https://angularjs.org/

EVOLUCIÓN DEL DESARROLLO WEB

1990

Cliente

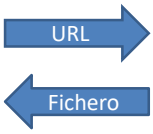
Tim Berners-Lee (CERN)

- ENQUIRE
  - Hipertexto
  - HTTP
  - HTML

Servidor

Robert Cailliau

- CERN httpd
  - Servidor de ficheros
  - HTTP – URL
  - Ficheros estáticos de texto



Origen

- La Web no fue concebida para el desarrollo de aplicaciones. El problema que se pretendía resolver su inventor, Tim Berners-Lee, era el cómo organizar información a través de enlaces.
- De hecho la Web nació en el laboratorio de partículas CERN básicamente para agrupar un conjunto muy grande de información y datos del acelerador de partículas que se encontraba muy dispersa y aislada.
- Mediante un protocolo muy simple (HTTP), un sistema de localización de recursos (URL) y un lenguaje de marcas (HTML) se podía poner a disposición de todo científico en el mundo la información existente en el CERN de tal forma que mediante enlaces se pudiese acceder a información relacionada con la consultada.

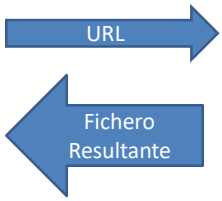
HTML

Cliente

- Navegadores
  - HTML

Servidor

- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
    - Texto
    - HTML
  - Ficheros dinámicos
    - CGI (C, Perl)
    - …
    - Servlet



CGI

- Por la necesidad que el servidor Web pudiese devolver páginas Web dinámicas y no únicamente contenido estático residente en ficheros HTML se desarrolló la tecnología CGI (Common Gateway Interface) donde el servidor Web invocaba un programa el cual se ejecutaba, devolvía la página Web y el servidor Web remitía este flujo de datos al navegador.
- Un programa CGI podía ser cualquier programa que la máquina pudiese ejecutar: un programa en C, o en Visual Basic o en Perl. Normalmente se elegía este último por ser un lenguaje de script el cual podía ser traslado con facilidad de una arquitectura a otra. CGI era únicamente una pasarela que comunicaba el servidor Web con el ejecutable que devolvía la página Web.

Formularios

Cliente

- Navegadores
  - HTML 3.2
  - Formularios
  - Plug-in
    - Applet
    - ActiveX
  - Scripting de cliente
    - JavaScript

Servidor

- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
    - Texto, HTML, Imágenes, …
  - Ficheros dinámicos
  - Sricpting de servidor
    - ASP
    - PHP
    - …
    - JSP

Scripting

- CGI era una solución cómoda de realizar páginas Web dinámicas pero tenía un grave problema de rendimiento que lo hizo insostenible en cuanto la demanda de la Web comenzó a disparar las peticiones de los servidores Web.
- Para agilizar esto, los principales servidores Web del momento (Netscape e IIS) desarrollaron un sistema para la ejecución dinámica de aplicaciones usando el propio contexto del servidor Web. En el caso de Netscape se le denominó NSAPI (Netscape Server Application Program Interface) y en el caso de IIS se le llamó ISAPI.

Web 2.0

Cliente

- Navegadores
  - HTML 4
  - CSS
  - DOM
  - AJAX
  - RIA
    - Shockwave Flash
    - Silverlight
  - JS Framework

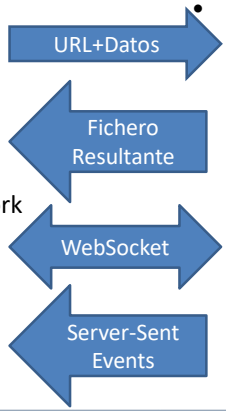
Servidor

- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
  - Ficheros dinámicos
  - Sricpting de servidor
  - Servidor de aplicaciones
    - ASP.NET
    - J2EE
  - Web Services
    - WS XML
    - RestFul

Actualidad

Cliente

- Navegadores
  - HTML 5
  - CSS 3
  - ~~RIA~~
  - Móviles
  - JS RIA Framework
  - EcmaScript 6



Servidor

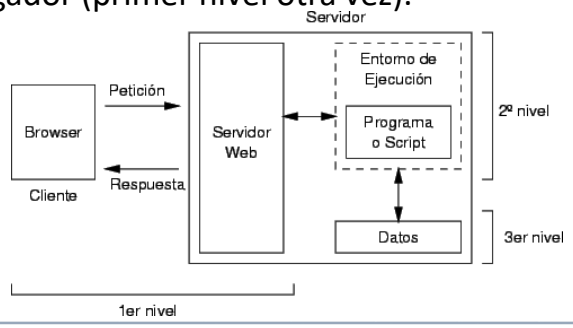
- Servidor Web
  - Servidor de ficheros
  - Ficheros estáticos
  - Ficheros dinámicos
  - Sricpting de servidor
  - Servidor de aplicaciones
    - NodeJS
  - Web Services
  - Server-Sent Events
    - Notificaciones PUSH

Aplicación Web

- Conjunto de páginas que residen en un directorio web y sus subdirectorios.
- Cualquier página puede ser el punto de entrada de la aplicación, no se debe confundir con el concepto de página principal con el de página por defecto.
- Externamente, no existe el concepto de aplicación como entidad única.
- Internamente, dependiendo de la tecnología utilizada una aplicación puede ser una entidad única o una colección de objetos independientes.

Arquitectura

- Una aplicación Web típica recogerá datos del usuario (primer nivel), los enviará al servidor, que ejecutará un programa (segundo y tercer nivel) y cuyo resultado será formateado y presentado al usuario en el navegador (primer nivel otra vez).



Ventajas e inconvenientes

Ventajas

- Inmediatez y accesibilidad
- No ocupan espacio local
- Actualizaciones inmediatas
- No hay problemas de compatibilidad
- Multiplataforma
- Consumo de recursos bajo
- Portables
- Alta escalabilidad y disponibilidad

Inconvenientes

- Interfaz de interacción con el usuario muy limitada
- Bese tecnológica inadecuada
- Incompatibilidades entre navegadores
- Bajo rendimiento al ser interpretado
- Cesión tecnológica
- Seguridad menos robusta

Single-page application (SPA)

- Un single-page application (SPA), o aplicación de página única es una aplicación web o es un sitio web que utiliza una sola página con el propósito de dar una experiencia más fluida a los usuarios como una aplicación de escritorio.
- En un SPA todo el códigos de HTML, JavaScript y CSS se carga de una sola vez o los recursos necesarios se cargan dinámicamente cuando lo requiera la página y se van agregando, normalmente como respuesta de los acciones del usuario.
- La página no se tiene que cargar otra vez en ningún punto del proceso, tampoco se transfiere a otra página, aunque las tecnologías modernas (como el pushState() API del HTML5) pueden permitir la navegabilidad en páginas lógicas dentro de la aplicación.
- La interacción con las aplicaciones de página única pueden involucrar comunicaciones dinámicas con el servidor web que está por detrás, habitualmente utilizando AJAX o WebSocket (HTML5).

Estado actual de la adopción de los nuevos estándares

- http://caniuse.com/
- HTML 5
  - http://html5test.com
  - http://html5demos.com
- CSS
  - http://css3test.com/
- EcmaScript
  - https://kangax.github.io/compat-table/es6/
- Navegadores mas utilizados
  - http://www.netmarketshare.com/

INTRODUCCIÓN A ANGULARJS

Introducción

- JQuery realmente no tiene un mayor nivel de abstracción que el propio JavaScript estándar del navegador.
- JQuery debería haber sido realmente el API nativo de los navegadores y no una librería que pretender cubrir los huecos dejados por lo navegadores como las incompatibilidades entre ellos.
- AngularJS es un framework que pretende definir la arquitectura de la aplicación.
- Define claramente la separación entre el modelo de datos, la vista y el controlador.
- Estás obligado por AngularJS a seguir esa estructura. Ya no haces el JavaScript como tu quieres sino como AngularJS te manda. Y eso realmente da mucha potencia a la aplicación ya que, al seguir una arquitectura definida, AngularJS puede ayudarte en la aplicación y a tener todo mucho mas estructurado.

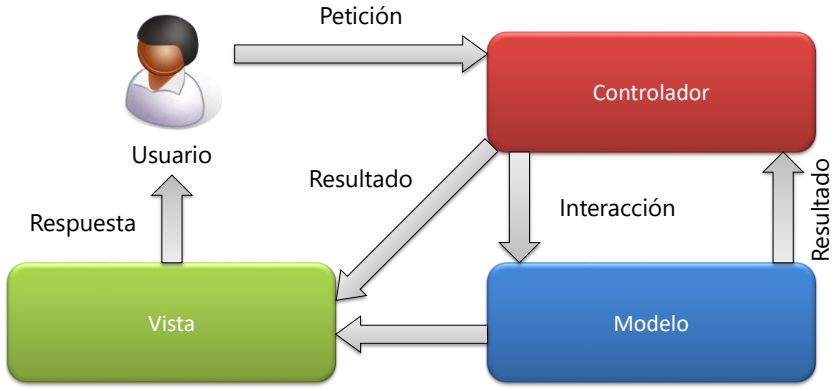
¿Qué es Angular?

- AngularJS es un framework estructural para aplicaciones web dinámicas siguiendo el patrón MVC (MVW).
- Permite utilizar el HTML como lenguaje de plantillas y extender la sintaxis del HTML para expresar los componentes de la aplicación de forma declarativa, clara y sucinta .
- El enlazado de datos y la inyección de dependencias eliminan gran parte del código que de otra forma se tendría que escribir. Y todo sucede dentro del navegador, lo que lo convierte en un buen socio para cualquier tecnología de servidor.
- Angular es lo habría sido el HTML, de haber sido diseñado para aplicaciones. HTML es un gran lenguaje declarativo para documentos estáticos. No contiene mucho para la creación de aplicaciones, como resultado de ello, la creación de aplicaciones web es un continuo ejercicio de engañar al navegador para que así haga lo que se desea.

El patrón MVC

- El Modelo Vista Controlador (MVC) es un patrón de arquitectura de software (presentación) que separa los datos y la lógica de negocio de una aplicación del interfaz de usuario y del módulo encargado de gestionar los eventos y las comunicaciones.
- Este patrón de diseño se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones, prueba y su posterior mantenimiento.
- Para todo tipo de sistemas (Escritorio, Web, Movil, …) y de tecnologías (Java, Ruby, Python, Perl, Flex, SmallTalk, .Net …)

El patrón MVC



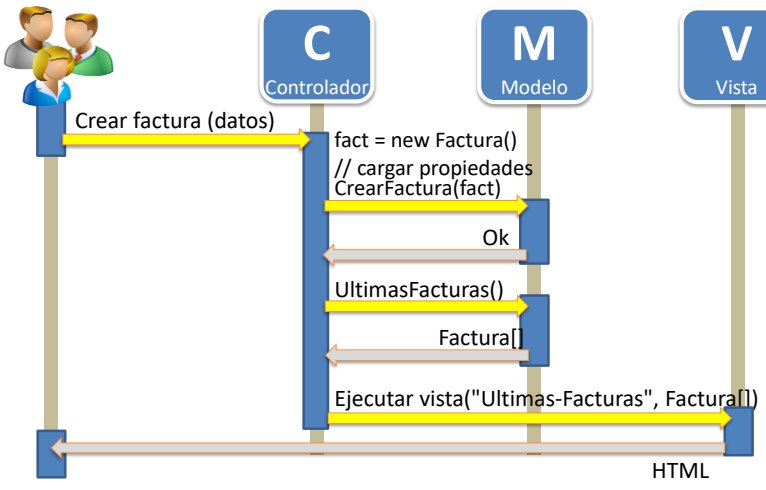
El patrón MVC

- Representación de los **datos del dominio**
- Lógica de **negocio**
- Mecanismos de **persistencia**

- **Interfaz** de usuario
- Incluye elementos de **interacción**

- **Intermediario** entre Modelo y Vista
- **Mapea acciones** de usuario → acciones del Modelo
- **Selecciona** las vistas y les **suministra** información

El patrón MVC



Componentes principales en MVC

- Enrutado:
  - Tablas de Rutas
- Model:
  - Lógica de negocio y Entidades de Dominio
- Controllers:
  - Lógica de presentación
- Views:
  - Modelo de presentación, Modelo de estado

Modelo, Vista, Controlador

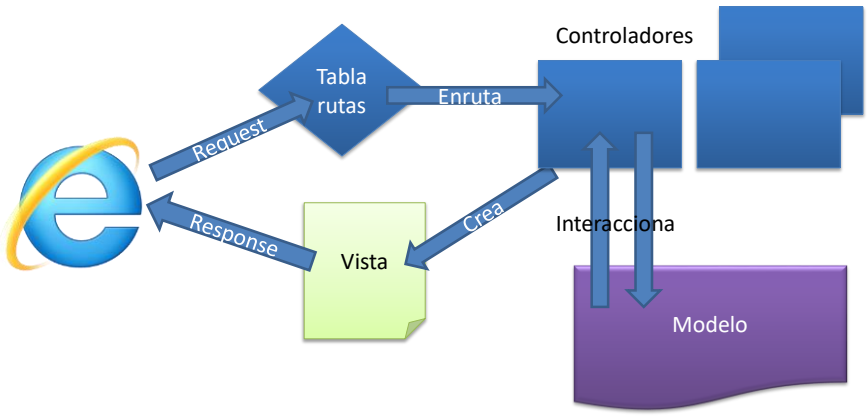


Tabla de rutas

- Dada una URL decide qué acción de qué controlador procesa esta acción
- Sólo tiene en cuenta la URL (nada de parámetros POST, query string, …)
- Ruta por defecto:
  - /{controller}/{action}/{id}
- Ruta con Angular:
  - #{controller}/{action}/{id}

Controladores

- Los controladores son los componentes que controlan la interacción del usuario, trabajan con el modelo y por último seleccionan una vista para representar la interfaz de usuario.
- Exponen **acciones** que se encargan de procesar las peticiones
- Cada acción debe devolver un resultado, que es algo que el framework debe hacer (mandar una vista, un fichero binario, un 404, …)
- Hablan con el modelo pero son «tontos»

Modelo

- Los objetos de modelo son las partes de la aplicación que implementan la lógica del dominio de datos de la aplicación.
- A menudo, los objetos de modelo recuperan y almacenan el estado del modelo en una base de datos.
- Encapsula toda la lógica de nuestra aplicación
- Responde a peticiones de los controladores

Vistas

- Las vistas son los componentes que muestra la interfaz de usuario de la aplicación.
- Normalmente, esta interfaz de usuario se crea a partir de los datos de modelo.
- Se encarga únicamente de temas de presentación.
- Es «básicamente» código HTML (con un poco de server-side)
- NO acceden a BBDD, NO toman decisiones, NO hacen nada de nada salvo…
- … mostrar información

Características

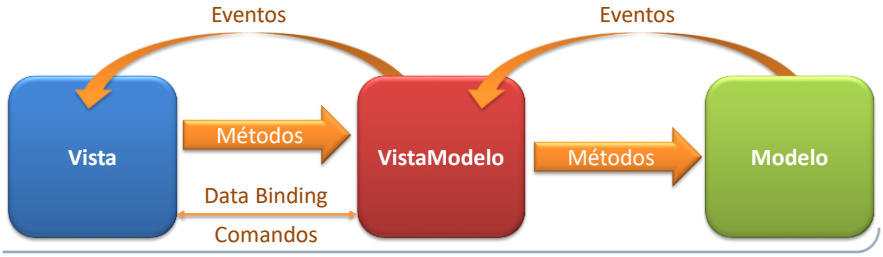
- El modelo MVC ayuda a crear aplicaciones que separan los diferentes aspectos de la aplicación (lógica de entrada, lógica de negocios y lógica de la interfaz de usuario), a la vez que proporciona un vago acoplamiento entre estos elementos.
- El modelo especifica dónde se debería encontrar cada tipo de lógica en la aplicación.
  - La lógica de la interfaz de usuario pertenece a la vista.
  - La lógica de entrada pertenece al controlador.
  - La lógica de negocios pertenece al modelo.
- Esta separación ayuda a administrar la complejidad al compilar una aplicación, ya que le permite centrarse en cada momento en un único aspecto de la implementación.
- El acoplamiento vago entre los tres componentes principales de una aplicación MVC también favorece el desarrollo paralelo.

Puntos fuertes de MVC…

- Puntos fuertes de MVC…
  - Modelo muy simple de entender
  - Modelo muy cercano a la web
  - Admite una buena separación de responsabilidades
  - Permite la automatización de las pruebas unitarias de modelos y controladores.
- … y no tan fuertes
  - Curva de aprendizaje más alta

Model View ViewModel (MVVM)

- El **Modelo** es la entidad que representa el concepto de negocio.
- La **Vista** es la representación gráfica del control o un conjunto de controles que muestran el Modelo de datos en pantalla.
- La **VistaModelo** es la que une todo. Contiene la lógica del interfaz de usuario, los comandos, los eventos y una referencia al Modelo.



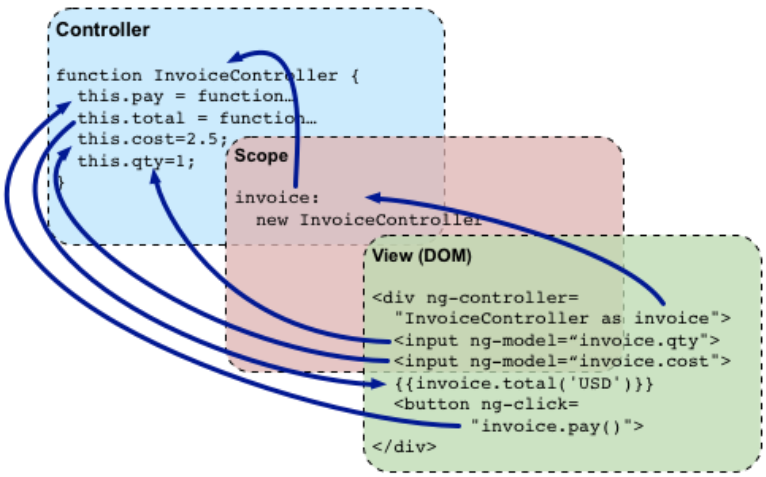
Model-View-Whatever (MVW)

- Angular ha sido re-definido como un framework Model-View-Whatever, es decir, Modelo-Vista-Lo-que-sea.
- Pretende de esta forma huir de la discusión entre MVC vs MVP vs MVVM.
- La aplicación se organiza entorno a Controllers responsables de construir los ViewModels que se enlazarán a las vistas.
- El papel que juegan los Controllers es un tanto limitado, por lo que no se puedan comparar a los Controllers de la típica arquitectura web MVC donde se encargan gestionar las operaciones.
- En Angular, el Controller se limita a construir un ViewModel (Scope) que luego toma el control completo de las operaciones realizadas en la vista, por lo que básicamente actúan como factorías de ViewModels.

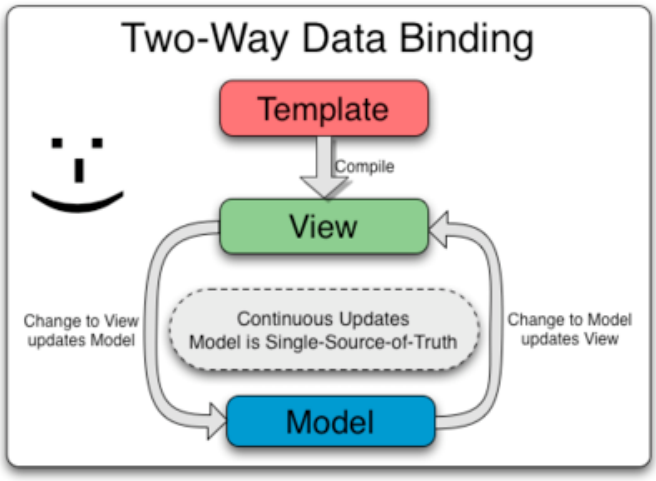
Conceptos

- **Model**: los datos mostrados al usuario en la vista y con que interactúa el usuario
- **View**: lo que ve el usuario (DOM)
  - **Template**: HTML con marcas adicionales
  - **Expressions**: las variables de acceso y funciones del scope
  - **Filter**: formatean el valor de una expresión para su visualización
  - **Directives**: extienden el HTML con atributos y elementos personalizados
- **Controller**: la lógica de negocio detrás vistas
- **Scope**: contexto en el que se almacena el modelo para que los controladores, las directivas y las expresiones puedan acceder a él
  - **Data Binding**: sincronización de datos entre el modelo y la vista
- **Module**: un contenedor para las diferentes partes de una aplicación, incluyendo controladores, servicios, filtros, directivas que configura el inyector
  - **Service**: lógica de negocio reutilizable independiente de las vistas
  - **Dependency Injection**: Crea y enlaza objetos y funciones
  - **Injector**: contenedor de inyección de dependencias
  - **Compiler**: analiza la plantilla e instancia las directivas y expresiones

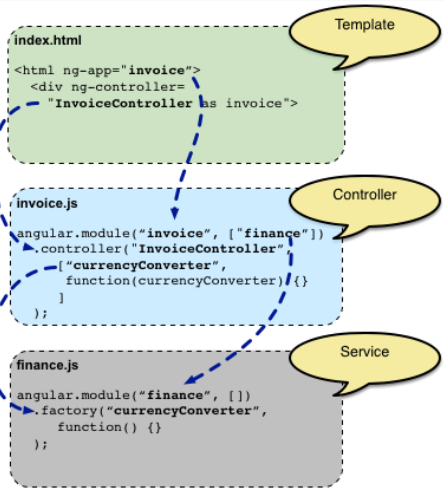
Scope



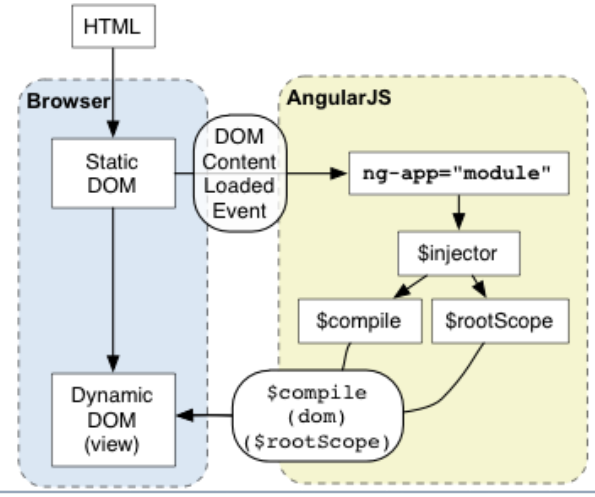
Data Binding



Services



Bootstrap



El objeto Angular

- El objeto Angular expone la funcionalidad básica del framework y suministra una serie de métodos de utilidad:
  - angular.module, angular.injector, angular.bind, angular.bootstrap,
  - angular.merge, angular.extend, angular.identity,
  - angular.isArray, angular.isDate, angular.isElement, angular.isFunction, angular.isNumber, angular.isObject, angular.isString
  - angular.isDefined, angular.isUndefined,
  - angular.fromJson, angular.toJson,
  - angular.forEach, angular.element, angular.equals, angular.copy,
  - angular.noop,
  - angular.reloadWithDebugInfo,

Estructura de ficheros

Estructural

- App
  - Index.html
  - CSS
    - App.css
  - Core
    - Angular.js
    - Jquery.js
  - Images
  - JS
    - app.js
    - Controladores
    - Servicios
  - Views
    - Controlador1
    - Controlador2

Funcional

- App
  - Index.html
  - CSS
    - App.css
  - Core
    - Angular.js
    - Jquery.js
  - Images
  - JS
    - app.js
  - Modulo1
    - Controlador1
      - Js
      - views
    - Controlador2

Añadir AngularJS a una página

- Se pueden utilizar dos estrategias diferentes:
  - Incluir ficheros locales (https://code.angularjs.org).
  - Incluir ficheros compartidos en una CDN (Content Delivery Network).
- Se pueden descargar los ficheros locales desde https://angularjs.org/:
  - Versión de producción: para servidores web se encuentra minimizada y compactada para reducir al máximo su tamaño.
  - Versión de desarrollo: para desarrollo y pruebas sin comprimir y depurable.

```
<head>
<script type="text/javascript" src="/js/angular.min.js"></script>
<script src="/js/i18n/angular-locale_es-es.js"></script>
</head>
```

- Las CDN permiten compartir contenidos comunes entre diferentes sitios y evitar descargas al aprovechar la cache de los navegadores:

```
<script type="text/javascript"
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.23/angular.min.js">
</script>
<script type="text/javascript" src="https://code.angularjs.org/1.2.23/i18n
/angular-locale_es-es.js"> </script>
```

- Es conveniente incluirlos después de las hojas de estilos para asegurar la importación de todos los estilos o al final del cuerpo para mejorar la percepción del usuario.

MÓDULOS

Module

- Un módulo es un contenedor para las diferentes partes de la aplicación: controladores, servicios, filtros, directivas, etc.
- Las aplicaciones Angular no tienen un método principal. De forma declarativa se especifica cual es el módulo de arranque de la aplicación. Las ventajas de este enfoque son:
  - El proceso declarativo es más fácil de entender.
  - Puede empaquetar el código como módulos reutilizables.
  - Los módulos pueden ser cargados en cualquier orden (o incluso en paralelo), porque los módulos retrasan la ejecución.
  - En las pruebas unitarias sólo se tienen que cargar los módulos pertinentes.
  - Las pruebas de extremo a extremo pueden utilizar módulos para sobrescribir la configuración.

Declaración

- Declarar:
```
<div ng-app="myApp">
```
- Implementar:
```
var myAppModule = angular.module('myApp', []);
myAppModule.filter('greet', function() {
 return function(name) {
    return 'Hello, ' + name + '!';
  };
});
angular.module('invoice', ['ngRoute','ngResource'])
.controller('InvoiceController', function() { … });
```
- Recuperar un módulo existente (getter):
```
var app = angular.module('myApp');
```


Bloques

```
angular.module('myModule', [injectables])
    .config(function(injectables) { … })
    .run(function(injectables) { … })
    .controller('controllerName', function() { … })
    .value('valueName', 123)
    .factory('factoryName', function() { … })
    .directive('directiveName', ...)
    .filter('filterName', ...)
    …;
```

Inyección de dependencias

- La inyección de dependencias quiere decir que a la hora de definir cualquier componente, ya sea un Controller, un Service u otra cosa, se debe indicar de qué otros componentes depende y Angular se encargará de proporcionárselos a través de la función constructora (o de la función factoría que lo construya, que para el caso es lo mismo).
- Todos los componentes que se registran en Angular se les asignan un nombre y son singleton, es decir, sólo existe una instancia de ellos en la aplicación y si hay varios componentes que dependen de un mismo objeto, todos recibirán la misma instancia del objeto.
- Esto es lo que permite utilizar fácilmente los servicios para almacenar estado, ya que dos Controllers que dependan de un mismo servicio estarán utilizando el mismo objeto y, por tanto, podrán compartir información a través de él.

```
app.controller("ConAJAXController",
   function($scope,$log,$http) {...})
```


Problemas con los minimificadores

- Dado que el JavaScript es interpretado, es necesario enviar el código fuente al cliente, lo que combinado con la creciente complejidad de las librerías y frameworks, hace que el tamaño pueda ser tan significativo como para afectar al rendimiento. Para paliar, en parte, el problema se realiza el proceso de minimizar. Los minimificadores de Javascript (o minificadores) reducen el peso en bytes del código compactándolo de distintas maneras:
  - Eliminan los comentarios y el código de depuración.
  - Eliminan los caracteres de sangrado: espacios blancos, tabuladores, saltos de línea, …
  - Sustituyen los nombres de variables y parámetros por los nombres mas cortos posibles.
- Esta última técnica puede desbaratar la inyección de dependencias de Angular, basada en la correspondencia de los nombres de los parámetros con elementos existentes. Angular soluciona el problema sustituyendo la función por un array con la función precedida por cadenas (no se minimizan) con los nombre de los paramentos inyectables.

```
app.controller("ConAJAXController",
   ['$scope','$log','$http',function(s,l,h) {...}])
```

CONTROLADORES

Controllers

- En Angular, un controlador es una función constructora del JavaScript, que se utiliza para poblar el Scope se Angular.
- Cuando un controlador aparezca declarado en el DOM a través de la directiva ng-controller, Angular creará una instancia de un nuevo objeto controlador, utilizando la función constructora del controlador especificado. También crea un nuevo scope secundario que estará disponible como un parámetro inyectable en la función constructora del controlador como $scope.
- Se deben usar los controladores para:
  - Configurar el estado inicial del objeto $scope.
  - Añadir comportamiento al objeto $scope.
- No se deben usar para:
  - Manipular DOM
  - Formatear la entrada o filtrar la salida
  - Compartir código o estado a través de los controladores
  - Administrar el ciclo de vida de los otros componentes

Declaración

- Declarar:

```
<div ng-controller="myController">
```

- Implementar:

```
angular.module('myApp',[])
.controller('myController', ['$scope',
function($scope) {
  $scope.myModel = 'Hola!';
  $scope.metodo = function(value) { … };
  $scope.metodo2 = function(value) { … };
}]);
```


Anidamiento

- $rootScope: scope raíz accesible desde cualquier punto de la aplicación (JavaScript y HTML) y donde definir valores independientemente del controlador actual.

```
.controller('miCtrl', function($scope, $rootScope){
    $scope.scopeNormal = "Local";
    $rootScope.scopeRaiz = "Raíz";
  });
<div ng-controller="miCtrl">
  {{ scopeNormal }} - {{ scopeRaiz }}
</div>
```

- $parent: en caso de conflicto de nombres, permite acceder al scope "padre" en caso de tener un controlador anidado en otro o al scope raíz. Solo en HTML.

```
<div ng-controller="unoCtrl">
  <div ng-controller="otroCtrl">
    {{ repetido }} --- {{ $parent.repetido }}
  </div>
</div>
```

Nota: Google Chrome → Angular Batarang

Scope con nombre

- Declarar:

```
<div ng-controller="myController as vm">
    {{vm.algo}}
```

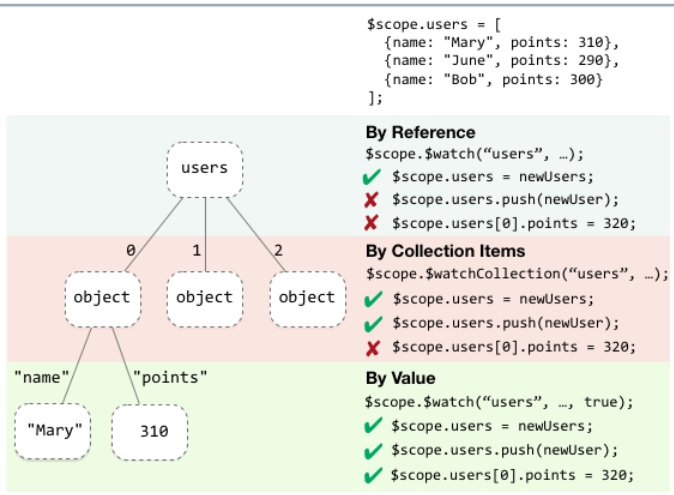
- Implementar:

```
angular.module('myApp',[])
.controller('myController', function() {
  this.myModel = 'Hola!';
  this.metodo = function(value) { … };
  this.metodo2 = function(value) { … };
});
```


Vigilar cambios en el modelo

- Solo cuando se cambia el objeto:
  - $scope.$watch('name', function(newValue, oldValue) {…});
- Cuando se cambia el objeto o una de sus propiedades a primer nivel:
  - $scope.$watchCollection('names', function(newNames, oldNames) {…});
- Cuando se cambia el objeto o una de sus propiedades a cualquier nivel.
  - $scope.$watch('name', function(newValue, oldValue) {…}**, true**);

Vigilar cambios en el modelo



Eventos

- Disparar el evento en el controlador actual y en todos sus contenedores.

  $emit ('myEvent', args...);
- Disparar el evento en el controlador actual, en todos sus contenedores y todos sus anidados.

  $broadcast ('myEvent', args...);
- Interceptar evento con un controlador de evento:

  $on('myEvent', function(event, args...) {…});
- El objeto event expone:
  - targetScope - {Scope}, currentScope - {Scope}, name - {string} , stopPropagation - {function=}, preventDefault - {function}, defaultPrevented - {boolean}

VISTAS

Plantillas

- En Angular, las plantillas se escriben en HTML añadiéndole elementos y atributos específicos Angular.
- Angular combina la plantilla con la información del modelo y el controlador para hacer que la vista dinámica que el usuario visualiza en el navegador.
- Los tipos de elementos y atributos de Angular que se utilizan son:
  - Marcadores: Utilizando la notación de doble llave {{}} para vincular expresiones a elementos del modelo.
  - Directivas : Un atributo o elemento que amplia a un elemento DOM existente o representa un componente DOM reutilizable.
  - Filtros: formatean los datos a visualizar.
  - Controles de formulario: Validan la entrada del usuario.

Marcadores

- Los marcadores AngularJS son expresiones similares a las de JavaScript que se colocan entre llaves dobles. Aceptan valores vinculados y son evaluadas por el "compilador" de plantillas que las sustituye por su resultado.
  {{expresión}}
- Las expresiones AngularJS se diferencian de las JavaScript en:
  - Contexto: en JavaScript el objeto global o la ventana y en Angular es el scope.
  - Permisivo: en JavaScript al evaluar propiedades no definidas genera un ReferenceError o TypeError, el Angular no emite error.
  - Control Flujo: Una expresión Angular no puede utilizar: condicionales, bucles o excepciones.
  - Declaraciones de funciones: No se pueden declarar funciones en una expresión angular, incluso dentro de la directiva ng-init.
  - RegExp: No se pueden crear expresiones regulares en una expresión Angular.
  - Eval: El eval () de JavaScript se sustituye por el método $ eval () en una expresión angular.
  - Filtros: Puede utilizar filtros en las expresiones para dar formato a los datos antes de mostrarla.
- Estas limitaciones se pueden salvar creando un método de controlador y llamar al método desde la expresión.

Directivas

- Las directivas tienen el aspecto de elementos o atributos en el código HTML, pero son interpretadas por angular al generar las vistas, lo que nos permite modificar el DOM o añadir nuevos comportamientos.
- Una directiva de atributo se utiliza como si fuese un atributo más de un elemento HTML y puede llevar parámetros o no, pero además tiene la ventaja de que desde ella podemos referenciar el $scope (el ViewModel) para realizar acciones o leer datos:

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

Directivas de formulario

- **ng-init**: Inicializa datos del modelo

```
<div ng-init="saludo='Hola mundo'">
```

- **ng-model**: vinculan bidireccionalmente el contenido de un elemento de formulario (input, select, textarea) a un valor del $scope.

```
<input type="text" ng-model="saludo">
```

- **ng-show**: permite que un elemento de la página se haga visible o invisible en función de cualquier valor del $scope y por lo tanto de nuestro modelo.

```
<input id="nombre" name="nombre" type="text" ng-
model="nombre" ng-show="tieneNombre ===false"/>
```

- **ng-hide**: la opuesta a la directiva ng-show.
- **ng-disabled**: permite habilitar o deshabilitar un elemento de entrada de datos como un <input> un <select> o un <button> en función de una condición.

```
<input id="nombre" name="nombre" type="text" ng-
model="nombre" ng-disabled="tieneNombre ===false"/>
```


Directivas de vinculación

- **ng-bind**: Vincula el contenido de la etiqueta en modo textual (textContent).

```
<div>{{mensaje}}</div>
<div ng-bind="mensaje"></div>
```

- **ng-bind-html**: Vincula el contenido de la etiqueta en modo HTML (innerHTML).

```
<div ng-bind-html="mensaje"></div>
```

- **ng-non-bindable**: No vincula el contenido, no se evalúan las expresiones en el contenido

```
<div ng-non-bindable>{{mensaje}}</div>
```

- **ng-src**: Vincula el atributo src con una expresión y asegura que no se solicite la URI hasta que no se haya resuelto la expresión.

```
<img ng-src="/images/{{fileName}}" >
```

- **ng-href**: Vincula el atributo href con una expresión y asegura que el hipervínculo no este disponible URI hasta que no se haya resuelto la expresión.

```
<a ng-href="{{url}}" >Pincha aqui</a>
```

Directivas de estilo

- **ng-style**: genera el atributo style partiendo de un objeto JSON del scope.
  - JS
    ```
    $scope.estilo={
      color:"#FF0000",
      backgroundColor:'yellow'
    }
    ```
  - HTML
    ```
    <div ng-style="estilo"  >texto</div>
    ```
- **ng-class**: genera el atributo class con las clases indicadas por un objeto JSON del scope. Especializaciones: **ng-class-odd** y **ng-class-even**.
  - CSS
    ```
    .error { color:red; }
    .remarcado { text-decoration: line-through; }
    .importante { font-weight: bold; }
    ```
  - JS
    ```
    $scope.clasesCss={error:true, remarcado:false, importante:true }
    ```
  - HTML
    ```
    <div ng-class="clasesCss">texto</div>
    ```


Directivas condicionales

- **ng-if**: determina que exista o no una etiqueta dependiendo de una condición.
  ```
  <div ng-if="condición">
  ```
- **ng-switch**: determina que cual es la etiqueta que va ha existir dependiendo de un valor. Esta directiva se complementa con las siguientes:
  - **on**: propiedad del $scope con el valor que actúa de selector.
  - **ng-switch-when**: Su valor es un literal que se compara con el de la propiedad del on, si coinciden la etiqueta existirá.
  - **ng-switch-default**: Si no se cumple ninguna condición de ng-switch-when se muestra el que tenga la directiva ng-switch-default.

```
<div ng-switch on="expresion">
    <div ng-switch-when="A">En valor hay una A</div>
    <div ng-switch-when="B">En valor hay una B</div>
    <div ng-switch-default>Ni una A ni una B</div>
</div>
```

Directivas iterativas: ng-repeat

- Recorre una colección generando la etiqueta para cada uno de sus elementos. AngularJS crea una serie se variables para usar en la iteración:
  - $index: Un número que indica el nº de elementos(de 0 a n-1).
  - $first / $last: Vale true si es el primer o el último elemento del bucle.
  - $middle : Vale true si no es ni el primer ni el último elemento del bucle.
  - $even / $odd : Vale true si es un elemento con $index par o impar (0 es par).

```
<tr ng-repeat="provincia in provincias"
   style="background-color:{{$even?'white':'yellow'}}">
     <td>{{$index}}</td>
     <td>{{provincia.idProvincia}}</td>
     <td>{{provincia.nombre}}</td>
</tr>
```

- Si cambian los datos, AngularJS automáticamente borrará todos los tag y los volverá a crear con todos los datos. Como esto puede ser poco eficiente, se puede indicar qué columna es la clave única del objeto y así optimizar el proceso.

```
ng-repeat="provincia in provincias track by provincia.idProvincia"
```

- ng-repeat sólo permite repetir un único tag, para repetir un grupo se utilizan las directivas **ng-repeat-start** y **ng-repeat-end**:

```
<strong ng-repeat-start="provincia in provincias track by provincia.idProvincia"
>{{provincia.idProvincia}}</strong> <i>{{provincia.nombre}}</i><br ng-repeat-end>
```


Directivas iterativas: ng-options

- Genera las etiquetas option de un select recorriendo una colección generando. Para que pueda asignar correctamente el value y el contenido la expresión debe ser:

  "valor **as** *etiqueta* **for** *item* **in** *items* **track by** *claveunica"*

  - items: La propiedad del $scope que contiene el array de elementos a mostrar en el <select>.
  - item: Variable que contendrá cada uno de los elementos de items.
  - valor: Una expresión que use la variable item que será el value del option.
  - etiqueta: Una expresión usando item que será el contenido de la etiqueta <option>.
  - claveunica: La propiedad que es clave única.

```
<select ng-model="Envio.Provincia" ng-options="p.id as p.nombre for p in
provincias track by p.id" >
  <option value="">--Elige opcion--</option>
</select>
```

- Equivale a:

```
<select ng-model="nombre2" >
  <option value="">--Elige opcion--</option>
  <option ng-repeat="p in provincias track by p.id" value="{{p.id}}">{{
p.nombre }}</option>
</select>
```

Directivas de eventos

- ng-click, ng-dblclick: Cuando se produce el evento click o doble click.
- ng-focus, ng-blur: Cuando se produce el evento focus o blur.
- ng-change: Cuando se cambia el tag <input> o select> pero no cuando se cambia a consecuencias del propio modelo.
- ng-copy, ng-cut, ng-paste: Cuando se produce el evento copy, cut o paste.
- ng-keydown, ng-keyup, ng-keypress: Cuando se produce el evento keydown, keyup o keypress.
- ng-mousedown: Cuando se produce el evento mousedown
- ng-mouseenter: Cuando se produce el evento mouseenter
- ng-mouseleave: Cuando se produce el evento mouseleave
- ng-mousemove: Cuando se produce el evento mousemove
- ng-mouseover: Cuando se produce el evento mouseover
- ng-mouseup: Cuando se produce el evento mouseup

```
<input type="button" value="Haz Clic" ng-click="procesarClic()">
<input type="button" value="haz clic" ng-click="numero=2; otra=dato " />
```


Otras directivas

- ng-list: Convierte un array de cadenas en una cadena separando los elementos con el delimitador indicado (coma por defecto):

  ```
  <input name="namesInput" ng-model="names" ng-list=', ' required>
  ```
- ng-include: se puede usar directamente como un tag HTML en vez de como un atributo HTML. Se usa para cargar trozos de HTML en la página. Es como los include de la parte de servidor pero ahora desde JavaScript. Como el src puede recibir variables, las cadenas constantes deben ir entre comillas:

  ```
  <ng-include src="'cabecera.html'"></ng-include>
  ```

Cambio de vistas

```
<button ng-click="changeView(1)">Vista 1</button>
<button ng-click="changeView(2)">Vista 2</button>
<div class="container"><div ng-include="tpl"></div></div>

angular.module('myApp', []).controller('userCtrl', function($scope) {
      $scope.tpl = 'myUsers_List.htm';
      $scope.changeView = function(opc) {
        switch(opc) {
          case 1:
          default:
            $scope.tpl = 'myUsers_List.htm';
            break;
          case 2:
            $scope.tpl = 'myUsers_Form.htm';
            break;
        }
      };
});
```


$templateCache

- La primera vez que se utiliza una plantilla, se carga en la memoria caché de plantilla para su recuperación rápida.
- Se puede cargar plantillas directamente en la memoria caché con una etiqueta de script, o utilizando el servicio $templateCache directamente.
- A través de la etiqueta de script:

```
<script type="text/ng-template" id="templateId.html">
  <p>This is the content of the template</p>
</script>
```

- A través del servicio $templateCache:

```
app.run(function($templateCache) {
  $templateCache.put('templateId.html', 'This is the content of the
template');
});
```

- Para recuperar la plantilla más tarde:

```
<div ng-include=" 'templateId.html' "></div>
```

- Desde JavaScript:

```
$templateCache.get('templateId.html');
```

Filtros

- Los filtros en Angular son los encargados de procesar la información antes de mostrarla en pantalla. Permiten hacer muchas cosas, pero lo más habitual es utilizarlos para modificar los valores a presentar o aplicarles formato.
- Se aplican dentro de las expresiones de databinding y se indican con el carácter |. Al igual que las directivas, los filtros pueden recibir parámetros (:) que modifiquen su comportamiento.

```
<ul>
    <li ng-repeat="c in customers | filter:{region: 'Madrid'}">
        {{c.name | pluralize | toUpperCase}}
    </li>
</ul>
```

- Se puede usar los filtro directamente desde JavaScript, utilizando el servicio $filter que devuelve la función de conversión asociada al nombre de filtro pasado.

```
var filtro=$filter("currency");
rslt=filtro($scope.importe);
```


Filtros

- **number**: se aplica sobre números para limitar el nº de decimales que se muestran de dicho número (3 por defecto), también cambia el separador de decimales al idioma actual.
  `<div>{{importe | number:2}}</div>3`
- **date**: se aplica sobre fechas para mostrar un formato concreto. Se puede expresar mediante marcadores (dd,MM,yyyy,HH,mm, ...) o formatos predefinidos (medium, short, fullDate, longDate, mediumDate, shortDate, mediumTime, shortTime):
  `<div>{{miFecha | date:'dd/MM/yyyy'}}</div>`
- **currency**: muestra un número con el símbolo de la moneda local y con el número de decimales correctos (2 en “es-es”).
  `<div>{{ importe | currency }}</div>`
- **lowercase**: transformará un String a minúsculas.
  `<div>{{ nombre | lowercase }}</div>`
- **uppercase**: transformará un String a mayúsculas.
  `<div>{{ nombre | uppercase }}</div>`
- **orderBy**: permite ordenar un array de datos.El resultado será el mismo array pero ordenado.
  `{{array | orderBy:'propiedad' }}`
  `{{array | orderBy:['propiedad1','propiedad2',...,'propiedadn'] }}`
- **limitTo**: permite limitar el nº de elementos que se muestran del array. Si el valor es negativo lo que muestra es los 'n' últimos elementos.
  `<tr ng-repeat="p in provincias | limitTo:2">`
- **json**: convierte un objeto JavaScript en una cadena JSON, utilizado para depuración.

Filtro filter

Filtra el array para que el array resultante sólo tenga ciertos datos del array original, soporta varios tipos formas de filtrar.

Búsqueda en todas las propiedades: Que alguna propiedad contenga el texto pasado (%texto%), no es sensible a mayúsculas/minusculas.

```
<tr ng-repeat="p in provincias | filter:'la'">
```

Búsqueda en una propiedad: Se puede bucar por texto o valor.

```
<tr ng-repeat="p in provincias | filter:{ nombre:'la'}'">
```

Búsqueda en varias propiedades: haciendo un AND entre ellas.

```
<tr ng-repeat="p in provincias | filter:{
nombre:'la',insular:true}'">
```

Búsqueda combinada: Usando como nombre de propiedad "$" se busca en todas las propiedades.

```
<tr ng-repeat="p in provincias | filter:{
$:'la',insular:true}'">
```


Filtro filter

Comparador: el filtro filter dispone de un segundo parámetro para indicar cómo se hace la comparación: true (el texto a buscar debe ser igual al valor de la propiedad, es sensible a mayúsculas/minusculas) o el nombre de una función booleana con dos parámetros que realice la comparación.

```
$scope.distinto=function(a, b) {
   return a != b;
}
<tr ng-repeat="p in provincias | filter:{ nombre:'la'}:distinto">
```

Búsqueda personalizada: permite utilizar una función booleana con un parámetros que representa cada objeto y que realice la comparación.

```
$scope.consultaPersonalizada=function(v) {
   return v.nombre.indexOf('la') == -1 && v.insular;
}
<tr ng-repeat="p in provincias | filter:consultaPersonalizada">
```

Crear filtros

- Crear un nuevo filtro es casi tan sencillo como crear una función a la que se pasa un valor y devuelve el valor transformado.
- La forma de crear un filtro es llamando al método filter de un módulo. Este método acepta una función de factoría, siendo la función de factoría la que retornará nuestra función de filtro.

```
app.filter("capitalice",function () {
  return function (s) {
    if (typeof (s)==="string") {
      return s.charAt(0).toUpperCase() +
          s.substring(1, s.length).toLowerCase();
    }
    return s;
  };
});
```

```
<div>{{ nombre | capitalice}}</div>
```


FORMULARIOS

Encapsulación

- AngularJS encapsula automáticamente cada formulario en una propiedad del $scope del controlador con el mismo nombre que el atributo name del formulario y del tipo FormController.
- Expone una serie de propiedades para gestionar los cambios, la validación y los errores:
  - $pristine: Vale true si el formulario o campo aún no ha sido modificado por el usuario, si no vale false.
  - $dirty: Vale true si el formulario o campo ya ha sido modificado por el usuario, si no vale false.
  - $valid: Vale true si el formulario o el campo son válidos, es decir, si cumplen todas las validaciones que se han indicado en los campos.
  - $invalid: Vale true si el formulario o el campo son inválidos, es decir, si se incumple alguna de las validaciones que se han indicado en los campos.
  - $error: Contiene propiedades para indicar el tipo de error que se ha producido.

Encapsulación

- Para cada control del formulario se crea una propiedad en el FormController controlador con el mismo nombre que el atributo name del formulario y del tipo NgModelController que permiten gestionar los cambios, la validación y los errores con las mismas propiedades vistas anteriormente.

```
<form name="miForm" >
   Nombre:<input type="text" ng-model="model.nombre"
     name="nombre" required > <br>
   Correo eletronico:<input type="text" ng-
     model="model.email" name="email" required>
</form>

if($scope.miForm.$dirty) {…}
if($scope.miForm.nombre.$invalid) {…}
```

Directivas de formularios

- form
- input
- input[checkbox]
- input[date]
- input[datetime-local]
- input[email]
- input[month]
- input[number]
- input[radio]
- input[text]
- input[time]
- input[url]
- input[week]
- select
- textarea
- button

Tipos de Validaciones

- required: el campo es requerido, aplicable a cualquier <input> , <select> o <textarea>, con la directiva required o ng-required (vincula al modelo).

```
<input type="text" ng-model="model.nombre"
  name="nombre" required >
<input type="text" ng-model="model.nombre"
  name="nombre" ng-required="reqNombre" >
```

- ng-minlength: El campo debe tener un nº mínimo de caracteres
- ng-maxlength: El campo debe tener un nº máximo de caracteres
- ng-trim: Se establece a false no se recorte automáticamente la entrada.
- ng-pattern: El campo debe seguir una expresión regular, applicable a <input type="number|text|email" o a <textarea>.

```
<input type="text" ng-model="model.nombre"
  name="nombre" ng-minlength="3" ng-maxlength="50" ng-
  pattern="/^[a-zA-Z]*$/" >
```

Tipos de Validaciones

- number: El campo debe ser un número entero (type="number"), para números reales es necesario incluir el atributo step="0.01".
- min: El campo debe tener un valor mínimo
- max: El campo debe tener un valor máximo

```
<input type="number" ng-model="model.edad" name="edad"
  min="18" max="99" >
```

- url: El campo debe tener el formato de una URL. Deberemos indicar en el <input> que type="URL".

```
<input type="url" ng-model="model.sitioweb"
  name="sitioweb"  >
```

- email: El campo debe tener el formato de un correo electrónico. Deberemos indicar en el <input> que type="email".

```
<input type="email" ng-model="model.correo"
  name="correo" >
```


Comprobando las validaciones

- $valid y $invalid indican a nivel global si pasa la validación. Para saber que validaciones ha fallado, $error dispone de una propiedad por cada tipo de validación.

```
$scope.miForm.nombre.$error.required
$scope.miForm.$error.email
```

- Si se consulta a través de un NgModelController indica:
  - Si vale true es que la validación ha fallado
  - Si vale false es que la validación es correcta
  - Si vale undefined es que no se está validando porque no se le ha definido
- Si se consulta a través del formulario indica:
  - Si vale false es que no hay ningún campo que incumpla dicha validación.
  - Si vale un array. El contenido del array son objetos de la clase NgModelController correspondientes a cada uno de los campos en los que ha fallado esa validación
  - Si vale undefined es que ningún campo tiene esa validando
- Con la directiva novalidate en el tag <form> se evita que el navegador haga sus propias validaciones que choquen con las de AngularJS.

```
<form name="miForm" novalidate >
```

Notificación de errores

- Con las directivas ng-if y ng-show se puede controlar cuando se muestran los mensajes de error.

```
<span ng-show="miForm.nombre.$error.maxlength">El
  tamaño m&aacute;ximo debe ser 50</span>
<span ng-show="miForm.nombre.$error.required">El
  campo es requerido</span>
```

- Para no mostrar los mensajes desde el principio sin dar oportunidad al usuario a introducir los datos:

```
<span ng-show="miForm.nombre.$error.required &&
  miForm.nombre.$dirty" …
```

- Para resumir en un mensaje varias causas de error:

```
<span ng-show=" miForm.nombre.$error.minlength
  || miForm.nombre.$error.maxlength">El nombre
  debe tener entre 3 y 50 letras</span>
```


Enviar formulario

- La directiva ng-disabled permite deshabilitar el botón de submit mientras no sea valido.

```
<button ng-click="enviar()" ng-
  disabled="miForm.$invalid" >Enviar</button>
```

- Para evitar posibles errores seria conveniente volver a comprobar:

```
$scope.enviar=function() {
  if ($scope.miForm.$valid) {
    …
  } else {
    alert("Hay datos inválidos");
  }
}
```

Avisos visuales

- AngularJS esta preparado para cambiar el estilo visual de los controles según su estado, utilizando class de CSS.
- En caso de estar definidos AngularJS aplicará automáticamente uno de los siguientes estilos:
  - ng-valid
  - ng-valid-*key* (Ej: ng-valid-required)
  - ng-invalid
  - ng-invalid-*key* (Ej: ng-valid-required)
  - ng-pristine
  - ng-dirty
- Para personalizar el estilo:

```
.ng-invalid {
  border-color: red;
}
```


ngModelOptions

- Permite afinar cómo se hacen las actualizaciones de modelo.
- updateOn : especifica el evento (o eventos separados por espacios) que desencadenan la actualización del modelo. default representa el evento predeterminado del control.
- debounce: milisegundos de espera antes de actualizar el modelo (por defecto 0, actualización inmediata).

```
ng-model-options="{ updateOn: 'default blur', debounce: { 'default':
    500, 'blur': 0 } }"
```

- allowInvalid : permite actualizar el modelo con valores que no ha superado la validación.
- getterSetter : true para tratar el valor de ngModel como funciones getters / setters.

```
var _name = 'Brian';
$scope.user = {
   name: function(newName) {
      return arguments.length ? (_name = newName) : _name;
   }
};
```

SERVICIOS

Servicios

- Los servicios permiten agrupar funcionalidad que luego estará disponible para ser usada en los Controllers, mejorando la claridad del código y favoreciendo la reutilización.
- A diferencia de la mayoría de arquitecturas en que los servicios suelen ser objetos stateless (sin estado), en Angular los servicios se utilizan también para mantener estado.
- En Angular los servicios son la herramienta básica para mantener el estado de la aplicación (capa cliente) y compartir información entre los Controllers.
- Un servicio nunca interacciona con la propia página, sólo con otros servicios o con un servidor.
- AngularJS tiene 5 tipos distintos de servicios:
  - Completo: Proveedores
  - Atajos: Constantes, Valores, Servicios, Factorías

Tipo de Servicios: constant y value

- Una constant es un servicio al que le pasamos directamente el valor de dicho servicio. Su principal característica es que se puede inyectar en cualquier sitio. Se define llamando al método **constant** de un módulo al que pasaremos el nombre de la constante y su valor (valores escalares, objetos, JSON, función).
- Un value es un servicio al que le pasamos directamente el valor de dicho servicio. Se define llamando al método **value** de un módulo. Idéntico al constant salvo que solo se pueden inyectar a otros servicios.

```
app.constant("idioma","es-es");
app.value("version","1.1.0.234");
app.controller("PruebaController",
["$scope","idioma",
function($scope, idioma) {
  $scope.constante=idioma;
}]);
```


Tipo de Servicios: service

- Un service es un servicio al que le pasamos directamente el valor de dicho servicio. Se define llamando al método service de un módulo. A dicho método se le pasa el nombre del servicio y el nombre de una clase JavaScript. Será AngularJS el que cree internamente una instancia de la clase invocando la función constructora. A diferencia de constant y value, se puede inyectar otros servicios en el propio constructor.

```
function MiClase(otraClase) {
  this.prop=otraClase;
  this.metodo=function() { … }
}
app.value("miValor", new otraClase());
app.service("miClase", ["miValor", MiClase]);
app.controller("PruebaController",
            ["$scope", "miClase",function($scope, miClase) {
    $scope.valor= miClase.metodo();
}]);
```

Tipo de Servicios: factory

- El método factory asocia un nombre a una función para que ésta devuelva ahora el valor del servicio. Es decir que tenemos una función JavaScript que actúa como factoría, retornando la propia función de factoría el valor del servicio.

```
app.factory("miFactory", function(){
    var esPrivado= "Solo desde el método";

    var srv = {
        esAccesible: "Directamente",
        getPrivado: function(){ return esPrivado; },
        setPrivado: function(valor){ esPrivado=valor; }
    }
    return srv;
});
app.controller("PruebaController",
            ["$scope", "miFactory",function($scope, miFactory) {
    $scope.valor= miFactory.getPrivado();
    $scope.otrovalor= miFactory.esAccesible;
}]);
```


Tipo de Servicios: provider

- Un provider es como un factory pero permite que se configure antes de crear el valor del servicio. La función del factory es sustituida en el provider por un método público llamado $get obligatoriamente dentro de un objeto. Al ser un objeto puede contener propiedades y métodos de configuración.

```
function MiProvider() {
  var cfg="";
  this.setMiConfig=function(configuracion) {
    this.cfg=configuracion;
  };
  this.$get=function() {
    var loc = this.cfg;
    …
    return rslt;
  }
}
app.provider("elProvider", MiProvider);
app.config(["elProvider",function(elProvider) {
  elProvider.setMiConfig({ … });
}]);
```

Servicios predefinidos

- **$window**: Una referencia al objeto de ventana del navegador. Como referencia global en JavaScript puede causar problemas en los sitemas de prueba. En Angular siempre se usa a través del servicio de $window.
- **$document**: Una envoltura jQuery o jqLite del objeto window.document del navegador.
- **$rootElement**: El elemento raíz de la aplicación Angular. Este es el elemento donde se declaró ngApp o el elemento pasado a angular.bootstrap.
- **$location**: El servicio de $location analiza la URL en la barra de direcciones del navegador (basado en el window.location) y hace que la URL este disponible para la aplicación (enrutado).
- **$interval**: Envoltorio de Angular para window.setInterval. La función fn se ejecuta cada delay milisegundos.
- **$timeout**: Envoltorio de Angular para window.setTimeout.

Servicios predefinidos

- **$rootScope**: Cada aplicación tiene un único ámbito raíz. Todos los demás ámbitos son ámbitos descendiente del ámbito raíz.
- **$log**: Servicio simple para el registro. Escribe el mensaje en la consola del navegador (si existe).
- **$filter**: Los filtros se utilizan para dar formato a los datos mostrados al usuario.
- **$parse**: Convierte una expresión Angular en una función.
- **$animate**: Expone una serie de métodos de utilidad DOM de apoyo a la animación.
- **$q**: Expone los métodos necesarios para la implementacion de promesas.
- **$http**: Facilita la comunicación con los servidores remotos HTTP a través del objeto XMLHttpRequest o mediante JSONP.

decorator

- Un servicio decorador intercepta la creación de un servicio, lo que le permite anular o modificar el comportamiento del servicio.
- El objeto devuelto por el decorador puede ser el servicio original, un nuevo objeto de servicio que sustituye o envuelve al servicio original.
- El método decorator del módulo recibe el nombre del servicio y una función que debe devolver la nueva versión del servicio, a la que se inyecta $delegate con la instancia del servicio original que puede ser parcheado, configurado, decorado o ignorado.

```
app.decorator('$log', ['$delegate', function($delegate) {
     $delegate.warn = $delegate.error;
     return $delegate;
}]);
```


ACCESO AL SERVIDOR

Servicio $http

- El servicio $http permite hacer peticiones AJAX al servidor.
- Es realmente como el objeto XMLHttpRequest o el método ajax() de JQuery. La diferencia con estos dos últimos es que está integrado con Angular como un servicio (con todas las ventajas de ellos conlleva) pero principalmente porque notifica a AngularJS que ha habido un cambio en el modelo de JavaScript y actualiza la vista y el resto de dependencias adecuadamente.
- $http acepta como parámetro un único objeto llamado config con todas las propiedades que necesita para la petición:
  - method: El método HTTP para hacer la petición. Sus posibles valores son: GET, POST, PUT, DELETE, etc.
  - url: La URL de donde queremos obtener los datos.
  - data: Si usamos el método POST o PUT aquí pondremos los datos a mandar en el body de la petición HTTP
  - params: Un objeto que se pondrá como parámetros de la URL.

Servicio $http

- El método success(fn) permite asignar la función que se ejecuta si todo ha funcionado correctamente, la función recibirá los siguiente argumentos :
  - data: Un objeto JavaScript correspondiente a los datos JSON que ha recibido
  - status: Es el estado HTTP devuelto. Su valor siempre será entre 200 y 299 ya que si se llama a esta función significa que la petición ha tenido éxito.
  - headers: Es una función que acepta como único parámetro el nombre de una cabecera HTTP y devuelve su valor.
  - config: El mismo objeto config usado para configurar la petición.
- En caso de error se llama la función indicada con el método error() que tiene la misma firma que la del success.

Servicio $http

```
app.controller("ConAJAXController",
    ['$scope','$log','$http',function($scope,$log,$http)
{
…
  $http({
    method: 'GET',
    url: 'datos.json'
  }).success(function(data, status, headers, config) {
      $scope.modelo=data;
  }).error(function(data, status, headers, config) {
      alert("Ha fallado. Estado HTTP:"+status);
  });
…
```


Promise Pattern

- El Promise Pattern es un patrón de organización de código que permite encadenar llamadas a métodos que se ejecutaran a la conclusión del anterior (flujos).
- Simplifica y soluciona los problemas comunes con el patrón Callback:
  - Llamadas anidadas
  - Complejidad de código

  o.m(1, 2, f(m1(3, f1(4,5,ff(8))) → o.m(1, 2).f().m1(3).f1(4 , 5).ff(8)
- Aunque se utiliza extensamente para las operaciones asíncronas, no es exclusivo de las mismas.
- El servicio $q es un servicio de AngularJS que contiene toda la funcionalidad de las promesas (está basado en la implementación Q de Kris Kowal).
- La librería JQuery incluye el objeto $.Deferred desde la versión 1.5.
- Las promesas se han incorporado a los objetos estándar de JavaScript en la versión 6.

Objeto Promise

- Una “promesa” es un objeto que actúa como proxy en los casos en los que no se puede utilizar el verdadero valor porque aún no se conoce (no se ha generado, llegado, …) pero se debe continuar sin esperar a que este disponible (no se puede bloquear la función esperando a su obtención).
- Una “promesa” puede tener los siguientes estados:
  - Pendiente: Aún no se sabe si se podrá o no obtener el resultado.
  - Resuelta: Se ha podido obtener el resultado (defered.resolve())
  - Rechazada: Ha habido algún tipo de error y no se ha podido obtener el resultado (defered.reject())
- Los métodos del objeto promesa devuelven al propio objeto para permitir apilar llamadas sucesivas.
- Como objeto, la promesa se puede almacenar en una variable, pasar como parámetro o devolver desde una función, lo que permite aplicar los métodos en distintos puntos del código.

Invocar promesas

- El objeto Promise expone los métodos:
  - then(fnPendiente, fnError): Recibe como parámetro la función a ejecutar cuando termine la anterior y, opcionalmente, la función a ejecutar en caso de que falle la anterior.
  - catch(fnError): Recibe como parámetro la función a ejecutar en caso de que falle.
  - finally(fnNueva): Recibe como parámetro la función a ejecutar independientemente de si ha sido resuelta o rechazada.

```
list().then(calcular, ponError).then(guardar)
```

Servicio $http

- El servicio $http ha sido redefinido utilizando el patrón Promise, por lo que los métodos success y error han quedado obsoletos.
- La función $http y los atajos ($http.get, $http.head, $http.post, $http.put, $http.delete, $http.jsonp, $http.patch) devuelven una promesa.

```
var promesa = $http({method: 'GET', url: '/someUrl'})
  .then(function successCallback(response) { … }
  , function errorCallback(response) { … });
```

- El parametro response es un objeto con las siguientes propiedades:
  - data {string|Object}: El cuerpo de la respuesta transformado con las funciones de transformación.
  - status {number}: HTTP status code de la respuesta.
  - headers {function([headerName])}: Función cabecera de la respuesta.
  - config {Object}: Objeto configuracion de la petición.
  - statusText {string}: HTTP status text de la respuesta.

Atajos

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

```
$http.get('/someUrl', config)
        .then(successCallback, errorCallback);
$http.post('/someUrl', data, config)
        .then(successCallback, errorCallback);
```

Servicio RESTFul

```
app.factory('entidadDAO', [ '$http', function($http) {
    var baseUrl = '/dao/entidad';
    return {
        query : function() { return $http.get(baseUrl); },
        get : function(id) { return $http.get(baseUrl + '/' + id); },
        add : function(item) { return $http.post(baseUrl + '/', item); },
        charge : function(id, item) {
            return $http.put(baseUrl + '/' + id, item);
        },
        remove : function(id, item) {
            return $http.delete(baseUrl + '/' + id, item);
        }
    };
} ]);
```


Recursos

- El módulo ngResource dispone de un servicio llamado $resource que simplifica el acceso a los servicios web RESTFul.
- Este módulo no está incluido en la distribución de base de Angular, por lo que para usarlo tenemos que instalarlo, inyectarlo como dependencia en el módulo principal de nuestra aplicación e inyectar el servicio a los controladores o servicios que lo necesiten.

```
<script src="angular-resource.js"></script>
angular.module("app", ["ngResource"])
      .factory('DAO',function($resource){
```

Configuración

- Para obtener el objeto de servicio la factoría $resource recibe los siguientes parámetros:
  - url: Plantilla parametrizada de la URL base del servicio RESTful. Los parámetros van precedidos por : antes del nombre y se mapean con las propiedades por defecto.
  - paramDefaults: Pares nombre/valor con los valores por defecto para los parámetros de plantilla URL. Si el valor del parámetro tiene el prefijo @ entonces el valor de ese parámetro se extrae de la propiedad correspondiente en el objeto de datos.
  - actions: Conjunto de declaraciones de acciones personalizadas adicionales que extienden al conjunto predeterminado de acciones de recursos.
  - options: Ajustes personalizados al comportamiento por defecto.

```
dao = $resource('http://localhost/dao/entidad/:id'
    , { id:'@id_modelo'}
    , { update : {method:'PUT'} });
```


Métodos de servicio

- Métodos de servicio
  - query: Usa el método HTTP GET y espera un array de objetos JSON como respuesta.
    ```
    $scope.listado = dao.query();
    ```
  - get: Usa el método HTTP GET con los parámetros suministrados y espera un objeto JSON como respuesta.
    ```
    $scope.model = dao.get({id:listado[idx].id_modelo});
    ```
- Métodos de instancia
  - $save: Usa el método HTTP POST y recibe las funciones de sucedido y error.
    ```
    $scope.model.$save(fnSuccess, fnError);
    ```
  - $delete: Usa el método HTTP DELETE y recibe las funciones de sucedido y error.
    ```
    $scope.model.$delete(fnSuccess, fnError);
    ```
  - $remove: Igual que $delete.
- Para crear una nueva instancia donde aplicar el método $save:
    ```
    $scope.model = new dao();
    ```

ENRUTADO

Introducción

- El enrutado permite tener una aplicación de una sola página, pero que es capaz de representar URL distintas, simulando lo que sería una navegación a través de la aplicación, pero sin salirnos nunca de la página inicial. Esto permite:
  - **Memorizar rutas profundas dentro de nuestra aplicación**. Podemos contar con enlaces que nos lleven a partes internas (deeplinks), de modo que no estemos obligados a entrar en la aplicación a través de la pantalla inicial.
  - Eso **facilita también el uso natural del sistema de favoritos** (o marcadores) del navegador, así como el historial. Es decir, gracias a las rutas internas, seremos capaces de guardar en favoritos un estado determinado de la aplicación. A través del uso del historial del navegador, para ir hacia delante y atrás en las páginas, podremos navegar entre pantallas de la aplicación con los botones del navegador.
  - **Mantener vistas en archivos independientes**, lo que reduce su complejidad y administrar los controladores que van a facilitar el procesamiento dentro de ellas.

Rutas internas

- En las URL, la “almohadilla“, el carácter “#“, sirve para hacer rutas a anclas internas: zonas de una página.
- Cuando se pide al navegador que acceda a una ruta creada con “#“ éste no va a recargar la página (cargando un nuevo documento que pierde el contexto actual), lo que hará es buscar el ancla que corresponda y mover el scroll de la página a ese lugar.
  - http://example.com/index.html
  - http://example.com/index. html#/seccion
  - http://example.com/index. html#/pagina_interna
- Es importante fijarse en el patrón "#/“, sirve para hacer lo que se llaman "enlaces internos" dentro del mismo documento HTML.
- En el caso de AngularJS no habrá un movimiento de scroll, pues con Javascript se detectará el cambio de ruta en la barra de direcciones para intercambiar la vista que se está mostrando.

ngRoute

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

```
<script src="angular-route.js"></script>
angular.module("app", ["ngRoute"])
```

Hash Bag

- Dado que los robots indexadores de contenido de los buscadores no siguen los enlaces al interior de la pagina (dado que asumen como ya escaneada), el uso del enrutado con # que carga dinámicamente el contenido impide el referenciado en los buscadores.
- Para indicarle al robot que debe solicitar el enlace interno se añade una ! después de la # quedando la URL:
  `http://www.example.com/index.html#!ruta`
- Es necesario configurar el servicio $location, que encapsula el window.location, con el símbolo a utilizar:

```
angular.module("app", ["ngRoute"])
.config(['$locationProvider',
  function($locationProvider) {
    $locationProvider.hashPrefix('!');
  }
]);
```


Configurar el sistema de enrutado

- El sistema de enrutado de AngularJS nos permite configurar las rutas que queremos crear en nuestra aplicación de una manera declarativa.
- El servicio $routeprovider contiene los métodos necesarios para la configuración:
  - when(), permite indicar qué se debe hacer en cada ruta que se desee configurar
  - otherwise(), marca un comportamiento cuando se intente acceder a cualquier otra ruta no declarada.
- Esta configuración se debe realizar dentro del método config(), que pertenece al modelo. De hecho, solo podemos inyectar $routeProvider en el método config() de configuración.

```
angular.module("app", ["ngRoute"])
  .config(function($routeProvider){
    //configuración y definición de las rutas
  });
```

Rutas

- Las rutas las configuras por medio del método when() que recibe dos parámetros:
  - la ruta que se está configurando
  - un objeto que tendrá los valores asociados a esa ruta, que contendrá como mínimo las siguientes propiedades:
    - "controller", para indicar el controlador
    - "controllerAs", el nombre con el que se conocerá el scope dentro de esa plantilla
    - "templateUrl", el nombre del archivo, o ruta, donde se encuentra el HTML de la vista que se debe cargar cuando se acceda a la ruta.
- Se concatenaran tantas invocaciones al método when como rutas estén disponibles.

Rutas

```
.config(['$routeProvider',  function($routeProvider) {
   $routeProvider
    .when('/Book/:bookId', {
     templateUrl: 'book.html', controller: 'BookCtrl', controllerAs: 'book'
    })
    .when('/Book/:bookId/ch/:chapterId', {
     templateUrl: 'chapter.html', controller: 'ChapterCtrl', controllerAs: 'chapter'
    })
    .otherwise({
     redirectTo: '/'
    });
}])
.controller('MainCtrl', ['$route', '$routeParams', '$location', function($route, $routeParams, $location) {
            this.$route = $route;
            this.$location = $location;
            this.$routeParams = $routeParams;
}])
```

Parámetros

- Para definir un parámetro en la URL hay que incluir el nombre de la variable en el path de la ruta precedido por ":":
  .when('/ruta/subruta/:parametro', { … })
- Se pueden definir mas de un parámetro y combinarlos con partes estáticas utilizando "/" como separador:
  '/ruta/subruta/:param1/:param2/algo/:param3'
- El último parámetro pude ser opcional y se marca con el carácter "?" al final, que toma el valor undefined si no aparece en la ruta:
  '/ruta/subruta/:param1/:param2**?**'
- El último parámetro pude ser indefinido, toma como valor el resto de la ruta con los separadores incluidos, opcional y se marca con el carácter "*" al final.
  '/ruta/subruta/:param1/algo/:param2***'
- Los parámetros QueryString no participan en el enrutado pero si entran en el controlador como parámetros.
- El servicio $routeParams (es necesario inyectarlo al controlador), contiene tantas propiedades como parámetros se hayan definido en la ruta y el QueryString de la URL, que toman el valor utilizado en la URL invocada.

ngView

- La directiva ngView indica al AngularJS que las vistas se deben desplegar en ese contenedor.

```
<div ng-controller="MainCtrl as main">
  <a href="#/Book/Moby">Moby</a> |
  <a href="#/Book/Moby/ch/1">Moby: Ch1</a> |
  <a href="#/Book/Gatsby">Gatsby</a> |
  <a href="#/Book/Gatsby/ch/4?key=value">Gatsby: Ch4</a> |
  <a href="#/Book/Scarlet">Scarlet Letter</a><br/>

  <div>
    <div ng-view></div>
  </div>
</div>
```

Mantenimiento de estado

- Uno de los posibles aspectos problemáticos del enrutado es que los controladores se invocan con cada vista donde se estén usando, ejecutando la función que los define cada vez que se carga la ruta.
- Por este motivo todos los datos que se inicializan en los controladores se vuelven a poner a sus valores predeterminados cuando carga cualquier vista que trabaje con ese controlador.
- El mantenimiento de estado se puede realizar mediante:
  - $rootScope: Zona de memoria (ámbito) común para todo el modulo. Provoca acoplamientos de datos y comportamientos inesperados.
  - Servicios: Comunes para todo el modulo. El patrón singleton, utilizado en la inyección de dependencias, asegura que sólo existe una instancia de ellos en la aplicación, por lo que no pierden su estado, y, si hay varios componentes que dependen de un mismo objeto, todos recibirán la misma instancia del objeto. Permiten desarrollos mas robustos.

UTILIDADES

Módulos adicionales no cargados

- **ngAnimate**: proporciona soporte para animaciones basadas en CSS, así como animaciones basadas en JavaScript a través de ganchos de devolución de llamada.
- **ngAria**: proporciona soporte para los atributos comunes ARIA que transmiten estado o información semántica acerca de la aplicación para los usuarios de las tecnologías de asistencia, tales como lectores de pantalla.
- **ngCookies**: proporciona una envoltura conveniente para la lectura y la escritura cookies en el navegador.
- **ngMessages:** proporciona soporte mejorado para la visualización de mensajes de error dentro de las plantillas (normalmente dentro de formularios o al representar objetos de localización que devuelven pares clave / valor).
- **ngMock**: proporciona soporte para inyectar y simular servicios Angular en pruebas unitarias.
- **ngResource**: proporciona soporte para la interacción con los servicios REST a través del servicio $resource.
- **ngRoute:** ofrece servicios y directivas de enrutamiento y enlaces profundos para aplicaciones Angular.
- **ngSanitize**: proporciona funcionalidad para desinfectar el HTML.
- **ngTouch**: ofrece eventos táctiles y otros asistentes para dispositivos táctiles.

Módulos de terceros

- Angular Modules (ngmodules):
  - http://ngmodules.org/
- AngularUI
  - http://angular-ui.github.io/
- Angular Material (Google's Material Design Specification)
  - https://material.angularjs.org/
- Página para pruebas online:
  - Plunker (http://plnkr.co)
  - JSFiddle (http://jsfiddle.net)

Instalación de utilidades

Consideraciones previas

- Las utilidades son de línea de comandos.
- Para ejecutar los comandos es necesario abrir la consola comandos (Símbolo del sistema)
- Siempre que se realice una instalación o creación es conveniente "Ejecutar como Administrador" para evitar otros problemas.
- En algunos casos el firewall de Windows y las aplicaciones antivirus pueden dar problemas.

GIT: Software de control de versiones

- Descargar e instalar:
  - https://git-scm.com/
- Verificar desde consola de comandos:
  - git

Node.js: Entorno en tiempo de ejecución

- **Node.js: Entorno en tiempo de ejecución**
  - Descargar e instalar:
  - https://nodejs.org
  - Verificar desde consola de comandos:
  - node --version
- **npm: Node Package Manager**
  - Aunque se instala con el Node es conveniente actualizarlo:
    - npm update -g npm
  - Verificar desde consola de comandos:
    - npm --version
  - Generar fichero de dependencias package.json:
    - npm init

Express: Infraestructura de aplicaciones web Node.js

- Instalación:
  - npm install -g express-generator
- Generar un servidor y sitio web:
  - express cursoserver
- Descargar dependencias
  - cd cursoserver && npm install
- Levantar servidor:
  - SET DEBUG=cursoserver:* & npm start
- Directorio de cliente de la aplicación web
  - cd cursoserver\public ← Copiar app. web

Bower: Gestor de dependencias del frontend web

- http://bower.io/
- Instalación:
  - npm install -g bower
- Generar fichero de dependencias bower.json:
  - bower init
- Descargar, instalar y registrar dependencia de una librería o framework:
  - bower install jquery -save

Grunt: Automatización de tareas

- Instalación general:
  - npm install -g grunt-cli
  - npm install -g grunt-init
- Instalación local de los módulos en la aplicación (añadir al fichero package.json de directorio de la aplicación o crearlo si no existe):

```
{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-nodeunit": "~0.4.1",
    "grunt-contrib-uglify": "~0.5.0"
    "grunt-shell": "~0.7.0"
  }
}
```


Grunt: Automatización de tareas

- Descargar e instalar localmente los módulos en la aplicación (situarse en el directorio de la aplicación)
  - npm install --save-dev
- Fichero de tareas: gruntfile.js
- Ejecutar tareas:
  - grunt
  - grunt jshint
  - grunt reléase
  - grunt serve

Karma: Gestor de Pruebas unitarias de JavaScript

- Instalación general:
  - npm install -g karma
  - npm install -g karma-cli
- Generar fichero de configuración karma.conf.js:
  - karma init
- Ingenierías de pruebas unitarias disponibles:
  - http://jasmine.github.io/
  - http://qunitjs.com/
  - http://mochajs.org
  - https://github.com/caolan/nodeunit
  - https://github.com/nealxyc/nunit.js

Yeoman: Generador del esqueleto web

- http://Yeoman.io
- Instalación:
  - npm install -g yo
  - npm install -g generator-angular
  - npm install -g generator-karma
- Generar un servidor y sitio web:
  - yo angular
- Descargar dependencias si es necesario
  - bower install & npm install
- Preparar entorno de ejecución y levantar el servidor en modo prueba:
  - grunt serve

AngulaJS Avanzado

Angular 1.5

DIRECTIVAS Y COMPONENTES

Introducción

- Las directivas son marcas en los elementos del árbol DOM, en los nodos del HTML, que indican al compilador de Angular ($compile) que debe asignar cierto comportamiento a dichos elementos o transformarlos según corresponda.
- Podríamos decir que las directivas nos permiten añadir comportamiento dinámico al árbol DOM haciendo uso de las propias del AngularJS o extender la funcionalidad hasta donde necesitemos creando las nuestras propias.
- Según el patrón, la recomendación es que las directivas en el único sitio donde se puede manipular el árbol DOM, para que entre dentro del ciclo de vida de compilación, binding y renderización del HTML.
- La creación de directivas es la técnica que va a permitir crear nuestros propios componentes visuales encapsulando las posibles complejidades en la implementación, normalizando y parametrizándolos según nuestras necesidades.
- Para AngularJS, "compilación" significa fijar directrices al HTML para que sea interactivo.

Ámbito de aplicación

- Las directivas, según su ámbito de aplicación a los nodos del árbol DOM, se pueden asociar mediante :
  - la declaración de un atributo (‘A’) en cualquier elemento del DOM
    ```
    <div my-dir></div>
    ```
  - la declaración de un clase CSS (‘C’) en el atributo class de un elemento del DOM
    ```
    <div class="my-dir"></div>
    ```
  - la asignación a un comentario (‘M’) en cualquier bloque de código comentado
    ```
    <!-- directive: my-dir -->
    ```
  - la declaración de un elemento (‘E’) del propio árbol DOM
    ```
    <my-dir></my-dir>
    ```
- Se puede combinar la declaración de un elemento con la asignación de un atributo ‘EA’ o añadirle también la declaración en un comentario ‘EAC’
- Lo habitual es definir las directivas a nivel de elemento y atributo.

Características del las directivas

- Prefijos en Directivas
  - El equipo de Angular recomienda que las directivas tengan siempre un prefijo de forma que no choquen con otras directivas creadas por otros desarrolladores o con actuales o futuras etiquetas de HTML. Por eso las directivas de AngularJS empiezan siempre por “ng”.
- Nombre
  - El nombre de las directivas utiliza la notación camel sigue el formato de los identificadores en JavaScript (sin -, ng-if → ngIf). Es el nombre que se usa en la documentación de AngularJS y el que se usa cuando se crea una nueva directiva.
- Variaciones del nombre en HTML
  - Se puede usar en la página HTML siguiendo las siguientes reglas:
    - Añadir el prefijo data- o x- para que los validadores HTML sepan que no es estándar.
    - Separar cada palabra con algunos de los siguientes caracteres ”-” , ”:” o “_”.

  ngView → **ng-view**, ng:view, ng_view, **data-ng-view**, data-ng:view, data-ng_view, x-ng-view, x-ng:view, x-ng_view

Crear directivas

- Crear una nueva directivas es definir un objeto que interpreta el $compiler y devuelve un segmento HTML (plantilla que puede incluir otras directivas).
- La forma de crear una directivas es llamando al método directive de un módulo. Este método recibe una función factoría que genera el objeto con la definición de la directiva.

```
app.directive("mySaludo",[function() {
    return {
      restrict:"E",
      replace : true,
      template:"<h1>Hola {{nombre}}</h1>",
      scope:{ nombre:"@" }
    };
}]);

<my-saludo nombre='Mundo'></my-saludo>
```

Definición de la directiva

- **restrict**: Cadena que indica el tipo de ámbito donde puede usarse la directiva: 'A', 'E', M, 'C' o 'EA'. Por defecto 'EA'.
- **template** : Cadena con el HTML por el que se sustituirá la directiva (excluyente con templateUrl).
- **templateUrl** : Cadena con la URL del fichero que contiene el HTML por el que se sustituirá la directiva (excluyente con template).
- **replace**: Si vale false el contenido del template se añadirá dentro del tag de la propia directiva. Pero si vale true se quitará el tag de la directiva y solo estará el contenido del template.
- **transclude**: Indica como se conserva el contenido interno del elemento html cuando se reemplaza.
- **scope**: Ámbito de la directiva respecto al controlador.
- **require**: permite declarar como abligatorios parámetros en la directiva que hacen referencia a otras directivas.
- **controller**: permite declarar un controlador a nivel de la directiva cuando tiene su propio scope (se ejecuta antes de la compilación).
- **link**: Función que manipula el DOM resultante de la compilación de la directiva (se ejecuta después de la compilación).

Scope de la directiva

- Para tener exactamente el mismo scope que hay en el controlador:
  `scope:false`
- Para tener un nuevo scope pero que hereda del scope del controlador (copia):
  `scope:true`
- Para tener un nuevo scope propio que NO hereda del controlador:

```
scope: {
   nombrePublico: "@nombreLocal", otro:"=", ...
}
```

  - Donde el nombre publico es el que se utiliza como atributo de la etiqueta y el nombre local (opcional) se utiliza en la plantilla.
  - La "@" vincula a una propiedad de ámbito local con el valor de un atributo DOM (unidireccional).
  - El "=" indica un enlace bidireccional (en cuyo caso para enlazar a valores literales deben ir delimitados por ''), si =? es opcional.

Función Link

- Para las directivas que no se limitan a expandir la plantilla y desean modificar el DOM se utiliza la función asociada a la opción link que permite registrar eventos del DOM, generar o modificar el DOM mediante código.
- Se ejecuta después de que la plantilla haya sido clonada y es donde reside la lógica de la directiva.
- Si es necesario, es la responsable de generar todo el árbol DOM resultante en las directivas que no cuentan con una plantilla.
- La función link tiene los siguiente parámetros:
  - scope: Es el scope de la directiva
  - element: el elemento del árbol DOM al que está vinculado la directiva.
  - attrs : Array con los pares de nombre y valor de los atributos “normalizados” de la directiva declarados en el elemento HTML.
  - controller: Opcional, para cuando transclude : 'element'.
  - transcludeFn: Opcional, para cuando transclude : 'element'.
- El elemento del árbol DOM (parámetro element) está envuelto por un objeto de la implementación de jQuery del propio Angular (jqLite), a través del cual que se puede manipular el nodo HTML y todos sus hijos antes de renderizarse como si fuera un selector jQuery.

transclude

- Cuando se reemplaza un elemento html por otro usando una directiva, por defecto se reemplaza también el contenido interno de ese elemento html. Para conservar el contenido, se debe poner transclude:true en la directiva.
- En el template, con la directiva ng-transclude se recupera el contenido interno:

```
transclude : true,
template : '<div><p ng-transclude></p></div>'
```

- Para conservar toda la etiqueta se debe poner transclude:'element', en cuyo caso no se puede usar template ni templateUrl ni ng-transclude, se debe generar el nuevo contenido por medio de la función link.

```
transclude : 'element',
link : function (scope, element, attrs, controller, transcludeFn){
     element.after(transcludeFn());
     element.after("<p>Added Element</p>");
}
```

Validación personalizada

- Angular proporciona una implementación básica para los tipos de entrada más común: texto, número, URL, correo electrónico, fecha, radio, casillas de verificación; así como algunas directrices para la validación: required, pattern, minlength, maxlength, min, max.
- Además de los validadores que ngModelController contiene, se le puede indicar funciones propias de validación añadiéndolas al objeto $validators. Como hemos dicho, el parámetro ctrl es una instancia de ngModelController, la correspondiente al modelo de nif en este caso.
- Las funciones que se añadan al objeto $validators reciben los parámetros modelValue y viewValue. Son inyectados y albergan el valor del input en el modelo y en la vista.
- Angular llama internamente a la función $setValidity junto con el valor de respuesta de la función de link (true si el valor es válido y false si el valor es inválido). Cada vez que un input se modifica (hay una llamada a $setViewValue) o cuando el valor del modelo cambia, se llama a las funciones de validación registradas. La validación tiene lugar cuando se ejecutan los parsers y los formatters dados de alta en el controlador (objeto $parsers y objeto $formatters). Las validaciones que no se han podido realizar se almacenan por su clave en ngModelController.error.

Validación personalizada

```
app.directive('valInteger', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      ctrl.$validators.valInteger = function(modelValue, viewValue) {
        if (ctrl.$isEmpty(modelValue)) {
          // tratamos los modelos vacíos como correctos
          return true;
        }
        return /^\-?\d+$/.test(viewValue);
      };
    }
  };
});
<input type="text" name="edad" ng-model="vm.edad" val-integer >
<span ng-show=" miForm.edad.$error.valInteger>Error ...
```

Componentes

- Los componentes (versión 1.5) son una nueva manera de realizar el tipo de trabajo que antes venía realizándose con las directivas de elementos.
- En Angular, un componente es un tipo especial de directiva (directivas componente) que utiliza una configuración simple mas adecuada para una estructura de la aplicación basada en componentes, haciendo más fácil escribir una aplicación basada en el uso de componentes web o utilizando el estilo de arquitectura de la aplicación del Angular 2.
- Ventajas de componentes:
  - configuración más simple que en las directivas
  - promover configuraciones normalizadas y las mejores prácticas
  - optimizado para la arquitectura basada en componentes
  - escribir componente hará más fácil pasar a Angular 2
- Cuando no utilizar componentes:
  - para las directivas que se basan en la manipulación DOM, la adición de los controladores de eventos, etc., debido a que las funciones de compilación y enlace aun no están disponibles
  - cuando se necesita opciones avanzadas de definición de directiva como prioridad, terminal, múltiples elementos
  - cuando se desea una directiva que se desencadene por un atributo o una clase de CSS, en lugar de un elemento

Características de los componentes

- Los componentes sólo controlan su propia Vista y Datos: Los componentes nunca deben modificar cualquier dato o DOM que esté fuera de su propio ámbito.
- Los componentes son elementos aislados con Entradas y Salidas bien definidas: Sólo el componente titular de los datos debería modificarlos, para que sea fácil predecir quienes pueden cambiar los datos y cuando.
  - Las entradas se realizan mediante enlaces @ y <. El símbolo < denota enlaces de un solo sentido (unidireccional) que están disponibles desde 1.5 (no $watch).
  - Las salidas se realizan con enlace &, que son funciones de retorno a los eventos de componentes.
  - En lugar de utilizar el enlace bidireccional, el componente llama al evento de salida correcto con los datos cambiados.
  - De esta manera, el contenedor del componente puede decidir qué hacer con el evento y los datos cambiados.

Ciclo de vida de los componentes

- Los componentes tienen un ciclo de vida bien definido: Cada componente puede implementar "ganchos" del ciclo de vida, métodos que serán llamados en ciertos momentos de la vida del componente.
- Los siguientes métodos de enlace se pueden implementar:
  - $**onInit**(): Invocado en cada controlador después de que todos los controladores en un elemento se han construido y hayan inicializado sus enlaces. Este es un buen lugar para poner el código de inicialización para su controlador.
  - $**onChanges** (changesObj): Invocado cada vez que se actualizan los enlaces de un solo sentido. El changesObj es un hash cuyas claves son los nombres de las propiedades vinculadas que han cambiado, y los valores son objeto de formulario {CurrentValue, PreviousValue, isFirstChange ()}. Utilizar este gancho para desencadenar cambios dentro de un componente, como la clonación del valor determinado para evitar la mutación accidental del valor externo.
  - $**onDestroy**(): Invocado cuando el controlador que le contiene se destruye. Utilizar este gancho para la liberación de los recursos externos, watches y controladores de eventos.
  - $**postLink**(): Invocado después de este elemento controlador y sus hijos se han enlazado. Similar a la función de post-enlace de este gancho se puede utilizar para configurar los controladores de eventos DOM y hacer la manipulación DOM directa. Tenga en cuenta que los elementos secundarios que contienen directivas templateUrl no se han compilado y vinculado, ya que están a la espera de su plantilla para cargar de forma asíncrona y su propia compilación y vinculación ha sido suspendido hasta que eso ocurra.

Crear un componente

- La forma de crear un componente es llamando al método component de un módulo. Este método recibe el objeto, no una función factoría como las directivas, con la definición del componente.

```
app.component('calc', {
    templateUrl : 'calc.html',
    controller : calcController,
    controllerAs : 'ctrl',
    bindings : { init : '<', onUpdate : '&'}
});

<calc init="123" on-update="resulado(rslt)">
</calc>
```

Definición del componente

- **template** : Cadena con la plantilla HTML del componente (excluyente con templateUrl) o una funcion ($element, $attrs) que devuelva una cadena con el contenido HTML de la plantilla.
- **templateUrl** : Cadena con la URL del fichero que contiene la plantilla HTML del componente (excluyente con template) o una funcion ($element, $attrs) que devuelva una cadena con la URL de la plantilla.
- **controller**: Función constructora del controlador o una cadena con el nombre de un controlador registrado.
- **controllerAs**: nombre identificador para hacer referencia al scope del componente, por defecto es $ctrl.
- **transclude**: Indica si se conserva el contenido interno de la etiqueta cuando se reemplaza. Deshabilitado por defecto.
- **bindings**: Define enlaces entre los atributos DOM y propiedades de los componentes.

Bindings

- La propiedades enlazadas al controlador del componente y no al scope.

```
bindings: {
     init: '<', onUpdate: '&'
}
```

- La transformación de la notación Camel de JavaScript a HTML que no distingue ente mayúsculas y minúsculas es:
  - `onUpdate` ← `on-update, ON-UPDATE, …`
- Los enlaces & se definen como una cadena con la firma del método en el controlador contenedor a invocar por el componente:

```
<calc on-update="vm.resulado(param1,param2)">
```

- El controlador del componente (o su plantilla) invocan la función vinculada con un único argumento: un array con la firma de los parámetros del método:

```
this.onUpdate({param1:'valor',param2:rslt});
```

Introducción a Angula2

Novedades

- Escalabilidad estaba limitada por:
  - Lenguaje JavaScript
  - Precarga del código.
- TypeScript: validación, intellisense y refactoring.
- Desarrollo basado en componentes.
- Anotaciones (metadatos), declarativa frente a imperativa.
- Doble binding, reconstruido según el patrón observable, las aplicaciones son hasta cinco veces más rápidas.
- La inyección de dependencias, carga dinámica de modulos.
- Aparición de Angular Universal, podemos ejecutar Angular en el servidor.
- Aplicaciones híbridas, solución low cost para aplicaciones para móviles.
- Está cada vez más orientado a grandes desarrollos empresariales.

TypeScript

- El lenguaje TypeScript fue ideado por Anders Hejlsberg –autor de Turbo Pascal, Delphi, C# y arquitecto principal de .NET- como un superconjunto de JavaScript que permitiese utilizar tipos estáticos y otras características de los lenguajes avanzados como la Programación Orientada a Objetos.
- TypeScript es un lenguaje Open Source basado en JavaScript y que se integra perfectamente con otro código JavaScript, solucionando algunos de los principales problemas que tiene JavaScript:
  - Falta de tipado fuerte y estático.
  - Falta de "Syntactic Sugar" para la creación de clases
  - Falta de interfaces (aumenta el acoplamiento ya que obliga a programar hacia la implementación)
  - Módulos (parcialmente resuelto con require.js, aunque está lejos de ser perfecto)
- TypeScript es un lenguaje con una sintaxis bastante parecida a la de C# y Java, por lo que hace fácil para los desarrolladores con experiencia en estos lenguajes el aprender TypeScript.

TypeScript

- Lo que uno programa con TypeScript, tras grabar los cambios, se convierte en JavaScript perfectamente ejecutable en todos los navegadores actuales (y antiguos), pero habremos podido:
  - abordar desarrollos de gran complejidad,
  - organizando el código fuente en módulos,
  - utilizando características de auténtica orientación a objetos, y
  - disponer de recursos de edición avanzada del código fuente (Intellisense, compleción de código, "snippets", etc.), tanto en Visual Studio, como en otros editores populares, como Sublime Text, Eclipse, WebStorm, etc., cosa impensable hasta este momento, pero factible ahora mismo gracias a los "plug-in" de TypeScript disponibles para estos editores.

TypeScript

- Amplia la sintaxis del JavaScript con la definición y comprobación de:
  - Tipos básicos
    - booleans, number, string, Any y Void.
  - Clases e interfaces
  - Herencia
  - Módulos
  - Genéricos
  - Anotaciones
- Otra de las ventajas que otorga Typescript es que permite comunicarse con código JavaScript ya creado e incluso añadirle "tipos" a través de unos ficheros d.ts que indican los tipos que reciben y devuelven las funciones de una librería.
- Ya existen fichero de definiciones para la mayoría de los framework mas populares (http://definitelytyped.org/).

TypeScript

- El TypeScript se traduce ("transpila": compilar un lenguaje de alto nivel a otro de alto nivel) a JavaScript cumpliendo el estándar ECMAScript totalmente compatible con las versiones existentes.
- De hecho, el "transpilador" deja elegir la versión ECMAScript del resultado, permitiendo adoptar la novedades mucho antes de que los navegadores las soporten, basta dejar el resultado en la versión mas ampliamente difundida.
- El equipo de TypeScript provee una herramienta de línea de comandos para hacer esta compilación, se encuentra en npm y se puede instalar con el siguiente comando:
  - npm install -g typescript
- Podemos usarlo escribiendo
  - tsc helloworld.ts
- Los diversos "plug-in" se pueden descargar en:
  - https://www.typescriptlang.org