ESCUELA:
TECNOLOGÍA

ANGULAR.JS NIVEL EXPERTO

indra

ÍNDICE

- Conceptos avanzados
  - Directivas personalizadas
  - Eventos
  - $templateCache
  - Routing
  - Animaciones
- Servidor (Patrones y recursos)
  - Promise
  - $q
  - $http
  - $resource
- Testing
  - Test unitarios
  - Test E2E
- Angular.JS y navegadores antiguos (IE)
- Novedades Angular 1.3.x

DESARROLLO AVANZADO WEB CON ANGULAR JS

ESCUELA:
TECNOLOGÍA

- NOMBRE APELLIDO PROFESOR
  Javier Martín
- VER PERFIL COMPLETO:
- CONTACTO

  jmartin@grupoloyal.com

Indra Open University

OBJETIVO ESTRATÉGICO:

ESCUELA:
TECNOLOGÍA

ACCION FORMATIVA: Angular.JS nivel experto

Dirigido a:	Profesional de la compañía
Requisitos:	Conocimientos básicos de Angular.JS y conocimientos avanzados de Javascript, Ajax, HTML y CSS.
Tipo de Formación: Metodología	Presencial
Duración:	20 horas

OBJETIVOS:

El objetivo del curso es que un alumno con conocimientos básicos de Angular.JS sea capaz de desarrollar una aplicación SPA (Single Page Application) y mejorar su rendimiento y optimizarla para navegadores antiguos. También sea capaz de comunicar la aplicación con el servidor a través de peticiones AJAX. Al finalizar el curso el alumno también tendrá los conocimientos básicos para poder testear las aplicaciones angular de forma automática a través de test unitarios y E2E.

PRESENTACIÓN:

Curso Angular.JS nivel experto

CONTENIDOS:

- Conceptos avanzados
  - Directivas personalizadas
  - Eventos
  - $templateCache
  - Routing
- Animaciones
- Servidor (Patrones y recursos)
  - Promise
  - $q
  - $http
  - $resource
- Testing
  - Test unitarios
  - Test E2E
- Angular.JS y navegadores antiguos (IE)
- Novedades Angular 1.3.x

Indra Open University

AngulaJS Avanzado

HERRAMIENTAS DE DESARROLLO

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, la configuración del proxy 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

- 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
- Configuración:
  - npm config edit
  - proxy=http://usr:pwd@proxy.dominion.com:8080 ← Símbolos: %HEX ASCII
- Generar fichero de dependencias package.json:
  - npm init
- Instalación de paquetes:
  - npm install -g grunt-cli grunt-init karma karma-cli ← Global (CLI)
  - npm install karma jasmine-core jshint lite-server --save --save-dev
  - npm install ← Dependencias en package.json
- Arranque del servidor:
  - npm start

lite-server

- Servidor de desarrollo ligero en Node para una única aplicación web, la abre en el navegador, vigila los fuentes HTML, Javascript y CSS, notificando al navegador los cambios usando sockets para que refresque la página. Tiene una página de respaldo cuando no se encuentra una ruta.
- Instalación:
  - npm install -g lite-server
- Configurar creando el fichero bs-config.json:

```
{
  "port": 8000,
  "files": ["./src/**/*.{html,htm,css,js}"],
  "server": { "baseDir": "./src" }
}
```

- Para arrancar el servidor:
  - lite-server

TEST UNITARIOS

Ingeniería de Software

- El JavaScript es un lenguaje muy poco apropiado para trabajar en un entorno de calidad de software.
- En descargo del lenguaje JavaScript y de su autor, Brendan Eich, hay que decir que los problemas que han forzado esta evolución del lenguaje (así como las críticas ancestrales de la comunidad de desarrolladores) vienen dados por lo que habitualmente se llama "morir de éxito".
- Jamás se pensó que un lenguaje que Eich tuvo que montar en 12 días como una especie de "demo" para Mozilla, pasase a ser omnipresente en miles de millones de páginas Web.
- O como el propio Hejlsberg comenta:

  "JavaScript se creó –como mucho- para escribir cien o doscientas líneas de código, y no los cientos de miles necesarias para algunas aplicaciones actuales."

Principios fundamentales

- Las pruebas exhaustivas no son viables
- El proceso de pruebas no puede demostrar la ausencia de defectos
- Las pruebas no garantizan ni mejoran la calidad del software
- Las pruebas tienen un coste
- Hay que ejecutar las pruebas bajo diferentes condiciones
- Inicio temprano de pruebas

Desarrollo Guiado por Pruebas (TDD)

- El Desarrollo Guiado por Pruebas, es una técnica de programación (definida por KentBeck); consistente en desarrollar primero el código que pruebe una característica o funcionalidad deseada antes que el código que implementa dicha funcionalidad.
- El objetivo a lograr es que no exista ninguna funcionalidad que no esté avalada por una prueba.
- Lo primero que hay que aprender de TDD son sus reglas básicas:
  - No añadir código sin escribir antes una prueba que falle
  - Eliminar el Código Duplicado empleando Refactorización

Ritmo TDD

- TDD invita a seguir una serie de tareas ordenadas, que a menudo se denomina ritmo TDD, y que se basa en los siguientes pasos:
  1. Escribir una prueba que demuestre la necesidad de escribir código.
  2. Escribir el mínimo código para que el código de pruebas compile
  3. Implementar exclusivamente la funcionalidad demandada por las pruebas
  4. Mejorar el código (Refactoring) sin añadir funcionalidad
  5. Volver al primer paso
- Este ritmo permite formalizar las tareas que se han de realizar para conseguir un código fácil de mantener, bien diseñado y que se puede probar automáticamente.

Las tres partes del test: AAA

- Arrange (Preparar):
  - Inicializa objetos y establece el valor de los datos que se pasa al método en pruebas de tal forma que los resultados sean predecibles.
- Act (Actuar)
  - Invoca al método a probar con los parámetros preparados.
- Assert (Afirmar)
  - Comprobar si la acción del método probado se comporta de la forma prevista. Puede tomar la forma de:
    - Aserción: Es una afirmación que se hace sobre el resultado y puede ser cierta o no.
    - Expectativa: Es la expresión del resultado esperado que puede cumplirse o no.

Arrange (Preparar)

- **Fixture**: Es el término se utiliza para hablar de los datos de contexto de las pruebas, aquellos que se necesitan para construir el escenario que requiere la prueba.
- **Dummy**: Objeto que se pasa como argumento pero nunca se usa realmente. Normalmente, los objetos dummy se usan sólo para rellenar listas de parámetros.
- **Fake**: Objeto que tiene una implementación que realmente funciona pero, por lo general, usa una simplificación que le hace inapropiado para producción (como una base de datos en memoria por ejemplo).
- **Stub**: Objeto que proporciona respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. Los stubs pueden también grabar información sobre las llamadas.
- **Mock**: Objeto preprogramado con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. Son más complejos que los stubs aunque sus diferencias son sutiles.

Beneficios de TDD

- Reducen el número de errores y bugs ya que éstos, aplicando TDD, se detectan antes incluso de crearlos.
- Facilitan entender el código y que, eligiendo una buena nomenclatura, sirven de documentación.
- Facilitan mantener el código:
  - Protege ante cambios, los errores que surgen al aplicar un cambio se detectan (y corrigen) antes de subir ese cambio.
  - Protegen ante errores de regresión (rollbacks a versiones anteriores).
  - Dan confianza.
- Facilitan desarrollar ciñéndose a los requisitos.
- Ayudan a encontrar inconsistencias en los requisitos
- Ayudan a especificar comportamientos
- Ayudan a refactorizar para mejorar la calidad del código (Clean code)
- A medio/largo plazo aumenta (y mucho) la productividad.

JSLint y JSHint

- Los analizadores de código son herramientas que realizan la lectura del código fuente y devuelve observaciones o puntos en los que tu código puede mejorarse desde la percepción de buenas prácticas de programación y código limpio.
- JSHint es un analizador online de código JavaScript (basado en el JSLint creado por Douglas Crockford) que nos permitirá mostrar puntos en los que tu código no cumpla unas determinadas reglas establecidas de “código limpio“.
- El funcionamiento de JSHint es el siguiente: toma nuestro código, lo escanea y, si encuentra un problema, devuelve un mensaje describiéndolo y mostrando su ubicación aproximada.
- Para descargar e instalar:
  - npm install -g jshint
- Existen “plug-in” para la mayoría de los entornos de desarrollo (http://jshint.com). Se puede automatizar con GRUNT o GULP.

Jasmine

- Jasmine es un framework de desarrollo dirigido por comportamiento (behavior-driven development, BDD) para probar código JavaScript.
  - No depende de ninguna otra librería JavaScript.
  - No requiere un DOM.
  - Tiene una sintaxis obvia y limpia para que se pueda escribir pruebas fácilmente.
- Prácticamente se ha convertido en el estándar de facto para el desarrollo con JavaScript.
- Para su instalación “standalone”, descargar y descomprimir:
  - https://github.com/jasmine/jasmine/releases
- Mediante npm:
  - npm install -g jasmine

SpecRunner.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Jasmine Spec Runner v2.5.0</title>
  <link rel="shortcut icon" type="image/png" href="lib/jasmine-2.5.0/jasmine_favicon.png">
  <link rel="stylesheet" href="lib/jasmine-2.5.0/jasmine.css">
  <script src="lib/jasmine-2.5.0/jasmine.js"></script>
  <script src="lib/jasmine-2.5.0/jasmine-html.js"></script>
  <script src="lib/jasmine-2.5.0/boot.js"></script>
  <script type="text/javascript" src="angular.js"></script>
  <script type="text/javascript" src="angular-mocks.js"></script>
  <!-- include source files here... -->
  <script src="src/Player.js"></script>
  <script src="src/Song.js"></script>
  <!-- include spec files here... -->
  <script src="spec/SpecHelper.js"></script>
  <script src="spec/PlayerSpec.js"></script>
</head>
<body></body>
</html>
```

Suites

- Una “suite” es un nombre que describe al género o sección qué se va a pasar por un conjunto de pruebas unitarias, además es una herramienta que es el núcleo que se necesita para poder tener un orden en el momento de crear las pruebas.
- Las “suites” se crean con la función **describe**, que es una función global y con la cual se inicia toda prueba unitaria, además consta con dos parámetros:

```
describe("Una suite es sólo una función", function() {
//…
});
```

- El primer parámetro es una cadena de caracteres donde se define el nombre descriptivo de la prueba unitaria.
- El segundo parámetro es una función con el código que ejecutará la prueba de código.
- Se pueden anidar “suites” para estructurar conjuntos complejos y facilitar la legibilidad y la búsqueda, basta con crear un describe dentro de otro.

Especificaciones

- Una especificación contiene una o más expectativas (algo que se espera) que ponen a prueba el estado del código. Una expectativa de Jasmine es una afirmación que debe ser verdadera pero puede ser falsa.
- Una especificación con todas las expectativas verdaderas es una especificación que pasa la prueba, pero con una o más falsas es una especificación que falla.
- Las especificaciones se definen dentro de una Suite llamando a la función global del Jasmine **it** que, al igual que describe, recibe una cadena y una función. La cadena es el título de la especificación y la función es la especificación o prueba.
  - it("y así es una especificación", function() {
  - //…
  - });
- **describe** y **it** son funciones: pueden contener cualquier código ejecutable necesario para implementar la prueba y las reglas de alcance de JavaScript se aplican, por lo que las variables declaradas en un describe están disponibles para cualquier bloque it dentro de la suite.

Expectativas

- Las expectativas se construyen con la función expect que obtiene un valor real de una expresión y lo comparan mediante una función Matcher con un el valor esperado (constante).

  expect(valor actual).matchers(valor esperado);
- Los matchers son funciones que implementan comparaciones booleanas entre el valor actual y el esperado, ellos son los responsables de informar a Jasmine si la expectativa se cumple o es falsa.
- Cualquier matcher puede evaluarse como una afirmación negativa mediante el encadenamiento a la llamada expect de un not antes de llamar al matcher.

  expect(valor actual)**.not**.matchers(valor esperado);
- También existe la posibilidad de escribir matchers personalizados para cuando el dominio de un proyecto consiste en afirmaciones específicas que no se incluyen en los ya definidos.

Matchers

- .toEqual(y); verifica si ambos valores son iguales ==.
- .toBe(y); verifica si ambos objetos son idénticos ===.
- .toMatch(pattern); verifica si el valor pertenece al patrón establecido.
- .toBeDefined(); verifica si el valor está definido.
- .toBeUndefined(); verifica si el valor es indefinido.
- .toBeNull(); verifica si el valor es nulo.
- .toBeNaN(); verifica si el valor es NaN.
- .toBeCloseTo (n, d); verifica la precisión matemática (número de decimales).
- .toContain(y); verifica si el valor actual contiene el esperado.

Matchers

- .toBeTruthy(); verifica si el valor es verdadero.
- .toBeFalsy(); verifica si el valor es falso.
- .toBeLessThan(y); verifica si el valor actual es menor que el esperado.
- .toBeLessThanOrEqual (y); verifica si el valor actual es menor o igual que el esperado.
- .toBeGreaterThan(y); verifica si el valor actual es mayor que el esperado.
- .toBeGreaterThanOrEqual (y); verifica si el valor actual es mayor o igual que el esperado.
- .toThrow(); verifica si una función lanza una excepción.
- .toThrowError(e); verifica si una función lanza una excepción específica.

Forzar fallos

- La función fail(msg) hace que una especificación falle. Puede llevar un mensaje de fallo o error de un objeto como un parámetro.

```
describe("Una especificación utilizando la función a prueba", function() {
  var foo = function(x, callBack) {
    if (x) {
      callBack();
    }
  };
  it("no debe llamar a la devolución de llamada", function() {
    foo(false, function() {
      fail("Devolución de llamada ha sido llamada");
    });
  });
});
```

Montaje y desmontaje

- Para montar el escenario de pruebas suele ser necesario definir e inicializar un conjunto de variables. Para evitar la duplicidad de código y mantener las variables inicializadas en un solo lugar además de mantener la modularidad, Jasmine suministra las funciones globales :
  - beforeAll(fn) se ejecuta solo una vez antes empezar a ejecutar las especificaciones del "describe".
  - beforeEach(fn) se ejecuta antes de cada especificación dentro del "describe".
  - afterEach(fn) se ejecuta después de cada especificación dentro del "describe".
  - afterAll(fn) se ejecuta solo una vez después de ejecutar todas las especificaciones del "describe".

```
describe("operaciones aritméticas", function(){
    var cal;
    beforeEach(function(){ calc = new Calculadora(); });
    it("adicion", function(){ expect(calc.suma(4)).toEqual(4); });
    it("multiplicacion", function(){ expect(calc.multiplica(7)).toEqual(0); });
    // …
```

- Otra manera de compartir las variables entre una beforeEach, it y afterEach es a través de la palabra clave this. Cada expectativa beforeEach/it/afterEach tiene el mismo objeto vacío this que se restablece de nuevo a vacío para de la siguiente expectativa beforeEach/it/afterEach.

Desactivación parcial

- Las Suites se pueden desactivar renombrando la función describe por xdescribe. Estas suites y las especificaciones dentro de ellas se omiten cuando se ejecuta y por lo tanto sus resultados no aparecerán entre los resultados de la prueba.
- De igual forma, las especificaciones se desactivan renombrando it por xit, pero en este caso aparecen en los resultados como pendientes (pending).
- Cualquier especificación declarada sin un cuerpo función también estará marcada pendiente en los resultados.

```
it('puede ser declarada con "it", pero sin una función');
```

- Y si se llama a la función de pending en cualquier parte del cuerpo de las especificaciones, independientemente de las expectativas, la especificación quedará marcada como pendiente. La cadena que se pasa a pending será tratada como una razón y aparece cuando termine la suite.

```
it('se puede llamar a "pending"en el cuerpo de las especificaciones', function() {
    expect(true).toBe(false);
    pending('esto es por lo que está pendiente');
});
```

Ejecución de pruebas específicas

- En determinados casos (desarrollo) interesa limitar las pruebas que se ejecutan. Si se pone el foco en determinadas suites o especificaciones solo se ejecutaran las pruebas que tengan el foco, marcando el resto como pendientes.
- Las Suites se enfocan renombrando la función describe por fdescribe. Estas suites y las especificaciones dentro de ellas son las que se ejecutan.
- De igual forma, las especificaciones se enfocan renombrando it por fit.
- Si se enfoca una suite que no tiene enfocada ninguna especificación se ejecutan todas sus especificaciones, pero si tiene alguna enfocada solo se ejecutaran las que tengan el foco.
- Si se enfoca una especificación se ejecutara independientemente de que su suite esté o no enfocada.
- La funciones de montaje y desmontaje se ejecutaran si la suite tiene alguna especificación con foco.
- Si ninguna suite o especificación tiene el foco se ejecutaran todas las pruebas normalmente.

Espías

- Jasmine tiene funciones dobles de prueba llamados espías. Los espías de Jasmine permiten interceptar y sustituir métodos de los objetos.
- Mediante el espía se sustituye el método asíncrono de tal manera que cualquier llamada al mismo recibe una promesa resuelta de inmediato con un valor de prueba (stub).
- El espía no pasa por el método real, por lo tanto no entra en contacto con el servidor.
- En lugar de crear un objeto sustituto, se inyecta el verdadero objeto y se sustituye el método crítico con un espía Jasmine.

Espías

- Un espía puede interceptar cualquier función y hacer un seguimiento a las llamadas y todos los argumentos.

```
beforeEach(function() {
  fnc = spyOn(calc, 'suma');
  prop = spyOnProperty(calc, 'pantalla', 'set')
});
```

- Un espía sólo existe en el bloque describe o it en que se define, y se eliminará después de cada especificación.
- Hay comparadores (matchers) especiales para interactuar con los espías.
  - .toHaveBeenCalled() pasará si el espía fue llamado.
  - .toHaveBeenCalledTimes(n) pasará si el espía se llama el número de veces especificado.
  - .toHaveBeenCalledWith(…) pasará si la lista de argumentos coincide con alguna de las llamadas grabadas a la espía.
  - .toHaveBeenCalledBefore(esperado): pasará si el espía se llama antes que el espía pasado por parámetro.

Seguimiento de llamadas

- El proxi del espía añade la propiedad calls que permite:
  - all(): Obtener la matriz de llamadas sin procesar para este espía.
  - allArgs(): Obtener todos los argumentos para cada invocación de este espía en el orden en que fueron recibidos.
  - any(): Comprobar si se ha invocado este espía.
  - argsFor(índice): Obtener los argumentos que se pasaron a una invocación específica de este espía.
  - count(): Obtener el número de invocaciones de este espía.
  - first(): Obtener la primera invocación de este espía.
  - mostRecent(): Obtener la invocación más reciente de este espía.
  - reset(): Restablecer el espía como si nunca se hubiera llamado.
  - saveArgumentsByValue(): Establecer que se haga un clon superficial de argumentos pasados a cada invocación.

```
spyOn(foo, 'setBar');
expect(foo.setBar.calls.any()).toEqual(false);
foo.setBar();
expect(foo.setBar.calls.count()).toBy(1);
```

Cambiar comportamiento

- Adicionalmente el proxi del espía puede añadir los siguientes comportamiento:
  - callFake(fn): Llamar a una implementación falsa cuando se invoca.
  - callThrough(): Llamar a la implementación real cuando se invoca.
  - exec(): Ejecutar la estrategia de espionaje actual.
  - identity(): Devolver la información de identificación para el espía.
  - returnValue(valor): Devolver un valor cuando se invoca.
  - returnValues(... values): Devolver uno de los valores especificados (secuencialmente) cada vez que se invoca el espía.
  - stub(): No haga nada cuando se invoca. Este es el valor predeterminado.
  - throwError(algo): Lanzar un error cuando se invoca.

```
spyOn(foo, "getBar").and.returnValue(745);
spyOn(foo, "getBar").and.callFake(function(arguments, can, be, received) {
  return 745;
});
spyOn(foo, "forbidden").and.throwError("quux");
```


Karma

- Karma es una herramienta de línea de comandos JavaScript que se puede utilizar para generar un servidor web que carga el código fuente de la aplicación y ejecuta sus pruebas.
- Puede configurar Karma para funcionar contra una serie de navegadores, que es útil para estar seguro de que su aplicación funciona en todos los navegadores que necesita soportar.
- Karma se ejecuta en la línea de comandos y mostrará los resultados de sus pruebas en la línea de comandos una vez que se ejecute en el navegador.
- Karma es una aplicación NodeJS, y debe ser instalado a través de npm:
  - npm install karma karma-jasmine jasmine-core karma-chrome-launcher --save-dev
  - npm install -g karma-cli

Configuración

- Creación del fichero karma.config.js
  - karma init
- Parámetros de configuración:

```
basePath: '.',
frameworks: ['jasmine'],
plugins: [ 'karma-chrome-launcher', 'karma-jasmine', ],
files: [
    'app/js/vendors/angular/angular.js',
    'app/js/vendors/angular/angular-mocks.js',
    'app/**/*.js',
    'test/**/*.js',
  ],
port: 9876,
browsers: ['Chrome'],
```

- Lanzar el servidor:
  - karma start

Generar informes de cobertura de código

- Por defecto Karma nos proporciona un informe al que llama progress, que no es más que la salida por consola que nos indica qué ha sucedido con nuestros tests.
- Podemos añadir otros, como karma-coverage: con el que generaremos informes visuales (html) mostrando el porcentaje de código cubierto por nuestros tests.
- Para instalar los generadores:
  - npm install karma-coverage --save-dev
- Una vez configurado el fichero karma.conf.js se arranca con:
  - karma start karma.conf.js --reporters progress,coverage

Cobertura de código

- Los informes de cobertura de código nos permiten ver que parte del código ha sido o no probada adecuadamente por nuestras pruebas unitarias.
- Una vez que las pruebas se completen, aparecerá una nueva carpeta /coverage en el proyecto. Si se abre el archivo index.html en el navegador se debería ver un informe con el código fuente y los valores de cobertura del código.
- Usando los porcentajes de cobertura del código, podemos establecer la cantidad de código (instrucciones, líneas, caminos, funciones) que debe ser probado. Depende de cada organización determinar la cantidad de código que deben cubrir las pruebas unitarias.

Generar informes de cobertura de código

- En el fichero karma.conf.js añadiremos o modificaremos las entradas:

```
plugins: [
    …
    'karma-coverage'
],
reporters: ['progress', 'coverage'],
preprocessors: {
    'app/js/**/*.js': ['coverage']
},
coverageReporter: {
    type : 'html',
    dir: 'coverage/',
},
```

ngMock

- La regla de oro de las pruebas unitarias, es que una unidad (unit) tiene que ser testeada sin utilizar ninguna de sus dependencias.
- Eso quiere decir que si una unidad tiene dependencias hay que reemplazarlas por mocks.
- Angular también proporciona el módulo ngMock, que proporciona simuladores para sus pruebas.
- Esto se utiliza para inyectar y simular servicios de Angular dentro de las pruebas de unidad.
- Además, es capaz de extender otros módulos por lo que son síncronos.
- Tener pruebas síncronas los mantiene mucho más limpio y más fácil de trabajar.
- Aunque es un módulo del core es necesario descargarlo por separado y cargarlo en el entorno de pruebas después del modulo principal dado que manipula determinadas funcionalidades:

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


Preparación de la prueba

- La función **module** permite cargar los módulos para que estén disponibles para la prueba en Jasmine.

```
beforeEach(module("myApp"));
```

- La función **inject** permite mediante la inyección de dependencias la obtención de los servicios necesarios para la prueba:

```
var $controller;
beforeEach(inject(function(_$controller_) {
  // El inyector desenvuelve los guiones bajos (_) de alrededor de
       los nombres de los parámetros cuando se compara
  $controller = _$controller_;
}));
 it("Una sola vez", inject(function($filter) { … }));
```

Controlador

```
angular.module("myApp").controller('myController', function($scope) {
        var vm = this;
        vm.nombre = 'MUNDO';
        vm.resultado = '';

        vm.saluda = function() {
                vm.resultado = "Hola " + vm.nombre;
        };
        vm.despide = function() {
                vm.resultado = "Adios " + vm.nombre;
        };
}]);
```


Comprobación de un controlador

```
describe("Probar Controlador: myController", function() {
  beforeEach(module("myApp"));

  var scope, ctrl;
  beforeEach(inject(function($rootScope, $controller) {
      scope = $rootScope.$new();
      ctrl = $controller("myController", {$scope:scope});
  }));
  it("debe estar definida un model nombre con valor 'MUNDO'", function() {
    expect(ctrl.nombre).toBeDefined();
    expect(ctrl.nombre).toBe("MUNDO");
  });
  it("prueba el método saluda", function() {
    ctrl.nombre = 'javi';
    ctrl.saluda();
    expect(ctrl.resultado).toBe("Hola javi");
  });
  it("prueba el método despide", function() {
    ctrl.despide();
    expect(ctrl.resultado).toBe("Adios MUNDO");
  });
});
```

Filtro

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


Comprobación de un filtro

```
describe("Pruebas de filtros", function() {
  describe("Filtro: capitalice", function() {
    beforeEach(module("core"));

    var filtro;
    beforeEach(inject(function($filter) {
       filtro = $filter('capitalice');
    }));
    it("Sin valor", function() {
      expect(filtro('')).toBe('');
      expect(filtro()).toBeUndefined();
    });
    it("Con todo en mayúsculas", function() { expect(filtro('HOLA MUNDO')).toBe("Hola mundo"); });
    it("Con todo en minúsculas", function() { expect(filtro('hola mundo')).toBe("Hola mundo");  });
    it("Una letra", function() { expect(filtro('a')).toBe('A'); });
    it("Un numero", function() { expect(filtro('1234')).toBe('1234'); });
  });
});
```

Servicio

```
angular.module("myApp").factory('auth', [
function() {
        return {
                usuario : '(vacio)',
                nivel: '',
                roles: []
        };
} ]);
```


Comprobación de un servicio

```
describe("Pruebas de servicios", function() {
   beforeEach(module("myApp"));

   var srv;
   beforeEach(inject(function(auth) {
      srv = auth;
   }));
   it("Sin roles", function() {
     expect(srv.roles).toBeDefined();
     expect(srv.roles.length).toBe(0);
   });
   it("Sin usuario", function() {
     expect(srv.usuario).toBe("(vacio)");
   });
});
```

Directiva

```
angular.module("myApp").directive('aGreatEye',
function () {
    return {
        restrict: 'E',
        replace: true,
        template: '<h1>lidless, wreathed in flame, {{1 +
1}} times</h1>'
    };
});
```


Comprobación de una directiva

```
describe('Pruebas de directivas', function() {
  var $compile, $rootScope;

  beforeEach(module('myApp'));

  beforeEach(inject(function(_$compile_, _$rootScope_){
    $compile = _$compile_;
    $rootScope = _$rootScope_;
  }));

  it('Reemplaza el elemento con el contenido apropiado', function() {
    // Compilar el trozo de HTML que contiene la directiva
    var element = $compile("<a-great-eye></a-great-eye>")($rootScope);
    // Disparar todos los watches, por lo que la expresión {{1 + 1}} sera evaluada
    $rootScope.$digest();
    // Comprobar que el elemento compilado contiene el contenido de la plantilla
    expect(element.html()).toContain("lidless, wreathed in flame, 2 times");
  });
});
```

Componente

```
function HeroDetailController() {
  var ctrl = this;

  ctrl.delete = function() {
    ctrl.onDelete({hero: ctrl.hero});
  };

  ctrl.update = function(prop, value) {
    ctrl.onUpdate({hero: ctrl.hero, prop: prop, value: value});
  };
}

angular.module('heroApp').component('heroDetail', {
  templateUrl: 'heroDetail.html',
  controller: HeroDetailController,
  bindings: {
    hero: '<',
    onDelete: '&',
    onUpdate: '&'
  }
});
```


Comprobación de un componente

```
describe('component: heroDetail', function() {
  var $componentController;

  beforeEach(module('heroApp'));
  beforeEach(inject(function(_$componentController_) {
    $componentController = _$componentController_;
  }));

  it('should expose a `hero` object', function() {
    // Here we are passing actual bindings to the component
    var bindings = {hero: {name: 'Wolverine'}};
    var ctrl = $componentController('heroDetail', null, bindings);

    expect(ctrl.hero).toBeDefined();
    expect(ctrl.hero.name).toBe('Wolverine');
  });
```

$httpBackend

- El servicio $httpBackend del nucleo, como su nombre indica, proporciona un servicio de bajo nivel equivalente a un backend para interactuar con XMLHttpRequest.
- Este servicio evita los problemas relacionados con las incompatibilidades entre los navegadores y los detalles para permitir las peticiones entre dominios diferentes utilizando JSONP.
- Sobre el servicio $httpBackend está situado el $http.

- El módulo ngMock contiene un servicio con el mismo nombre ($httpBackend) que permite realizar simulaciones relacionadas con las llamadas HTTP, obtener respuestas sin realizar realmente la petición.
- Cuando se carga el script de ngMock después del script del núcleo, sobrescribe el servicio $httpBackend del core con el de ngMock.

$httpBackend

- $httpBackend permite asociar respuestas directas a peticiones concretas:

```
var $httpBackend, authRequestHandler;;
beforeEach(module('MyApp'));
beforeEach(inject(function($injector) {
    $httpBackend = $injector.get('$httpBackend');
    $httpBackend.when('GET', '/auth.py')
      .respond({userId: 'userX'}, {'A-Token': 'xxx'});
});
```

- Cuando un controlador o servicio realiza una petición a través de $http obtiene la respuesta registrada en $httpBackend.
- Dado que las peticiones $http son asíncronas, el método flush() permite simular la llegada de la respuesta:

```
ctrl = $controller("myController", {$scope:scope});
ctrl.cargar();
$httpBackend.flush();
expect(ctrl.listado.lenght).toBe(4);
```

http://www.protractortest.org/

TEST E2E

Introducción

- A medida que las aplicaciones crecen en tamaño y complejidad, se vuelve imposible depender de pruebas manuales para verificar la corrección de las nuevas características, errores de captura y avisos de regresión.
- Las pruebas unitarias son la primera línea de defensa para la captura de errores, pero a veces las circunstancias requieran la integración entre componentes que no se pueden capturar en una prueba unitarias .
- Las pruebas de extremo a extremo (E2E: end to end) se hacen para encontrar estos problemas.
- El equipo de Angular ha desarrollado Protractor que simula las interacciones del usuario con el interfaz (navegador) y que ayudará a verificar el estado de una aplicación Angular.
- Protractor es una aplicación Node.js para ejecutar pruebas de extremo a extremo que también están escritas en JavaScript y se ejecutan con el propio Node.
- Protractor utiliza WebDriver para controlar los navegadores y simular las acciones del usuario.

Introducción

- Protractor utiliza Jasmine para su sintaxis prueba.
- Al igual que en las pruebas unitarias, el archivo de pruebas se compone de uno o más bloques describe de it que describen los requisitos de su aplicación.
- Las bloques it están hechos de comandos y expectativas .
- Los comandos indican a Protractor que haga algo con la aplicación, como navegar a una página o hacer clic en un botón.
- Las expectativas indican a Protractor afirmaciones sobre algo acerca del estado de la aplicación, tales como el valor de un campo o la URL actual.
- Si alguna expectativa dentro de un bloque it falla, el ejecutor marca en it como "fallido" y continúa con el siguiente bloque.
- Los archivos de prueba también pueden tener bloques beforeEach y afterEach, que se ejecutarán antes o después de cada bloque it, independientemente de si el bloque pasa o falla.

Instalación

- Se utiliza npm para instalar globalmente Protractor:
  - npm install -g protractor
- Esto instalará dos herramientas de línea de comandos, protractor y WebDriver-manager, para asegurarse de que está funcionando.
  - protractor –versión
- El WebDriver-Manager es una herramienta de ayuda para obtener fácilmente una instancia de un servidor en ejecución Selenium. Para descargar los binarios necesarios:
  - webdriver-manager update
- Para poner en marcha el servidor:
  - webdriver-manager start
- Las pruebas Protractor enviarán solicitudes a este servidor para controlar un navegador local, el servidor debe estar en funcionamiento durante todo el proceso de pruebas.
- Se puede ver información sobre el estado del servidor en:
  - http://localhost:4444/wd/hub

Configuración y Ejecución

- Se configura un fichero con las pruebas a realizar:

```
// Fichero: e2e.conf.js
  exports.config = {
  framework: 'jasmine',
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['test/*.e2e.js'],
  multiCapabilities: [
     //{ browserName: 'firefox' },
     { browserName: 'chrome' }
   ]
};
```

- Se lanzan las pruebas (con Selenium Server en ejecución):
  - protractor e2e.conf.js

Elementos Globales

- browser- Envoltura alrededor de una instancia de WebDriver, utilizado para la navegación y la información de toda la página.
  - El método browser.get carga una página.
  - Protractor espera que Angular esté presente la página, por lo que generará un error si la página que está intentando cargar no contiene la biblioteca Angular.
- element- Función de ayuda para encontrar e interactuar con los elementos DOM de la página que se está probando.
  - La función element busca un elemento en la página.
  - Se requiere un parámetro: una estrategia de localización del elemento dentro de la página.
- by - Colección de estrategias para localizar elementos.
  - Por ejemplo, los elementos pueden ser encontrados por el selector CSS, por el ID, por el atributo ng-model, …
- protractor- Espacio de nombres de Angular que envuelve el espacio de nombres WebDriver.
  - Contiene variables y clases estáticas, tales como protractor.Key que se enumera los códigos de teclas especiales del teclado.

Visión de conjunto

- Protractor exporta la función global element, que con un localizador devolverá un ElementFinder .
- Esta función recupera un solo elemento, si se necesita recuperar varios elementos, la función element.all obtiene la colección de elementos localizados.
- El ElementFinder tiene un conjunto de métodos de acción, tales como click(), getText(), y sendKeys().
- Esta es la forma principal para interactuar con un elemento (etiqueta) de la página y obtener información de respuesta de el.
- Cuando se buscan elementos en Protractor todas las acciones son asíncronas:
  - Por debajo, todas las acciones se envían al navegador mediante el protocolo SON Webdriver Wire Protocol.
  - El navegador realiza la acción tal y como un usuario lo haría de forma nativa o manual.

Localizadores

- Un localizador de Protractor dice cómo encontrar un cierto elemento DOM.
- Los localizadores más comunes son:
  - by.css('.myclass')
  - by.id('myid')
  - by.model('name')
  - by.binding('bindingname')
- Los localizadores se pasan a la función element:
  - var tag = element(by.css('some-css'));
  - var arr = element.all(by.css('some-css'));
- Aunque existe una notación abreviada para CSS similar a jQuery:
  - var tag = $('some-css');
- Para encontrar subelementos o listas de subelementos:
  - var uno = element(by.css('some-css')).element(by.tagName('tag-within-css'));
  - var varios = element(by.css('some-css')).all(by.tagName('tag-within-css'));

ElementFinder

- La función element() devuelve un objeto ElementFinder.
- El ElementFinder sabe cómo localizar el elemento DOM utilizando el localizador que se pasa como un parámetro, pero en realidad no lo ha hecho todavía.
- No se pondrá en contacto con el navegador hasta que se llame a un método de acción.
- ElementFinder permite invocar acciones como si se produjesen directamente en el navegador.
- Dado que todas las acciones son asíncronas, todos los métodos de acción devuelven una promesa.
- Las acciones sucesivas se encolan y se mandan la navegador ordenadamente.
- Para acciones que deban esperar se usan las promesas:

```
element(by.model('nombre')).getText().then(function(text) {
  expect(text).toBe("MUNDO");
 });
```


La prueba

```
describe('Primera prueba con Protractor', function() {
  it('introducir nombre y saludar', function() {
    browser.get('http://localhost:4200/');
    var txt = element(by.model('vm.nombre'));
    txt.clear();
    txt.sendKeys('Mundo');
    browser.sleep(5000);
    element(by.id('btnSaluda')).click();
    expect(element(by.binding('vm.msg')).getText()).
       toEqual('Hola Mundo');
    browser.sleep(5000);
  });
});
```

Selenium

- El Selenium es un conjunto de herramientas para automatizar los navegadores web, robot que simula la interacción del usuario con el navegador, originalmente pensado como entorno de pruebas de software para aplicaciones basadas en la web.
- Como principales herramientas Selenium cuenta con:
  - Selenium IDE: una herramienta para grabar y reproducir secuencias de acciones con el navegador que permite crear pruebas sin usar un lenguaje de scripting para pruebas.
  - Selenium Core: API para escribir pruebas automatizadas y de regresión en un amplio número de lenguajes de programación populares incluyendo Java, C#, Ruby, Groovy, Perl, Php y Python.
  - WebDriver: interfaces que permite ejecutar las pruebas de forma nativa usando la mayoría de los navegadores web modernos en diferentes sistemas operativos como Windows, Linux y OSX.
  - Selenium Grid: Permite ejecutar muchas pruebas de un mismo grupo en paralelo o pruebas en múltiples entornos. Tiene la ventaja que un conjunto de pruebas muy grande puede dividirse en varias maquinas remotas para una ejecución más rápida o si se necesitan repetir las mismas pruebas en múltiples entornos.

Selenium IDE

- Es el entorno de desarrollo integrado para pruebas con Selenium que permite grabar, editar y depurar fácilmente las pruebas.
- Solo está disponible como una extensión de Firefox y Chrome.
- Se pueden desarrollar automáticamente scripts al crear una grabación y de esa manera se puede editar manualmente con sentencias y comandos para que la reproducción de nuestra grabación sea correcta
- Los scripts se generan en un lenguaje de scripting especial para Selenium a menudo denominado Selanese.
- Selanese provee comandos que dicen al Selenium que hacer y pueden ser:
  - **Acciones**: son comandos que generalmente manipulan el estado de la aplicación, ejecutan acciones sobre objetos del navegador, como hacer click en un enlace, escribir en cajas de texto o seleccionar de una lista de opciones. Muchas acciones pueden ser llamadas con el sufijo "AndWait" que indica la acción hará que el navegador realice una llamada al servidor y que se debe esperar a una nueva página se cargue.
  - **Descriptores de acceso**: examinan el estado de la página y almacenan los resultados en variables.
  - **Aserciones**: son como descriptores de acceso, pero las muestras confirman que el estado de la solicitud se ajusta a lo que se esperaba, verifican la presencia de un texto en particular o la existencia de elementos.

Selenium IDE

- Dispone de una selección inteligente de campos usando ID, nombre, Xpath o DOM según se necesite.
- Para la depuración permite la configuración de los puntos de interrupción, iniciar y detener la ejecución de un caso de prueba desde cualquier punto dentro del caso de prueba e inspeccionar la forma en el caso de prueba se comporta en ese punto.
- ~~Permite exportar los casos de prueba a Java, C# y Ruby, actuando como embriones en la creación de los casos de prueba para WebDriver.~~
- Selenium IDE dispone de un amplio conjunto de extensiones adicionales que ayudan o simplifican la elaboración de los casos de pruebas.

Ejecutar en línea de comandos

- Requiere tener instalado NodeJS (https://nodejs.org)
- Instalar CLI
  - npm install -g selenium-side-runner
- Instalar los WebDriver:
  - Crear una carpeta y referenciarla en el PATH del sistema.
  - Descargar los drivers (https://www.seleniumhq.org/download/)
  - Copiarlos a la carpeta creada.
- Para ejecutar las suites de pruebas:
  - selenium-side-runner project.side project2.side *.side
- Para ejecutar en diferentes navegadores:
  - selenium-side-runner *.side -c "browserName=Chrome"
  - selenium-side-runner *.side -c "browserName=firefox"

WebDriver

```
@BeforeClass
public static void setUpClass() throws Exception {
        System.setProperty("webdriver.chrome.driver", "C:/Archivos/.../chromedriver.exe");
}
@Before
public void setUp() throws Exception {
  driver = new ChromeDriver();
  baseUrl = "http://localhost/";
  driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}
@Test
public void testLoginOK() throws Exception {
  driver.get(baseUrl + "/login.php");
  driver.findElement(By.id("login")).sendKeys("admin");
  driver.findElement(By.id("password")).sendKeys("admin");
  driver.findElement(By.cssSelector("input[type=\"submit\"]")).click();
  try {
    assertEquals("", driver.findElement(By.cssSelector("img[title=\"Main Menu\"]")).getText());
  } catch (Error e) {
    verificationErrors.append(e.toString());
  }
}
```

Maven

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>3.13.0</version>
</dependency>
```


WebDriver

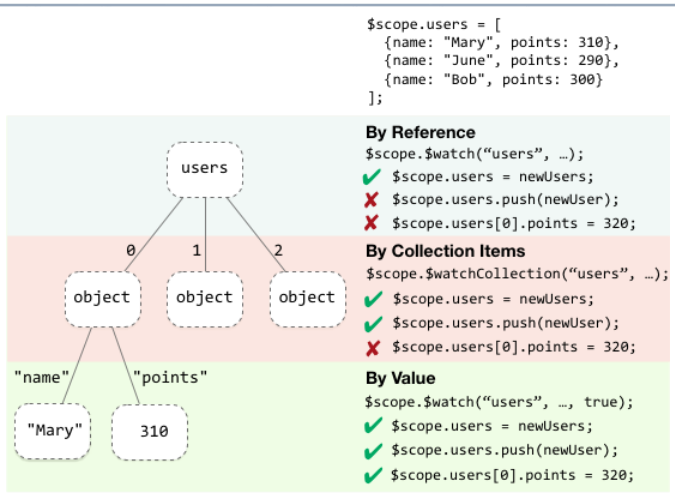
```
var selenium = require('selenium-webdriver');
describe('Selenium Demo', function() {
   var driver;
   beforeEach(function(done) {
      driver = new selenium.Builder().
         withCapabilities(selenium.Capabilities.chrome()).build();
      driver.get('http://google.es/').then(done);
   });
   afterEach(function(done) {
      driver.quit().then(done);
   });
   it('Buscar test', function(done) {
      driver.findElement(selenium.By.id('lst-ib')).sendKeys('test');
      driver.findElement(selenium.By.name('btnK')).click();
      var element = driver.findElement(selenium.By.tagName('body'));
      element.getAttribute('id').then(function(id) {
         expect(id).toBe('gsr');
         done();
      });
   });
})
```

ALGUNOS CONCEPTOS AVANZADOS

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}

Ciclo de Vida del Scope

- El flujo normal de un navegador recibiendo un evento es que se realiza una devolución de llamada correspondiente de JavaScript. Una vez que la devolución de llamada se completa los navegadores re-escriben el DOM y retornan a la espera de más eventos.
- Cuando el explorador llama desde el JavaScript se ejecuta el código fuera del contexto de ejecución de Angular, lo que significa que Angular no es consciente de las modificaciones del modelo.
- Para procesar adecuadamente las modificaciones del modelo a ejecución tiene que entrar en el contexto de ejecución Angular utilizando el método $apply.
- Sólo las modificaciones del modelo que se ejecuten dentro del método $apply serán tenido en cuenta adecuadamente por Angular.
- Por ejemplo si una directiva escucha los eventos del DOM, como ng-clic se debe evaluar la expresión dentro del método $apply.

Ciclo de Vida del Scope

- Después de evaluar la expresión, el método $apply realizará un $digest.
- En la fase $digest ,el scope examina todas las expresiones $watch y las compara con el valor anterior.
- Este chequeo “sucio” se realiza de forma asincrónica.
- Esto significa que una asignación como $scope.username="angular" no causará de forma inmediata un $watch para notificarlo, en su lugar la notificación $watch se retrasará hasta la fase $digest.
- Este retraso es conveniente, puesto que fusiona multiples actualizaciones del modelo en una sola notificación $watch, así como se garantiza que durante la notificación de $watch no hay otros $watches que se estén ejecutando.
- Si un $watch cambia un valor del modelo, forzará un ciclo $digest adicional.

Ciclo de Vida del Scope

- **Creación**: El scope raíz es creado durante el arranque de la aplicación por el $injector. Durante la vinculación de la plantilla, algunas directivas crean nuevos scopes secundarios.
- **Registro de vigilantes**: Durante la vinculación de las directivas de plantilla se registran los Watches en el ámbito de aplicación. Estos Watches serán utilizados para propagar los valores del modelo para el DOM.
- **Mutación del Modelo**: Para que las mutaciones se puedan observar de forma correcta, se deben hacer dentro de scope.$apply(). Las APIs de Angular hacen esto de forma implícita, así que no se necesita ninguna llamada adicional a $apply al trabajar de forma sincrónica en los controladores, o trabajando de forma asincrónica con $http, $timeout o servicios $interval.
- **Observación de la mutación**: Al final del $apply, Angular realiza un ciclo $digest sobre el ámbito de la raíz, que luego se propaga lo largo de todos los ámbitos secundarios. Durante el ciclo de $digest, todas las expresiones o funciones $watch se comprueban para la mutación de modelo y si se detecta una mutación, el oyente $watch es llamado.
- **Destrucción del Scope**: Cuando el scope secundario ya no es necesario, es responsabilidad del creador del scope secundario destruirlo mediante la API scope.$destroy(). Esto detendrá la propagación de las llamadas a $digest del ámbito secundario y permitirá que la memoria utilizada por los modelos del ámbito secundario sean reclamados por el recolector de basura (garbage collector).

$templateCache

- La primera vez que se utiliza una plantilla, se carga en la memoria caché de plantillas 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');
```

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>
  - la declaración de un clase CSS (‘C’) en el atributo class de un elemento del DOM

  - 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 obligatorios 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'.

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.

```
link: function(scope, element, attrs, controller, transcludeFn) {
   var color = element.css( "background-color" );
   element.on( "mouseover", function() {
      element.css('background-color', 'yellow');
   });
   element.on( "mouseleave", function() {
      element.css('background-color', color);
   });
}
```


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>");
}
```

require

- Marca que se requiere otra directiva para inyectar su controlador como el cuarto argumento de la función de enlace (link).
- El nombre puede ser prefijado con:
  - (sin prefijo): Busca el controlador requerido en el elemento actual y lanzar un error si no lo encuentra (obligatorio).
  - ?- Busca el controlador requerido en el elemento actual e inyecta null si no lo encuentra (opcional).
  - ^- Busca el controlador requerido en el elemento actual y sus contenedores (obligatorio).
  - ?^- Busca el controlador requerido en el elemento actual y sus contenedores (opcional).
  - ^^- Busca el controlador requerido en los contenedores del elemento actual (obligatorio).
  - ?^^- Busca el controlador requerido en los contenedores del elemento actual (opcional).

require

```
angular.module('MyCore').directive('myTabs', function() {
  return {
    restrict: 'E',
    transclude: true,
    scope: {},
    controller: ['$scope', function MyTabsController($scope) {
      var panes = $scope.panes = [];
      $scope.select = function(pane) {
        angular.forEach(panes, function(item) {
          item.selected = item == pane;
        });
      };
      this.addPane = function(pane) {
        panes.push(pane);
        if (panes.length === 1)
          $scope.select(pane);
      };
    }],
    template: '<table><tr><td ng-repeat="pane in panes" ng-click="select(pane)" ' +
        'ng-class="{\'tab-active\':pane.selected,\'tab-inactive\':!pane.selected}">{{pane.title}}</td></tr></table>' +
        '<div class="tab-content" ng-transclude></div>',
  };
});
```

require

```
angular.module('MyCore').directive('myPane', function() {
  return {
    require: '^^myTabs',
    restrict: 'E',
    transclude: true,
    scope: { title: '@' },
    link: function(scope, element, attrs, tabsCtrl) {
      tabsCtrl.addPane(scope);
    },
    template: '<div class="tab-pane" ng-show="selected" ng-transclude></div>'
  };
});

<my-tabs>
    <my-pane title="Hello">
        <p>Lorem ipsum dolor sit amet</p>
    </my-pane>
    <my-pane title="World">
        <em>Mauris elementum elementum enim at suscipit.</em>
    </my-pane>
</my-tabs>
```


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


ANIMACIONES

ngAnimate

- Una animación es cuando la transformación de un elemento HTML da la ilusión de movimiento.
- El módulo ngAnimate no realiza la animación a los elementos HTML, pero vigila los eventos de ciertas directivas, como ocultar o mostrar de un elemento HTML, y le asigna al elemento clases CSS predefinidas que se pueden utilizar para hacer animaciones.
- 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-animate.js"></script>
angular.module("app", ["ngAnimate"])
```


Directivas “conscientes de animación”

- ngRepeat: entrar, salir y moverse
- ngView: entrar y salir
- ngInclude: entrar y salir
- ngSwitch : entrar y salir
- ngIf : entrar y salir
- ngClass : agregar y quitar (la clase(s) CSS presente)
- ngShow y ngHide : agregar y quitar (el valor de la clase ng-hide)
- form y ngModel : agregar y quitar (dirty, pristine, valid, invalid y todas las demás validaciones)
- ngMessages : agregar y quitar (ng-active & ng-inactive)
- ngMessage : entrar y salir

Animaciones basadas en CSS

- Basadas en clase CSS, ngAnimate no requieren código JavaScript.
- Mediante el uso de una clase CSS que a la que hacemos referencia en nuestro código HTML y CSS podemos crear una animación que será asignada por ngAnimate cuando la directiva subyacente lleve a cabo una operación.
- Basta con crear reglas CSS que añadan a la clase de estilo como sufijos las diferentes operaciones:
  - ng-hide: cuando un elemento del DOM pasa de visible a invisible,
  - ng-show: cuando un elemento del DOM pasa de invisible a visible,
  - ng-enter: cuando un elemento nuevo se añade al DOM,
  - ng-leave: cuando un elemento es eliminado o deja de estar en el DOM
  - ng-move: cuando un elemento cambia de posición en el DOM.
- El cambio siempre tiene dos efectos: anterior y actual.
  - .ng-enter y .ng-enter.ng-enter-active
  - .ng-leave y .ng-leave.ng-leave-active

Animaciones basadas en CSS

```
.demo {
  transition: all linear 0.5s;
  background-color: lightblue;
  height: 100px;
}

.demo.ng-hide {
  height: 0;
}

<input type="checkbox" ng-model="myCheck">
<div class="demo" ng-hide="myCheck"></div>
```

NAVEGACIÓN

ng-include

- La directiva ng-include se usa para cargar trozos de HTML en la página.
- Permite implementar el mecanismo mas simple para la navegación entre las diferentes partes de las aplicación de una sola pagina.
- Se puede usar directamente como un tag HTML en vez de como un atributo HTML.
- Es como los include de la parte de servidor pero ahora desde JavaScript.
- La carga se hace bajo demanda mediante una petición GET de AJAX al servidor.
- Como el src puede recibir variables, las cadenas constantes deben ir entre comillas:

```
<div ng-include="tmpl"></div>
<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;
        }
      };
});
```


Enrutado

- 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>
```

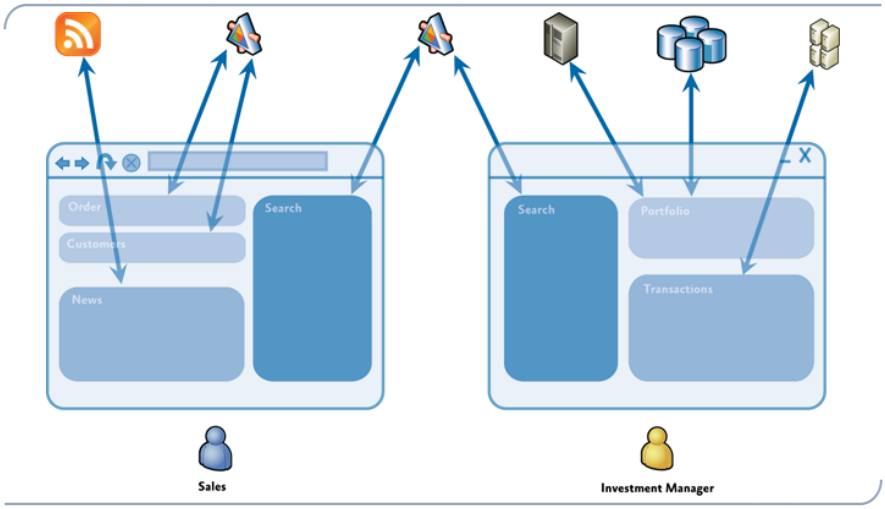
Mantenimiento de estado

- 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.

AngularUI Router

- Angular permite el manejo de rutas mediante su módulo ngRoute pero en aplicaciones medianas o grandes puede ser bastante limitado.
- Para este problema surgió ui-router que permite manejar las rutas alrededor de states.
- AngularUI Router es un framework de enrutamiento para AngularJS, que permite organizar las partes de su interfaz en una máquina de estados.
- A diferencia del servicio de ruta $route del núcleo, que se organiza en torno a rutas URL, IU-Router está organizado en torno a los estados, que pueden tener opcionalmente rutas, así como otro tipo de comportamiento.
- Los Estados están ligados a vistas con nombre, anidadas y en paralelo, lo que permite administrar poderosamente la interfaz de su aplicación.
- El patrón Composite View es uno de los clásicos a la hora de desarrollar la capa de presentación y define que la vista puede estar compuesta por varias subvistas que se actualizan de forma independiente.

Patrón Composite View



ui.router

- El módulo Angular ui-router es el substituto natural de ng-route cuando las aplicaciones necesitan una gestión de la vista compleja.
- Además incorpora muchos otras funcionalidades como vistas anidadas, abstractas, múltiples vistas, etc.
- 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.

  https://angular-ui.github.io/ui-router/

```
<script src="angular-ui-router.min.js"></script>
angular.module("app", ["ui.router"])
```

Configurar el sistema de estados

- El sistema de enrutado de AngularJS nos permite configurar los estados que queremos crear en nuestra aplicación de una manera declarativa.
- El servicio $stateProvider contiene los métodos necesarios para la configuración:
  - state(), permite asociar un nombre de estado con una ruta, una vista y un controlador.
  - otherwise(), para estados no reconocidos.
- Esta configuración se debe realizar dentro del método config() del modulo. De hecho, solo podemos inyectar $routeProvider en el método config() de configuración.

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


Estados

```
.config(function($stateProvider){
$stateProvider
   .state('state1', {
    url: "/state1",
    templateUrl: "partials/state1.html"
   })
   .state('state1.list', {
    url: "/list",
    templateUrl: "partials/state1.list.html",
    controller: function($scope) {
     $scope.items = ["A", "List", "Of", "Items"];
    }
   })
```

Directivas

- Para que ui.router pueda mostrar el contenido de los estados necesitamos agregar la directiva ui-view a nuestro documento html.
  - <ui-view></ui-view>
  - <div ui-view>
- Hay tres formas principales para activar un estado:
  - Invocando $state.go ().

    $state.go('contact.detail') //will go to the 'contact.detail' state

    $state.go('^') // will go to a parent state.
  - Haciendo clic en un enlace que contiene la directiva ui-sref.

    <a ui-sref="contacts.detail({ id: contact.id })">{{ contact.name }}

    <a ui-sref="estado1">Estado1
  - Navegando a la URL asociada con el estado.

Sub vistas

- Hasta ahora es muy similar a ng-router, pero ui-router permite una mayor flexibilidad, podemos hacer que la vista este compuesta por varias subvistas diferentes. Cada una de ellas se actualiza de forma independiente.

```
$stateProvider.state('estado1', {
        url: "/estado1",
        views:{
          "subvista1":{ templateUrl: "plantilla1.html"},
          "subvista2": { templateUrl: "plantilla2.html"}
        }
      }).state('estado2', {
        url: "/estado2",
        views:{
          "subvista1":{ templateUrl: "plantilla3.html"},
          "subvista2": { templateUrl: "plantilla4.html"}
        }
      })
    });
```

```
<div ui-view="subvista1"></div><div ui-view="subvista2"></div>

<a ui-sref="estado1">Estado1</a><a ui-sref="estado2">Estado2</a>
```

Patrones y recursos

SERVIDOR

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.

Objeto Defered

- El objeto Defered gestiona la creación del objeto Promise y los cambios de estados del mismo.

```
this.list = function() {
  var defered=$q.defer();
  var promise=defered.promise;

  $http({
    method: 'GET',
    url: baseUrl + '/datos'
  }).success(function(data, status, headers, config) {
    defered.resolve(data);
  }).error(function(data, status, headers, config) {
    defered.reject(status);
  });

  return promise;
}
```

Invocar promesas

- El objeto Promise expone los métodos:
  - then(fnResuelta, 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)
```


Validación personalizada (servidor)

- Es común que haya validaciones que no podamos realizarlas en lado del cliente y tengamos que recurrir a información almacenada en el servidor.
- Al requerir una petición al servidor, la validación debe ser asincrona, no se puede registrar en el objeto $validators del controlador de la directiva ng-model como el resto de validaciones, en este caso hay que registrarla en el objeto $asyncValidators.
- Este objeto se encarga de las validaciones asíncronas, así como las llamadas http.
- Se utiliza el mecanismo de promesas: la promesa se resuelve cuando alias está registrado, y se rechaza cuando no lo está (registrando un error en el controlador del modelo).

Validación personalizada (servidor)

```
app.directive('username', function($q, $timeout) {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      ctrl.$asyncValidators.username = function(modelValue, viewValue) {
        // tratamos los modelos vacíos como correctos
        if (ctrl.$isEmpty(modelValue)) { return $q.when(); }
        var def = $q.defer();
        $http.get('validate/' + modelValue).
              then(function(response) {
                    if(response.data.isValid)
                          def.resolve();
                    } else {
                          def.reject();
                    }
              }, function(response) {
              def.reject();
              });
        return def.promise;
};}}});
```


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);
  });
…
```

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);
```

REST (REpresentational State Transfer)

- Un **estilo de arquitectura** para desarrollar aplicaciones web distribuidas que se basa en el uso del protocolo HTTP e Hypermedia.
- Definido en el 2000 por Roy Fielding, para no reinventar la rueda, se basa en aprovechar lo que ya estaba definido en el HTTP pero que no se utilizaba.
- El HTTP ya define 8 métodos (algunas veces referidos como "verbos") que indica la acción que desea que se efectúe sobre el recurso identificado:
  - HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT
- El HTTP permite en el encabezado transmitir la información de comportamiento:
  - Accept, Content-type, Response (códigos de estado), Authorization, Cache-control, …

Objetivos de los servicios REST

- Desacoplar el cliente del backend
- Mayor escalabilidad
  - Sin estado en el backend.
- Separación de problemas
- División de responsabilidades
- API uniforme para todos los clientes
  - Disponer de una interfaz uniforme (basada en URIs)

Uso de la cabecera

- **Request**: Método /uri?parámetros
  - GET: Recupera el recurso
    - Todos: Sin parámetros
    - Uno: Con parámetros
  - POST: Crea un nuevo recurso
  - PUT: Edita el recurso
  - DELETE: Elimina el recurso
- **Accept**: Indica al servidor el formato o posibles formatos esperados, utilizando MIME.
- **Content-type**: Indica en que formato está codificado el cuerpo, utilizando MIME
- **Response**: Código de estado con el que el servidor informa del resultado de la petición.

Peticiones

Request: GET /users
    Response: 200
    content-type:application/json
Request: GET /users/11
    Response: 200
    content-type:application/json
Request: POST /users
    Response: 201 Created
    content-type:application/json
    body
Request: PUT /users/11
    Response: 200
    content-type:application/json
    body
Request: DELETE /users/11
    Response: 204 No Content

Fake Online REST API
for Testing and Prototyping
https://jsonplaceholder.typicode.com/

Richardson Maturity Model

http://www.crummy.com/writing/speaking/2008-QCon/act3.html

- # Nivel 1 (Pobre): Se usan URIs para identificar recursos:
  - Se debe identificar un recurso
    /invoices/?page=2 → /invoices/page/2
  - Se construyen con nombres nunca con verbos
    /getUser/{id} → /users/{id}/
    /users/{id}/edit/login → users/{id}/access-token
  - Deberían tener una estructura jerárquica
    /invoices/user/{id} → /user/{id}/invoices
- # Nivel 2 (Medio): Se usa el protocolo HTTP adecuadamente
- # Nivel 3 (Óptimo): Se implementa hypermedia.

Hypermedia

- Se basa en la idea de enlazar recursos: propiedades que son enlaces a otros recursos.
- Para que sea útil, el cliente debe saber que en la respuesta hay contenido hypermedia.
- En content-type es clave para esto
  - Un tipo genérico no aporta nada:
    Content-Type: text/xml
  - Se pueden crear tipos propios
    Content-Type:application/servicio+xml

JSON Hypertext Application Language

- RFC4627 http://tools.ietf.org/html/draft-kelly-json-hal-00
- Content-Type: application/hal+json

```
{
   "_links": {
     "self": {"href": "/orders/523" },
     "warehouse": {"href": "/warehouse/56" },
     "invoice": {"href": "/invoices/873“}
   },
   , "currency": "USD"
   , "status": "shipped"
   , "total": 10.20
}
```


Patrón Agregado (Aggregate)

- Una Agregación es un grupo de objetos asociados que deben tratarse como una unidad a la hora de manipular sus datos.
- El patrón Agregado es ampliamente utilizado en los modelos de datos basados en Diseños Orientados al Dominio (DDD).
- Proporciona un forma de encapsular nuestras entidades y los accesos y relaciones que se establecen entre las mismas de manera que se simplifique la complejidad del sistema en la medida de lo posible.
- Cada Agregación cuenta con una Entidad Raíz (root) y una Frontera (boundary):
  - La Entidad Raíz es una Entidad contenida en la Agregación de la que colgarán el resto de entidades del agregado y será el único punto de entrada a la Agregación.
  - La Frontera define qué está dentro de la Agregación y qué no.
- La Agregación es la unidad de persistencia, se recupera toda y se almacena toda.

Servicio RESTFul

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


Seguridad

- La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución.
- Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado.
- Las aplicaciones JavaScript no pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript.
- Los navegadores emplean un método estricto para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.
- Si el código JavaScript se descarga desde la siguiente URL: http://www.ejemplo.com
- Las funciones y métodos incluidos en ese código no pueden acceder a:
  - **https**://www.ejemplo.com/scripts/codigo2.js
  - http://www.ejemplo.com**:8080**/scripts/codigo2.js
  - http://**scripts**.ejemplo.com/codigo2.js
  - http://**192.168.0.1**/scripts/codigo2.js
- La propiedad document.domain se emplea para permitir el acceso entre subdominios del dominio principal de la aplicación.

CORS

- Un recurso hace una solicitud HTTP de origen cruzado cuando solicita otro recurso de un dominio distinto al que pertenece y, por razones de seguridad, los exploradores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script.
- XMLHttpRequest sigue la política de mismo-origen, por lo que solo puede hacer solicitudes HTTP a su propio dominio. Para mejorar las aplicaciones web, los desarrolladores pidieron a los proveedores de navegadores que permitieran a XMLHttpRequest realizar solicitudes de dominio cruzado.
- El Grupo de Trabajo de Aplicaciones Web del W3C recomienda el nuevo mecanismo de Intercambio de Recursos de Origen Cruzado (CORS, Cross-origin resource sharing). Los servidores deben indicar al navegador mediante cabeceras si aceptan peticiones cruzadas y con que características:
  - "Access-Control-Allow-Origin", "*"
  - "Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept"
  - "Access-Control-Allow-Methods", "GET, POST, PUT, DELETE“
- Soporte: Chrome 3+ Firefox 3.5+ Opera 12+ Safari 4+ Internet Explorer 8+

Para habilitar CORS en servidores específicos consultar http://enable-cors.org

Proxy a un servidor de back-end

- Se puede usar el soporte de proxy en el servidor de desarrollo del webpack para desviar ciertas URL a un servidor backend.
- Crea un archivo proxy.conf.json junto a angular.json.

```
{
  "/api": {
    "target": "http://localhost:4321",
    "pathRewrite": { "^/api": "/ws" },
    "secure": false,
    "logLevel": "debug"
  }
}
```

- Cambiar la configuración del Angular CLI en angular.json:

```
"architect": {
  "serve": {
    "builder": "@angular-devkit/build-angular:dev-server",
    "options": {
      ...
      "proxyConfig": "proxy.conf.json"
    },
```

Desactivar la seguridad de Chrome

- Pasos para Windows:
  - Localizar el acceso directo al navegador (icono) y crear una copia como “Chrome Desarrollo”.
  - Botón derecho -> Propiedades -> Destino
  - Editar el destino añadiendo el parámetro al final. ej: "C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" --disable-web-security
  - Aceptar el cambio y lanzar Chrome
- Para desactivar parcialmente la seguridad:

  --allow-file-access

  --allow-file-access-from-files

  --allow-cross-origin-auth-prompt
- Referencia a otros parametros:
  - http://peter.sh/experiments/chromium-command-line-switches/

JWT: JSON Web Tokens

https://jwt.io

- JSON Web Token (JWT) es un estándar abierto (RFC-7519) basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios y garantizar que sean válidos y seguros.
- El caso más común de uso de los JWT es para manejar la autenticación en aplicaciones móviles o web. Para esto cuando el usuario se quiere autenticar manda sus datos de inicio del sesión al servidor, este genera el JWT y se lo manda a la aplicación cliente, posteriormente en cada petición el cliente envía este token que el servidor usa para verificar que el usuario este correctamente autenticado y saber quien es.
- Se puede usar con plataformas IDaaS (Identity-as-a-Service) como Auth0 que eliminan la complejidad de la autenticación y su gestión.
- También es posible usarlo para transferir cualquier datos entre servicios de nuestra aplicación y asegurarnos de que sean siempre válido. Por ejemplo si tenemos un servicio de envío de email otro servicio podría enviar una petición con un JWT junto al contenido del mail o cualquier otro dato necesario y que estemos seguros que esos datos no fueron alterados de ninguna forma.

Interceptores

- Una característica importante de $http es la interceptación: la capacidad de declarar interceptores que se sitúan entre la aplicación y el backend.
- Cuando la aplicación hace una petición, los interceptores la transforman antes de enviarla al servidor.
- Los interceptores pueden transformar la respuesta en su camino de regreso antes de que la aplicación la vea.
- Esto es útil para múltiples escenarios, desde la autenticación hasta el registro.
- Cuando hay varios interceptores en una aplicación, Angular los aplica en el orden en que se registraron.

Interceptores

```
app.config(['$httpProvider', function ($httpProvider) {
   $httpProvider.interceptors.push(["$q", "AuthService", function ($q, auth) {
      return {
         'request': function (config) {
            var v = auth.isAutenticated();
            if (config.withCredentials && auth.isAutenticated())
               config.headers['Authorization'] = auth.AuthorizationHeader();
            return config;
         },
         'requestError': function(rejection) { return $q.reject(rejection); },
         'response': function (response) { return response; },
         'responseError': function(rejection) { return $q.reject(rejection); },
      };
   }]);
}]);
```

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();
```