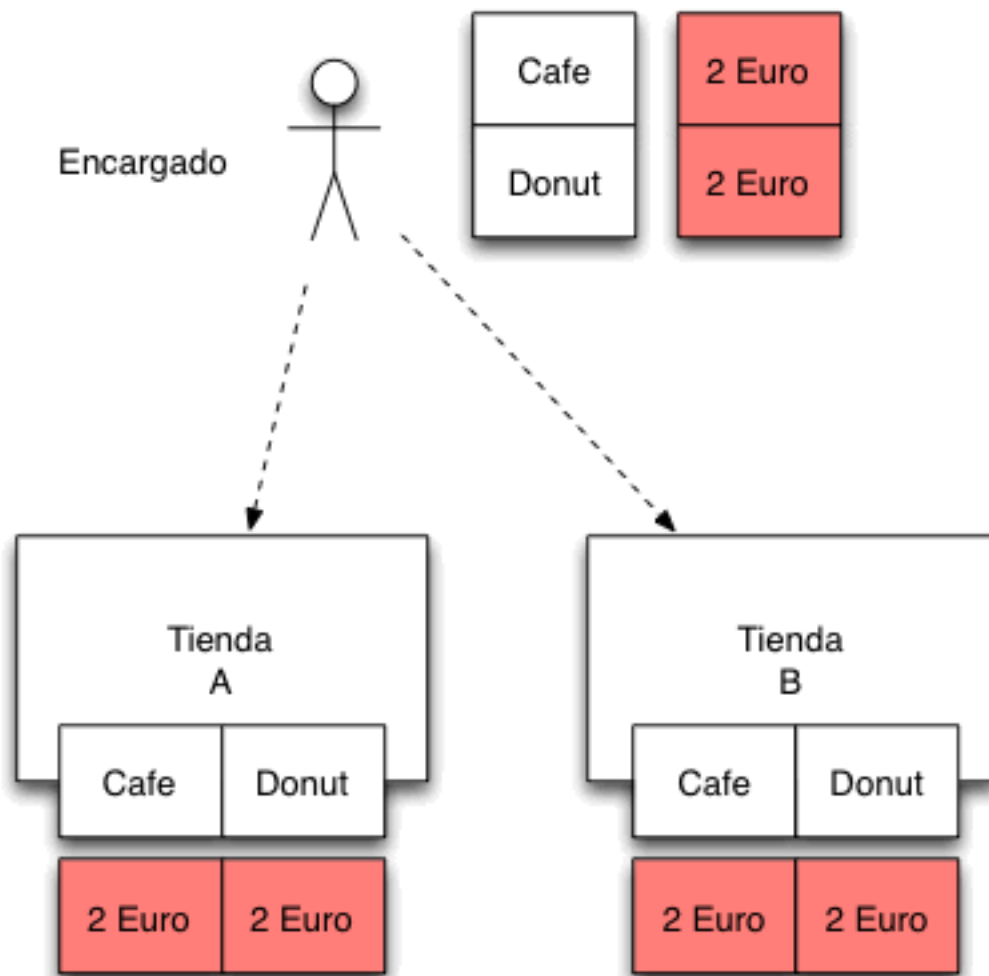
En estos momentos tenemos un problema serio de asignación de responsabilidades pero no es fácil de detectar. Vamos a poner un ejemplo similar basado en la vida cotidiana. Imaginemos una cafetería donde hay un encargado que pone precio a los cafés que vendemos.



Resulta que el negocio nos ha ido bien y hemos abierto otras 3 cafeterías. Así que cuando nuestro encargado cambie el precio del café, deberá ir a cada una de las cafeterías y cambiar el precio en ellas también.

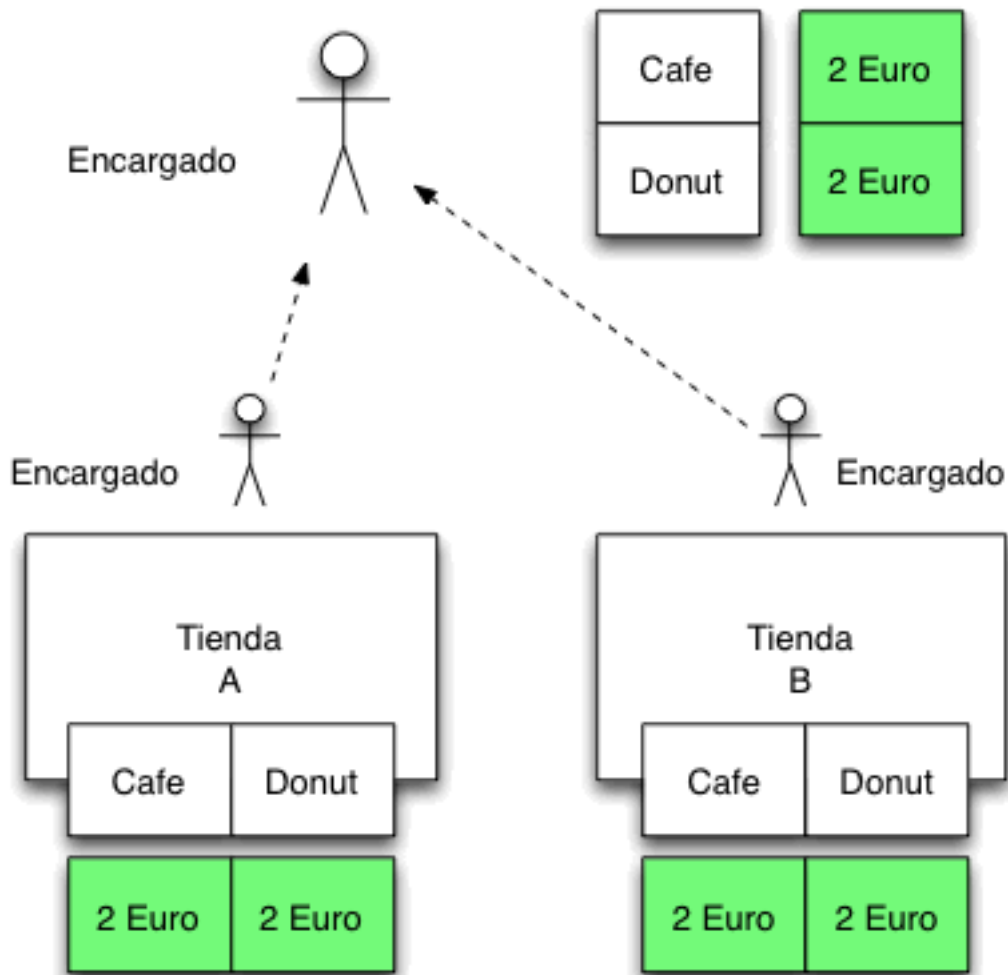


Ahora bien ¿Qué sucederá cuando el número de cafeterías o el número de productos aumente?. Nuestro encargado no podrá asumir de forma eficiente el cambio de los precios de todos los productos.



## División de Responsabilidades

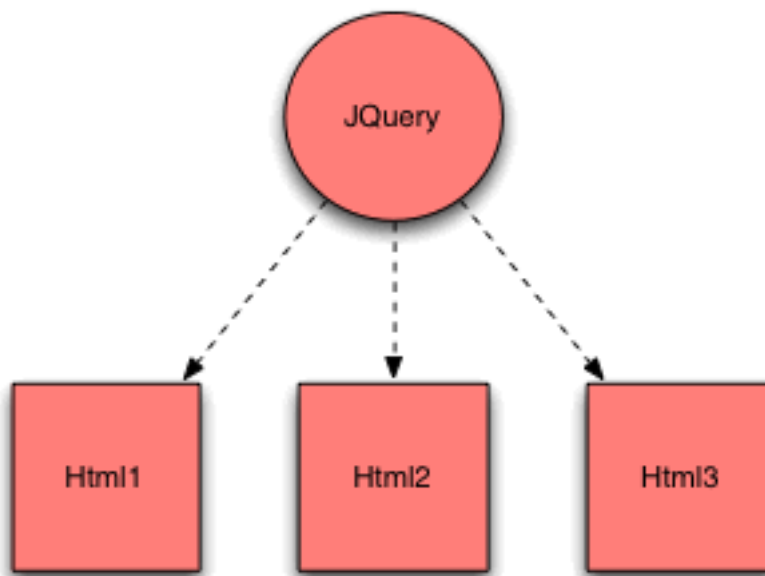
Es fácil ver que nuestro Encargado esta asumiendo demasiadas responsabilidades y no puede realizar todas de una forma eficaz. ¿Cuál puede ser una solución razonable y escalable? . En este caso podemos dividir las responsabilidades y contratar un encargado en cada una de las cafeterías de tal forma que sean estos encargados los que cambien los precios en cada una de ellas, basándose en los precios que les transmite el Encargado principal.



Este es un ejemplo elemental de uno de los patrones más conocidos: **El patrón observador/observable** ya que cada uno de los empleados de las cafeterías observan los cambios que realiza el encargado principal.

## jQuery y Responsabilidad

En el ejemplo que acabamos de construir con JQuery estamos ante un problema de división de responsabilidades ya que jQuery se encarga de todo y tenemos las responsabilidades demasiado centralizadas. Esto finalmente genera más trabajo



## Angular.js y Responsabilidades

Angular.js es un Framework de Javascript que está enfocado hacia la división de responsabilidades. Vamos a ir introduciéndolo paulatinamente en los siguientes capítulos.

# Introducción a Angular.js

Acabamos de ver los problemas que tiene trabajar con un framework como jQuery en cuanto a la evolución de la aplicación se refiere. Es hora de introducir Angular.js como alternativa. Para ello, lo primero que haremos será descargarnos de su página web la última versión de angular :

<https://www.angularjs.org/>

Descargado el framework, vamos a crear el mismo ejemplo de jQuery con Angular.js :

```
<html ng-app>
<head>
<script type="text/javascript" src="angular.min.js">
</script>
<script type="text/javascript">

function controladorFactura($scope) {

$scope.factura = {"id":1,"concepto":"mac","importe":1000};

}

</script>
</head>
<body ng-controller="controladorFactura">
<p>{{ factura.id }}</p>
<p>{{ factura.concepto }}</p>
<p>{{ factura.importe }}</p>
</body>
</html>
```

En un principio parece incomprensible, es normal dado que el paradigma de programación de Angular.js es muy diferente al de jQuery. Vamos a ir explicando el código poco a poco.

## Angular y Directivas

Lo primero que podemos ver es que el fichero HTML tiene un atributo **ng-app** a nivel de la etiqueta `<html>`

**`<html ng-app>`**

A este tipo de atributos se les conoce en el mundo de Angular como **directivas**: es el concepto fundamental de todo el framework. Cada directiva tiene una funcionalidad diferente. En nuestro caso la directiva `ng-app` se encarga de definir los límites de la aplicación Angular e inicializarla. En este caso, la aplicación Angular ocupa todo el documento HTML.



Una vez limitado el alcance de la aplicación e introducido el concepto de directiva, es el momento de avanzar.

## Angular y Controladores

La siguiente directiva encontrada es la directiva ng-controller, que afecta al body, su código es :

```
<body ng-controller="controladorFactura">
```

Esta directiva tiene varias funcionalidades . La primera y más importante es la que define el concepto de Controlador. Un Controlador es un objeto que está ligado con una vista.



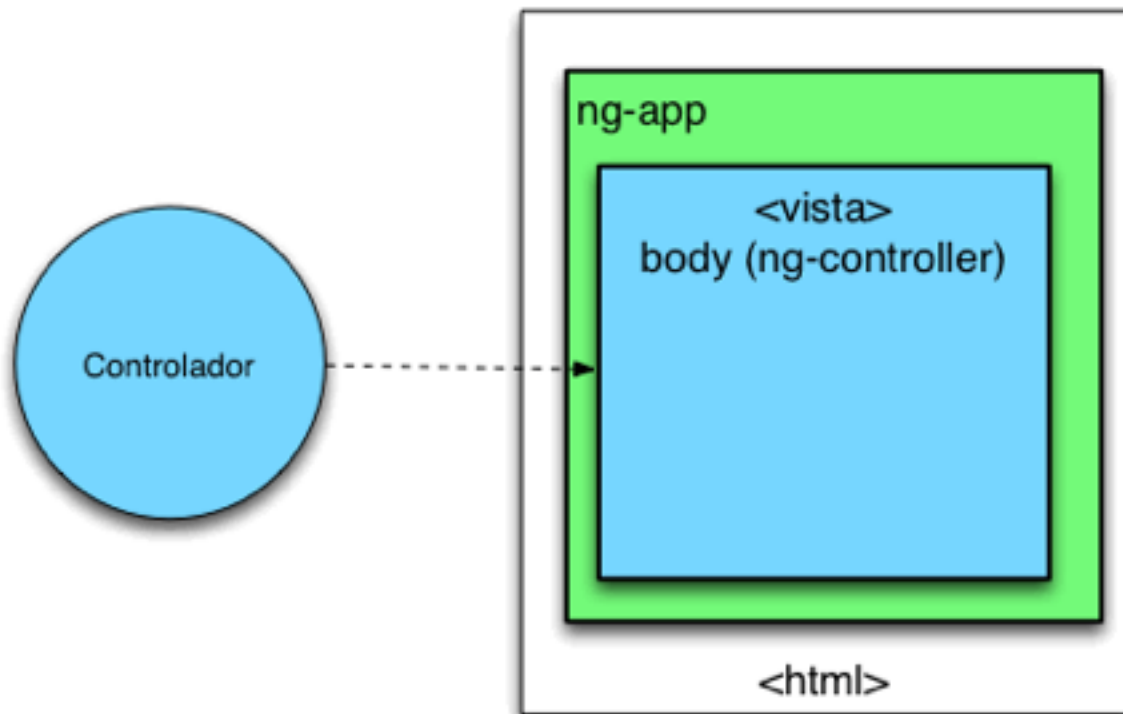
## Angular y Vistas

Una vista es un bloque de código HTML. En Angular, un controlador está relacionado con el bloque de código HTML, que se encuentra dentro de la etiqueta en la que lo hemos declarado. En nuestro caso el controlador ha sido declarado a nivel de <body>.

```
<body ng-controller="controladorFactura">
```



Por lo tanto, la vista a la que el controlador está asociado es todo el `<body>` de nuestro documento.



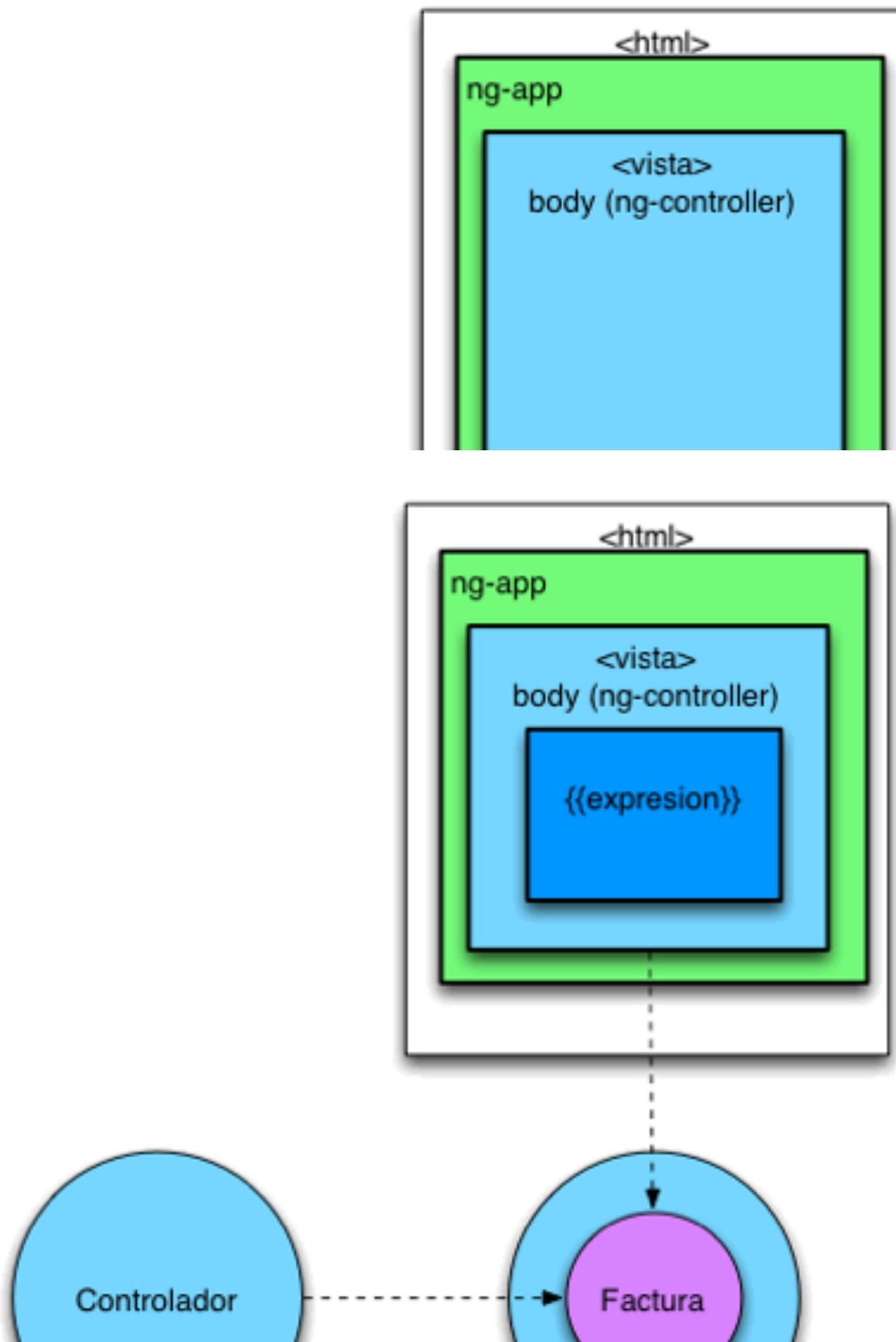
## Angular Controladores y Javascript

Ya sabemos que un Controlador está relacionado con una Vista. Para gestionar esta relación, debemos declarar una función de Javascript con el mismo nombre del controlador. A través de esta función nos comunicaremos con la vista.

```
function controladorFactura($scope) {  
}
```

## Angular y \$scope

La función de JavaScript es muy sencilla, sin embargo puede generar dudas inmediatas, ya que dispone de una variable denominada \$scope. **¿Qué es \$scope?**  
**:Es una variable que funciona como un contenedor compartido por el Controlador y la Vista.**



Así pues, si queremos asignar datos desde el controlador a la Vista, no tenemos mas que insertarlos en la variable \$scope. Automáticamente, la vista se percatará de los cambios que el \$scope ha sufrido y se actualizará.

```
function controladorFactura($scope) {  
  
    $scope.factura = {"id":1,"concepto":"mac","importe":1000};  
  
}
```

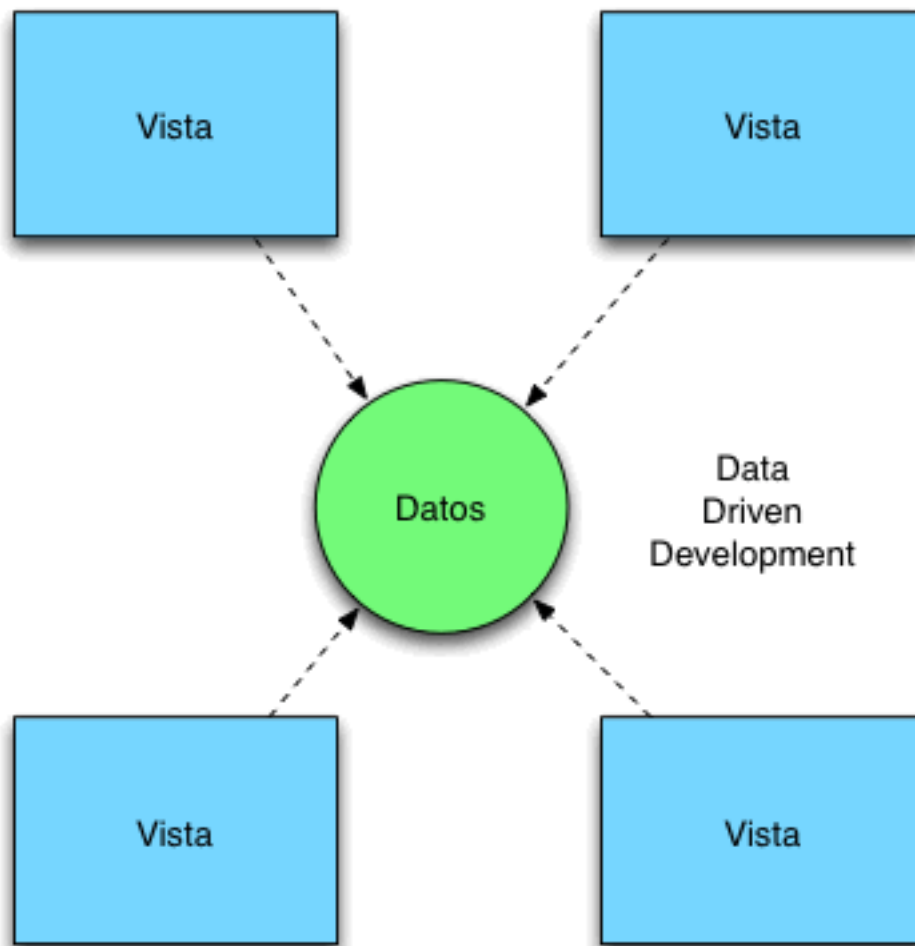
## Angular y Expresiones

Para que la vista puede mostrar la información almacenada en el \$scope usaremos **una expresión de Angular**. Las expresiones son sencillos bloques de código que van entre {{ }} y que Angular es capaz de interpretar.

Vamos a verlo en código :

```
<body ng-controller="controladorFactura">  
<p>{{factura.id}}</p>  
<p>{{factura.concepto}}</p>  
<p>{{factura.importe}}</p>  
</body>
```

De esta forma, la vista quedará completamente configurada para mostrar la información de nuestra factura que se encuentra almacenada en el \$scope así como los cambios que en el \$scope se produzcan. A diferencia del mundo de jQuery, en las que las vistas no tienen responsabilidad ninguna, en este caso tienen la responsabilidad de actualizar sus propios datos.



Así pues dará lo mismo las veces que cambiemos los datos que se encuentran ubicados en el `$scope`: las vistas asumirán su responsabilidad y cambiarán la información que se presenta.

2

iphone

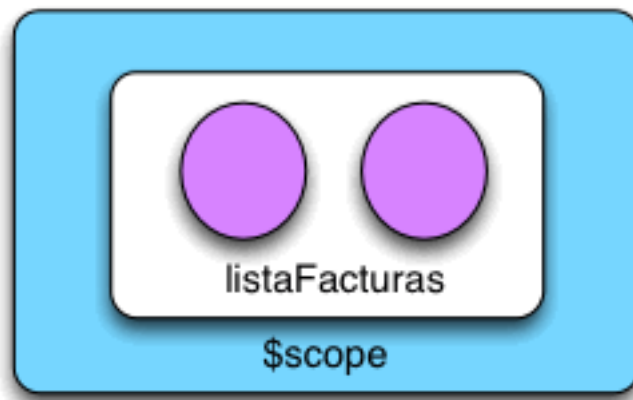
700

## Data Driven Design

Esta nueva forma de programar aplicaciones web se esta comenzando a denominar Data Driven Design o Data Driven Development, ya que nos encontramos ante una arquitectura que se centra en gestionar los datos y después, a través de observadores, actualizarlos de forma transparente en la vista que corresponda.

# Angular y Colecciones

En el capítulo anterior hemos terminado de construir nuestro primer ejemplo con Angular.js y el concepto de Factura. A través de este ejemplo hemos explicado los conceptos fundamentales. Es momento de continuar y ver cómo trabajar con una colección o lista de Facturas.

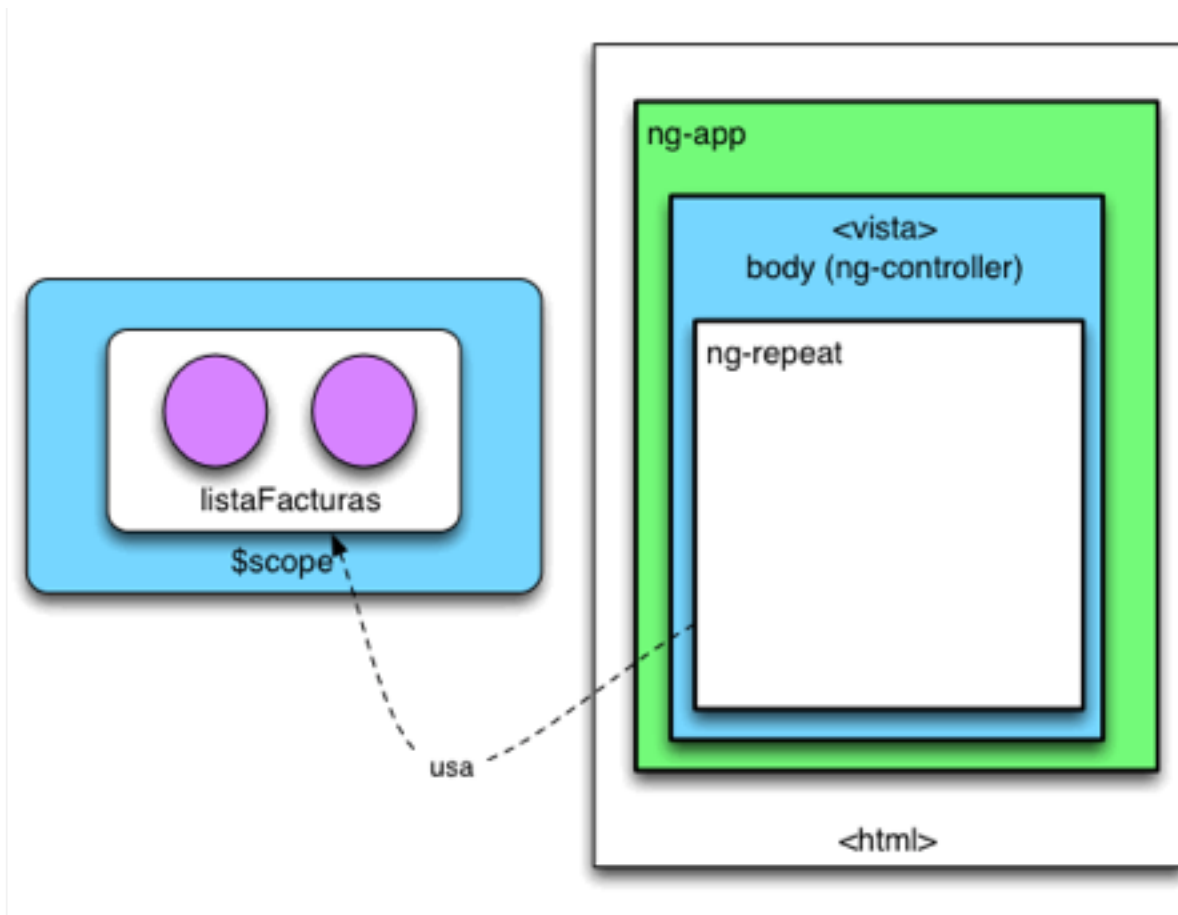


Para ello, lo primero que vamos a hacer es generar un Array en Javascript al que denominaremos “listaFacturas” y añadir otra Factura más para tener dos. Hecho esto, añadiremos la lista de Facturas al \$scope de nuestro controlador. Vamos a verlo detalladamente:

```
function controladorPersona($scope) {  
  
    var factura= {"id":"A","concepto":"mac","importe":1000};  
    var  factura2= {"id":"B","concepto":"Iphone","importe":700};  
    var listaFacturas=[];  
    listaFacturas.push(factura);  
    listaFacturas.push(factura2);  
    $scope.listaFacturas=listaFacturas;  
}
```

## ng-repeat y listaFacturas

Es momento de modificar nuestra **Vista** para que sea capaz de presentarnos una lista de Facturas. Para ello nos vamos a apoyar en la directiva **ng-repeat** encargada de mostrar una lista de objetos que se encuentren almacenados en el `$scope`.



Para ello construiremos una tabla y usaremos la directiva `ng-repeat` a nivel de `<tr>`

```
<table>
<tr ng-repeat="factura in listaFacturas">
  <td>{{factura.id}}</td>
  <td>{{factura.concepto}}</td>
  <td>{{factura.importe}}</td>
</tr>
```

</table>

Realizada esta operación, las dos facturas que tenemos en la lista se muestran en la pantalla.

---

A mac	1000
B Iphone	700

ng-click y nuevas facturas.

El siguiente paso es añadir un nuevo botón a la Vista de tal forma que podamos añadir nuevas facturas al Array que tenemos. Para ello nos apoyaremos en la directiva ng-click.

```
<body ng-controller="controladorFacturas">
<table>
<tr ng-repeat="factura in listaFacturas">
<td>{{ factura.id }}</td>
<td>{{ factura.concepto }}</td>
<td>{{ factura.importe }}</td>
</tr>
</table>
<input type="button" ng-click="addFactura()" value="nuevo"/>
</body>
```

Como vemos, hemos usado la directiva “ng-click” que se encargará de recoger el evento de pulsar el botón. Para gestionar este evento, modificaremos el \$scope de nuestro controlador y añadiremos la función addFactura().



```
$scope.addFactura=function() {

$scope.listaFacturas.push({"id":"C","concepto":"Nexus","importe":300});
}
```

Esta función usará el método push de la clase Array para añadir un nuevo elemento a nuestra lista. Hecho esto, la lista contendrá un nuevo registro.

---

1 A mac 1000

2 B Iphone 700

3 C Nexus 300

nuevo

## Reglas de negocio

Frecuentemente debemos implementar reglas de negocio en nuestra aplicación. En este ejemplo vamos a añadir una regla de negocio sencilla, la capacidad de calcular el importe con IVA de la Factura. Para ello, de entrada nos vamos a apoyar en \$scope y añadirle una nueva función.

```
$scope.calcularIva=function(factura) {

    return factura.importe *1.21;
}
```

Hecha esta operación, podremos modificar la vista y añadir una nueva columna que nos presente el precio con IVA incluido.

```
<tr ng-repeat="factura in listaFacturas">
<td>{{$index+1}}</td>
<td>{{factura.id}}</td>
<td>{{factura.concepto}}</td>
<td>{{factura.importe}}</td>
```

```
<td>{{calcularIva(factura)}}</td>  
</tr>
```

La nueva tabla será la encargada de calcular el precio con IVA:

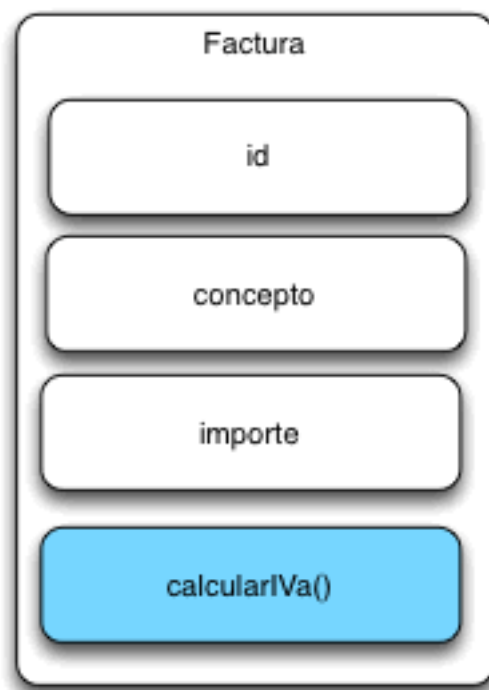
---

1	A	mac	1000	1210
2	B	Iphone	700	847

Lamentablemente, no es la mejor opción ya que pueden existir muchos controladores que quieran calcular el IVA de una Factura concreta.

## Reglas de negocio y Business Objects

Para solventar este problema, nos vamos a apoyar en los ejemplos habituales de programación orientada a objeto y vamos a crear una clase que contenga la función y podamos así reutilizarla. Esto es algo para lo cual Angular no tiene gran soporte.



Vamos a ver su código :

```
function Factura(json) {  
  
    this.id=json.id;  
    this.concepto=json.concepto;  
    this.importe=json.importe  
  
    this.calcularIva= function () {  
  
        return this.importe*1.21;  
    }  
  
}
```

Creada esta clase, podemos hacer uso de ella a la hora de crear nuestros objetos, añadirlos a la lista y asignarlos al \$scope.

```
var facturaDatos= {"id":"A","concepto":"mac","importe":1000};  
var facturaDatos2= {"id":"B","concepto":"Iphone","importe":700};  
var listaFacturas=[];  
listaFacturas.push(new Factura(facturaDatos));  
listaFacturas.push(new Factura(facturaDatos2));  
$scope.listaFacturas=listaFacturas;
```

Construyendo los objetos de esta forma podremos hacer uso de la nueva función que hemos creado a nivel de la Vista.

```
<tr ng-repeat="factura in listaFacturas">  
  <td>{{factura.id}}</td>  
  <td>{{factura.concepto}}</td>  
  <td>{{factura.importe}}</td>  
  <td>{{factura.calcularIva()}}</td>  
</tr>
```

El resultado será el mismo:

---

1	A mac	1000	1210
2	B Iphone	700	847

## Borrar Facturas

Hemos terminado de construir las dos operaciones más básicas: listar Facturas y añadir nuevas Facturas. Vamos a definir la operación de borrado basándonos en lo que hemos aprendido. En primer lugar, vamos a añadir una nueva columna a la tabla que nos permita disponer de un botón de borrado al que asociaremos una función.

```
<td>  
<input type="button" ng-click="removeFactura(factura)"  
value="Borrar"/>  
</td>
```

Esto nos cambiará la forma en la que la tabla se muestra en algo semejante a lo siguiente:

---

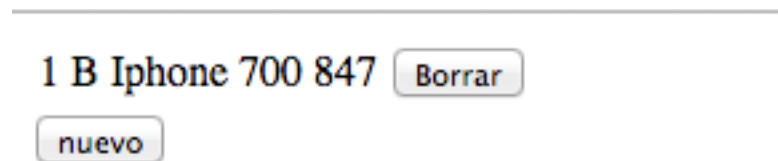
1	B Iphone	700	847	Borrar
nuevo				

Implementada esta primera parte, añadimos una nueva función a nuestro \$scope que se encargue de eliminar un objeto de la lista. Para ello nos vamos a apoyar en la función filter que tienen los Arrays de Javascript y que permiten realizar un filtrado a partir de una condición inicial.

```
$scope.removeFactura=function(factura) {
```

```
$scope.listaFacturas=$scope.listaFacturas.filter(function (f) {  
    return f!==factura;  
})  
}
```

En este caso hemos definido un filtro que cumplirán todas las Facturas excepto la situada en la fila que acabamos de pulsar. Por lo tanto, la nueva lista de Facturas únicamente contendrá una única Factura.



## ng-model y formularios

Hasta este momento hemos trabajado con Angular.js añadiendo nuevas facturas de forma automática. Pero esto no es realmente lo que queremos, sino que nos vendría bien tener un formulario en el cuál rellenar los distintos campos de la Factura. Vamos a modificar nuestro HTML para añadirlo. Cada campo dispondrá de una directiva ng-model.

```
<form>  
<p>  
Id:<input type="text" ng-model="nuevaFactura.id"/>  
</p>  
  
<p>  
Concepto:<input type="text" ng-model="nuevaFactura.concepto"/>  
</p>  
<p>  
Importe:<input type="text" ng-model="nuevaFactura.importe"/>  
</p>
```

```
<input type="button" ng-click="addFactura()" value="nuevo"/>
</form>
```

Realizada esta primera modificación, la página HTML mostrará un nuevo formulario.

---

A mac	1000 1210	<input type="button" value="Borrar"/>
B Iphone	700 847	<input type="button" value="Borrar"/>

Id:

Concepto:

Importe:

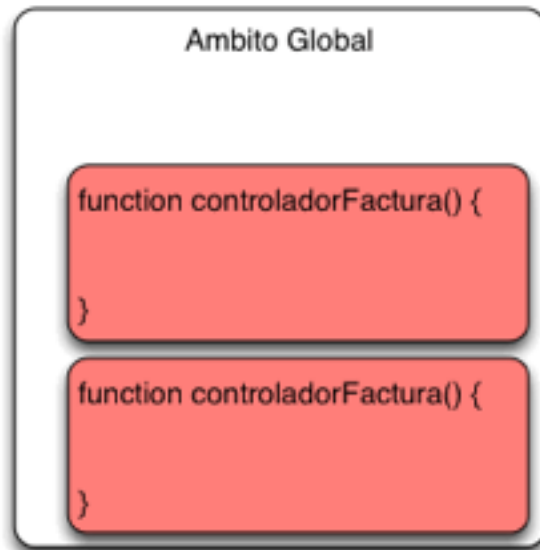
El uso de la directiva ng-model nos creará un nuevo objeto en el \$scope que se denomina nuevaFactura y que podemos añadir a la lista a través de la función addFactura():

```
$scope.addFactura=function() {
    $scope.listaFacturas.push(new Factura($scope.nuevaFactura));
}
```

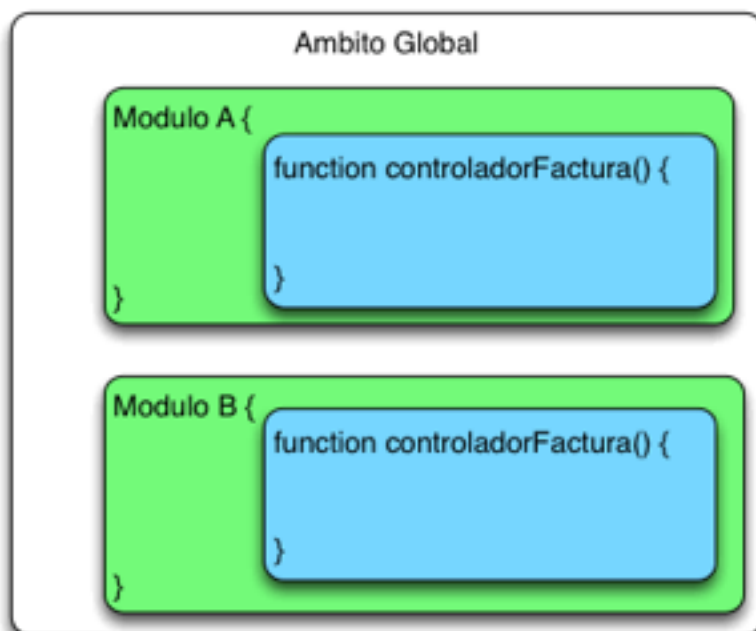
## Angular, Visibilidad y Modulos

Ya está la aplicación funcionando, sin embargo tenemos un problema y es que nuestra función está definida a nivel global del código de Javascript. Esto genera un

problema de diseño, ya que cuando la aplicación crezca de tamaño, nos podemos encontrar con funciones que tengan el mismo nombre en el ámbito global.



Para evitar este problema y permitir que la aplicación pueda crecer de tamaño vamos a incluir todo nuestro código en un módulo que es un patrón de diseño a nivel de Javascript de tal forma que cada módulo tenga sus propias funciones y nos aísle del resto.



Vamos a ver cómo queda en código :

```
angular.module('moduloFacturas', []).  
controller("controladorFacturas",function($scope) {  
  
var facturaDatos= {"id":"A","concepto":"mac","importe":1000};  
var facturaDatos2= {"id":"B","concepto":"Iphone","importe":700};  
var listaFacturas=[];  
listaFacturas.push(new Factura(facturaDatos));  
listaFacturas.push(new Factura(facturaDatos2));  
$scope.listaFacturas=listaFacturas;  
//resto de código  
});
```

Una vez realizada esta operación, indicamos a Angular.js en qué módulos se apoya la aplicación

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

Hemos terminado de gestionar las operaciones básicas sobre nuestra tabla utilizando Angular. Sin embargo nos quedan muchas tareas que realizar, ya que en estos momentos tenemos todos los datos almacenados en el Cliente. Es momento de comenzar a trabajar con un servidor.

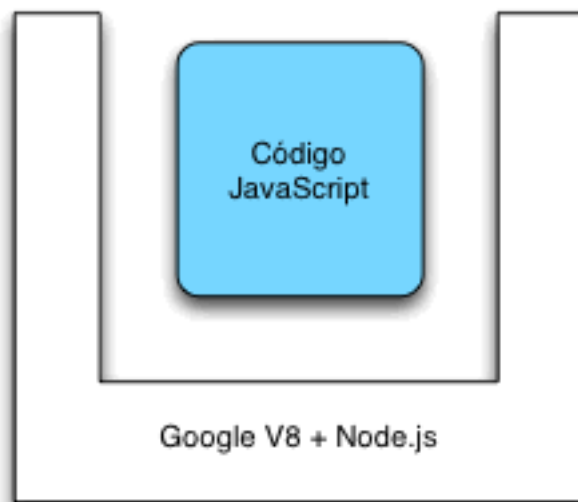


# Angular.js y Node.js

Por ahora la aplicación que hemos desarrollado nos funciona en nuestro navegador pero en ningún momento se conecta con el servidor para almacenar o leer datos de forma remota. Es momento de abordar este problema, para ello vamos a usar como tecnología Node.js aunque podríamos haber usado cualquier otra tecnología desde PHP, Java o .NET para lanzar servidor. He elegido Node.js por su sencillez.

## ¿Que es Node.js?

Node.js no es ni más ni menos que el motor V8 de Javascript de Google Chrome modificado para que pueda correr en un entorno servidor e interpretar el código Javascript que nosotros solicitemos.

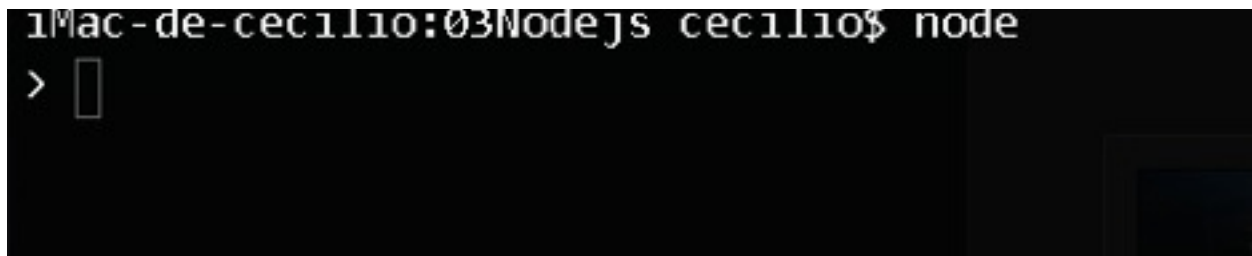


## Instalación de Node.js

La instalación de Node.js varía según la plataforma pero suele ser siempre relativamente sencilla. Para instalar Node.js iremos a su pagina web y nos descargaremos la versión correspondiente.

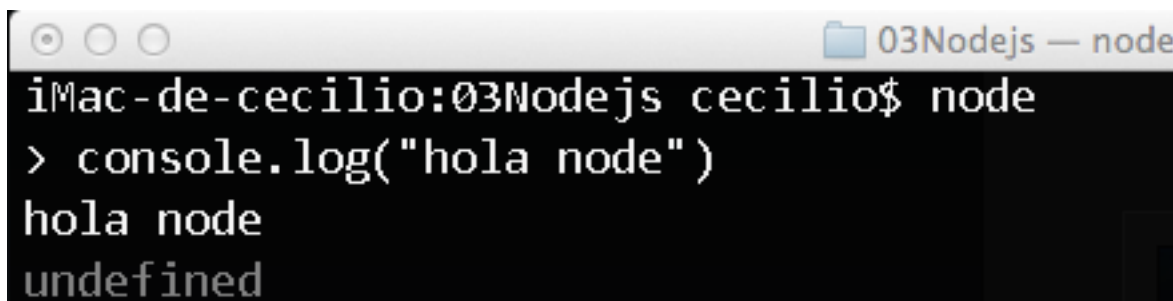
<http://nodejs.org/>

En la instalación solo hay que seguir los pasos del asistente. Terminada la instalación, abrimos una consola de sistema y comprobamos que Node.js funciona. Para ello, desde la consola de consola escribiremos “node” y pulsaremos enter.



```
iMac-de-cecilio:03Nodejs cecilio$ node
> 
```

Acabamos de entrar en la consola de Node.: nos aparece un simbolo “>” para que empecemos a escribir comandos sobre el interprete de Node. Escribimos uno de los comandos mas sencillos de Javascript : `console.log (“hola node”)`; pulsamos intro y nos imprimirá el mensaje por pantalla.

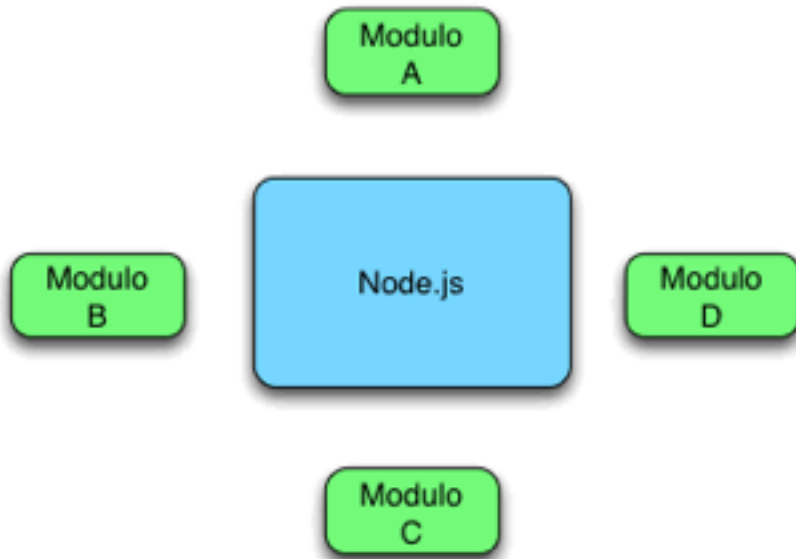


```
iMac-de-cecilio:03Nodejs cecilio$ node
> console.log("hola node")
hola node
undefined
```

Una vez realizada esta operación, escribimos `process().exit()`; y salimos del interprete de Node. Volvemos a estar a nivel de consola de sistema operativo.

## Node y NPM

Una de las ventajas que tiene Node es que es un sistema extremadamente modular que nos permite hacer prácticamente cualquier cosa. Eso sí, para hacerla tendremos que instalar los módulos que correspondan.



En nuestro caso vamos a convertir Node en un servidor Web que nos pueda enviar información en formato JSON que es lo que necesitamos para nuestra aplicación cliente. Para ello necesitamos instalar Express.js , que es un framework MVC de Servidor para Node. Vamos a utilizar ahora otra herramienta de Node : NPM ( Node Packaged Modules ) que nos instala cualquier módulo necesario . Desde la consola escribimos:

```
npm install express
```

**Puede haber gente que en sistemas linux necesite ejecutarlo con el comando sudo.** Si la ejecución es correcta, nos instalará el framework con las dependencias necesarias y veremos en la consola algo similar a esto.

```
— utils-merge@1.0.0
— escape-html@1.0.1
— merge-descriptors@0.0.2
— cookie@0.1.2
— cookie-signature@1.0.4
— finalhandler@0.1.0
— range-parser@1.0.0
— fresh@0.2.2
— media-typer@0.2.0
— vary@0.1.0
— methods@1.1.0
— parseurl@1.3.0
— buffer-crc32@0.2.3
— serve-static@1.5.2
— depd@0.4.4
— path-to-regexp@0.1.3
— qs@1.2.2
— proxy-addr@1.0.1 (ipaddr.js@0.1.2)
```

## Node y Servidor

Una vez instaladas las dependencias, es momento de crear nuestro primer programa en Node.js . Así pues, crearemos un fichero que se llame ServidorNodePublica.js con el siguiente contenido.

```
///////// Creo Facturas/////////
var listaFacturas=[];

var facturaDatos= {"id":"A","concepto":"mac","importe":1000};
var facturaDatos2= {"id":"B","concepto":"Iphone","importe":700};

listaFacturas.push(facturaDatos);
listaFacturas.push(facturaDatos2);

///////// Cargo Express ///////////

var express=require("express");
var app=express();
```

```
////////// URL de Facturas ////////////

app.get("/facturas",function(req,res) {

res.send(listaFacturas);

});

////////// Inicio el Servidor ////////////

var servidor= app.listen(3000,function() {

    console.log("servidor arrancado puerto 3000");

});
```

A pesar de la sencillez del código, éste merece una explicación detallada. En primer lugar genero la lista de Facturas que antes tenía en el cliente desde el servidor, ya que las vamos a solicitar vía Ajax.

```
var listaFacturas=[];

var facturaDatos= {"id":"A","concepto":"mac","importe":1000};
var facturaDatos2= {"id":"B","concepto":"Iphone","importe":700};

listaFacturas.push(facturaDatos);
listaFacturas.push(facturaDatos2);
```

El segundo paso es iniciar el framework Express.js y generar una url “/facturas” que nos devuelva la lista de facturas en formato JSON.

```
////////// Cargo Express ////////////

var express=require("express");
var app=express();

////////// URL de Facturas ////////////

app.get("/facturas",function(req,res) {
```

```
res.send(listaFacturas);  
  
});
```

Por último, solicito a Express.js que inicie un Servidor en el puerto 3000.

```
////////// Inicio el Servidor //////////
```

```
var servidor= app.listen(3000,function() {  
  
    console.log("servidor arrancado puerto 3000");  
  
});
```

Realizadas todas estas operaciones, salvamos este fichero en la carpeta actual donde usamos npm y le llamamos ServidorNodePublica .js. El siguiente paso es arrancar la aplicación ejecutando:

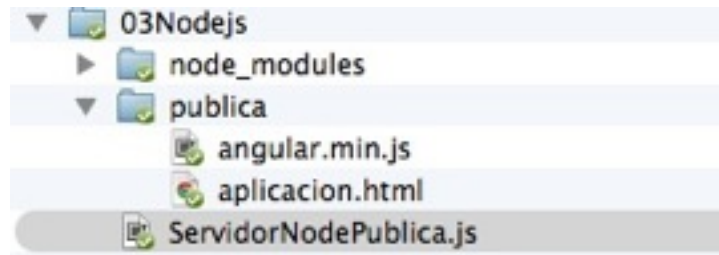
```
node ServidorNodePublica.js
```

Realizadas estas operaciones abrimos un navegador ya accedemos a la Url que nos devolverá la lista



## Node y Angular.js

Hemos terminado de instalar Node y configurarlo como servidor Web. Ahora podemos instalar nuestra aplicación en el servidor. Para ello vamos a construir una carpeta que se denomine “publica” donde almacenaremos el fichero HTML que tenemos , el javascript de Angular.js y nuestra Factura.



Realizada esta operación, modificaremos el fichero de Servidor de Node para que añada una carpeta de contenidos estáticos a la que podamos acceder.

```
app.use("/", express.static(__dirname+"/publica", 'public'));
```

```
var servidor= app.listen(3000,function() {  
  
    console.log("servidor arrancado puerto 3000");  
});
```

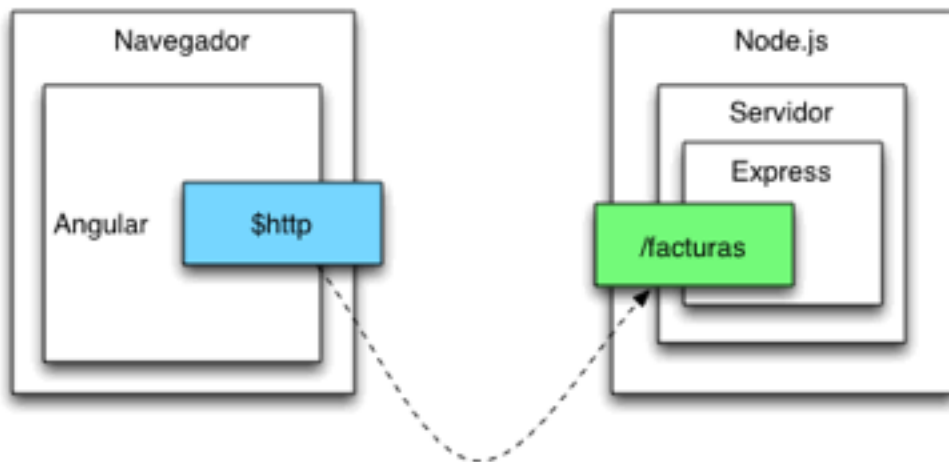
Como vemos, el código a incluir es muy sencillo pues se trata de una única línea.

```
app.use("/", express.static(__dirname+"/publica", 'public'));
```

Esta línea da de alta la carpeta “publica” como recurso accesible desde un navegador. Una vez hecho esto, podemos solicitar sin ningún problema el fichero “aplicacion.html” al servidor. Reiniciamos node (**lo paramos con Ctrl C y lo volvemos a lanzar**) La aplicación de Angular.js carga sin problemas y hemos terminado de configurar un servidor básico.



En el siguiente capítulo comenzaremos a trabajar con Ajax





# Angular y Ajax

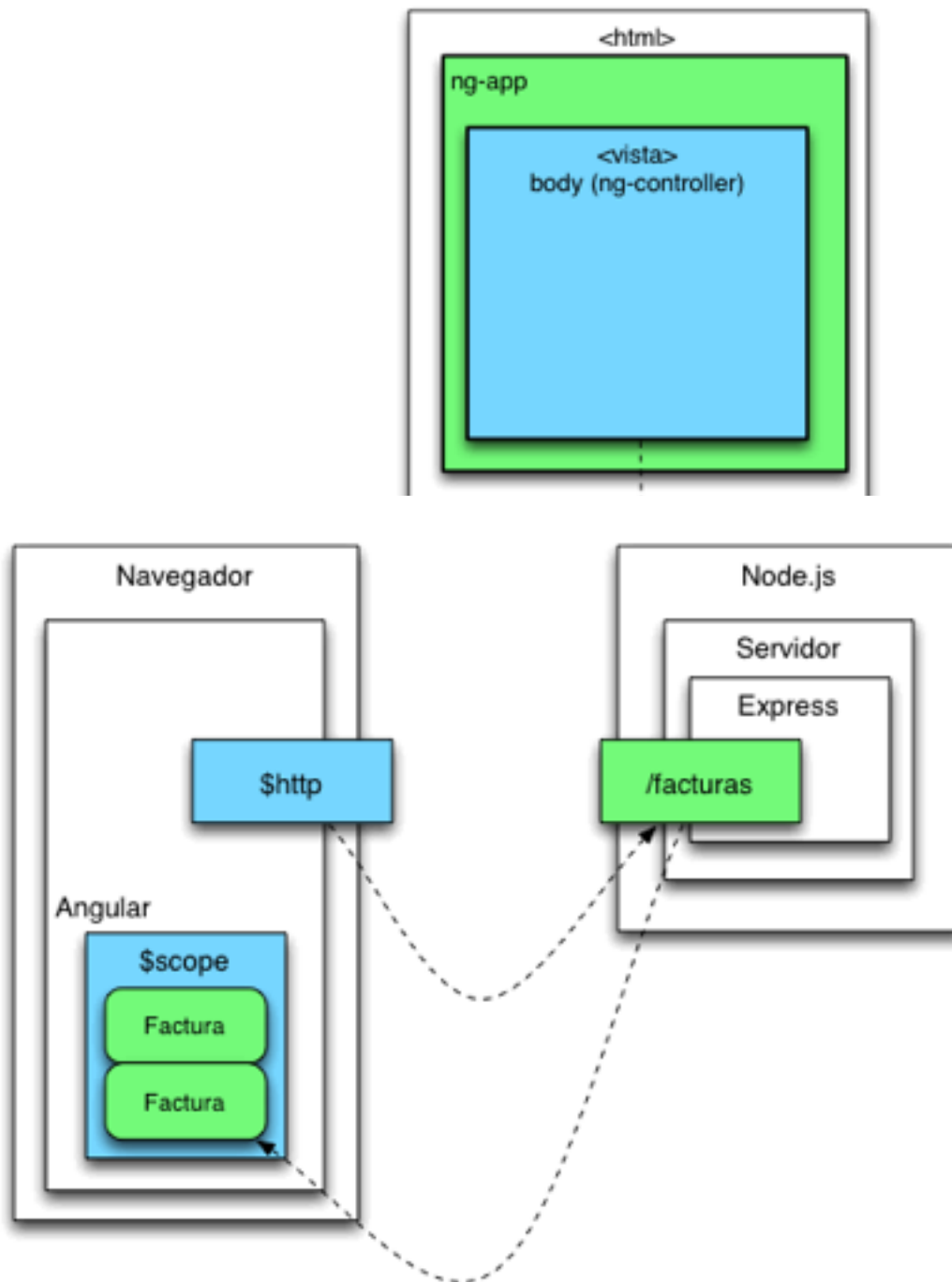
En el capítulo anterior hemos configurado Node.js para que nos sirva como Servidor Web. Es momento de comenzar a utilizarlo con Angular. En estos momentos nuestra aplicación carga una lista de Facturas en el \$scope de su controlador.

```
var facturaDatos= {"id":"A","concepto":"mac","importe":1000};
var facturaDatos2= {"id":"B","concepto":"Iphone","importe":700};
var listaFacturas=[];
listaFacturas.push(new Factura(facturaDatos));
listaFacturas.push(new Factura(facturaDatos2));
$scope.listaFacturas=listaFacturas;
```

Es momento de comenzar a realizar los primeros cambios y cargar esta lista desde el Servidor. Para ello, el primer paso será modificar nuestra aplicación Cliente para que tenga las capacidades de realizar una petición HTTP.

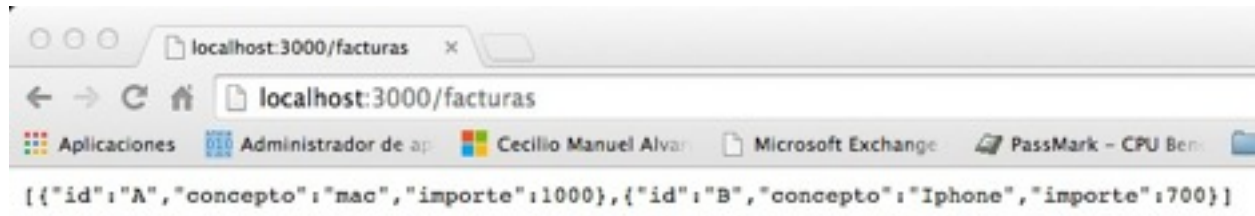
## El servicio de \$http

Hemos estado usando durante los capítulos anteriores la variable \$scope. Esta variable nos permite almacenar objetos que son compartidos entre el controlador y la vista. **\$scope es un ejemplo de servicio a nivel de Angular.** Un servicio, como su nombre indica, es una clase que provee de una funcionalidad determinada al framework. En el caso del \$scope permite compartir objetos entre la Vista y el Controlador.



Vamos a introducir en este capítulo un nuevo servicio, **el servicio de \$http** que evidentemente está orientado a realizar peticiones HTTP vía Ajax. Si recordamos,

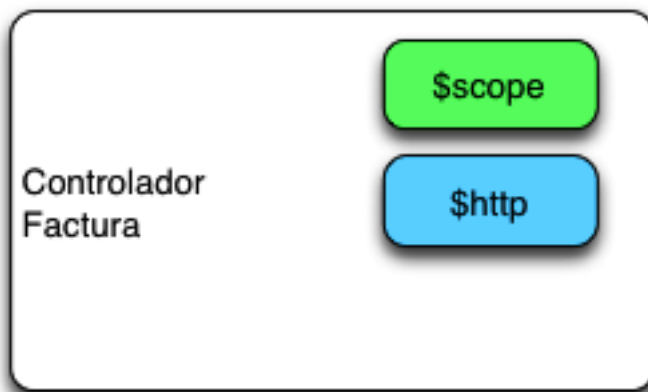
en el capítulo anterior hemos definido una Url que nos devuelve una lista de Facturas en el Servidor.



Así pues, el siguiente paso es introducir y usar el servicio `$http` de Angular para traernos los datos del Servidor al cliente.

## Usando `$http`

Para poder comenzar a usar el servicio, deberemos inyectarlo a nivel del módulo de igual forma que en los primeros capítulos inyectamos `$scope`.



Vamos a ver como se implementa en código :

```
angular.module('moduloFacturas',  
[]).controller("controladorFacturas",function($scope,$http) {
```

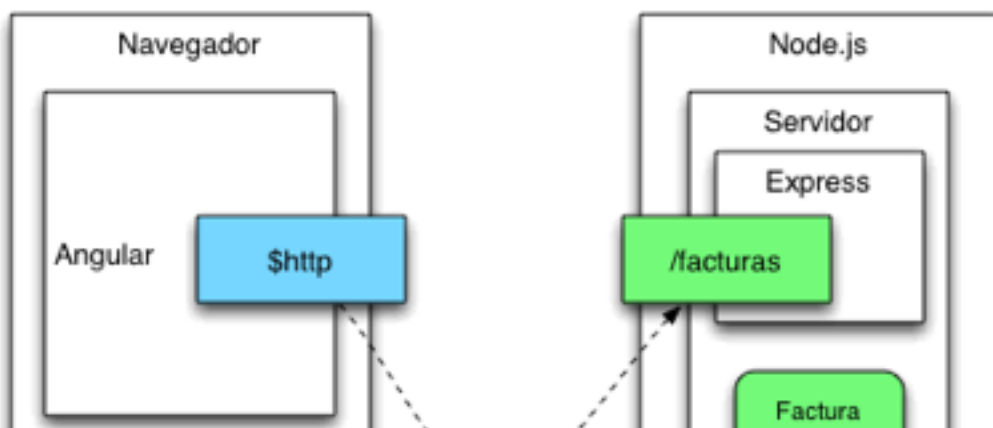
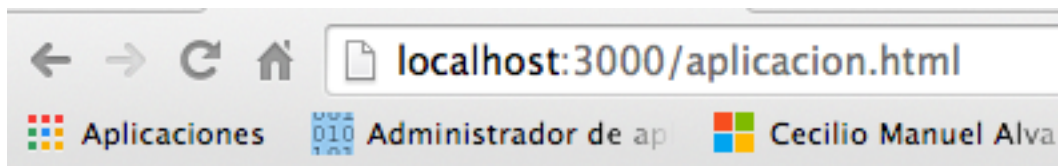
```
.....  
}
```

Una vez que hayamos inyectado el servicio de \$http, podemos pasar a usarlo. Para ello invocaremos la Url de /facturas utilizando su método get(). Realizada esta operación, invocaremos el método de callback de toda llamada Ajax, que aquí se denomina success().

```
var petition= $http.get("facturas");
```

```
petition.success(function(listaFacturas) {  
  var listaObjetosFacturas=[];  
  for (var i=0;i<listaFacturas.length;i++) {  
    listaObjetosFacturas.push(new Factura(listaFacturas[i]));  
  }  
  $scope.listaFacturas=listaObjetosFacturas;  
});
```

Como puede verse, realizamos una petición \$http que nos devuelve una lista de Facturas en formato JSON, la cuál convertimos a un Array de Objetos y añadimos al \$scope.



Hemos eliminado la lista de objetos que teníamos en el Cliente la hemos substituido por una petición Ajax de servidor . El resultado es idéntico a los ejemplos anteriores, solamente que en este caso los datos se cargan desde Node.js.

## \$http y POST

El siguiente paso será insertar nuevas Facturas en el Servidor a través de una petición de tipo POST usando el servicio \$http.

Para ello deberemos modificar nuestra aplicación de Servidor para que soporte peticiones de tipo POST. Al ser Node y Express productos muy modulares, **necesitamos instalar un nuevo módulo que soporte este tipo de peticiones.** Así pues volveremos a usar la herramienta npm y ejecutaremos la siguiente instrucción:

```
npm install body-parser
```

Esta extensión nos permitirá utilizar Node.js para gestionar de una forma sencilla las peticiones POST. Realizada esta operación, simplemente usamos el framework Express.js con el nuevo módulo y añadimos una nueva ruta que capture las peticiones de tipo POST y nos añada una nueva Factura a la lista que tenemos en el servidor.

```
var bodyParser = require('body-parser');
app.use( bodyParser.json() );
app.post("/facturas",function(req,res) {

listaFacturas.push(req.body);
res.send("ok");

});
```

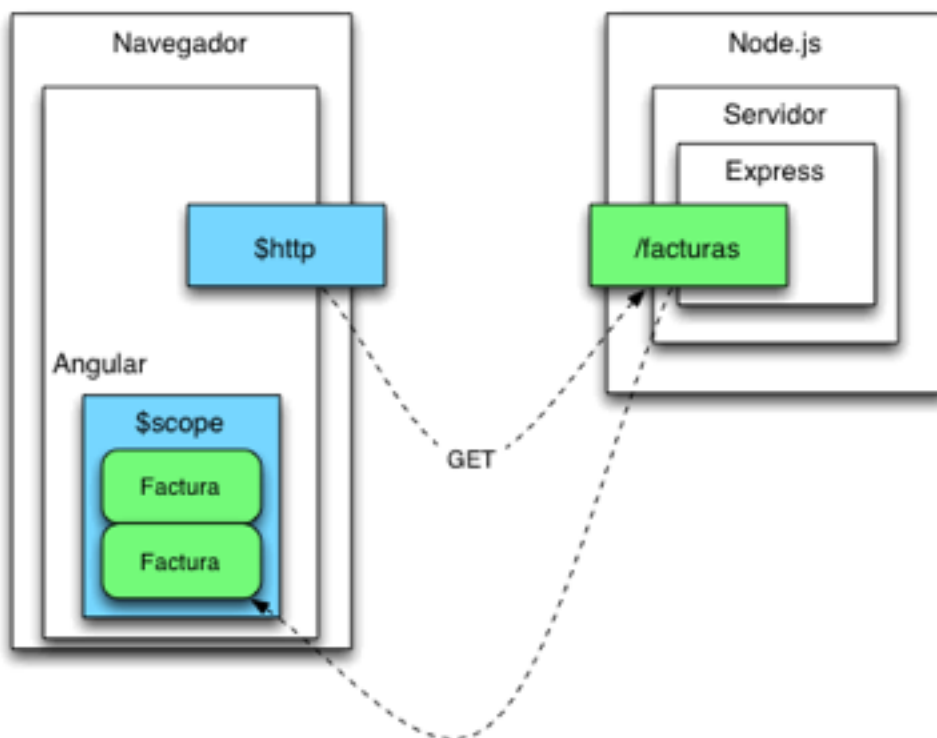
Hemos terminado de construir la parte del Servidor. Nos queda por completar la parte que es puramente de Cliente. Vamos a ver como gestionarla desde Angular:

```
$scope.addFactura=function() {  
  
    var peticion= $http.post("facturas",$scope.nuevaFactura);  
  
    peticion.success(function(mensaje) {  
  
        console.log("ok");  
    });  
}
```

Como podemos ver, la operativa es realmente intuitiva usamos \$http.post y le pasamos una nueva factura.

```
var peticion= $http.post("facturas",$scope.nuevaFactura);
```

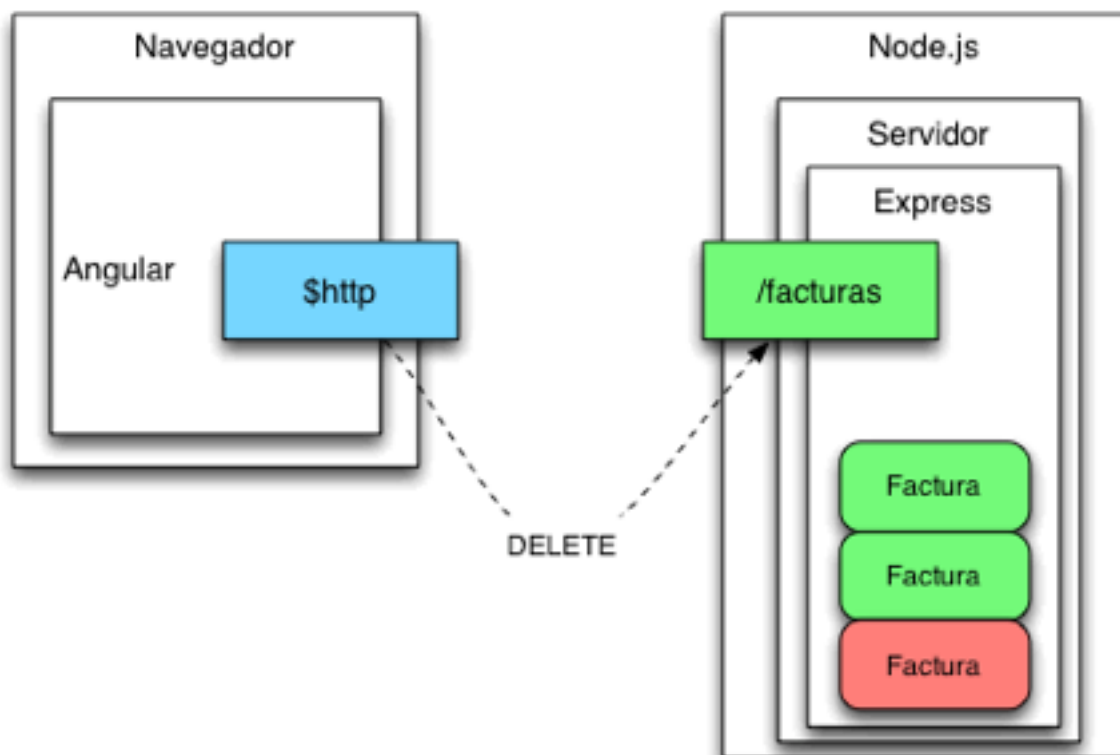
Hecho esto, habremos insertado un nuevo registro a nivel del servidor. Sin embargo la parte cliente no está actualizada, así pues deberemos realizar otra petición Ajax dentro del método success de la petición POST que vuelva a seleccionar la lista de Facturas.



Así pues el método `success()` de nuestra petición `post` deberá implementar el siguiente código que nos refresque la lista:

```
peticion.success(function(datos) {  
  var peticion= $http.get("facturas");  
  
  peticion.success(function(listaFacturas) {  
  
    var listaObjetosFacturas=[];  
    for (var i=0;i<listaFacturas.length;i++) {  
  
      listaObjetosFacturas.push(new Factura(listaFacturas[i]));  
  
    }  
  
    $scope.listaFacturas=listaObjetosFacturas;  
  
  });  
});
```

Acabamos de construir el código de la aplicación que es capaz de listar e insertar datos en el Servidor vía Ajax.



## \$http y DELETE

Nos queda todavía una última operación por realizar: **la operación de borrado**.

Nos encontramos ante una casuística similar a la anterior y por lo tanto necesitamos implementar el lado Servidor primero para luego enlazar con el Cliente.

Vamos a ver como quedaría la aplicación de Node.js a nivel de Servidor cuando gestiona una petición de borrado utilizando una petición DELETE:

```
app.delete("/facturas/id/:id",function(req,res) {  
  
    listaFacturas=listaFacturas.filter(function (factura) {  
  
        return req.params.id!==factura.id;  
    });  
    res.send("ok");  
});
```

Como podemos ver, la petición de DELETE únicamente recibe el id de la Factura que queremos borrar. Una vez recibido, se encarga de borrar la Factura usando la función de filter. Nos queda por implementar la parte del Cliente en Angular . Para ello vamos a usar \$http su método delete.

```
$scope.removeFactura=function(factura) {  
  
    var peticion= $http.delete("facturas/id/"+factura.id);  
  
    peticion.success(function(datos) {  
  
        //resto de código  
    });  
  
}
```

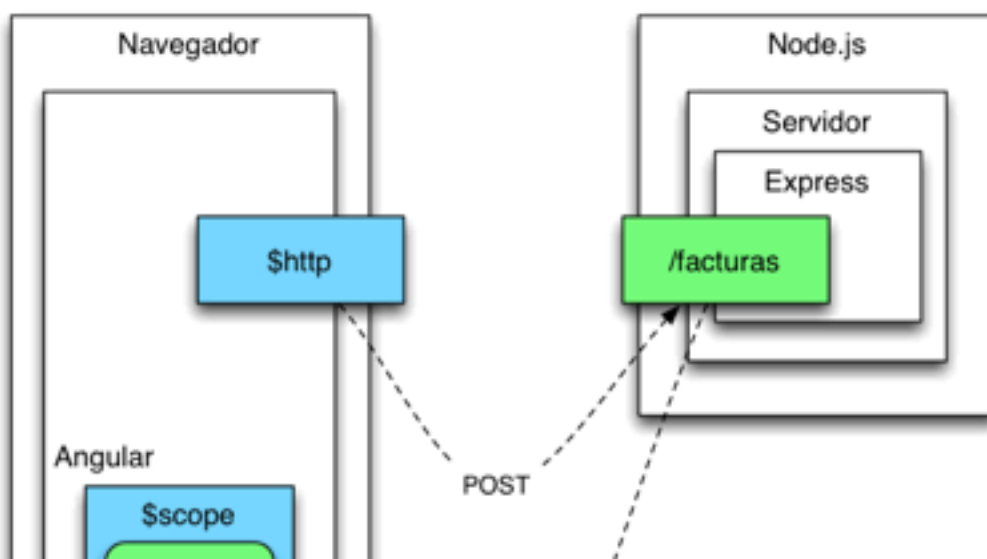
Acabamos de borrar la factura del servidor como nos paso anteriormente pero debemos volver a actualizar la lista y para ello, en el método success realizaremos una petición GET.



El código será el siguiente:

```
peticion.success(function(datos) {  
  
    var peticion= $http.get("facturas");  
  
    peticion.success(function(listaFacturas) {  
  
        var listaObjetosFacturas=[];  
  
        for (var i=0;i<listaFacturas.length;i++) {  
  
            listaObjetosFacturas.push(new Factura(listaFacturas[i]));  
  
        }  
  
        $scope.listaFacturas=listaObjetosFacturas;  
  
    });  
  
});
```

Hemos terminado de realizar las operaciones con el servicio de \$http y todo funciona de forma correcta. Sin embargo hay algo que podemos mejorar.



## Angular y DRY Networking

El problema que tenemos en estos momentos es que estamos realizando más peticiones HTTP de las que realmente necesitamos **ya que nos estamos trayendo una nueva lista de facturas cada vez que insertamos o borramos contenido para actualizar nuestra lista.** Pronto nos daremos cuenta de que teníamos cargada la lista de Facturas en el Cliente salvo la última. Lo único que necesitamos es añadir la nueva Factura que acabamos de insertar a nivel de servidor en el Cliente y no volver a recargar la lista por completo. Vamos a modificar el código que tenemos a nivel de Servidor para gestionar la petición POST de una forma mejor y que nos devuelva nuestra factura.

```
app.post("/facturas",function(req,res) {  
  
    listaFacturas.push(req.body);  
    res.send(req.body);  
  
});
```

La operación que estamos realizando es bastante sencilla. Estamos insertando la Factura a nivel de servidor y una vez insertada la estamos devolviendo como respuesta de la petición POST en formato JSON. De esta forma, con una única petición HTTP podemos actualizar el Servidor y actualizar el Cliente. Sin tener que repetir continuamente el envío de la lista de Facturas actualizada. Hecho esto, el código Cliente de Angular.js quedará mucho más sencillo.

```
$scope.addFactura=function() {  
  
    var peticion= $http.post("facturas",$scope.nuevaFactura);  
  
    peticion.success(function(factura) {  
  
        $scope.listaFacturas.push(factura);  
  
    });  
};
```

```
}
```

## Borrado y DRY Neworking

La misma operativa se puede definir para la acción de borrar. Modificaremos el código del servidor para que nos devuelva el id de la Factura que deseamos eliminar del Cliente.

```
app.delete("/facturas/id/:id",function(req,res) {  
  
    listaFacturas.filter(function (factura) {  
  
        return req.params.id!==factura.id;  
    });  
  
    res.send(req.params.id);  
  
});
```

Modificaremos el código de Cliente para que nos elimine el elemento de la lista.

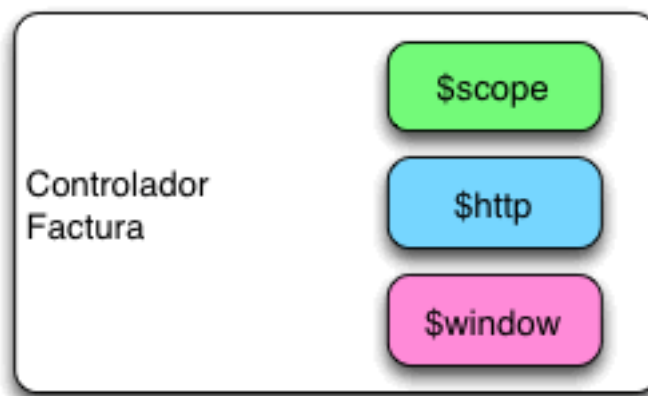
```
$scope.removeFactura=function(factura) {  
  
    var peticion= $http.delete("facturas/id/"+factura.id);  
    peticion.success(function(id) {  
        $scope.listaFacturas=$scope.listaFacturas.filter(function (factura) {  
  
            return id!==factura.id;  
        });  
    }  
}
```

Hemos terminado de simplificar el uso del servicio \$http reduciendo las peticiones y el tráfico de datos entre cliente y servidor. Hemos aplicado el principio DRY pero a nivel de red (**DRY NETWORKING**). Está claro que no siempre podremos aplicar este principio ya que existen casos en los que otras aplicaciones actualizan

los datos. En este tipo de situaciones deberemos apoyarnos en conceptos más avanzados que quedan fuera del contenido del libro y que se denominan “promesas”.

## El servicio \$window

Para familiarizarnos con el uso de servicios, vamos a utilizar otro de los más habituales: el servicio \$window. \$window nos da acceso a los cuadros de diálogo de Javascript (confirm,alert,prompt). Vamos a usarlo para pedir confirmación de la eliminación de registros. El primer paso como siempre será inyectar el servicio.



Vamos a ver el código :

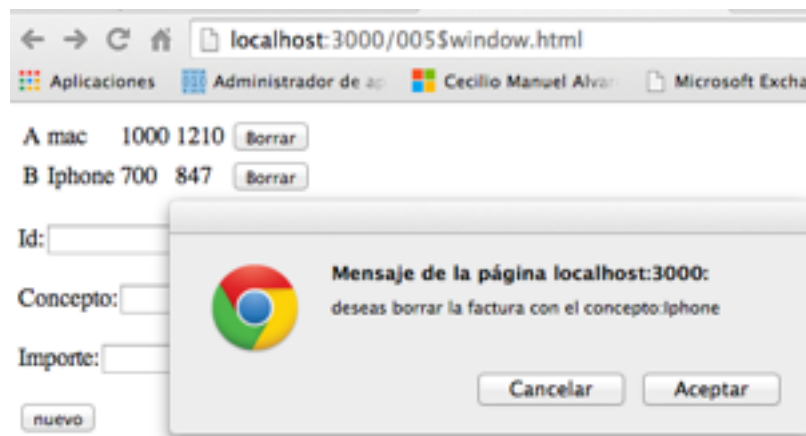
```
angular.module('moduloFacturas', []).controller("controladorFacturas",  
    function($scope,$http,$window) {
```

Realizado este primer paso, vamos a utilizar el servicio en el evento de borrar para primero pedir confirmación y luego borrar el registro en caso afirmativo.

```
$scope.removeFactura=function(factura) {
```

```
if($window.confirm("deseas borrar la factura con el  
concepto:"+factura.concepto)) {  
  
var peticion= $http.delete("facturas/id/"+factura.id);  
  
peticion.success(function(id) {  
  
    $scope.listaFacturas=$scope.listaFacturas.filter(function (factura) {  
  
        return id!=factura.id;  
    });  
  
});  
}  
}
```

Una vez construido el código, si pulsamos sobre el botón de borrar, nos saldrá el siguiente mensaje.



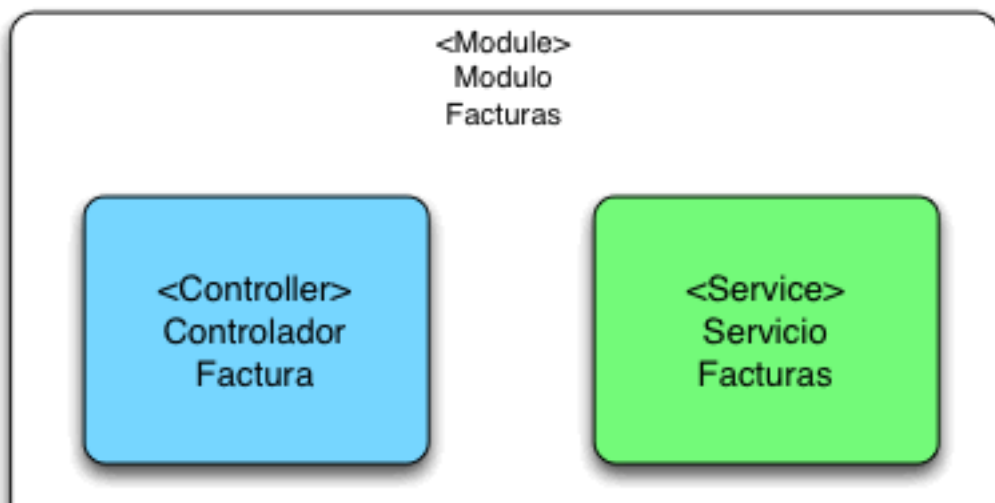
Hemos terminado de gestionar las peticiones Ajax y de introducir el concepto de servicio es momento de seguir avanzando.

# Creación de Servicios

En el capítulo anterior hemos usado fundamentalmente el servicio \$http para gestionar las peticiones Ajax con el Servidor. En este capítulo vamos a intentar organizar un poco más nuestra aplicación, separando aún más las responsabilidades. En este momento es el controlador el que se encarga de gestionar prácticamente todo. Vamos a aislar la responsabilidad de las peticiones Ajax de los propios controladores .



Así, otros controladores u otras piezas de la aplicación pueden usarlas y de esta manera el código es reutilizable. Para poder realizar esta operación deberemos crear nuestros propios Servicios especializados.



## Creación de ServicioFacturas

Igual que en los primeros capítulos del libro hemos creado el ControladorFacturas dentro del módulo de Facturas, en este capítulo crearemos un Servicio dentro del módulo al cuál denominaremos ServicioFacturas. Vamos a ver cómo se implementa a nivel de código:

```
angular.module('moduloFacturas').controller("controladorFacturas",
    function($scope,$http,$window,servicioFacturas) {

// código del controlador omitido

}).service("servicioFacturas", function($http){

    this.listaFacturas=function() {

        return $http.get("facturas");
    }

    this.addFactura=function(factura) {

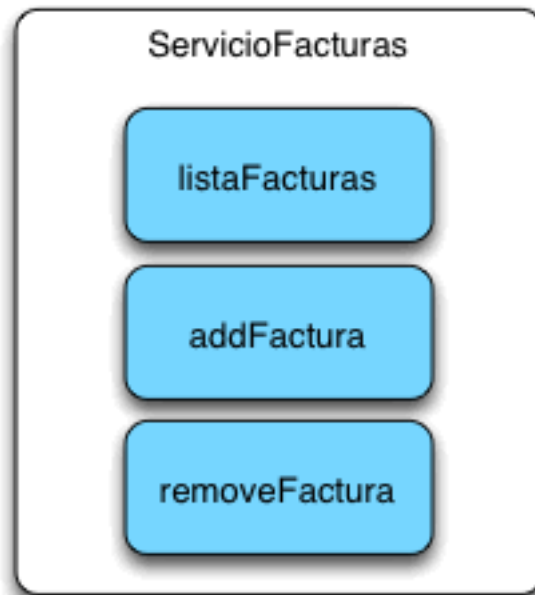
        return $http.post('/facturas',factura);
    }
    this.removeFactura=function(factura) {

        return $http.delete('/facturas/id/'+factura.id)

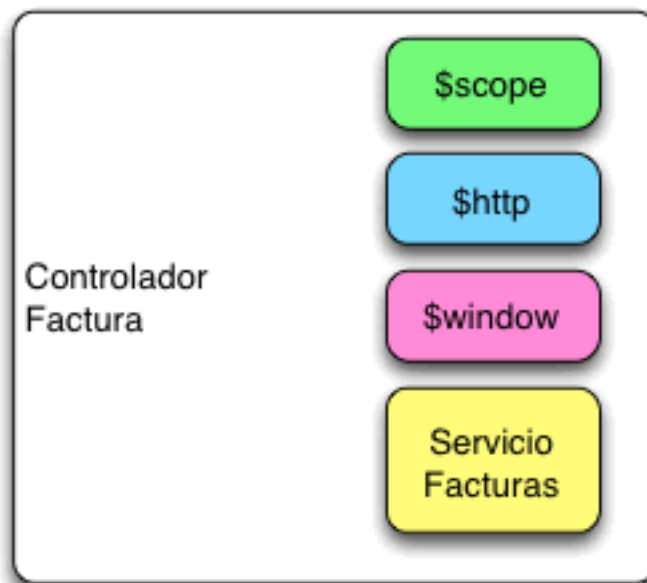
    }

});
```

Acabamos de crear el Servicio como un elemento dentro del Módulo. Un Servicio es una clase Singleton que agrupa un conjunto de funcionalidad con el objetivo de poderse reutilizar.



Una vez creado el servicio, el controlador podrá hacer uso de él. Para ello deberemos inyectarlo como pasa con el resto de servicios que ya tenemos.





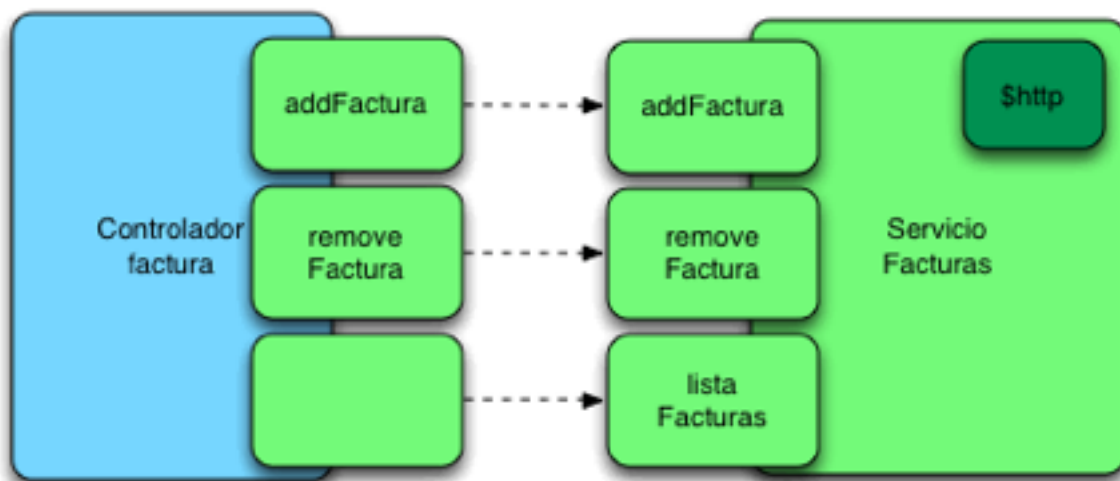
El código es similar a situaciones anteriores :

```
angular.module('moduloFacturas', []).controller("controladorFacturas",  
    function($scope,$http,$window,servicioFacturas) {  
  
    }  
}
```

Realizada esta operación, nuestro controlador podrá delegar en la clase de servicio para realizar todas las llamadas Ajax.



Veamos cómo queda el controlador una vez realizados los cambios y construido el servicio en el cual delega una gran parte de su funcionalidad



Vamos a verlo en código :

```
var peticion= servicioFacturas.listaFacturas();

    peticion.success(function(listaFacturas) {

        var listaObjetosFacturas=[];

        for (var i=0;i<listaFacturas.length;i++) {

            listaObjetosFacturas.push(new Factura(listaFacturas[i]));

        }

        $scope.listaFacturas=listaObjetosFacturas;

    });

$scope.addFactura=function() {

    var peticion=
servicioFacturas.addFactura($scope.nuevaFactura);

    peticion.success(function(factura) {

        $scope.listaFacturas.push(new Factura(factura));

    });

}

$scope.removeFactura=function(factura) {

    if($window.confirm("deseas borrar la factura con el
    concepto:"+factura.concepto)) {

var peticion= servicioFacturas.removeFactura(factura);
```

```
peticion.success(function(id) {  
  
    $scope.listaFacturas=$scope.listaFacturas.filter(function (factura) {  
  
        return id!==factura.id;  
    });  
  
});  
}
```

Como podemos ver los cambios no son complejos y simplemente sustituimos el uso de \$http por el uso del ServicioFacturas.

## Servicios y Objetos

En estos momentos tenemos una funcionalidad candidata a ser parte de un Servicio. Me refiero al siguiente bloque de código, que se encarga de convertir una estructura JSON proveniente del servidor en un conjunto de objetos de Javascript.

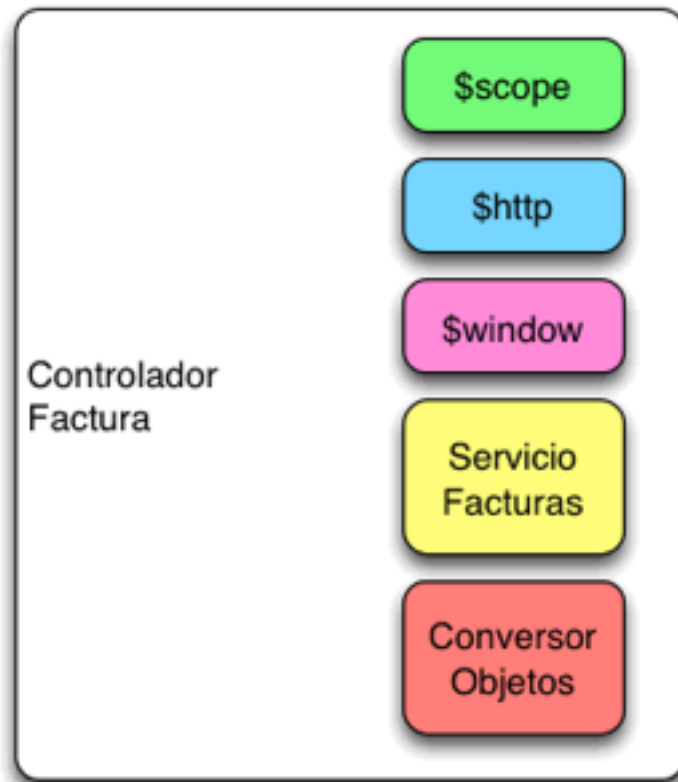
```
var listaObjetosFacturas=[];  
  
for (var i=0;i<listaFacturas.length;i++) {  
  
    listaObjetosFacturas.push(new Factura(listaFacturas[i]));  
  
}
```

Para evitar repeticiones de código vamos a convertir también esta lógica en un servicio al que llamaremos conversorObjetos:

```
.service("conversorObjetos", function(){  
    this.convertirFacturas=function(listaFacturas) {  
  
        var listaObjetosFacturas=[];  
  
        for (var i=0;i<listaFacturas.length;i++) {
```

```
        listaObjetosFacturas.push(new Factura(listaFacturas[i]));  
    }  
    return listaObjetosFacturas;  
}  
  
});
```

Realizada esta operación, el siguiente paso será que nuestro Controlador utilice este servicio.



Como siempre inyectamos el nuevo servicio que nos ayuda a simplificar el código :

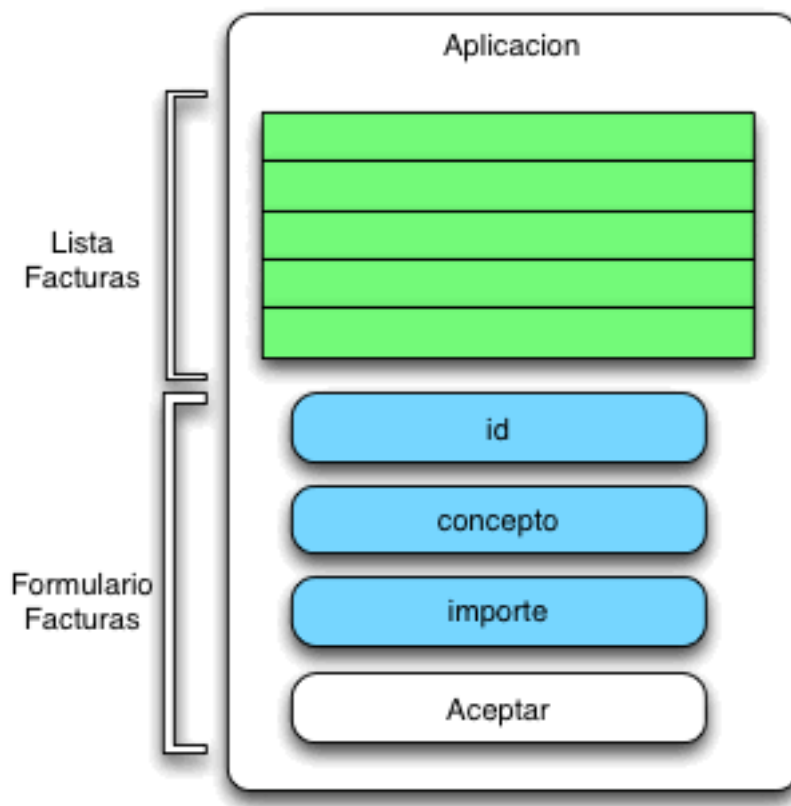
```
function($scope,$http,$window,  
servicioFacturas,conversorObjetos) {
```

```
var petition= servicioFacturas.listaFacturas();  
  
petition.success(function(listaFacturas) {  
  
    $scope.listaFacturas=conversorObjetos.convertirFacturas(listaFacturas);  
  
});
```

Hemos aprovechado el capítulo para crear dos Servicios y dividir mejor las responsabilidades, aislando el controlador de las llamadas Ajax y de las transformaciones.

# Angular y SPA

En el capítulo anterior hemos organizado el código de la capa de Servicios. Es momento de organizar el código que está más ligado a la capa de presentación. En estos momentos tenemos una página que incluye una lista y un formulario.



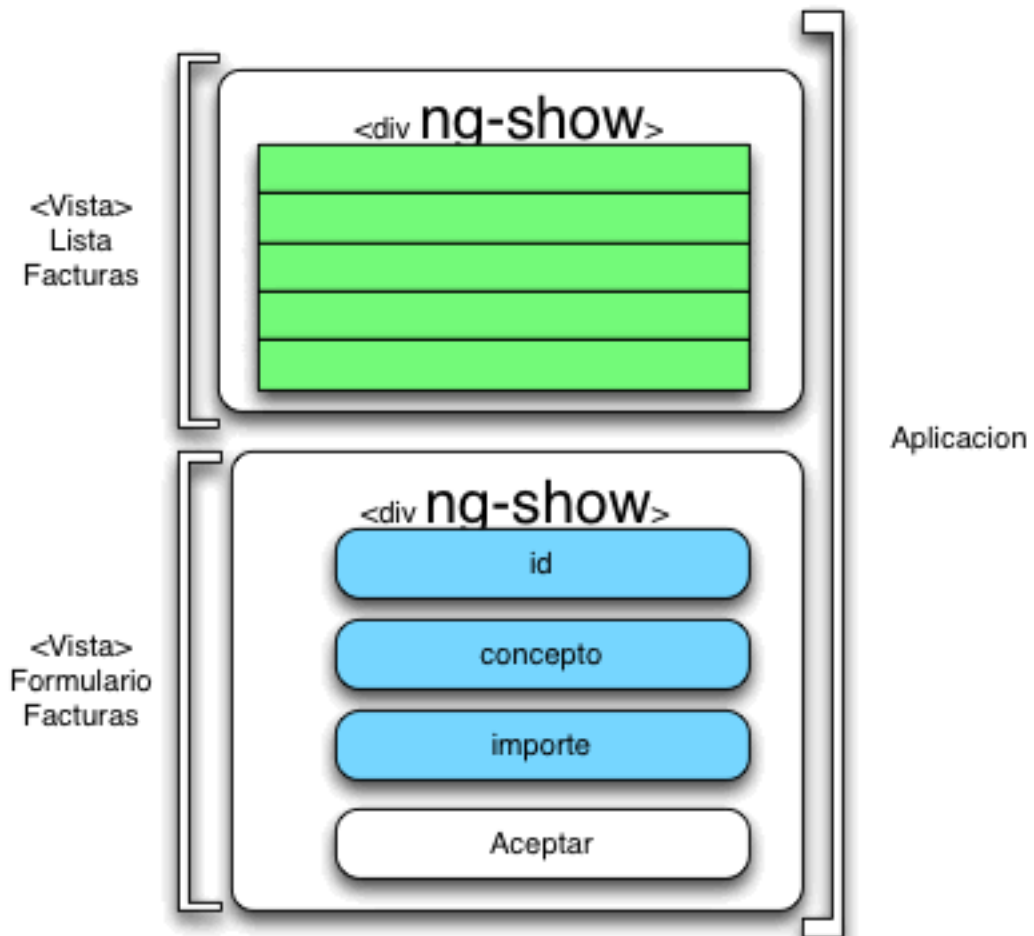
Es evidente que ésta no es la estructura de elementos que deseamos. Vamos a dividir nuestra aplicación utilizando dos DIV de HTML, de tal manera que cada uno de estos elementos incluya una de las vistas.

ARQ



## Directiva ng-show

Acabamos de añadir dos divisores a la aplicación que delimitan cada una de las Vistas que la aplicación va a tener. Vamos a añadir a cada uno de los <div> una directiva ng-show que nos permitirá determinar cuándo una parte de la aplicación se muestra y cuando se oculta.



Vamos a terminar de clarificar las ideas mostrando el nuevo código HTML:

```
<body ng-controller="controladorFacturas">
<div ng-show="mostrarLista">
<table >
<tr ng-repeat="factura in listaFacturas">
<td>{{factura.id}}</td>
<td>{{factura.concepto}}</td>
<td>{{factura.importe}}</td>
<td>{{factura.calcularIva()}}</td>
<td><input type="button" ng-click="removeFactura(factura)" value="Borrar"/></td>
</tr>
</table>
<input type="button" ng-click="verFormulario()" value="nuevo"/>
</div>
<div ng-show="mostrarFormulario">
<form>
<p>
Id:<input type="text" ng-model="nuevaFactura.id"/>
</p>

<p>
Concepto:<input type="text" ng-model="nuevaFactura.concepto"/>
</p>
<p>
Importe:<input type="text" ng-model="nuevaFactura.importe"/>
</p>

<input type="button" ng-click="addFactura()" value="nuevo"/>
</form>
</div>
</body>
```

Como podemos ver, hemos añadido etiquetas <div> y la directiva ng-show. Eso sí, cada directiva ng-show está ligada a una variable que deberemos asignar en el \$scope.



```
<div ng-show="mostrarLista">  
<div ng-show="mostrarFormulario">
```

## \$scopes y variables

Vamos a ver detalladamente las modificaciones de código que tenemos que hacer a nivel de nuestro controlador para gestionar los divs que acabamos de crear:

```
$scope.mostrarFormulario=false;  
$scope.mostrarLista=true;  
  
var peticion= servicioFacturas.listaFacturas();  
  
peticion.success(function(listaFacturas) {  
  
    $scope.listaFacturas=conversorObjetos.convertirFacturas(listaFacturas);  
  
        $scope.mostrarFormulario=false;  
        $scope.mostrarLista=true;  
  
    });  
  
$scope.addFactura=function() {  
  
    var peticion= servicioFacturas.addFactura($scope.nuevaFactura);  
  
    peticion.success(function(factura) {  
  
        $scope.listaFacturas.push(new Factura(factura));  
        $scope.mostrarFormulario=false;  
        $scope.mostrarLista=true;  
    });  
}  
  
$scope.verFormulario=function() {
```

```
    $scope.mostrarFormulario=true;
    $scope.mostrarLista=false;
}
```

//omitimos el código de eliminar

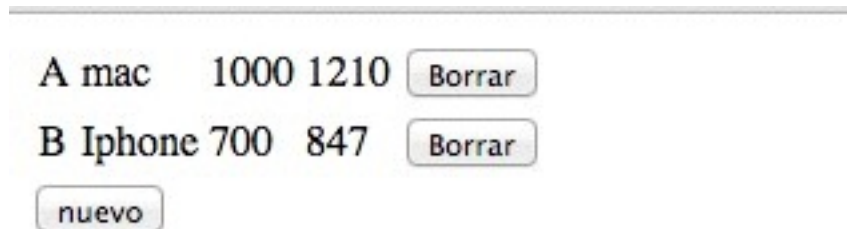
Como podemos ver, ahora tenemos dos nuevas variables en el \$scope:

```
$scope.mostrarFormulario=false;
$scope.mostrarLista=true;
```

Son las encargadas de mostrar y ocultar una u otra vista. Para ello hemos añadido un nuevo botón debajo de la tabla .

```
<input type="button" ng-click="verFormulario()" value="nuevo"/>
```

Esto nos permitirá cambiar de la Vista de tabla a la de Formulario



Para ello hemos creado una función a nivel del \$scope que intercambia los valores de la vista.

```
$scope.verFormulario=function() {

    $scope.mostrarFormulario=true;
    $scope.mostrarLista=false;

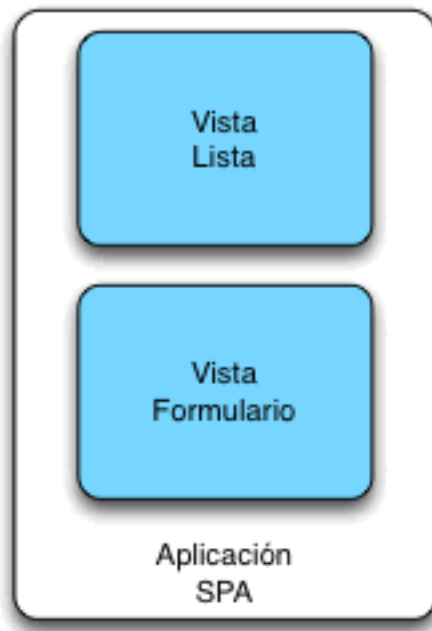
}
```

Pulsando este botón pararemos a la Vista de formulario y podremos añadir una nueva Factura.



Formulario para añadir una nueva factura. Incluye tres campos de texto etiquetados como 'Id:', 'Concepto:', y 'Importe:', y un botón etiquetado como 'nuevo'.

Una vez implementado todo, la aplicación se comportará como una **Aplicación SPA. SPA (Simple Page Application)**: se trata de una aplicación que tiene todas sus vistas cargadas en un único fichero HTML y las va intercambiando.



## La directiva ng-switch

Aunque hemos conseguido separar las vistas, lo hemos hecho de una forma muy elemental. Vamos a trabajarlo un poco más a través de la directiva ng-switch que nos permite reducir más el código. Para ello vamos a definir un <div> que englobe a los demás:

```
<body ng-controller="controladorFacturas">
<div ng-switch="vista">
<div ng-switch-when="lista">
<table >
<tr ng-repeat="factura in listaFacturas">
<td>{{factura.id}}</td>
<td>{{factura.concepto}}</td>
<td>{{factura.importe}}</td>
<td>{{factura.calcularIva()}}</td>
<td><input type="button" ng-click="removeFactura(factura)" value="Borrar"/>
</td>
</tr>
</table>
<input type="button" ng-click="verFormulario()" value="nuevo"/>
</div>
<div ng-switch-when="formulario">
<form>
<p>
Id:<input type="text" ng-model="nuevaFactura.id"/>
</p>

<p>
Concepto:<input type="text" ng-model="nuevaFactura.concepto"/>
</p>
<p>
Importe:<input type="text" ng-model="nuevaFactura.importe"/>
</p>

<input type="button" ng-click="addFactura()" value="nuevo"/>
</form>
</div>
</div>
```

Las directivas son similares a las de ng-show , lo que simplificaremos será el código del controlador que quedará como sigue :

```
$scope.vista="lista";
$scope.nuevaFactura={};

var peticion= servicioFacturas.listaFacturas();
peticion.success(function(listaFacturas) {
    $scope.listaFacturas=conversorObjetos.convertirFacturas(listaFacturas);

    $scope.vista="lista";
});

$scope.addFactura=function() {

console.log($scope.nuevaFactura);
var peticion= servicioFacturas.addFactura($scope.nuevaFactura);

peticion.success(function(factura) {

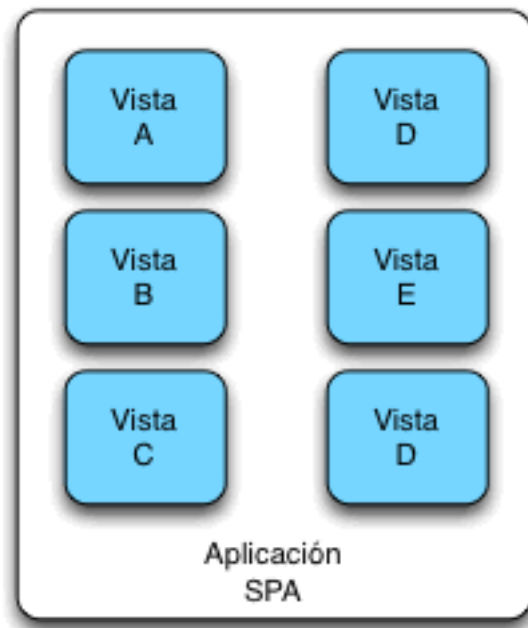
    $scope.listaFacturas.push(new Factura(factura));
    $scope.vista="lista";
});
}

$scope.verFormulario=function() {

    $scope.vista="formulario";

}
```

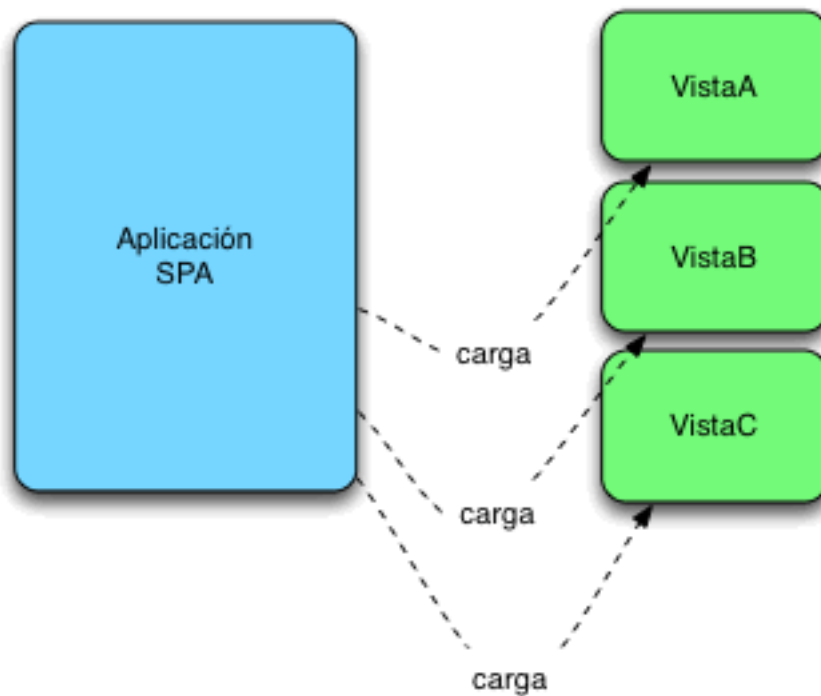
En este caso, como podemos ver, con una única variable gestionamos ambas vistas. Sin embargo nos queda solventar un problema. Para una aplicación pequeña este sistema funciona correctamente. **Ahora bien, cuando la aplicación incrementa su tamaño, tendremos que cargar de forma inicial cientos de Vistas lo cuál no es viable**



En el siguiente capítulo abordaremos esta problemática.

# Angular y ngRoute

Cuando la aplicación vaya creciendo de tamaño necesitaremos aplicar otro enfoque que nos permita una mayor flexibilidad. Para ello vamos a instalar una **extensión de Angular.js que se denomina ng-route e implementa un sistema de enrulado que aporta mayor flexibilidad y permite la carga dinámica de contenidos.**



## Instalación ng-route

La instalación del sistema de enrulado no es excesivamente compleja. El primer paso es el obtener el nuevo modulo de enrulado. Para ello vamos a la zona de descargas de Angular.js y pulsamos sobre extras o módulos adicionales.



Una vez situados en extras, seleccionamos el fichero de angular-route.js y lo descargamos .

```
angular-mocks.js  
angular-resource.js  
angular-resource.min.js  
angular-resource.min.js.map  
angular-route.js  
angular-route.min.js
```

Obtenido el fichero, el siguiente paso es ubicarlo dentro de nuestro HTML

```
<script type="text/javascript" src="factura.js">  
</script>  
<script type="text/javascript" src="angular.min.js">
```



```
</script>
```

```
<script src="angular-route.js" type="text/javascript">  
</script>
```

## Configuración del módulo

Nosotros ya tenemos configurado un módulo que es el módulo de nuestra aplicación “moduloFacturas”. Vamos a inyectar el módulo de ngRoute dentro de nuestro módulo para poder usarlo sin problemas.



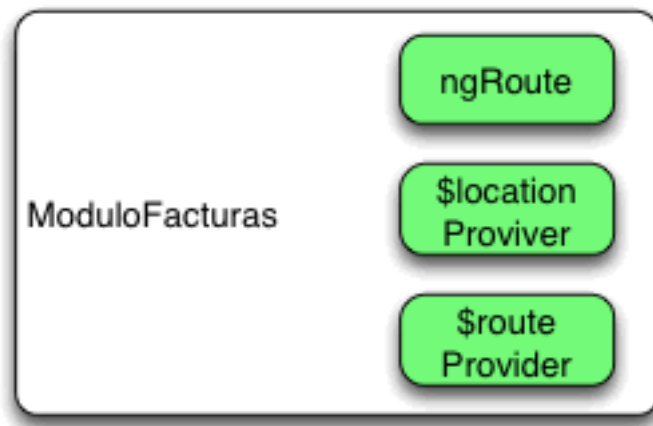
Vamos a mostrar el código fuente y comentarlo en detalle:

```
angular.module('moduloFacturas',['ngRoute'])  
.constant('baseUrl', 'http://localhost:3000/')  
.config(function($routeProvider,$locationProvider) {  
  
$locationProvider.html5Mode(true);  
  
});
```

El código no es tan complejo como pueda parecer al principio. Tenemos dos nuevos conceptos que explicar. El primero de ellos es el concepto de **Constante**, que hace referencia a un valor no modificable de la aplicación. En nuestro caso se define la Url del servidor de donde nos vamos a bajar la aplicación.

```
constant('baseUrl', 'http://localhost:3000/')
```

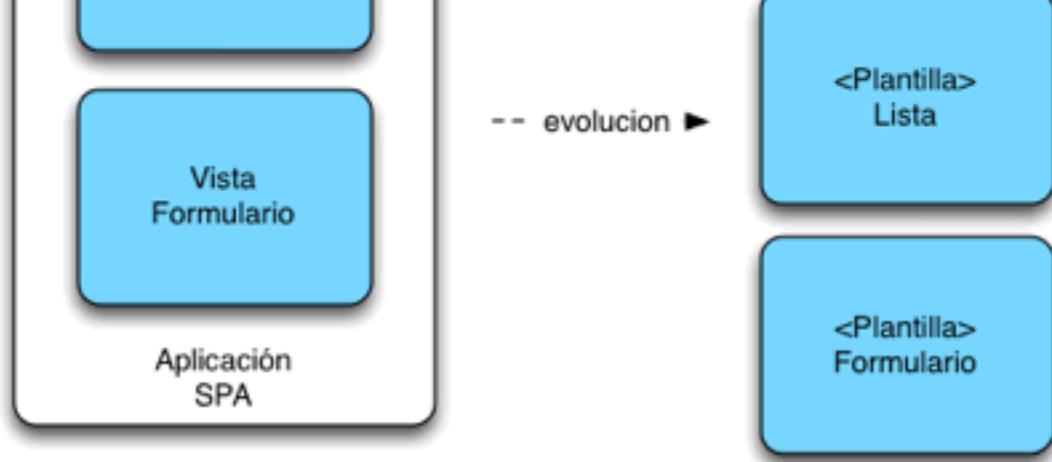
El segundo concepto es la parte de configuración que se ejecuta cuando el framework se inicializa. En nuestro caso estamos inyectando dos objetos \$locationProvider y \$routeProvider.



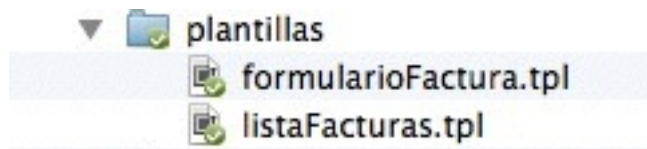
El \$locaciónProvider se encarga de establecer el modo en las que las URL del navegador se generan. En este caso, apoyándose en HTML5 de tal forma que, cuando cambiemos de Vista, la URL que muestra el navegador cambie también.

## Plantillas y \$routeProvider

El proveedor de rutas es otro servicio que nos permitirá cambiar dinámicamente la vista que tenemos seleccionada. Para poder utilizar el proveedor de rutas, primero deberemos partir nuestra aplicación SPA en varias vistas, cada una almacenada en un fichero de plantilla.



Para ello vamos a crear una carpeta de plantillas y ubicar en ella nuestras dos vistas a las que denominaremos (**listaFactura.tpl** y **formularioFactura.tpl**)



## La directiva ng-view

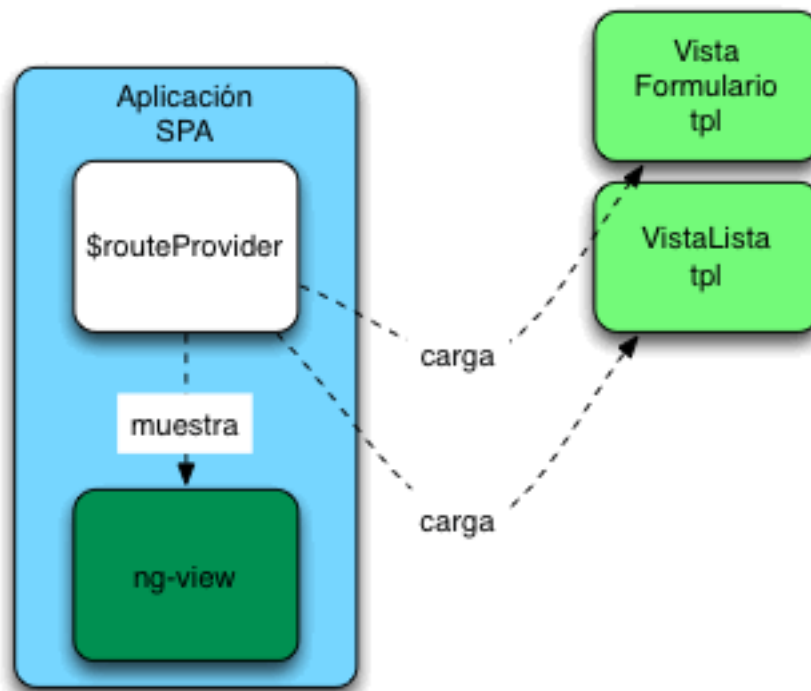
Realizada esta operación, vamos a usar una nueva directiva que se denomina ng-view para cargar una plantilla u otra en la página de forma dinámica. Para ello eliminaremos todo el contenido HTML que teníamos y **dejaremos simplemente un div con la directiva ng-view.**



Vamos a verlo en código :

```
<body ng-controller="controladorFacturas">  
  <div ng-view></div>  
</div>  
</body>
```

El sistema de enrutado será el encargado de cargar una vista u otra dependiendo de lo que nosotros solicitemos.



El código usa el método when del \$routeProvider para cargar una plantilla u otra:

```
.config(function($routeProvider,$locationProvider) {  
  $locationProvider.html5Mode(true);  
  $routeProvider.when("/vistaListaFacturas",{
```

```
        templateUrl: "/plantillas/listaFacturas.tpl"

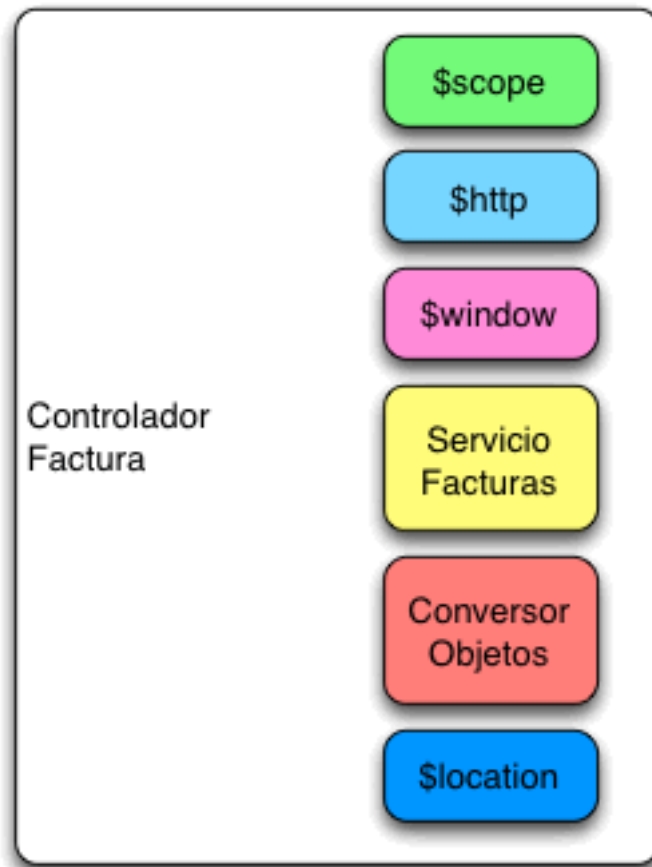
    });
$routeProvider.when("/vistaFormularioFacturas",{

    templateUrl: "/plantillas/formularioFactura.tpl"
});
$routeProvider.otherwise({

    templateUrl: "/plantillas/listaFacturas.tpl"
});
});
```

## El servicio \$location

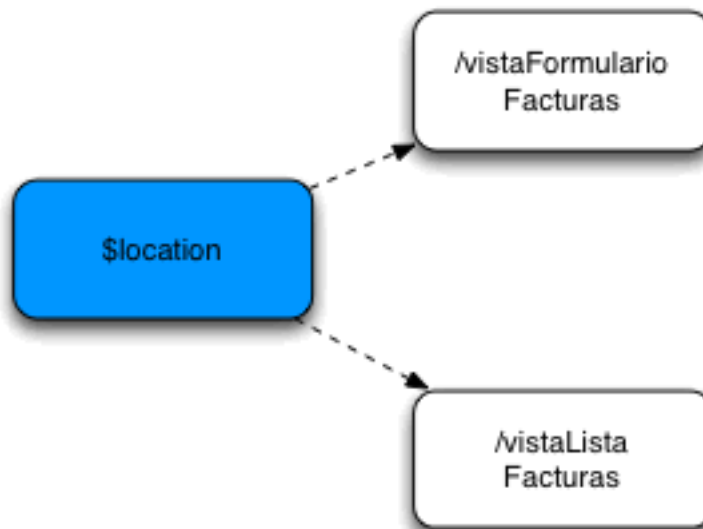
Una vez realizada esta operación, nos queda ver como nos cambiamos de vista utilizando los botones de la aplicación. Para ello el primer paso es inyectar el servicio de localización a nivel de nuestro controlador



Vamos a ver el código:

```
controller("controladorFacturas",  
    function($scope,$http,$window,servicioFacturas,conversorObjetos,  
$location) {  
  
    .....  
  
}
```

El siguiente paso es utilizar el servicio de **\$location** para cambiar de vista según vamos pulsando los diferentes botones.



Vamos a mostrar los bloques de código que varían:

```
$scope.verFormulario=function() {  
  
    $location.path("/vistaFormularioFacturas");  
  
}  
$scope.addFactura=function() {  
  
    console.log($scope.nuevaFactura);  
    var peticion= servicioFacturas.addFactura($scope.nuevaFactura);  
  
    peticion.success(function(factura) {  
  
        $scope.listaFacturas.push(new Factura(factura));  
        $scope.vista="lista";  
        $location.path("/vistaListaFacturas");  
    });  
}
```

Como podemos ver, es muy sencillo de manejar: simplemente usamos el servicio de `$location` con su método `path` y cambiamos la URL según nuestras necesidades :

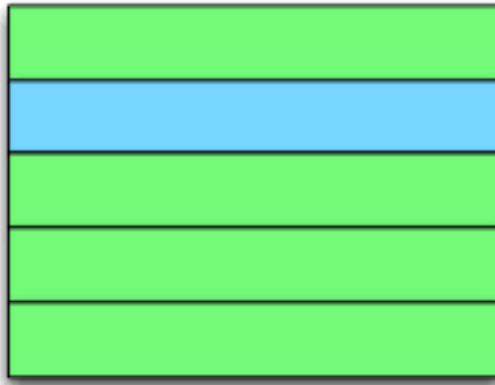
```
$location.path("/vistaListaFacturas");
```

A partir de este momento, la aplicación tendrá una estructura SPA, pero cargará todos los contenidos de forma dinámica.



# Angular Directivas

Uno de los conceptos que más diferencia a Angular.js de otros frameworks es el concepto de Directiva. En muchas ocasiones cuando programamos una web, nos encontramos con la necesidad de cambiar contenido de forma dinámica. Por ejemplo, podemos tener una tabla HTML y queremos cambiar dinámicamente el color de una celda cuando pasamos por encima.



Normalmente este efecto lo solemos implementar con jQuery y el evento hover de la siguiente forma o similar:

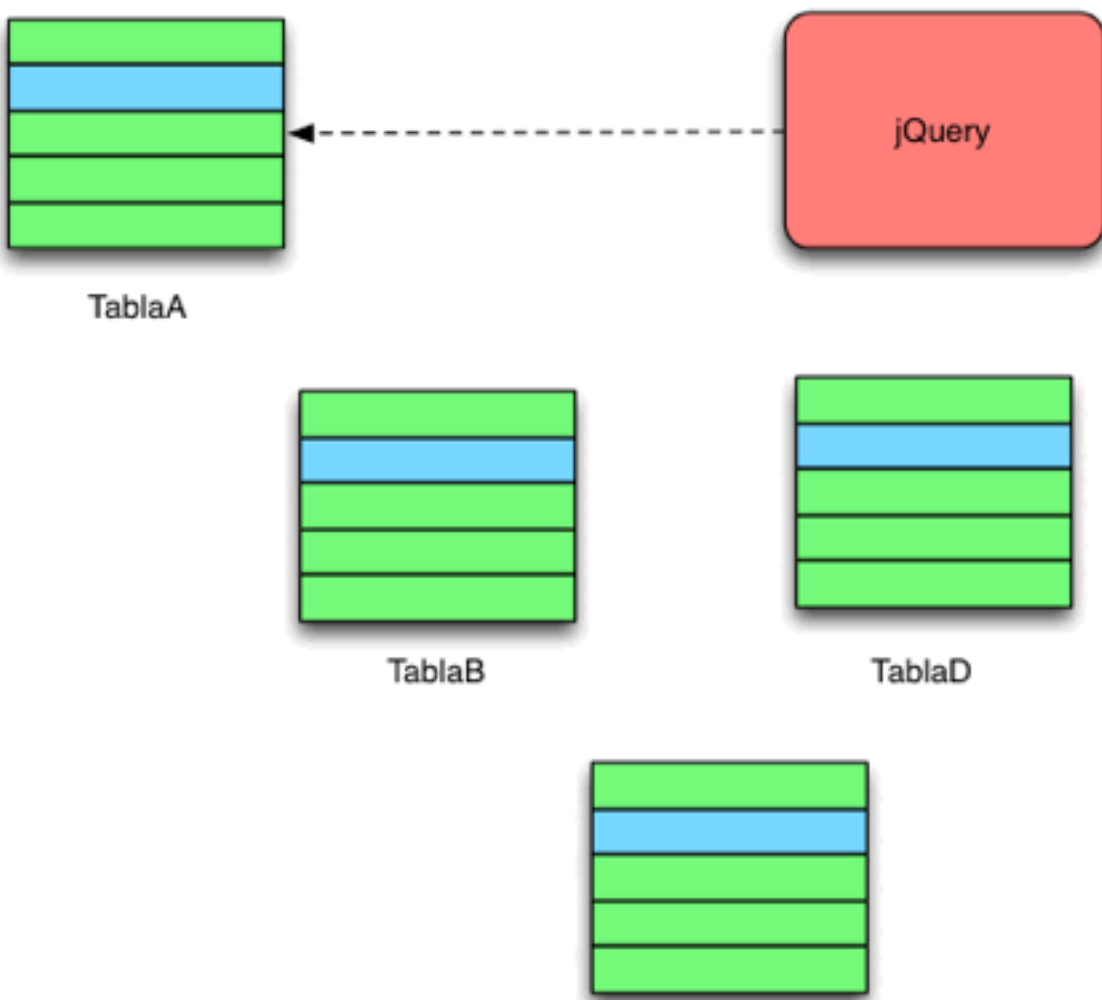
```
$( "tablaA tr" ).hover(  
  function() {  
    $( this ).css("background-color","lightblue");  
  }, function() {  
    $( this ).css("background-color","");  
  });
```

El resultado será el siguiente:



## Reutilización de código

Aunque este tipo de efectos quedan muy bien y se construyen con JQuery de una forma muy rápida, tienen un problema muy importante. JQuery está muy centrado en la capa de presentación y muy ligado a ella.



En nuestro caso, este efecto está ligado a la “tablaA”, sin embargo tendremos muchas tablas en nuestra aplicación según ésta crezca (Arquitectura SPA). Así pues, no es tan fácil reutilizar el código de jQuery como nosotros queremos. Ahora mismo afecta sólo a una tabla, podemos modificarlo para que quizás afecte a todas , pero no es tan fácil adaptarlo mucho más allá de esto. Tenemos un problema con el principio DRY (Dont Repeat Yourself).

## Creación de directivas

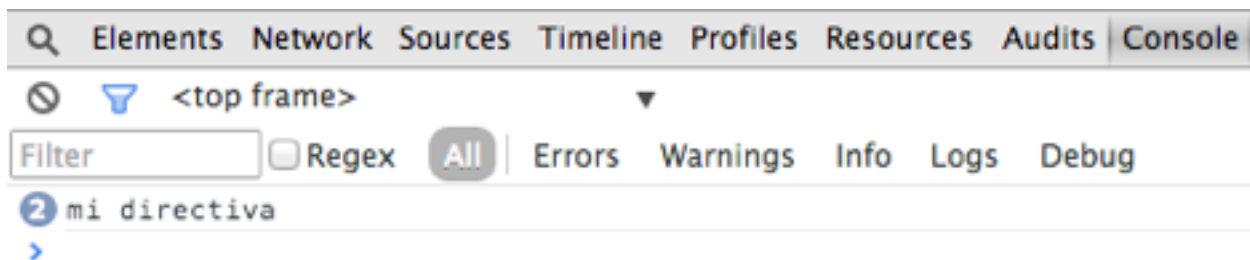
Angular soluciona este problema ya que nos permite crear nuestras propias directivas a nivel de código que luego podremos reutilizar a voluntad. Crear una directiva es bastante sencillo **pero convertirse en experto es diferente**. Vamos a crear un par de directivas elementales para explicar el concepto. La primera es una directiva que simplemente saca un mensaje por consola cuando la aplicamos:

```
.directive("mensaje",function() {  
  
    return {  
  
        link: function(scope,elem,attr) {  
  
            console.log("mi directiva");  
        }  
    }  
  
})
```

Las directivas tienen una función de link que se ejecuta cuando dicha directiva se procesa. Esta función recibe tres parámetros el scope en el cual se ejecuta, el elemento en el que nos encontramos y los atributos que este elemento tiene. En nuestro primer ejemplo no vamos a hacer uso de ninguno de los parámetros simplemente vamos a usar la directiva para imprimir un mensaje por la consola. Para ello tendremos que modificar nuestro código HTML y añadir la directiva:

```
<div>
<table >
<tr ng-repeat="factura in listaFacturas" mensaje>
<td>{{factura.id}}</td>
<td>{{factura.concepto}}</td>
<td>{{factura.importe}}</td>
<td>{{factura.calcularIva()}}</td>
<td><input type="button" ng-click="removeFactura(factura)" value="Borrar"/></td>
</tr>
</table>
<input type="button" ng-click="verFormulario()" value="nuevo"/>
</div>
```

Hecho esto, la directiva se procesará y por cada Factura nos imprimirá un mensaje por la consola.



Acabamos de crear la directiva más sencilla.

## Directiva Iluminar

Es evidente que la directiva que acabamos de crear no sirve para más que clarificar el concepto. Vamos a construir una nueva directiva que implemente la misma funcionalidad habida en el ejemplo inicial de jQuery (ilumina la fila) . Llamaremos a esta directiva “iluminar”:

```

.directive("iluminar",function() {

    return {

        link: function(scope,elem,attr) {

            elem.bind('mouseenter',function(evento) {

                elem.css("background-color","lightblue");

            });

            elem.bind('mouseleave',function(evento) {

                elem.css("background-color","");

            });

        }

    }

});

```

El código es bastante sencillo de entender ya que Angular usa una versión reducida de jQuery que se denomina jQLite para realizar las modificaciones en la capa de presentación. En este caso usamos la función de `bind()` para crear dos eventos **“mouseenter”** y **“mouseleave”**, que se encargarán de cambiar el color de la fila por la cual pasemos el ratón. Evidentemente para que esto funcione necesitamos aplicar la nueva directiva:

```

<tr ng-repeat="factura in listaFacturas" iluminar>
<td>{{factura.id}}</td>
<td>{{factura.concepto}}</td>
<td>{{factura.importe}}</td>
<td>{{factura.calcularIva()}}</td>
<td><input type="button" ng-click="removeFactura(factura)" value="Borrar"/>
</td>
</tr>

```

```

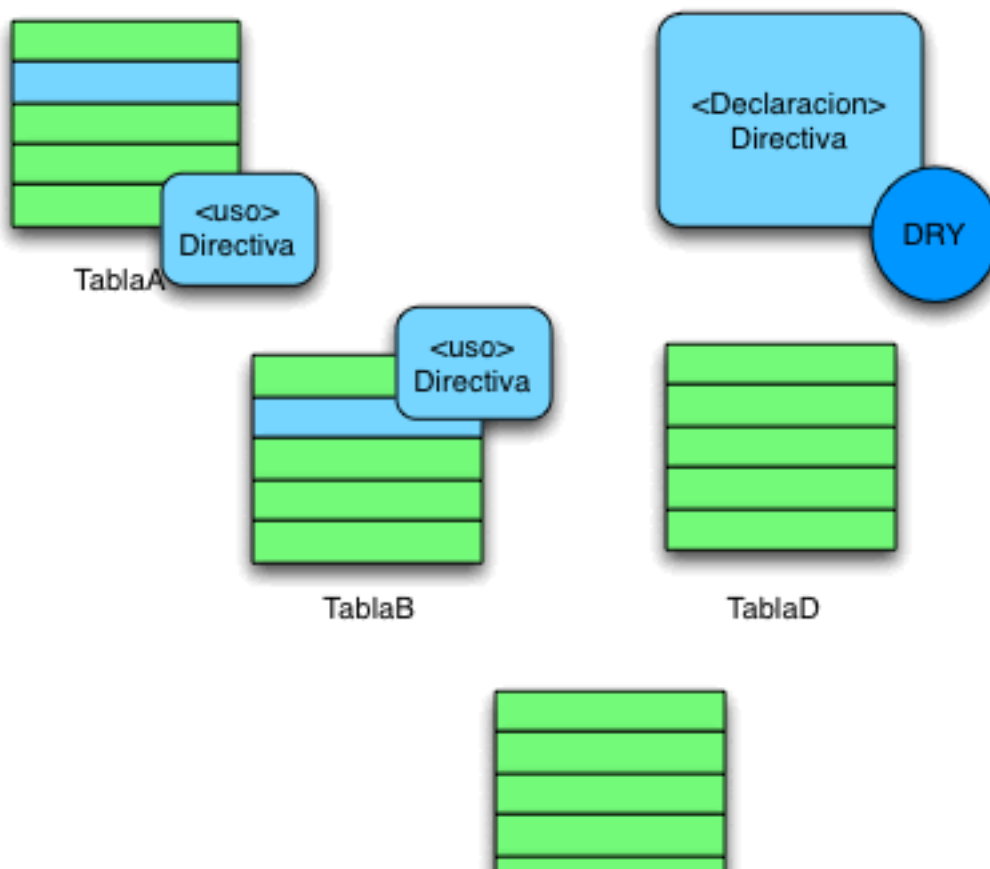
</table>
<input type="button" ng-click="verFormulario()" value="nuevo"/>
</div>

```

El resultado a nivel de capa de presentación será idéntico al de jQuery:



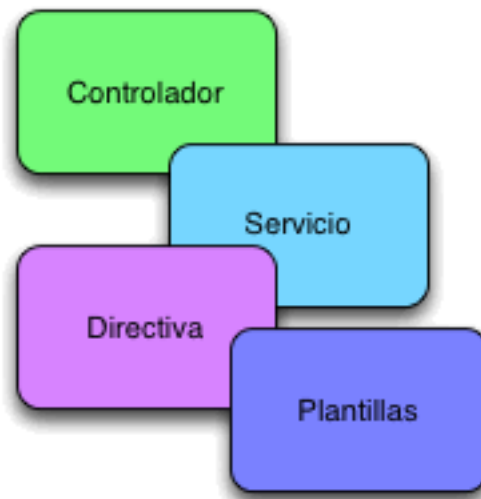
La ventaja que tenemos ahora es que la directiva se ha definido una vez y la podemos aplicar a conveniencia en los bloques de código que deseemos, a diferencia de JQuery que se aplica a un conjunto de nodos determinados



Las directivas son una parte fundamental de Angular.js y abren unas posibilidades de extensibilidad gigantescas. Son también uno de los conceptos más difíciles de abordar y de convertirse en experto.

# Angular y Organización

Hemos tratado en los anteriores capítulos muchos conceptos y ahora dichos conceptos están agrupados. Es momento de organizar un poco más el código de tal forma que nos sea más sencillo trabajar.



Para ello vamos a construir la siguiente estructura de carpetas diseñada de forma que cada tipo de responsabilidad esté separada (esta es una estructura muy básica ya que tenemos poco código)

## Aplicación.html

En primer lugar tenemos el fichero aplicación.html que va a quedar prácticamente vacío a nivel de contenido, ya que delegará en los otros para cargar los ficheros según la necesidad.



```
<html ng-app="moduloFacturas">
<head>
<script type="text/javascript" src="script/factura.js">
</script>
<script type="text/javascript" src="script/angular.min.js">
</script>
<script type="text/javascript" src="script/angular-route.js">
</script>

<script type="text/javascript">

var modulo = angular.module('moduloFacturas',['ngRoute']);

</script>

<script src="rutas/rutas.js" type="text/javascript">
</script>
<script src="directivas/directivas.js" type="text/javascript">
</script>
<script src="servicios/servicios.js" type="text/javascript">
</script>
<script src="controladores/controladores.js" type="text/javascript">
</script>
</head>
<body ng-controller="controladorFacturas">
<div ng-view></div>
</div>
</body>
</html>
```

Una vez tenemos claro como queda este fichero, lo vamos a revisar línea a línea.

## Carpeta Script

Esta carpeta contendrá todos los ficheros .js que necesitamos para inicializar Angular pero que no corresponden a la aplicación en sí.

```
<script type="text/javascript" src="script/factura.js">
</script>
<script type="text/javascript" src="script/angular.min.js">
```

```
</script>  
<script type="text/javascript" src="script/angular-route.js">  
</script>
```

## Carpeta Rutas

Esta carpeta contendrá el sistema de rutas de la aplicación. En nuestro caso se trata de un sencillo fichero rutas.js con el siguiente código:

```
modulo.constant('baseUrl', 'http://localhost:3000/')  
.config(function($routeProvider,$locationProvider) {  
  
    $locationProvider.html5Mode(true);  
  
    $routeProvider.when("/vistaListaFacturas",{  
  
        templateUrl:"/plantillas/listaFacturas.tpl"  
  
    });  
    $routeProvider.when("/vistaFormularioFacturas",{  
  
        templateUrl:"/plantillas/formularioFactura.tpl"  
    });  
    $routeProvider.otherwise({  
  
        templateUrl:"/plantillas/listaFacturas.tpl"  
    });  
  
});
```

## Carpeta Directivas

Esta carpeta puede que termine conteniendo muchas directivas. En nuestro caso solo contiene una ya que el ejemplo era muy sencillo.

```
modulo.directive("iluminar",function() {  
  
    return {
```

```
        link: function(scope,elem,attr) {

            elem.bind('mouseenter',function(evento) {

                elem.css("background-color","lightblue");

            });

            elem.bind('mouseleave',function(evento) {

                elem.css("background-color","");

            });

        }

    });
```

## Carpeta Controladores

Esta carpeta contendrá el controlador con el que hemos estado trabajando durante todo el libro y cuyo código es el más extenso.

```
modulo.controller("controladorFacturas",
    function($scope,$http,$window,servicioFacturas,conversorObjetos,
    $location) {

        $scope.nuevaFactura={};

        var peticion= servicioFacturas.listaFacturas();

        peticion.success(function(listaFacturas) {

            $scope.listaFacturas=conversorObjetos.convertirFacturas(listaFacturas);

        });

        $scope.addFactura=function() {

            console.log($scope.nuevaFactura);
```

```
var peticion= servicioFacturas.addFactura($scope.nuevaFactura);

peticion.success(function(factura) {

    $scope.listaFacturas.push(new Factura(factura));
    $scope.vista="lista";
    $location.path("/vistaListaFacturas");
});
}

$scope.removeFactura=function(factura) {

    if($window.confirm("deseas borrar la factura con el
concepto:"+factura.concepto)) {

        var peticion= servicioFacturas.removeFactura(factura);

        peticion.success(function(id) {

            $scope.listaFacturas=$scope.listaFacturas.filter(function
(factura) {

                return id!==factura.id;
            });

        });
    }
}

$scope.verFormulario=function() {

    $location.path("/vistaFormularioFacturas");

}

})
```

## Carpeta de Servicios

Esta carpeta almacena los servicios que, en nuestro caso, contiene tanto el servicioFacturas como el conversorObjetos.

```
modulo.service("servicioFacturas", function($http){

    this.listaFacturas=function() {

        return $http.get("facturas");
    }

    this.addFactura=function(factura) {

        return $http.post('/facturas',factura);

    }
    this.removeFactura=function(factura) {

        return $http.delete('/facturas/id/'+factura.id)

    }

}).service("conversorObjetos", function(){

    this.convertirFacturas=function(listaFacturas) {

        var listaObjetosFacturas=[];

        for (var i=0;i<listaFacturas.length;i++) {

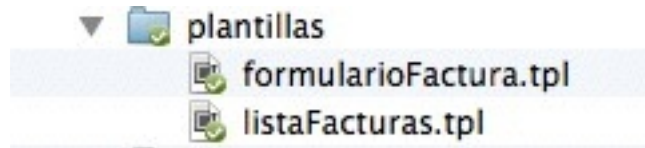
            listaObjetosFacturas.push(new Factura(listaFacturas[i]));

        }
        return listaObjetosFacturas;
    }

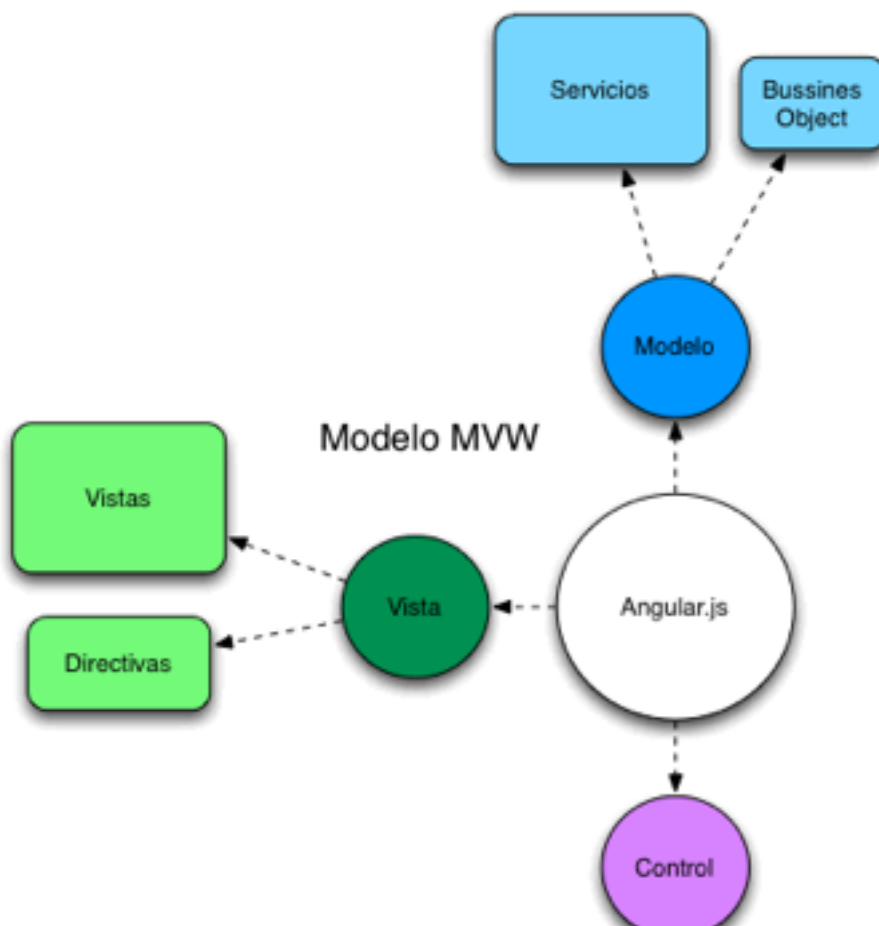
});
```

## Carpeta de Plantillas

Esta carpeta se encarga de almacenar las plantillas con las distintas vistas que Angular.js soporta.



Hemos terminado de organizar la aplicación de Angular.js de una forma mucho más modular(existen muchas variaciones). De esta forma podemos ver de una manera mucho más clara cómo se dividen las responsabilidades en una Aplicación Angular y cómo cada uno de los conceptos ocupa su lugar dentro de las típicas arquitecturas MVC.



A este modelo que tiene tantas opciones en cuanto a la gestión de las capas se le suele denominar MVW (Model View WhatEver), ya que Angular soporta una gran flexibilidad en cuanto al patrón MVC se refiere.

## Conclusiones

Han quedado muchas cuestiones que tratar de Angular.js tales como filtros, resources, watches, promesas, validaciones , directivas complejas etc. Angular es un universo nuevo que ha venido para quedarse, junto con los nuevos frameworks de Javascript como Backbone, Ember o Meteor y se hará en los próximos años un hueco importante en la forma que tenemos de programar aplicaciones web. Tenemos que irlos conociendo paulatinamente e integrar estos nuevos conceptos en nuestras futuras arquitecturas. Espero que esta introducción ayude a aclarar dudas.

Puedes ponerte en contacto conmigo vía :

Correo:

[contacto@arquitecturajava.com](mailto:contacto@arquitecturajava.com)

LinkedIn

[https://www.linkedin.com/profile/view?id=2949192&trk=tab\\_pro](https://www.linkedin.com/profile/view?id=2949192&trk=tab_pro)

Twitter

<https://twitter.com/arquitectojava>



# Bibliografia

Desarrollo Agil con Angular.js (Carlos Azaustre)

Pro Angular JS (Adam Freeman)

Mastering Web Application with AngularJS (Pawel Kozlowski)